

# CONTROLLO DELLA CONCORRENZA

# ESECUZIONE CONCORRENTE DI UN INSIEME DI TRANSAZIONI

- Finora abbiamo considerato transazioni singole
- In generale però nello stesso momento più transazioni che accedono una stessa base di dati potrebbero essere in esecuzione
- Come è possibile eseguire concorrentemente un insieme di transazioni?
- Quali problemi possono sorgere e come si possono risolvere?

# ESECUZIONE SERIALE DI TRANSAZIONI

- Un DBMS, dovendo supportare l'esecuzione di diverse transazioni che accedono a dati condivisi, potrebbe eseguire tali transazioni in sequenza (“**serial execution**”)
- Ad esempio, due transazioni T1 e T2 potrebbero essere eseguite in questo modo, in cui si evidenzia la successione temporale (“**schedule**”) delle operazioni elementari sul DB:

T1	T2
R(X)	
W(X)	
Commit	
	R(Y)
	W(Y)
	Commit

# ESECUZIONE CONCORRENTE DI TRANSAZIONI

- In alternativa, il DBMS può eseguire più transazioni in concorrenza, alternando l'esecuzione di operazioni di una transazione con quella di operazioni di altre transazioni (“**interleaved execution**”)
- Eseguire più transazioni concorrentemente è necessario per garantire **buone prestazioni**:
  - Si sfrutta il fatto che, mentre una transazione è in attesa del completamento di una operazione di I/O, un'altra può utilizzare la CPU, il che porta ad aumentare il “**throughput**” (**n. transazioni elaborate nell'unità di tempo**) del sistema
  - Se si ha una transazione “breve” e una “lunga”, l'esecuzione concorrente porta a ridurre il tempo medio di risposta del sistema

T1	T2
R(X)	
	R(Y)
	W(Y)
	Commit
W(X)	
Commit	

# RIDUZIONE DEL TEMPO DI RISPOSTA

- T1 è “lunga”, T2 è “breve”; per semplicità ogni riga della tabella è un’unità di tempo

time	T1	T2
1	R(X1)	
2	W(X1)	
...		
999	R(X500)	
1000	W(X500)	
1001	Commit	
1002		R(Y)
1003		W(Y)
1004		Commit

$$\text{Tempo medio di risposta} = (1001 + (1004-1))/2 = 1002$$

T2 richiede a time = 2  
di iniziare

time	T1	T2
1	R(X1)	
2		R(Y)
3		W(Y)
4		Commit
5	W(X1)	
...		
1002	R(X500)	
1003	W(X500)	
1004	Commit	

