Gestione della concorrenza

Controllo della concorrenza

- La **concorrenza** è **fondamentale**: decine o centinaia di transazioni al secondo, non possono essere seriali
  - Esempi: banche, prenotazioni aeree
- **Modello di riferimento**:
  - **Operazioni** di input-output su **oggetti astratti** x, y, z
- **Problema**:
  - **Anomalie** causate dall'**esecuzione concorrente**, che quindi va governata

Perdita di aggiornamento

- Due transazioni identiche:
  - $T_1 : r(x), x \leftarrow x + 1, w(x)$
  - $T_2 : r(x), x \leftarrow x + 1, w(x)$
- Inizialmente $x = 2$; dopo un'esecuzione seriale $x = 4$
- Un'esecuzione concorrente:

T_1	T_2
bot	
$r(x)$	
$x \leftarrow x + 1$	
	bot
	$r(x)$
	$x \leftarrow x + 1$
$w(x)$	
commit	
	$w(x)$
	commit

- Un aggiornamento viene perso: $x = 3$

Lettura sporca

T_1	T_2
bot	
$r(x)$	
$x \leftarrow x+1$	
$w(x)$	
	bot
	$r(x)$
abort	
	commit

- T_2 ha letto uno **stato intermedio** ("**sporco**") e lo può comunicare all'esterno

Letture inconsistenti

T_1	T_2
bot	
$r(x)$	
	bot
	$r(x)$
	$x \leftarrow x + 1$
	$w(x)$
	commit
$r(x)$	
commit	

- T_1 **legge due valori diversi** per x

Aggiornamenti fantasma

- Assumiamo di avere il vincolo $y + z = 1000$:

T_1	T_2
bot	
$r(y)$	
	bot
	$r(y)$
	$y \leftarrow y - 100$
	$r(z)$
	$z \leftarrow z + 100$
	$w(y)$
	$w(z)$
	commit
$r(z)$	
$s \leftarrow y + z$	
commit	

- $s = 1100$: il vincolo sembra non soddisfatto, T_1 vede un aggiornamento non coerente

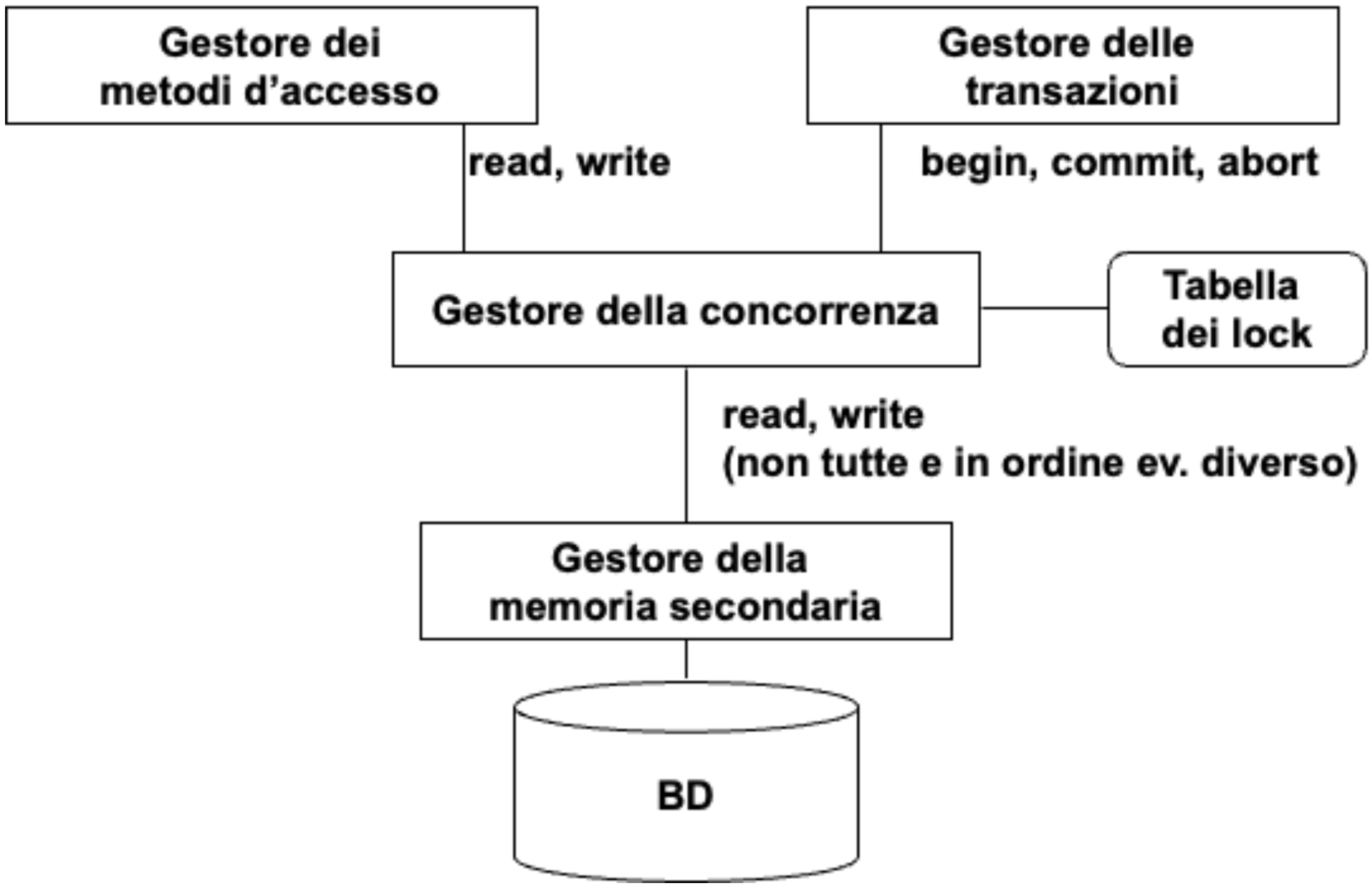
Inserimento fantasma

T_1	T_2
bot	
legge gli stipdenti degli impiegati del dip. A e calcola la media	
	bot
	inserisce un impiegato in A
	commit
legge gli stipdenti degli impiegati del dip. A e calcola la media	
commit	

Anomalie

- Perdita di aggiornamento
  - W-W
- Lettura sporca
  - R-W (o W-W) con abort
- Letture inconsistenti
  - R-W
- Aggiornamento fantasma
  - R-W
- Inserimento fantasma
  - R-W su dato "nuovo"

Gestore della concorrenza



Schedule

- Uno ***schedule*** S è una sequenza di operazioni di lettura/scrittura di transazioni concorrenti
- Esempio:

$$S : r_1(x) \quad r_2(z) \quad w_1(x) \quad w_2(z)$$

- dove
  - $r_1(x)$ rappresenta la lettura dell'oggetto x da parte della transazione T_1
  - $w_2(z)$ rappresenta la scrittura dell'oggetto z da parte della transazione T_2
- Le operazioni compaiono nello schedule nell'ordine temporale di esecuzione sulla base di dati

Controllo di concorrenza

- Il **controllo della concorrenza** è eseguito dallo ***scheduler***, che tiene traccia di tutte le operazioni eseguite sulla base di dati dalle transazioni e decide se accettare o rifiutare le operazioni che vengono via via richieste
- Obiettivo: **evitare le anomalie**

- Per il momento, assumiamo che **l'esito** (commit/abort) delle transazioni sia **noto a priori** (ipotesi **commit-proiezione**)
  - In questo modo possiamo **rimuovere** dallo schedule tutte le **transazioni abortite**
  - Si noti che tale assunzione **non consente** di trattare **alcune anomalie** (lettura sporca)

