06 – Gestione delle transazioni

La transazione fornisce un meccanismo per descrivere le unità logiche di elaborazione delle basi di dati. I sistemi di gestione delle transazioni sono sistemi con grandi basi di dati e centinaia di utenti che eseguono transazioni contemporaneamente (Esempi: sistemi di prenotazione, bancari, per la gestione delle carte di credito, ...).

DBMS Single-User Vs. Multi-User

E' possibile classificare i database system in base al numero di utenti che possono utilizzare il sistema in modo *concorrente*.

Un DBMS è *single-user* (monoutente) se al più un utente per volta può usare il sistema.

Un DBMS è *multi-user* se più utenti possono usare il sistema concorrentemente.

Più utenti possono accedere al database simultaneamente grazie al concetto di multiprogrammazione, che consente ad un computer di elaborare più programmi o transazioni simultaneamente.

Quando si ha una sola CPU, necessariamente si elabora un processo alla volta. L'illusione di avere più programmi che eseguono contemporaneamente è fornita dai sistemi operativi multiprogrammati. Tali sistemi eseguono alcune istruzioni da un programma, poi lo sospendono e ne eseguono alcune da altri programmi, e così via. Un programma è riattivato dal punto in cui era stato sospeso.

Su sistemi monoprocessore, l'esecuzione concorrente dei programmi è quindi intervallata (*interleaved*): L'interleaving ha luogo in genere quando un programma effettua un I/O (operazione che lascia la CPU inattiva). Su sistemi multiprocessore, invece, l'esecuzione dei programmi avviene realmente in parallelo.

Esempio.

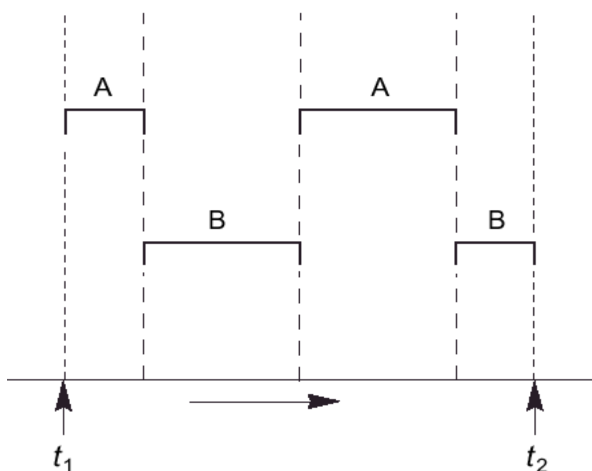


Illustrazione 1: Due processi interleaved su un sistema con una CPU.

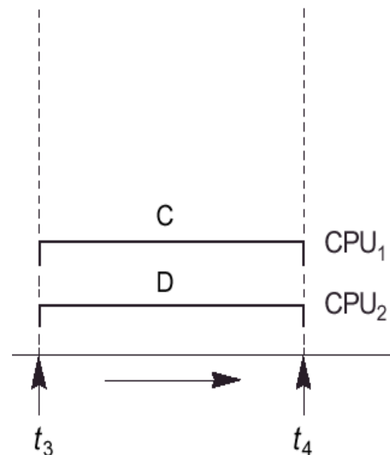


Illustrazione 2: Due processi in parallelo su un sistema con due CPU.

Le transazioni

Informalmente una transazione è un insieme di operazioni che accedono al db, viste logicamente come un'istruzione singola ed indivisibile. Per mostrare la gestione delle transazioni, useremo una visione semplificata di un database system: un db è visto come una collezione di data item, ognuno con un nome. La dimensione del data item è detta *granularità*: può essere un singolo campo di un record, così come un intero blocco di un disco.

Con tale semplificazione, le possibili operazioni di accesso al db che una transazione può effettuare sono:

- *Read_item(X)*: legge l'item di nome X in una variabile di programma. Per semplicità assumiamo che anche la variabile si chiami X.
- *Write_item(X)*: scrive il valore della variabile di programma X nell'elemento X del database.

Read_item(X) . Sappiamo che l'unità di trasferimento di dati è il blocco. Passi per eseguire un comando Read_item(X):

1. Trovare l'indirizzo del blocco che contiene X.
2. Copiare tale blocco in un buffer in memoria centrale (se il blocco non è già in un buffer).
3. Copiare l'elemento X dal buffer alla variabile di programma X.