$$\text{Tempo medio di risposta} = (1004 + 3)/2 = 503.5$$

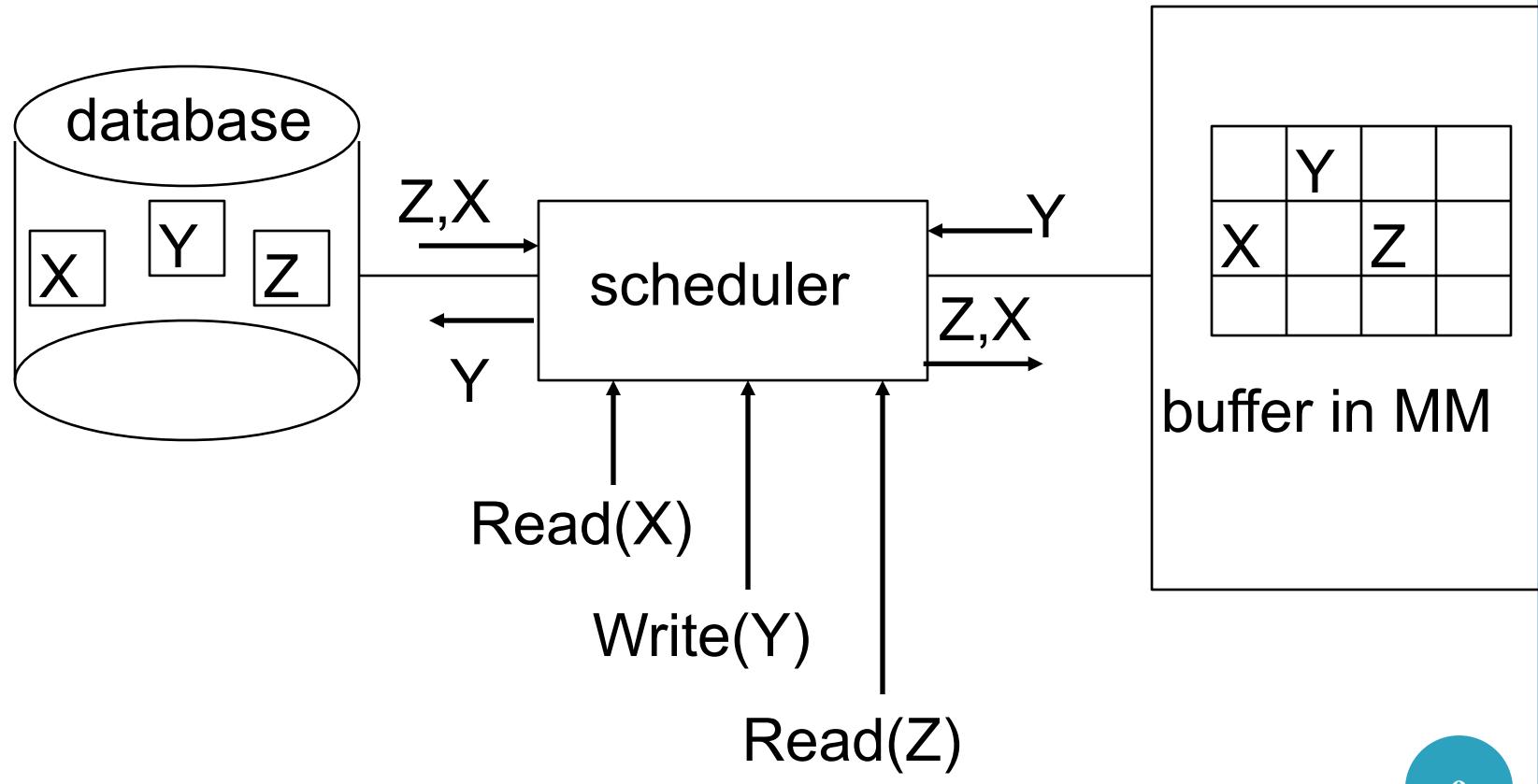
# CONTROLLO DELLA CONCORRENZA

- Come abbiamo visto, l'esecuzione concorrente **migliora il tempo medio di risposta del sistema**
- L'esecuzione concorrente potrebbe però generare anomalie
- Scopo del gestore della concorrenza è garantire consistenza e isolamento delle transazioni eseguite concurrentemente, evitando possibili anomalie

# ARCHITETTURA DI RIFERIMENTO

- Transazioni come sequenze di operazioni di input/output (Read/Write) su oggetti astratti x, y, z
  - Valori di attributi
  - Tuple
  - Tabelle
  - ...
- ogni operazione di input/output legge blocchi di memoria secondaria in pagine di buffer o scrive pagine di buffer in blocchi di memoria secondaria

# ARCHITETTURA DI RIFERIMENTO



# ANOMALIE

- Il Transaction Manager deve garantire che transazioni che eseguono in concorrenza non interferiscano tra loro. Se ciò non avviene, si possono avere 4 tipi base di problemi, esemplificati dai seguenti scenari:

**Lost Update:** due persone, in due agenzie diverse, comprano entrambe l'ultimo biglietto per il concerto degli U2 a Roma (!?)

**Dirty Read:** nel programma dei concerti degli U2 figura una tappa a Bologna il 15/07/02, ma quando provate a comprare un biglietto per quella data vi viene detto che in realtà non è ancora stata fissata (!?)

**Unrepeatable Read:** per il concerto degli U2 (finalmente la data è stata fissata!) vedete che il prezzo è di 40 €, ci pensate su 5 minuti, ma il prezzo nel frattempo è salito a 50 € (!?)

**Phantom Row:** volete comprare i biglietti di tutte e due le tappe degli U2 in Italia, ma quando comprate i biglietti scoprirete che le tappe sono diventate 3 (!?)

