Transazioni: view/conflict serializzabili/grafo dei conflitti

Si individua lo *schedule* come sequenza di operazioni di I/O di transazioni concorrenti. Ad esempio:

$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$

che va intesa come (lettura della transizione 1 su X, lettura della transizione 2 su z, scrittura della tr. 1 su x, scrittura della tr. 2 su z)

Ignoriamo quindi le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule (*commit-proiezione*). Il controllo della concorrenza e delle transazioni è dato dallo *scheduler*, eseguendo o riordinando le operazioni in un certo modo.
Esso può essere:

- seriale, transazioni separate una alla volta

  $S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$

- serializzabile, produce lo stesso risultato di uno schedule seriale

Se sono equivalenti, lo schedule può essere definito buono.
Lo schedule viene costruito nel mentre si eseguono le transazioni e quando una transazione fa il commit/abort, le operazioni sono rimosse dallo schedule e, se una di queste producesse uno schedule non serializzabile, l'operazione viene messa in attesa finché la sua esecuzione non viola la serializzabilità (magari venendo posticipata).
Si danno due definizioni preliminari:

- $r_i(x)$ ***legge-da*** $w_j(x)$ in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è $w_k(x)$ fra $r_i(x)$ e $w_j(x)$ in S
- $w_i(x)$ in uno schedule S è ***scrittura finale*** *se* è l'ultima scrittura dell'oggetto x in S

- Esempio: $S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  - $r_2(x)$ legge da $w_0(x)$
  - $r_1(x)$ legge da $w_0(x)$
  - *Scritture finali:* $w_0(x)$, $w_2(x)$, $w_2(z)$

Diremo che due schedule sono *view-equivalenti* $(S_i \approx_V S_i)$ se hanno la stessa relazione *legge-da* e le stesse scritture finali. Nell'esempio che segue, tutti leggono da w_0 e le scritture finali sono le stesse, quindi le schedule sono view-equivalenti:

$S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z) \approx_V$
$S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$

Uno schedule è *view-serializzabile* (VSR) se è view-equivalente ad un qualche schedule seriale.
Esempio: S_3 è view-serializzabile perché view-equivalente con S_4 che è seriale.
Le garanzie che fornisce sono l'assenza di:

- Perdita di Aggiornamento
  $S_7 : r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$
- Letture Inconsistanti
  $S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
- Aggiornamento Fantasma
  $S_9 : r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$

S_7, S_8, S_9 sono tutte non view-serializzabili.

Per esempio, posso decidere di fare tutte le operazioni della transazione 1, oppure tutte le operazioni della transazione 2. A tale scopo, se fossero equivalenti, si avrebbe lo stesso ordine di esecuzione su:

- $r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)$
- $r_2(x)\ w_2(x)\ r_1(x)\ w_1(x)$ (Questo legge (con r_1) da w_2 mentre S_7 no

Scritto da Gabriel

Inoltre, la verifica della view-equivalenza di due dati schedule è lineare sulla lunghezza dello schedule e decidere sulla view serializzabilità di uno schedule S è un problema "difficile" perchè occorre provare tutte i possibili schedule seriali, ottenuti per permutazioni dell'ordine delle transazioni (coefficiente binomiale).

Introduciamo poi la *conflict-serializzabilità*:

- Un'azione a_i è in *conflitto* con a_j ($i \neq j$), se operano sullo stesso oggetto e almeno una di esse è una scrittura. Due casi:
  - conflitto *read-write* (*rw* o *wr*)
  - conflitto *write-write* (*ww*).

Schedule conflict-equivalenti ($S_i \approx_C S_j$): includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi

Dunque, ad esempio qui le operazioni non devono essere invertite, in particolare:
S_1 : $r_1(x)$ $w_1(x)$ $w_2(x)$
S_2: $w_2(x)$ $r_1(x)$ $w_1(x)$
Quelli segnati in giallo rappresentano i conflitti, dato che operano tutti su *x*.