Schedule seriale

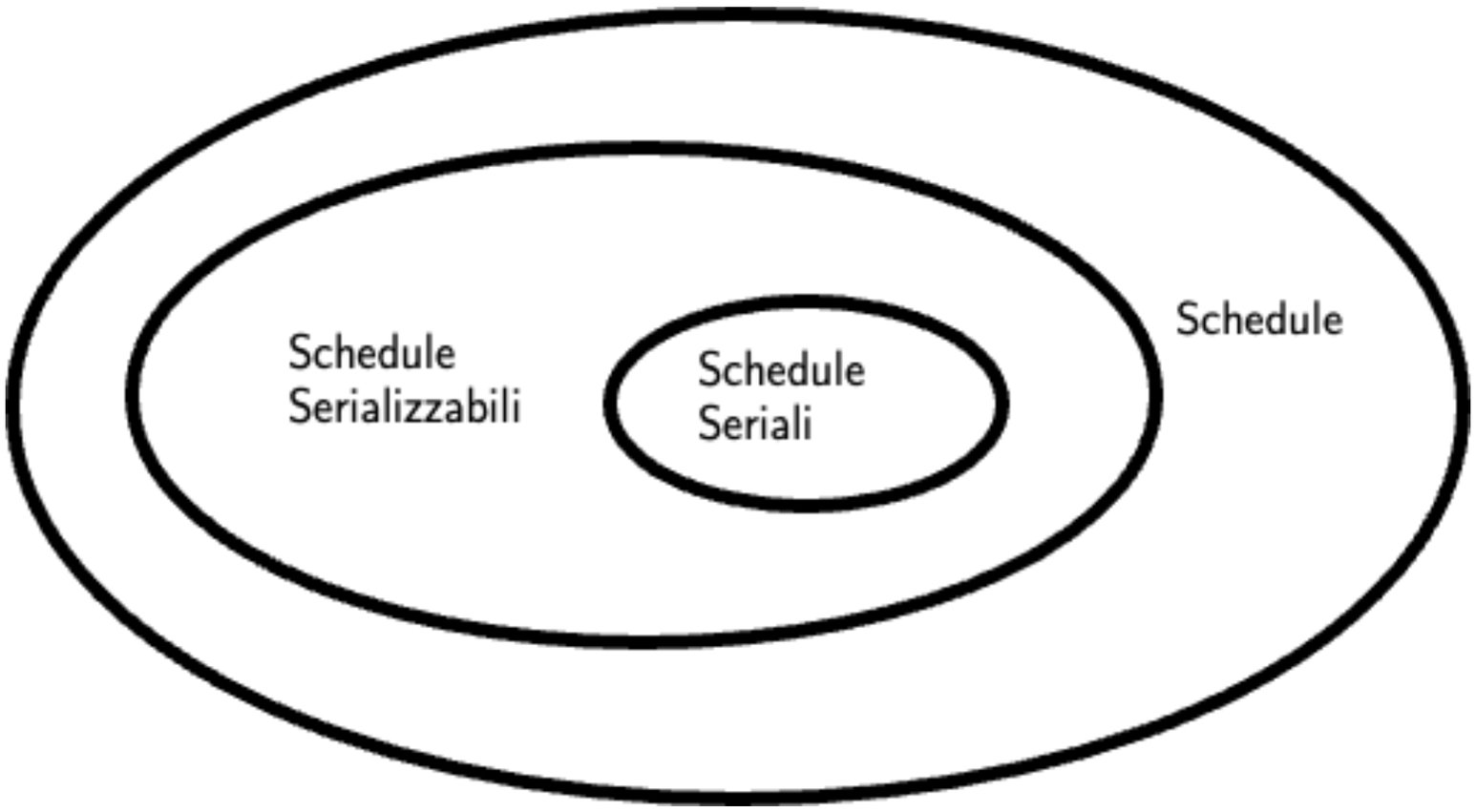
- Uno **schedule** di un insieme di transazioni $T = \{T_1, \ldots, T_n\}$ è detto **seriale** se, per ogni coppia di transazioni $T_i, T_j \in T$, tutte le operazioni di T_i sono eseguite prima di qualsiasi operazione di T_j, o viceversa
- Esempio:
  - $T = \{T_0, T_1, T_2\}$
    $S = r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)$
  - $w_1(y)\ r_2(x)\ r_2(y)\ w_2(z)\ w_2(z)$

Schedule serializzabile

- Uno **schedule** di un insieme di transazioni è **serializzabile** se la sua esecuzione produce lo stesso risultato di uno schedule seriale sulle stesse transazioni
- Richiede una nozione di **equivalenza fra schedule**

Idea base

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



View-Serializzabilità

- Diciamo che esiste la relazione **legge-da** tra le operazioni $r_i(x)$ e $w_j(x)$ presenti in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è nessun $w_k(x)$, con $k \neq j$ tra $r_i(x)$ e $w_j(x)$ in S
- La scrittura $w_j(x)$ in S è detta **scrittura finale su** x se è l'ultima scrittura su x in S
- Due schedule S_i e S_j sono detti **view-equivalenti**, $S_i \approx_v S_j$ se hanno la **stessa relazione legge-da** e le **stesse scritture finali** su **ogni oggetto**
- Uno schedule S è **view-serializzabile** se è view-equivalente ad un qualche schedule seriale
- L'insieme degli schedule view-serializzabili è indicato con VSR

Esempio

- Consideriamo i seguenti schedule:
  - $S_3 = w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  - $S_4 = w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$ (schedule seriale)
- S_3 è view-equivalente allo schedule seriale S_4?
  - legge-da$(S_3) = w_0(x)r_2(x), w_0(x)r_1(x)$
  - finale$(S_3) = w_2(x), w_2(z)$
  - legge-da$(S_4) = w_0(x)r_1(x), w_0(x)r_2(x)$
  - finale$(S_4) = w_2(x), w_2(z)$
  - Si, S_3 è view-equivalente allo schedule seriale S_4
    - Quindi è view-serializzabile

Esempio

- Consideriamo i seguenti schedule:
  - $S_4 = w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$ (schedule seriale)
  - $S_5 = w_0(x)\ r_2(x)\ w_2(x)\ r_1(x)\ w_2(z)$
- S_5 è view-equivalente allo schedule seriale S_4?
- legge-da$(S_4) = w_0(x)r_1(x), w_0(x)r_2(x)$
- finale$(S_4) = w_2(x), w_2(z)$
- legge-da$(S_5) = w_0(x)r_2(x), w_2(x)r_1(x)$
- finale$(S_5) = w_2(x), w_2(z)$
- No, S_5 non è view-equivalente allo schedule seriale S_4
  - Non vuol dire che non sia view-serializzabile, proviamo un altro schedule seriale

Esempio

- Consideriamo i seguenti schedule:
  - $S_5 = w_0(x)\ r_2(x)\ w_2(x)\ r_1(x)\ w_2(z)$
  - $S_6 = w_0(x)\ r_2(x)\ w_2(x)\ w_2(z)\ r_1(x)$ (schedule seriale)
- S_5 è view-equivalente allo schedule seriale S_6?
  - legge-da$(S_5) = w_0(x)r_2(x), w_2(x)r_1(x)$
  - finale$(S_5) = w_2(x), w_2(z)$
  - legge-da$(S_6) = w_0(x)r_2(x), w_2(x)r_1(x)$
  - finale$(S_6) = w_2(x), w_2(z)$
  - Si, S_5 è view-equivalente allo schedule seriale S_6
    - Quindi è view-serializzabile

Esempio

- Consideriamo i seguenti schedule:
  - $S_7 = r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$
    - nessuno schedule seriale view-equivalente
    - è una perdita di aggiornamento
  - $S_8 = r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
    - nessuno schedule seriale view-equivalente
    - è una lettura inconsistente
  - $S_9 = r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$
    - nessuno schedule seriale view-equivalente
    - è un aggiornamento fantasma
- S_7, S_8 e S_9 non sono view serializzabili
  - Non sono view-equivalenti a nessuno schedule seriale

