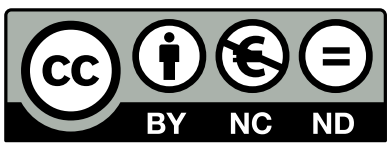
Note sulla Gestione della Concorrenza

Nicola Tonellotto
nicola.tonellotto@unipi.it

Version date: 14 maggio 2025



Queste note sono state preparate per gli studenti del Corso di Laurea in Ingegneria Informatica dell'Università di Pisa. Il materiale è fortemente ispirato dal libro "*Transactional Information Systems. Theory, Algorithms, and the Practice of Concurrency Control and Recovery*" (2001) di Gerhard Weikum e Gottfried Vossen, edito da Morgan Kaufmann.

Sono consapevole che questo documento contiene errori, e sono felice di migliorarli. Contattatemi pure per commenti, suggerimenti e per correzioni.

Indice

Pagina intenzionalmente bianca.

Introduzione

Una base di dati (*database*, DB) è un modello di alcuni aspetti della realtà progettato per fornire informazioni su quegli aspetti della realtà rappresentati. Nello schema di una base di dati sono definite delle regole, chiamate *vincoli di integrità*, che devono essere soddisfatte da tutti gli stati della base di dati. Se tutti i vincoli di integrità di una base di dati sono soddisfatti, essa è in uno *stato consistente*. Quando la realtà cambia, il modello della realtà nella base di dati deve cambiare di conseguenza. Questo cambiamento è implementato da un'applicazione, chiamata *transazione*, che modifica lo stato della base di dati, partendo da uno stato consistente e terminando in uno stato consistente. In tal caso l'esecuzione dell'operazione è corretta, cioè la transazione è *consistente*.
Una transazione è un'astrazione di programmazione che raggruppa un insieme di operazioni sulla base di dati. Dal punto di vista di un programmatore una transazione corrisponde all'*esecuzione di un programma*: diverse esecuzioni dello stesso programma producono diverse transazioni.
I sistemi di gestione delle basi di dati (*database management systems*, DBMS) devono implementare meccanismi per garantire che una transazione sia consistente anche quando molteplici transazioni sono eseguite concorrentemente, permettendo agli utenti di assumere che ogni loro applicazione sulla base di dati sia eseguita senza interferenze con altre applicazioni in esecuzione contemporaneamente.
Una transazione ha le seguenti proprietà:

- *atomicità*: solo le transazioni che terminano con successo (*committed*) modificano la base di dati; se una transazione è interrotta a causa di un errore o di un guasto (*aborted*), lo stato della base di dati non deve cambiare, come se nessuna delle operazione della transazione interrotta fosse mai avvenuta.
- *isolamento*: quando una transazione è eseguita contemporaneamente ad altre, lo stato finale della base di dati deve essere lo stesso che otterrebbe se la transazione fosse eseguita da sola.
- *durabilità*: gli effetti delle transazioni terminate con successo devono sopravvivere ai guasti.

Queste proprietà sono garantite dal DBMS. Se sono presenti i vincoli di integrità e se tutte le transazioni sono consistenti, l'esecuzione concorrente delle transazione in un DBMS garantisce la proprietà di *consistenza* grazie alla proprietà di isolamento. Queste quattro proprietà delle transazioni (atomicità, consistenza, isolamento e durabilità) sono spesso identificate con l'acronimo ACID. La componente di un DBMS che garantisce la proprietà di isolamento è detta *gestore della concorrenza*, mentre la componente di un DBMS che garantisce atomicità e durabilità è detta *gestore dell'affidabilità*.

1 Transazioni

Per i nostri scopi, indicheremo con le lettere $x, y, z, \ldots$ gli oggetti, indivisibili e tra loro disgiunti, di una base di dati D.
Una transazione, indicata con le lettere $T, T_1, T_2, \ldots$, è una *sequenza*, cioè un insieme ordinato di elementi, composta da operazioni elementari di lettura e scrittura e da operazioni di transazione.

1.1 Definizione *Operazioni elementari*

Assumiamo D sia una base di dati. Si definisce *operazione elementare* su un oggetto $x \in D$ un'operazione che si assume essere indivisibile.

Ogni operazione elementare è sia atomica che completamente isolata dalle altre operazioni.
Noi ci limiteremo a trattare unicamente due tipi di operazioni elementari, corrispondenti alle letture e alle scritture sugli oggetti di una base di dati.

1.2 Definizione *Operazioni di lettura e scrittura*

Assumiamo D sia una base di dati. Su un oggetto $x \in D$ è unicamente possibile effettuare le operazioni elementari di *lettura*, indicata con $r(x)$, e *scrittura*, indicata con $w(x)$.

Si osservi che queste definizioni di operazioni sono in una forma di astrazione molto alta: l'implementazione all'interno di un calcolatore di tali operazioni richiede numerose azioni a più basso livello, ma per i nostri scopi ignoreremo tali ulteriori azioni. Inoltre stiamo implicitamente assumendo che la base di dati sia un insieme prefissato di oggetti indipendenti che possono essere unicamente letti o modificati, cioè ignoriamo le operazioni di inserimento e cancellazione degli oggetti. Non tratteremo queste operazioni per semplicità di presentazione.

1.3 Definizione *Operazioni di transazione*

Assumiamo T sia una transazione. Le operazioni di transazione sono:

- *begin* $b(T)$: indica l'inizio di una transazione;
- *commit* $c(T)$: indica la terminazione con successo della transazione;
- *abort* $a(T)$: indica la terminazione di una transazione a causa di un guasto.

