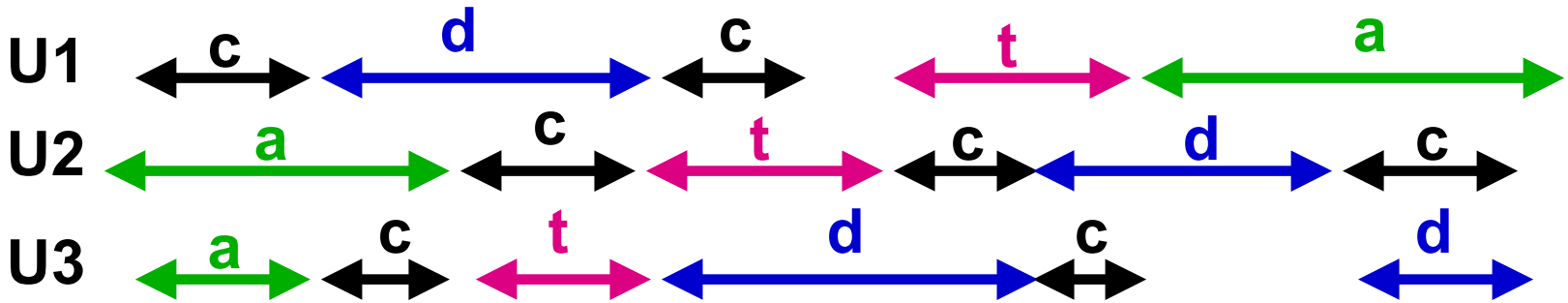
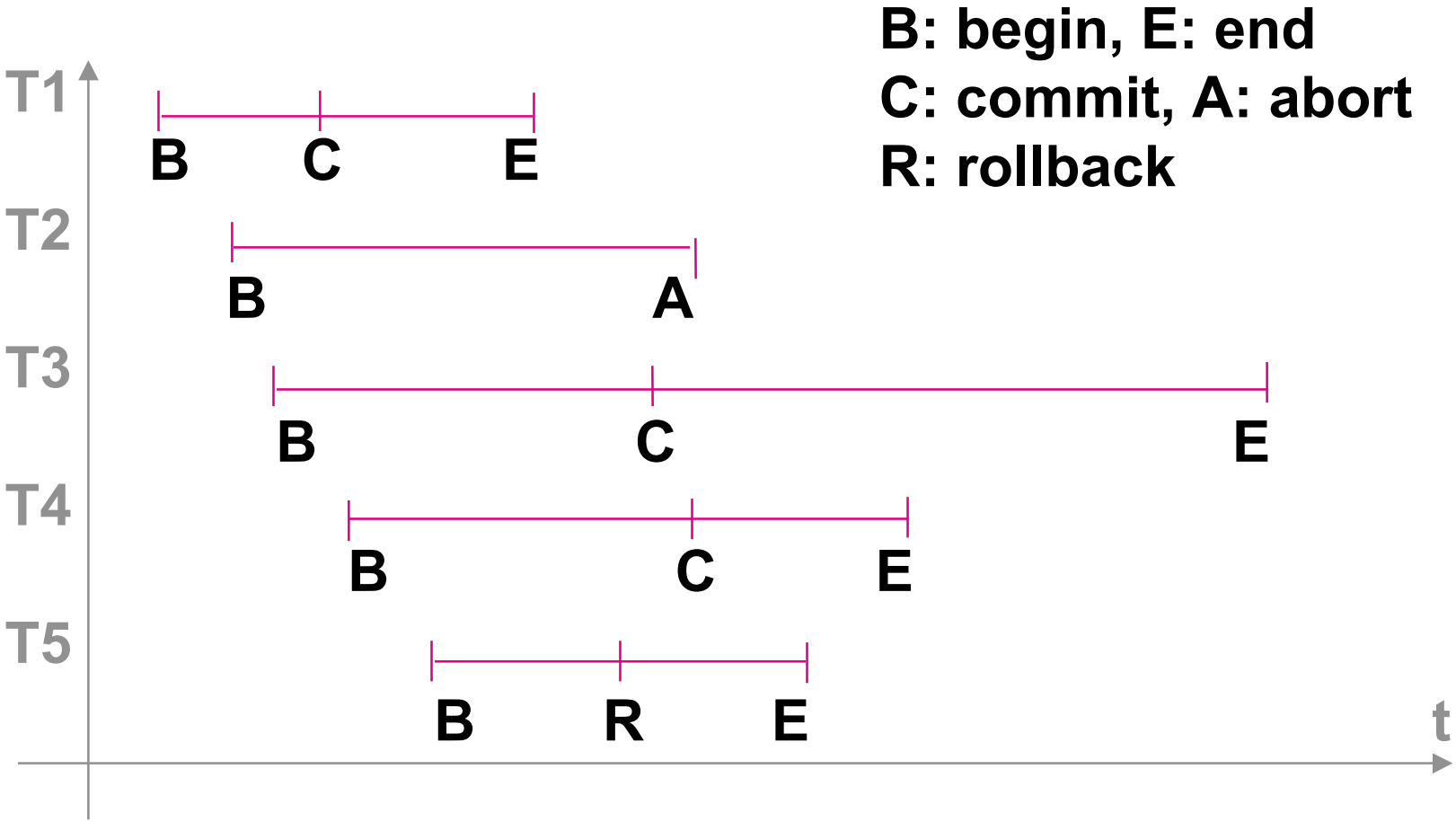
Controllo di concorrenza