Write_item(X) . Passi per eseguire un comando Write_item(X)

1. Trovare l'indirizzo del blocco che contiene X.
2. Copiare tale blocco in un buffer in memoria centrale (se il blocco non è già in un buffer).
3. Copiare l'elemento X dalla variabile di programma di nome X nella sua locazione nel buffer.
4. Memorizzare il blocco aggiornato dal buffer al disco.

Il controllo della concorrenza e dei meccanismi di recovery riguardano principalmente i comandi di accesso al db in una transazione. Transazioni inviate da più utenti, che possono accedere e aggiornare gli elementi del db, sono eseguite concorrentemente. Se l'esecuzione concorrente non è controllata, si possono avere problemi di *database inconsistente*.

Esempio. Supponiamo di avere un db per la prenotazione di posti in aereo, in cui è memorizzato un record per ogni volo. Supponiamo di avere due transazioni, T_1 e T_2: la transazione T_1 cancella N prenotazioni da un volo il cui numero di posti occupati è memorizzato nell'item del db di nome X, e riserva lo stesso numero su un altro volo il cui numero di posti occupati è memorizzato nell'item di nome Y. T_2 prenota M posti sul primo volo referenziato nella transazione T_1.

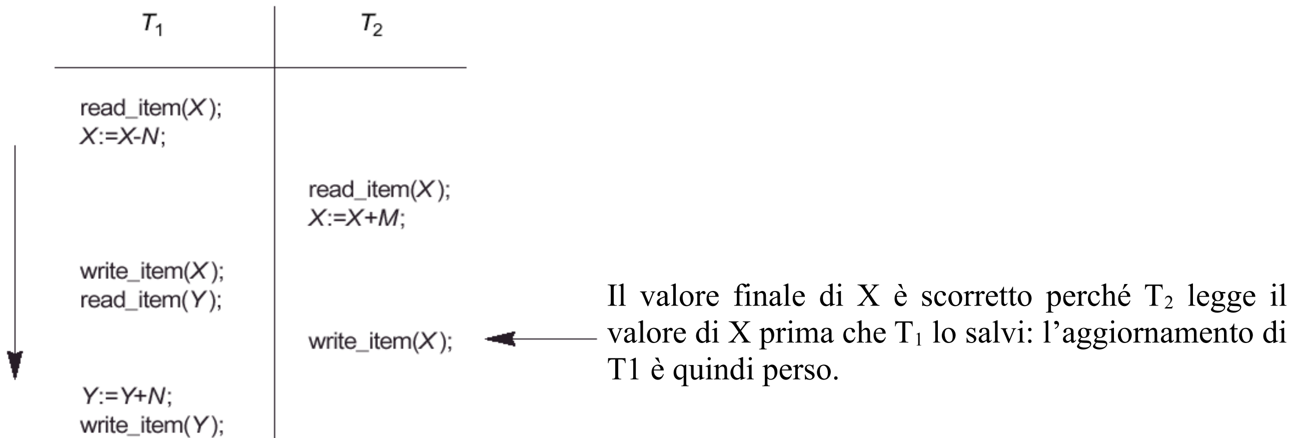
Ecco le due transazioni per la gestione dei posti:

(a) T_1	(b) T_2
read_item (X); X:=X-N; write_item (X); read_item (Y); Y:=Y+N; write_item (Y);	read_item (X); X:=X+M; write_item (X);

Vediamo i possibili problemi che possono sorgere eseguendo T1 e T2 concorrentemente .

Aggiornamento Perso

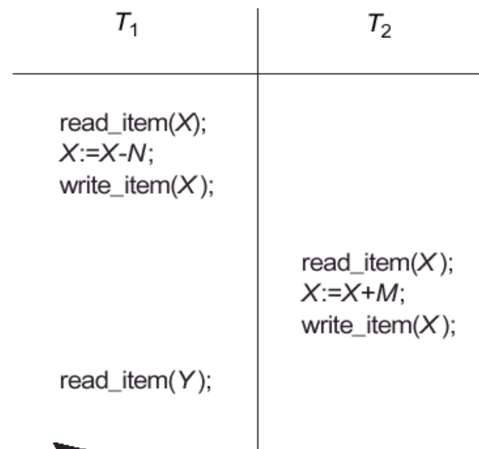
Supponiamo che T1 e T2 siano avviate insieme e che le loro operazioni siano interleaved dal sistema operativo nel modo seguente:



Il valore finale di X è scorretto perché T_2 legge il valore di X prima che T_1 lo salvi: l'aggiornamento di T1 è quindi perso.