L'operazione di *begin* di una transazione deve essere eseguita a partire da una base di dati in uno stato consistente (*consistenza*). L'operazione di *commit* di una transazio-

ne, dopo aver effettuato tutte le operazioni elementari, deve garantire che la base di dati sia in uno stato consistente (*consistenza)* e che lo stato sia memorizzato in modo persistente (*durabilità)*. L'operazione di *abort* di una transazione indica che tutte le operazioni elementari della transazione devono essere annullate, come se la transazione non fosse mai avvenuta (*atomicità*).
Siamo ora in grado di definire cosa sia una transazione dal punto di vista dal DBMS.

1.4 Definizione *Transazione*

Una *transazione* T è una sequenza di operazioni di lettura e scrittura che inizia con l'operazione di *begin* $b(T)$ e che termina con l'operazione di *commit* $c(T)$ oppure con l'operazione di *abort* $a(T)$.

Tipicamente le operazioni di transazione non sono esplicitamente indicate dal programmatore; *begin* e *commit* sono aggiunte automaticamente dal DBMS, mentre *abort* è generato automaticamente dal gestore della concorrenza.
Indicheremo la generica transazione T con:

$$T = p_1 \cdots p_n,$$

dove p_i è una operazione elementare, cioè $p_i \in \{r(x), w(x)\}$ per un qualche oggetto $x \in D$. Per identificare meglio le letture e e le scritture su x di una transazione T_i scriveremo $r_i(x)$ e $w_i(x)$, rispettivamente. Assumiamo implicitamente che una qualsiasi transazione T_i includa, anche se non specificata, un'operazione di *begin* b_i prima della prima operazione di lettura o scrittura della transazione e una operazione di *commit* c_i dopo l'ultima operazioni di letture o scrittura della transazione. D'ora in avanti fare l'ipotesi, detta di *commit-proiezione*, che nessuna transazione fallisca[1].
Un esempio di transazione è il seguente:

$$T_1 = r_1(x)w_1(x)r_1(y)r_1(x)w_1(x).$$

In questa transazione l'oggetto x è letto e scritto due volte. Senza perdita di generalità, è ragionevole supporre che in una transazione ogni oggetto sia letto e scritto al più una volta. Nel seguito assumeremo che questo sia sempre vero [2].
Per riassumere, abbiamo fatto le seguenti ipotesi semplificative:

- la base di dati è un insieme prefissato di oggetti indipendenti che possono essere unicamente letti o modificati;

[1] La gestione delle transazioni che falliscono è demandata al gestore dell'affidabilità.

[2] Nel caso generale basta usare $r_{ij}(x)$ al posto di $r_i(x)$ e $w_{ij}(x)$ al posto di $w_i(x)$ per indicare con l'indice j più occorrenze della stessa operazione elementare sullo stesso oggetto.

- una transazione legge o modifica un dato oggetto al più una volta;
- tutte le transazione terminano con successo.

Adesso vediamo come il gestore della concorrenza di un DBMS deve eseguire le transazioni.

2 Gestione delle Transazioni

Il gestore della concorrenza deve garantire che le transazioni eseguite da un DBMS siano gestite in modo sicuro e corretto, mantendo la base di dati in uno stato consistendo e rispettando le proprietà ACID.
Le transazioni in esecuzione su un DBMS non sono eseguite una dopo l'altra, operazione dopo operazione. Sebbene un'esecuzione sequenziale sia semplice da implementare e garantisca la consistenda della base di dati, essa risulta inefficiente, perchè il DBMS non sfrutta pienamente le risorse di calcolo disponibili.
Nella realtà, le transazioni in un DBMS sono eseguite in parallelo: più transazioni sono attive nello stesso momento sulla base di dati, ovvero operazioni, cioè letture e scritture, di diverse transazioni sono eseguite simultaneamente su dati potenzialmente condivisi, anche se ciò non implica necessariamente che tutte le operazioni siano svolte contemporaneamente a livello hardware.
Eseguire le transazioni in parallelo offre diversi vantaggi, permettendo di:

- massimizzare le prestazioni del sistema;
- ridurre i tempi di attesa degli utenti;
- bilanciare carichi di lavoro diversi;
- gestire grandi volumi di dati e richieste simultanee.

Operativamente, il gestore delle transazioni riceve richieste SQL decomposte in operazioni di lettura e scrittura, e le esegue in parallelo. Senza nessun meccanismo di controllo della concorrenza, l'esecuzione concorrente di tali operazioni può generare delle *anomalie*, cioè dei problemi, tra operazioni di lettura e scrittura di diverse transazioni.
Vediamo alcuni esempi classici di problemi che possono sorgere tra operazioni di lettura e scrittura di diverse transazioni senza la presenza di opportuni meccanismi di gestione della concorrenza.

Perdita di aggiornamento. Supponiamo di avere due transazioni T_1 e T_2:

$$T_1 = r_1(x)w_1(x),$$
$$T_2 = r_2(x)w_2(x),$$

che leggono e scrivono sullo stesso oggetto x, il cui valore iniziale è 2.

Per garantire la proprietà di isolamento, le due transazioni dovrebbero essere eseguite in sequenza, cioè T_1T_2 oppure T_2T_1; in entrambi i casi, il valore finale dell'oggetto sarebbe 4. Tuttavia potrebbe succedere che le operazioni, senza controllo, possano essere eseguite nel seguente ordine:

T_1	T_2
b_1	
$r_1(x)$	
$x \leftarrow x+1$	
	b_2
	$r_2(x)$
	$x \leftarrow x+1$
$w_1(x)$	
c_1	
	$w_2(x)$
	c_2

In questo caso, la transazione T_2 leggerebbe il valore di x con $r_2(x)$ prima che la scrittura $w_1(x)$ di T_1 abbia avuto luogo, andando a scrivere nella base di dati con $w_2(x)$ il valore 3, lasciando quindi la base di dati in uno stato inconsistente.

