

## TRANSAZIONI

- Per mantenere le informazioni consistenti è necessario controllare opportunamente le sequenze di accessi e aggiornamenti ai dati.
- Tali sequenze sono dette **transazioni**.
- Ogni transazione è eseguita o completamente (cioè effettua il commit), oppure per nulla (cioè effettua l'abort) se si verifica un qualche errore (hardware o software) durante l'esecuzione.
- Necessità di garantire che le transazioni eseguite concorrentemente si comportino come se fossero state eseguite in sequenza.
- Necessità di tecniche per ripristinare uno stato corretto della base di dati a fronte di malfunzionamenti di sistema.

## TRANSAZIONI: CONCETTI DI BASE

Gli utenti interagiscono con la base di dati attraverso programmi applicativi ai quali viene dato il nome di *transazioni*.

Il termine *transazione* viene utilizzato con diversi significati:

1. la **richiesta o il messaggio di input** che avvia l'operazione desiderata dall'utente;
2. **tutti gli effetti** dell'esecuzione dell'operazione che l'utente ha richiesto;
3. **tutti i programmi** che eseguono l'operazione richiesta dall'utente.

Tale diversità dipende dal fatto che l'uso di transazioni coinvolge diverse tipologie di utenti.

## TRANSAZIONI: CONCETTI DI BASE

Nel seguito, si adotterà la seconda definizione:

*Una transazione è un insieme parzialmente ordinato di operazioni di lettura e scrittura; essa costituisce l'effetto dell'esecuzione di programmi che effettuano le funzioni desiderate dagli utenti.*

L'insieme di operazioni che costituiscono una transazione deve soddisfare alcune proprietà, note come **ACIDity property** (da *Atomicità, Consistenza, Isolamento, Durabilità*).

## TRANSAZIONI: ATOMICITÀ

- è detta anche proprietà *tutto-o-niente*
- tutte le operazioni di una transazione devono essere trattate come una singola unità: o vengono eseguite tutte, oppure non ne viene eseguita alcuna
- l'atomicità delle transazioni è assicurata dal sottosistema di *ripristino* (recovery).

## TRANSAZIONI: CONSISTENZA

- una transazione deve agire sulla base di dati in modo corretto
- se viene eseguita su una base di dati in assenza di altre transazioni, la transazione trasforma la base di dati da uno stato consistente (cioè che riflette lo stato reale del mondo che la base di dati deve modellare) ad un altro stato ancora consistente
- l'esecuzione di un insieme di transazioni corrette e concorrenti deve a sua volta mantenere consistente la base di dati
- il sottosistema di *controllo della concorrenza* (concurrency control) sincronizza le transazioni concorrenti in modo da assicurare esecuzioni concorrenti libere da interferenze.

## **TRANSAZIONI: ISOLAMENTO**

- ogni transazione deve sempre osservare una base di dati consistente, cioè, non può leggere risultati intermedi di altre transazioni
- la proprietà di isolamento è assicurata dal sottosistema di controllo della concorrenza che isola gli effetti di una transazione fino alla terminazione della stessa.

## **TRANSAZIONI: DURABILITÀ (PERSISTENZA)**

- i risultati di una transazione terminata con successo devono essere resi permanenti nella base di dati nonostante possibili malfunzionamenti del sistema
- la persistenza è assicurata dal sottosistema di ripristino
- tale sottosistema può inoltre fornire misure aggiuntive, quali back-up su supporti diversi e journaling delle transazioni, per garantire la durabilità anche a fronte di guasti ai dispositivi di memorizzazione.

## TRANSAZIONI: ESEMPIO

Si consideri una transazione bancaria di addebito sul conto corrente del signor Rossi.

- è atomica se effettua entrambe le procedure di prelevamento ed aggiornamento del conto corrente
- è consistente se l'importo prelevato è lo stesso addebitato sul conto
- è isolata se il programma transazione non si accorge di altri programmi che leggono e scrivono sul conto corrente del signor Rossi
- è duratura (persistente) se, terminata la transazione, il saldo del conto riflette il prelievo effettuato.



# TRANSAZIONI: PROPRIETÀ ACID

Le proprietà ACID vengono assicurate utilizzando due insiemi distinti di algoritmi o protocolli, che assicurano:

## 1. l'atomicità dell'esecuzione

mantenere la consistenza globale della base di dati e quindi assicurare la proprietà di consistenza delle transazioni (anche concorrenti)

protocolli di *controllo della concorrenza*

## 2. l'atomicità del fallimento

assicura l'atomicità, l'isolamento e la persistenza

protocolli di *ripristino*.

## TRANSAZIONI FLAT

- tipo di transazione più semplice
- usate in tutti i DBMS disponibili in commercio
- tecniche di implementazione e limitazioni sono ben note
- un solo *livello di controllo* a cui appartengono tutte le istruzioni eseguite
- transazioni semplici, di breve durata, senza alcuna struttura gerarchica

## TRANSAZIONI FLAT

Tutte le istruzioni eseguite devono essere contenute tra le istruzioni *BeginWork* e *CommitWork*.

- l'istruzione *BeginWork* dichiara l'inizio di una transazione flat
- l'istruzione *CommitWork* è invocata per indicare che il sistema ha raggiunto un nuovo stato consistente.

La transazione può terminare la propria esecuzione con successo (commit) e rendere definitivi i cambiamenti prodotti sulla base di dati dalle istruzioni eseguite tra *BeginWork* e *CommitWork*, oppure sarà disfatta (cioè i suoi effetti saranno annullati) e tutti gli aggiornamenti eseguiti andranno persi (abort).

In questo caso, si dice che viene eseguito il *rollback* della transazione.

## TRANSAZIONI FLAT

- una transazione flat  $T_i$  è un insieme di operazioni sui dati su cui è definito un ordinamento parziale  $<_i$ , che specifica l'ordine con cui le operazioni vengono eseguite
- una transazione può, in modo esclusivo, effettuare abort oppure commit
- l'istruzione di abort o di commit deve essere l'ultima istruzione effettuata dalla transazione
- fissato un dato  $x$ , le operazioni di scrittura e lettura su tale dato sono ordinabili tramite la relazione di ordinamento  $<_i$  [cioè una transazione che vuole leggere (scrivere) un dato  $x$  deve attendere che qualsiasi altra operazione di scrittura (lettura) termini, prima di poter eseguire la lettura (scrittura) richiesta].

## TRANSAZIONI FLAT: SUPPORTO IN SQL

- i vari DBMS forniscono specifiche istruzioni SQL per supportare l'uso di transazioni flat
- tali istruzioni sono invocate all'interno di programmi applicativi e consentono al programmatore di identificare l'inizio e la fine di una transazione
- l'istruzione `exec sql connect:user_name identified by :user_pws` (cioè l'istruzione di connessione) avvia la transazione, cioè implementa la primitiva `BeginWork`
- l'istruzione `exec sql commit work` esegue il commit di una transazione
- l'istruzione `exec sql rollback` esegue il rollback di una transazione.