**Lost Update**: due persone, in due agenzie diverse, comprano entrambe l'ultimo biglietto per il concerto degli U2 a Roma (!?)

## ANOMALIE - LOST UPDATE

- Il seguente schedule mostra un caso tipico di lost update, in cui per comodità si evidenziano anche le operazioni che modificano il valore del dato X e si mostra come varia il valore di X nel DB

T1	X	T2
R(X)	1	
X=X-1	1	
	1	R(X)
	1	X=X-1
W(X)	0	
Commit	0	
	0	W(X)
	0	Commit

Questo update viene perso!

R(X): operazione SQL di lettura di un valore dalla base di dati e sua memorizzazione in una variabile locale X

Esempio: lettura del numero di biglietti per il concerto degli U2 di Bologna

W(X): operazione SQL di modifica della base di dati, partendo dal valore di X

Esempio: scrivo sul DB il nuovo numero di biglietti disponibili per il concerto degli U2 di Bologna

- Il problema nasce perché T2 legge il valore di X prima che T1 (che lo ha già letto) lo modifichi ("entrambe vedono l'ultimo biglietto")

**Dirty Read:** nel programma dei concerti degli U2 figura una tappa a Bologna il 15/07/02, ma quando provate a comprare un biglietto per quella data vi viene detto che in realtà non è ancora stata fissata (!?)

## ANOMALIE - DIRTY READ

- In questo caso il problema è che una transazione legge un dato “che non c’è”:

Esempio: X corrisponde al calendario dei concerti degli U2, modificato da T1 aggiungendo la nuova tappa

T1	X	T2
R(X)	0	
X=X+1	0	
W(X)	1	
	1	R(X)
Rollback	0	
	0	...
	0	Commit

Questa lettura è “sporca”!

Esempio: T2 legge il calendario e trova la nuova data, se poi prova comprare il biglietto più avanti nel codice, la nuova data non compare più

- Quanto svolto da T2 si basa su un valore di X “intermedio”, e quindi non stabile (“la data definitiva non è il 15/07/02”)
- Le conseguenze sono impredicibili (dipende cosa fa T2) e si presenterebbero anche se T1 non abortisse

**Unrepeatable Read:** per il concerto degli U2 (finalmente la data è stata fissata!) vedete che il prezzo è di 40 €, ci pensate su 5 minuti, ma il prezzo nel frattempo è salito a 50 € (!?)

## ANOMALIE - UNREPEATABLE READ

- Ora il problema è che una transazione legge due volte un dato e trova valori diversi (“il prezzo nel frattempo è aumentato”):

T1	X	T2
R(X)	0	
	0	R(X)
	0	$X=X+1$
	1	W(X)
	1	Commit
R(X)	1	
Commit	1	

Le 2 letture sono tra loro inconsistenti!

Esempio: X corrisponde al prezzo del concerto degli U2

- Anche in questo caso si possono avere gravi conseguenze

**Phantom Row:** volete comprare i biglietti di tutte e due le tappe degli U2 in Italia, ma quando comprate i biglietti scoprite che le tappe sono diventate 3 (!?)

## ANOMALIE - PHANTOM ROW

- Questo caso si può presentare quando vengono inserite o cancellate tuple che un'altra transazione dovrebbe logicamente considerare
  - Nell'esempio la tupla  $t_3$  è un “phantom”, in quanto T1 “non la vede”

Data	Città	Stato	NBigliettiDisponibili
12/06/19	Bologna	Italia	50
18/07/19	Firenze	Italia	32
5/08/19	Milano	Italia	250

T1 «non vede» questa tupla

T1  
 UPDATE Tappe  
 SET NBigliettiDisponibili = NBigliettiDisponibili -1  
 WHERE Stato = Italia

T2  
 INSERT INTO Tappe  
 VALUES ('5/08/19','Milano','Italia',250)

