Gestione dei dati

Parte 1
Gestione della concorrenza

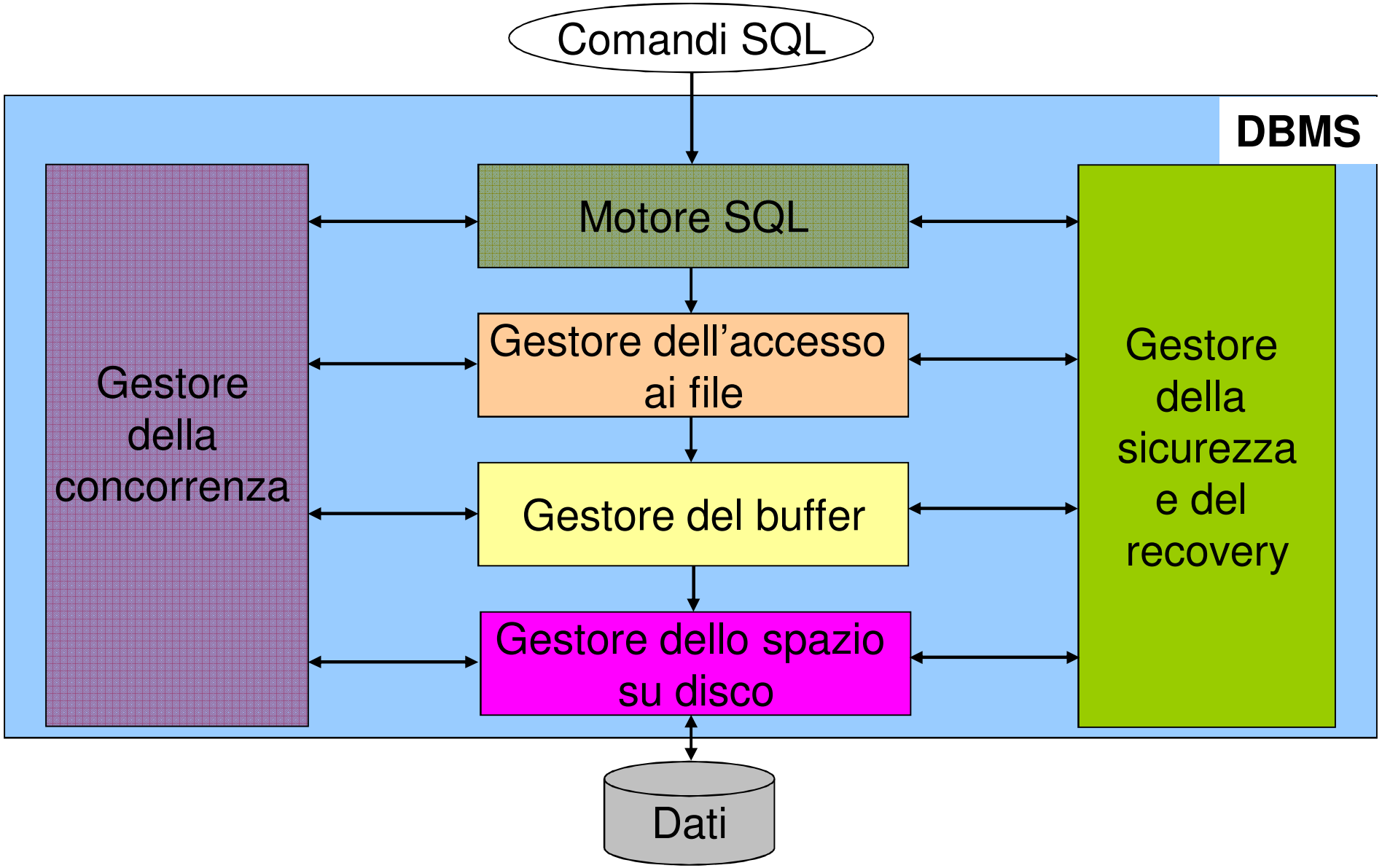
Maurizio Lenzerini, Riccardo Rosati

Facoltà di Ingegneria dell'informazione, informatica e statistica
Sapienza Università di Roma
Anno Accademico 2012/2013

http://www.dis.uniroma1.it/~rosati/gd/

SAPIENZA
UNIVERSITÀ DI ROMA

Architettura di un DBMS



Sommario

1. Transazioni, concorrenza e serializzabilità
2. View-serializzabilità
3. Conflict-serializzabilità
4. Gestione della concorrenza mediante lock
5. Recuperabilità delle transazioni
6. Gestione della concorrenza mediante timestamp

1.1
Transazioni, concorrenza e serializzabilità

Il concetto di transazione

Una transazione modella l'esecuzione di una procedura software costituita da un insieme di istruzioni che:

- prevedono letture/scritture sulla base di dati,
- e **costituiscono una singola unità (operazione) logica**

Sintatticamente, noi assumeremo che ogni transazione contenga:

- una istruzione "begin"
- una istruzione "end"
- una tra le seguenti istruzioni:
  - "commit" (rendi definitivo il lavoro svolto sulla base di dati)
  - "rollback" o "abort" (disfa il lavoro svolto sulla base di dati)

Esempio di (vera) transazione

```
begin
   writeln('Inserire importo, conto di partenza, conto di arrivo');
   read (Importo, contoPartenza, contoArrivo);
   EXEC SQL
      select Saldo into :saldoCorrente
      from ContiCorrenti
      where Numero = :contoPartenza
   if saldoCorrente < Importo
   then begin
      writeln('Saldo Insufficiente');
      ABORT;
   end;
   else begin
      EXEC SQL
         UPDATE ContiCorrenti
         set Saldo=:saldoCorrente - :Importo
         where Numero = :contoPartenza;
      writeln('Operazione eseguita con successo');
      COMMIT;
   end;
end;
```

Tabella ContiCorrenti

Numero	Saldo

Effetto di una transazione

Sia DB una base di dati

Sia T una transazione su DB

Effetto o **risultato** di T = stato della base di dati DB dopo l'esecuzione di T

Come vedremo dopo, ogni transazione deve godere di un insieme di proprietà (chiamate ACID)

Concorrenza

Il throughput di un sistema è il numero di transazioni per secondo (tps) accettate dal sistema a regime

In un DBMS, si vuole ottenere un throughput dell'ordine di 100-1000tps

Questo impone un alto grado di concorrenza tra le transazioni in esecuzione

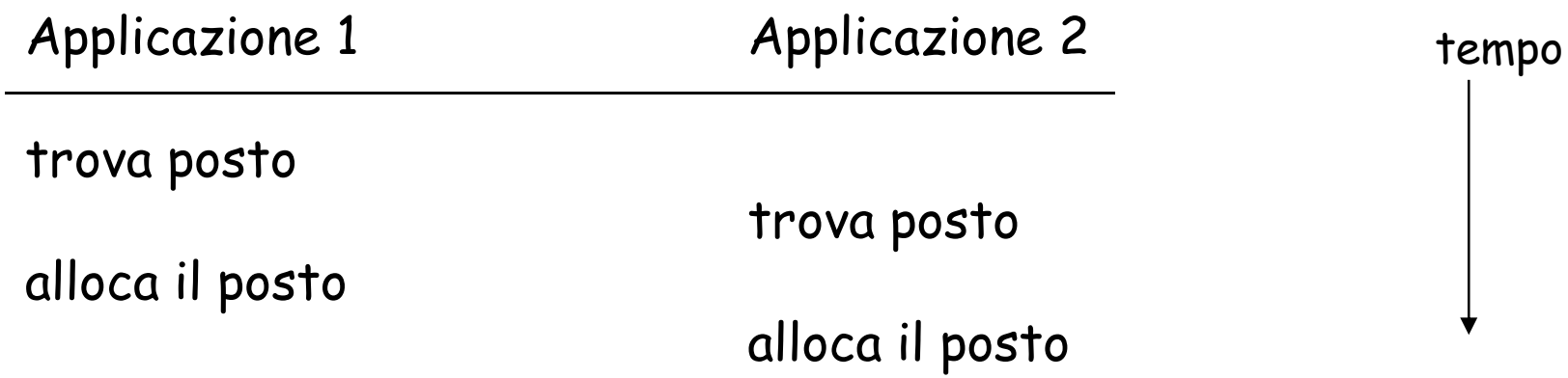
- Esempio: Se una transazione impiega mediamente 0.1 secondi ad essere eseguita, ad un throughput di 100tps corrispondono mediamente 10 transazioni in esecuzione concorrente

Esempi tipici: applicazioni bancarie, di prenotazione voli

Concorrenza: esempio

Supponiamo che una transazione venga eseguita simultaneamente da due o più applicazioni (ad esempio due agenzie di viaggi), allo scopo di prenotare un posto sullo stesso volo

È possibile un'evoluzione temporale come la seguente:



A questo punto, abbiamo una doppia prenotazione dello stesso posto

Proprietà di isolamento delle transazioni

Il DBMS transazionale gestisce questi problemi garantendo la proprietà di isolamento

La proprietà di isolamento di una transazione garantisce che essa sia eseguita come se non ci fosse concorrenza

Questa proprietà è assicurata facendo in modo che ciascun insieme di transazioni concorrenti sottoposte al sistema sia "serializzabile" (si veda dopo)

Proprietà desiderabili delle transazioni

L'isolamento è solo una delle proprietà desiderabili. La lista completa delle proprietà desiderabili (proprietà ACID) è:

1. Atomicità: per ogni transazione, o tutte le sue azioni sono eseguite, oppure nessuna sua azione viene eseguita
2. Consistenza: l'esecuzione di ogni transazione deve lasciare la base di dati in uno stato corretto, cioè in cui tutti i vincoli di integrità sono soddisfatti
3. Isolamento: ogni transazione deve essere eseguita in modo che i suoi effetti non siano influenzati da altre transazioni eseguite nello stesso intervallo di tempo
4. Durabilità: l'effetto di ogni transazione che è andata a buon fine deve essere registrato in modo permanente nella base di dati

Schedule e schedule seriali

Dato un insieme di transazioni T1,T2,…,Tn, una sequenza S di esecuzione di azioni di tali transazioni **che rispetta l'ordine nell'ambito di ogni singola transazione** (cioè tale che se l'azione a viene prima dell'azione b in Ti, allora a viene prima di b anche in S) è detta **schedule** (su T1,T2,…,Tn)

Se lo schedule non contiene tutte le azioni di tutte le transazioni, viene detto parziale

Uno schedule S è seriale se le azioni di ogni transazione in S terminano prima di ogni azione di una diversa transazione in S, ovvero se in S non si alternano azioni di transazioni diverse

Serializzabilità

Uno schedule è serializzabile se l'esito della sua esecuzione è lo stesso che si avrebbe se le azioni in esso contenute fossero eseguite in almeno una qualunque delle possibili sequenze seriali costituite dalle stesse transazioni. Più precisamente, uno schedule S su T1,T2,…,Tn è serializzabile se esiste uno schedule seriale su T1,T2,…,Tn equivalente ad S

Due schedule sono equivalenti se, per ogni stato iniziale, l'esecuzione di uno produce lo stesso risultato che produce l'altro

Serializzabilità

Ad esempio, date le due transazioni

T1 (x=x+x; x= x+2)

T2 (x= x**2; x=x+2)

(dove x denota un qualunque elemento della base di dati)

i possibili **schedule seriali** su di esse sono I seguenti:

S1: x=x+x; x= x+2; x= x**2; x=x+2

S2: x= x**2; x=x+2; x=x+x; x= x+2

Uno **schedule non seriale** è:

S3: x=x+x; x= x**2; x= x+2; x=x+2

Serializzabilità

S3: x=x+x; x= x**2; x= x+2; x=x+2

S3 è **serializzabile**?

No, infatti se ad esempio nello stato iniziale della base di dati si ha x=3, allora:

- L'effetto di S3 è x=((3+3)**2)+2+2=40
- L'effetto di S1 è x= x=(((3+3)+2)**2)+2= 66
- L'effetto di S2 è x=((3**2)+2)+((3**2)+2)+2=24

Questo dimostra che S3 non è equivalente a nessuno schedule seriale su T1 e T2, e pertanto S3 non è serializzabile

Studio della serializzabilità

Per garantire l'isolamento è necessario garantire la serializzabilità di un insieme di transazioni. Lo studio della serializzabilità prevede:

- Analisi delle possibili anomalie da non-isolamento che si possono verificare durante un'esecuzione concorrente
- Introduzione di una notazione astratta e semplificata per rappresentare le transazioni
- Definizione di diversi livelli di serializzabilità e corrispondenti tipi di anomalie che vi possono comparire
- Definizione di proprietà e protocolli per garantire i diversi livelli di serializzabilità

Trarremo una conclusione generale: la garanzia di serializzabilità va a scapito del livello di concorrenza possibile tra transazioni che accedono alle stesse risorse

In compenso, sarà possibile definire con precisione i compromessi tra throughput e isolamento

Notazione

Una transazione che va a buon fine può essere rappresentata come una sequenza di

- comandi begin/commit per delimitare la transazione
- azioni elementari di scrittura e di lettura di un elemento (attributo, record, tabella) nel database
- operazioni di lettura/modifica in memoria locale

T_1	T_2
Begin	Begin
READ(A,t)	READ(A,s)
t := t+100	s := s*2
WRITE(A,t)	WRITE(A,s)
READ(B,t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)
commit	commit

Uno schedule seriale

T_1	T_2	A 25	B 25
begin			
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
commit			
	begin		
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250
	commit		

Uno schedule serializzabile

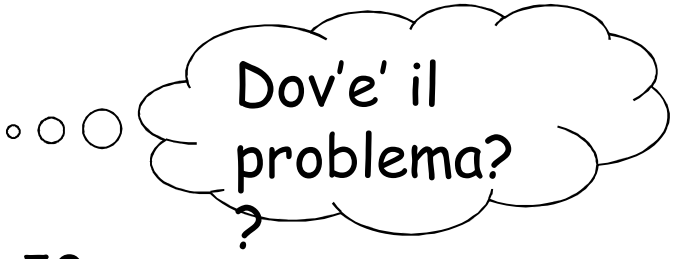
T_1	T_2	A	B
		25	25
begin	begin		
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
commit			
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250
	commit		

I valori finali di A e B sono gli stessi dello schedule seriale T_1 T_2, indipendentemente dal valore iniziale di A e B

Possiamo infatti dimostrare che, se all'inizio A=B=c (c costante), allora alla fine della esecuzione dello schedule si ha: A=B=2(c+100)

Uno schedule non serializzabile

T_1	T_2	A	B
		25	25
begin	begin		
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
	commit		
READ(B,t)			
t := t+100			
WRITE(B,t)			150
commit			



Analisi delle anomalie

Tre tipi di anomalie rispetto alla proprietà di isolamento (e alla serializzabilità):

1. perdita di aggiornamento (**update loss**)
2. lettura non ripetibile (**unrepeatable read**)
3. aggiornamento fantasma (**ghost update**)

Perdita di aggiornamento (update loss)

- Date due transazioni T_1, T_2 identiche:

  READ(A, x), x := x + 1, WRITE(A, x)

- L'esecuzione seriale con valore iniziale A=2 produce A=4, risultato di due aggiornamenti successivi
- Si consideri ora lo schedule seguente:

T_1	T_2
begin	begin
READ(A,x)	
x := x+1	
	READ(A,x)
	x := x+1
WRITE(A,x)	
commit	
	WRITE(A,x)
	commit

Il risultato finale è A=3, ed il primo dei due aggiornamenti si è perso: T_2 legge il valore iniziale di A, e scrive il valore finale. In questo caso, è come se T1 non fosse stato eseguito!

Lettura non ripetibile (unrepeatable read)

T_1 esegue due letture consecutive dello stesso dato:

T_1	T_2
begin	begin
READ(A,x)	
	READ(A,x)
	x := x+1
	WRITE(A,x)
READ(A,x)	commit
commit	

Tuttavia, a causa dell'aggiornamento concorrente da parte di T2, T1 legge due valori consecutivi diversi nel contesto della stessa transazione

Aggiornamento fantasma (ghost update)

Si assuma il vincolo di integrità A + B = 1000

T_1	T_2
begin	begin
READ(A,y)	
	READ(A,y)
	y := y-100
	READ(B,z)
	z = z+100
	WRITE(A,y)
	WRITE(B,z)
	commit
READ(B,z)	
// qui y+z = 1100	
commit	

T1 legge il valore di A iniziale, mentre legge un valore di B che è stato modificato da T2. Ciò ha l'effetto di presentare a T1 uno stato in cui il vincolo è violato. Si noti ancora una volta che sia T1 sia T2 in situazione di isolamento portano invece ad un risultato corretto

Scheduler

Lo **scheduler** fa parte del gestore della concorrenza ed opera così:

- Accoglie una transazione e le assegna un identificatore unico
- Comanda al buffer manager del DBMS di leggere/scrivere sul DB secondo una particolare sequenza
- Non è in grado di interpretare né le specifiche operazioni sui dati in memoria locale, né vincoli sull'ordine di esecuzione delle transazioni (**ogni sequenza seriale delle transazioni che si presentano allo scheduler è accettabile**): la serializzabilità non può dipendere né dal tipo di operazioni eseguite né dalla base di dati in input

Notazione semplificata

Per quanto detto in precedenza, nel seguito della trattazione riguardante la serializzabilità, possiamo semplificare la notazione per le transazioni:

- ogni transazione è denotata da Ti (dove i è un indice intero non negativo che identifica la transazione stessa)
- ogni azione della transazione diversa da read, write o commit viene ignorata (non viene rappresentata)
- ogni azione (read, write, o commit) della transazione Ti ha il pedice I

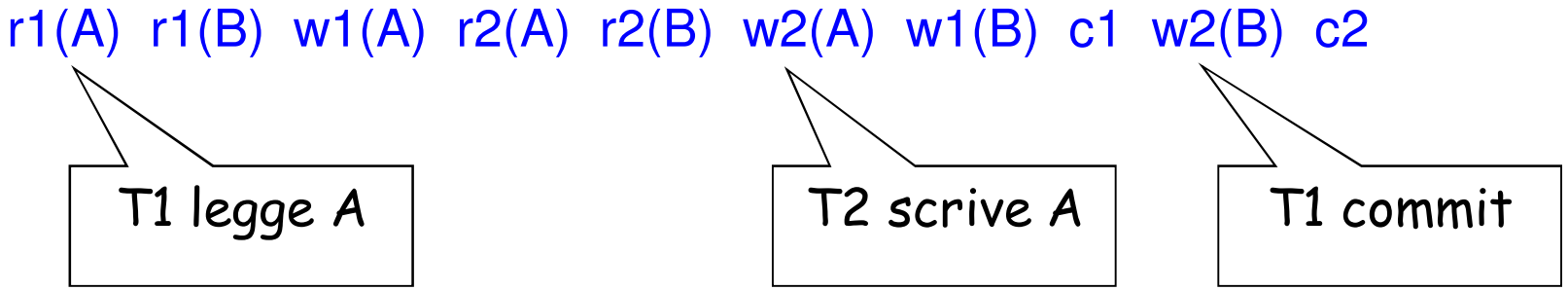
Notazione semplificata

Nella notazione semplificata, le transazioni degli esempi precedenti si scrivono:

T1: r1(A) r1(B) w1(A) w1(B) c1

T2: r2(A) r2(B) w2(A) w2(B) c2

Un esempio di schedule (completo):



Serializzabilità ed equivalenza tra schedule

La definizione generale di serializzabilità che abbiamo visto prima si basa sulla nozione di equivalenza tra due schedule: Uno schedule è serializzabile se e solo se esso è equivalente ad uno schedule seriale sulle stesse transazioni

A seconda del livello di astrazione che adottiamo per caratterizzare gli effetti delle transazioni, otteniamo diverse definizioni della relazione di equivalenza, alle quali corrispondono diversi livelli di serializzabilità

Data una certa definizione di equivalenza, interessa definire due tipi di algoritmi:

- Algoritmi per il test di equivalenza: dati due schedule, determina se essi sono equivalenti
- Algoritmi per il test di serializzabilità: dato uno schedule, determina se esso è equivalente ad una qualunque schedule seriale

Uno dei metodi più usati per gestire la concorrenza si basa sulla definizione di regole di comportamento (protocolli) dello scheduler e delle transazioni che garantiscono un accettabile livello di serializzabilità, senza eseguire alcun test esplicito

Due assunzioni importanti

Facciamo per il momento due assunzioni importanti

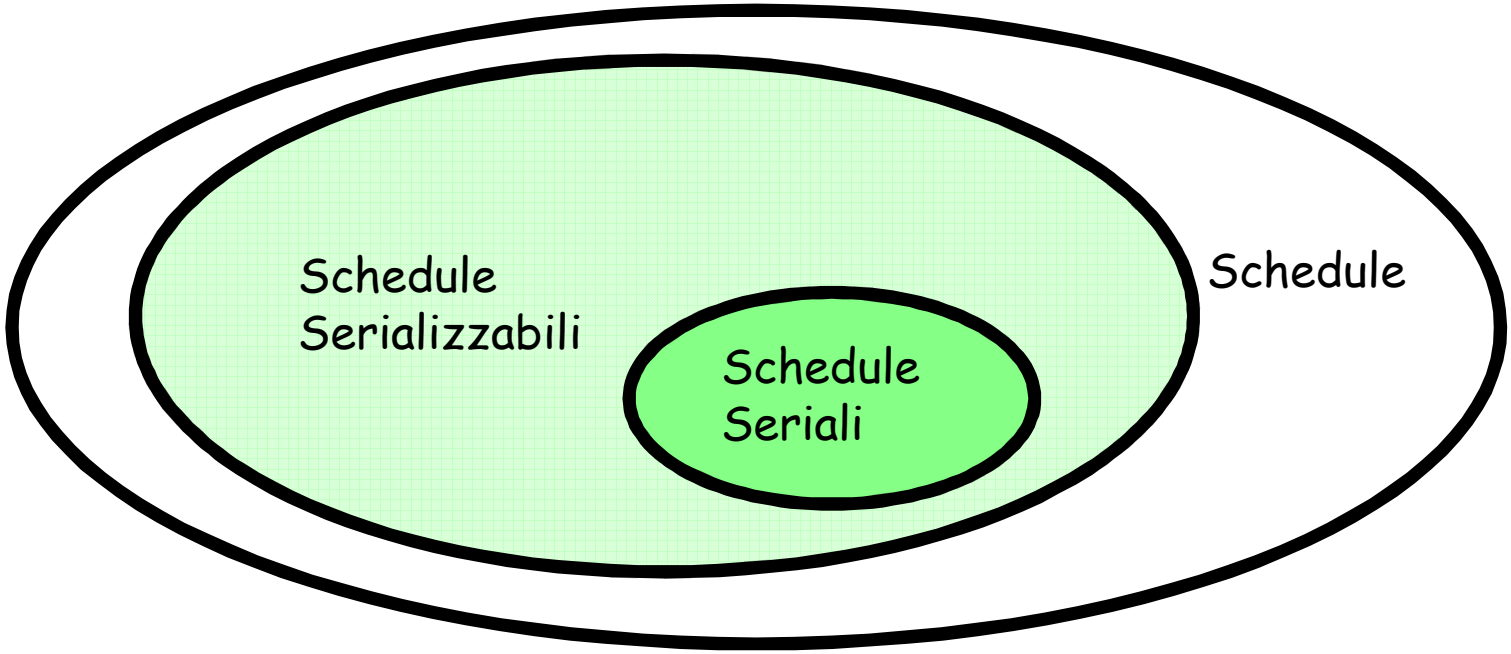
1. Nessuna transazione legge o scrive lo stesso dato due volte

2. Nessuna transazione esegue il rollback (ovvero, tutte le transazioni vanno a buon fine)

Successivamente, rilasseremo la seconda assunzione

Classi di schedule

Idea base del nostro studio: individuare classi di schedule serializzabili che siano sottoclassi delle schedule possibili e la cui proprietà di serializzabilità sia verificabile con complessità ragionevole



Definiremo diverse nozioni di serializzabilità, iniziando con:
- la view-serializzabilità
- la conflict-serializzabilità

1.2
View-serializzabilità

View-equivalenza e view-serializzabilità

Definizioni preliminari:

- in uno schedule S, diciamo che $r_i(x)$ LEGGE-DA $w_j(x)$ se $w_j(x)$ precede $r_i(x)$ in S, e se tra $w_j(x)$ e $r_i(x)$ non c'è alcuna azione $w_k(x)$
- in uno schedule S, diciamo che $w_i(x)$ è una SCRITTURA FINALE se $w_i(x)$ è l'ultima azione di scrittura su x in S

Definizione di view-equivalenza: siano S1 e S2 due schedule (completi) sulle stesse transazioni. Allora S1 è view-equivalente a S2 se S1 ed S2 hanno la stessa relazione LEGGE-DA e le stesse SCRITTURE FINALI

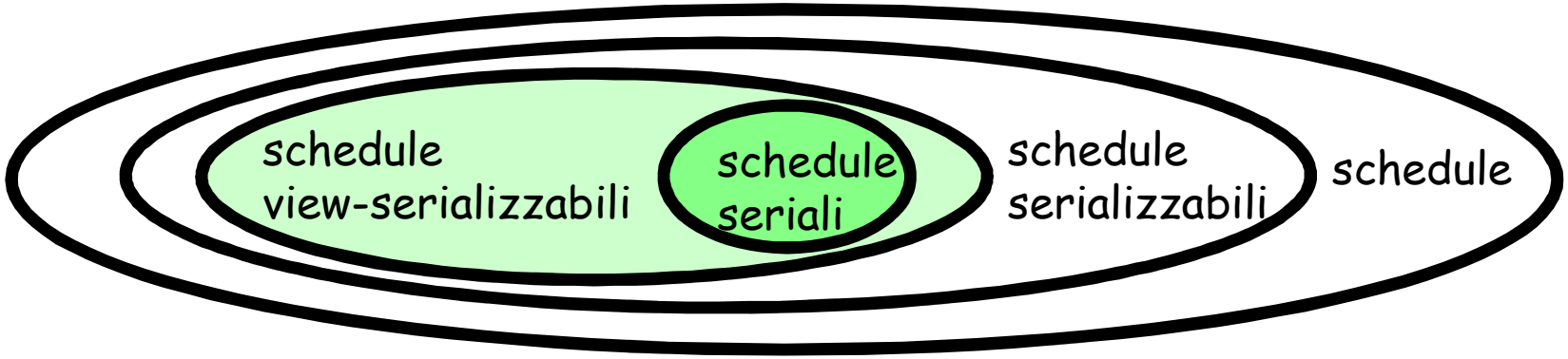
Definizione di view-serializzabilità: uno schedule S (completo) è view-serializzabile se esiste uno schedule S' seriale che è view-equivalente ad S

View-serializzabilità

- Esistono schedule serializzabili che non sono view-serializzabili. Ad esempio:

  read1(A,t) read2(A,s) s:=100 write2(A,s) t:=100 write1(A,t)

  è serializzabile, ma non view-serializzabile
- Si noti però che per dedurre che lo schedule dell'esempio precedente è serializzabile, dobbiamo tenere conto delle operazioni che le transazioni eseguono sulla memoria locale
- Se invece ci limitiamo al modello astratto di transazione che noi consideriamo (in cui, in particolare, teniamo conto delle sole operazioni di lettura e scrittura), allora la view-equivalenza e la view-serializzabilità sono le nozioni più generali possibili



Proprietà della view-equivalenza

- Dati due schedule, determinare se essi sono view-equivalenti ha complessità polinomiale rispetto alla dimensione dei due schedule

- Dato uno schedule, verificare se esso è view-serializzabile è un problema NP-completo
  - che il problema sia in NP è semplice da verificare: infatti un algoritmo non deterministico polinomiale per verificare se S è view-serializzabile è: scegli non deterministicamente uno schedule S' seriale sulle stesse transazioni di S, e verifica in tempo polinomiale se S' è view-equivalente ad S
  - che il problema sia NP-hard è molto più complicato da dimostrare

- Per questo, la view-serializzabilità non è utilizzata in pratica

Esercizio 1

- Considerare i seguenti schedule:
  1. w0(x) r2(x) r1(x) w2(x) w2(z)
  2. w0(x) r1(x) r2(x) w2(x) w2(z)
  3. w0(x) r1(x) w1(x) r2(x) w1(z)
  4. w0(x) r1(x) w1(x) w1(z) r2(x)

e dire quali di essi sono view-serializzabili

- Analizzare i seguenti schedule, verificare che non sono view-serializzabili, e dire quali anomalie presentano
  1. r1(x) r2(x) w2(x) w1(x)
  2. r1(x) r2(x) w2(x) r1(x)
  3. r1(x) r2(x) r2(y) w2(x) w2(y) r1(y)

1.3
Conflict-serializzabilità

La nozione di conflitto

Due azioni sono in conflitto in uno schedule se:

- appartengono a diverse transazioni
- agiscono sullo stesso elemento
- almeno una delle azioni è un'operazione di scrittura (write)

La nozione di conflitto

Si può facilmente osservare che:

- **se due azioni consecutive che appartengono a due transazioni non sono in conflitto, allora possono essere scambiate** senza alterare l'effetto dello schedule. Infatti:
  - coppie di letture consecutive di uno stesso elemento in transazioni diverse possono essere scambiate
  - una lettura nella transazione T1 di un elemento X ed una consecutiva scrittura nella transazione T2 di un elemento Y≠X possono essere scambiate
- **due azioni consecutive che appartengono alle stesse transazioni non possono essere scambiate**, perché l'effetto della transazione potrebbe cambiare

La nozione di conflitto

- **due azioni consecutive che appartengono a transazioni diverse e sono in conflitto non possono essere scambiate**, perché:
  - scambiando l'ordine di due scritture di transazioni diverse w1(A) w2(A) che agiscono consecutivamente sullo stesso elemento, può cambiare il valore finale di A
  - scambiando l'ordine di lettura e scrittura consecutive r1(A) w2(A) di transazioni diverse sullo stesso elemento, T1 può leggere valori diversi di A (prima e dopo la scrittura da parte di T2, rispettivamente)

- (ricordiamo inoltre che **due azioni consecutive che appartengono alla stessa transazione non possono essere scambiate** per la definizione stessa di schedule)

Conflict-equivalenza

Due schedule S1 e S2 sulle stesse transazioni sono conflict-equivalenti se S1 può essere trasformato in S2 mediante una sequenza di scambi (swaps) tra azioni consecutive non in conflitto tra loro

Esempio:

S = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)

è conflict-equivalente a:

S' = r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)

perché può essere trasformato in S' mediante la seguente sequenza di scambi:

r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)
r1(A) w1(A) r2(A) r1(B) w2(A) w1(B) r2(B) w2(B)
r1(A) w1(A) r1(B) r2(A) w2(A) w1(B) r2(B) w2(B)
r1(A) w1(A) r1(B) r2(A) w1(B) w2(A) r2(B) w2(B)
r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)

Esercizio 2

Dimostrare la seguente proprietà.

Due schedule S1 e S2 sulle stesse transazioni T1,…,Tn sono conflict-equivalenti se e solo se non esistono due azioni a_i di Ti e b_j di Tj (con Ti e Tj appartenenti a T1,…Tn) tali che

- a_i e b_j sono in conflitto tra loro
- l'ordine con cui le due azioni compaiono (anche non consecutivamente) in S1 è diverso dall'ordine in cui esse compaiono (anche non consecutivamente) in S2

Conflict-serializzabilità

Uno schedule S è conflict-serializzabile se esiste uno schedule S' seriale che è conflict-equivalente ad S

Come si verifica la proprietà di conflict-serializzabilità?

Analizzando il grafo di precedenza associato ad uno schedule.

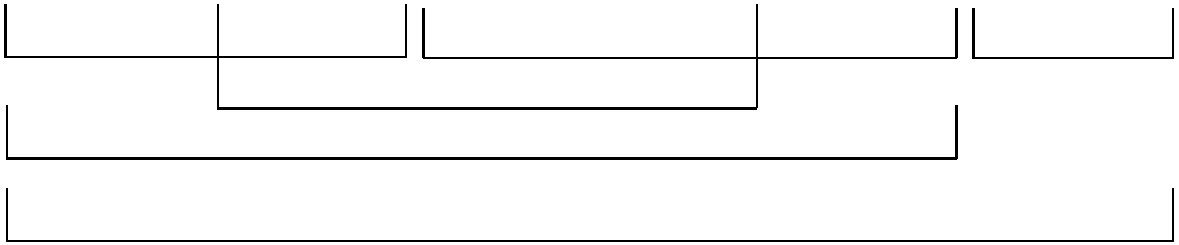
Grafo di precedenza

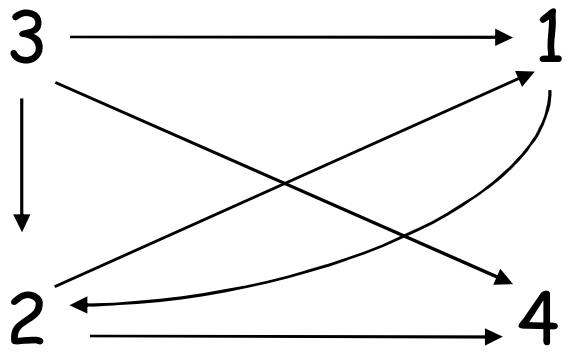
Dato uno schedule S su T1,…,Tn, il **grafo di precedenza** P(S) associato ad S è definito così:

- i nodi di P(S) sono le transazioni {T1,…, Tn} di S
- gli archi E di P(S) sono determinati così: l'arco Ti → Tj è in E se e solo se esistono due azioni Pi(A), Qj(A) che agiscono sullo stesso oggetto A in S tali che:
  1. $Pi(A) <_S Qj(A)$ (cioè Pi(A) viene prima di Qj(A) in S, anche non consecutivamente)
  2. almeno una tra Pi(A) e Qj(A) è un'operazione di write

Esempio di grafo di precedenza

S: $w_3(A)\ w_2(C)\ r_1(A)\ w_1(B)\ r_1(C)\ w_2(A)\ r_4(A)\ w_4(D)$





Come usare il grafo delle precedenze

Teorema (conflict-serializzabilità) Uno schedule S è conflict-serializzabile se e solo se il grafo delle precedenze P(S) associato ad S è aciclico

Per dimostrare il teorema:

- osserviamo subito che se S è uno schedule seriale, allora il grafo delle precedenze P(S) è aciclico (si dimostra immediatamente)
- dimostriamo un lemma preliminare

Lemma preliminare

Lemma Se due schedule S1 ed S2 sono conflict-equivalenti, allora P(S1)=P(S2)

Dimostrazione Siano S1 ed S2 due schedule conflict-equivalenti, ed assumiamo che P(S1) ≠ P(S2). Allora P(S1) e P(S2) hanno gli stessi nodi ed archi diversi, cioè esiste un arco Ti → Tj in P(S1) che non è in P(S2). Ma Ti → Tj in P(S1) significa che S1 ha la forma:

…pi(A)... qj(A)…

con pi, qj in conflitto. Poiché P(S2) ha gli stessi nodi di P(S1), S2 contiene qj(A) e pi(A), e siccome P(S2) non contiene l'arco Ti → Tj, dobbiamo concludere che qj(A) $<_{S2}$ pi(A). Ma allora S1 e S2 differiscono nell'ordine di una coppia di azioni in conflitto, e quindi non possono essere trasformate una nell'altra tramite scambio di azioni non in conflitto, cioè non sono conflict-equivalenti, ed otteniamo una contraddizione. Dobbiamo quindi concludere che P(S1)=P(S2).

Non vale il viceversa

Se valesse il viceversa del lemma precedente, il teorema sarebbe dimostrato.Tuttavia, il viceversa del lemma precedente non vale, perché si può dimostrare che P(S1)=P(S2) non implica che S1 ed S2 siano conflict-equivalenti.

Infatti:

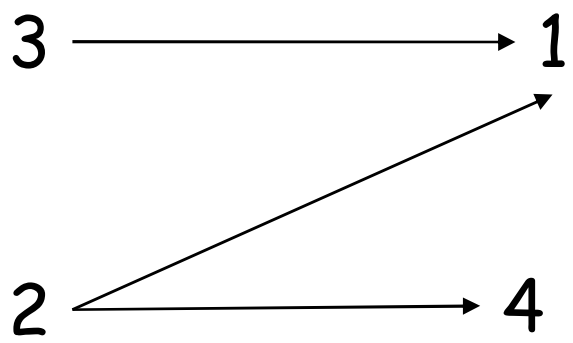
S1 = w1(A) r2(A) w2(B) r1(B)
S2 = r2(A) w1(A) r1(B) w2(B)

hanno lo stesso grafo delle precedenze, ma non sono conflict-equivalenti, perché passare da S1 a S2 richiede scambi tra azioni in conflitto.

Ordine topologico di un grafo

Dato un grafo G, si dice ordine topologico di G un qualunque ordinamento totale S (cioè sequenza) dei nodi di G tale che se l'arco Ti → Tj è nel grafo G, allora Ti viene prima di Tj nella sequenza S.

Esempio:



Possibili ordini topologici:
3 2 1 4
3 2 4 1
2 3 4 1
2 3 1 4

Le seguenti proposizioni sono facili da dimostrare:

- se il grafo G è aciclico, allora esiste almeno un ordine topologico di G
- se S è un ordine topologico di G, ed esiste un cammino dal nodo n1 al nodo n2 in G, allora n1 viene prima di n2 in S

Esercizio 3

Dimostrare le due proposizioni appena menzionate, ovvero:

1. Se il grafo G è aciclico, allora esiste almeno un ordine topologico di G

2. Se S è un ordine topologico di G, ed esiste un cammino dal nodo n1 al nodo n2 in G, allora n1 viene prima di n2 in S

Dimostrazione del teorema della conflict-serializzabilità

Prima parte ($\Leftarrow$):

Dobbiamo dimostrare che se S è conflict-serializzabile, allora il grafo delle precedenze P(s) è aciclico. Se S è conflict-serializzabile, allora esiste uno schedule seriale S' sulle stesse transazioni che è conflict-equivalente ad S. Siccome S' è seriale, il grafo delle precedenze P(S') associato ad S' è aciclico. Ma per il lemma preliminare, poiché S è conflict-equivalente ad S', si ha che P(S)=P(S'), e quindi possiamo concludere che P(S) è aciclico.

Dimostrazione del teorema della conflict-serializzabilità

Seconda parte (⇒):

Assumiamo che S sia definito sulle transazioni T1,…,Tn, e che P(S) sia aciclico. Allora esiste almeno un ordine topologico di P(S), cioè una sequenza dei suoi nodi tale che se Ti → Tj è nel grafo, allora Ti viene prima di Tj nella sequenza. Ad un tale ordine topologico di P(S) corrisponde lo schedule seriale S' su T1,…,Tn tale che se Ti → Tj è nel grafo, allora tutte le azioni di Ti vengono prima di tutte quelle di Tj in S'. Si può vedere facilmente che questo schedule seriale è conflict-equivalente ad S. Infatti, se non lo fosse, ci sarebbe una coppia di azioni in conflitto a_h e b_k tale che ($a_h <_{S'} b_k$) e ($b_k <_S a_h$). Ma ($b_k <_S a_h$) significa che il cammino Tk → … → Th è nel grafo P(S), e quindi (vedi Esercizio 3.2) Tk viene prima di Th in ogni ordine topologico di P(S). Tuttavia, ($a_h <_{S'} b_k$) significa che Th viene prima di Tk in S', e questo contraddice il fatto che S' corrisponde ad un ordine topologico di P(S).