## TRANSAZIONI FLAT: LIMITAZIONI

- una transazione flat non permette di effettuare il commit oppure l'abort di parti distinte di essa nè di eseguire modifiche sulla base di dati in molteplici passi, effettuando molteplici commit
- esempio di problema nell'uso di transazioni flat: pianificazione di un viaggio Pittsburgh-Finale Ligure

BeginWork

prenota il volo da Pittsburgh a Londra

prenota il volo da Londra a Milano, stesso giorno

prenota il volo da Milano a Genova, stesso giorno

- proposte di estensioni: transazioni concatenate, transazioni annidate, transazioni multilivello, ...

trade-off potere espressivo - semplicità

## CONTROLLO DELLA CONCORRENZA

**Scopo:** garantire l'integrità della base di dati in presenza di accessi concorrenti da parte di più utenti

necessità di sincronizzare le transazioni eseguite concorrentemente.

**Esempio di problema:** Base di dati che organizza le informazioni sui conti dei clienti di una banca. Il sig. Rossi è titolare di due conti: un conto corrente (intestato anche alla sig.ra Rossi) e un libretto di risparmio, i cui saldi sono Lit. 100000 e Lit. 1000000. Con la transazione  $T_1$  il sig. Rossi trasferisce Lit. 150000 dal libretto di risparmio al conto corrente. Contemporaneamente con la transazione  $T_2$  la sig.ra Rossi deposita Lit. 500000 sul conto corrente.

## CONTROLLO DELLA CONCORRENZA: ESEMPIO

$T_1$

Read l\_r;

l\_r = l\_r - 150000;

Write l\_r;

Read c/c;

c/c = c/c + 150000;

Write c/c;

Commit  $T_1$ .

$T_2$

Read c/c;

c/c = c/c + 500000;

Write c/c;

Commit  $T_2$ .

La somma depositata da  $T_2$  è persa (*lost update*)  $\Rightarrow$  non si ha l'effetto voluto sulla base di dati.



## SERIALIZZABILITÀ

- l'esecuzione concorrente di più transazioni genera un'alternanza di computazioni da parte delle varie transazioni, detta *interleaving*

la sequenza di operazioni generata viene definita *schedule*

- l'interleaving tra le transazioni  $T_1$  e  $T_2$  nell'esempio produce uno stato della base di dati scorretto; si sarebbe ottenuto uno stato corretto se ciascuna transazione fosse stata eseguita da sola o se le due transazioni fossero state eseguite l'una dopo l'altra, consecutivamente
- uno **schedule seriale** è uno schedule in cui le transazioni vengono eseguite in sequenza.

## SCHEDULE SERIALE: ESEMPIO

### Schedule 1

$T_1$

Read l\_r;

$l_r = l_r - 150000$ ;

Write l\_r;

Read c/c;

$c/c = c/c + 150000$ ;

Write c/c;

Commit  $T_1$

$T_2$

Read c/c;

$c/c = c/c + 500000$ ;

Write c/c;

Commit  $T_2$

Date  $n$  transazioni, si possono costruire  $n!$  diversi schedule seriali.

## SCHEDULE EQUIVALENTI

Due schedule  $S_1$  e  $S_2$  sono computazionalmente **equivalenti**, indicato con  $S_1 \equiv S_2$ , se:

1. l'insieme delle transazioni considerate in  $S_1$  coincide con l'insieme delle transazioni considerate in  $S_2$ ;
2. per ogni dato  $Q$ , se in  $S_1$  la transazione  $T_i$  esegue  $read(Q)$  e il valore di  $Q$  letto da  $T_i$  era stato modificato da  $T_k$ , lo stesso avviene in  $S_2$ ;
3. per ogni dato  $Q$ , se in  $S_1$  la transazione  $T_i$  è l'ultima transazione ad eseguire l'operazione  $write(Q)$ , lo stesso avviene in  $S_2$ .

## SCHEDULE EQUIVALENTI

- la condizione (1) assicura che lo stesso insieme di transazioni partecipi ad entrambi gli schedule
- la condizione (2) assicura che ogni transazione riceva gli stessi valori in ingresso in entrambi gli schedule (e quindi esegua la stessa computazione)
- la condizione (3), insieme alla condizione (2), assicura che entrambi gli schedule producano lo stesso stato finale della base di dati.

## SCHEDULE EQUIVALENTI: ESEMPIO

### Schedule 2

$T_1$

Read l\_r;

$l_r = l_r - 150000;$

Write l\_r;

Read c/c;

$c/c = c/c + 150000;$

Write c/c;

Commit  $T_1$

$T_2$

Read c/c;

$c/c = c/c + 500000;$

Write c/c;

Commit  $T_2$ .

è equivalente allo schedule seriale  $T_2, T_1$ .

## SCHEDULE SERIALIZZABILI

- uno schedule  $S$  è **serializzabile** se e solo se esiste uno schedule seriale  $S'$  tale che  $S \equiv S'$
- lo schedule 2 è serializzabile
- due algoritmi per la verifica della serializzabilità di uno schedule:
  1. assume che le transazioni debbano leggere i dati prima di modificarli (ogni operazione di write è sempre preceduta da un'operazione di read)  
semplice ed efficiente
  2. non fa questa assunzione: le transazioni non devono necessariamente leggere i dati prima di modificarli.

## ESEMPIO DI SCHEDULE NON SERIALIZZABILE

### Schedule 3

$T_1$

Read l\_r;

l\_r = l\_r - 150000;

Write l\_r;

Read c/c;

c/c = c/c + 150000;

Write c/c;

Commit  $T_1$

$T_2$

Read l\_r;

l\_r = l\_r - 100000;

Write l\_r;

Read c/c;

c/c = c/c + 100000;

Write c/c;

Commit  $T_2$

## ESEMPIO DI SCHEDULE NON SERIALIZZABILE

- le transazioni  $T_1$  e  $T_2$  trasferiscono rispettivamente 150000 e 100000 dal libretto di risparmio al conto corrente
- se lo schedule fosse corretto, la somma esistente sul conto corrente sommata alla somma esistente sul libretto di risparmio sarebbe invariata
- si supponga che prima di iniziare l'esecuzione,  $l_r = 400000$  e  $c/c = 200000$   
al termine dell'esecuzione,  $l_r = 250000$   
mentre  $c/c = 300000$   
 $\Rightarrow$  la somma delle due quantità non si è mantenuta costante
- vi sono dei lost update: il decremento di  $l_r$  di  $T_2$  e l'incremento di  $c/c$  di  $T_1$ .



## ALGORITMO 1: Scritture precedute da letture

Dato uno schedule  $S$  a cui partecipano le transazioni  $T_1, \dots, T_n$ , si costruisce un grafo diretto, detto *grafo di precedenza*  $G_S = (V, E)$ , definito come segue:

1. L'insieme dei vertici  $V$  consiste di tutte le transazioni che partecipano allo schedule  $S$ .
2. L'insieme degli archi  $E$  consiste di tutti gli archi  $T_i \rightarrow T_j$ ,  $i, j = 1, \dots, n$ , tali che  $T_i$  e  $T_j$  verificano una delle due seguenti condizioni:
  - (a)  $T_i$  esegue  $write(Q)$  prima che  $T_j$  esegua  $read(Q)$ .
  - (b)  $T_i$  esegue  $read(Q)$  prima che  $T_j$  esegua  $write(Q)$ .