I vantaggi della concorrenza

- **sul server (1 cpu, 1 disco, 1 sistema di trasmissione) è possibile il parallelismo tra:**
  - **elaborazione :cpu (c)**
  - **operazioni di I/O: disco (d)**
  - **operazioni di trasmissione (t)**
  - **attesa dai client (a)**



I vantaggi della concorrenza



Problemi dovuti alla Concorrenza (Lost Update & Dirty Read)

- T1:

```
BEGIN TRANSACTION;
UPDATE CC
SET S = S + 3
WHERE CCN=100;
COMMIT WORK
```

supp. inizialmente:
S(100)=20

- T2:

```
BEGIN TRANSACTION;
UPDATE CC
SET S = S + 6
WHERE CCN=100;
COMMIT WORK
```

dopo esecuzione seriale:
S(100)=29

Sequenza di azioni di I/O che produce l'errore

oppure

Lost Update (perdita modifiche)

T1

- R(x)=S(100) [=20]
- x=x+3
- W(x) [=23]
- commit

T2

- R(y)=S(100) [=20]
- y=y+6
- W(y) [=26]
- commit

alla fine S(100)=26 !!

Dirty Read (lettura sporca)

T1		T2	
• R(x)=S(100)	[=20]		
• x=x+3			
• W(x)	[=23]		
• **rollback**		• R(y)=S(100)	[=23]
	[=20]	• y=y+6	
		• W(y)	[=29]
		• commit	

alla fine S(100)=29 !!

Cascading Abort (effetto domino)

T1	T2
• R(x)=S(100) [=20]	
• x=x+3	
• W(x) [=23]	
...	• R(y)=S(100) [=23]
	• y=y+6
	• W(y) [=29]
	• **commit**
• **rollback** [=20]	
	anche T2 deve abortire!

Altro problema (Update Fantasma)

- T1:
  BEGIN TRANSACTION;
  UPDATE CC
  SET S = S - 10
  WHERE CCN=100;
  UPDATE CC
  SET S = S + 10
  WHERE CCN=200;
  COMMIT WORK

  supp. inizialmente
  S(100)=20, S(200)=30

- T2:
  BEGIN TRANSACTION;
  SELECT SUM(S)
  FROM CC
  WHERE
  CCN IN (100, 200);
  COMMIT WORK

  [la somma dei due saldi
  è in ogni caso 50]

Update Fantasma (Incons. Analysis)