Algoritmo per la conflict-serializzabilità

Il teorema appena dimostrato ci consente di derivare immediatamente un algoritmo per verificare se uno schedule è conflict-serializzabile:

- costruisci il grafo delle precedenze P(S) relativo ad S
- verifica se P(S) è aciclico
- restituisci true se P(S) è aciclico, false altrimenti

Verifica di aciclicità di un grafo

come si fa a verificare se un grafo è aciclico?

esiste un algoritmo che è in grado di fare questa verifica in tempo polinomiale

ricordiamo che, dato un grafo G, un nodo N di G si dice nodo **sorgente** se non ha archi entranti

Algoritmo per la verifica di aciclicità

INPUT: grafo G
OUTPUT: se G è aciclico stampa un ordine topologico di G,
altrimenti stampa “G è ciclico”

```
{
  while G contiene almeno un nodo do {
    if in G non c’è nessun nodo sorgente
    then stampa “G è ciclico”
    else {
      seleziona un nodo N sorgente
      stampa l’etichetta di N
      elimina da G il nodo N e tutti gli archi uscenti da N
    }
  }
}
```

Complessità della verifica di conflict-serializzabilità

l'algoritmo per la verifica di aciclicità del grafo ha complessità polinomiale

a questo punto siamo in grado di stabilire la complessità del precedente algoritmo per la conflict-serializzabilità:

è immediato verificare che l'algoritmo ha complessità polinomiale rispetto alla dimensione dello schedule S

Schedule seriali conflict-equivalenti

Dalla teorema della conflict-serializzabilità segue una interessante proprietà:

se T è un ordine topologico del grafo delle precedenze di S, allora T rappresenta uno schedule seriale che è conflict-equivalente a S (e viceversa)

Pertanto: se il precedente algoritmo di verifica della aciclicità del grafo delle precedenze G restituisce un ordine topologico, tale ordine topologico rappresenta uno schedule seriale equivalente allo schedule il cui grafo delle precedenze è G

Esercizio 4

Dire se il seguente schedule è conflict serializzabile:

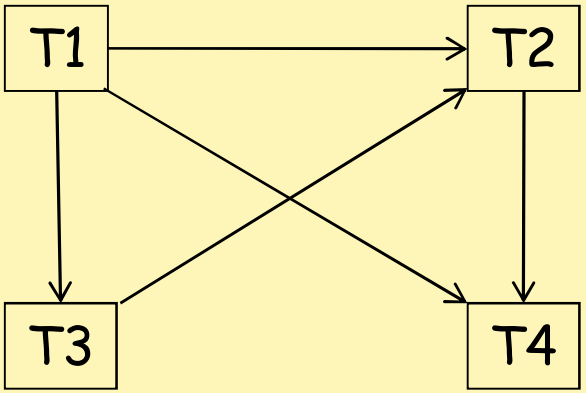
w1(x) r2(x) w1(z) r2(z) r3(x) r4(z) w4(z) w2(x)

In caso positivo, scrivere uno schedule seriale che è conflict-equivalente a tale schedule.

Soluzione esercizio 4

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

Grafo delle precedenze di S:



Il grafo è aciclico , quindi S è conflict-serializzabile

L'unico ordine topologico è T1 T3 T2 T4

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.

Confronto con la view-serializzabilità

La prima proprietà importante da osservare per effettuare un confronto tra conflict-serializzabilità e view-serializzabilità è la seguente:

Teorema Siano S1 e S2 due schedule sulle stesse transazioni. Se S1 e S2 sono conflict-equivalenti, allora sono view-equivalenti

Sulla base del teorema precedente si può poi dimostrare:

Teorema Sia S uno schedule. Se S è conflict-serializzabile, allora è anche view-serializzabile

Esercizio 5

Dimostrare i due teoremi precedenti.

Confronto con la view-serializzabilità

Abbiamo visto che uno schedule che è conflict-serializzabile è anche view-serializzabile.

È importante osservare che, al contrario, esistono schedule che sono view-serializzabili ma che non sono conflict-serializzabili.

Ad esempio,

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

è view-serializzabile, ma non è conflict-serializzabile

Esercizio 6

Considerare il seguente schedule

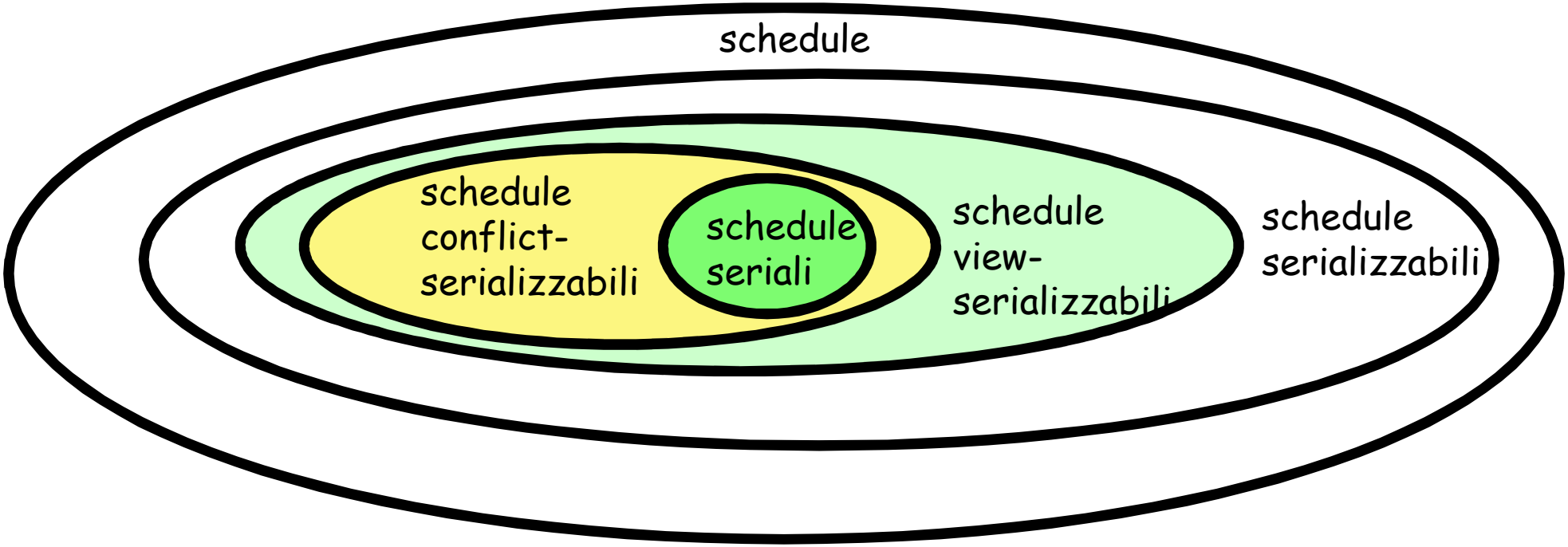
$$w_1(y)\ w_2(y)\ w_2(x)\ w_1(x)\ w_3(x)$$

e

- dire se è view-serializzabile o no
- dire se è conflict-serializzabile o no.

Confronto tra conflict- e view-serializzabilità

Graficamente, il confronto può essere riassunto nel seguente modo:



Scheduler basato sulla conflict-serializzabilità

Uno scheduler basato sulla conflict-serializzabilità:

- riceve la sequenza di azioni delle transazioni attive, nell'ordine concorrente in cui gli vengono proposte
- mantiene il grafo delle precedenze associato
- ogni volta che gli viene proposta una nuova azione, aggiorna il grafo delle precedenze dello schedule corrente (non necessariamente completo), e
  - se è stato introdotto un ciclo, annulla la transazione alla quale appartiene l'azione che ha introdotto il ciclo (l'annullamento di una transazione è un'operazione complessa, che può avere conseguenze anche su altre transazioni)
  - altrimenti, accetta l'azione e continua

Scheduler basato sulla conflict-serializzabilità

Poiché mantenere il grafo delle precedenze può essere molto costoso computazionalmente (tenendo presente che il grafo può facilmente essere dell'ordine dei 1000 nodi), la conflict-serializzabilità non è usata nei sistemi commerciali.

Tuttavia, al contrario della view-serializzabiità, la conflict-serializzabilità è utilizzata in applicazioni sofisticate in cui il controllo della concorrenza è affidato ad un sottosistema specializzato (e tolto dalla giurisdizione del DBMS)

1.4
Gestione della concorrenza mediante lock

Controllo di concorrenza tramite lock

- Abbiamo visto che la view-serializzabilità e la conflict-serializzabilità non sono usati nei sistemi commerciali
- Analizzeremo ora un metodo per il controllo della concorrenza che invece viene usato nei sistemi commerciali. Questo metodo è basato sull'utilizzo dei lock
- Nei metodi basati sui lock, una transazione, per operare su un elemento della base di dati deve chiedere ed ottenere il permesso. Il lock è un meccanismo che consente appunto ad una transazione di ottenere il permesso di operare su un elemento della base di dati

Primitive di lock esclusivo

- Per il momento, considereremo lock esclusivi, e nel seguito introdurremo anche lock non esclusivi
- Introduciamo allora due nuove operazioni (oltre a quelle di lettura e scrittura), che richiedono l'utilizzo ed il rilascio esclusivo di una risorsa (elemento A della base di dati):
  - Lock (esclusivo): $l_i(A)$
  - Unlock: $u_i(A)$
- La prima operazione $l_i(A)$ significa che la transazione Ti richiede l'uso esclusivo dell'elemento A della base di dati
- La seconda operazione $u_i(A)$ significa che la transazione Ti rilascia il lock su A, cioè rinuncia all'uso di A

Transazioni ben formate e schedule legali

Nell'uso dei lock esclusivi, dobbiamo rispettare due regole:

- Regola 1: Diciamo che una transazione Ti è ben formata se ogni azione pi(A) (lettura o scrittura di A) da parte di Ti è contenuta in una "sezione critica" definita da una coppia di lock-unlock su A:

  Ti: … $l_i(A)$ … $p_i(A)$ … $u_i(A)$ ...

- Regola 2: diciamo che uno schedule S con lock è legale se nessuna transazione esegue l'operazione di lock su A nel momento in cui un'altra transazione ha già ottenuto l'uso esclusivo di A e non ha eseguito l'operazione di unlock su A

S: ……$l_i(A)$ …….. $u_i(A)$ ……

no $l_j(A)$

Esempi di schedule con lock esclusivi

S1: l1(A) l1(B) r1(A) w1(B) l2(B) u1(A) u1(B) r2(B) w2(B) u2(B) l3(B) r3(B) u3(B)

T1 ben formata, ma
S1 non legale

S2: l1(A) r1(A) w1(B) u1(A) u1(B) l2(B) r2(B) w2(B) l3(B) r3(B) u3(B)

T1 non ben formata:
write senza lock.
T2 non ben formata:
lock senza unlock.

S2 non legale

S3:

l1(A) r1(A) u1(A)	l1(B) w1(B) u1(B)	l2(B) r2(B) w2(B) u2(B)	l3(B) r3(B) u3(B)

T1, T2, T3 ben
formate, ed S3 legale

Scheduler basato su lock esclusivi

In un sistema per la gestione della concorrenza basato su lock esclusivi, lo scheduler si comporta nel seguente modo:

1. Se arriva una richiesta di operazione da parte della transazione Ti, e tale richiesta dimostra che la transazione non è ben formata, lo scheduler annulla la transazione Ti (con tutte le conseguenze del caso)
2. Quando arriva una richiesta di lock su A da parte di una transazione Ti mentre un'altra transazione Tj ha l'uso esclusivo di A, riconosce che lo schedule corrente diventerebbe illegale se Ti ottenesse l'uso di A, e quindi blocca Ti, facendola procedere nella sua esecuzione solo dopo che Tj avrà eseguito l'operazione di unlock su A
3. Lo scheduler fa procedere lo schedule aggiornando la tabella dei lock

Quindi, lo scheduler assicura che lo schedule corrente sia costituito da transazioni ben formate e sia legale.

Esempio di comportamento dello scheduler

T1	T2
l1(A); r1(A)	
A:=A+100; w1(A);	
	l2(A) – bloccato!
l1(B); r1(B); u1(A);	
	l2(A) – ripreso!
	r2(A)
	A:=Ax2; w2(A); u2(A)
B:=B+100; w1(B); u1(B)	
	l2(B); r2(B)
	B:=Bx2; w2(B); u2(B)

Le due regole non sono sufficienti per la serializzabilità

T1	T2	A 25	B 25
l1(A); r1(A)			
A:=A+100; w1(A); u1(A)		125	
	l2(A); r2(A)		
	A:=Ax2; w2(A); u2(A)	250	
	l2(B); r2(B)		
	B:=Bx2; w2(B); u2(B)		50
l1(B); r1(B)			
B:=B+100; w1(B); u1(B)			150
		250	150

Aggiornamento fantasma: l'isolamento viene "rotto" nonostante i lock

Two-Phase Locking (con lock esclusivi)

Abbiamo appena visto che le regole che impongono
- transazioni ben formate
- schedule legali

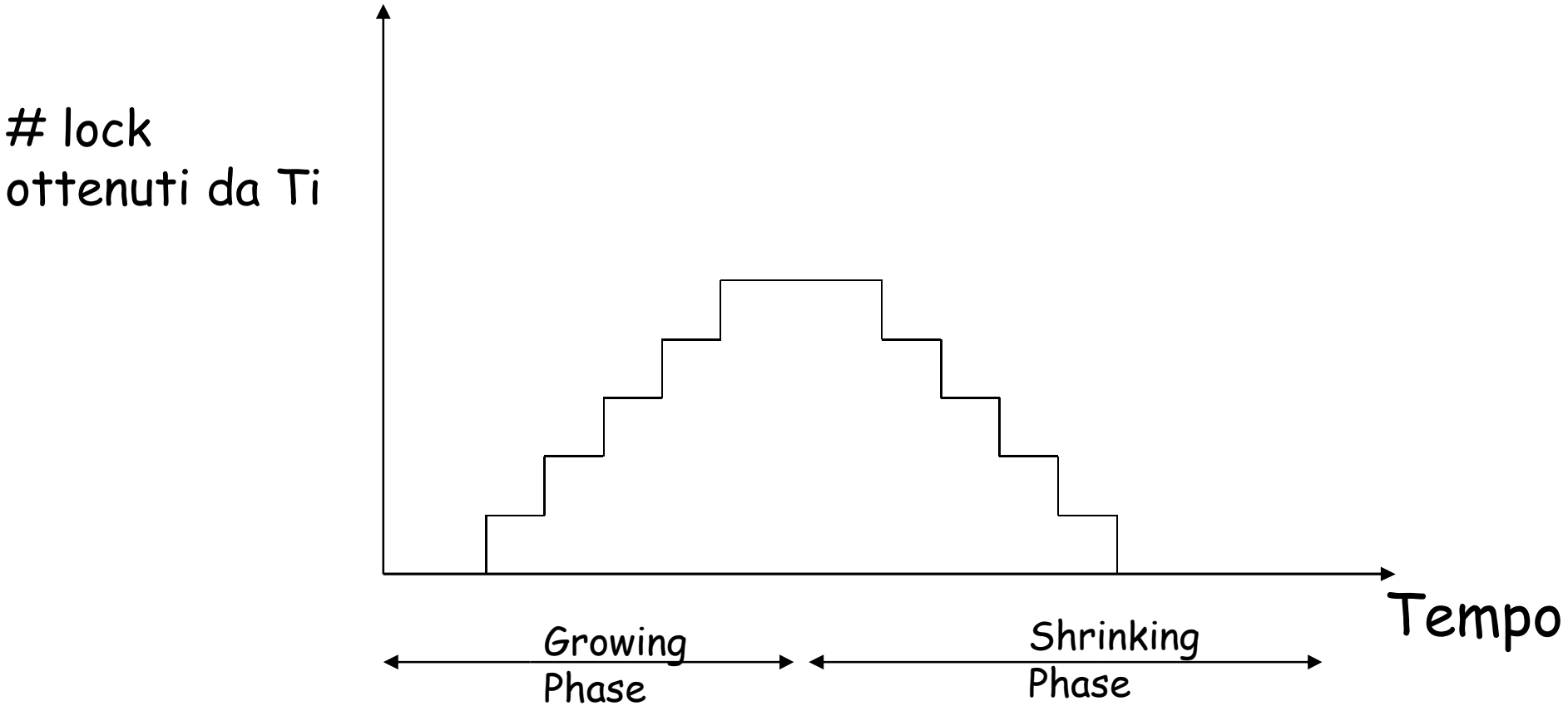
non sono sufficienti a garantire la serializzabilità, anche in presenza di lock.

Al fine di giungere ad una politica corretta di controllo di concorrenza mediante l'uso dei lock esclusivi, si introduce un'ulteriore regola (o protocollo), detta del "Two-Phase Locking (2PL)":

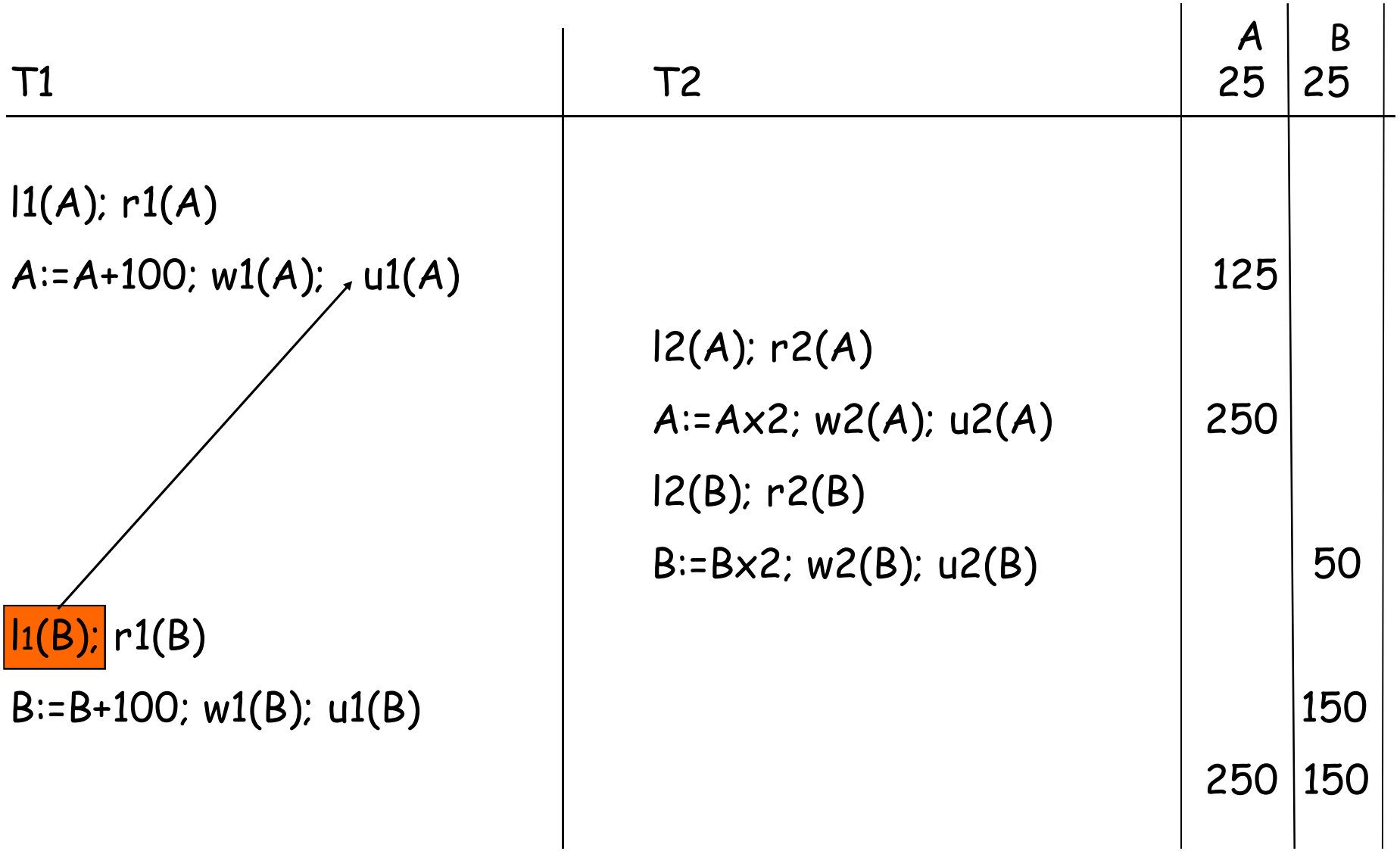
Uno schedule (con lock esclusivi) segue il protocollo del two-phase locking se in ogni transazione Ti presente nello schedule tutte le operazioni di lock di Ti precedono tutte le operazioni di unlock di Ti.