Uso della View-Serializzabilità

- **Complessità**:
  - la **verifica** della view-equivalenza di due dati schedule ha complessità **polinomiale**
  - il **decidere** sulla view-serializzabilità di uno schedule è un problema **NP-completo**
    - È necessario confrontate lo schedule con tutti i possibili schedule seriali
- **Non** è **utilizzabile in pratica**
  - Soluzione: definiamo una **condizione di equivalenza più ristretta**, che non copra tutti i casi di equivalenza tra schedule coperti della view-equivalenza, ma che sia **utilizzabile nella pratica** (la procedura di verifica abbia cioè una complessità inferiore)

Conflict-Serializzabilità

- **Un'operazione** a_i è **in conflitto con un'altra operazione** a_j, con $i \neq j$, se operano sullo stesso oggetto e almeno una di esse è una scrittura
  - Nota bene: $a_i(x)a_j(x) \neq a_j(x)a_i(x)$, cioè **nei conflitti conta l'ordine**
- Esistono due casi:
  - conflitto read-write (R-W o W-R)
  - conflitto write-write (W-W)
- Due schedule S_i e S_j sono detti **conflict-equivalenti**, $S_i \approx_c S_j$ se hanno le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi
- Uno schedule S è **conflict-serializzabile** se è conflict-equivalente ad un qualche schedule seriale
- L'insieme degli schedule conflict-serializzabili è indicato con CSR

VSR e *CSR*

- **Teorema**: Ogni schedule conflict-serializzabile è view-serializzabile, ma non necessariamente viceversa

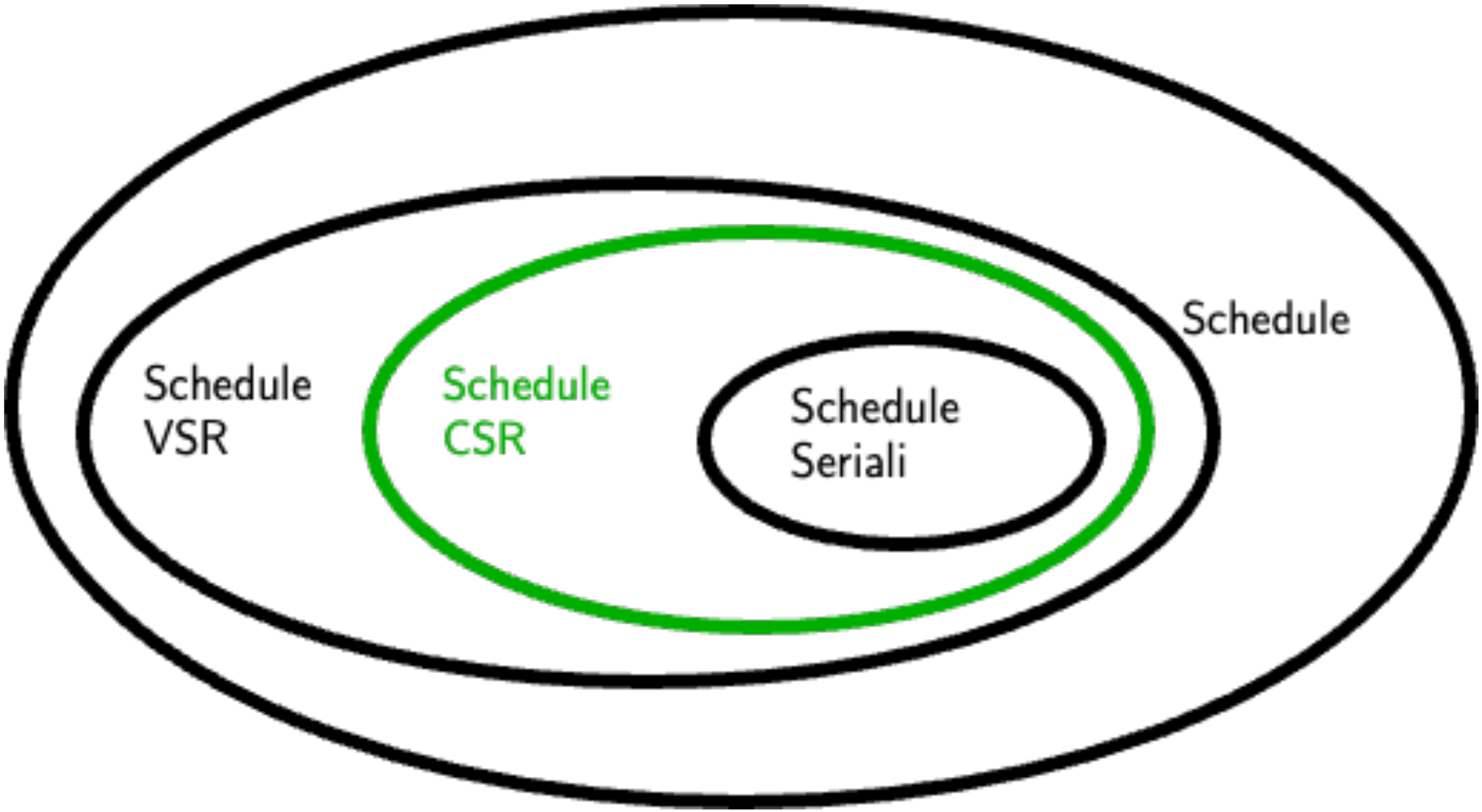
- Contro-esempio per la non necessità:
  - $r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$
- view-serializzabile: view-equivalente a
  - $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
- conflict-serializzabile:
  - No

VSR e *CSR*

- **Teorema**: Ogni schedule conflict-serializzabile è view-serializzabile, ma non necessariamente viceversa

- Per dimostrare che CSR implica VSR è sufficiente dimostrare che la conflict-equivalenza $\approx_c$ implica la view-equivalenza $\approx_v$, cioè che se due schedule sono $\approx_c$ allora sono $\approx_v$
- Quindi, supponiamo $S_1 \approx_c S_2$ e dimostriamo che $S_1 \approx_v S_2$. I due schedule hanno:
  - **stesse scritture finali**: se così non fosse, ci sarebbero almeno due scritture in ordine diverso e poiché due scritture sono in conflitto i due schedule non sarebbero $\approx_c$
  - **stessa relazione "legge-da"**: se così non fosse, ci sarebbero scritture in ordine diverso o coppie lettura-scrittura in ordine diverso e quindi, come sopra sarebbe violata la $\approx_c$

VSR e *CSR*



Verifica della Conflict-Serializzabilità

- Per mezzo del **grafo dei conflitti**:
  - un **nodo** per ogni **transazione** T_i
  - un **arco** (**orientato**) da T_i a T_j se c'è almeno un **conflitto** fra un'azione a_i e un'azione a_j tale che a_i **precede** a_j