## ALGORITMO 1: Scritture precedute da letture

Se esiste un arco da  $T_i$  a  $T_j$  allora nello schedule seriale  $S'$ , equivalente a  $S$ ,  $T_i$  deve essere eseguito prima di  $T_j$

$\Rightarrow$  lo schedule  $S$  è serializzabile sse il grafo  $G_S$  è aciclico

l'ordine di serializzabilità si ottiene con un ordinamento topologico:

sia  $L$  una lista vuota

1. sia  $T_i$  un nodo in  $G_S$  senza archi entranti (esiste sicuramente perchè il grafo è aciclico)
2. si appende  $T_i$  alla lista  $L$
3. si rimuove  $T_i$  dal grafo e si rimuovono tutti gli archi uscenti da  $T_i$

si rieseguono i passi 1, 2, 3 fino a che il grafo è vuoto

il costo dell'algoritmo è nell'ordine di  $n^2$ , dove  $n$  è il numero di nodi (e cioè il numero di transazioni) del grafo.

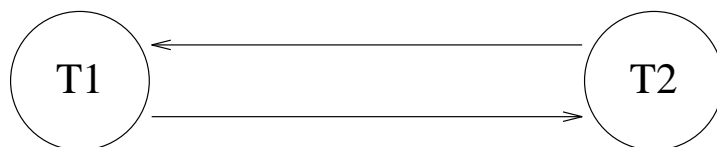
## ALGORITMO 1: Esempi

- il grafo corrispondente allo schedule 2 è



il grafo non ha cicli, quindi lo schedule 2 è serializzabile

- il grafo corrispondente allo schedule 3 è



il grafo ha un ciclo, quindi lo schedule 3 non è serializzabile.

## ALGORITMO 2: Scrittura non condizionata

- si considerano scritture non necessariamente precedute da letture

un tipico esempio e' l'istruzione SQL  
UPDATE Impiegati SET Salario = 1000;

- in questo caso non esiste un algoritmo efficiente (cioè polinomiale)

### Esempio:

#### Schedule 4

$T_1$

Read  $I_r$ ;

Write  $I_r$ ;

Commit  $T_1$

$T_2$

Write  $I_r$ ;

Commit  $T_2$

## ALGORITMO 2: Scrittura non condizionata

### Esempio - segue

- lo schedule 4 non è serializzabile

non è infatti equivalente né allo schedule seriale che esegue prima  $T_1$  e poi  $T_2$  né allo schedule seriale che esegue prima  $T_2$  e poi  $T_1$

- $T_1$  legge  $l_r$  prima che sia modificato da  $T_2 \Rightarrow T_1$  deve precedere  $T_2$  nello schedule seriale equivalente

$T_1$  esegue l'ultima operazione di scrittura  
 $\Rightarrow T_1$  deve seguirere  $T_2$  nello schedule seriale equivalente

si ha quindi una contraddizione.

## ALGORITMO 2: Scrittura non condizionata

Prima possibilità: modificare l'algoritmo 1 per confrontare operazioni di scrittura introducendo un arco  $T_i \rightarrow T_j$  nel grafo  $G_S$  se, nello schedule  $S$ ,  $T_j$  scrive il dato  $Q$  dopo che  $T_i$  scrive  $Q$ .

### Problema:

#### Schedule 5

$T_1$

Read  $l_r$ ;

Write  $l_r$ ;

Commit  $T_1$

$T_2$

Write  $l_r$ ;

Commit  $T_2$

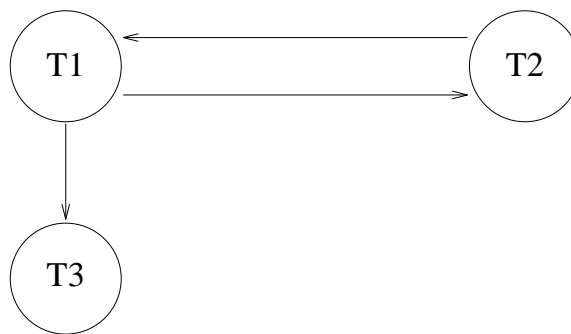
$T_3$

Write  $l_r$ ;

Commit  $T_3$

## ALGORITMO 2: Scrittura non condizionata

il grafo ottenuto con l'algoritmo 1 modificato è



il grafo è ciclico, quindi lo schedule sembrerebbe non serializzabile

invece è equivalente allo schedule seriale  $T_1, T_2, T_3$

i valori di  $l_r$  scritti da  $T_1$  e  $T_2$  non sono letti da  $T_3$  (*useless writes*).

## ALGORITMO 2: Scrittura non condizionata

Idea: usare archi etichettati, dove archi con la stessa etichetta rappresentano scelte alternative  $\Rightarrow$  *poligrafo*

- Sia  $\{T_0, \dots, T_n\}$  l'insieme di transazioni considerate in uno schedule  $S$ . Siano  $T_b$  e  $T_f$  due transazioni *dummy*, tali che  $T_b$  esegue un'operazione di  $write(Q)$  per ogni dato  $Q$  riferito in  $S$  e  $T_f$  esegue un'operazione di  $read(Q)$  per ogni dato  $Q$  riferito in  $S$ .
- Si costruisce un nuovo schedule  $S'$  ottenuto da  $S$  eseguendo  $T_b$  come prima operazione ed eseguendo  $T_f$  come ultima operazione.
- Il grafo di precedenza con etichette per  $S'$  si costruisce come segue:



1. Si introduce un nodo per ogni transazione in  $S'$  e un arco  $T_i \xrightarrow{0} T_j$  per ogni coppia di transazioni  $T_i$  e  $T_j$  tali che  $T_j$  legge il valore di un dato  $Q$  scritto da  $T_i$ .
  
2. Si rimuovono tutti gli archi incidenti su nodi che rappresentano transazioni *useless*. Una transazione è *useless* se non esiste alcun cammino, nel grafo di precedenza, che connette  $T_i$  a  $T_f$ .
  
3. Per ogni dato  $Q$  tale che:
  - $T_j$  legge il valore di  $Q$  scritto da  $T_i$ ;
  - $T_k$  esegue  $write(Q)$  e  $T_k \neq T_b$ :
  1. se  $T_i = T_b$  e  $T_j \neq T_f$ , si inserisce l'arco  $T_j \xrightarrow{0} T_k$ ;
  2. se  $T_i \neq T_b$  e  $T_j = T_f$ , si inserisce l'arco  $T_k \xrightarrow{0} T_i$ ;
  3. se  $T_i \neq T_b$  e  $T_j \neq T_f$ , si inserisce la coppia di archi  $T_k \xrightarrow{p} T_i$  e  $T_j \xrightarrow{p} T_k$ , con  $p$  numero intero maggiore di 0 e mai usato prima nell'etichettare gli archi.