Lettura sporca. Supponiamo di avere due transazioni T_1 e T_2:

$$T_1 = r_1(x)w_1(x),$$
$$T_2 = r_2(x),$$

che sono eseguite nel seguente ordine:

T_1	T_2
b_1	
$r_1(x)$	
$x \leftarrow x+1$	
$w_1(x)$	
	b_2
	$r_2(x)$
a_1	
	c_2

In questo caso la transazione T_1 termina a causa di un guasto, e le sue operazioni sono annullate. Tuttavia la transazione T_2 ha letto con $r_2(x)$ uno stato intermedio "sporco" dell'oggetto x, che non è consistente con lo stato della base di dati al termine dell'esecuzione delle due transazioni.

Letture inconsistenti. Supponiamo di avere due transazioni T_1 e T_2:

$$T_1 = r_1(x)r_1(x),$$
$$T_2 = r_2(x)w_2(x).$$

In questo caso, si noti che la transazione T_1 legge due volte lo stesso oggetto. Supponiamo che le transazioni siano eseguite nel seguente ordine:

T_1	T_2
b_1	
$r_1(x)$	
	b_2
	$r_2(x)$
	$x \leftarrow x + 1$
	$w_2(x)$
	c_2
$r_1(x)$	
c_1	

In questo caso la transazione T_1 legge due valori diversi dell'oggetto x.

Aggiornamenti fantasma. Supponiamo di avere due oggetti y e z della base di dati con vincolo di integrità $y + z = 1000$. Supponiamo di avere due transazioni T_1 e T_2:

$$T_1 = r_1(y)r_1(z),$$
$$T_2 = r_2(y)r_2(z)w_2(y)w_2(z),$$

che sono eseguite nel seguente ordine:

T_1	T_2
b_1	
$r_1(y)$	
	b_2
	$r_2(y)$
	$y \leftarrow y - 100$
	$r_2(z)$
	$z \leftarrow z + 100$
	$w_2(y)$
	$w_2(z)$
	c_2
$r_1(z)$	
$s \leftarrow y + z$	
c_1	

Durante l'esecuzione della transazione T_2, la variable locale s assume il valore 1100, non consistente con il vincolo di integrità.

È quindi necessario avere dei meccanismi per analizzare l'esecuzione di un insieme di transazioni concorrenti, per poter rapidamente identificare sequenze di operazioni elementari che causano anomalie.
La componente del sistema di gestione della concorrenza che tiene traccia di tutte le operazioni elementari richieste ed eseguite sulla base di dati dalle transazioni, e che ha il compito di decidere se accettare o rifiutare le operazioni che vengono via via richieste si chiama *scheduler*. Tali decisioni devono garantire che non si creino anomalie, e che il mondo esterno abbia sempre l'impressione che l'esecuzione delle transazioni sia avvenuta in modo seriale, secondo un qualche ordinamento delle transazioni.

3 Schedule

Per trattare formalmente l'esecuzione concorrente di un insieme di transazioni è necessario definire il concetto di *schedule*, che rappresenta una possibile esecuzione di operazioni di transazioni concorrenti.

3.1 Definizione *Schedule*

Sia $\mathcal{T} = \{T_1, \ldots, T_n\}$ un insieme di transazioni. Uno *schedule* S su $\mathcal{T}$ è una sequenza di operazioni tali che:

- le operazioni di S sono quelle di $T_1, \ldots, T_n$;
- S preserva l'ordinamento tra le operazioni appartenenti alla stessa transazione.

Per esempio, supponendo di avere tre transazioni T_1, T_2 e T_3:

$$T_1 = r_1(x)w_1(x)w_1(y),$$
$$T_2 = r_2(x)w_2(y),$$
$$T_3 = r_3(x)w_3(x),$$

un possibile schedule S è il seguente:

$$S = r_1(x)r_2(x)r_3(x)w_1(x)w_2(y)w_1(y)w_3(x).$$

Un insieme molto importante di schedule sono gli schedule seriali.

3.2 Definizione *Schedule seriali*

Sia $\mathcal{T} = \{T_1, \ldots, T_n\}$ un insieme di transazioni. Uno schedule S su $\mathcal{T}$ è uno *schedule seriale* se, per ogni coppia di transazioni $T_i, T_j \in \mathcal{T}$, tutte le operazioni di T_i sono eseguite prima di qualsiasi operazione di T_j, o viceversa.

Per le transazioni dell'esempio precedente, un possibile schedule seriale è il seguente:

$$r_2(x)w_2(y)r_3(x)w_3(x)r_1(x)w_1(x)w_1(y)$$

Si noti che, con queste tre transazioni, è possibile avere 6 schedule seriali:

$$T_1T_2T_3, \quad T_1T_3T_2, \quad T_2T_1T_3, \quad T_2T_3T_1, \quad T_3T_1T_2, \quad T_3T_2T_1.$$

Il nostro scopo è di progettare dei criteri di correttezza per gli schedule, in modo tale che lo scheduler possa accettare o meno l'esecuzione di un particolare schedule.
Assumendo che gli schedule debbano mantenere la base di dati in uno stato consistente, e che ogni singola transazione sia capace di assicurarlo, è ragionevole concludere, per induzione, gli schedule seriali siano corretti. Per questioni di prestazioni, uno scheduler non può accettare solo schedule seriali. Tuttavia, è possibile definire schedule non seriali ma corretti, detti *schedule serializzabili*.

3.3 Definizione *Schedule serializzabili*

Uno schedule di un insieme di transazioni è uno *schedule serializzabile* se la sua esecuzione produce lo stesso risultato di uno schedule seriale sulle stesse transazioni.

La relazione tra tutti i possibili schedule, schedule serializzabili e schedule seriali è illustrata in Figura 1.