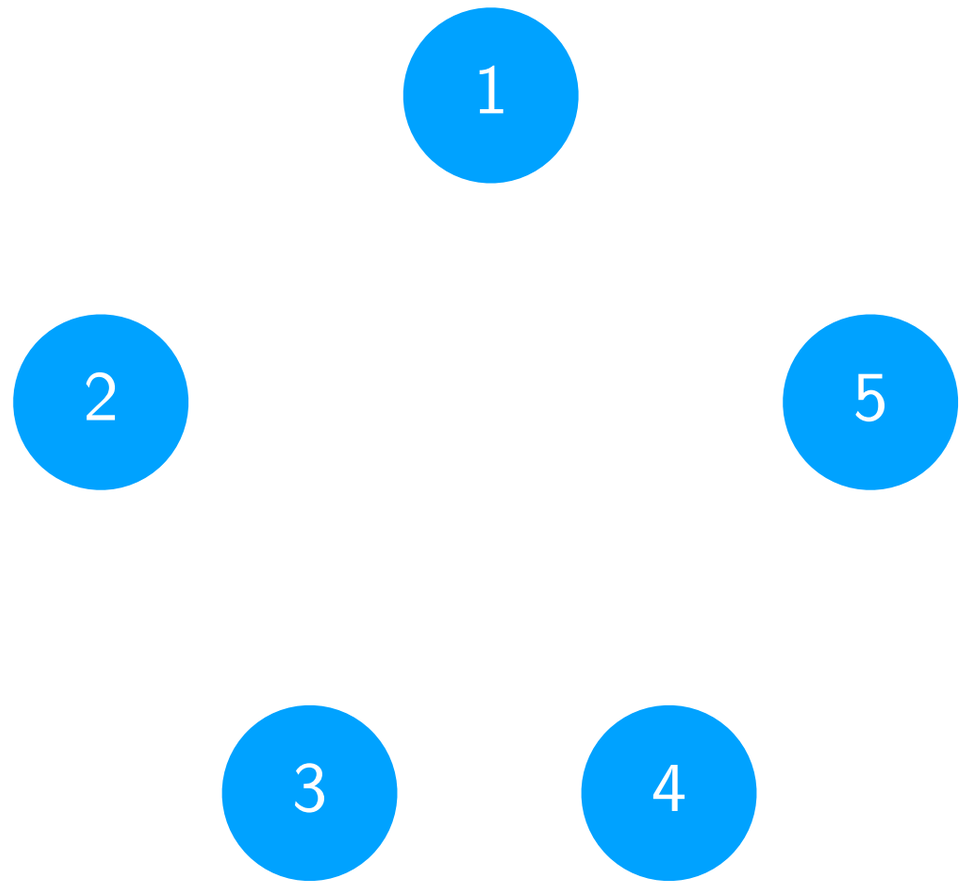
- **Teorema**: Uno schedule è in CSR se e solo se il **grafo è aciclico**

Esempio

- $S = w_1(x)w_2(x)r_3(x)r_1(y)w_2(y)r_1(z)w_3(z)r_4(z)w_4(y)w_5(y)$
- $x : w_1\ w_2\ r_3$
- $y : r_1\ w_2\ w_4\ w_5$
- $z : r_1\ w_3\ r_4$

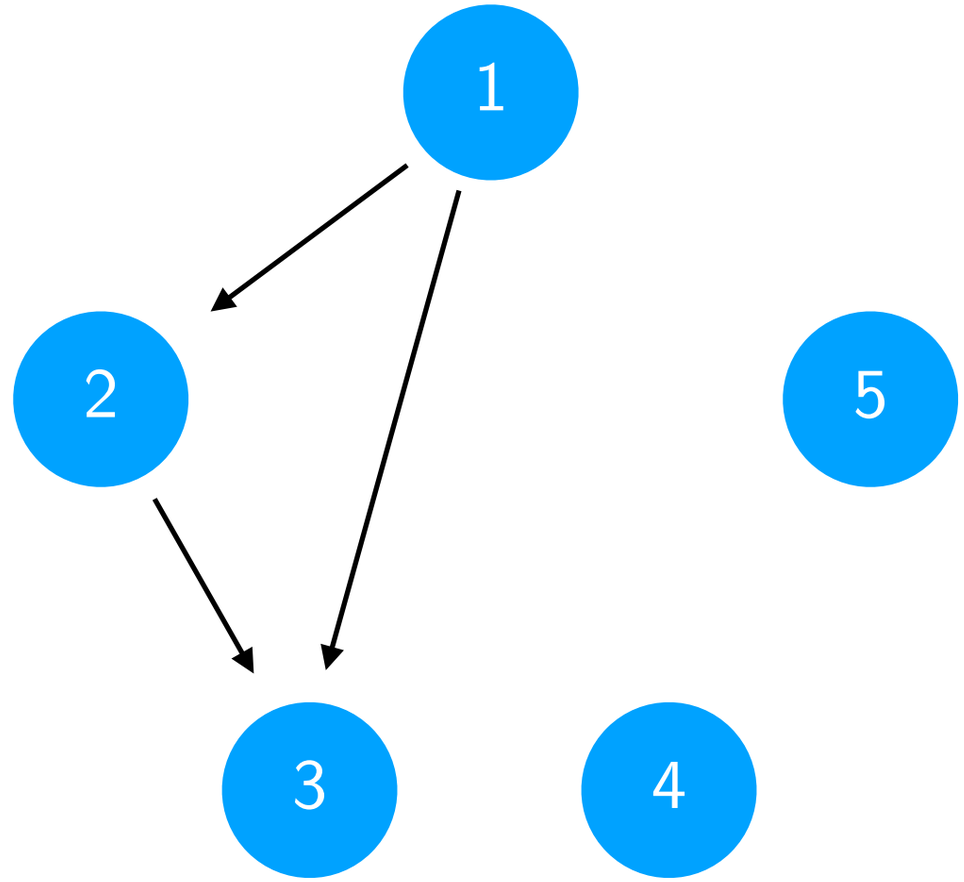
Esempio

- $S = w_1(x)w_2(x)r_3(x)r_1(y)w_2(y)r_1(z)w_3(z)r_4(z)w_4(y)w_5(y)$
- $x : w_1\ w_2\ r_3$
- $y : r_1\ w_2\ w_4\ w_5$
- $z : r_1\ w_3\ r_4$



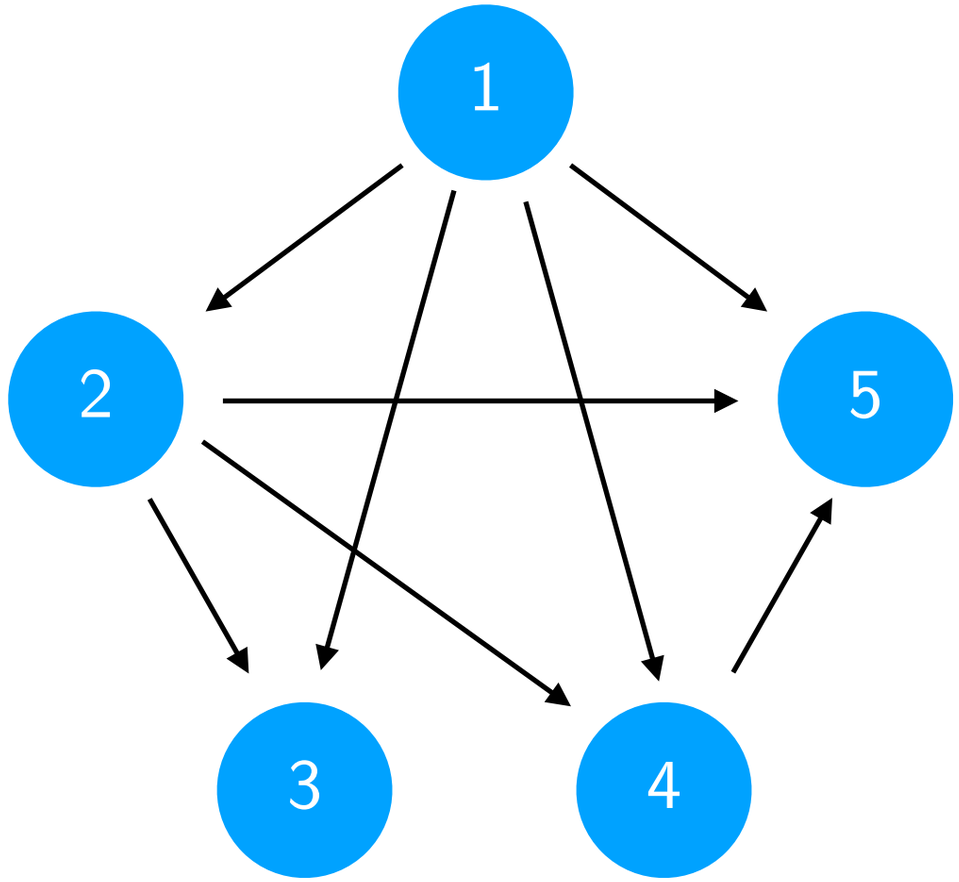
Esempio

- $S = w_1(x)w_2(x)r_3(x)r_1(y)w_2(y)r_1(z)w_3(z)r_4(z)w_4(y)w_5(y)$
- $x : w_1\ w_2\ r_3$
- $y : r_1\ w_2\ w_4\ w_5$
- $z : r_1\ w_3\ r_4$