Ad esempio, anche $r_1(x)$ e $w_2(x)$ sono in conflitto.
Oppure similmente $w_2(x)$ e $r_1(x)$ sono in conflitto.

$$\begin{array}{lccc} & 1_a & & 2_a \\ S_1: & r_1(x) & w_1(x) & w_2(x) \\ S_2: & w_2(x) & r_1(x) & w_1(x) \\ & & 1_a & 2_a \end{array}$$

L'oggetto è la tupla; di fatto che l'operazione nella seconda riga, legge un valore diverso rispetto al valore prodotto in origine. Devono leggere gli stessi attributi sulla stessa tupla; se hanno un diverso attributo sono due oggetti diversi. Quindi semplicemente i conflitti devono essere nello stesso ordine di apparizione.

Uno schedule è *conflict-serializable* se è conflict-equivalente ad un qualche schedule seriale.
Inoltre, l'insieme degli schedule conflict-serializzabili si indica con **CSR.**
Inoltre:

- Ogni schedule conflict-serializzabile (CSR) è view-serializzabile (VSR)
- Ci sono schedule view-serializzabili (VSR) che non sono conflict-serializzabili (CSR)
- CSR implica VSR
- VSR non implica CSR

Esempio per dimostrare che VSR ⇏ CSR

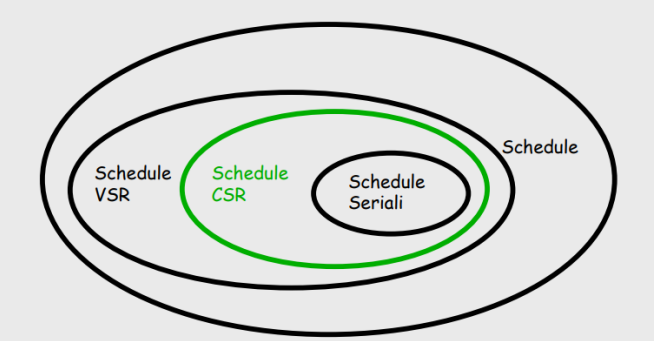
$S_1 : r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$

- VSR: View-equivalenza: $S_1 \approx_V r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
- non CSR
  - $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$ inverte $w_1(x)$ e $w_2(x)$
  - $w_2(x)\ r_1(x)\ w_1(x)\ w_3(x)$ inverte $r_1(x)$ e $w_2(x)$

Dato che le operazioni complessivamente compaiono nello stesso ordine, allora sono view-serializzabili. Si ha invece un ordine diverso rispetto ai conflitti, in quanto come spiegato qui si ha un diverso ordine di applicazione di letture e scritture che porta ad operare con valori diversi e anche inconsistenti

$w_2(x)\ r_1(x)\ w_1(x)\ w_3(x)$ inverte $r_1(x)$ e $w_2(x)$

Insiemisticamente parlando:

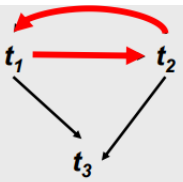


Citiamo i *grafi*, intesa come struttura tale da collegare i singoli punti nel piano con un insieme di archi e descrivere un ciclo se, partendo anche da un solo nodo *n*, è possibile seguire questi e tornare ad *n* percorrendo un arco in più.
Ci serve per creare un *grado dei conflitti* così generato:

- 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

Uno schedule è in CSR sse il grafo è aciclico.

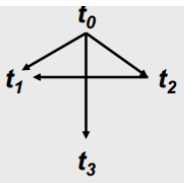
$r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$



Ciclo ⇒ non CSR

In questo secondo caso, si va da t_2 a t_1 in quanto la lettura/scrittura agisce su *x*

$w_0(x)\ r_1(x)\ w_0(z)\ r_2(x)\ r_3(z)\ w_3(z)\ w_1(x)$



Nessun Ciclo ⇒ CSR

Normalmente costa troppo la view-serializzabilità, ma non la conflict-serializzabilità, perché interessa trovare conflitti e capire se possono essere CSR e quindi risolvibili, piuttosto che esaminare tutte le singole operazioni per capire se l'ordine di letture/scritture è effettivamente quello indicato.

Supponendo di avere un sistema con 100 transazioni al secondo, ciascuna di durata di 5 secondi e che accede a 10 oggetti/pagine, leggendone 2 per secondo. In ogni secondo ci sono quindi 500 transazioni e quindi il sistema deve costruire un grafo con 500 nodi e almeno fino a 5000 archi.
In pratica se non con basi di dati "poco usate", questa situazione non è trattabile.