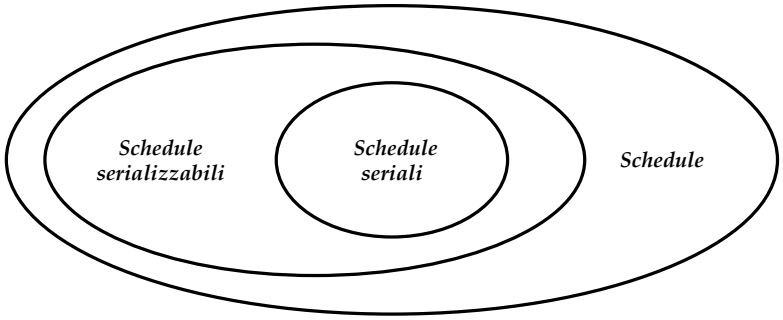


Figura 1: Insiemi di schedule, schedule serializzabili e schedule seriali.

La definizione di schedule serializzabile è inutile nella pratica perchè non ci fornisce alcun metodo operativo per identificarli. È necessario introdurre un concetto di *equivalenza*, per poter determinare se un dato schedule è equivalente o meno a uno schedule seriale, cioè se è serializzabile, secondo un'equivalenza che ancora non conosciamo. Quindi quello che faremo nel seguito sarà:

- definire delle opportune equivalenze fra schedule;
- definire uno schedule serializzabile secondo la classe di equivalenza.

Verificare che uno schedule sia serializzabile secondo una classe di equivalenza deve essere efficiente, in modo che lo scheduler non impeghi troppo tempo a decidere a riguardo.

4 *View* Equivalenza

Siccome due schedule equivalenti devono lasciare la base di dati nello stesso stato consistente, diamo un nome preciso alle ultime operazioni di scrittura di uno schedule.

4.1 Definizione *Insieme "scritture-finali"*

Dato uno schedule S, il suo insieme *scritture-finali* $\mathsf{Finali}(S)$ è composto da tutte le ultime scritture $w_i(x)$ di S per un qualsiasi oggetto x della base di dati.

Per esempio, considerando il seguente schedule:

$$S = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x),$$

le sue scritture finali formano il seguente insieme:

$$\mathsf{Finali}(S) = \{w_1(x), w_1(y), w_2(z)\}.$$

È ragionevole aspettarsi che due schedule equivalenti abbiano le stesse scritture finali. Nelle applicazioni reali, le transazioni leggono dalla base di dati dei valori che saranno usati in altri programmi. È altresì ragionevole supporre che due schedule equivalenti leggano, e quindi possano comunicare all'esterno, gli stessi dati. I valori letti direttamente dallo stato iniziale della base di dati sono per definizione consistenti, quindi non pongono problemi. Tuttavia, dopo che un oggetto è stato scritto durante uno schedule, le cose cambiano. Infatti se una lettura avviene dopo che un dato è modificato, è importante tener presente quale è stata l'ultima scrittura su tale oggetto.
Definiamo quindi il concetto di letture precedute da una scrittura in un dato schedule.

4.2 Definizione *Relazione "legge-da"*

Dato uno schedule S, la sua relazione *legge-da* $\mathsf{LeggeDa}(S)$ è composta da tutte le coppie di operazioni $(w_i(x), r_j(x))$ tra le operazioni di S con $i \neq j$ tali che:

- $w_i(x)$ precede $r_j(x)$, cioè la lettura avviene dopo una scrittura sullo stesso oggetto di transazioni diverse;
- non esiste alcuna operazione $w_k(x)$ con $k \neq i$ tra $w_i(x)$ e $r_j(x)$, cioè dopo la scrittura non c'è nessun'altra scrittura prima della lettura.

Per esempio, nel seguente schedule S sono indicate, con delle frecce, le coppie che compongono la relazione $\mathsf{LeggeDa}(S)$:

$$S = \overbrace{w_0(x)\overbrace{r_2(x)r_1}(x)}w_2(x)r_0(x)$$
$$\mathsf{LeggeDa}(S) = \{(w_0(x)r_2(x), w_0(x)r_1(x), w_2(x)r_0(x)\}$$

Si noti che le due operazioni di ogni elemento di una relazione *legge-da* devono avvenire sullo stesso oggetto, ma non necessariamente due coppie diverse devono coinvolgere lo stesso oggetto, come nel seguente esempio:

$$S = \overbrace{w_0(x)r_1}(x)w_1(x)\overbrace{w_1(y)r_2}(y)$$
$$\mathsf{LeggeDa}(S) = \{(w_0(x)r_1(x), w_1(y)r_2(y)\}$$

Siamo ora in grado di fornire una prima relazione di equivalenza fra schedule.

4.3 Definizione *View Equivalenza*

Due schedule S_1 e S_2 sono detti *view-equivalenti*, in simboli $S_1 \approx_v S_2$, se hanno le stesse operazioni, la stessa relazione *legge-da*, cioè se $\mathsf{LeggeDa}(S_1) = \mathsf{LeggeDa}(S_2)$, e lo stesso insieme *scritture-finali*, cioè se $\mathsf{Finali}(S_1) = \mathsf{Finali}(S_2)$.

Per esempio, consideriamo i due schedule seguenti:

$$S_1 = w_0(x)r_2(x)r_1(x)w_2(x)w_2(z),$$
$$S_2 = w_0(x)r_1(x)r_2(x)w_2(x)w_2(z).$$

Lo schedule S_1 è *view*-equivalente allo schedule S_2? Abbiamo

$$\mathsf{LeggeDa}(S_1) = \{w_0(x)r_2(x), w_0(x)r_1(x)\} \qquad \mathsf{Finale}(S_1) = \{w_2(x), w_2(z)\}$$
$$\mathsf{LeggeDa}(S_2) = \{w_0(x)r_1(x), w_0(x)r_2(x)\} \qquad \mathsf{Finale}(S_2) = \{w_2(x), w_2(z)\}$$

quindi possiamo concludere che S_1 è *view*-equivalente a S_2.
Si noti che, nell'esempio precedente, lo schedule S_2 è uno schedule seriale. Quindi siamo in grado ora di definire chiaramente cosa sia uno schedule serializzabile rispetto alla *view* equivalenza.