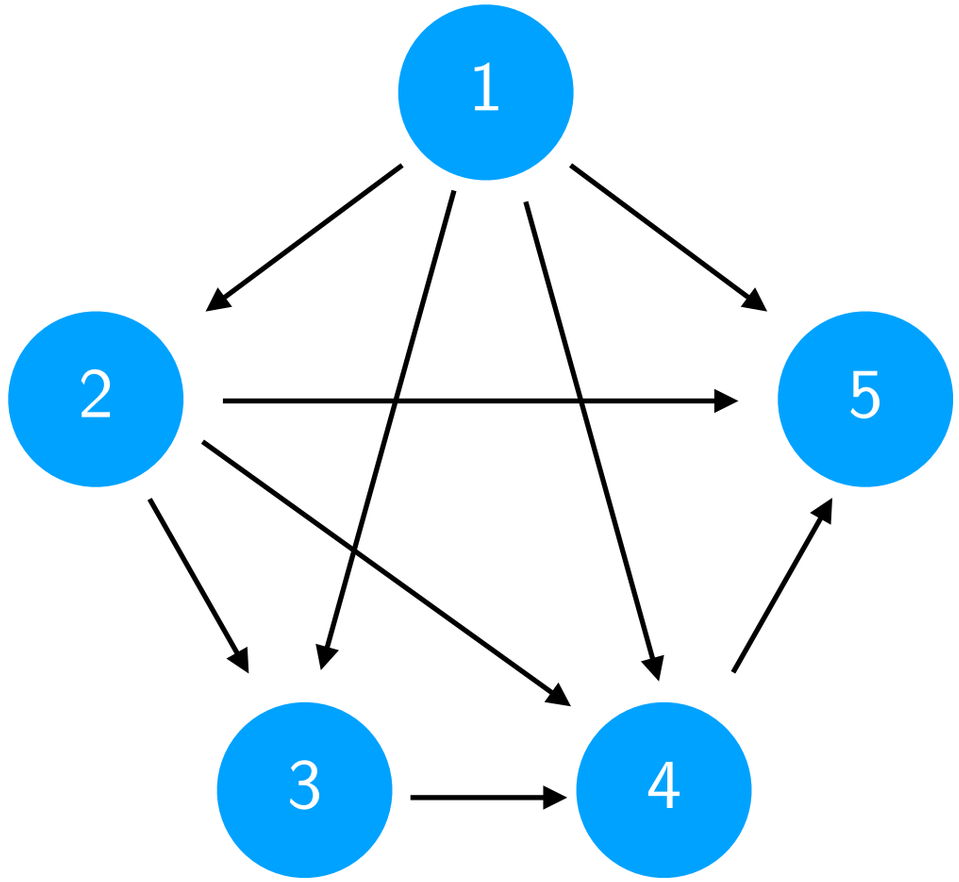
Esempio

- $S = w_1(x)w_2(x)r_3(x)r_1(y)w_2(y)r_1(z)w_3(z)r_4(z)w_4(y)w_5(y)$
- $x : w_1\ w_2\ r_3$
- $y : r_1\ w_2\ w_4\ w_5$
- $z : r_1\ w_3\ r_4$



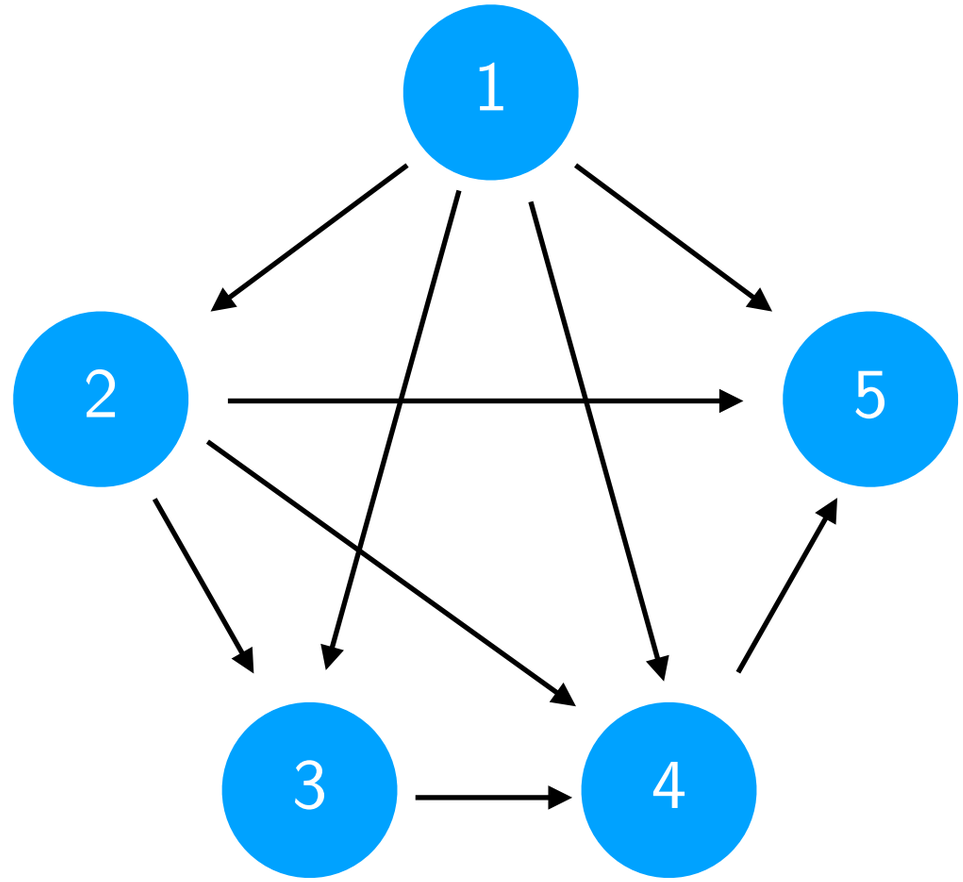
Esempio

- $S = w_1(x)w_2(x)r_3(x)r_1(y)w_2(y)r_1(z)w_3(z)r_4(z)w_4(y)w_5(y)$
- $x : w_1\ w_2\ r_3$
- $y : r_1\ w_2\ w_4\ w_5$
- $z : r_1\ w_3\ r_4$