T1	T2
R(t1)	
R(t2)	
...	
W(t1)	
W(t2)	
	Insert(t3)
...	
Commit	
	Commit

# ANOMALIE E SERIALIZZABILITÀ

- L'interleaving tra le transazioni T1 e T2 negli esempi presentati in precedenza produce uno stato della base di dati scorretto
- Si sarebbe ottenuto uno stato corretto se negli esempi precedenti ciascuna transazione fosse stata eseguita da sola o se le due transazioni fossero state eseguite l'una dopo l'altra, consecutivamente
- **Schedule seriale:** schedule in cui le transazioni vengono eseguite in sequenza
- Con uno schedule seriale **l'isolamento è totale**
- Sappiamo però che uno schedule seriale potrebbe portare a tempi medi di risposta del sistema più alti

T1	T2
R(X)	
W(X)	
Commit	
	R(Y)
	W(Y)
	Commit

# ANOMALIE E SERIALIZZABILITÀ

- Esistono schedule non seriali che comunque non causano anomalie
- **Schedule serializzabile:** produce lo «stesso» risultato (= stato della base di dati, altri output osservabili) di uno schedule seriale delle stesse transazioni, quindi evita le anomalie garantisce isolamento totale
  - richiede una nozione di equivalenza tra schedule (non la vediamo)

# PROTOCOLLI DI CONTROLLO DELLA CONCORRENZA

- Per assicurare la serializzabilità durante l'esecuzione di più transazioni, il DBMS utilizza un **protocollo di controllo della concorrenza**, eseguito dal gestore delle transazioni, che porta **all'esecuzione di uno schedule serializzabile** (che quindi non genera anomalie)
- **Accento su isolamento**
- Esistono molteplici protocolli, tra i più noti
  - Protocolli basati su lock
  - Protocolli basati su timestamp
- Vediamo solo il primo tipo, più usato nei DBMS commerciali

# PROTOCOLLI DI LOCKING

- Una comune tecnica usata dai DBMS per evitare i problemi visti consiste nell'uso di **lock**
  - I lock (“blocchi”) sono un **meccanismo comunemente usato** dai sistemi **operativi** per disciplinare l’accesso a risorse condivise
  - Per eseguire un’operazione è prima necessario “acquisire” un lock sulla **risorsa interessata** (ad es. una tupla)
  - La richiesta di lock è **implicita**, e quindi **non è visibile a livello SQL**
    - ...ma dopo vedremo che anche in SQL si può fare qualcosa
- I lock sono di vario tipo (DB2 ne ha ben 12!); quelli di base sono:
  - **S** (**Shared**): un lock condiviso è necessario per **leggere**
  - **X** (**eXclusive**): un lock esclusivo è necessario per **scrivere/modificare**

# PROTOCOLLI DI LOCKING – COMPATIBILITÀ TRA OPERAZIONI

- Due operazioni sono in **conflitto** se
  - sono sullo stesso dato e
  - almeno una delle due è un'operazione di scrittura
- Due operazioni sono **compatibili** se:
  - si riferiscono a dati diversi oppure
  - si riferiscono allo stesso dato ma entrambe sono operazioni di lettura

**Idea base:** ritardare l'esecuzione di operazioni in conflitto

# PROTOCOLLI DI LOCKING – COMPATIBILITÀ TRA LOCK

- Il **Lock Manager** è un modulo del DBMS che si occupa di tener traccia di quali sono le risorse correntemente in uso e di quali transazioni le stanno usando (e in che modo)
- Quando una transazione T vuole operare su un dato Y, viene inviata la richiesta di acquisizione del lock corrispondente al Lock Manager
- Il lock viene accordato a T in funzione della seguente tabella di compatibilità

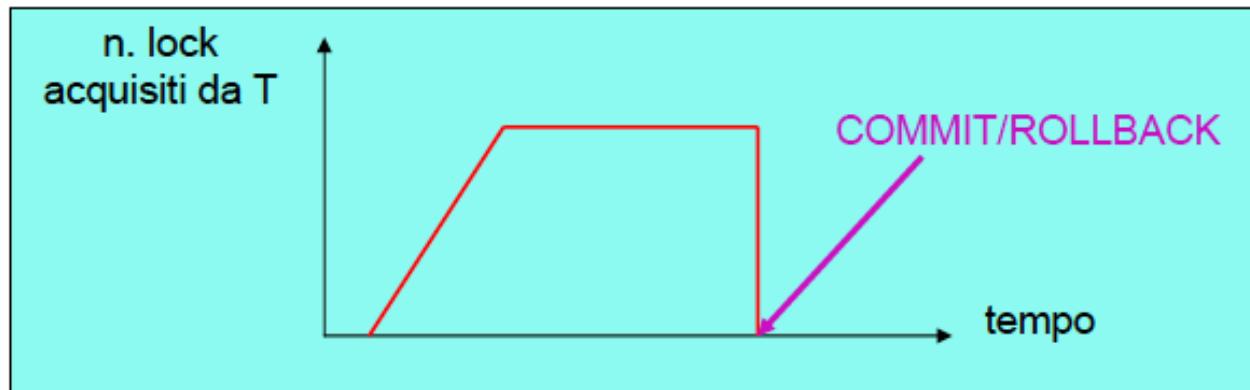
Su Y un'altra transazione ha un lock di tipo