T2		T1	
• sum=0		• R(x)=S(100)	[=20]
• R(s)=S(200)	[=30]	• x=x-10	
• sum=sum+s	[=30]	• W(x)	[=10]
• R(s)=S(100)	[=10]	• R(y)=S(200)	[=30]
• sum=sum+s	[=40]	• y=y+10	
• print(sum)		• W(y)	[=40]
• commit		• commit	
stampa 40 !!		S(100)=10, S(200)=40	

Controllo di concorrenza

Si vuole garantire una esecuzione concorrente equivalente alla esecuzione seriale:

serializzabilità

TECNOLOGIA PREVALENTE :

LOCKING (BLOCCAGGIO)

Per evitare il problema, T1 acquisisce un "lock" sul record CCN = 100 (oppure sulla pagina che contiene il record, oppure sulla relazione che contiene il record). T2 che cerca di acquisire il dato entra in "stato di wait" . Quando T1 rilascia il lock T2 può visitare il dato.

Primitive di locking

r-lock: lock in lettura (Shared, s-lock)

w-lock: lock in scrittura (eXclusive, x-lock)

unlock

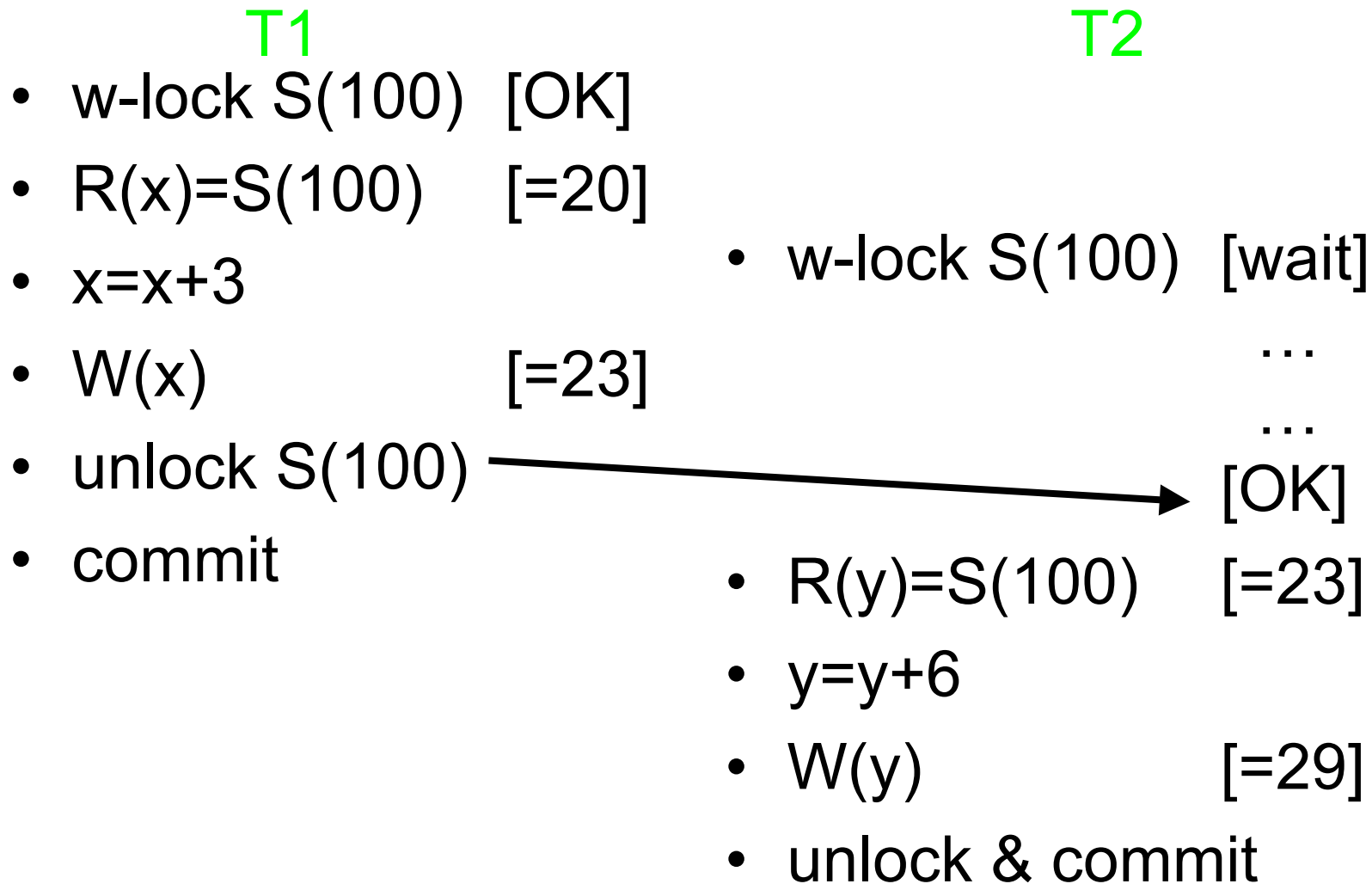
STATO DI UN OGGETTO:

libero

r-locked (bloccato da un lettore)

w-locked (bloccato da uno scrittore)

Locking (vs lost update)



Transazioni ben formate

- **ogni read di un oggetto**
  è preceduto da r-lock ed è seguito da unlock
- **ogni write di un oggetto**
  è preceduto da w-lock ed è seguito da unlock

Un r-lock entra in conflitto con un w-lock (e viceversa), mentre non entra in conflitto con un altro r-lock (cioè due transazioni di lettura non interferiscono)

Un w-lock entra in conflitto con un w-lock (e viceversa)

Tabella dei conflitti

	r-locked	w-locked
r-lock	OK	NO
w-lock	NO	NO

SI : blocco della risorsa,
il programma procede
NO : il programma
va in attesa che
la risorsa venga sbloccata

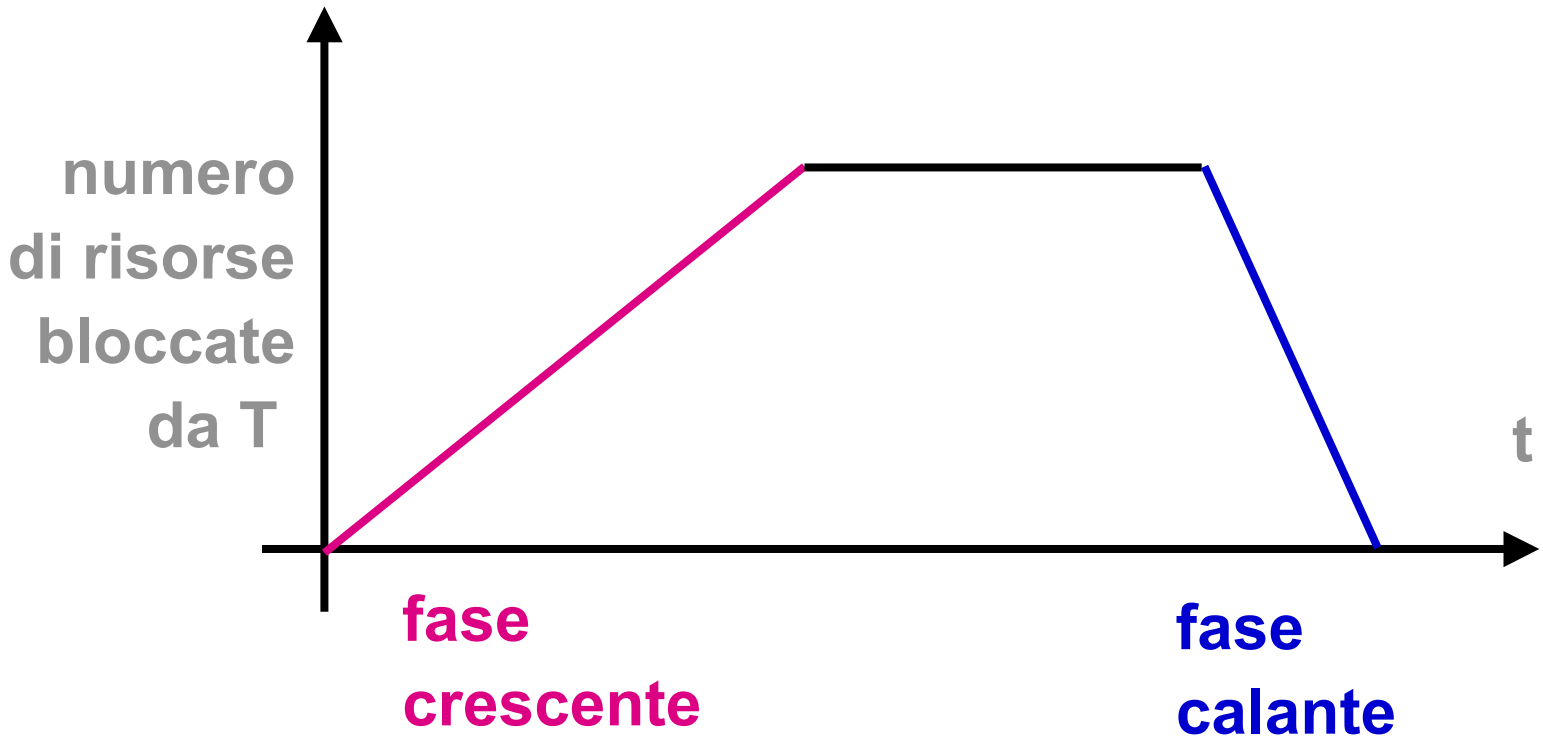
contatore dei lettori

r-lock
r-counter <- r-counter + 1

unlock
r-counter <- r-counter - 1

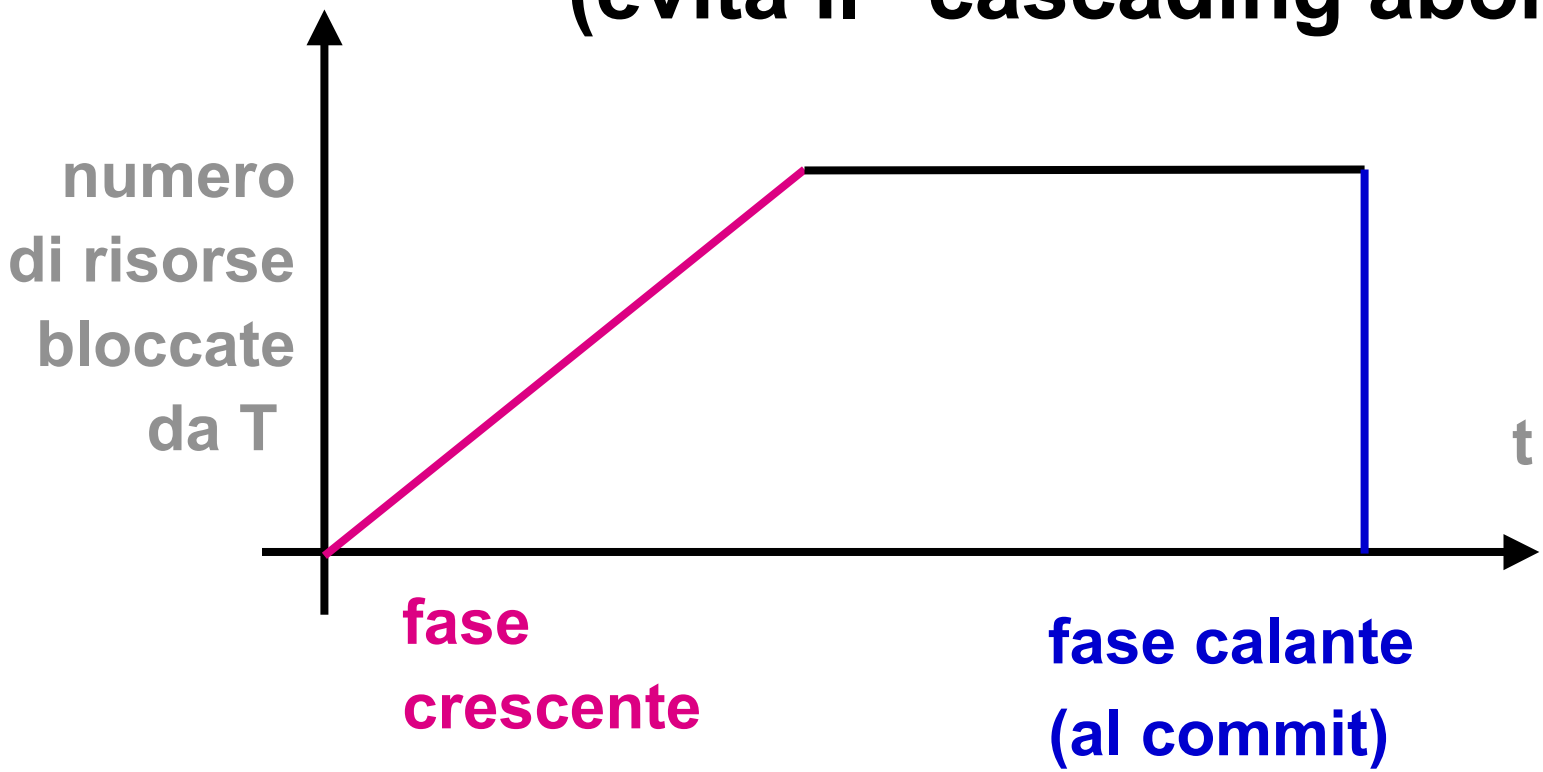
Locking a due fasi (2PL)