## ALGORITMO 2

- regola (c): se una transazione  $T_i$  scrive un dato  $Q$  che viene letto da una transazione  $T_j$ , una transazione  $T_k$  non può essere inserita tra  $T_i$  e  $T_j$

quindi  $T_k$  può essere inserita nell'ordinamento seriale o prima di  $T_i$  o dopo  $T_j$

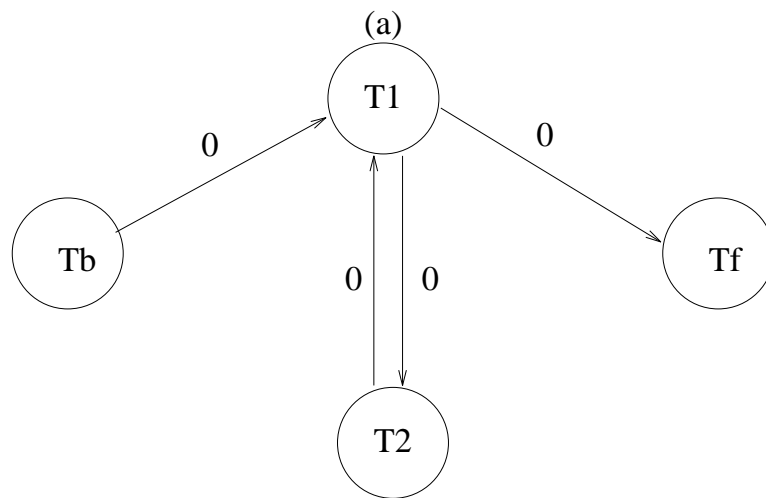
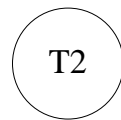
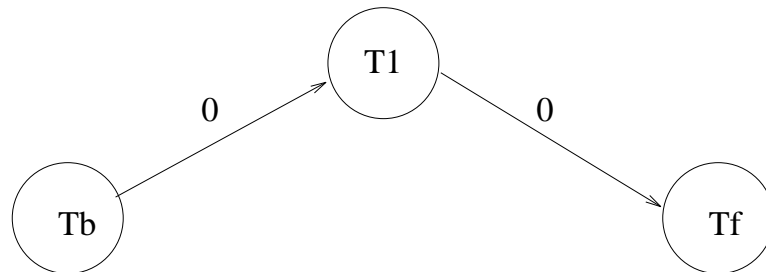
- l'inserzione di due archi con etichette diverse da zero denota che sono possibili due scelte

è necessario esaminare ogni possibile grafo che si ottiene scegliendo uno solo degli archi che hanno la stessa etichetta

se almeno uno di tali grafi è aciclico, lo schedule corrispondente è serializzabile

- se ci sono  $n$  coppie di archi con la stessa etichetta i possibili grafi sono  $2^n$ ; il problema è NP-completo.

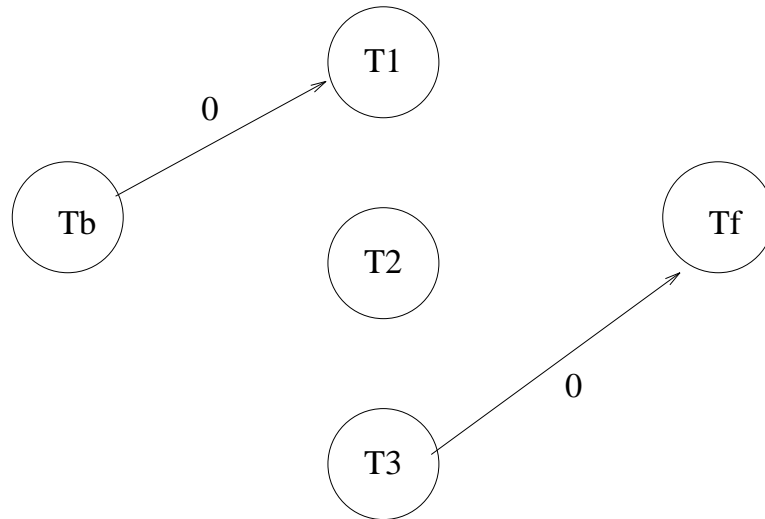
## Grafo dello Schedule 4



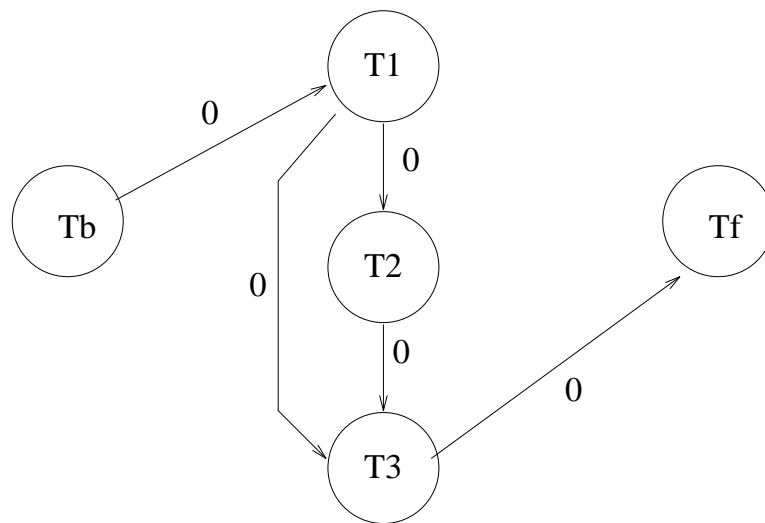
(b)

poichè il grafo è ciclico e tutti gli archi hanno etichetta 0 lo schedule non è serializzabile.

## Grafo dello schedule 5



(a)



(b)

poiché il grafo non ha cicli, lo schedule è serializzabile ed è equivalente allo schedule seriale  $T_1, T_2, T_3$ .

## ALGORITMO 2: Esempio

### Schedule 6

$T_1$

Read l<sub>r</sub>;

Write l<sub>r</sub>;

Commit  $T_1$

$T_2$

Write l<sub>r</sub>;

Commit  $T_2$

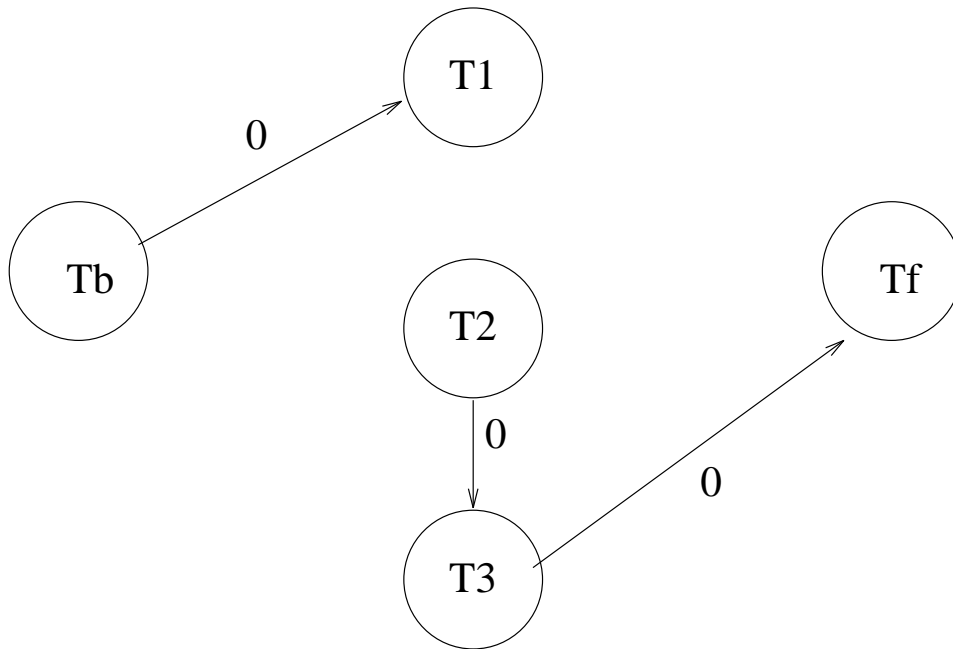
$T_3$

Read l<sub>r</sub>;

