Il controllo della concorrenza

Index

Introduction

In sistemi di calcolo con un a sola CPU i programmi sono eseguiti concorrentemente in modo *interleaved* (interfogliato), quindi la CPU può:

- eseguire alcune istruzioni di un programma
- sospendere quel programma
- eseguire istruzioni di altri programmi
- ritornare ad eseguire istruzioni del primo

Questo tipo di esecuzione è detta concorrente e permette un uso efficiente della CPU

Accesso concorrente alla BD

In un DBMS la principale risorsa a cui tutti i programmi accedono in modo concorrente è la **base di dati**. Se sulla BD vengono effettuate solo letture (la BD non viene mai modificata), l'accesso concorrente non crea problemi. Se sulla BD vengono effettuate anche scritture (la BD viene modificata), l'accesso concorrente può creare problemi e quindi deve essere controllato

---

Transazione

Una **transazione** è l'esecuzione di una parte di un programma che rappresenta un'unità logica di accesso o modifica del contenuto della base di dati

Proprietà delle transazioni

Le proprietà logiche delle transazioni si racchiudono sotto l'acronimo **ACID** (Atomicità, Consistenza, Isolamento, Durabilità). Analizziamole nel dettaglio:

- **atomicità** → la transazione è indivisibile nella sua esecuzione e la sua **esecuzione deve essere totale o nulla**, non sono ammesse esecuzioni parziali (se per qualche problema una transizione non va a termine bisogna fare un rollback sui dati)
- **consistenza** → quando una transazione il database si trova in uno stato consistente e quando la transazione termina, il database deve essere in un altro stato consistente, ovvero **non deve violare eventuali vincoli di integrità**, quindi non devono verificarsi contraddizioni tra i dati archiviati nel DB
- **isolamento** → ogni transazione deve essere eseguita **in modo isolato e indipendente** dalle altre transazione, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione (è ammesso che il risultato cambi a causa di diverse operazioni, è un problema quando ci viene restituito un dato che non era quello richiesto)
- **durabilità** → detta anche persistenza, si riferisce al fatto che una volta che una transazione abbia richiesto un *commit work*, i cambiamenti apportati **non dovranno più essere persi**. Per evitare che nel lasso di tempo fra il momento in cui la base di dati si impegna a scrivere le modifiche e quelli in cui li scrive effettivamente si verifichino perdite di dati dovuti a malfunzionamenti, vengono tenuti dei registri di log dove sono annotate tutte le operazioni sul DB

Schedule di un insieme di transazioni

Per **schedule** si intende un insieme di T transizioni nella cui esecuzione viene mantenuto l'ordine delle singole operazioni di una transizioni, ma ci può essere interleaving tra le transizioni (esecuzione di una parte di una transizione, e una parte di un'altra transizione)

Schedule seriale

Si parla di **schedule seriale**, quando lo schedule è ottenuto permutando le transazioni in T, quindi uno schedule seriale corrisponde ad una esecuzione **sequenziale** (non interfoglia) delle transazioni

---

Problemi

Consideriamo le due transazioni:

T_1
$read(X)$
$X:=X-N$
$write(X)$
$read(Y)$
$Y:=Y+N$
$write(Y)$

T_2
$read(X)$
$X:=X+M$
$write(X)$

Si possono presentare tre diversi tipi di problemi a causa dell'interleaving

Aggiornamento perso (lost update)

Consideriamo il seguente schedule di T_1 e T_2

T_1	T_2
read(X)	
X:=X-N	
	read(X)
	X:=X+M
write(X)	
read(Y)	
	write(X)
Y:=Y+N	
write(Y)	

Se il valore iniziale di X è X_0 al termine dell'esecuzione dello schedule il valore X è $X_0 + M$ invece di $X_0 - N + M$
L'**aggiornamento** di X prodotto da T_1 viene **perso** (lost update o ghost update)

Dato sporco (dirty data)

Consideriamo il seguente schedule di T_1 e T_2

T_1	T_2
read(X)	
X:=X-N	
write(X)	
	read(X)
	X:=X+M
read(Y)	
T_1 *fallisce*	
	write(X)

