:: Basi di Dati - Complementi ::

Esercizi: Serializzabilità di Schedule

Teoria

Il concetto di serializzabilità di schedule si inserisce nella teoria del controllo di concorrenza tra transazioni, qui considerate come una sequenza di azioni in lettura o scrittura, gli *schedule* appunto. Le operazioni di ogni transazione sono contraddistinte da un proprio indice numerico, onde evitare confusione tra quelle attive concorrenti. Un esempio di schedule è il seguente:

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

In generale, $r_i(ogg)$ rappresenta la lettura dell'oggetto *ogg* da parte della transazione t_i, mentre $w_i(ogg)$ ne indica la scrittura. L'esecuzione delle varie operazioni di I/O sugli oggetti segue l'ordine con cui sono riportati negli schedule.

Il modulo che si occupa del controllo di concorrenza è lo *scheduler*, e dovrà rifiutare gli schedule che porterebbero a inconsistenze o anomalie ed accettare tutti gli altri. Un metodo teorico per studiare quali schedule sono corretti e quali no è considerarne solo le *commit-proiezioni*, ovvero tutte e sole le operazioni delle transazioni che non producono un *abort*. Da un punto di vista pratico non è una strada perseguibile, ma ce ne preoccuperemo poi. Con questi presupposti, baseremo il controllo della concorrenza sulle definizioni di *schedule seriale* e *serializzabile*.

Uno schedule si dice *seriale* se per ogni transazione tutte le loro operazioni sono eseguite consecutivamente, senza essere inframmezzate da altre. Uno schedule non seriale si dice *serializzabile* se produce lo stesso risultato di uno schedule seriale delle stesse transazioni. Assumiamo che una sequenza di transazioni seriali sia corretta, e di conseguenza lo sia anche uno schedule serializzabile. Ma ho degli strumenti pratici per capire quando lo è? Certo, e si basano sui concetti di *view-equivalenza* e i suoi raffinamenti, ovvero *conflict-equivalenza* e il *locking a due (e tre) fasi*. Cosa sono e come verificarli lo vedremo nei prossimi paragrafi in modo pratico.

View-equivalenza

Cenni teorici

Per stabilire se due schedule sono *view-equivalenti* bisogna verificare che abbiano le stesse relazioni *legge-da* e le stesse *scritture finali*.

In generale un'operazione $r_i(x)$ legge da $w_j(x)$ se la scrittura precede la lettura e non vi sono altre scritture sullo stesso oggetto (o risorsa) da parte di altre transazioni. Ad esempio, dato il seguente schedule:

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

$r_1(x)$ non legge da nessuno, mentre $r_2(x)$ legge da $w_1(x)$

Una scrittura invece si dice *finale* per un dato oggetto se, molto banalmente, è l'ultima scrittura su di esso nello schedule. Riprendendo l'esempio di prima:

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

$w_1(x)$ è scrittura finale per l'oggetto x, mentre $w_2(y)$ lo è per y.

Uno schedule viene detto *view-serializzabile* (VSR) se è view-equivalente ad uno seriale. Verificare la view-equivalenza di uno schedule generico è un problema NP-difficile, quindi a questo sistema è preferibile applicare condizioni più restrittive che ne abbassino la complessità.

Come determinare se uno schedule è view-serializzabile

Utilizziamo uno schedule d'esempio:

$r_1(x)\ r_1(t)\ r_2(z)\ w_3(x)\ w_1(x)\ r_1(y)\ w_3(t)\ w_2(x)\ w_1(y)$

Dovendo verificare la view-equivalenza, dovremo determinare le relazioni *legge da* e le *scritture finali*, che per comodità riporteremo in forma tabellare.
Iniziamo dalle *leggi da*, e mettiamo su una colonna tutte le operazioni in lettura e su una seconda colonna le prime scritture sulla stessa risorsa che le precedono:

LETTURA	LEGGE DA
$r_1(x)$	-
$r_1(t)$	-
$r_2(z)$	-
$r_1(y)$	-

Quindi nel caso particolare non abbiamo individuato operazioni di lettura che leggono da qualcuno.

Passiamo ora alle *scritture finali* e, partendo dal fondo, teniamo traccia per ogni risorsa delle ultime scritture su essa. In tabella avremo una colonna per le risorse, una per le scritture finali ed una aggiuntiva su eventuali altre operazioni di scrittura sulla stessa risorsa.