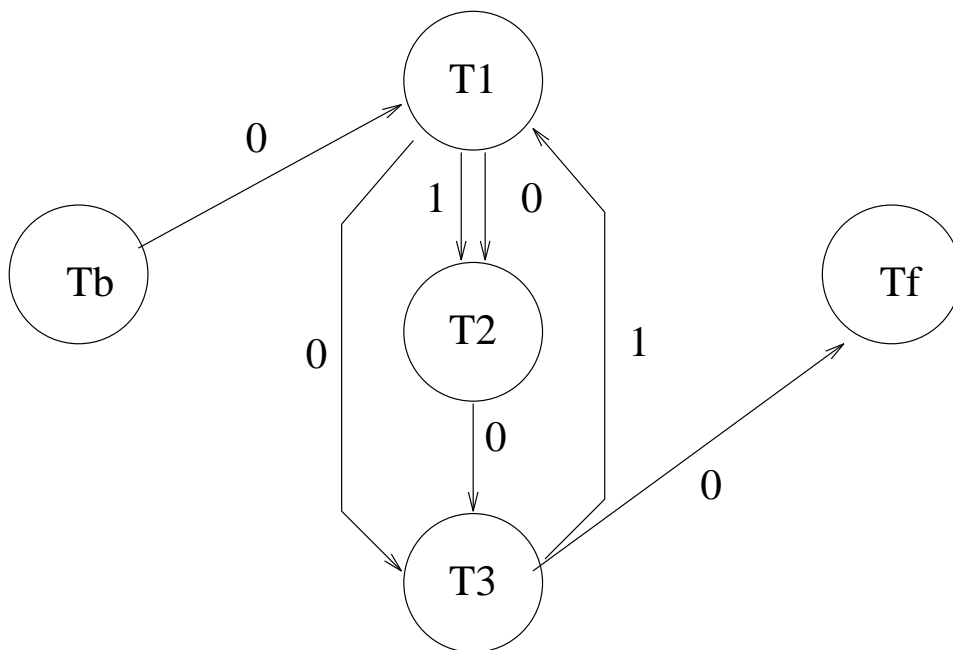
Write l<sub>r</sub>;

Commit  $T_3$

## Poligrafo dello schedule 6

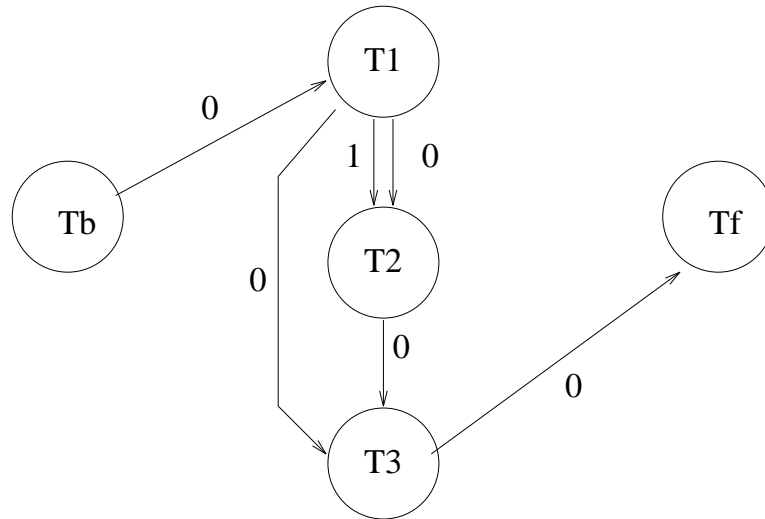


(a)

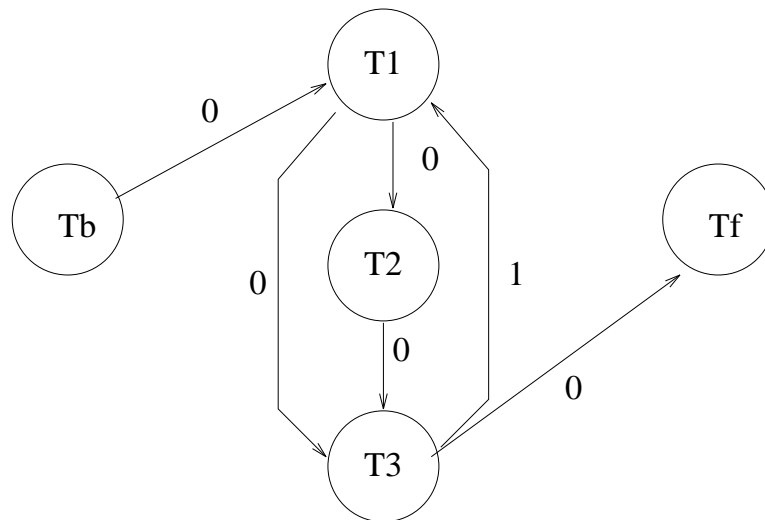


(b)

## Grafi dello schedule 6



(a)



(b)

poiché il grafo (a) è aciclico, lo schedule 6 è serializzabile

lo schedule seriale equivalente è  $T_1, T_2, T_3$ .

## PROTOCOLLI DI CONTROLLO DELLA CONCORRENZA

- abbiamo visto due algoritmi per determinare se uno schedule è serializzabile  
tali algoritmi assumono che lo schedule sia noto
- per assicurare la serializzabilità durante l'esecuzione di più transazioni si utilizza un protocollo di controllo della concorrenza
- tra i protocolli proposti i più noti sono il protocollo **two-phase locking** e il protocollo **timestamp ordering**.



## PROTOCOLLI BASATI SU LOCK

- idea base: ritardare l'esecuzione di operazioni in conflitto imponendo che le transazioni pongano dei blocchi (*lock*) sui dati per poter effettuare operazioni di lettura e scrittura
- due operazioni si dicono in conflitto se almeno una delle due è un'operazione di scrittura
- una transazione può accedere ad un dato solo se ha un lock su quel dato.