Si usano quindi i *lock*, per cui:

- Tutte le letture per una risorsa *x* sono precedute da *r_lock(x)* (lock condiviso) e seguite da unlock
- Tutte le scritture per una risorsa *x* sono precedute da *w_lock(x)* (lock esclusivo) e seguite da unlock

Scritto da Gabriel

Quando una transazione prima legge e poi scrive una risorsa *x* può:

- richiedere subito *w_lock(x)*
- chiedere prima *r_lock(x)* e poi *w_lock(x) (lock escalation)*

Si tiene quindi conto dei lock e dei possibili conflitti, in una apposita *tavola dei conflitti*. Per ogni risorsa:

- Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata quando il contatore scende a zero
- Un valore booleano tiene conto se c'è un *w_lock*

	libera	***r_locked***	***w_locked***
r_lock(x)	OK / r_locked conta(x)++	OK / r_locked conta(x)++	NO / w_locked
w_lock(x)	OK / w_locked	NO / r_locked	NO / w_locked
unlock(x)	error	OK / if (--conta(x)=0) libera else r_locked	OK / not w_locked

← libera

Va in *error* nel caso della terza casella della prima colonna perché la risorsa è già libera. Se la risorsa non è concessa, la transazione richiedente è posta in attesa (eventualmente in coda), fino a quando la risorsa non diventa disponibile.

Normalmente tutti i sistemi implementano un *locking a due fasi* (2PL), che garantisce "a priori" CSR sulla base di:

- fase crescente, acquisendo i lock necessari
- fase decrescente, perché i lock si rilasciano pian piano

Il deadlock si evita perché *una transazione, dopo aver rilasciato un lock, non può acquisirne altri*.
Si rimane in coda per tutte le richieste del lock; se non è fattibile eseguire quella richiesta, poi si rilascia se e quando serve.
Ritornando al problema dell'*aggiornamento fantasma,* cioè un dato che improvvisamente appare aggiornato e per t_1 non è coerente.

- Assumiamo vincolo $y + z = 1000$:

t_1	t_2
bot	
$r_1(y)$	
	bot
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = y + z$	
commit	

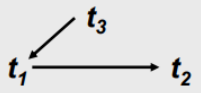
- $s = 1100$: il vincolo sembra non soddisfatto, t_1 vede un aggiornamento non coerente

t_1	t_2	x	y	z
bot		free	free	free
$r\_lock_1(x)$		1:read		
$r_1(x)$				
	bot			
	$w\_lock_2(y)$		2:write	
	$r_2(y)$			
$r\_lock_1(y)$			1:wait	
	$y = y - 100$			
	$w\_lock_2(z)$			2:write
	$r_2(z)$			
	$z = z + 100$			
	$w_2(y)$			
	$w_2(z)$			
	commit			
	$unlock_2(y)$		1:read	
$r_1(y)$				
$r\_lock_1(z)$				1:wait
	$unlock_2(z)$			1:read
$r_1(z)$				
	eot			
$s = x + y + z$				
commit				
$unlock_1(x)$		free		
$unlock_1(y)$			free	
$unlock_1(z)$				free
eot				

Attenzione alla relazione tra 2PL (Locking a due fasi) e CSR.

Consideriamo:

$S: r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$



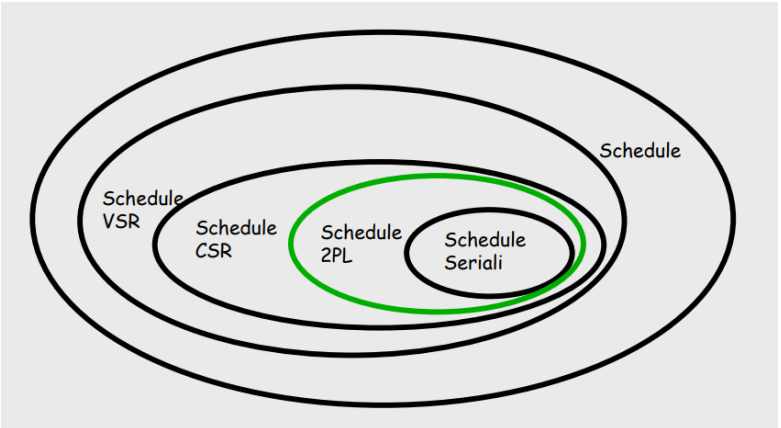
Si nota che S sia CSR.
Affinché sia 2PL dovrebbe essere strutturato in questo modo (*con il locking che va fatto solamente nel caso di una relazione leggi-da* e *se le due transizioni operano sulla stessa risorsa*):