		S	X
T richiede un lock di tipo	S	OK	NO
	X	NO	NO

- Quando T ha terminato di usare Y, può rilasciare il lock (**unlock(Y)**)

# PROTOCOLLI DI LOCKING – STRONG 2 PHASE LOCKING (STRONG 2 PL)

- Il modo con cui le transazioni rilasciano i lock acquisiti è la chiave per risolvere i problemi di concorrenza
- Si può dimostrare che se
  - Una transazione prima acquisisce tutti i lock necessari
  - Rilascia i lock solo al termine dell'esecuzione (**COMMIT** o **ROLLBACK**)allora l'Isolation è garantita



- Come effetto collaterale si possono verificare **deadlock**, ossia situazioni di stallo, che vengono risolte facendo abortire una transazione

# ASSENZA DI LOST UPDATE

- Lo schedule visto precedentemente si modifica come segue:

T1	X	T2
R(X)	1	
X=X-1	1	
	1 R(X)	
	1 X=X-1	
W(X)	0	
Commit	0	
	0 W(X)	
	0 Commit	

T1	X	T2
S-lock(X)	1	
R(X)	1	
X=X-1	1	
	1 S-lock(X)	
	1 R(X)	
	1 X=X-1	
X-lock(X)	1	
wait	1 X-lock(X)	
wait	1 wait	

- Né T1 né T2 riescono ad acquisire il lock per poter modificare X (restano in attesa, “**wait**”); si verifica quindi un deadlock. Se il sistema decide di abortire, ad esempio, T2, allora T1 può proseguire

# ASSENZA DI DIRTY READ

T1	X	T2
R(X)	0	
X=X+1	0	
W(X)	1	
	1	R(X)
Rollback	0	
	0	...
	0	Commit

T1	X	T2
S-lock(X)	0	
R(X)	0	
X=X+1	0	
X-lock(X)	0	
W(X)	1	
	1	S-lock(X)
	0	wait
Rollback	0	wait
Unlock(X)	0	wait
	0	R(X)

- In questo caso l'esecuzione corretta richiede che T2 aspetti la terminazione di T1 prima di poter leggere il valore di X

# ASSENZA DI UNREPEATABLE READ

T1	X	T2
R(X)	0	
	0	R(X)
	1	X=X+1
	1	W(X)
	1	Commit
R(X)	1	
Commit	1	

T1	X	T2
S-lock(X)	0	
R(X)	0	
	0	S-lock(X)
	0	R(X)
	0	X=X+1
	0	X-lock(X)
	0	wait
R(X)	0	wait
Commit	0	wait
Unlock(X)	0	wait
	1	W(X)

- Anche in questo caso T2 viene messa in attesa, e T1 ha quindi la garanzia di poter leggere sempre lo stesso valore di X

# ASSENZA DI PHANTOM ROW

- Questo è, tra quelli visti, il problema più difficile da risolvere. Le soluzioni adattabili sono varie, e differiscono in complessità e livello di concorrenza che permettono:
  - ➡ ■ Si può acquisire un **S-lock** su tutta la **table**, e poi richiedere gli **X-lock** per le **tuple** che si vogliono modificare
  - Si introduce un nuovo tipo di lock, detto “**predicate lock**”, che riguarda tutte le **tuple** che soddisfano un **predicato** ( Stato = ‘Italia’ nell’esempio)
  - Se esiste un indice su **Stato**, si deve porre un **lock sulla foglia** che contiene ‘Italia’
- Nei DBMS la situazione è in realtà più complessa, sia per i tipi di lock presenti, sia per la “**granularità**” con cui i lock possono essere richiesti e acquisiti (a livello di attributo, tupla, pagina, relazione, ...)