## PROTOCOLLI BASATI SU LOCK

due tipi di lock:

**shared:** se una transazione  $T$  ha ottenuto un lock con modalità shared (condivisa) sul dato  $Q$ , allora  $T$  può leggere questo dato ma non può modificarlo

**exclusive** se una transazione  $T$  ha ottenuto un lock con modalità exclusive (esclusiva) sul dato  $Q$ , allora  $T$  può sia leggere che modificare  $Q$ .

uno shared lock sul dato  $Q$  viene richiesto con l'istruzione  $L_s(Q)$

un exclusive lock viene richiesto con l'istruzione  $L_x(Q)$

il lock su un dato  $Q$  è rilasciato con l'istruzione  $U_n(Q)$ .

## PROTOCOLLI BASATI SU LOCK

matrice di compatibilità dei lock:

	<b>Shared</b>	<b>Exclusive</b>
<b>Shared</b>	<i>vero</i>	<i>falso</i>
<b>Exclusive</b>	<i>vero</i>	<i>falso</i>

- se  $T_i$  richiede un lock di modalità A sul dato  $Q$  su cui  $T_j$  ( $T_i \neq T_j$ ) possiede un lock di modalità B, a  $T_i$  può essere concesso immediatamente un lock su  $Q$  se il lock di modalità A è *compatibile* con il lock di modalità B.

## PROTOCOLLO TWO-PHASE LOCKING

- ogni transazione deve effettuare richieste di lock e unlock in due fasi distinte:

*fase di acquisizione* in cui una transazione può ottenere dei lock, ma non può rilasciarne

*fase di rilascio* in cui una transazione può rilasciare dei lock, ma non può ottenerne di nuovi

- inizialmente, una transazione si trova nella fase di acquisizione e acquisisce i lock necessari

non appena la transazione rilascia un lock, inizia la fase di rilascio e non può più essere richiesto alcun lock.

## PROTOCOLLO TWO-PHASE LOCKING

- il protocollo two-phase locking assicura la serializzabilità, ma è suscettibile di **dead-lock**

un sistema si dice in deadlock se esiste un insieme di transazioni tali che ogni transazione dell'insieme attende un'altra transazione dell'insieme per completare la propria esecuzione

- *prevenzione* del deadlock: locking statico [tutti i lock necessari richiesti prima dell'inizio della transazione], ordinamento parziale sui dati, schemi *wait-die* e *wound-wait*  
*rilevazione* (wait-for-graph) e *risoluzione* del deadlock.

## PROTOCOLLO TWO-PHASE LOCKING

### Schedule 7

$T_1$	$T_2$
Read a;	Read a;
Read b;	Read b;
Read c;	Commit $T_2$ .
Write a;	
Commit $T_1$	

$T_1$  deve acquisire un exclusive lock sul dato  $a$   
 $\Rightarrow$  non è ammissibile alcuna esecuzione con-  
corrente delle due transazioni

in realtà  $T_1$  ha bisogno dell'exclusive lock sul  
dato  $a$  solo alla fine  $\Rightarrow$  potrebbe acquisire  
inizialmente uno shared lock sul dato  $a$ , pro-  
muovendolo poi ad exclusive lock.

## PROTOCOLLO TWO-PHASE LOCKING

- per aumentare il grado di parallelismo viene consentito l'uso di **conversioni**:

uno shared lock può essere promosso a exclusive lock e un exclusive lock può essere regredito ad uno shared lock

- la promozione di lock può avvenire solo durante la fase di acquisizione, la regressione solo durante la fase di rilascio.

## PROTOCOLLO TIMESTAMP ORDERING

- assegna un timestamp ad ogni transazione e ad ogni dato è associato il timestamp dell' ultima transazione che l'ha letto o scritto
- nei protocolli di locking, l'ordine di esecuzione di transazioni in conflitto è determinato in fase di esecuzione, al tentativo di acquisire un lock incompatibile  
i protocolli *timestamp ordering* selezionano tale ordine in anticipo
- ad ogni transazione  $T_i$  è associata un' etichetta  $TS(T_i)$ , assegnata dal sistema prima che  $T_i$  inizi la propria esecuzione  
l'assegnazione di etichette è monotona (clock di sistema o contatore logico).



## PROTOCOLLO TIMESTAMP ORDERING

- le etichette delle transazioni determinano l'ordine di serializzabilità:

se  $TS(T_i) < TS(T_j)$  lo schedule prodotto è equivalente ad uno schedule seriale in cui  $T_i$  precede  $T_j$

- ad ogni dato  $Q$  si associa:

**W-timestamp(Q):** il timestamp più alto fra tutte le transazioni che hanno eseguito  $write(Q)$  con successo

**R-timestamp(Q):** il timestamp più alto fra tutte le transazioni che hanno eseguito  $read(Q)$  con successo

aggiornate ogni volta che si esegue  $write(Q)$  o  $read(Q)$ .

- se  $T_i$  vuole eseguire  $\text{read}(Q)$ :
  - se  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , la lettura è rifiutata e viene effettuato il rollback di  $T_i$  [il valore è stato sovrascritto]
  - se  $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$ , la lettura viene eseguita e  $\text{R-timestamp}(Q) = \max(\text{R-timestamp}(Q), \text{TS}(T_i))$
  
- se  $T_i$  vuole eseguire  $\text{write}(Q)$ :
  - se  $\text{TS}(T_i) < \text{R-timestamp}(Q)$ , la scrittura è rifiutata e viene effettuato il rollback di  $T_i$  [il valore è già stato letto]
  - se  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , la scrittura viene ignorata [scrittura di un valore obsoleto]
  - altrimenti, la scrittura viene eseguita e  $\text{W-timestamp}(Q) = \max(\text{W-timestamp}(Q), \text{TS}(T_i))$ .

## PROTOCOLLO TIMESTAMP ORDERING

- se si effettua il rollback di una transazione  $T_i$ , le si assegna un nuovo timestamp e si effettua il restart
- assicura la serializzabilità e l'assenza di deadlock
- possibile problema: rollback a cascata

si può controllare che una transazione legga solo valori modificati da transazioni che hanno terminato con successo (bit di commit).

## ESEMPIO

### Schedule 8

$T_1$	$T_2$
Read b;	Read b;
Read a;	$b := b - 50$ ;
Display $a + b$ ;	Write b;
Commit $T_1$	Read a;
	$a := a + 50$ ;
	Write a;
	Display $a + b$ ;
	Commit $T_2$

$TS(T_1) = 100$ ,  $TS(T_2) = 102$ , R-timestamp(a) = 80, W-timestamp(a) = 80, R-timestamp(b) = 90, W-timestamp(b) = 90

Read(b) della transazione  $T_1$ , poi tutte le operazioni della transazione  $T_2$ , poi Read(a) della transazione  $T_1 \Rightarrow$  rollback

lo schedule proposto non è serializzabile

## PROTOCOLLI DI COMMIT

Tre principali tipi di malfunzionamento:

- **malfunzionamenti del disco:** le informazioni residenti su disco vengono perse (rottura della testina, errori durante il trasferimento dei dati)
- **malfunzionamenti di alimentazione:** le informazioni memorizzate in memoria centrale e nei registri vengono perse
- **errori nel software:** si possono generare risultati scorretti e il sistema potrebbe essere in uno stato inconsistente (errori logici ed errori di sistema).