Le due fasi del Two-Phase Locking

Schema di lock e unlock in una transazione che segue il protocollo 2PL (con lock esclusivi)

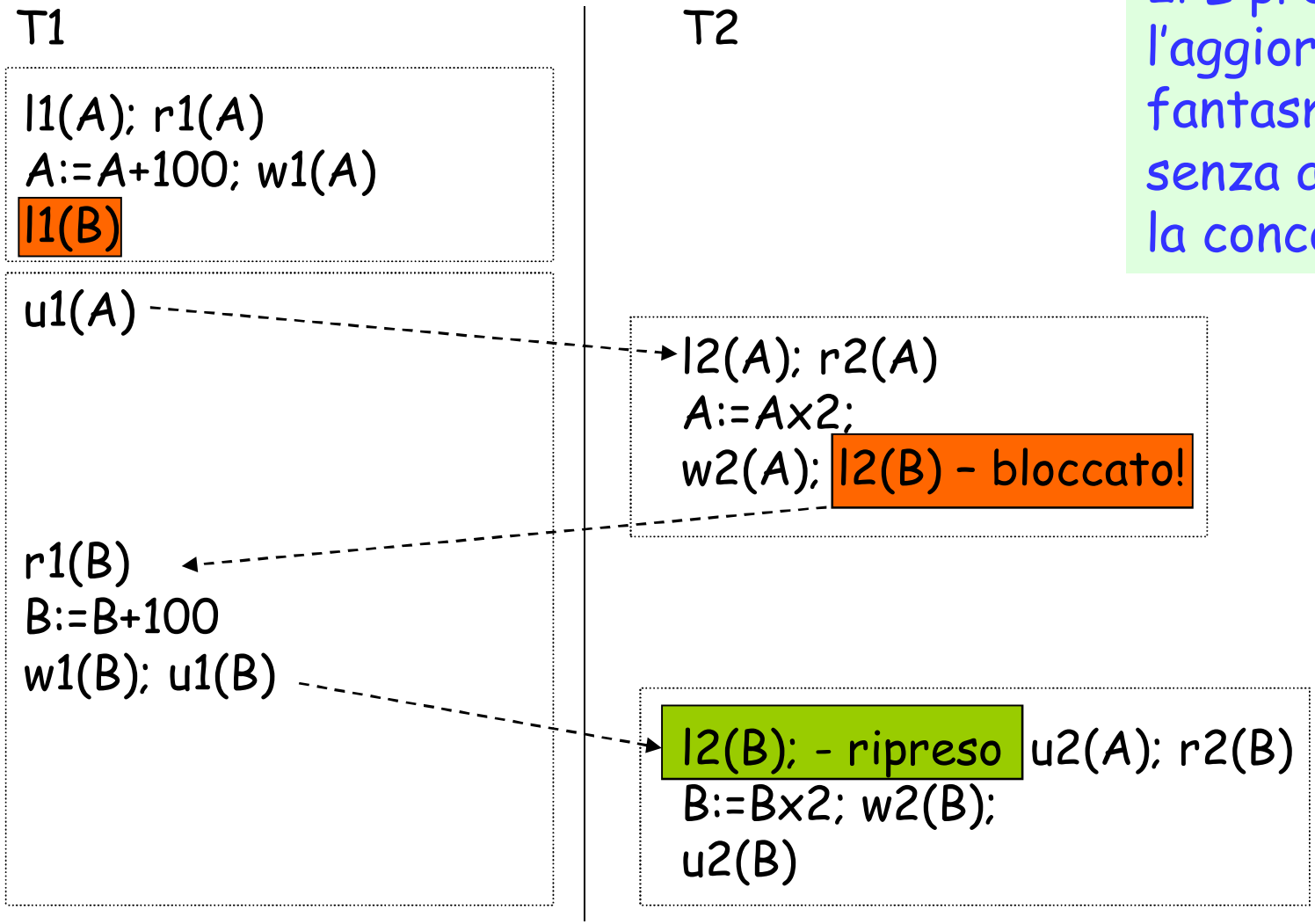


Trasformiamo lo schedule dell'esempio precedente in schedule 2PL



Esempio di scheduler nel protocollo 2PL

Anche se le transazioni seguono il 2PL, lo scheduler deve assicurare che lo schedule corrente sia legale



Si noti come il 2PL prevenga l'aggiornamento fantasma senza annullare la concorrenza

Il rischio di stallo (deadlock)

T1	T2
l1(A); r1(A)	
	l2(B); r2(B)
A:=A+100;	
	B:=Bx2
w1(A)	
	w2(B)
l1(B) – bloccato!	l2(A) – bloccato!

S: l1(A) r1(A) l2(B) r2(B) w1(A) w2(B) l1(B) l2(A)

Lo scheduler riconosce che affiché S sia legale, sia T1 sia T2 devono essere bloccati, ma non può fare procedere nessuna delle due transazioni. Questa è una condizione di stallo. Torneremo più avanti sui metodi per affrontare il problema dello stallo.

Chi specifica i comandi lock/unlock

Sebbene abbiamo detto in precedenza che sono le transazioni, mediante opportuni comandi, a chiedere e rilasciare i lock, questo in realtà non è necessario.

Infatti, si può progettare lo scheduler in modo che esso riceva solo i comandi di read, write, e commit da parte delle transazioni, e sia esso ad inserire i comandi di lock-unlock rispettando le condizioni che:

- ogni transazione sia ben formata
- lo schedule sia legale (se possibile)
- ogni transazione, estesa con i comandi di lock/unlock inseriti dallo scheduler, segua il protocollo 2PL

Per questa ragione, anche in presenza di meccanismi di locking, uno schedule viene spesso denotato dalla semplice sequenza di operazioni read, write e commit. Ad esempio, lo schedule

l1(A) r1(A) l1(B) u1(A) l2(A) w2(A) r1(B) w1(B) u1(B) l2(B) u2(A) r2(B) w2(B) u2(B)

si può descrivere come:

r1(A) w2(A) r1(B) w1(B) r2(B) w2(B)

Scheduler basato su lock esclusivi e su 2PL

uno scheduler basato sul protocollo 2PL con lock esclusivi deve assicurarsi che:

- lo schedule corrente sia legale
- lo schedule corrente sia costituito da transazioni ben formate che seguono il protollo del two-phase locking

Scheduler basato su lock esclusivi e su 2PL

Tenendo conto di quanto appena detto, possiamo riassumere come si comporta lo scheduler in un sistema per la gestione della concorrenza basato su lock esclusivi e sul two-phase locking nell'analisi dello schedule corrente (ovviamente, non necessariamente completo):

1. Se arriva una richiesta di operazione da parte della transazione Ti, e tale richiesta dimostra che la transazione non è ben formata, lo scheduler annulla la transazione Ti (con tutte le conseguenze del caso)
2. Quando arriva una richiesta di lock da parte di una transazione Ti che dimostra che essa non segue il protocollo del two-phase locking, lo scheduler annulla la transazione Ti (con tutte le conseguenze del caso)
3. Quando si prefigura una richiesta di lock su A da parte di una transazione Ti mentre un'altra transazione Tj ha l'uso esclusivo di A, riconosce che lo schedule corrente diventerebbe illegale se Ti ottenesse l'uso di A, e quindi blocca Ti, facendola procedere nella sua esecuzione solo dopo che Tj avrà eseguito l'operazione di unlock su A. Se riconosce una situazione di stallo, effettua azioni per risolvere il problema.
4. Lo scheduler fa procedere lo schedule aggiornando la tabella dei lock

Si noti che (1) e (2) non possono mai verificarsi se è lo scheduler che inserisce i comandi di lock/unlock.

Confronto tra 2PL e conflict-serializzabilità

Allo scopo di confrontare 2PL e conflict-serializzabilità, utilizziamo l'osservazione precedente, e notiamo che ogni schedule S che comprenda anche operazioni di lock e unlock, può essere considerato uno schedule senza tali operazioni (semplicemente le possiamo ignorare)

Teorema Ogni schedule legale costituito da transazioni ben formate che seguono il protocollo del two-phase locking (con lock esclusivi) è conflict-serializzabile.

Dimostrazione Sia S uno schedule legale costituito da transazioni ben formate che seguono il protocollo del two-phase locking (con lock esclusivi). Per dimostrare che S è conflict-serializzabile, procediamo per induzione sul numero N di transazioni in S.

Passo base: Se N=1, S è seriale, e quindi ovviamente conflict-serializzabile.

Continua la dimostrazione

Passo induttivo: Supponiamo che S sia definito su T1,…,TN, e sia Ti la transazione che esegue la prima operazione di unlock in S. Diciamo che tale operazione è ui(X), e dimostriamo che possiamo spostare tutte le operazioni di Ti in fronte ad S senza scavalcare azioni in conflitto. Consideriamo l'azione wi(Y) di Ti (analogo ragionamento varrebbe con ri(Y)), e mostriamo che essa non può essere preceduta in S da alcuna azione in conflitto. Prendiamo infatti in considerazione ad esempio l'azione wj(Y) con j diverso da i (ma analogo ragionamento vale se consideriamo una diversa azione in conflitto con wi(Y)), e notiamo che se precedesse wi(Y) in S avremmo

… wj(Y) … uj(Y) … li(Y) … wi(Y) …

Ma siccome Ti è la transazione che esegue la prima operazione di unlock ui(X) in S, avremmo

… ui(X)… wj(Y) … uj(Y) … li(Y) … wi(Y) …

oppure

… wj(Y) …ui(X) … uj(Y) … li(Y) … wi(Y) …

In entrambi i casi, ui(X) precederebbe li(Y) in S, e quindi Ti non seguirebbe il 2PL.

Continua la dimostrazione

Possiamo quindi concludere che, spostando tutte le azioni di Ti in fronte ad S (mettendo ovviamente a posto anche le operazioni di lock e unlock) otteniamo uno schedule S' che è conflict-equivalente ad S, e che ha la forma

(azioni di Ti) (azioni delle altre transazioni di S)

La coda della sequenza S' è uno schedule legale costituito da (N-1) transazioni ben formate che seguono il protocollo del two-phase locking.

Per l'ipotesi induttiva, esso è conflict-serializzabile, e cioè esiste uno schedule seriale S'' sulle (N-1) transazioni che è conflict-equivalente ad S'. Ne segue che Ti concatenato ad S'' è uno schedule seriale che è conflict-equivalente ad S, ovvero S è conflict-serializzabile.

Cosa dice intuitivamente il teorema

Il teorema precedente dice che, dato uno schedule legale costituito da N transazioni ben formate che seguono il protocollo del two-phase locking (con locking esclusivi), noi sappiamo che il suo effetto è conflict-equivalente allo schedule seriale che ordina le transazioni di S secondo questa regola:

1. considera come prima la transazione che esegue la prima operazione di unlock in S
2. considera come seconda la transazione che, tra le (N-1) rimaste, esegue la prima operazione di unlock in S
3. …..

N-1. considera come (N-1)esima la transazione che, tra le due rimaste, esegue la prima operazione di unlock in S

N. considera come N-esima l'ultima transazione rimasta

Un'altra intuizione

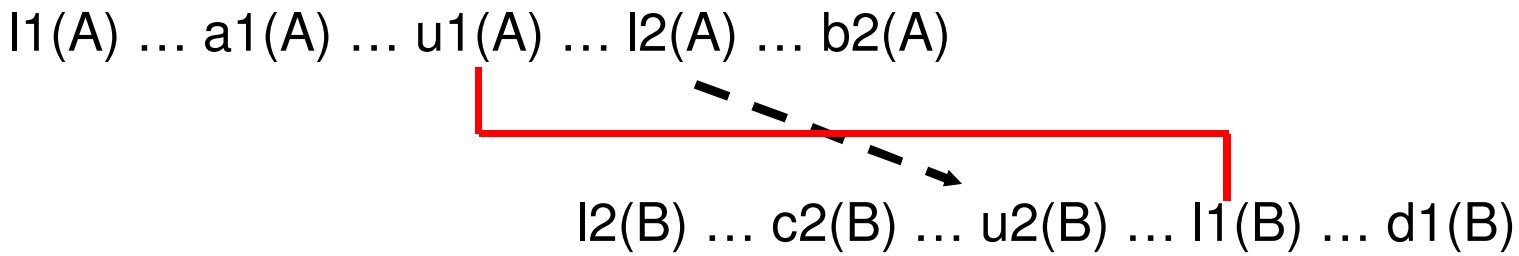
Supponiamo che S segua il protocollo 2PL. Se T1 e T2 sono in conflitto in S su un elemento A, allora esiste un'azione a1 in conflitto con un'azione b2 su A. Supponiamo che a1 venga prima di b2 in S. Affinché S segua 2PL, T1 deve avere avuto il lock su A e deve aver eseguito unlock su A prima di b2:

l1(A) … a1(A) … u1(A) … l2(A) … b2(A)

Supponiamo per assurdo che S non sia conflict-serializzabile, in particolare perché esiste un'azione c2 su B che in S viene prima di d1 su B in conflitto con c2. Ma questo vorrebbe dire che T2 ha avuto il lock su B ed eseguito unlock su B prima di d1:

l2(B) … c2(B) … u2(B) … l1(B) … d1(B)

Ma u2(B) non può venire prima di l2(A), il che implica che u1(A) viene prima di l1(B), e cioè che S non segue il 2PL:



In altre parole, il 2PL impone che se tra T1 e T2, una delle due (diciamo T1) vince sull'altra per quanto riguarda il primo conflitto (cioè ottiene il lock prima), allora T1 vince su T2 su ogni altro conflitto. Questo garantisce la conflict-serializzabilità, perché posso sempre assumere che le varie transazioni siano state serializzate secondo l'ordine dei vincitori.

Esercizi

1 – Dire se lo schedule

S: r1(x) w1(x) r2(x) w2(x) r3(y) w2(y)

può essere completato da comandi di lock/unlock in modo che tutte le transazioni siano ben formate, seguano il protocollo 2PL, e lo schedule risultante sia legale.

l1(x) r1(x) w1(x) u1(x) l2(x) r2(x) w2(x) l3(y) r3(y) u3(y) l2(y) w2(y) u2(y) u2(x)

2 - Dire se lo schedule

S: r1(x) w1(x) r2(x) w2(x) r3(y) w1(y)

può essere completato da comandi di lock/unlock in modo che tutte le transazioni siano ben formate, seguano il protocollo 2PL, e lo schedule risultante sia legale.

3 - Dire se lo schedule

S: r1(x) w1(x) r2(x) w2(x) r3(y) w1(y)

è conflict-serializzabile

Confronto tra 2PL e conflict-serializzabilità

Teorema Esistono schedule che sono conflict-serializzabili ma sono tali che lo scheduler non può trasformarli in schedule legali con comandi per la gestione di lock esclusivi, costituiti da transazioni ben formate e che seguono il protocollo del two-phase locking.

Confronto tra 2PL e conflict-serializzabilità

Dimostrazione

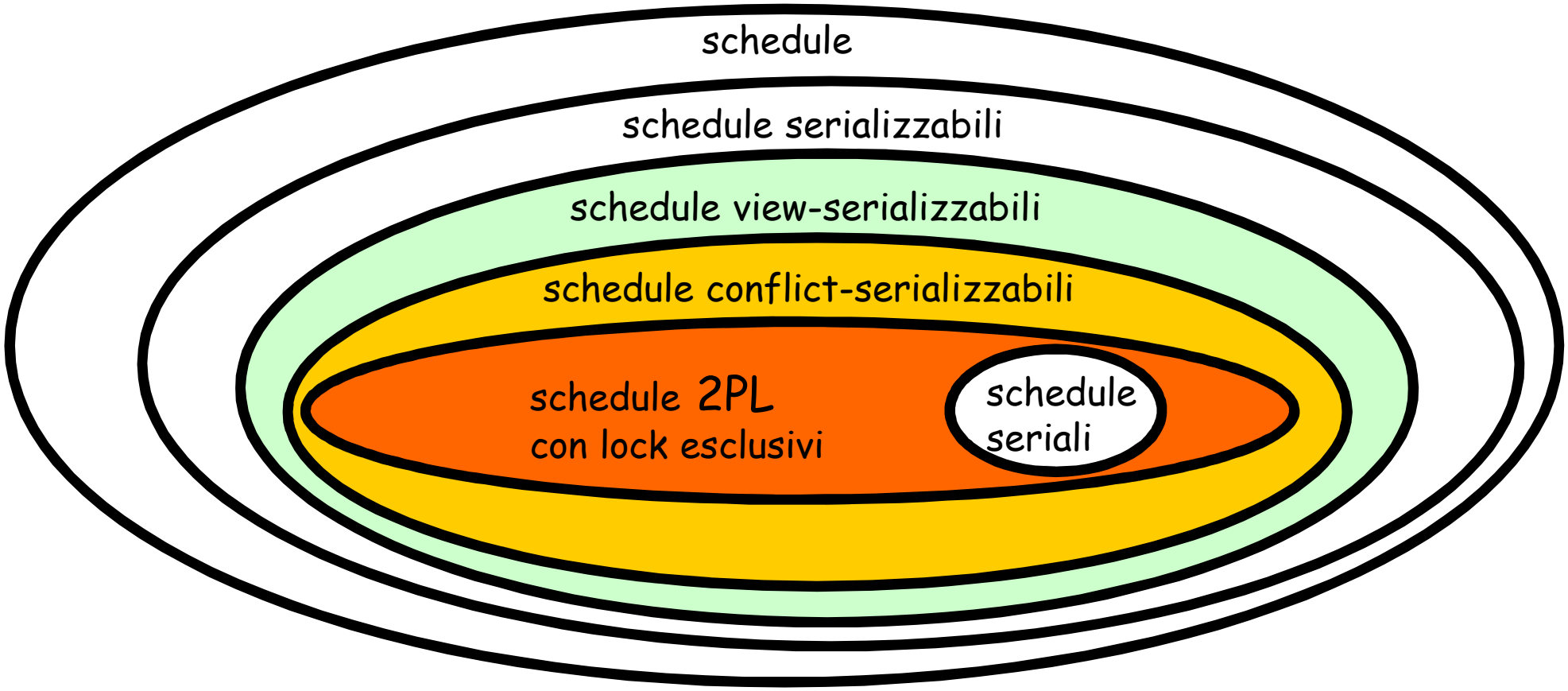
Basta considerare il seguente schedule:

S: r1(x) w1(x) r2(x) w2(x) r3(y) w1(y)

S è chiaramente conflict-serializzabile (lo schedule seriale T3,T1,T2 è conflict-equivalente ad S), ma si può dimostrare che lo scheduler non può inserire in S i comandi di lock/unlock in modo che tutte le transazioni siano ben formate, seguano il protocollo 2PL, e lo schedule risultante sia legale. Infatti, lo scheduler dovrebbe inserire il comando u1(x) prima dell'operazione r2(x), perchè altrimenti r2(x) non disporrebbe dell'uso esclusivo di x, e dovrebbe inserire il comando l1(y) dopo r3(y), perché altrimenti r3(y) non disporrebbe dell'uso esclusivo di y. Ne segue che lo scheduler sarebbe costretto ad inserire i comandi di lock/unlock in modo che il protocollo 2PL non sia rispettato.