Aggiornamento temporaneo

Problema dell'aggiornamento temporaneo (o lettura sporca): una transazione aggiorna un elemento ma poi fallisce per qualche motivo. L'elemento aggiornato è però letto da un'altra transazione prima che esso sia riportato al suo valore originario.



La transazione T_1 fallisce e deve riportare X al suo valore originario; nel frattempo T_2 ha letto il valore "temporaneamente" scorretto di X.

Il valore di X letto da T_2 è detto dato sporco, perché è stato creato da una transazione annullata.

Totalizzazione scorretta

Problema della totalizzazione scorretta: se una transazione sta calcolando una funzione di aggregazione su un certo insieme di record, mentre altre transazioni stanno aggiornando alcuni di tali record, la funzione può calcolare alcuni valori prima dell'aggiornamento ed altri dopo.

Una transazione T3 sta calcolando il numero totale di prenotazioni su tutti i voli mentre T1 è in esecuzione:

T_1	T_3
	sum:=0;
	read_item(*A*);
	sum:=*sum*+*A*;
	⋮
read_item(*X*);	
X:=*X*-N;	
write_item(*X*);	
	read_item(*X*);
	sum:=*sum*+*X*;
	read_item(*Y*);
	sum:=*sum*+*Y*;
read_item(*Y*);	
Y:=*Y*+*N*;	
write_item(*Y*);	

Il risultato di T3 è sbagliato, poiché T3 legge il valore di X dopo che sono stati sottratti N posti, e prima che gli stessi siano sommati ad Y.

Letture non ripetibili

Problema delle letture non ripetibili: avviene se una transazione T_1 legge due volte lo stesso item, ma tra le due letture una transazione T_2 ne ha modificato il valore.

Esempio: durante una prenotazione di posti aerei, un cliente chiede informazioni su più voli. Quando il cliente decide, la transazione deve rileggere il numero di posti disponibili sul volo scelto per completare la prenotazione, ma potrebbe non trovare più la stessa disponibilità.

Recovery

Quando viene inoltrata una transazione, il sistema deve far sì che tutte le operazioni siano completate con successo ed il loro effetto sia registrato permanentemente nel db, oppure la transazione annullata non abbia effetti né sul db né su qualunque altra transazione.

Le failure vengono in genere suddivise in fallimenti di transazione, di sistema e di media.

Possibili ragioni di una failure:

1. Un crash di sistema durante l'esecuzione della transazione.
2. Errore di transazione o di sistema (Esempi: overflow, divisione per zero, valori errati di parametri, ...)
3. Errori locali o condizione eccezionali rilevati dalla transazione. (Esempio: i dati per la transazione possono non essere trovati o essere non validi, tipo un ABORT programmato a fronte di richiesta di un prelievo da un fondo scoperto.)
4. Controllo della concorrenza. (Il metodo di controllo della concorrenza può decidere di abortire la transazione perché viola la serializzabilità o perché varie transazioni sono in deadlock.)
5. Fallimento di disco. (Alcuni blocchi di disco possono perdere i dati per un malfunzionamento in lettura i scrittura, o a causa di un crash della testina del disco.)
6. Problemi fisici e catastrofi (Esempi: fuoco, sabotaggio, furto, caduta di tensione, errato montaggio di nastro da parte dell'operatore, ...)

I problemi 1-4 sono i più frequenti, ed è più facile effettuarne il recovery.

Transazioni

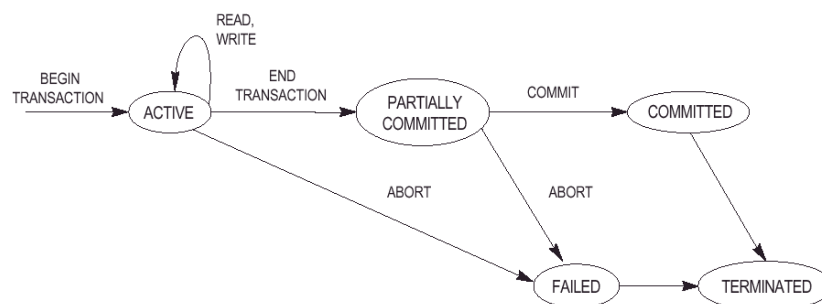
Una transazione è un'unità atomica di lavoro che o è completata nella sua interezza o è integralmente annullata. Per motivi di recovery, il sistema deve tenere traccia dell'inizio e della fine o dell'abort di ogni transazione. Il manager di recovery tiene quindi traccia delle seguenti operazioni:

- BEGIN_TRANSACTION: marca l'inizio dell'esecuzione della transazione.
- READ o WRITE: specifica operazioni di lettura o scrittura sul db, eseguite come parte di una transazione.
- END_TRANSACTION: specifica che le operazioni di READ e WRITE sono finite e marca il limite di fine di esecuzione della transazione.
- COMMIT_TRANSACTION: segnala la fine con successo della transazione, in modo che qualsiasi cambiamento può essere reso permanente, senza possibilità di annullarlo.
- ROLL-BACK (o ABORT): segnala che la transazione è terminata senza successo e tutti i cambiamenti o effetti nel db devono essere annullati.

Operazioni addizionali:

- UNDO: simile al roll-back, eccetto che si applica ad un'operazione singola piuttosto che a una intera transazione.
- REDO: specifica che certe operazioni devono essere ripetute.

Si può usare un diagramma degli stati per evidenziare come evolve lo stato di una transazione:



Nello stato "partially committed", alcune tecniche di recovery richiedono di effettuare dei controlli per garantire che la transazione non influisca con altre.

Per effettuare il recovery di transazioni abortite, il sistema mantiene un log (o journal) per tenere traccia delle operazioni che modificano il database. Il system log contiene quindi informazioni su tutte le transazioni che modificano i valori degli item del database.

Il log è strutturato come una lista di record. In ogni record è memorizzato un ID univoco della transazione T, generato in automatico dal sistema.

Tipi di entry possibili nel log:

- **[start_transaction, T]** la transazione T ha iniziato la sua esecuzione.
- **[write_item, T, X, old_value, new_value]** la transazione T ha cambiato il valore dell'item X da old_value a new_value.
- **[read_item, T, X]** La transazione T ha letto l'item X.
- **[commit, T]** La transazione T è terminata con successo e le modifiche possono essere memorizzate in modo permanente.

- **[abort, T]** La transazione T è fallita.

Il file di log deve essere tenuto su disco. Aggiornare il log implica copiare il blocco dal disco al buffer in memoria, aggiornare il buffer e riscrivere il buffer su disco. Di fronte ad una failure, solo le entry su disco vengono usate nel processo di recovery. Poiché un blocco viene tenuto in memoria finché non è pieno, prima che una transazione raggiunga il punto di commit, ogni parte del log in memoria deve essere scritta (*scrittura forzata* o *force writing*).

Proprietà delle transazioni

Le transazioni dovrebbero possedere alcune proprietà (dette *ACID properties*, dalle loro iniziali):

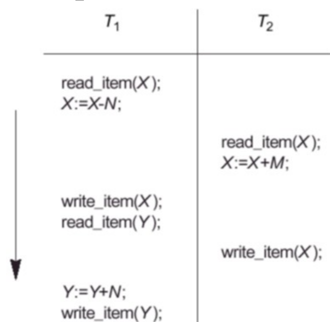
- *Atomicità*: una transazione è un'unità atomica di elaborazione da eseguire o completamente o per niente (responsabilità del recovery subsystem).
- *Consistency preserving*: una transazione deve far passare il database da uno stato consistente ad un altro (responsabilità dei programmatori).
- *Isolation*: Una transazione non deve rendere visibili i suoi aggiornamenti ad altre transazioni finché non è committed (responsabilità del sistema per il controllo della concorrenza)
- *Durability*: Se una transazione cambia il database, e il cambiamento è committed, queste modifiche non devono essere perse a causa di fallimenti successivi (responsabilità del sistema di gestione dell'affidabilità)

Schedule

Informalmente, uno schedule è l'ordine in cui sono eseguite le operazioni di più transazioni processate in modo interleaved.

Formalmente, uno schedule (o storia) S di n transazioni T_1, T_2, ..., T_n è un ordinamento delle operazioni delle transazioni, soggetto al vincolo che per ogni transazione T_i che partecipa in S, le operazioni in T_i in S devono apparire nello stesso ordine di apparizione in T_i. Supponiamo che l'ordinamento delle operazioni in S sia totale.