Esempio

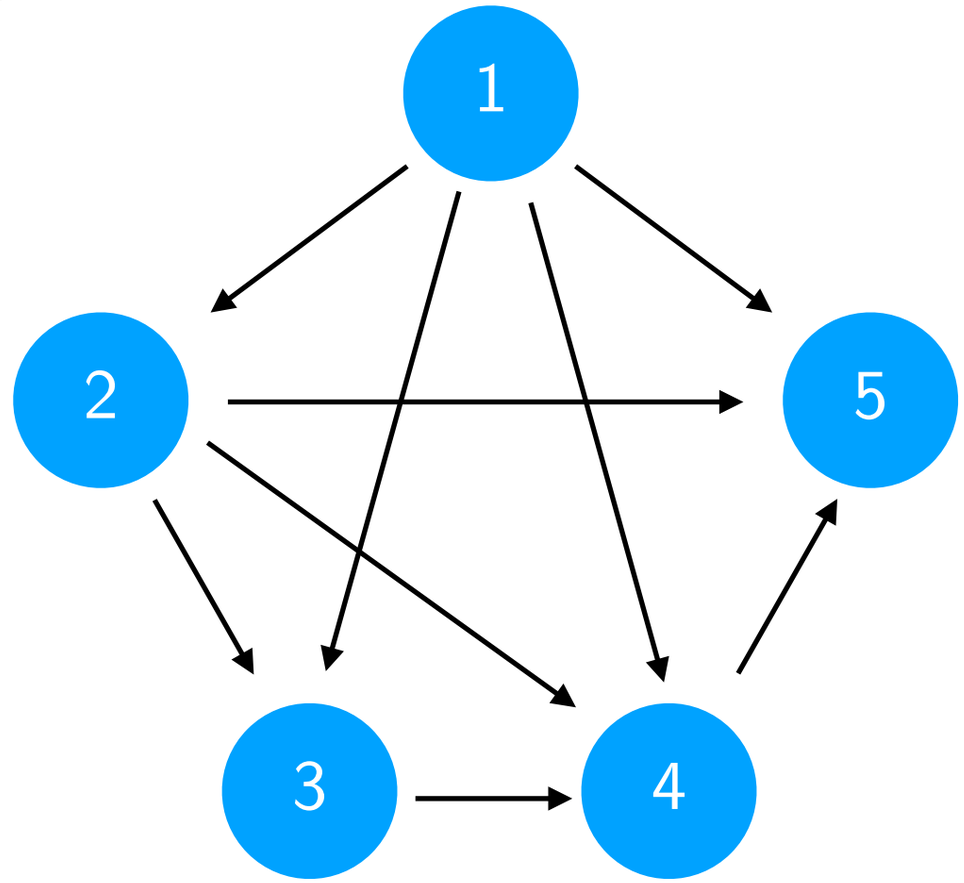
- $S = w_1(x)w_2(x)r_3(x)r_1(y)w_2(y)r_1(z)w_3(z)r_4(z)w_4(y)w_5(y)$
- $x : w_1\ w_2\ r_3$
- $y : r_1\ w_2\ w_4\ w_5$
- $z : r_1\ w_3\ r_4$
- Il grafo è aciclico
  - S è CSR, quindi anche VSR



Esempio

- $S = w_1(x)w_2(x)r_3(x)r_1(y)w_2(y)r_1(z)w_3(z)r_4(z)w_4(y)w_5(y)$

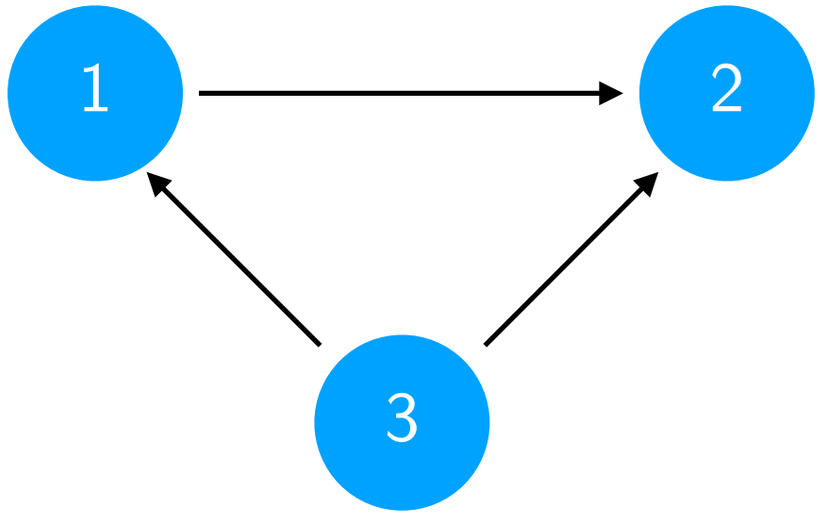
- Cerchiamo uno schedule seriale conflict-equivalente
  - T_1 ha conflitti con tutte le transazioni, quindi la mettiamo per prima
  - T_2 ha conflitti con tutte le transazioni rimanenti, quindi la mettiamo per seconda
  - T_3 deve venire prima di T_4
  - T_4 deve venire prima di T_5



- $w_1(x)r_1(y)r_1(z)w_2(x)w_2(y)r_3(x)w_3(z)r_4(z)w_4(y)w_5(y)$

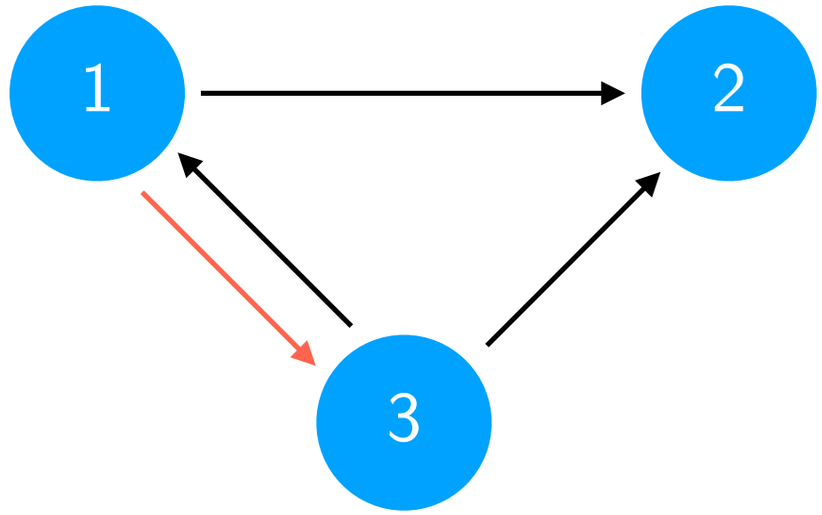
Esempio

- $S = r_1(y)w_3(z)r_1(z)r_2(z)w_3(x)w_1(x)w_2(x)r_3(y)$
- $x : w_3\ w_1\ w_2$
- $y : r_1\ r_3$
- $z : w_3\ r_1\ r_2$
- Il grafo è aciclico
  - S è CSR, quindi anche VSR
  - $w_3(z)w_3(x)r_3(y)r_1(y)r_1(z)w_1(x)r_2(z)w_2(x)$



Esempio

- $S' = r_1(y)r_1(z)w_3(z)r_1(z)r_2(z)w_3(x)w_1(x)w_2(x)r_3(y)$
- $x : w_3\ w_1\ w_2$
- $y : r_1\ r_3$
- $z : r_1\ w_3\ r_1\ r_2$
- Il grafo non è aciclico
  - S non è CSR
  - Ma S è VSR:
    - $r_1(y)r_1(z)w_1(x)w_3(z)w_3(x)r_3(y)r_2(z)w_2(x)$



Esercizio

- Dire se i seguenti due schedule sono view-equivalenti o conflict-equivalenti o nessuna delle due cose:
  - $S_1 = w_2(x)r_2(x)w_1(x)r_1(x)w_2(y)r_2(y)w_1(x)w_2(z)$
  - $S_2 = w_1(x)r_1(x)w_2(x)r_2(x)w_1(x)w_2(y)r_2(y)w_2(z)$

- Soluzione:
  - Sono view-equivalenti ma non conflict-equivalenti
  - VSR ma non CSR, schedule seriale equivalente $T_2\ T_1$

Verifica della Conflict-Serializzabilità

- **Anche la conflict-serializabilità**, pur più rapidamente verificabile (l'algoritmo, con opportune strutture dati richiede **tempo lineare**), è **inutilizzabile in pratica**
- La tecnica sarebbe efficiente se potessimo **conoscere il grafo dall'inizio**, ma così non è: uno scheduler deve operare "**incrementalmente**", cioè ad ogni richiesta di operazione decidere se eseguirla subito oppure fare qualcos'altro; non è praticabile mantenere il grafo, aggiornarlo e verificarne l'aciclicità ad ogni richiesta di operazione
- Inoltre, la tecnica **si basa** sull'ipotesi di **commit-proiezione**
- **In pratica**, si utilizzano tecniche che
  - garantiscono la conflict-serializzabilità **senza** dover **costruire il grafo**
  - **non richiedono** l'ipotesi della **commit-proiezione**