Per una transazione T una azione di unlock non può precedere una azione di lock



Locking a due fasi stretto

Come il 2PL, ma i blocchi sono rilasciati tutti alla fine della transazione (evita il "cascading abort")



Conseguenze

a transazioni ben formate

b politica dei conflitti come da tabella

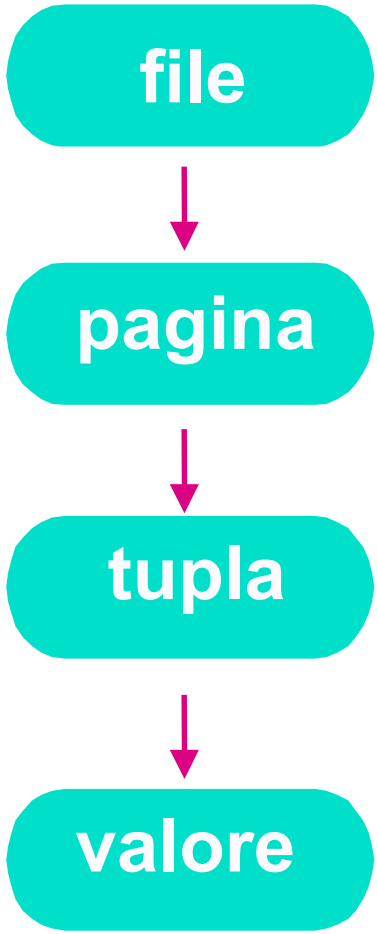
c locking a due fasi (stretto)

serializzabilità

sono possibili anche tecniche alternative (es. utilizzo dei *timestamp*)

Granularità del locking

lock
a livello di:
relazione
pagina
tupla
valore di un attributo
di una tupla



(più la granularità è ridotta, più è elevata la concorrenza)

Sistema di Locking

I *LOCKS* vengono acquisiti in caso di modifica dei dati, oppure quando si vogliono leggere dati "*consistenti*".

Ad esempio, supponiamo che T1 voglia leggere la relazione CC senza che avvengano modifiche durante la lettura e che T2 voglia modificare alcune n-ple di CC.