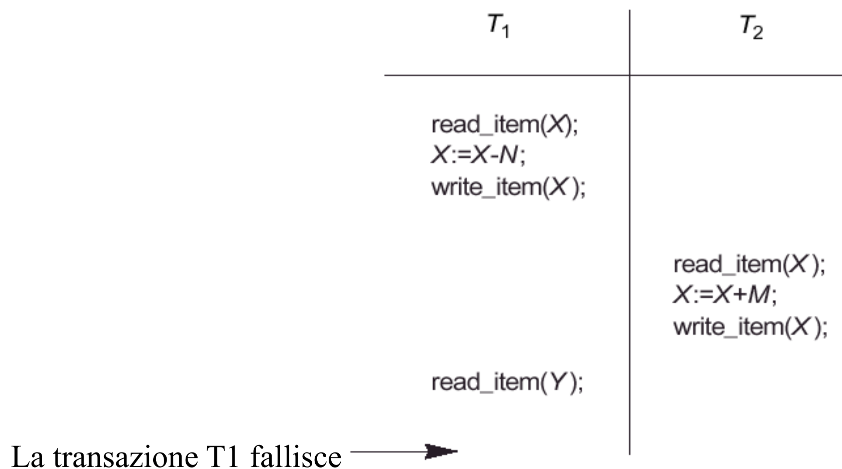
Esempio 1.



Lo schedule per queste transazioni, che chiamiamo S_a, può essere espresso come:

S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$

Esempio 2.



Lo schedule S_b per queste transazioni può essere espresso come:

S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1

Conflitto di operazioni

Due operazioni in uno schedule sono in conflitto se:

1. appartengono a differenti transazioni,
2. accedono allo stesso elemento X,
3. almeno una delle due operazioni è una write_item(X).

Esempio.

S_a': $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1

Generano conflitti	**Non generano conflitti**
$r_2(X)$ e $w_1(X)$	$r_1(x)$ e $r_2(x)$
$w_1(X)$ e $w_2(X)$	$w_2(X)$ e $w_1(Y)$
$r_1(X)$ e $w_2(X)$	

Uno schedule S di n transazioni T_1, T_2, ..., T_n è uno schedule **completo** se valgono le seguenti condizioni:

1. Le operazioni in S sono esattamente quelle in T_1, T_2, ..., T_n, incluso un'operazione di commit o di abort come ultima operazione di ogni transazione in S.
2. Per ogni coppia di operazioni dalla stessa transazione T_i, il loro ordine di occorrenza in S è lo stesso che in T_i.
3. Per ogni coppia di operazioni in conflitto, una deve occorrere prima dell'altra nello schedule.

Uno schedule completo non contiene transazioni attive, perché sono tutte commmitted o aborted.

Dato uno schedule S, si definisce **proiezione committed *C(S)***, uno schedule che contiene solo le

operazioni in S che appartengono a transazioni committed.

È importante caratterizzare i tipi di schedule in base alla possibilità di effettuare recovery. Vorremmo garantire che per una transazione committed non è mai necessario il roll-back. Uno schedule con tale proprietà è detto ***recoverable***. Alternativamente, uno schedule S è detto recoverable se nessuna transazione T in S fa un commit, finché tutte le transazioni T', che hanno scritto un elemento letto da T, hanno fatto un commit.

Esempio.

Sia

$$S_a': r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1$$

S_a' è recoverable, ma soffre del problema dell'aggiornamento perso.

Sia

$$S_c: r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1$$

S_c non è recoverable perché T_2 legge X da T_1 e poi fa il commit prima di T_1. Se T_1 abortisce dopo c_2, allora il valore di X che T_2 legge non è più valido e T_2 deve essere abortito dopo aver fatto commit.

Se lo modifichiamo in

$$r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$$

S_c diviene uno schedule recoverable.

Negli schedule recoverable nessuna transazione commited ha necessità di roll-back. Si possono però avere roll-back in cascata se una transazione non committed legge un dato scritto da una transazione fallita. Uno schedule è detto evitare il roll-back in cascata se ogni transazione nello schedule legge elementi scritti solo da transazioni committed.