- w_lock1(x) r1(x) w1(x) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Dato che esiste anche w1(y) che va in conflitto con r3(y), allora per precedenza da parte del grafo dei conflitti (il cui ordine sarebbe T3, T1, T2)
Si dovrebbe mettere prima w_lock1(y) prima di unlock1(x)

- w_lock1(x) r1(x) w1(x) w_lock1(y) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Sempre per il grafo dei conflitti se si ha la lettura di y da parte di r3, essa deve precedere T2 e non è compatibile con w1(y) che segue
Quindi *CSR ⇏ 2PL*. Tuttavia, 2PL ⇒ CSR



Scritto da Gabriel

Due transazioni si dicono *serializzabili* se il risultato delle transazioni è uguale a quello eseguito sequenzialmente, dunque non avendo sovrapposizioni temporali.
Per esempio avendo:
r2(x) r1(x) r2(y) w2(y) r1(y) w1(x)
Vediamo che le operazioni sono eseguite in maniera "ordinata", dunque a delle letture seguono delle scritture. In particolare, una lettura di T1 su x e due letture di T2 su x e y, una scrittura di T2 su y e una lettura/scrittura di t1 su x. *Non ci sono transazioni sovrapposte.*
Se si scambiassero:

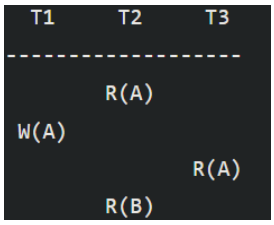
- r1(x) con r2(y)
- r1(x) con w2(y)

avremmo:
r2(y) r2(y) w2(y) r1(x) r1(y) w1(x)
Anche qui, avremmo una lettura e scrittura di T2, poi due letture di T1 su x e y e una operazione in scrittura in modo isolato, quindi senza che l'altra legga quel valore "pendente".
Dunque, tutto bene.

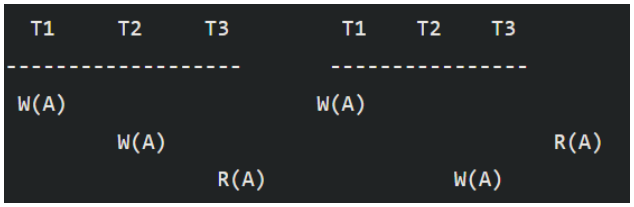
Per introdurre il concetto di *view-derivabilità* tra due schedule devono essere soddisfatte due condizioni:

- lettura iniziale, cioè se una transazione T1 legge il dato A dal database nello schedule S1, allora anche nello schedule S2 anche T1 deve leggere A dal database.

Ad esempio T2 che legge da:

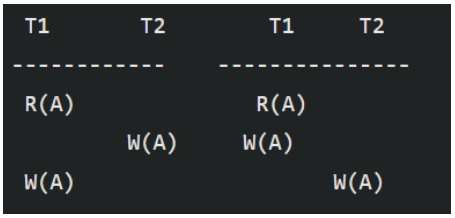


- lettura aggiornata, Se T_i stesse leggendo A che viene aggiornato da T_j in S1, allora in S2 anche T_i dovrebbe leggere A che è aggiornato da T_j.



Qui sopra abbiamo l'esempio in cui T3 legge un valore aggiornato da T2 (in S1) e anche T3 legge A aggiornato da T1 (dentro S2). Questo non è view equivalente.