4.4 Definizione *Schedule view-serializzabile; VSR*

Uno schedule S è *view*-serializzabile se è *view*-equivalente a un qualche schedule seriale con le stesse transazioni di S.
L'insieme di tutti gli schedule *view*-serializzabili è indicato con VSR.

Possiamo sostituire la Figura 1 con la Figura 2.

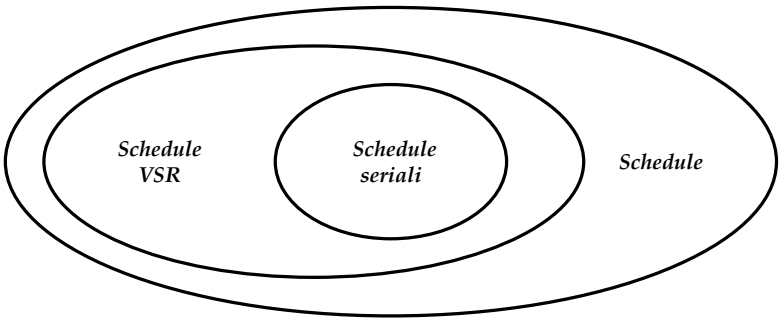


Figura 2: Insiemi di schedule, schedule VSR e schedule seriali.

Consideriamo i seguenti schedule:

$$S_1 = r_1(x)r_2(x)w_1(x)w_2(x),$$
$$S_2 = r_1(x)r_2(x)w_2(x)r_1(x),$$
$$S_3 = r_1(x)r_1(y)r_2(z)r_2(y)w_2(y)w_2(z)r_1(z).$$

Tutti convolgono due transazioni T_1 e T_2, quindi possono essere *view*-equivalenti unicamente a due schedule seriali T_1T_2 o T_2T_1. È facile verificare che nessuno degli schedule S_1, S_2 e S_3 è *view*-equivalente ad almeno uno schedule seriale, quindi non sono *view*-serializzabili. A un esame più attento, si può notare che S_1 è un'anomalia di perdita di aggiornamento, S_2 è un'anomalia di lettura inconsistente, e S_3 è un'anomalia di aggiornamento fantasma.
Mentre la verifica della *view*-equivalenza tra due schedule dati ha complessità polinomiale, decidere sulla *view*-serializzabilità di uno schedule è un problema NP-completo. Intuitivamente infatti è necessario calcolare la *view*-equivalenza tra lo schedule e tutti i possibili schedule seriali ottenibili dalle stesse transazioni, il cui numero cresce esponenzialmente con il numero di transazioni.
Ne consegue che il concetto di *view*-serializzabilità non sia utilizzabile nella pratica per costruire uno scheduler efficiente. Per far ciò dobbiamo definire una relazione di equivalenza fra schedule più ristretta, ma che dia luogo a una serializzibilità più veloce da verificare.

5 *Conflict* Equivalenza

Come abbiamo già intuito, i problemi con transazioni concorrenti sorgono a causa delle operazioni di scrittura. Diamo una definizione formale di questi problemi.

5.1 Definizione *Operazioni in conflitto*

Due operazioni di transazioni diverse sono *in conflitto* se accedono allo stesso oggetto e almeno una delle due è una scrittura.

In base alla definizione possono esistere conflitti di lettura-scrittura e conflitti di scrittura-scrittura. Per esempio, se abbiamo:

$$S = w_1(x)r_1(y)w_1(y)w_2(x)w_2(y),$$

le operazioni sui singoli oggetti x e y, seguendo l'ordine di S, sono:

$$x : w_1(x)w_2(x),$$
$$y : r_1(y)w_1(y)w_2(y),$$

e quindi abbiamo i seguenti conflitti:

$$w_1(x)w_2(x),\ r_1(y)w_2(y),\ w_1(y)w_2(y).$$

Possiamo usare il concetto di conflitto per definire una nuova relazione di equivalenza fra schedule.

5.2 Definizione *Conflict Equivalenza*

Due schedule S_1 e S_2 sono detti *conflict-equivalenti*, in simboli $S_1 \approx_c S_2$, se hanno le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi.

Si noti che, nella definizione di *conflict* equivalenza, *l'ordinamento delle operazioni in conflitto è significativa*, cioè $r_i(x)w_j(x) \neq w_j(x)r_i(x)$.
Verificare la *conflict* equivalenza di due schedule è semplice: basta calcolare i conflitti di entrambi gli schedule e verificare la loro uguaglianza, tenendo presente l'ordinamento delle operazioni.

5.3 Definizione *Schedule conflict-serializzabile; CSR*

Uno schedule S è *conflit*-serializzabile se è *conflit*-equivalente a un qualche schedule serial con le stesse transazioni di S.
L'insieme di tutti gli schedule *conflict*-serializzabili è indicato con CSR.

Consideriamo lo schedule:

$$S = r_1(x)r_2(x)r_1(z)w_1(x)w_2(y)r_3(z)w_3(y)w_3(z).$$

Analizziamo le operazioni sui singoli oggetti, per identificare i conflitti:

$$x : r_1(x)r_2(x)w_1(x) \rightarrow r_2(x)w_1(x)$$
$$y : w_2(y)w_3(y) \rightarrow w_2(y)w_3(y)$$
$$z : r_1(z)r_3(z)w_3(z) \rightarrow r_1(z)w_3(z)$$

Dai conflitti, si deduce che in un qualsiasi schedule seriale *conflict*-equivalente a S, T_2 deve precedere T_1 (a causa del conflitto $r_2(x)w_1(x)$) e T_3 (a causa del conflitto $w_2(y)w_3(y)$). Inoltre T_1 deve precedere T_3 (a causa del conflitto $r_1(z)w_3(z)$). Quindi l'unico schedule seriale potenzialmente *conflict*-equivalente a S risulta essere $S' = T_2T_1T_3$, cioè:

$$S' = r_2(x)w_2(y)r_1(x)r_1(z)w_1(x)r_3(z)w_3(y)w_3(z).$$

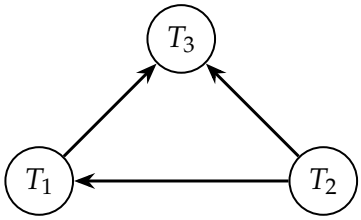
I conflitti ordinati di S' coincidono con quelli di S, quindi possiamo concludere che S è *conflict*-serializzabile.
La differenza fondamentale tra *view*-serializzabilità e *conflict*-serializzabilità consiste nel fatto che la *conflict*-serializzabilità può essere verificata in modo efficiente. Per fare ciò, dato uno schedule, dobbiamo construire il suo "grafo dei conflitti".