Confronto tra conflict-serializzabilità e 2PL

Indichiamo con “schedule 2PL con lock esclusivi” la classe degli schedule con comandi per la gestione di lock esclusivi, costituiti da transazioni ben formate e che seguono il protocollo del two-phase locking. Graficamente, il confronto può essere riassunto nel seguente modo:



Lock condivisi (shared locks)

Con i lock esclusivi, una transazione che legge A deve effettuare unlock per permettere ad un'altra di leggere A:

S: ... l1(A) r1(A) u1(A) … l2(A) r2(A) u2(A) …

In realtà, le due letture protette da lock non creano alcun conflitto. Introduciamo quindi un nuovo tipo di lock: il lock condiviso (shared lock). Denotiamo con sli(A) il comando della transazione Ti che chiede il lock condiviso su A.

Con i lock condivisi, nell'esempio precedente, otteniamo:

S: ... sl1(A) r1(A) sl2(A) r2(A) …. u1(A) u2(A)

Le primitive di lock ora diventano:

xli(A): lock in modalità esclusiva (detto anche lock in scrittura)
sli(A): lock in modalità condivisa (detto anche lock in lettura)
ui(A): unlock

Transazioni ben formate con lock condivisi

Nell'uso dei lock esclusivi e condivisi, dobbiamo rispettare la seguente regola.

Regola 1: Diciamo che una transazione Ti è **ben formata** se

- ogni lettura ri(A) è preceduta o da sli(A) o da xli(A) senza un ui(A) in mezzo,
- ogni scrittura wi(A) è preceduta da xli(A) senza un ui(A) in mezzo
- ogni operazioni di lock (sl o xl) su A da parte di Ti è seguita da unlock su A da parte di Ti

Si noti che accettiamo che una transazione Ti prima esegua sli(A), presumibilmente per leggere A con una ri(A), e poi esegua xli(A), presumibilmente per scrivere con una wi(A). Il passaggio da un lock condiviso ad un lock esclusivo sullo stesso elemento da parte di una transazione si chiama "lock upgrade".

Schedule legali con lock condivisi

Nell'uso dei lock esclusivi e condivisi, dobbiamo rispettare anche la seguente regola

Regola 2: Diciamo che uno schedule S è **legale** se

- un xli(A) non è seguito da un xlj(A) o da un slj(A) (con j diverso da i) senza che ci sia in mezzo ui(A)
- un sli(A) non è seguito da un xlj(A) (con j diverso da i) senza che ci sia in mezzo ui(A)

Two-phase locking (con lock condivisi)

Con l'uso dei lock esclusivi e condivisi, la regola del two-phase locking diventa la seguente.

Uno schedule (con lock esclusivi e condivisi) segue il protocollo del two-phase locking se in ogni transazione Ti presente nello schedule tutte le operazioni di lock (condiviso ed esclusivo) di Ti precedono tutte le operazioni di unlock di Ti.

In altre parole, nessuna azione sli(X) o xli(X) può essere preceduta da ui(Y) nello schedule.

Politica dello scheduler per gestire i lock

- Lo scheduler basato su 2PL e su lock esclusivi e condivisi si comporta in modo analogo allo scheduler basato su 2PL e su lock esclusivi, tenendo conto però della complicazione data dai lock condivisi.
- La politica adottata dallo scheduler per gestire i lock richiesti dalle transazioni si può riassumere nella cosiddetta "matrice di compatibilità", che dice quando lo scheduler soddisfa la richiesta di lock su un elemento A (da parte della transazione Tj), richiesta rappresentata in colonna, a fronte di un lock già ottenuto da parte di Ti su A del tipo specificato nella riga
- Nella matrice, "S" sta per lock condiviso e "X" sta per lock esclusivo, "yes" sta per richiesta soddisfatta e "no" sta per richiesta rifiutata.

Tipo di lock già ottenuto da Ti su A	Nuovo lock richiesto da Tj ≠ Ti su A: S	Nuovo lock richiesto da Tj ≠ Ti su A: X
S	yes	no
X	no	no

Politica dello scheduler per gestire i lock

- Si noti che il problema di inserire automaticamente i comandi di lock-unlock da parte dello scheduler è ora più complesso rispetto al caso di soli lock esclusivi

- Inoltre, è anche più complessa la gestione del comando di unlock. In particolare, è importante osservare che, al momento di unlock su A da parte di Ti, ci possono essere diverse transazioni in attesa di un lock (esclusivo o condiviso) su A, e lo scheduler deve scegliere la transazione da soddisfare. Sono possibili diversi metodi:
  - First-come-first-served, ovvero gestione a coda
  - Favorisci chi chiede lock condiviso
  - Favorisci chi vuole passare da lock condiviso a lock esclusivo

  Il primo metodo è il più democratico ed il più usato, anche perché evita il fenomeno dello "starvation", cioè la situazione in cui la richiesta di una transazione non viene mai soddisfatta

Proprietà del two-phase locking (con lock condivisi)

Le proprietà del two-phase locking con lock condivisi ed esclusivi rimangono immutate rispetto al caso di soli lock esclusivi:

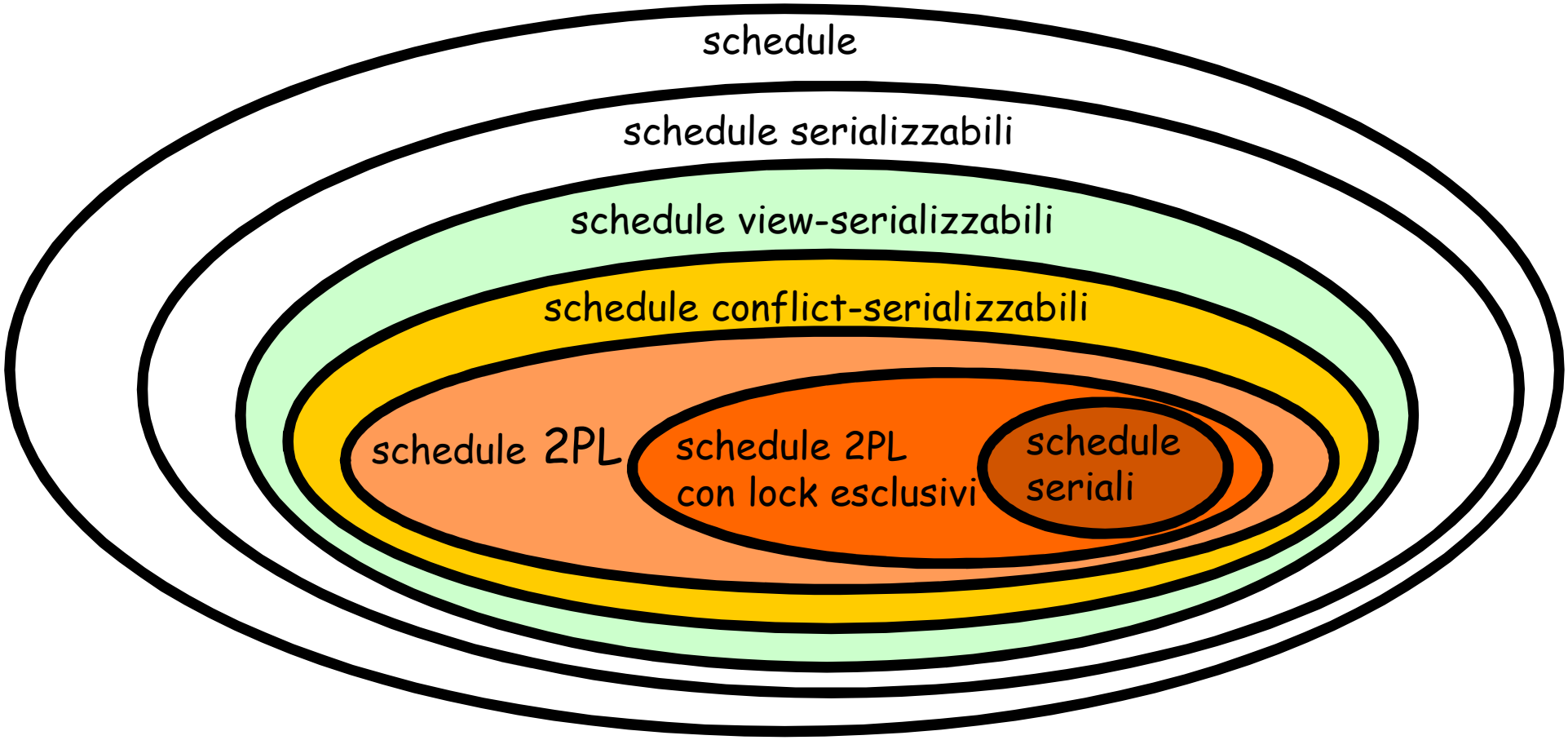
- **Teorema** Ogni schedule legale costituito da transazioni ben formate che seguono il protocollo del two-phase locking (con lock esclusivi e condivisi) è conflict-serializzabile.
- **Teorema** Esistono schedule che sono conflict-serializzabili ma sono tali che lo scheduler non può trasformarli in schedule legali con comandi per la gestione di lock esclusivi e condivisi, costituiti da transazioni ben formate e che seguono il protocollo del two-phase locking.
- Anche con i lock condivisi esiste il pericolo di stallo, come ad esempio in:

sl1(A) sl2(A) xl1(A) xl2(A)

Inoltre, è evidente che gli schedule legali costituiti da transazioni ben formate che seguono il protocollo del two-phase locking con lock esclusivi e condivisi sono un soprainsieme degli schedule 2PL permessi con i soli lock esclusivi

Confronto tra conflict-serializzabilità e 2PL

Indichiamo con “schedule 2PL” la classe degli schedule legali con comandi per la gestione di lock esclusivi e condivisi, costituiti da transazioni ben formate e che seguono il protocollo del two-phase locking. Graficamente, il confronto può essere riassunto nel seguente modo:



La gestione dello stallo

- Ricordiamo che lo stallo (deadlock) è una condizione in cui due transazioni T1 e T2 si trovano ad occupare due risorse A e B e ciascuna chiede il lock esclusivo sulla risorsa occupata dall'altra. Ciò provoca un blocco delle due transazioni, che non possono procedere
- La probabilità di deadlock cresce in modo lineare con il numero di transazioni ed in modo quadratico con il numero di richieste di lock da parte di ogni transazione

T1	T2
xl1(A); r1(A)	
	xl2(B); r2(B)
A:=A+100;	
	B:=Bx2
w1(A)	
	w2(B)
sl1(B) – bloccato!	
	sl2(A) – bloccato!

Tecniche utilizzate per gestire lo stallo

1. Uso del timeout
2. Riconoscimento dello stallo, e sua soluzione
3. Prevenzione dello stallo

Uso del timeout

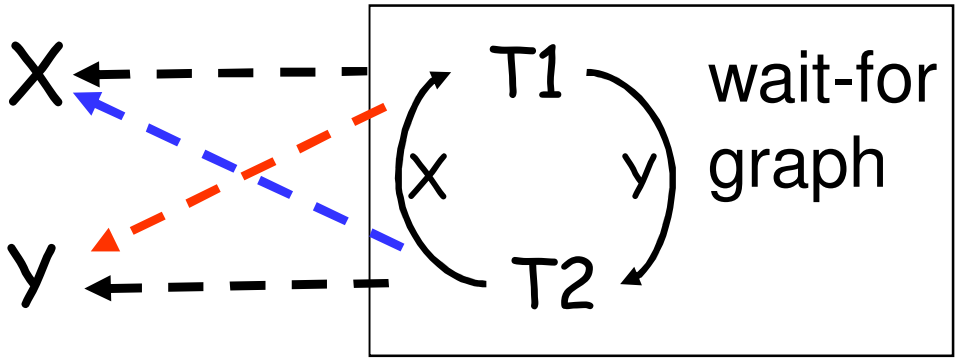
- Viene fissato un tempo *t* di timeout oltre il quale le transazioni in attesa di lock vengono annullate (uccise)

- Vantaggi
  - Molto semplice

- Svantaggi
  - *t* alto risolve tardi il problema
  - *t* basso uccide troppe transazioni

Riconoscimento dello stallo

- Costruzione incrementale del grafo di attesa (wait-for graph): i nodi sono le transazioni, e l'arco da Ti a Tj in questo grafo significa che Ti sta aspettando che Tj liberi una risorsa
- Quando si genera un ciclo, lo scheduler risolve lo stallo scegliendo la transazione vittima (ad esempio quella che ha effettuato meno lavoro – ma attenzione al blocco individuale, cioè il rischio di uccidere ripetutamente la stessa transazione) ed eseguendo il suo rollback (con tutte le conseguenze del caso)
- Esempio: $sl_1(X)\ sl_2(Y)\ r_1(X)\ r_2(Y)\ xl_1(Y)\ xl_2(X)$



Prevenzione dello stallo: wait-die

Ad ogni transazione Ti è assegnata una priorità pr(Ti) (ad esempio, un numero che indica quanto è vecchia) al momento in cui arriva allo scheduler, in modo che diverse transazioni abbiano diverse priorità

Si utlizza la seguente regola:

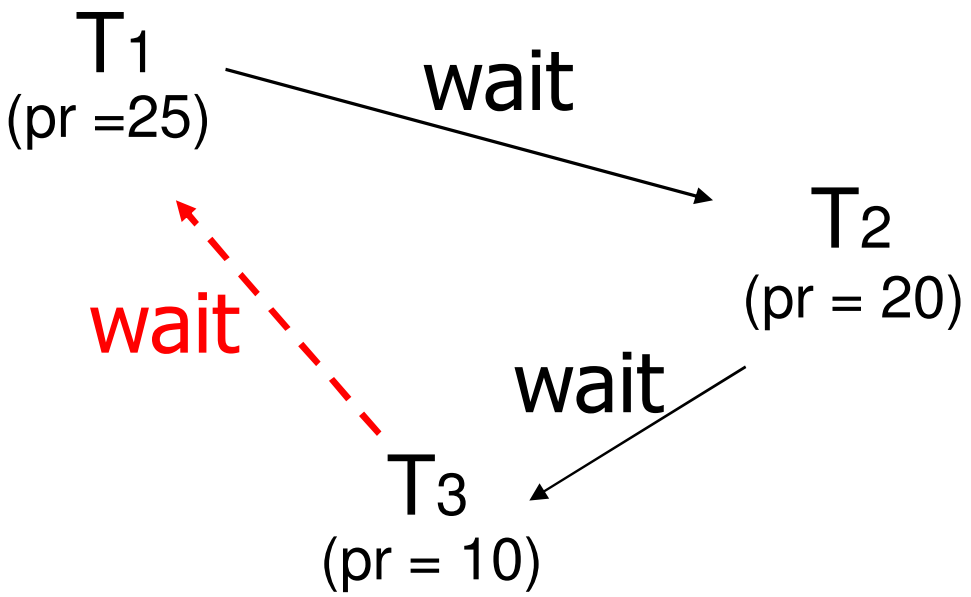
in caso di conflitto su un lock, T_i può attendere T_j solo se ha priorità maggiore, cioè se pr(Ti) > pr(Tj), altrimenti Ti viene uccisa e subisce il rollback

In pratica, all'arrivo di un nuovo arco Ti → Tj nel wait-for graph:

- se pr(Ti) > pr(Tj): nessuna azione di prevenzione
- se pr(Ti) < pr(Tj): Ti subisce il rollback

Esempio di wait-die

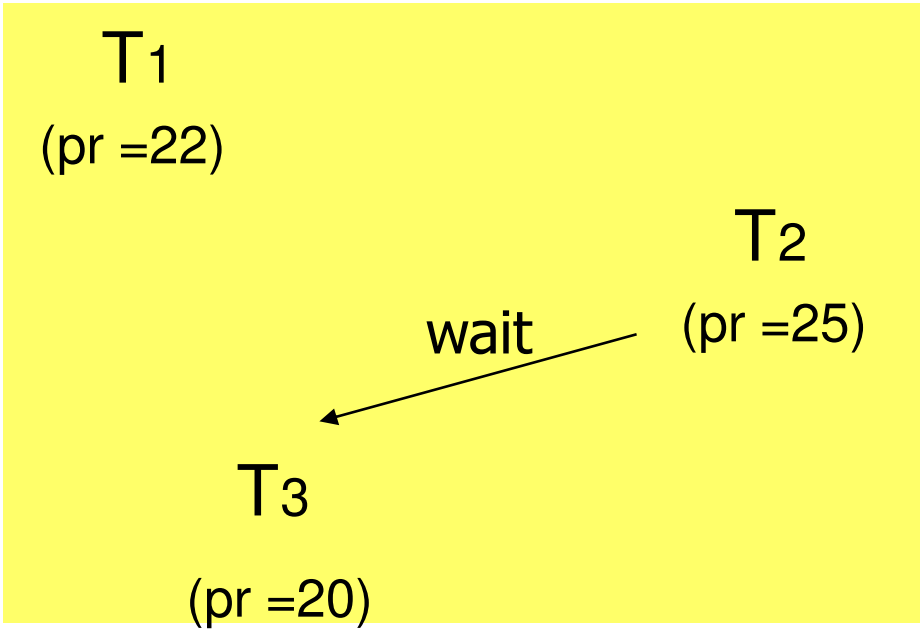
```
...
xl1(Y)    T1 accede a Y
xl3(X)    T3 accede a X
xl2(X)    T2 attende T3
xl1(X)    T1 attende T2
xl3(Y)    T3 uccisa
```



T3 uccisa

Altro esempio di wait-die

...
xl3(X) T3 accede a X
xl2(X) T2 attende T3
xl1(X) T1 attende da T3 (e da T2?)



Si noti che pr(T1) è maggiore di pr(T3), ma minore di pr(T2). Se mettessimo T1 in attesa di T3, scavalcheremmo T2, e dovremmo mettere T2 in attesa di T1, col rischio di starvation. Siccome T1 non può essere messo in attesa di T2, il metodo più semplice consiste nell'uccidere T1.

1.5
Recuperabilità delle transazioni

Il problema del rollback

- Finora abbiamo svolto il nostro studio assumendo che tutte le transazioni eseguissero il commit. Ora rilasciamo questa assunzione, e accettiamo che una transazione esegua il rollback

- La prima osservazione è che, in questa nuova situazione, la nozione di serializzabilità finora studiata non è più sufficiente a garantire le proprietà ACID delle transazioni

- Il problema è testimoniato dell'esistenza di una nuova anomalia, chiamata "della lettura sporca"

Nuova anomalia: lettura sporca (dirty read)

Sono date due transazioni T1 e T2 identiche:

READ(A, x), x:=x+1, WRITE(A, x)

Si consideri il seguente schedule (si noti che T1 esegue il rollback):

T_1	T_2
begin	begin
READ(A,x)	
x := x+1	
WRITE(A,x)	
	READ(A,x)
	x := x+1
rollback	
	WRITE(A,x)
	commit

Il problema è che T2 legge un valore scritto da T1 prima che T1 decida per il commit o il rollback. Di conseguenza, T2 legge un valore "sporco", che viene poi rimosso dall'azione di rollback. Il calcolo di A da parte di T2 si basa quindi su un input errato.

Commit o rollback?

Si noti che al termine di ogni transazione:

- Se la transazione ha effettuato commit:
  - il sistema deve garantire che la transazione si concluda a buon fine, con i suoi effetti permanentemente registrati nella base di dati
- Se la transazione ha effettuato rollback:
  - il sistema deve garantire che la transazione non lasci alcun effetto, nè diretto nè indiretto

Cascading rollback

Si noti che ci sono casi in cui il rollback di una transazione Ti può provocare il rollback di altre transazioni, a cascata. In particolare:

- se un'altra transazione Tj ha letto da Ti, bisogna uccidere (rollback) Tj
- se un'altra transazione Th ha letto da Tj, bisogna uccidere (rollback) Th
- e così via, ricorsivamente

Il fenomeno è chiamato cascading rollback

- il cascading rollback, se necessario, è gestito dal Recovery Manager
- ma l'ideale è trovare dei metodi che evitino il cascading rollback

Schedule recuperabili

Se in uno schedule, una transazione Ti che ha letto da Tj esegue il commit prima di Tj, il rischio è che Tj faccia successivamente il rollback, con il risultato che Ti può lasciare un effetto sulla base di dati che dipende da un'operazione (di Ti) poi annullata, e che quindi dovrebbe essere considerata "mai effettivamente eseguita". Per cogliere questo concetto, si dice che Ti non è recuperabile.