Esempio. Sia S_e:

$$r_1(X); w_1(X); r_2(X); r_1(Y); w_2(Y); w_1(Y); c_1; c_2$$

S_e non è cascadeless: T_2 legge X scritto da T_1 prima che T_1 raggiunga il commit o l'abort. Ecco la versione modificata.

$$S'_e: r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); c_2$$

Uno schedule è detto *stretto* se le transazioni non possono né leggere né scrivere un elemento X finché l'ultima transazione che ha scritto X non è completata (con commit o abort). Schedule stretti semplificano il processo di recovery poiché occorre solo ripristinare la *before image* (old_value) di un dato X.

Esempio. Sia S_f:

$$w_1(X,5); w_2(X,8); a_1$$

Supponiamo che inizialmente sia X = 9. S_f è cascadeless ma non stretto. Se T_1 fallisce, la procedura di recovery ripristina il valore di X a 9, anche se è stato modificato da T2.

Serializzabilità di schedule

Oltre a caratterizzare gli schedule in base alla possibilità di recovery, vorremmo classificarli anche in base al loro comportamento in ambiente concorrente. Uno schedule è ***seriale*** se per ogni transazione T nello schedule, tutte le operazioni di T sono eseguite senza interleaving. Altrimenti è non seriale.

Esempio. I due possibili casi di schedule seriali con due transazioni T1 e T2:

a) T1 esegue prima di T2
b) T1 esegue dopo T2

a

T_1	T_2
read_item(X);	
X:=X-N;	
write_item(X);	
read_item(Y);	
Y:=Y+N;	
write_item(Y);	
	read_item(X);
	X:=X+M;
	write_item(X);

b

T_1	T_2
	read_item(X);
	X:=X+M;
	write_item(X);
read_item(X);	
X:=X-N;	
write_item(X);	
read_item(Y);	
Y:=Y+N;	
write_item(Y);	

Gli schedule seriali hanno diversi problemi. Innanzitutto limitano la concorrenza o le operazioni di interleaving: se una transazione aspetta una operazione di I/O, non si può allocare la CPU ad un'altra transazione; se una transazione T dura a lungo, le altre transazioni devono aspettare che finisca. Gli schedule seriali, in pratica, sono *inaccettabili*.

Gli schedule non seriali possono dare i problemi dell'aggiornamento perso, dell'aggiornamento temporaneo, della somma scorretta, etc.

Tuttavia esistono degli schedule non seriali che danno risultati corretti.

– Quali schedule non seriali danno sempre un risultato corretto?
– Quali, invece, danno sempre un risultato scorretto?

Uno schedule S di n transazioni è serializzabile se è "*equivalente*" a qualche schedule seriale delle stesse n transazioni. Dati *n* schedules, abbiamo *n!* possibili seriali.

Quando due schedule possono essere detti "equivalenti"? Due schedule sono detti result equivalent se producono lo stesso stato finale del db. (Non è una definizione accettabile: la produzione dello stesso stato può essere accidentale.)

Esempio. I due schedules sono result equivalent solo se X = 100.

S_1	S_2
read_item(X);	read_item(X);
X:=X+10;	X:=X*1.1;
write_item(X);	write_item(X);

Una definizione più appropriata è quella di conflict equivalent: Due schedule sono conflict equivalent se l'ordine di ogni coppia di operazioni in conflitto è lo stesso in entrambi gli schedule.

Esempio. in S_1 sono presenti $r_1(X)$; $w_2(X)$ e nello schedule S_2 sono in ordine inverso, $w_2(X)$; $r_1(X)$.

Il valore $r_1(X)$ può essere differente nei due schedule.

Uno schedule è <u>conflict serializzabile</u> se è conflict equivalent a qualche schedule seriale S'.

È possibile determinare, per mezzo di un semplice algoritmo, la conflict serializzabilità di uno schedule. Molti metodi per il controllo della concorrenza non testano la serializzabilità; piuttosto usano protocolli che garantiscono che uno schedule sarà serializzabile.

Esempio. Lo schedule D è conflict serializzabile: Read_Item(X) legge il valore di X scritto da T_1.