- scrittura finale, se una transazione T1 ha aggiornato A per ultimo in S1 allora in S2 anche T1 dovrà eseguire le scritture finali. Una scrittura è detta finale quando sia l'ultima su quell'oggetto.
  Per esempio su:
  r1(x) w1(x) w1(y) r2(x) w2(y) → w1(x) è scrittura finale per l'oggetto x, mentre w2(y) lo è per y



In questo esempio vediamo che non sono view-equivalenti perché l'operazione di scrittura finale in S1 è fatta da T1, mentre in S2 è fatta da T2.

Buon riassunto di *view-equivalenza*:

Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions:

1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

Read vs Initial Read: You may be confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read. This will be more clear once we will get to the example in the next section of this same article.

2. **Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.

Per il discorso di *conflict-serializzabile*, diciamo semplicemente che due operazioni sono in conflitto se:

- operano sullo stesso dato
- appartengono a due transizioni differenti
- almeno una delle operazioni è una scrittura

Detto in tre parole:

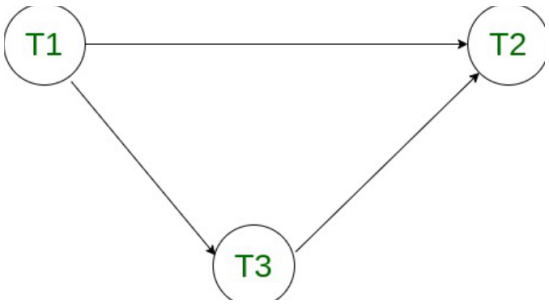
1) una transazione non potrà andare in conflitto con sé stessa a seguito di lettura/scrittura
2) non sapendo in un contesto reale in che ordine si eseguono le operazioni, banalmente, ogni volta che si ha una lettura/scrittura da parte di due transizioni diverse, si ha un conflitto.

Ad esempio:

Le operazioni che vanno in conflitto (si indicano dalla precedente alla successiva sono):

1) R3(X) e W2(X) [T3 -> T2]
2) W1(Y) e R3(Y) [T1 -> T3]
3) W1(Y) e W2(Y) [T1 -> T2]
4) R3(Y) e W2(Y) [T3 -> T2]

Va costruito il grafo dei conflitti, che segue le frecce indicate e ci si accorge che *dato che non ha cicli*, lo schedule è conflict-serializzabile.



Un esempio invece come, ragionando come prima:

S = r1(x), r2(y), r3(x), w3(x), w1(x), w1(y), r2(x), w2(x)

Si disegna il grafo dei conflitti avendo che:

- R1 dipende da W3 (w3(x) – r1(x))
- R3 dipende da W1 (w1(x) – r3(x))
- R2 dipende da W1 (w1(y) – r2(y))
- R2 dipende da W3 (w3(x) – r2(x))
- W2 dipende da R1 (w2(x) – r1(x))
- W2 dipende da R3 (w2(x) – r3(x))

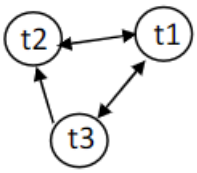
E quindi dato che si ha un ciclo e le operazioni non sono fatte in ordine, S non è né conflict-serializzabile né view-serializzabile.

Attenzione: per capire veramente se un conflict è view-serializzabile, basta semplicemente usare il grafo dei conflitti e ordinarsi le operazioni come per il grafo; se si vede che le scritture finali cioè tutte le scritture) sono nello stesso ordine e le dipendenze sono le stesse, allora è view-serializzabile.

Se è CSR, allora comunque è VSR.

Inoltre:

- per ordinare le transazioni, basta vedere dove vanno le frecce nel grafo dei conflitti oppure verificare l'ordine delle scritture. Spesso, se non sempre, l'ordine delle scritture finali corrisponde al giusto riordinamento delle transazioni.



$W_1(A)\ R_2(A)\ R_2(B)\ W_2(D)\ R_3(C)\ R_1(C)\ W_3(B)\ R_4(A)\ W_3(C).$

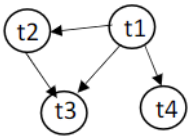
Nello schedule qui sopra:

1) scrive prima T1, T1 stessa e T2 leggono
2) scrive T2, T3 e T1 leggono
3) scrive T3, legge T4
4) scrive T3

Le operazioni sono fatte in maniera ordinata, le scritture non si sovrappongono.