# CONTROLLO DELLA CONCORRENZA IN SQL- LIVELLI DI ISOLAMENTO

- Come abbiamo detto, il protocollo Strong 2-Phase Locking garantisce schedule serializzabili e quindi transazioni completamente isolate
- Il prezzo da pagare è una maggiore rigidità del sistema
- Le anomalie di **lost update**, **unrepeatable read** e **dirty read** non si possono presentare
- Tuttavia, **a volte può essere sufficiente operare in modo solo «alcune» anomalie si possono presentare**
- Scegliere di operare a un certo livello di isolamento in cui si possono presentare delle anomalie ha il vantaggio di aumentare il grado di concorrenza raggiungibile, e quindi di migliorare le prestazioni
- In SQL, La scelta può essere effettuata **individualmente per ciascuna transazione**

# CONTROLLO DELLA CONCORRENZA IN SQL- LIVELLI DI ISOLAMENTO

- In SQL, il livello di isolamento può essere scelto, usando opportuni comandi, nel codice della transazione
- Lo standard definisce 4 livelli di isolamento
- Nessun livello può generare anomalie di lost update

Livello di isolamento	Lost update	Dirty read	Unrepeatable read	Phantom problem
<b>READ UNCOMMITTED</b>	NO	YES	YES	YES
<b>READ COMMITTED</b>	NO	NO	YES	YES
<b>REPEATABLE READ</b>	NO	NO	NO	YES
<b>SERIALIZABLE</b>	NO	NO	NO	NO

# LIVELLI DI ISOLAMENTO - SERIALIZZABILE

- Il più alto livello di isolamento è **SERIALIZZABILE**
- Assicura serializzabilità
- Questo livello garantisce che
  - T legga solo modifiche effettuate da transazioni che hanno effettuato il commit
  - nessun valore letto o scritto da T sia modificato prima che T abbia terminato la sua esecuzione
  - se T legge un insieme di valori basandosi su una qualche condizione di ricerca (SELECT), questo insieme non venga modificato da altre transazioni fino a che T non ha terminato
- Protocollo di locking: **Strong 2PL + acquisizione lock su tavelle e indici (li vedremo nel seguito)** per evitare Phantom rows
- Nessuna anomalia si può verificare

# ESEMPIO ANNOTAZIONE SERIALIZZABILE

T1
R(t1)
R(t2)
...
R(t1)
$t1.A = t1.A + 1$
W(t1)
R(t2)
Commit

Table V			
	A	B	
t1	3	5	...
t2	4	8	...

```
BEGIN TRANSACTION  
  
SELECT *  
FROM V  
WHERE A > 3;  
  
UPDATE V  
SET A = A + 1  
WHERE A = 3  
  
COMMIT;
```

T1
S-lock(V)
R(t1)
R(t2)
...
X-lock(V)
R(t1)
$t1.A = t1.A + 1$
W(t1)
R(t2)
...
Unlock(V)
Commit

# LIVELLI DI ISOLAMENTO – REPEATABLE READ

- Il livello **REPEATABLE READ** garantisce che
  - T legga solo modifiche effettuate da transazioni che hanno effettuato il commit
  - nessun valore letto o scritto da T sia modificato da altre transazioni prima che T abbia terminato
- Protocollo di locking: **Strong 2PL**, ma non vengono chiesti lock su tabelle (e su indici)
- Anomalia di phantom row possibile

# ESEMPIO ANNOTAZIONE REPEATABLE READ

T1
R(t1)
R(t2)
...
R(t1)
$t1.A = t1.A+1$
W(t1)
R(t2)
Commit

Table V

	A	B	...
t1	3	5	...
t2	4	8	...

```

BEGIN TRANSACTION
SELECT *
FROM V
WHERE A > 3;

UPDATE V
SET A = A + 1
WHERE A = 3

COMMIT;
  