Se il valore iniziale di X è X_0 al termine dell'esecuzione dello schedule il valore di X è $X_0 - N + M$ invece di $X_0 + M$
Il valore di X letto da T_2 è un **dato sporco** (temporaneo) in quanto prodotto da una transazione fallita. Per atomicità quindi bisogna pulire i dati e viene fatto attraverso un rollback a cascata

Aggregato non corretto

Consideriamo il seguente schedule di T_1 e T_2

T_1	T_3
	somma:=0
read(X)	
X:=X-N	
write(X)	
	read(X)
	somma:=somma+X
	read(Y)
	somma:=somma+Y
read(Y)	
Y:=Y+N	
write(Y)	

Se il valore iniziale di X è X_0 e il valore iniziale di Y è Y_0, al termine dell'esecuzione dello schedule il valore di $somma$ è $X_0 - N + Y_0$ invece di $X_0 + Y_0$
Il valore di $somma$ è un **dato aggregato**

> **Osservazione**
>
> Perché nei tre casi visti siamo portati a considerare gli schedule **non corretti**? Perché i valori prodotti non sono quelli che si avrebbero se le due transazioni fossero eseguite nel modo "naturale" cioè **sequenzialmente**

Serializzabilità

Tutti gli schedule **seriali** sono **corretti** (grazie alla proprietà di isolamento), invece uno schedule **non seriale** è corretto se è *serializzabile*, cioè se è "**equivalente**" ad uno schedule seriale

> **Warning**
>
> Per **equivalente** non si intende che due schedule (per ogni dati modificato) producono valori uguali. Si potrebbero essere corretti anche se non producono gli stessi valori

---

Equivalenza di schedule

Due schedule sono uguali se (per ogni dato modificato) producono valori uguali, dove due valori sono uguali solo se sono **prodotti dalla stessa sequenza di operazioni**

T_1	T_2
read(X)	
X:=X+N	
write(X)	
	read(X)
	X:=X-M
	write(X)

T_1	T_2
	read(X)
	X:=X-M
	write(X)
read(X)	
X:=X+N	
write(X)	

In questo caso infatti non sono equivalenti anche se danno lo stesso risultato su X, ma sono entrambi seriali quindi corretti

---

Testare la serializzabilità

Dobbiamo comunque considerare che esistono dei problemi “pratici”. Le transazioni infatti vengono sottomesse al sistema in modo continuo e quindi è difficile stabilire quando uno scheduler comincia e quando finisce.
Inoltre è praticamente impossibile determinare in anticipo in quale ordine le operazioni verranno eseguire in quanto esso è determinato in base a diversi fattori:

- carico del sistema
- ordine temporale in cui le transizioni vengono sottomesse al sistema
- priorità delle transazioni
  Infine se prima si eseguono le operazioni e poi si testa la serializzabilità dello schedule, i suoi effetti devono essere annullati e lo schedule risulta non serializzabile

---

Garantire la serializzabilità

L’approccio seguito nei sistemi è quello di determinare **metodi che garantiscano la serializzabilità** di uno schedule eliminando così la necessità di doverla testare ogni

volta

Metodi che garantiscono la serializzabilità

Si hanno due possibilità per garantire la serializzabilità:

- **protocolli**
- **timestamp**
  Nel dettaglio si ha: imporre dei *protocolli*, cioè delle regole, alle transazioni in modo da garantire la serializzabilità di ogni schedule oppure usare i *timestamp* delle transazioni, cioè degli identificatori delle transazioni che vengono generati dal sistema e in base ai quali le operazioni delle transazioni possono essere ordinate in modo da garantire serializzabilità

Item

Entrambi i metodi sopracitati fanno uso del concetto di *item*, ovvero l'unità a cui l'accesso è controllato
Le dimensioni degli item devono essere definite in base all'uso che viene fatto sulla base di dati in modo tale che in media una transazione acceda a pochi item

---

Granularità

Le dimensioni degli item usate da un sistema sono dette la sua **granularità**.
La granularità dell'item va dal singolo campo della tabella all'intera tabella e oltre.
Una granularità **grande** permette una **gestione efficiente** della concorrenza. Una granularità **piccola** può **sovraccaricare** il sistema, ma **aumenta il livello di concorrenza** (consente l'esecuzione concorrente di molte transazioni)