Uno schedule S è **recuperabile** (recoverable) se nessuna transazione in S esegue il commit prima che tutte le altre transazioni dalle quali essa ha letto abbiano eseguito il commit

Esempio di schedule recuperabile:

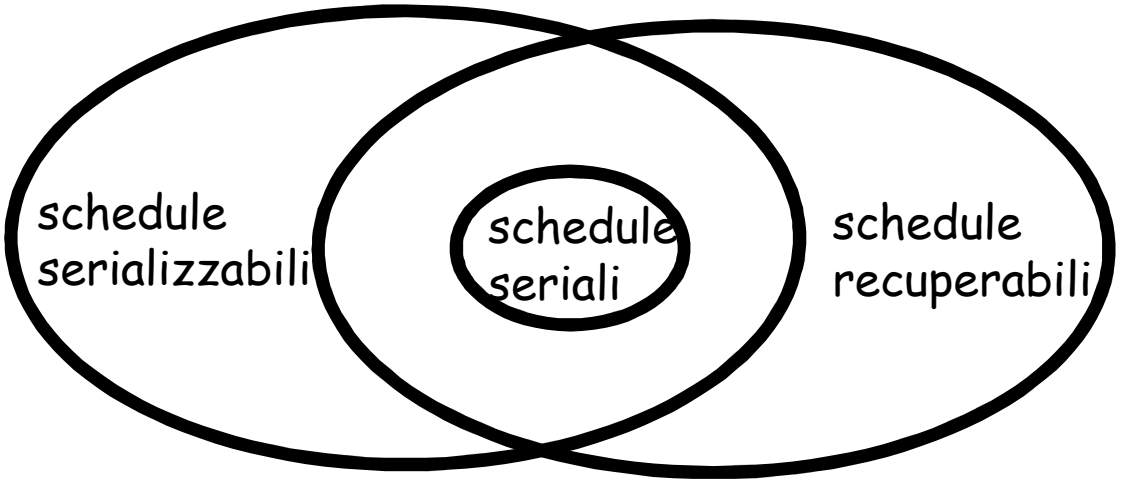
S: w1(A) w1(B) w2(A) r2(B) c1 c2

Esempio di schedule non recuperabile:

S: w1(A) w1(B) w2(A) r2(B) r3(A) c1 c3 c2

Serializzabilità e recuperabilità

I due concetti di serializzabilità e recuperabilità sono ortogonali, nel senso che ci sono schedule recuperabili che non sono serializzabili, e schedule serializzabili che non sono recuperabili. Ovviamente, ogni schedule seriale è recuperabile



Ad esempio, lo schedule

S1: w2(A) w1(B) w1(A) r2(B) c1 c2

è recuperabile ma non serializzabile (non è view-serializzabile), mentre lo schedule

S2: w1(A) w1(B) w2(A) r2(B) c2 c1

è serializzabile (in particolare, conflict-serializzabile) ma non recuperabile

Recuperabilità e cascading rollback

Gli schedule recuperabili non sventano il pericolo di cascading rollback.

Ad esempio nello schedule recuperabile

S: w2(A) w1(B) w1(A) r2(B)

se T1 esegue il rollback, T2 deve essere uccisa

Per evitare il cascading rollback occorre imporre una condizione più forte della recuperabilità:

Uno schedule evita il cascading rollback (ovvero è **ACR**, Avoid Cascading Rollback) se ogni transazione legge solo valori scritti da transazioni che hanno già eseguito il commit

Ad esempio, è ACR lo schedule

S: w2(A) w1(B) w1(A) c1 r2(B) c2

In altre parole, uno schedule ACR impedisce l'anomalia dirty data.

Riassumendo

- S è recuperabile se nessuna transazione in S esegue il commit prima che tutte le altre transazioni dalle quali essa ha letto abbiano eseguito il commit

  Esempio:

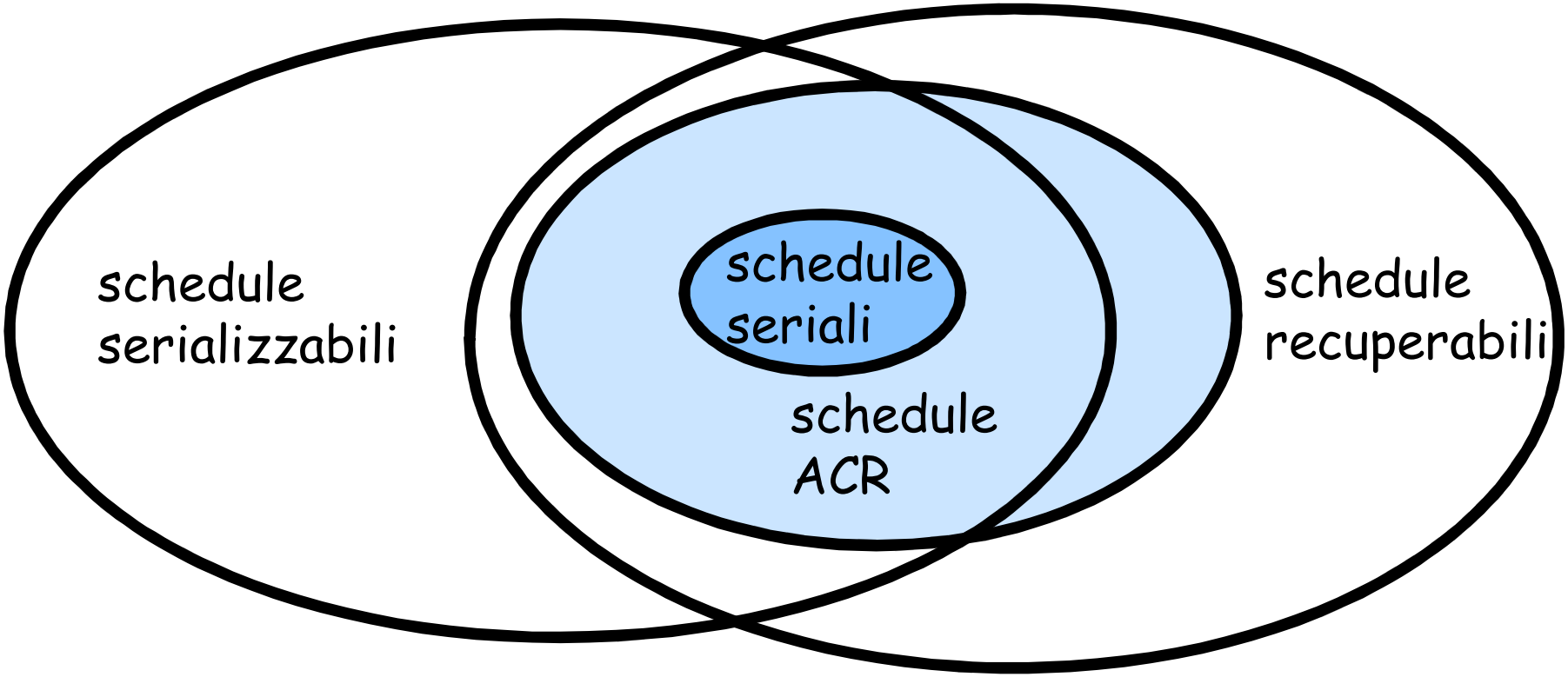
  w1(A) w1(B) w2(A) r2(B) c1 c2

- S è ACR, cioé evita il cascading rollback, se ogni transazione *legge* solo valori scritti da transazioni che hanno già eseguito commit

  Esempio:

  w1(A) w1(B) w2(A) c1 r2(B) c2

Confronto tra recuperabilità e ACR



Si noti che, analogamente al caso degli schedule recuperabili, nemmeno gli schedule ACR sono necessariamente serializzabili. Ovviamente, ogni schedule ACR è recuperabile, ed ogni schedule seriale è ACR

Recuperabilità e schedule 2PL

- Nella discussione sulla recuperabilità ci siamo fino ad ora occupati della relazione tra:
  - operazioni di lettura
  - operazioni di rollback
  - operazioni di commit

- Dobbiamo ora capire come, per garantire la recuperabilità, si debba modificare la gestione dei lock, e dunque la struttura del protocollo 2PL

Schedule stretti

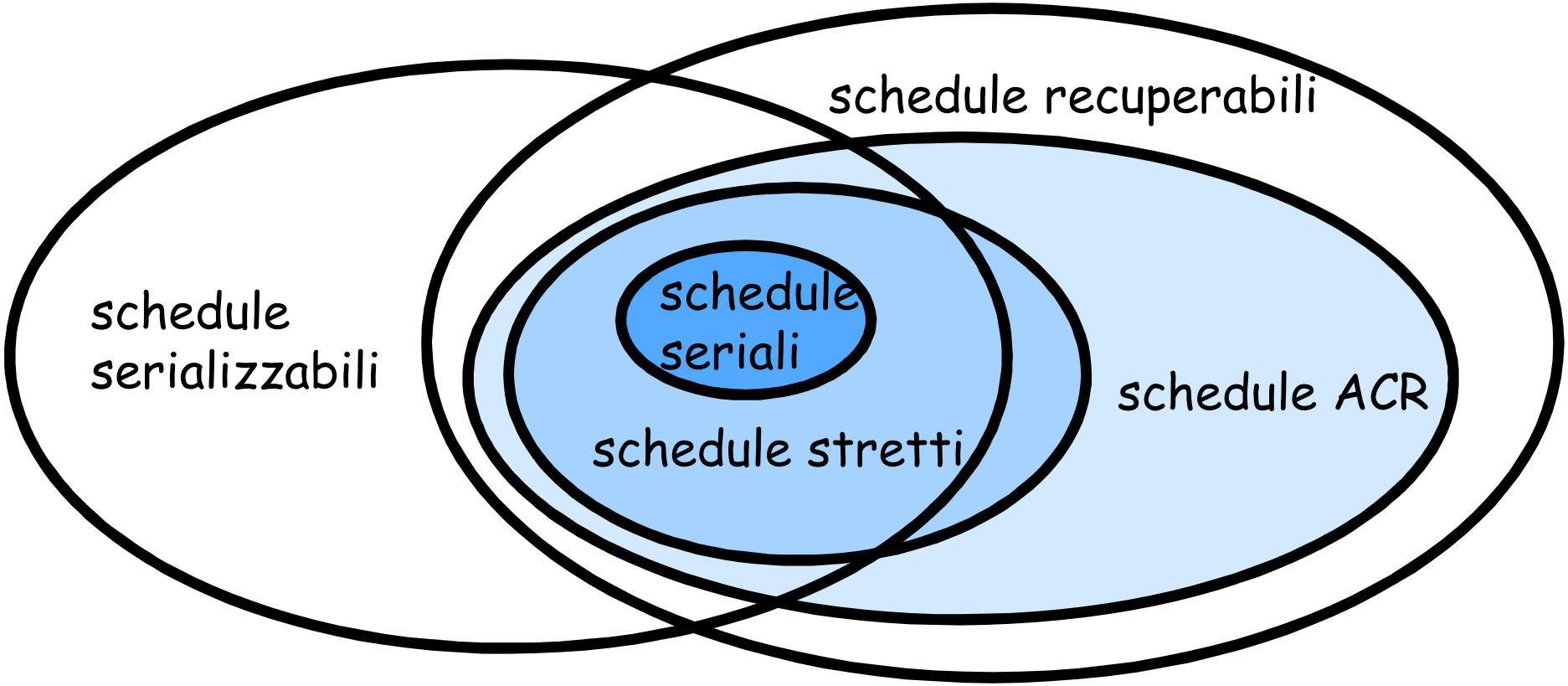
Diciamo che in uno schedule S una transazione Ti scrive su Tj se esiste in S un wj(A) seguito da wi(A), e tra queste operazioni non c'è alcuna operazione di scrittura su A

Diciamo che uno schedule è **stretto** se ogni transazione *legge* solo valori scritti da transazioni che hanno già eseguito commit, e *scrive* solo su transazioni che hanno già eseguito commit

Si può verificare facilmente che ogni schedule stretto è ACR, e quindi recuperabile.

L'importanza di schedule stretti sarà completamente chiara quando parleremo di recovery. Per il momento, osserviamo che per uno schedule stretto, quando una transazione Ti viene abortita, è semplice stabilire quali sono i valori da ripristinare nella base di dati per riflettere l'annullamento delle operazioni di Ti

Confronto tra schedule stretti e ACR



Ovviamente, ogni schedule seriale è stretto, ed ogni schedule stretto è ACR, e quindi recuperabile. Non tutti gli schedule ACR sono stretti.

Two-phase locking stretto (strict 2PL)

Uno schedule S segue il protocollo del **2PL stretto** (chiameremo un tale schedule **strict 2PL**) se segue il protocollo 2PL, ed inoltre i lock di ogni transazione Ti vengono mantenuti fino al commit o l'abort di Ti

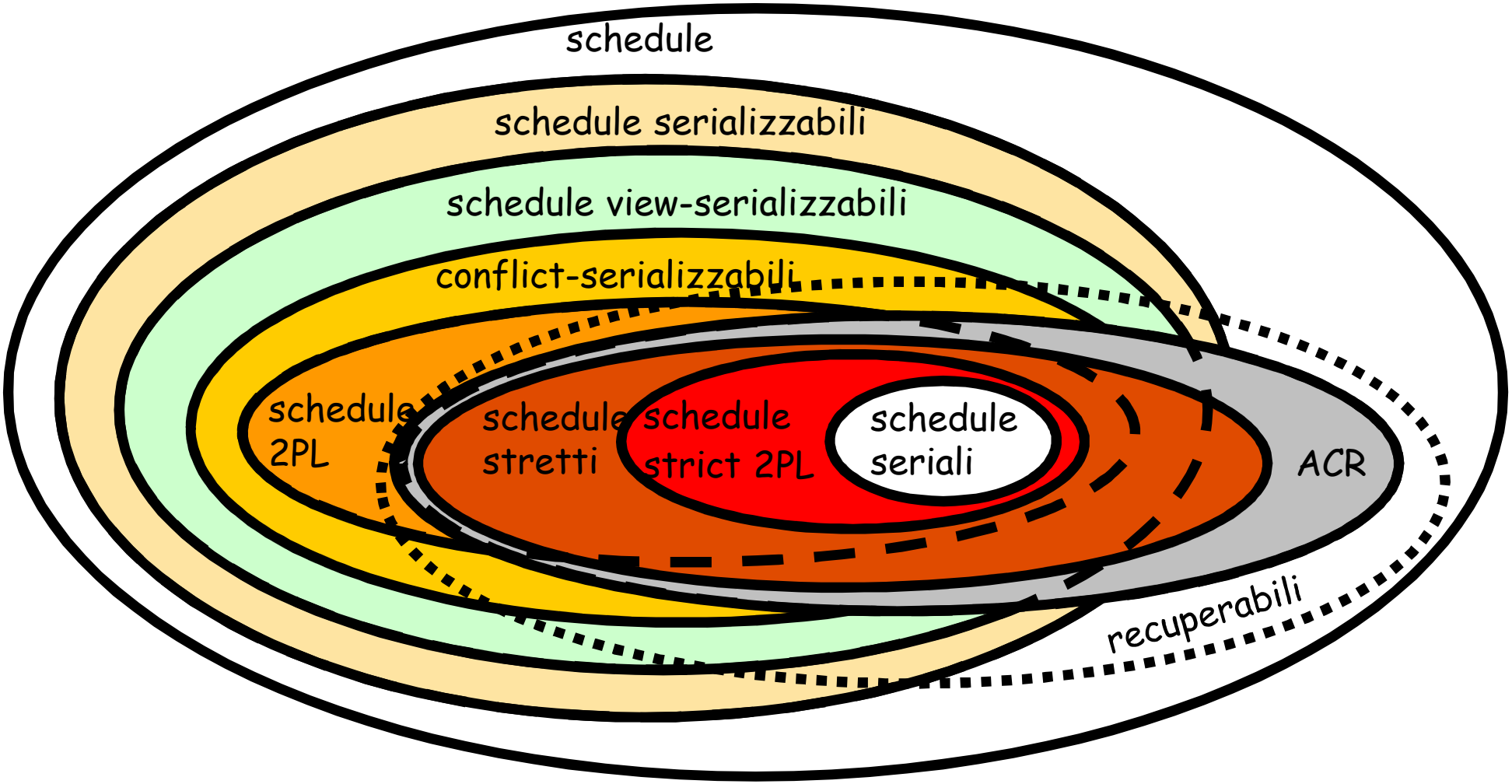
Tj	Ti
wj(A)	
.....	
commit	
uj(A)	
	ri(A)

Proprietà del 2PL stretto

- Ogni schedule strict 2PL evita il cascading rollback
  - infatti, una transazione T2 non può leggere un valore di un elemento X scritto da T1 fin quando T1 non rilascia il lock su X, il chè avviene dopo che T1 ha eseguito il commit
- Ogni schedule strict 2PL è serializzabile
  - si può dimostare che ogni schedule che segue il protocollo del 2PL stretto è conflict-equivalente allo schedule seriale in cui si ignorano le azioni delle transazioni che fanno rollback in S, ed in cui l'ordine delle transazioni è quella dettata dall'ordine dei comandi di commit (prima la transazione che per prima effettua il commit, poi la transazione che effettua per seconda il commit, e così via)

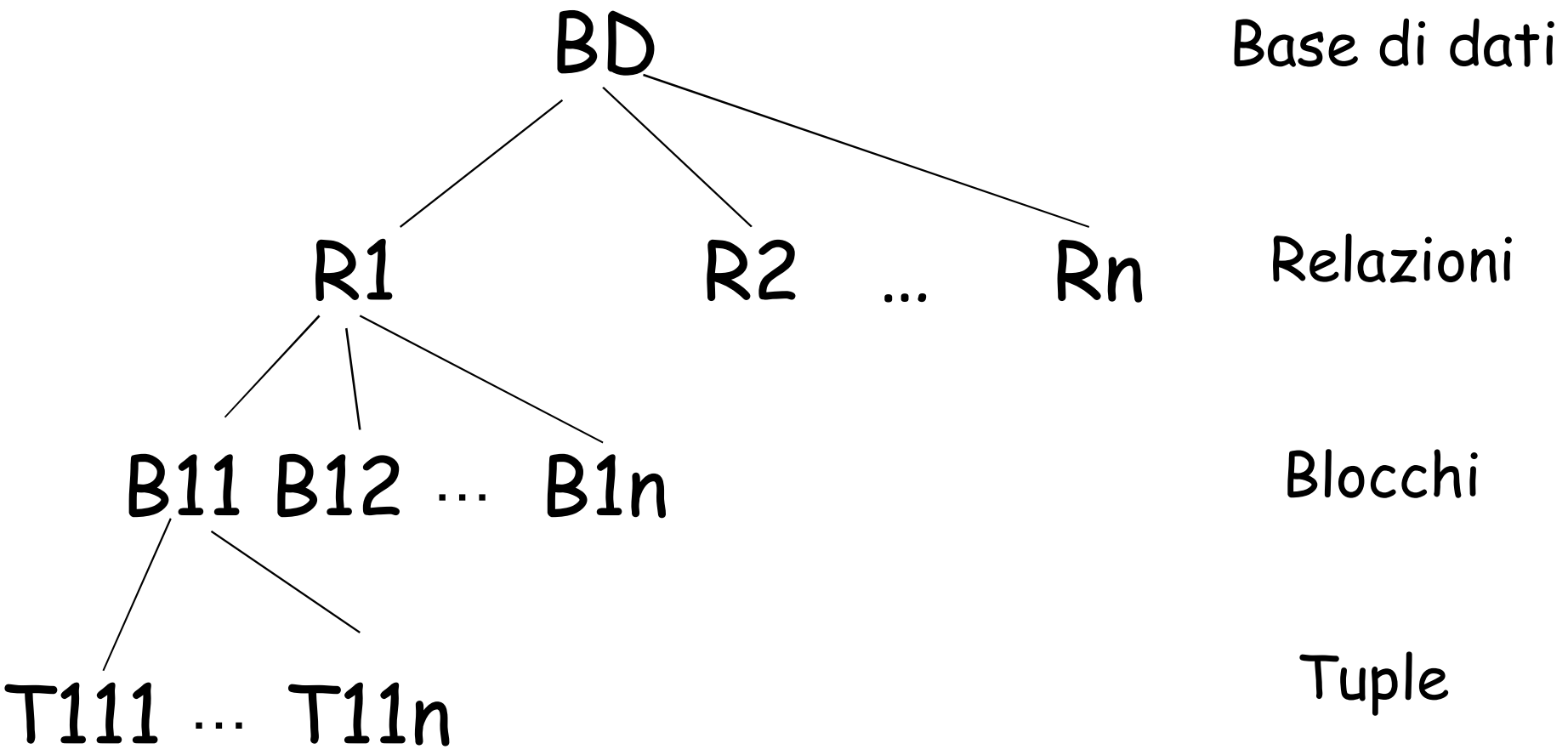
Confronto generale

Indichiamo con “schedule strict 2PL” la classe degli schedule legali con comandi per la gestione di lock esclusivi e condivisi, costituiti da transazioni ben formate e che seguono il protocollo del two-phase locking stretto.