5.4 Definizione *Grafo dei conflitti*

Il *grafo dei conflitti* $G(S)$ di uno schedule S è un grafo orientato in cui:

- esiste un nodo per ogni transazione di S;
- esiste un arco orientato $T_i \rightarrow T_j$, con $i \neq j$ se e solo se in S esiste un'operazione di T_i precedente a T_j e con essa in conflitto.

Se prendiamo in esame lo schedule S dell'ultimo esempio, il suo grafo dei conflitti $G(S)$ risulta:



La proprietà fondamentale del grafo dei conflitti è data dal seguente teorema.

5.5 Teorema *Conflict serializzabilità*

Uno schedule S è *conflict*-serializzabile se e solo se il suo grafo dei conflitti $G(S)$ è aciclico.

La verifica dalle aciclicità di un grafo ha complessità lineare rispetto al numeri di nodi del grafo, e quindi la verifica di *conflict*-serializzabilità può essere svolta in modo efficiente.
È possibile dimostrare la seguente relazione tra gli insiemi di schedule VSR e CSR.

5.6 **Teorema** $CSR \subset VSR$

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

Un contro-esempio per la non-necessità è il seguente schedule:

$$r_1(x)w_2(x)w_1(x)w_3(x)$$

Questo schedule è *view*-equivalente allo schedule seriale

$$r_1(x)w_1(x)w_2(x)w_3(x)$$

ma non è *conflict*-serializzabile a causa dei conflitti $r_1(x)w_2(x)$ e $(x)w_2(x)w_1(x)$, che creano un ciclo tra i nodi T_1 e T_2 del corrispondente grafo dei conflitti.
Alla luce di questo teorema, possiamo sostituire la Figura 2 con la Figura 3.

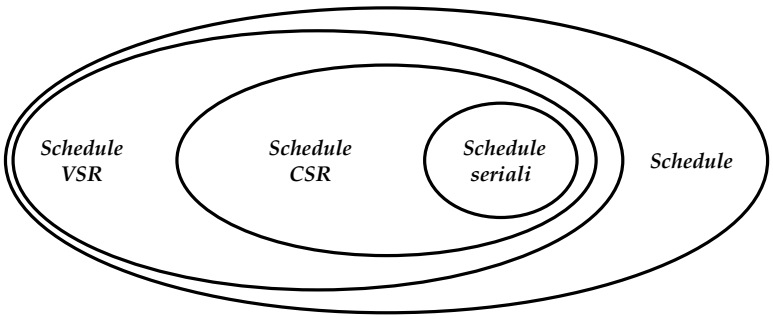


Figura 3: Insiemi di schedule, schedule VSR, schedule CSR e schedule seriali.

Anche la *conflict*-serializabilità, pur più rapidamente verificabile della *view*-serializzibilità, non è utilizzata nella pratica. Questo approccio sarebbe efficiente se potessimo conoscere il grafo dei conflitti fin 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 a ogni richiesta di operazione. Inoltre, nel caso di basi di dati distribuite su più calcolatori, creare e aggiornare un grafo in modalità distribuita per richiedere un tempo troppo elevato.
Tuttavia il concetto di *conflict*-serializzabilità è lo strumento principale per progettare algoritmi di scheduling per DMBS: qualsiasi schedule generato da questi algoritmi deve essere in CSR.

6 Locking

I più diffusi algoritmi di scheduling delle transazioni si basano sul concetto di *lock*.

6.1 Definizione *Lock*

Un *lock* su un oggetto x è un meccanismo di controllo della concorrenza che regola l'accesso simultaneo all'oggetto x.

I lock sono acquisiti (assegnati) e rilasciati (rimossi) dalle transazioni per mezzo dello scheduler. I lock possono essere di diversi tipi, a seconda del livello di accesso richiesto dalla transazione. Prima di eseguire una lettura sull'oggetto x, una transazione T_i deve acquisirne il *lock condiviso* o *read lock*; tale azione è indicata con $rl_i(x)$. Dopo aver eseguito la lettura, deve rilasciare il lock condiviso; tale azione è indicata con $ru_i(x)$. Prima di eseguire una scrittura sull'oggetto x, una transazione T_i deve acquisir il *lock esclusivo* o *write lock*; tale azione è indicata con $wl_i(x)$. Dopo aver eseguito la scrittura, deve rilasciare il lock condiviso; tale azione è indicata con $wu_i(x)$.
Senza nessun genere di controllo sull'assegnazione dei lock da parte dello scheduler, si possono verificare situazioni di conflitto tra i lock.

6.2 Definizione *Conflitto tra lock*

Due *lock* sullo stesso oggetto, acquisiti da due transazioni diverse, sono in conflitto se almeno uno dei due è un lock esclusivo.