T_1	T_2
read_item(*X*);	
X:=*X*-*N*;	
write_item(*X*);	
read_item(*Y*);	
Y:=*Y*+*N*;	
write_item(*Y*);	
	read_item(*X*);
	X:=*X*+*M*;
	write_item(*X*);

T_1	T_2
read_item(*X*);	
X:=*X*-*N*;	
write_item(*X*);	
	read_item(*X*);
	X:=*X*+*M*;
	write_item(*X*);
read_item(*Y*);	
Y:=*Y*+*N*;	
write_item(*Y*);	

Schedule D

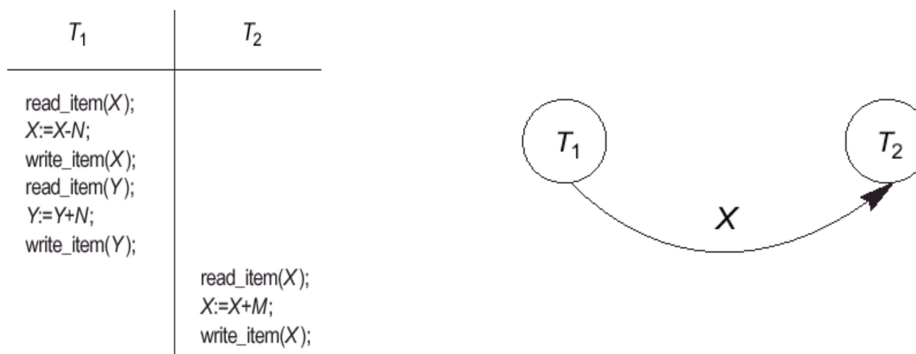
Algoritmo per la serializzabilità

L'algoritmo cerca solo le operazioni di read_item e write_item, per costruire un <u>grafo di precedenza</u> (o <u>grafo di serializzazione</u>). Un grafo di precedenza è un grafo diretto G = (N, E), con un insieme di nodi N={T_1, T_2, ..., T_n} ed un insieme di archi diretti E={e_1, e_2, … , e_m}. Ogni arco è della forma ($T_j \rightarrow T_k$), con 1≤ j, k ≤ n, ed è creato se un operazione in T_j appare nello schedule prima di qualche operazione in conflitto in T_k.

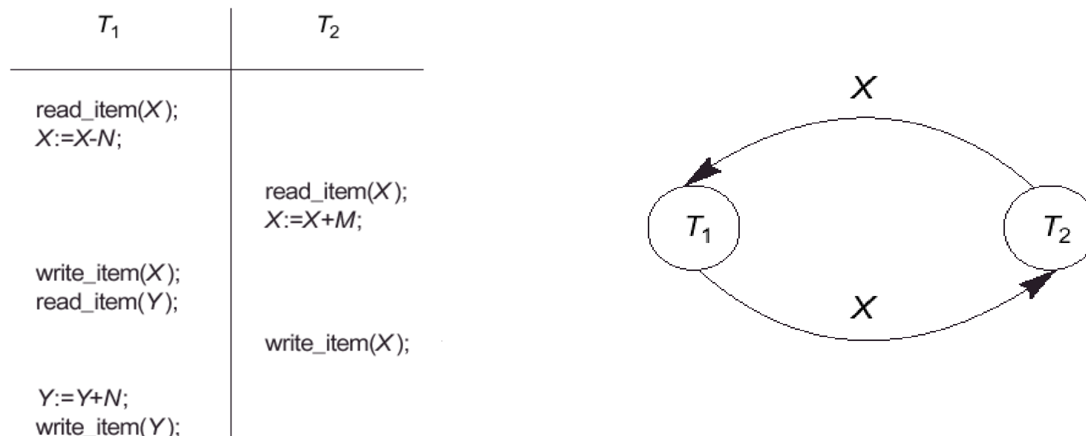
Algoritmo 1:

1. Per ogni transazione Ti nello schedule S, creare un nodo T_i nel grafo;
2. Per ogni caso in S dove T_j esegue una read_item(X) dopo che T_i esegue una write_item(X), creare un arco ($T_i \rightarrow T_j$) nel grafo;
3. Per ogni caso in S dove T_j esegue una write_item(X) dopo che T_i esegue una read_item(X), creare un arco ($T_i \rightarrow T_j$) nel grafo;
4. Per ogni caso in S dove T_j esegue una write_item(X) dopo che Ti esegue una write_item(X), creare un arco ($T_i \rightarrow T_j$) nel grafo;
5. Lo schedule è serializzabile se e solo se il grafo non contiene cicli.

Esempio 1. Uno schedule seriale con due transazioni, ed il relativo grafo di precedenza:



Esempio 2. Uno schedule non seriale con due transazioni, ed il relativo grafo di precedenza che ne evidenzia la non serializzabilità.