Granularità del locking – locking gerarchico

In molti DBMS è possibile specificare i lock a diverse granularità:



Opzioni per la granularità

- Utilizzando lock su oggetti grandi (ad esempio relazioni)
  - sono sufficienti pochi lock
  - la concorrenza è ridotta – le risorse restano bloccate con alta probabilità

- Utilizzando lock su oggetti piccoli (esempio record, campi)
  - occorrono più lock
  - la concorrenza aumenta

Gerarchie di oggetti e di lock

- Consideriamo gerarchie di oggetti strutturate ad albero
  - relazione → blocco → tupla
- I lock vengono richiesti dalla radice verso i discendenti
- I lock vengono rilasciati a partire dal nodo con lock e proseguendo verso la radice

Nuovi tipi di lock

- Come prima: shared and exclusive
  - X: exclusive lock
  - S: shared lock

- Tre nuovi tipi (si assume di lavorare su un nodo corrente):
  - ISL: intention shared lock – esprime l'intenzione di bloccare in modo condiviso almeno uno dei nodi che discendono dal nodo corrente
  - IXL: intention exclusive lock - esprime l'intenzione di bloccare in modo esclusivo almeno uno dei nodi che discendono dal nodo corrente
  - SIXL: shared intention-exclusive lock – blocca in modo condiviso il nodo corrente ed esprime l'intenzione di bloccare in modo esclusivo almeno uno dei nodi che discendono dal nodo corrente

Regole per i lock gerarchici

- Per richiedere un lock SL o ISL su un nodo, la transazione deve già avere un lock ISL sul nodo genitore
- Per richiedere un lock IXL, XL, o SIXL su un nodo, la transazione deve già avere un lock SIXL o IXL sul nodo genitore
- Ad esempio, per bloccare una tupla di Ri in scrittura:
  - IXL su DB
  - IXL su Ri
  - IXL sul blocco opportuno
  - XL sulla tupla

Matrice di compatibilità per lock gerarchici

Richiesta \ Stato risorsa	ISL	IXL	SL	SIXL	XL
ISL	Yes	Yes	Yes	Yes	No
IXL	Yes	Yes	No	No	No
SL	Yes	No	Yes	No	No
SIXL	Yes	No	No	No	No
XL	No	No	No	No	No

1.6
Gestione della concorrenza mediante timestamp

Altri metodi per il controllo di concorrenza

- I metodi basati su lock sono pessimistici, nel senso che prevengono comportamenti non serializzabili
- Altri metodi sono invece ottimistici, nel senso che portano ad eseguire le transazioni, salvo nel corso della esecuzione verificare qualche condizione che segnala comportamenti non serializzabili
- Il **metodo dei timestamp**, che adesso analizziamo, appartiene a questa categoria

Concorrenza basata su timestamp

- Ad ogni transazione T viene assegnato un **timestamp** ts(T) unico al momento del suo inizio (arrivo allo scheduler). Nel seguito, per semplicità di notazione, assumiamo che il timestamp di ogni transazione coincida con il suo pedice, cioè ts(Ti)=i.
- **I timestamp definiscono implicitamente un ordine totale sulle transazioni**, nel senso che le transazioni possono essere considerate ordinate secondo l'ordine di arrivo, e quindi secondo valori crescenti dei timestamp.
- **Ogni schedule che rispetta questo ordine è conflict-serializzabile**, perché è equivalente allo schedule seriale corrispondente all'ordinamento dei timestamp.
- Lo scheduler basato sui timestamp adotta inoltre dei meccanismi che assicurano che **ogni schedule eviti il cascading rollback**.
- L'uso dei timestamp consente di evitare del tutto l'uso di lock, e questo elimina anche il problema dei deadlock (in realtà, il deadlock può ancora avvenire, anche se è molto improbabile).

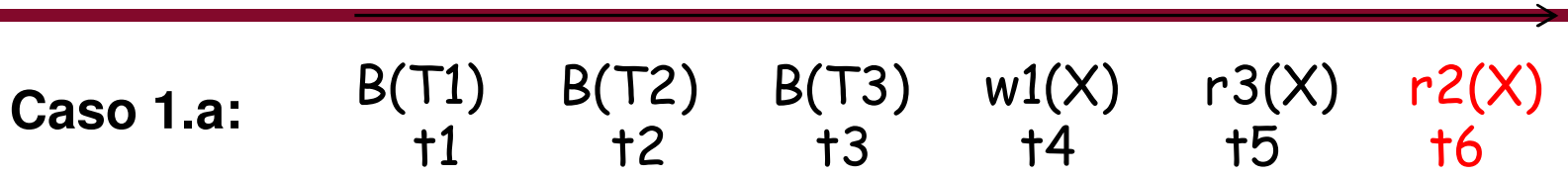
Uso dei timestamp

- Le transazioni eseguono le loro azioni liberamente, senza la necessità di seguire protocolli
- Ad ogni operazione di lettura o scrittura, lo scheduler controlla che i timestamp delle transazioni coinvolte non violino la serializzabilità dello schedule rispetto all'ordine totale indotto dai timestamp
- Ad ogni elemento X vengono associati due timestamp e un bit:
  - **rts**(X): uguale al timestamp più alto tra quelli delle transazioni che hanno letto X
  - **wts**(X): uguale al timestamp più alto tra quelli delle transazioni che hanno scritto X (che come vedremo coincide con il timestamp della transazione che ha effettuato l'ultima scrittura su X)
  - **cb**(X): un bit (chiamato il **commit-bit**), che è pari a true se la transazione che ha scritto per ultima X ha eseguito il commit, ed è false altrimenti (inizialmente è true)

Regole di timestamp

- Idea fondamentale:
  - le operazioni di una transazione T che avvengono nello schedule vanno pensate come se T fosse eseguita logicamente in modo “istantaneo”
  - il tempo di esecuzione logico di un’operazione in T è quello del timestamp di T, ovvero ts(T)
  - il commit-bit serve ad evitare la lettura sporca
- Quindi abbiamo un **tempo fisico** ed un **tempo logico** di esecuzione, e rts(X), wts(X) ci indicano il timestamp della transazione che ha effettuato l’ultima lettura/scrittura su X rispetto al tempo logico
- Un’operazione di T eseguita al tempo fisico t, è accettata se il suo ordine temporale fisico è compatibile con il tempo logico ts(T)
- Questo principio di compatibilità è messo in pratica dallo scheduler, che controlla le regole di validazione delle operazioni in tempo reale, e prende le opportune decisioni
- Come abbiamo detto, noi assumiamo che il timestamp di ogni transazione coincida con il suo pedice, cioè ts(Ti)=i. Nel seguito, t1,…,tn denotano i tempi fisici.

Regole di timestamp – caso 1a (read ok)



Consideriamo r2(X) rispetto all'ultima scrittura w1(X) su X:

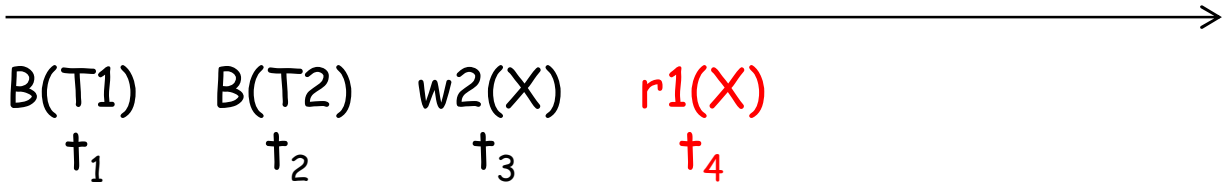
- il tempo fisico di r2(X) è t6, che è maggiore del tempo fisico di w1(X), cioè t4
- il tempo logico di r2(X) è ts(T2) che è maggiore del tempo logico di w1(X), cioè wts(X) = ts(T1)

Concludiamo che non c'è incompatibilità (**read ok**), e si procede così:

1. se cb(X) è true, allora
   - in generale, a fronte di una lettura di T su X, rts(X) deve essere posto uguale al massimo tra rts(X) e ts(T) – nell'esempio, sebbene r2(X) sia fisicamente successivo all'ultima lettura r3(X) su X, essa è logicamente precedente a r3(X), e quindi, se cb(X) fosse uguale a true, rts(X) rimarrebbe ts(T3)
   - l'azione r2(X) di T2 viene eseguita, e si continua
2. se cb(X) è false (come nell'esempio), allora T2 viene messa in attesa del commit che fa scattare cb(X) a true, o del rollback della transazione che ha scritto per ultima X

Regole di timestamp – caso 1b (read too late)

Caso 1.b:



Consideriamo r1(X) rispetto all'ultima scrittura w2(X) su X:

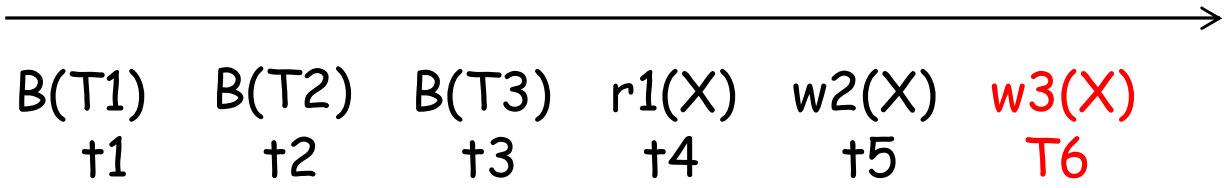
- il tempo fisico di r1(X) è t4, che è maggiore del tempo fisico di w2(X), cioè t3
- il tempo logico di r1(X) è ts(T1), che è minore del tempo logico di w2(X), cioè wts(X) = ts(T2)

Concludiamo che c'è incompatibilità: in altri termini, la lettura di X da parte di T1 avviene "troppo tardi" (**read too late**)

Si procede così: l'azione r1(X) di T1 non può essere eseguita, T1 subisce il rollback, e viene fatta ricominciare con un nuovo timestamp.

Regole di timestamp – caso 2a (write ok)

Caso 2.a:



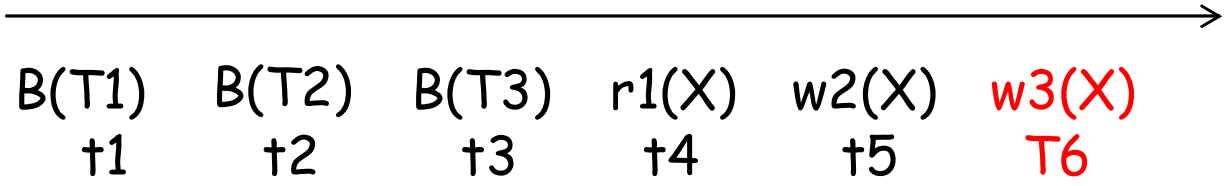
Consideriamo w3(X) rispetto all'ultima lettura r1(X) e all'ultima scrittura w2(X):

- il tempo fisico di w3(X) è maggiore del tempo fisico di r1(X) e w2(X)
- il tempo logico di w3(X) è maggiore del tempo logico di r1(X) e w2(X)

Concludiamo che non c'è incompatibilità (**write ok**)

Regole di timestamp – caso 2a (write ok)

Caso 2.a:



Si procede così:

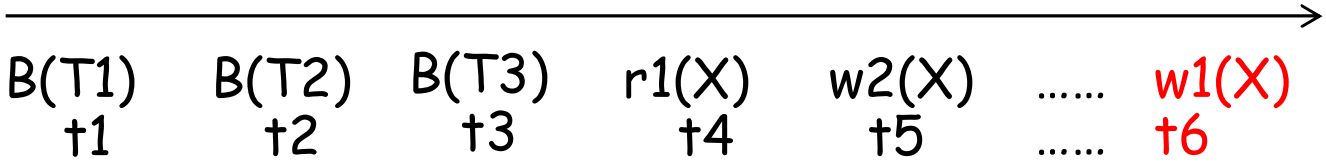
SE cb(X) è true (oppure nessuna transazione attiva ha scritto su X), allora

- si pone wts(X) pari a ts(T3)
- si pone cb(X) pari a false
- l'azione w3(X) di T3 viene eseguita, e si continua

ALTRIMENTI T3 viene messa in attesa del commit o rollback della transazione T' che per ultima ha scritto su X, cioè T3 è in attesa che cb(X) diventi true (si noti che cb(X) diventa true sia che T' faccia commit, sia che T' subisca il rollback)

Regole di timestamp – caso 2b (Thomas rule)

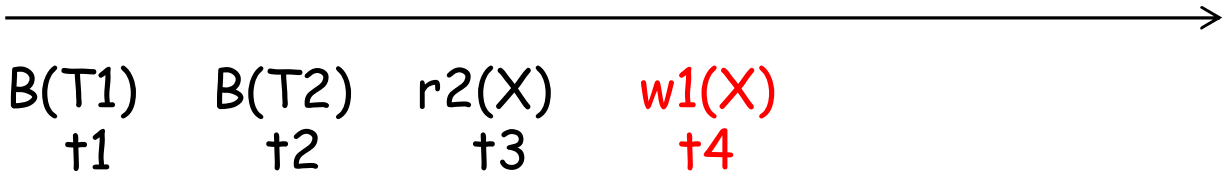
Caso 2.b:



- Consideriamo w1(X) rispetto all'ultima lettura r1(X): il tempo fisico di w1(X) è maggiore del tempo fisico di r1(X), e siccome w1(X) ed r1(X) appartengono alla stessa transazione, non c'è incompatibilità rispetto al tempo logico.
- È necessario però confrontare w1(X) anche con l'ultima scrittura w2(X) su X. Nel tempo logico, la scrittura di T2 è successiva a quella di T1, e se eseguissi w1(X), avrei un'anomalia di update loss. Allora:
  1. se cb(X) è true (cioè T2 ha già fatto commit) la soluzione è **ignorare w1(X)** e continuare. In questo modo, otteniamo l'effetto di sovrascrivere correttamente il valore che avrebbe scritto T1 con quello scritto da T2
  2. se cb(X) è false, dobbiamo **mettere T1 in attesa di cb(X)=true** ovvero del commit o del rollback della transazione che ha scritto per ultima X

Regole di timestamp – caso 2c (write too late)

Caso 2.c:



Consideriamo w1(X) rispetto all'ultima lettura r2(X) su X:

- il tempo fisico di w1(X) è t4, che è maggiore del tempo fisico di r2(X), cioè t3
- il tempo logico di w1(X) è ts(T1), che è minore del tempo logico di r2(X), cioè rts(X) = ts(T2)

Concludiamo che c'è incompatibilità (**write too late**).

Si procede così: l'azione w1(X) di T1 non può essere eseguita, T1 subisce il rollback, e viene fatta ricominciare con un nuovo timestamp.

Lo scheduler nel metodo dei timestamp

A fronte dell'operazione ri(X):

```
se          ts(Ti) >= wts(X)
allora      se cb(X)=true oppure se ts(Ti) = wts(X)                  // (caso 1.a)
            allora poni rts(X) = max(ts(Ti), rts(X)) ed esegui ri(X) // (caso 1.a.1)
            altrimenti metti Ti in attesa di cb(X)=true o del
                       rollback della transazione che ha scritto X   // (caso 1.a.2)
altrimenti  rollback(Ti)                                             // (caso 1.b)
```

A fronte dell'operazione wi(X):

```
se          ts(Ti) >= rts(X) e ts(Ti) >= wts(X)

allora      se cb(i)=true
            allora poni wts(X)=ts(Ti) e cb(X)=false, ed esegui wi(X) // (caso 2.a.1)
            altrimenti metti Ti in attesa di cb(X)=true o del
                       rollback della transazione che ha scritto X   // (caso 2.a.2)
altrimenti  se          ts(Ti) >= rts(X) e ts(Ti) < wts(X)           // (caso 2.b)
            allora      se cb(X)=true
                        allora ignora wi(X)                          // (caso 2.b.1)
                        altrimenti metti Ti in attesa di cb(X)=true
                                   o del rollback della transazione
                                   che ha scritto X                  // (caso 2.b.2)
            altrimenti rollback(Ti)                                  // (caso 2.c)
```

Lo scheduler nel metodo dei timestamp

A fronte dell'operazione commit **ci** da parte di Ti:

- per ogni elemento X della base di dati scritto da Ti,
  - poni cb(X) = true;
  - per ogni transazione Tj in attesa di cb(X)=true o del rollback della transazione che ha scritto per ultima X, consenti a Tj di procedere (Ti viene cioè tolta dallo stato di attesa);
- scegli la transazione da far procedere.

A fronte dell'operazione rollback bi da parte di Ti:

- per ogni elemento X scritto da Ti:
  - poni wts(X) uguale al timestamp della transazione Tj che ha scritto X prima di Ti, e poni cb(X) a true (si noti che Tj ha certamente fatto commit);
  - per ogni transazione Tj in attesa di cb(X) = true (con X scritto da Ti) o del rollback di Ti, consenti a Tj di procedere;
- scegli la transazione da far procedere.

Riassumendo

Nel metodo dei timestamp, una transazione Ti:

- non può leggere un dato scritto da una transazione con un timestamp maggiore (read too late)
- non può scrivere un dato letto da una transazione con un timestamp maggiore (write too late)
- se prova a scrivere un dato scritto da una transazione Tj con un timestamp maggiore, tale scrittura viene ignorata se Tj ha già fatto commit, altrimenti Ti viene sospesa fino al commit o al rollback di Tj (Thomas rule)
- quando legge un dato scritto da una transazione Tj con un timestamp minore che non ha ancora fatto commit (o rollback), Ti viene messa in attesa del commit (o rollback) di Tj (read ok, caso b)
- in tuti gli altri casi, può leggere e scrivere liberamente

Il deadlock nel metodo dei timestamp

Purtroppo il metodo dei timestamp non sventa il pericolo di deadlock, sebbene la probabilità di deadlock sia minore che nel caso del metodo con i lock.

Il deadlock nel metodo dei timestamp è legato all'uso del commit-bit.

Consideriamo ad esempio il seguente schedule:

w1(B), w2(A), w1(A), r2(B)

Al momento di w1(A), la transazione T1 viene messa in attesa del commit di T2. Al momento di r2(B), la transazione T2 viene messa in attesa del commit di T1.

Il deadlock nel metodo dei timestamp si affronta con le tecniche già viste nella trattazione del 2PL.