Lock

- Principio:
  - Tutte le **letture** sono **precedute** da `lock` e **seguite** da `unlock`
  - Tutte le **scritture** sono **precedute** da `lock` e **seguite** da `unlock`
- Il ***lock manager*** riceve queste richieste dalle transazioni e le **accoglie** o **rifiuta**

Lock condiviso ed esclusivo

- Per **aumentare la concorrenza** è possibile avere lock di tipo diverso, **condiviso** o **esclusivo**, usati in momenti diversi sulla stessa risorsa.
- Principio:
  - Tutte le **letture** sono **precedute** da `r_lock` (**lock condiviso**) e **seguite** da `unlock`
  - Tutte le **scritture** sono **precedute** da `w_lock` (**lock esclusivo**) e **seguite** da `unlock`
- Quando una transazione **prima legge e poi scrive un oggetto**, può:
  - richiedere **subito** un **lock esclusivo**
  - chiedere **prima** un **lock condiviso** e **poi** un **lock esclusivo** (*lock escalation*)
- Il ***lock manager*** riceve queste richieste dalle transazioni e le **accoglie** o **rifiuta**, sulla base della **tavola dei conflitti**

Comportamento dello scheduler

- La politica dello scheduler è basata sulla **tavola dei conflitti**
- Il lock manager riceve richieste di lock dalle transazioni e **concede/rifiuta le richieste** sulla base dei lock **precedentemente concessi** ad altre transazioni
- Quando viene concesso il lock su una risorsa ad una transazione, si dice che la **risorsa** è **acquisita** dalla transazione
- Nel momento dell'unlock, la **risorsa** viene **rilasciata**

Tavola dei conflitti

- Permette di realizzare la politica per la gestione dei conflitti

Richiesta	**Stato della risorsa**		
	`free`	`r_locked`	`w_locked`
`r_lock`	OK → `r_locked`	OK → `r_locked`	NO - (`w_locked`)
`w_lock`	OK → `w_locked`	NO (`r_locked`)	NO - (`r_locked`)
`unlock`	ERROR	OK - dipende*	OK → **`free`**

* Un contatore tiene il conto del numero di "lettori"; la risorsa è rilasciata solo quando il contatore scende a zero

- Se la risorsa non è concessa, la **transazione** richiedente è **posta in attesa** (eventualmente in coda), fino a quando la risorsa non diventa disponibile
- Il lock manager gestisce una **tabella dei lock**, per ricordare la situazione

Locking a due fasi

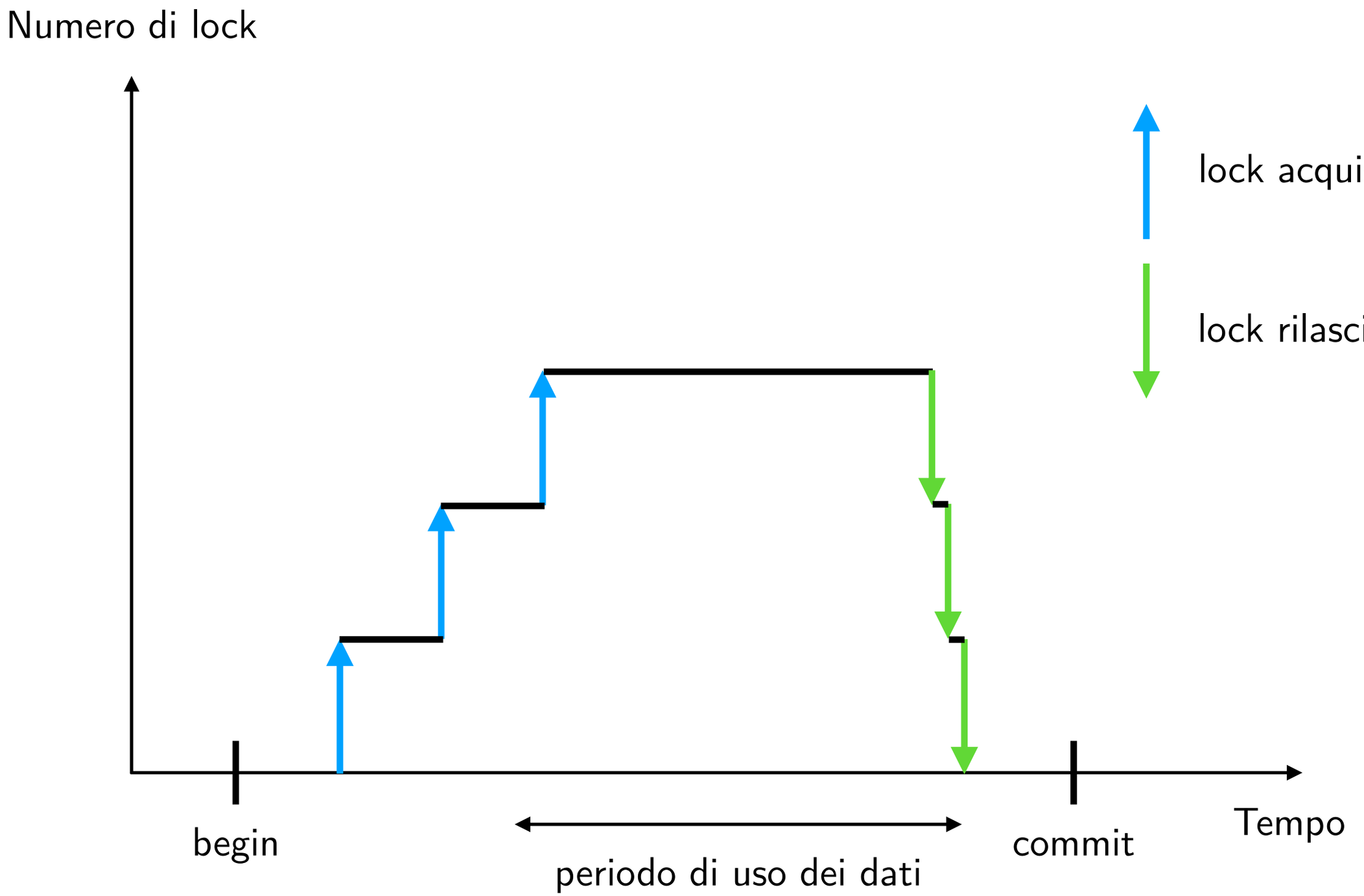
- Un **algoritmo di scheduling** usato da quasi tutti i sistemi commerciali
- Basato su **due regole**:
  - Se una transazione vuole leggere (scrivere) un dato, **prima deve acquisire un lock** condiviso (esclusivo) sul dato
    - Se la transazione **entra in conflitto** su un lock, **si pone in attesa**
  - Una transazione, **dopo aver rilasciato** un lock, **non può acquisirne** altri
- In altre parole:
  - Una transazione attraversa una prima fase di acquisizione di ciò che le serve
  - Poi comincia a rilasciare e non può acquisire altro

Esempio

$begin(T_1)$
$WL_1(B)$
$r_1(B)$
$B \leftarrow B - 50$
$w_1(B)$
$WL_1(A)$
$r_1(A)$
$UL_1(B)$
$A \leftarrow A + 50$
$w_1(A)$
$UL_1(A)$
$commit(T_1)$

$begin(T_1)$
$WL_1(B)$
$r_1(B)$
$B \leftarrow B - 50$
$w_1(B)$
$UL_1(B)$
$WL_1(A)$
$r_1(A)$
$A \leftarrow A + 50$
$w_1(A)$
$UL_1(A)$
$commit(T_1)$

Rappresentazione grafica del 2PL



2PL e CSR

- **Teorema**:
  - Ogni schedule 2PL è anche conflict-serializzabile, ma non necessariamente viceversa