Il *sottosistema di recovery* (ripristino) deve identificare i malfunzionamenti e ripristinare la base di dati allo stato (consistente) precedente il malfunzionamento.

# CLASSIFICAZIONE DELLE MEMORIE

- **memoria volatile** le informazioni contenute vengono perse in caso di cadute di sistema  
esempi: memoria principale e cache
- **memoria non volatile** le informazioni contenute sopravvivono a cadute di sistema, possono però essere perse a causa di altri malfunzionamenti  
esempi: disco e nastri magnetici
- **memoria stabile** le informazioni contenute non possono essere perse (astrazione teorica)

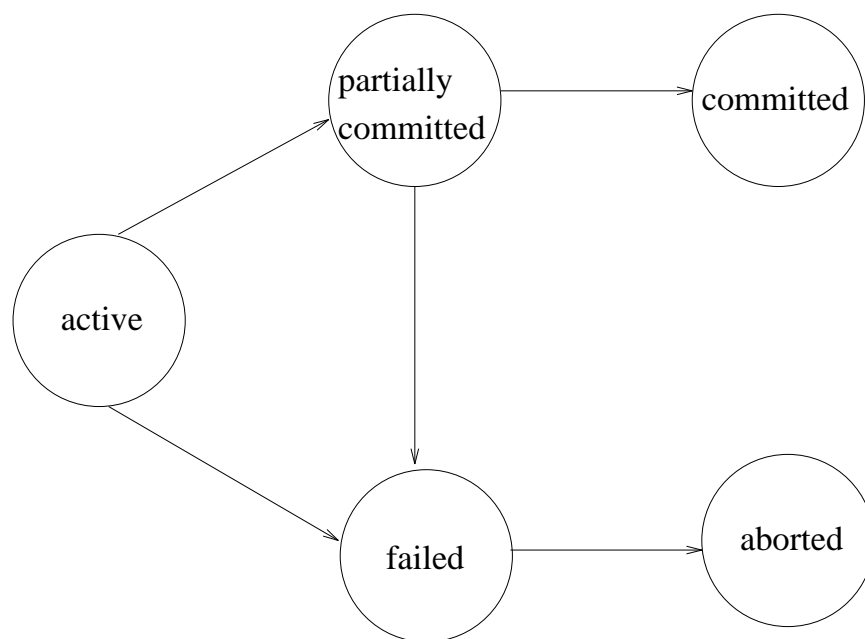
se ne implementano approssimazioni, duplicando le informazioni in diverse memorie non volatili con probabilità di fallimento indipendenti.

## MODELLO ASTRATTO PER L'ESECUZIONE DI TRANSAZIONI

una transazione è sempre in uno dei seguenti stati:

- **active**: lo stato iniziale
- **partially committed**: lo stato raggiunto dopo che è stata eseguita l'ultima istruzione
- **failed**: lo stato raggiunto dopo aver determinato che l'esecuzione non può procedere normalmente
- **aborted**: lo stato raggiunto dopo che la transazione ha subito un rollback e la base di dati è stata ripristinata allo stato precedente l'inizio della transazione
- **committed**: dopo il completamento con successo.

## MODELLO ASTRATTO PER L'ESECUZIONE DI TRANSAZIONI





## MODELLO ASTRATTO PER L'ESECUZIONE DI TRANSAZIONI

- è importante che una transazione non effettui *scritture esterne osservabili* (cioè scritture che non possono essere “cancelate”, ad es. su terminale o stampante) prima di entrare nello stato di commit
- dopo il rollback di una transazione, il sistema ha due possibilità:
  - **rieseguire la transazione** ha senso solo se la transazione è stata abortita a seguito di errori software o hardware non dipendenti dalla logica interna della transazione
  - **eliminare la transazione** se si verificano degli errori interni che possono essere corretti solo riscrivendo il programma applicativo.

## ESEMPIO

$T$  transazione che trasferisce Lit. 100000 dal conto A al conto B

$A = \text{Lit. } 1000000$ ,  $B = \text{Lit. } 15000000$

dopo la modifica di A e prima della modifica di B, si verifica una caduta di sistema e i contenuti della memoria vengono persi

- si riesegue  $T \Rightarrow$  stato (inconsistente) in cui  $A = \text{Lit. } 800000$  e  $B = \text{Lit. } 15.100000$
- non si riesegue  $T \Rightarrow$  stato corrente (inconsistente) in cui  $A = \text{Lit. } 900000$  e  $B = \text{Lit. } 15000000$

in entrambi i casi lo stato risultante è inconsistente, il problema è causato dal fatto di avere modificato la base di dati prima di avere la certezza che la transazione abbia terminato con successo.

## MECCANISMI DI RECOVERY CON LOG

- durante l'esecuzione di una transazione tutte le operazioni di scrittura sono registrate in un particolare file gestito dal sistema, detto file di *log*
- concettualmente, il log può essere pensato come un file sequenziale, nell'implementazione effettiva possono essere usati più file fisici
- ad ogni record inserito nel log viene attribuito un identificatore unico (LSN, log sequence number o numero di sequenza di log) che in genere è l'indirizzo logico del record

## MECCANISMI DI RECOVERY CON LOG

- la versione non volatile del log è memorizzata su memoria stabile

i record di log sono memorizzati in un primo momento solo nei buffer dei file di log nella memoria volatile

prima di cominciare il commit, tutti i record di log fino ad un certo punto, identificato da un LSN, vengono scritti su memoria stabile (il log viene *forzato* fino a quel LSN)

- tramite il log, il sistema può gestire qualsiasi malfunzionamento che non implichi la perdita di informazioni contenute in memoria non volatile.

## LOG INCREMENTALE CON MODIFICHE DIFFERITE

- tutte le operazioni di scrittura eseguite da una transazione vengono differite fino a che la transazione non entra nello stato *partially committed*, tutte le modifiche sono registrate nel file di log
- durante l'esecuzione di una transazione  $T_i$ :
  - prima che  $T_i$  cominci la propria esecuzione viene scritto nel log il record  $\langle T_i \text{ start} \rangle$
  - ogni operazione di scrittura  $write[x]$  produce un record di log della forma:  
 $\langle T_i, x, \text{nuovo valore} \rangle$
  - quando  $T_i$  entra in stato *partially committed*, viene scritto nel log il record  $\langle T_i \text{ commit} \rangle$ .

## LOG INCREMENTALE CON MODIFICHE DIFFERITE

- tutti i record di log sono scritti su memoria stabile, a questo punto, la transazione entra nello stato *committed*
- lo schema di recovery usa la procedura di recovery redo( $T_i$ ), che assegna a tutti i dati aggiornati dalla transazione  $T_i$  i nuovi valori

l'operazione redo deve essere *idempotente*, cioè, più esecuzioni in sequenza di tale operazione devono essere equivalenti ad un'esecuzione singola (questo assicura un comportamento corretto anche in presenza di malfunzionamenti durante il recovery)

- a seguito di un malfunzionamento, una transazione  $T_i$  viene rieseguita se il log contiene entrambi i record  $\langle T_i \text{ start} \rangle$  e  $\langle T_i \text{ commit} \rangle$ .