Esercizio

Descrivere il comportamento dello scheduler durante l'esecuzione della seguente sequenza:

r6(A) r8(A) r9(A) w8(A) w11(A) r10(A) c11

In particolare, descrivere quali azioni vengono eseguite, quali vengono sospese e le eventuali azioni di rollback, nonché l'evoluzione dei timestamp e del commit bit di A

Esempio di uso di timestamp

Azione	Effetto	Nuovi valori
r6(A)	ok	rts(A) = 6
r8(A)	ok	rts(A) = 8
r9(A)	ok	rts(A) = 9
w8(A)	no	T8 uccisa (write too late)
w11(A)	ok	wts(A) = 11, cb(A)=false
r10(A)	no	T10 uccisa (read too late)
c11	ok	cb(A) = true

Timestamp e conflict-serializzabilità

- Esistono schedule che sono conflict-serializzabili, ma che non sono accettati con il metodo dei timestamp, come ad esempio:

  r1(Y) r2(X) w1(X)

- Se S è uno schedule accettato con il metodo dei timestamp e non soggetto alla Thomas rule, allora lo schedule ottenuto da S eliminando tutte le azioni delle transazioni che hanno effettuato rollback è conflict-serializzabile

- Se uno schedule è accettato con il metodo dei timestamp, ed è soggetto alla Thomas rule, allora può non essere conflict-serializzabile, come ad esempio,

  r1(A) w2(A) c2 w1(A) c1

  Si noti però che se uno schedule S è accettato con il metodo dei timestamp ed è soggetto alla Thomas rule, allora lo schedule S' ottenuto da S eliminando le scritture ignorate dalla Thomas rule ed eliminando tutte le azioni delle transazioni che hanno effettuato rollback è conflict-serializzabile

Confronto tra timestamp e 2PL

- Esistono schedule che sono accettati con il metodo dei timestamp, ma che non sono schedule 2PL. Ad esempio,

  r1(A) w2(A) r3(A) r1(B) w2(B) r1(C) w3(C) r4(C) w4(B) w5(B)

  è accettato con il metodo dei timestamp, ma non è uno schedule 2PL, perché T2 prima deve rilasciare il lock su A affinché A venga letto da T3 con r3(A), e solo successivamente può chiedere il lock per scrivere su B con w2(B), perché se lo chiedesse prima, T1 non potrebbe effettuare la lettura di B.

- Esistono schedule che sono accettati con il metodo dei timestamp, ed appartengono agli schedule del 2PL stretto, ad esempio tutti gli schedule seriali come:

  r1(A) w1(A) r2(A) w2(A)

- Esistono schedule che sono in 2PL (stretto), ma non sono accettati con il metodo dei timestamp, come ad esempio:

  r1(B) r2(A) w2(A) r1(A) w1(A)

  in cui T2 acquisirebbe il timestamp dopo T1 ma scriverebbe A prima della lettura di A da parte di T1

Confronto tra timestamp e 2PL

- Attesa
  - 2PL: le transazioni vengono messe in attesa
  - TS: vengono uccise e fatte ricominciare (oppure vengono messe in attesa di commit di altre transazioni, vedi sotto)
- Ordine di serializzazione
  - 2PL: è imposto dai conflitti
  - TS: è imposto dai timestamp
- Necessità di attendere il commit di altre transazioni
  - 2PL: risolto dal protocollo strict 2PL
  - TS: attesa di cb(X) = true (casi read ok e Thomas rule)
- Deadlock
  - 2PL è soggetto a deadlock
  - TS può costringere a restart, ma il deadlock è improbabile

Confronto tra timestamp e 2PL

- Il metodo dei timestamp è superiore in genere quando le transazioni sono read only, o quando è raro che transazioni concorrenti leggano e scrivano lo stesso elemento

- Il 2PL è superiore nelle situazioni di alto conflitto perché:
  - il locking ritarda sì le transazioni quando sono in attesa di lock e può anche portare, in caso di deadlock, a rollback
  - ma la probabilità di rollback è superiore nel caso di timestamp, dando luogo a ritardi medi superiori che nel 2PL

Metodo dei timestamp multiversione

Idea: **non bloccare le istruzioni di lettura**, introducendo versioni diverse X1 … Xn di un elemento X in modo che un'azione di lettura possa sempre essere effettuata leggendo la versione "giusta" rispetto al tempo logico determinato dai timestamp

- Ogni scrittura wi(X) legale genera una nuova versione Xi (la notazione che usiamo fa sì che il pedice che denota la versione è uguale al timestamp della transazione che l'ha generata)
- Ad ogni versione Xh di X è assegnato il timestamp wts(Xh)=ts(Th), che indica il ts della transazione che ha scritto quella versione
- Ad ogni versione è assegnato il timestamp rts(Xi)=ts(Th), che indica il ts più alto tra quelle che hanno letto Xi

Nuove regole per l'utilizzo dei timestamp

Lo scheduler utilizza i timestamp come segue:

- A fronte di wi(X): se c'è già stata una lettura rj(Xk) tale che wts(Xk) < Ts(Ti) < Ts(Tj), allora la scrittura non viene accettata (write too late), altrimenti si esegue la scrittura sulla nuova versione Xi di X, e si pone wts(Xi) = ts(Ti).
- A fronte di ri(X): si esegue la lettura selezionando la versione Xj tale che wts(Xj) è il timestamp più alto tra quelli minori di o uguali a ts(Ti), cioè: Xj è tale che wts(Xj) <= ts(Ti), e non esiste alcuna versione Xh tale che wts(Xj) < wts(Xh) <= ts(Ti). Ovviamente, si aggiorna poi nel modo usuale rts(X_j).
- Quando esiste Xj tale che wts(Xj) è tale che nessuna transazione attiva ha timestamp minore di j, allora si eliminano le versioni di X che sono precedenti rispetto a Xj, dalla più vecchia alla più nuova.
- Per assicurare la recuperabilità, per ogni transazione Ti si ritarda il commit di Ti fino a che non sono stati eseguiti i commit da parte di tutte le transazioni Tj da cui Ti ha letto.

Nuove regole per l'utilizzo dei timestamp

Lo scheduler fa uso di opportune strutture di dati:

- Per ogni versione Xi lo scheduler mantiene un intervallo intervallo(Xi) = [wts, rts], dove wts è il timestamp della transazione che ha scritto Xi, ed rts è il timestamp più grande tra quelli delle transazioni che hanno letto Xi (se nessuno ha letto Xi, allora rts=wts).
- Denotiamo con intervalli(X) l'insieme

  { intervallo(Xi) | Xi è una versione di X }
- Quando processa ri(X), lo scheduler usa intervalli(X) per trovare la versione Xj tale che intervallo(Xj) = [wts, rts] ha il massimo wts minore o uguale al timestamp ts(Ti) di Ti. Se poi ts(Ti) > rts, allora rts viene posto uguale a ts(Ti)
- Quando processa wi(X), lo schedule usa intervalli(X) per trovare la versione Xj tale che intervallo(Xj) = [wts, rts] ha il massimo wts minore o uguale al timestamp ts(Ti) di Ti. Se poi rts > ts(Ti), allora wi(X) viene rifiutato, altrimenti wi(X) viene accettato, e viene creata la versione Xi con intervallo(Xi) = [wts, rts], dove wts = rts = ts(Ti).

Confronto con timestamp universione

Vantaggi:

- Non ci sono i commit bit (meccanismi di lock in scrittura)
- Non c'è mai attesa in lettura (niente sospensione per read ok)
- Non c'è mai rollback causato da letture (niente read too late)
- Non c'è mai attesa in scrittura (niente sospensione per write ok)
- Non c'è rischio di deadlock

Svantaggi:

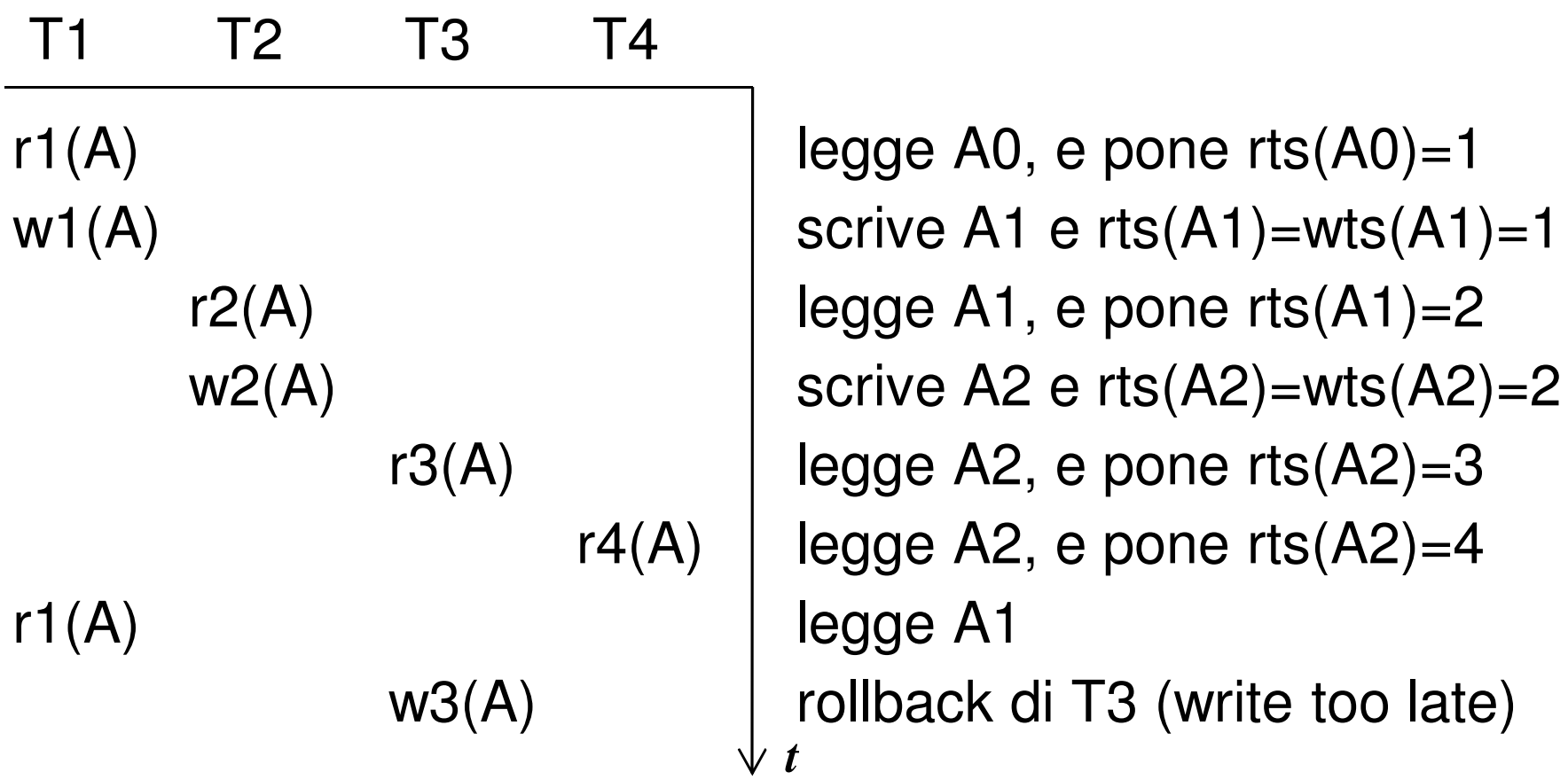
- Lo schedule NON è ACR (c'è il rischio di cascading rollback e quindi di basso throughput nelle situazioni di alto conflitto)
- Gestione di molte copie degli elementi della base di dati

Il metodo dei timestamp multiversione è il principale esempio del cosiddetto **Multiversion Concurrency Control (MVCC),** ovvero gestione della concorrenza basata su versioni multiple della base di dati

Esempio del timestamp multiversione

Supponiamo che la versione attuale di A sia A0, con rts(A)=0.

Sequenza S = r1(A) w1(A) r2(A) w2(A) r3(A) r4(A) r1(A) w3(A)



legge A0, e pone rts(A0)=1

scrive A1 e rts(A1)=wts(A1)=1

legge A1, e pone rts(A1)=2

scrive A2 e rts(A2)=wts(A2)=2

legge A2, e pone rts(A2)=3

legge A2, e pone rts(A2)=4

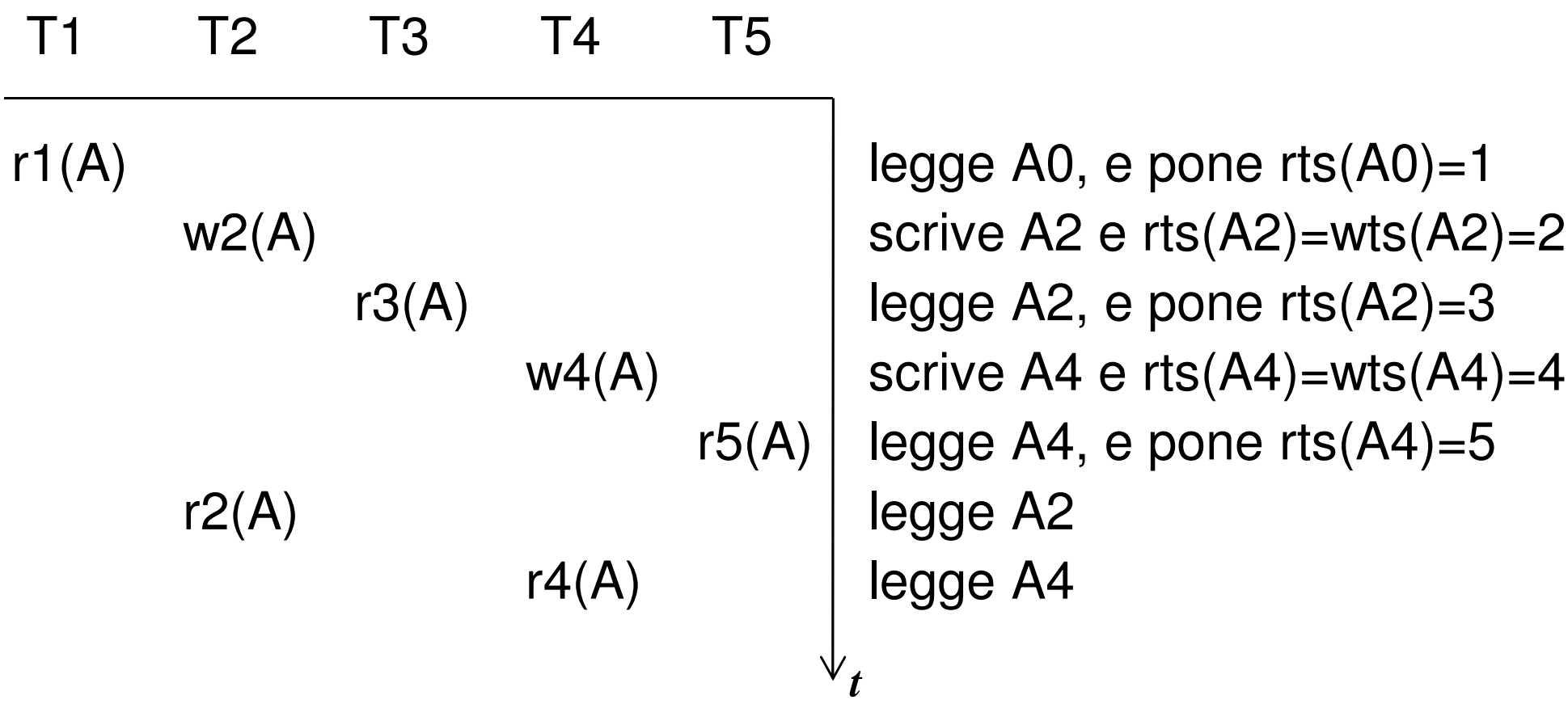
legge A1

rollback di T3 (write too late)

Altro esempio del timestamp multiversione

Supponiamo che la versione attuale di A sia A0, con rts(A)=0.

sequenza S = r1(A) w2(A) r3(A) w4(A) r5(A) r2(A) r4(A)



N.B.: lo schedule S NON è view-serializzabile!

Gestione della concorrenza in SQL

- In SQL-92 sono presenti costrutti per la definizione di transazioni e del livello di concorrenza
- Una singola istruzione (SELECT) è considerata come una unità atomica di esecuzione
- SQL non prevede una esplicita istruzione BEGIN TRANSACTION
- In SQL ogni transazione deve avere un esplicita istruzione di terminazione, che può essere o COMMIT o ROLLBACK

Esempio

```
        EXEC SQL WHENEVER sqlerror GO TO ESCI;
        EXEC SQL SET TRANSACTION
          READ WRITE , DIAGNOSTICS SIZE 8,
          ISOLATION LEVEL SERIALIZABLE;
        EXEC SQL INSERT INTO
          EMPLOYEE (Name, ID, Address)
          VALUES (‘John Doe',1234,‘xyz');
        EXEC SQL UPDATE EMPLOYEE
          SET Address = ‘abc'
          WHERE ID = 1000;
        EXEC SQL COMMIT;
        GOTO FINE;
ESCI:   EXEC SQL ROLLBACK;
FINE:   …
```

Letture fantasma

- Poiché SQL considera atomica una query, va considerata una ulteriore anomalia, la cosiddetta lettura fantasma
- Esempio:
  - T1 esegue la query SELECT * FROM R
  - T2 aggiunge alla relazione R un record r
  - T1 riesegue la query precedente, e il risultato della seconda lettura ha il record r in piu' (fantasma)
- La lettura fantasma è una versione dell'anomalia di lettura non ripetibile generalizzata al caso in cui la lettura avviene mediante una query che legge un insieme di record

L'istruzione SET TRANSACTION

- L'istruzione SET TRANSACTION permette di definire i seguenti aspetti:
- Modo di accesso: valore READ ONLY oppure READ WRITE
- Livello di isolamento della transazione: può assumere uno tra i seguenti valori:
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE (valore di default)
- Configurazione del numero di condizioni di errore gestibili contemporaneamente (DIAGNOSTIC SIZE n)

Livelli di isolamento

(Assumiamo che l'istruzione SET TRANSACTION sia relativa alla transazione Ti)

- SERIALIZABLE:
  - La transazione Ti legge solo da transazioni committed
  - Nessun valore letto o scritto dalla transazione Ti può essere modificato da altre transazioni fino al commit di Ti
  - L'insieme di record letto da Ti attraverso una query non può essere modificato da altre transazioni fino al commit di Ti (evita l'anomalia delle letture fantasma)

Livelli di isolamento

- REPEATABLE READ:
  - La transazione Ti legge solo da transazioni committed
  - Nessun valore letto o scritto dalla transazione Ti può essere modificato da altre transazioni fino al commit di Ti
  - L'insieme di record letto da Ti attraverso una query può essere modificato da altre transazioni prima del commit di Ti (sono quindi possibili letture fantasma)

Livelli di isolamento

- READ COMMITTED:
  - La transazione Ti legge solo da transazioni committed
  - Nessun valore scritto dalla transazione Ti può essere modificato da altre transazioni fino al commit di Ti, mentre i valori letti da Ti possono essere modificati (sono quindi possibili letture non ripetibili e letture fantsma)

Livelli di isolamento

- READ UNCOMMITTED:
  - La transazione Ti può leggere da qualsiasi transazione anche non committed (è quindi possibile il cascading rollback)
  - Sia i valori scritti che i valori letti da Ti possono essere modificati (sono quindi possibili letture sporche, letture non ripetibili e letture fantasma)

Gestione della concorrenza nei sistemi commerciali

- I gestori della concorrenza dei principali sistemi commerciali (Oracle, DB2, SQL Server, PostgreSQL) utilizzano scheduler basati su lock e/o su timestamp (in particolare, timestamp multiversione)
- Gli scheduler di tali sistemi spesso dividono le transazioni in due classi:
  - Le transazioni con read e write sono eseguite con 2PL
  - Le “read only” sono eseguite con il metodo dei timestamp multiversione