Due transazioni che acquisiscono un lock in lettura sullo stesso oggetto non possono entrare mai in conflitto, in quanto leggono il contenuto dell'oggetto, senza mai modificarlo. Nel caso di lock esclusivo, invece, la transazione che lo ha acquisito può modificarne il contenuto, quindi nessun'altra transazione può leggerlo, men che meno modificarlo.
Un *locking scheduler* è uno scheduler che, oltre a ricevere le richieste di operazioni da parte di transazioni concorrenti, deve gestire anche i lock, controllando quando le transazioni acquisiscono e rilasciano i propri lock. A tal fine deve implementare un opportuno *algoritmo di locking*, per l'assegnamento e il rilascio dei lock
L'algoritmo di locking più comune nei DBMS commerciali è l'algoritmo di *locking a due fasi* (*two-phase locking*, 2PL), implementato da uno *scheduler 2PL*. Esso si basa sulle seguenti regole:

1. quando riceve un'operazione $p_i(x)$, lo scheduler controlla se il lock $pl_i(x)$ è in conflitto con qualche $ql_j(x)$ già acquisito da un'altra transazione sullo stesso oggetto. In caso affermativo, esso blocca l'operazione $p_i(x)$, mettendo T_i in atte-

sa finchè non riesce ad acquisite il lock di cui ha bisogno. In caso negativo, lo scheduler assegna $pl_i(x)$ alla transazione ed esegue l'operazione $p_i(x)$.

2. Una volta che lo scheduler ha assegnato un lock $pl_i(x)$ a T_i a seguito di un'operazione $p_i(x)$, non può rilasciarlo *almeno* fino a quando l'operazione $p_i(x)$ non è terminata.
3. Una volta che lo scheduler rilascia un lock per una transazione, esso non può più assegnare *nessun* lock, per *nessun* oggetto, a quella transazione.

La regola 1 impedisce a due transazioni di accedere all stesso oggetto causando un conflitto tra lock. In tal modo, le operazioni in conflitto sono schedulate nello stesso ordine in cui i lock corrispondenti sono assegnati. La regola 2 garantisce che un lock non sia mai rilasciato prima che la corrispondente operazione non sia conclusa. La regola 3, chiamata *regola delle due fasi*, è la caratteristica del locking in due fasi. A seguito di tale regola, ogni transazione può essere divisa in due fasi: la *fase crescente*, durante la quale acquisice i lock, e la *fase calante*, durante la quale rilascia i lock precedentemente acquisiti. Intuitivamente, la sua funzione è quella di garantire che tutte le *coppie* di operazioni in conflitto di due transazioni siano schedulate nello stesso ordine.
Consideriamo due transazioni $T_1 = r_1(x)w_1(y)$ e $T_2 = w_2(x)w_2(y)$ e il seguente schedule:

$$S_1 = rl_1(x)r_1(x)ru_1(x)wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y)wl_1(y)w_1(y)wu_1(y).$$

Lo schedule S_1 non è *conflict*-serializzabile: ignorando i lock, le operazioni risultano eseguite nel seguente ordine $r_1(x)w_2(x)w_2(y)w_1(y)$, causando un ciclo $T_1 \to T_2 \to T_1$ nel grafo dei conflitti. Il problema di S_1 è che T_1 rilascia il lock $ru_1(x)$ e, successivamente, acquisisce il nuovo lock $wl_1(y)$, violando la regola delle due fasi. Nel mezzo, un'altra transazione T_2 scrive sia su x che su y, causando il ciclo nel grafo dei conflitti. Se T_1 avesse rispettato la regola delle due fasi, T_2 non sarebbe riuscita a eseguire le operazioni come riportato in S_1.
Uno scheduler 2PL avrebbe schedulato le operazioni in S_1 come segue:

1. Inizialmente nessuna transazione possiede alcun lock.
2. Arriva l'operazione $r_1(x)$. Lo scheduler assegna il lock $rl_1(x)$ e permette l'esecuzione dell'operazione.
3. Arriva l'operazione $w_2(x)$. Lo scheduler non può assegnare il lock $wl_2(x)$ poichè causerebbe un conflitto con il lock $rl_1(x)$, quindi blocca l'operazione $w_2(x)$ e mette la transazione T_2 in una coda di attesa sull'oggetto x.
4. Arriva l'operazione $w_1(y)$. Lo scheduler assegna il lock $wl_1(y)$ e permette l'ese-

cuzione dell'operazione.

5. Non essendoci altre operazioni in T_1, lo scheduler riceve il commit per T_1 (non indicato esplicitamente nello schedule), e rilascia i lock $rl_1(x)$ e $wl_1(y)$.
6. Lo scheduler controlla se ci sono transazioni in attesa sugli oggetti i cui lock sono appena stati rilasciati. Trovando T_2 in attesa su x, assegna il lock $wl_2(x)$ e permette l'esecuzione dell'operazione $w_2(x)$.
7. Arriva l'operazione $w_2(y)$. Lo scheduler assegna il lock $wl_2(y)$ e permette l'esecuzione dell'operazione.
8. Non essendoci altre operazioni in T_2, lo scheduler riceve il commit per T_2 (non indicato esplicitamente nello schedule), e rilascia i lock $wl_2(x)$ e $wl_2(y)$.

Questo schedule può essere scritto come:

$$S_2 = rl_1(x)r_1(x)wl_1(y)w_1(y)ru_1(x)wu_1(y)wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y).$$

Ignorando i lock, le operazioni risultano eseguite nel seguente ordine $r_1(x)w_1(y)w_2(x)w_2(y)$, ed essendo uno schedule seriale, esso è *conflict*-serializzabile.
Nel seguito, indichiamo con $2PL$ l'insieme di tutti gli schedule producibili da uno scheduler 2PL. L'importanza di uno scheduler 2PL è data dal seguente teorema.

6.3 Teorema $2PL \subset CSR$

Ogni schedule producibile da uno scheduler 2PL è *conflict*-serializzabile, ma non necessariamente viceversa.