È view serializzabile; i conflict ce ne stanno e di diversi.

Infatti, il grafo dei conflitti riporta;

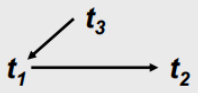


Attenzione alla relazione tra 2PL (Locking a due fasi) e CSR.

Scritto da Gabriel

Consideriamo:

$S: r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$



Si nota che S sia CSR.
Affinché sia 2PL dovrebbe essere strutturato in questo modo (*con il locking che va fatto solamente nel caso di una relazione leggi-da* e *se le due transizioni operano sulla stessa risorsa*):

- w_lock1(x) r1(x) w1(x) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Dato che esiste anche w1(y) che va in conflitto con r3(y), allora per precedenza da parte del grafo dei conflitti (il cui ordine sarebbe T3, T1, T2)
Si dovrebbe mettere prima w_lock1(y) prima di unlock1(x)

- w_lock1(x) r1(x) w1(x) w_lock1(y) unlock1(x) r2(x) w2(x) r3(y) w1(y)

Sempre per il grafo dei conflitti se si ha la lettura di y da parte di r3, essa deve precedere T2 e non è compatibile con w1(y) che segue.

Esercitazione 6: Transazioni

Le transazioni vengono registrate su disco fintanto che, eseguendo un update, prima di un CHECK sono effettivamente salvate ed attive; dobbiamo quindi registrare le transazioni per cui dobbiamo fare l'UNDO.
Infatti, normalmente, si cerca l'ultimo checkpoint partendo da sotto e si costruiscono gli insiemi di UNDO e REDO.
Ad esempio, le transazioni che non hanno eseguito un commit, potenzialmente con B/U/D/A, possono andare in fase di UNDO.
Dunque, una transazione che ha fatto il commit (quindi C) si mette in REDO (non ho nessuna garanzia dell'esecuzione corretta dell'operazione).

Sull'esempio delle slide:
UNDO = {T2, T3}
REDO = {T4, T5}
In un esempio: U(T2, O1, B1, A1)
Update per T2 di T1 indicando *B – Before ed A – After* per B1 (oggetto vecchio) e B2 (oggetto nuovo).
Vado poi a rieseguire, partendo dal basso, tutte le operazioni che sono in UNDO

1) D(06)
2) I(05, B7)
3) U(O3, B5)
4) U(O2, B3)
5) U(O1, B1)

T1 non ci va, dato che aveva anche fatto il commit prima del checkpoint.
T5 non si considera perché è in REDO; quindi si fa al secondo passaggio, rieseguendo tutte le operazioni di T5.
Andrò quindi a ripetere:

- B(T4)
- U(T4, O3, B4, A4)
- U(T5, O4, B6, A6)

È indifferente eseguire prima il REDO o l'UNDO.

Esercizio 1
Descrivere la ripresa a caldo, indicando la costituzione progressiva degli insiemi di UNDO e REDO e le azioni di recovery, a fronte del seguente log:

DUMP, B(T1), B(T2), B(T3), I(T1, O1, A1), D(T2, O2, B2), B(T4),
U(T4, O3, B3, A3), U(T1, O4, B4, A4), C(T2), CK(T1, T3, T4), B(T5), B(T6),
U(T5, O5, B5, A5), A(T3), CK(T1, T4, T5, T6), B(T7), A(T4),
U(T7, O6, B6, A6), U(T6, O3, B7, A7), B(T8), A(T7), guasto

Si percorre il log a ritroso fino al più recente checkpoint, cioè CK(T1, T4, T5, T6).
Si mette tutto in UNDO → UNDO = {T1, T4, T5, T6}
Si considera dentro l'UNDO anche T7 che fa l'ABORT e T8 che era già nell'UNDO.
Nessuna transazione ha fatto il commit e si ha REDO{}
UNDO = {T1, T4, T5, T6, T7, T8}

A questo punto si parte al contrario e si ripetono le seguenti operazioni: (mettendo precedente e successivo all'operazione):