Quando un utente vuole operare in modo esclusivo su un oggetto, acquisisce un INTENTION LOCK; potrà ottenere un w-lock solo quando il precedente w-lock verrà rilasciato e verrà il suo turno tra coloro che hanno posto INTENTION LOCK

Gestione gerarchica dei Lock

- **È anche possibile una gestione gerarchica (rispetto alla granularità) - Nuovi tipi di Lock**
- **["tradizionali"]**
  - **XL: esclusivo (w-lock)**
  - **SL: condiviso (r-lock)**
- **[con intenzioni riferite ai nodi discendenti]**
  - **ISL: intenzione di SL**
  - **IXL: intenzione di XL**
  - **SIXL: SL e intenzione di XL**

Gestione gerarchica dei Lock

- **I lock si richiedono dalla radice in giù**
- **Si rilasciano dal nodo bloccato in su**
- **Per richiedere SL o ISL su un nodo occorre avere già ISL o IXL sul nodo padre**
- **Per richiedere IXL, XL o SIXL su un nodo occorre avere già SIXL o IXL sul nodo padre**
- **Compatibilità:**
  - **ISL con ISL, IXL, SL, SIXL (no XL)**
  - **IXL con ISL, IXL ed SL con SL**

Sistema di Locking

DEFINIZIONE di *UNCOMMITTED DATA* (dati non definitivi): dati soggetti a modifica, dati che sono stati modificati da una transazione ancora in funzione (e che potrebbe "ABORTIRE").

REGOLA PER LA MODIFICA:
Nessuna transazione può modificare dati uncommitted.

Si potrebbe verificare la perdita delle modifiche, se una transazione che stava modificando gli stessi dati abortisce.
L'algoritmo di ripristino dei dati non è in grado di tenerne traccia, e conseguentemente, ricostruire le modifiche effettuate da più transazioni.

Livelli di Isolamento

Quando una transazione vuole *modificare* i dati deve acquisire w-locks (exclusive).

Quando una transazione vuole solo *leggere* dati *consistenti* deve acquisire r-locks (shared).

Nei sistemi commerciali (SQL)
sono possibili 3 livelli di isolamento per la lettura:

LIVELLO 1 (READ UNCOMMITTED)
Una transazione T vuole leggere qualsiasi dato, anche uncommitted, quindi non chiede locks. Rileggendo due volte lo stesso dato, T può trovarlo cambiato perché:

1. **un'altra transazione T2 lo ha cambiato**
2. **un'altra transazione T3, che lo aveva cambiato in precedenza , abortisce.**

Livelli di Isolamento

LIVELLO 2 (READ COMMITTED)

Una transazione T chiede un lock sul record da leggere, quindi non legge mai dati uncommitted.
Rilascia il lock dopo la lettura e prima di concludere la propria operazione sul DB. Però in un secondo tempo può trovare il record cambiato per la causa 1), mentre non si può verificare la 2).

LIVELLO 3 (SERIALIZABLE)

Una transazione pone locks su tutti i records che legge e li rilascia solo dopo aver terminato.
Non si verificano mai nè 1), nè 2).

Problema del Deadlock (stallo)

Situazione che si verifica quando due o più transazioni sono in stato di wait (attesa) per attendere il rilascio di oggetti da parte di altre transazioni in stato di wait.

T1 :

```
BEGIN TRANSACTION;
UPDATE CC
SET S =
   ( SELECT S FROM CC
     WHERE CCN=10 )
WHERE CCN=20 ;
COMMIT WORK;
```

T2 :

```
BEGIN TRANSACTION;
UPDATE CC
SET S =
   ( SELECT S FROM CC
     WHERE CCN=20 )
WHERE CCN=10 ;
COMMIT WORK
```

Insorgenza del deadlock

T1 esegue w-lock(D1) - T2 deve usare D1
T2 esegue w-lock(D2) - T1 deve usare D2

T1 attende una risorsa controllata da T2
T2 attende una risorsa controllata da T1

Deadlock

T1

- r-lock S(10) [OK]
- R(x)=S(10)

  …

  ...
- w-lock S(20) [wait]

  …

  …

  …

(attesa indefinita)

T2

- r-lock S(20) [OK]
- R(y)=S(20)