- Contro-esempio per la non-necessità

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

  - Viola il 2PL
  - È conflict-serializzabile

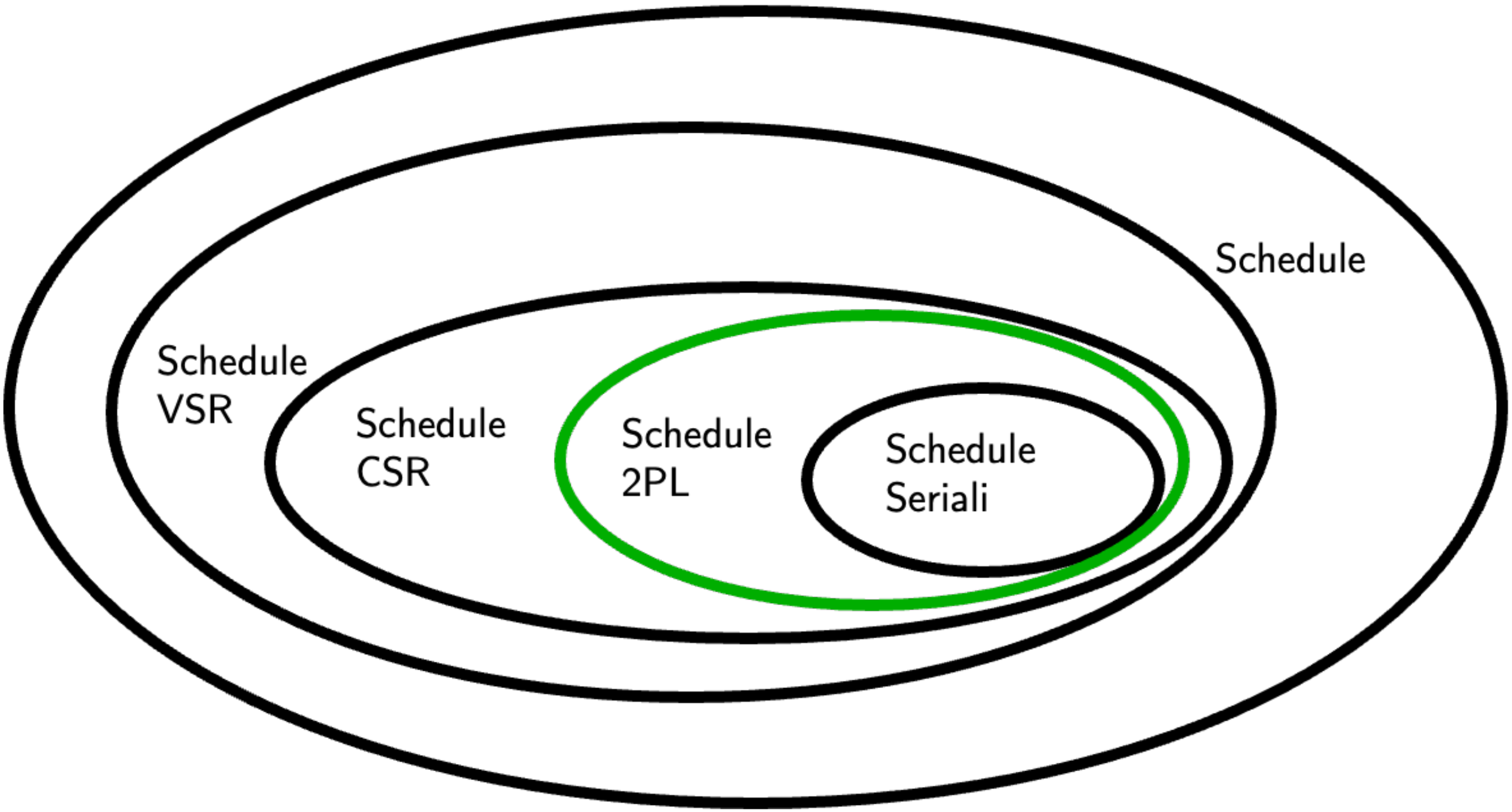
2PL e CSR

- **Dimostrazione**:
  - Sia S uno schedule 2PL
  - Consideriamo per ciascuna transazione nell'istante in cui ha tutte le risorse e sta per rilasciare la prima
  - Ordiniamo le transazioni in accordo con questo valore temporale e consideriamo lo schedule seriale corrispondente
- Vogliamo dimostrare che tale schedule è equivalente ad S:
  - Consideriamo un conflitto fra un'azione di t_i e un'azione di t_j con $i < j$; è possibile che compaiano in ordine invertito in S?
  - No, perché in tal caso t_j dovrebbe aver rilasciato la risorsa in questione prima della sua acquisizione da parte di t_i

2PL e CSR

- Dimostrazione alternativa:
  - Assumiamo, per assurdo, che esista uno schedule S tale che $S \in$ **2PL** e $S \notin CSR$.
  - Da $S \notin CSR$ segue che il grafo dei conflitti per S contiene un ciclo $t_1, t_2, \ldots, t_k, t_1$.
- Se esiste un arco (conflitto) tra t_1 e t_2, significa che esiste una risorsa x su cui si verifica il conflitto: t_2 può procedere solo se t_1 rilascia il lock su x così che t_2 lo può acquisire.
- Così avanti fino al conflitto tra t_k e t_1: t_1 deve acquisire il lock rilasciato da t_k, ma t_1 ha già rilasciato un lock per farlo acquisire da t_2 e quindi t_1 non rispetta il 2PL.

VSR, *CSR*, **2PL**


Schedule
Schedule
VSR
Schedule
CSR
Schedule
2PL
Schedule
Seriali

2PL e anomalie

- È facile vedere che **2PL risolve** le anomalie di **perdita di aggiornamento**, di **aggiornamento fantasma** e di **letture inconsistenti**
- Però **2PL presenta altre anomalie**:
  - ***Cascading rollback***: il fallimento di una transazione che ha scritto una risorsa deve causare il fallimento di tutte le transazioni che hanno letto il valore scritto
  - ***Deadlock*** (**attese incrociate** o **stallo**): due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra.
    - In generale, la **probabilità** di deadlock è **bassa**, **ma non nulla**

Esempio di *cascading rollback*

$begin(T_1)$
$WL_1(A)$
$r_1(A)$
$RL_1(B)$
$r_1(B)$
$w_1(A)$
$UL_1(A)$
$abort(T_1)$

$begin(T_2)$
$WL_2(A)$
$r_2(A)$
$w_2(A)$
$UL_2(A)$
$\ldots$

$begin(T_3)$
$RL_3(A)$
$r_3(A)$
$\ldots$

Esempio di *cascading rollback*

$begin(T_1)$
$WL_1(A)$
$r_1(A)$
$RL_1(B)$
$r_1(B)$
$w_1(A)$
$UL_1(A)$
$abort(T_1)$

$begin(T_2)$
$WL_2(A)$
$r_2(A)$
$w_2(A)$
$UL_2(A)$
...

$begin(T_3)$
$RL_3(A)$
$r_3(A)$
...

Quando T_1 fallisce, il fallimento
si deve trasmettere a T_2 e T_3

Esempio di *deadlock*

$begin(T_1)$
$WL_1(B)$
$r_1(B)$
$B \leftarrow B - 50$
$w_1(B)$

$begin(T_2)$
$RL_2(A)$
$r_2(A)$
$RL_2(B)$
$wait\ T_1$

$WL_1(A)$
$wait\ T_2$
$r_1(B)$
$UL_1(B)$

Locking a 2 fasi stretto

- Condizione aggiuntiva:
  - I lock possono essere **rilasciati solo dopo il commit**
- Elimina il rischio di letture sporche e quindi di rollback in cascata
- Supera la necessità dell'ipotesi di **commit-proiezione**