RISORSA	SCRITTURA FINALE	ALTRE SCRITTURE
x	$w_2(x)$	$w_1(x)$, $w_3(x)$
y	$w_1(y)$	-
z	-	-
t	$w_3(t)$	-

Dalla prima riga deduciamo che la transazione 2 deve seguire le transazioni 1 e 3, dato che è l'ultima a dover scrivere sull'oggetto x. Osservando la tabella delle relazioni legge da possiamo stabilire inoltre la precedenza della transazione 1 sulla 3, dato che l'operazione $r_1(x)$ non legge da $w_3(x)$. Riassumendo le varie considerazioni fatte, lo schedule seriale che otteniamo è il seguente (t_1, t_3, t_2):

$r_1(x)\ r_1(t)\ w_1(x)\ r_1(y)\ w_1(y)\ w_3(x)\ w_3(t)\ r_2(z)\ w_2(x)$

Lo confrontiamo con quello di partenza e verifichiamo che abbiano stesse relazioni *legge da* e stesse *scritture finali*.

LETTURA	LEGGE DA	RISORSA	SCRITTURA FINALE	ALTRE SCRITTURE
$r_1(x)$	-	x	$w_2(x)$	$w_1(x)$, $w_3(x)$
$r_1(t)$	-	y	$w_1(y)$	-
$r_2(z)$	-	z	-	-
$r_1(y)$	-	t	$w_3(t)$	-

Le tabelle sono identiche, quindi lo schedule dato è *view-serializzabile* (VSR).

Conflict-equivalenza

Cenni teorici

Verificare la view-serializzabilità di uno schedule è un'operazione molto complessa, dunque è preferibile sfruttare la nozione di *conflitti* e di *conflict-serializzabilità*.
Due operazioni si dicono *in conflitto* tra loro se soddisfano tre condizioni:

- appartengono a transazioni diverse
- riguardano la stessa risorsa
- almeno una delle due è una scrittura: lettura-scrittura, scrittura-lettura, scrittura-scrittura

Due schedule si dicono *conflict-equivalenti* se contengono le stesse operazioni e se le coppie di operazioni in conflitto appaiono con lo stesso ordine. Abbiamo ora tutti gli strumenti per determinare se uno schedule è *conflict-serializzabile*, ovvero quando è conflict-equivalente ad uno schedule seriale. L'insieme di tutti gli schedule conflict-serializzabili è chiamato CSR.

Come determinare se uno schedule è conflict-serializzabile

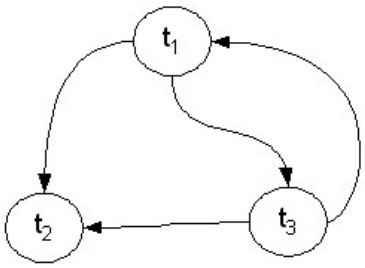
Riprendiamo lo schedule dell'esempio precedente:

$r_1(x)\ r_1(t)\ r_2(z)\ w_3(x)\ w_1(x)\ r_1(y)\ w_3(t)\ w_2(x)\ w_1(y)$

Come prima cosa dovremo individuare tutte le operazioni in conflitto. Partiamo dalla prima e verifichiamo le tre condizioni elencate prima per ognuna delle operazioni a seguire. Terminati i confronti, passiamo alla seconda operazione e ripetiamo la procedura e così via. Nel nostro caso avremo i seguenti conflitti:

- $r_1(x)$, $w_3(x)$
- $r_1(x)$, $w_2(x)$
- $r_1(t)$, $w_3(t)$
- $w_3(x)$, $w_1(x)$
- $w_3(x)$, $w_2(x)$
- $w_1(x)$, $w_2(x)$

Disegnamo a questo punto il *grafo dei conflitti*, ovvero un grafo orientato i cui nodi rappresentano le transazioni e gli archi la direzione delle operazioni in conflitto. Se tale grafo non presenta cicli, allora lo schedule è *conflict-serializzabile*. In questo caso otteniamo:



che forma un ciclo tra le transazioni t_1 e t_3. Dunque così com'è lo schedule d'esempio non è CSR.
Se venisse chiesto di scegliere una o più operazioni da eliminare per renderlo CSR, andranno ovviamente scelte quelle che concorrono a formare il ciclo, cercando di toglierne possibilmente il meno possibile. Ad esempio in questo caso sarebbe preferibile eliminare o solo $w_3(x)$ o solo $w_1(x)$, piuttosto che eliminare la coppia $r_1(x)$-$r_1(t)$ o la coppia $r_1(x)$-$w_3(t)$.

Notiamo infine che verificare che uno schedule è CSR ha complessità lineare, dal momento che si riduce all'analisi di ciclicità di un grafo orientato; pur avendo però diminuito la complessità rispetto al VSR, la sua risoluzione rimane piuttosto onerosa.