Se non ci sono cicli nel grafo di precedenza relativo ad uno schedule S possiamo creare uno schedule seriale equivalente S' ordinando le transazioni che partecipano allo schedule come segue: se esiste un arco fra T_i e T_j; T_i deve apparire prima di T_j nello schedule seriale equivalente.

(vedi esempio slide 71).

Supporto alle transazioni in SQL

Il concetto di transazione in SQL è simile a quanto visto finora: una transazione è una singola unità logica di lavoro con la proprietà dell'atomicità. Di default, in SQL ogni singola istruzione è una transazione. Non esiste uno statement di *Begin_Transaction*, poiché l'inizio di una transazione viene determinato implicitamente. Deve esserne però esplicitata la fine, con le istruzioni COMMIT o ROLLBACK.

Ogni transazione in SQL ha tre caratteristiche, specificate per mezzo dell'istruzione **SET TRANSACTION** che inizia una transazione:

- Modalità di accesso. Specifica se l'accesso ai dati è in sola lettura o in lettura/scrittura.
- Dimensione dell'Area Diagnostica. Specifica lo spazio da usare per informazioni all'utente sull'esecuzione delle transazioni.
- Isolation Level. Specifica la politica di gestione delle transazioni concorrenti.

È molto importante utilizzare due commit (uno prima e uno dopo). SET TRANSACTION deve essere la prima istruzione SQL di una transazione. Il COMMIT appena prima assicura che ciò sia vero. Il COMMIT alla fine rilascia le risorse possedute dalla transazione.

La *modalità di accesso* può essere specificata come READ ONLY o READ WRITE (default). La modalità READ WRITE permette l'esecuzione di comandi di aggiornamento, inserimento, cancellazione e creazione. La modalità READ ONLY serve unicamente per il recupero di dati.

Alcune transazioni effettuano istruzioni di SELECT su diverse tabelle e dovranno vedere dati coerenti, dati che si riferiscono allo stesso istante di tempo. **SET TRANSACTION READ ONLY** specifica questo meccanismo più protetto di gestione dei dati. Nessun comando può modificare i dati di un'area su cui vengono effettuate operazioni di SELECT attraverso questo tipo di transazioni.

L'opzione *dimensione dell'area di diagnosi* **DIAGNOSTIC SIZE n** specifica un valore intero n,

che indica il numero di condizioni che possono essere mantenute contemporaneamente nell'area di diagnosi. Tali condizioni forniscono informazioni di feedback (errori o eccezioni) riguardo i comandi SQL eseguiti più di recente.

L'opzione livello di isolamento è specificata usando l'istruzione **ISOLATION LEVEL <isolamento>** . I possibili valori sono:

- READ UNCOMMITTED
- READ COMMITTED
- REPETEABLE READ
- SERIALIZABLE

SERIALIZABLE è il livello predefinito.

Oltre alle già viste violazioni di "letture sporche" e "letture non ripetibili", usando SQL può sorgere il problema delle "letture fantasma". Supponiamo che una transazione T_1 legga una serie di righe basate su una condizione di WHERE. Se una transazione T_2 aggiunge dei valori che soddisfano la condizione di WHERE, una riesecuzione di T_1 vedrà delle righe nuove (*fantasma*), non presenti in precedenza.

Violazioni basate sui livelli di isolamento di SQL

	Tipo di violazione		
Livello di isolamento	**Lettura sporca**	**Lettura non ripetibile**	**Fantasma**
READ UNCOMMITTED	SI	SI	SI
READ COMMITTED	NO	SI	SI
REPEATABLE READ	NO	NO	SI
SERIALIZABLE	NO	NO	NO

Esempio di transazione SQL.

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTICS SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
    VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
    SET SALARY = SALARY * 1.1 WHERE DNO = 2;
EXEC SQL COMMIT;
```

```
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ...
```

La transazione produce prima l'inserimento di una nuova riga nella tabella EMPLOYEE e successivamente esegue l'aggiornamento dello stipendio di tutti gli impiegati che lavorano nel reparto 2. In caso di errore in una qualsiasi istruzione SQL, l’intera transazione viene annullata (rollback). Ogni valore di stipendio aggiornato viene ripristinato al suo valore precedente e la nuova riga viene rimossa.