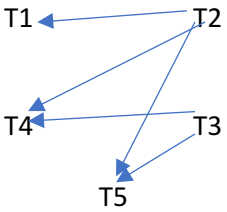
1) U (O3, B7)
2) U (O6, B6)
3) U (O5, B5)
4) U (O4, B4)
5) U (O3, B3)
6) Delete di O1 (essendo l'oggetto perso O1, vado a cancellarlo così lo reinserirò ancora)

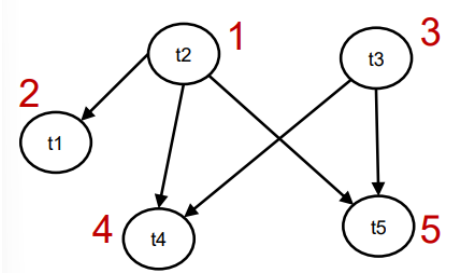
Viene poi ripercorso in avanti tutto il log per rieseguire le operazioni di REDO.
Prima quindi faccio la ripresa a freddo, prendendo lo stato del disco al momento del dump e ripeto tutte le operazioni fino a prima del guasto.
Con la ripresa a caldo, si crea un hardware utile che esegue correttamente le operazioni.

Esercizio 2
Considera il seguente schedule:
S = r2(x) r1(x) w3(t) w1(x) r3(y) r4(t) r2(y) w2(z) w5(y) w4(z)
Esempi di conflitto/conflitto:

S è conflict-serializable? Se sì, mostrare uno schedule che è conflict-equivalente.

Si listano i conflitti:





Quindi S è conflict-serializzabile.
(*Attenzione: capire l'ordine è semplice, senza scervellarsi. Basta vedere l'ordine che c'è scritto dallo schedule*). Oppure, equivalentemente, si ragiona con:
"Se S è conflict-serializzabile, è possibile costruire un grafo dei conflitti di S che è aciclico. Il percorso nel grafo che tocca tutte le transazioni, fornisce un ordine delle transazioni. Scrivendo le operazioni nell'ordine delle transazioni e rispettando l'ordine all'interno delle transazioni, si ottiene uno schedule seriale che è conflict-equivalente e quindi anche view-equivalente."

Scritto da Gabriel

Lo schedule CSR-equivalente è quindi:

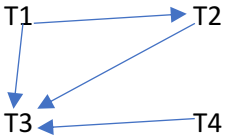
r2(x) r2(y) w2(z) r1(x) w1(x) w3(t) r3(y) r4(t) w4(z) w5(y)

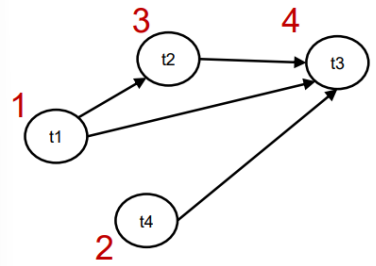
In questo modo i conflitti non sono stati invertiti (come se fossero eseguite da un unico client). Quindi, dato che CSR implica VSR, questa è corretta.

Esercizio 3

S = r1(x) w2(x) r3(x) w1(u) w3(v) r3(y) r2(y) w3(u) w4(t) w3(t)

Dire se è conflict-serializzabile e trovare uno schedule seriale conflict-equivalente

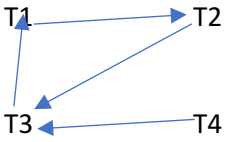




Esso è conflict-serializzabile:

Se per esempio si decidesse di spostare w1(u) nella posizione (prosegue la frase in pagina dopo):
S = r1(x) w2(x) r3(x) w3(v) r3(y) r2(y) w3(u) w1(u) w4(t) w3(t)

si genererebbe un ciclo:



Qualsiasi cosa succede, w1(u) genera conflitti. Se volessi eseguire operazioni di scrittura, dovrei quindi evitare un ciclo in qualche modo. L'operazione non può essere effettuata in quel punto; pima di effettuarla, occorre "rompere" il ciclo, cioè fare il commit o abort di t2 o t3 per togliere il nodo delle transazioni attive dal grafo.