Locking a due fasi

Cenni teorici

Le due strategie viste finora non garantiscono la serializzabilità, ma la verificano a posteriori. Nei sistemi utilizzati in pratica si preferisce invece fare in modo che essa sia in qualche modo assicurata, adottando particolari accorgimenti e spesso strutture dati d'appoggio. Una delle tecniche più utilizzate si basa sul *locking*, che fa uso di una variabile di *lock* per descrivere lo stato di una risorsa rispetto alle operazioni che lo riguardano. Quando una transazione vuole utilizzare una risorsa deve *prima* richiederne il lock, attendere finché non gli viene concesso e rilasciarlo dopo il suo utilizzo. In particolare, nel *locking a due fasi* (2PL) una volta che una transazione ha rilasciato un lock qualsiasi, non può più acquisirne altri. Quindi, se una transazione t_i ha bisogno del lock su una risorsa già occupata da t_j,

quest'ultima prima di rilasciarla dovrà acquisire il lock di tutte le risorse a cui deve ancora accedere, perché una volta unlockata una risorsa non potrà più lockarne nessuna.

Perché ci interessano tanto gli schedule generati con questa tecnica? *Perché sono serializzabili*.

Esistono due tipi di 2PL, a seconda della variabile di lock che viene utilizzata:
- a *2 stati*: il lock può essere *locked* (risorsa bloccata) o *unlocked* (risorsa rilasciata o comunque disponibile)
- a *3 stati*: il lock può essere *read locked*, *write locked* o *unlocked*. L'introduzione del read locked consente a due o più transazioni di condividere la stessa risorsa in lettura.

Ultima considerazione: gli schedule 2PL sono un sottoinsieme della classe di schedule CSR, che sono a loro volta sottoinsieme dei VSR. Ne consegue che se uno schedule è non view-serializzabile non è nemmeno conflict-serializzabile né tantomeno basato su 2PL!

Come determinare se uno schedule è generato da uno scheduler basato su 2PL

Riprendiamo lo schedule dell'esempio precedente:

$r_1(x)\ r_1(t)\ r_2(z)\ w_3(x)\ w_1(x)\ r_1(y)\ w_3(t)\ w_2(x)\ w_1(y)$

Per iniziare torna molto utile raggruppare le operazioni per transazione, così da facilitare le azioni di lock e unlock. Dunque avremo:
$t_1 = r_1(x)\ r_1(t)\ w_1(x)\ r_1(y)\ w_1(y)$
$t_2 = r_2(z)\ w_2(x)$
$t_3 = w_3(x)\ w_3(t)$

Ora dobbiamo stabilire operazione per operazione, mantenendo l'ordine dello schedule dato, a quale transazione dare il lock e su quale risorsa. Conviene mantenere queste informazioni in una tabella con una colonna per le operazioni, una per i lock/unlock e infine tante colonne quante sono le risorse, in cui indicare quale transazione ne detiene attualmente l'accesso. L'istruzione da scrivere nella colonna dei lock avrà la seguente forma: `lock (transazione, risorsa): OK/NO`, o il complementare `unlock (transazione, risorsa): OK/NO`.
Riempiamo la tabella con il nostro schedule:

OPERAZIONE	LOCK	X	Y	Z	T
$r_1(x)$	lock(t_1, x): OK	t_1			
$r_1(t)$	lock(t_1, t): OK	t_1			t_1
$r_2(z)$	lock(t_2, z): OK	t_1		t_2	t_1
$w_3(x)$	lock(t_1, y): OK	t_1	t_1	t_2	t_1
	unlock(t_1, x): NO	t_1	t_1	t_2	t_1

Osserviamo i passaggi effettuati a partire dall'operazione $w_3(x)$:
- la transazione 3 vuole effettuare una scrittura su x, quindi deve prima togliere il lock alla transazione che ne ha possesso, ovvero la t_1
- prima di unlockare una risorsa a t_1 devo prima bloccare tutte quelle risorse di cui in futuro avrà bisogno, quindi la y
- a questo punto unlocko x dalla transazione 1, e qui sta il problema: l'operazione successiva dello schedule è $w_1(x)$, ma avendo appena unlockato non potrò più rilockarlo

Quindi lo schedule in esame non è 2PL. Notare che in realtà noi già sapevamo l'esito della verifica: dato che lo schedule non era CSR, non poteva in alcun modo essere nemmeno 2PL!

Torna alla pagina di Basi di Dati - Complementi

Ultimo aggiornamento: June 30, 2009, at 03:34 PM