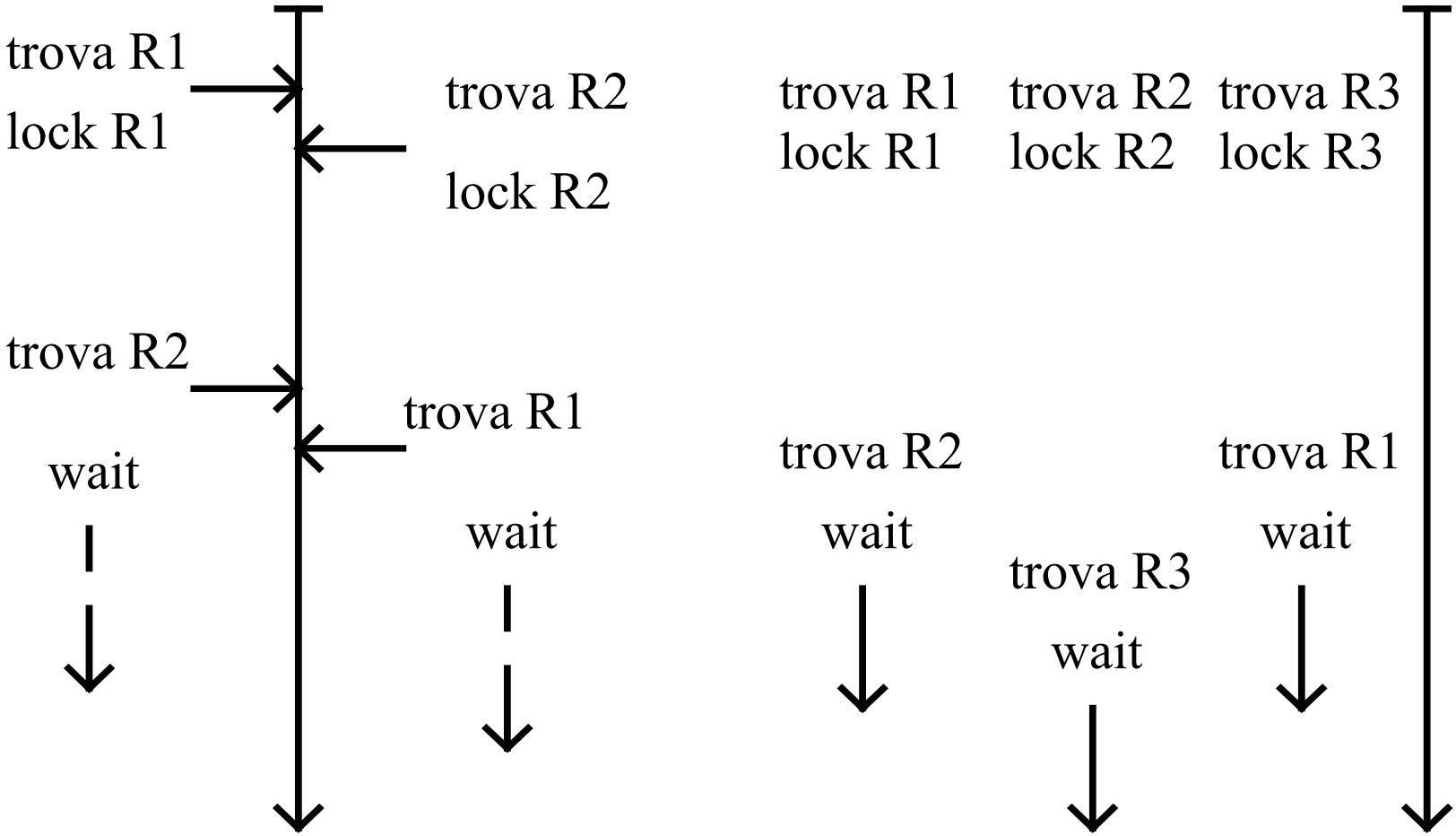
  ...
- w-lock S(10) [wait]

  …

  …

  ...

Esempi di Deadlock



Tecnica del Time-out

La tecnica di risoluzione più usata è quella del TIME-OUT.

Quando una transazione entra in stato di wait si attiva un TIME-OUT :

un'attesa eccessiva è interpretata come deadlock, dopo un certo tempo in attesa (scadenza del timeout) la transazione viene abortita.

La probabilità di avere DEADLOCKS è influenzata dalla *granularità* del LOCK (RELAZIONE, PAGINA, TUPLA).