Un contro-esempio per la non-necessità è il seguente schedule:

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

Questo schedule è *conflict*-equivalente allo schedule seriale

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

ma non può essere prodotto da uno scheduler 2PL. Infatti tale scheduler assegnerebbe subito il lock $wl_1(x)$, casuando la messa in attesa di T_2 fino al termine dell'esecuzione di T_1, ed quindi eseguendo l'operazione $w_2(x)$ per ultima.
Praticamente tutte le implementazioni dello scheduler 2PL usano una variante, detta *2PL stretto*. La differenza tra il 2PL stretto e il 2PL risiede nel fatto che uno scheduler 2PL stretto deve rilasciare tutti i lock di una transazione contemporaneamente, al termine della transazione stessa.

Una ragione per adottare uno scheduler 2PL stretto è la seguente. Pensiamo a quando uno scheduler 2PL può rilasciare un lock sull'oggetto x. Per poterlo fare, lo scheduler deve sapere che: (i) la transazione ha acquisito tutti i lock di cui avrà mai bisogno, e (ii) la transazione non eseguirà più alcuna operazione che coinvolga l'oggetto x. Sicuramente le due condizioni si verificheranno quando la transazione terminerà, sia con successo sia con fallimento.

Implementazione. Una possibile implementazione dei meccanismi di lock è tramite la *tabella dei lock*, gestita da un componente del gestore della concorrenza chiamato *gestore dei lock*. Le operazioni messe a disposizione da tale tabella sono `lock(t,x,m)` e `unlock(t,x)`, dove `t` indica la transazione che acquisisce/rilascia il lock, `x` indica l'oggetto su cui si applica il lock, e `m` indica la modalità del lock, condivisio o esclusivo. L'esecuzione dell'operazione `lock(t,x,m)` prevede che il gestore dei lock cerchi di richiedere il lock specificato inserendo un elemento nella tabella dei lock. Se un'altra transazione ha già acquisito un lock in conflitto, allora il gestore dei lock inserisce la richiesta di lock in una lista di attesa per l'oggetto in questione. L'esecuzione dell'operazione `unlock(t,x)` rilascia il lock specificato, e lo assegna a una delle richieste di lock in attesa, se presenti, che risulta quindi essere sbloccata.
Queste operazioni sono invocate molto frequentemente, quindi la loro esecuzione deve essere estremamente rapida. La tabella dei lock è tipicamente implementata con una tabella hash, in quanto risulta essere estremamente veloce per la ricerca basata sul contenuto. Una coppia chiave-valore di tale tabella ha come chiave l'oggetto x, e come valore due liste: una prima lista di lock che sono stati concessi su x e una seconda lista di richieste di lock che sono in attesa. Ogni lock e richiesta di lock contiene l'identificatore della transazione e la modalità del lock.

Deadlock. Uno scheduler 2PL produce schedule *conflict*-serializzabili, quindi risolve tutte le anomalie risolte da schedule in CSR. Purtroppo, però, introduce una nuova anomalia, chiamata *stallo* o *deadlock*.
Supponiamo di avere due transazioni T_1 e T_2:

$$T_1 = r_1(x)w_1(y),$$
$$T_2 = w_2(y)w_2(x),$$

le cui operazioni arrivano allo scheduler nell'ordine $r_1(x)w_2(y)w_1(y)w_2(x)$.
Uno scheduler 2PL schedula le suddette operazioni come segue:

1. Inizialmente nessuna transazione possiede alcun lock.
2. Arriva l'operazione $r_1(x)$. Lo scheduler assegna il lock $rl_1(x)$ e permette l'esecuzione dell'operazione.

3. Arriva l'operazione $w_2(y)$. Lo scheduler assegna il lock $wl_2(y)$ e permette l'esecuzione dell'operazione.
4. Arriva l'operazione $w_1(y)$. Lo scheduler non può assegnare il lock $wl_1(y)$ poichè causerebbe un conflitto con il lock $wl_2(y)$, quindi blocca l'operazione $w_1(y)$ e mette la transazione T_1 in una coda di attesa sull'oggetto y.
5. Arriva l'operazione $w_2(x)$. Lo scheduler non può assegnare il lock $wl_2(x)$ poichè causerebbe un conflitto con il lock $rl_1(x)$, quindi blocca l'operazione $w_2(x)$ e mette la transazione T_2 in una coda di attesa sull'oggetto x.

Entrambe le transazioni T_1 e T_2 sono bloccate, e non possono procedere senza violare le regole dello scheduler 2PL. Uno scheduler 2PL deve quindi avere una strategia per identificare i *deadlock*, in modo che nessuna transazione possa essere bloccata per sempre.

Una strategia è il meccanismo di *timeout*. Se una transazione rimane in attesa di un lock più a lungo di un dato lasso di tempo prefissato, detto *timeout*, lo scheduler assume che sia in una situazione di deadlock, e abortisce la transazione. Certamente lo scheduler potrebbe essersi sbagliato, in quanto la transazione poteva essere in attesa a causa di un'altra transazione estremamente lunga, ma abortire una transazione non causa problemi di consistenza di una base di dati. Per evitare troppi errori è possibile impostare un grande valore di *timeout*, tuttavia correndo il rischio di perdere molto tempo in caso di presenza effettiva di *deadlock*.

Un'altra strategia consiste consiste nell'identificazione dei *deadlock*. Per poterlo fare, lo scheduler mantiene aggiornato un grafo diretto chiamato *grafo delle attese*. I nodi di questo grafo solo le transazioni in esecuzione; un arco $T_i \to T_j$ è presente nel grafo se e solo se la transazione T_i è in attesa che la transazione T_j rilasci qualche lock. Se nel grafo delle attese è presente un ciclo, le transazioni in esso coinvolte sono in *deadlock*. Quindi lo scheduler può identificare i *deadlock* controllando la presenza di cicli nel grafo delle attese. Quando ciò si verifica, lo scheduler sceglie una transazione da abortire, rimuovendone il nodo dal grafo.