Esercizio 4

Indicare se i seguenti schedule sono VSR:

1. r1(x), r2(y), w1(y), r2(x) w2,(x)
2. r1(x), r2(y), w1(x), w1(y), r2(x) w2,(x)
3. r1(x), r1(y), r2(y), w2(z), w1(z), w3(z), w3(x)
4. r1(y), r1(y), w2(z), w1(z), w3(z), w3(x), w1(x)

Sappiamo individuare se sono VSR se invertendo le transazioni, non si hanno relazioni del tipo *"legge da"*.

1) Non è VSR: infatti invertendo ad esempio w1(y) con r2(y) oppure w2(x) con r1(x) si hanno relazioni di dipendenza.
2) Non è VSR, infatti si hanno varie dipendenze conflittuali, come w1(y) per r2(y) oppure r2(x) per w1(x). Dato che si hanno più relazioni di lettura/scrittura, cambiando l'ordine dei fattori, il risultato cambia.
3) In questo caso è VSR: infatti, non ci sono letture/scritture che concorrono, avendo una sola scrittura su X mentre tutte le altre agiscono su Z oppure su Y. Anche per w3, unico che esegue su x e z, quindi due valori diversi, non si ha ordine diverso di scrittura.
4) Non è VSR: infatti ho due scritture con T1 per x e z e due scritture su T3 per x e z; potenzialmente queste transazioni possono essere invertite e non avere lo stesso ordine di esecuzione finale.

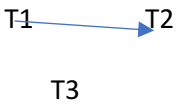
Esercizio 5

Classificare i seguenti schedule (come: NonSR, VSR, CSR). Nel caso uno schedule sia VSR oppure CSR, indicare tutti gli schedule seriali e esso equivalenti.

1. r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)
2. r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)

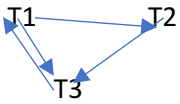
1) r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)

Indicando al solito i conflitti con i colori, si vede che disegnando si avrebbe:



Vedendo anche che ci sono due scritture di w2 in x e z e una sola scrittura di w1 in x, possiamo dire che è certamente conflict-equivalente (dal grafo), mentre per la VSR, si avrebbero le stesse scritture finali, in quanto le dipendenze presenti hanno una lettura iniziale su z e y che non interferiscono l'una con l'altra e le stesse scritture finali su x e z, anche qui che non interferiscono.

2) r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)



Come si vede, non è CSR essendoci un ciclo. Si vede che non è neanche VSR in quanto l'ordine delle scritture finali non viene rispettato dalle letture iniziali, portando quindi a differenti viste sul dato. Dunque, S non è né VSR che CSR.

Esercitazione 7: Esercizi vari per esame

Dato il seguente schema:

```
AEROPORTO(Città,Nazione,NumPiste)
VOLO(IdVolo,GiornoSett,CittàPart,OraPart,CittàArr,OraArr,TipoAereo)
AEREO(TipoAereo,NumPasseggeri,QtaMerci)
```

Esprimere le seguenti query in Algebra Relazionale:
1. Le nazioni da cui parte e arriva il volo con codice AZ274;
2. Le città da cui partono voli internazionali
3. Le città da cui partono solo voli nazionali
4. Le città con più piste;

1) $\pi_{Nazione}((\sigma_{IdVolo} = \text{'AZ274'} (VOLO)) \bowtie_{CittaPart=Citta \text{ OR } CittaArr=Citta} AEROPORTO))$
oppure
$\pi_{Nazione}((\sigma_{IdVolo} = \text{'AZ274'} (VOLO)) \bowtie_{CittaPart=Citta} AEROPORTO))$ U
$\pi_{Nazione}((\sigma_{IdVolo} = \text{'AZ274'} (VOLO)) \bowtie_{CittaArr=Citta} AEROPORTO))$
2) VP= (VOLO $\bowtie_{CittaPart=Citta}$ AEROPORTO);
VA=(VOLO $\bowtie_{CittaArr=Citta}$ AEROPORTO);
$\pi_{VP.CittaPart}$(VP $\bowtie_{VP.IdVolo=VA.IdVolo \text{ AND } VP.Nazione<>VA.Nazione}$ VA);
3) VP= (VOLO $\bowtie_{CittaPart=Citta}$ AEROPORTO);