## ESEMPIO

$T_1$	$T_2$
Read $l_r$	Read $c/c_2$
$l_r = l_r - 150000$	$c/c_2 = c/c_2 - 200000$
Write $l_r$	Write $c/c_2$
Read $c/c_1$	Commit $T_2$
$c/c_1 = c/c_1 + 150000$	
Write $c/c_1$	
Commit $T_1$	

viene eseguita prima  $T_1$  e poi  $T_2$

inizialmente  $l_r = 500000$        $c/c_1 = 600000$   
 $c/c_2 = 800000$ .

contenuto del log al termine dell'esecuzione delle due transazioni:

<  $T_1$ , start >  
<  $T_1$ , l\_r,350000 >  
<  $T_1$ , c/c1, 750000 >  
<  $T_1$ , commit >  
<  $T_2$ , start >  
<  $T_2$ , c/c2, 600000 >  
<  $T_2$ , commit >

se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c1", lo stato del log al momento del crash è il seguente:

<  $T_1$ , start >  
<  $T_1$ , l\_r,350000 >  
<  $T_1$ , c/c1, 750000 >

⇒ non si esegue alcuna operazione di redo



se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c2", lo stato del log al momento del crash è il seguente:

$\langle T_1, \text{start} \rangle$   
 $\langle T_1, l_r, 350000 \rangle$   
 $\langle T_1, c/c1, 750000 \rangle$   
 $\langle T_1, \text{commit} \rangle$   
 $\langle T_2, \text{start} \rangle$   
 $\langle T_2, c/c2, 600000 \rangle$

$\Rightarrow$  viene effettuato redo( $T_1$ ) e lo stato della base di dati diventa:

$l_r = 350000$        $c/c1 = 750000$        $c/c2 = 800000$

se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Commit  $T_2$ ", lo stato del log al momento del crash è il seguente:

<  $T_1$ , start >  
<  $T_1$ , l\_r, 350000 >  
<  $T_1$ , c/c1, 750000 >  
<  $T_1$ , commit >  
<  $T_2$ , start >  
<  $T_2$ , c/c2, 600000 >  
<  $T_2$ , commit >

⇒ vengono effettuate le operazioni redo( $T_1$ ) e redo( $T_2$ ) e lo stato della base di dati diventa: l\_r = 350000    c/c1 = 750000    c/c2 = 600000.

## LOG INCREMENTALE CON MODIFICHE IMMEDIATE

- gli aggiornamenti sono effettuati sulla base di dati e un *log incrementale* tiene traccia di tutti i cambiamenti; tali informazioni sono usate per riportare la base di dati allo stato consistente precedente in caso di malfunzionamento
- – prima che  $T_i$  cominci la propria esecuzione viene scritto nel log il record  $\langle T_i \text{ start} \rangle$
- ogni operazione di scrittura  $write[x]$  è preceduta dalla scrittura un record di log della forma:  
 $\langle T_i, x, \text{vecchio valore}, \text{nuovo valore} \rangle$
- quando  $T_i$  entra in stato *partially committed*, viene scritto nel log il record  $\langle T_i \text{ commit} \rangle$ .

## LOG INCREMENTALE CON MODIFICHE IMMEDIATE

- non si può aggiornare effettivamente la base di dati prima che il corrispondente record di log sia scritto in memoria stabile
- due procedure di ripristino:
  - $\text{undo}(T_i)$ : ripristina i valori di tutti i dati aggiornati da  $T_i$  ai vecchi valori
  - $\text{redo}(T_i)$ : pone il valore di tutti i dati aggiornati da  $T_i$  ai nuovi valori.

entrambe devono essere idempotenti

- a seguito di un malfunzionamento:
  - $T_i$  è disfatta se il log contiene  $\langle T_i \text{ start} \rangle$ , ma non  $\langle T_i \text{ commit} \rangle$
  - $T_i$  è rieseguita se il log contiene sia  $\langle T_i \text{ start} \rangle$  che  $\langle T_i \text{ commit} \rangle$ .

## ESEMPIO

contenuto del log al termine dell'esecuzione delle due transazioni:

```
< T1, start >  
< T1, l_r,350000 >  
< T1, c/c1, 750000 >  
< T1, commit >  
< T2, start >  
< T2, c/c2, 600000 >  
< T2, commit >
```

se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c1", lo stato del log al momento del crash è il seguente:

```
< T1, start >  
< T1, l_r,500000,350000 >  
< T1, c/c1, 600000,750000 >
```

⇒ si esegue l'azione di  $\text{undo}(T_1)$ , il nuovo stato dei dati è:  $l_r = 500000$       $c/c1 = 600000$       $c/c2 = 800000$ .

se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write c/c2", lo stato del log al momento del crash è il seguente:

<  $T_1$ , start >  
<  $T_1$ , l\_r, 500000, 350000 >  
<  $T_1$ , c/c1, 600000, 750000 >  
<  $T_1$ , commit >  
<  $T_2$ , start >  
<  $T_2$ , c/c2, 800000, 600000 >

⇒ viene effettuato redo( $T_1$ ) e undo( $T_2$ ), lo stato della base di dati diventa:

l\_r = 350000      c/c1 = 750000      c/c2 =  
800000

se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Commit  $T_2$ ", lo stato del log al momento del crash è il seguente:

<  $T_1$ , start >  
<  $T_1$ , l\_r, 500000, 350000 >  
<  $T_1$ , c/c1, 600000, 750000 >  
<  $T_1$ , commit >  
<  $T_2$ , start >  
<  $T_2$ , c/c2, 800000, 600000 >  
<  $T_2$ , commit >

⇒ vengono effettuate le operazioni redo( $T_1$ ) e redo( $T_2$ ) e lo stato della base di dati diventa: l\_r = 350000    c/c1 = 750000    c/c2 = 600000.

# CHECKPOINT

- meccanismo introdotto per ridurre i tempi di recovery
- il sistema periodicamente:
  - forza tutti i record di log residenti in memoria principale su memoria stabile
  - forza tutti i blocchi di buffer su disco
  - forza il record <checkpoint> su memoria stabile.
- dopo che si è verificato un malfunzionamento si esamina il log per determinare l'ultima transazione  $T_i$  che ha iniziato la propria esecuzione prima che avesse luogo l'ultimo checkpoint  
si considerano nella procedura di recovery tutte le transazioni iniziate dopo  $T_i$ .



## LIVELLI DI ISOLAMENTO

**grado 0** *chaos*: non consente che più transazioni aggiornino simultaneamente un dato

**grado 1** *browse*: non consente la sovrascrittura di dati aggiornati da transazioni che non hanno effettuato il commit

**grado 2** *cursor stability*: 1 + impone alle transazioni di leggere solo i dati che sono stati aggiornati da transazioni che hanno effettuato il commit

**grado 3** *repeatable reads*: una transazione che vuole leggere/scrivere un dato deve attendere il completamento di tutte le altre transazioni che hanno precedentemente letto/scritto quel dato

fornisce un isolamento completo.