```

T1
S-lock(t1)
R(t1)
S-lock(t2)
R(t2)
...
X-lock(t1)
R(t1)
$t1.A = t1.A+1$
W(t1)
X-lock(t2)
R(t2)
...
Unlock(t1)
Unlock(t2)
Commit

# LIVELLI DI ISOLAMENTO – READ COMMITTED

- Il livello **READ COMMITTED** assicura che
  - T legga solo modifiche effettuate da transazioni che hanno effettuato il commit
  - nessun valore scritto da T sia modificato da altre transazioni prima che T abbia terminato
  - un valore letto da T può però essere modificato da altre transazioni prima che T abbia terminato
- Protocollo di locking:
  - I lock **esclusivi** vengono mantenuti fino al **termine della transazione**
  - I lock **condivisi** vengono acquisiti e rilasciati **appena possibile**
- Anomalie di letture non ripetibili e phantom row possibili

# ESEMPIO ANNOTAZIONE READ COMMITTED

T1
R(t1)
R(t2)
...
R(t1)
$t1.A = t1.A+1$
W(t1)
R(t2)
Commit

Table V

	A	B	...
t1	3	5	...
t2	4	8	...

```
BEGIN TRANSACTION
SELECT *
FROM V
WHERE A > 3;

UPDATE V
SET A = A + 1
WHERE A = 3

COMMIT;
```

T1
S-lock(t1)
R(t1)
Unlock(t1)
S-lock(t2)
R(t2)
Unlock(t2)
...
X-lock(t1)
R(t1)
$t1.A = t1.A+1$
W(t1)
X-lock(t2)
R(t2)
...
Unlock(t1)
Unlock(t2)
Commit

# LIVELLI DI ISOLAMENTO – READ UNCOMMITTED

- Il livello **READ UNCOMMITTED** permette ad una transazione T di
  - vedere modifiche effettuate da transazioni concorrenti, anche se non hanno ancora effettuato il commit
  - nessun valore scritto da T può essere modificato da altre transazioni prima che T abbia terminato
  - non vengono acquisiti lock condivisi prima di effettuare le letture
- Protocollo di locking:
  - I lock **esclusivi** vengono mantenuti fino al **termine della transazione**
  - I lock **condivisi** non vengono richiesti
- Nessun lock su letture
- Tutte le anomalie di lettura sono possibili (ma non si perdonano aggiornamenti – no anomalia di lost update)

# ESEMPIO ANNOTAZIONE READ UNCOMMITTED

Table V

T1
R(t1)
R(t2)
...
R(t1)
$t1.A = t1.A + 1$
W(t1)
R(t2)
Commit

	A	B	...
t1	3	5	...
t2	4	8	...

```
BEGIN TRANSACTION
SELECT *
FROM V
WHERE A > 3;

UPDATE V
SET A = A + 1
WHERE A = 3

COMMIT;
```

T1
R(t1)
R(t2)
...
X-lock(t1)
R(t1)
$t1.A = t1.A + 1$
W(t1)
X-lock(t2)
R(t2)
...
Unlock(t1)
Unlock(t2)
Commit

# LIVELLI DI ISOLAMENTO IN SQL

- Il livello di isolamento può essere specificato contestualmente al comando di inizio transazione

```
BEGIN [ TRANSACTION ]  
[ ISOLATION LEVEL { SERIALIZABLE | REPEATABLE  
READ | READ COMMITED | READ UNCOMMITTED } ] )
```

- Può anche essere impostato con il comando SET TRANSACTION

```
SET TRANSACTION ISOLATION LEVEL  
{ SERIALIZABLE | REPEATABLE READ | READ  
COMMITED | READ UNCOMMITTED }
```

# LIVELLI DI ISOLAMENTO: QUALE SCEGLIERE?

- Il livello **SERIALIZABLE** è il più sicuro ed è quello preferibile per la maggioranza delle transazioni
- Alcune transazioni possono però essere eseguite a livelli di isolamento più bassi in modo da migliorare le prestazioni del sistema
- Spesso **READ COMMITTED** è il livello adeguato

## ESEMPIO

- Query che non hanno bisogno di risposte esatte (es, query statistiche)
  - **Quanti conti hanno saldi maggiore di 1000?**
- La serializzabilità non è necessaria (“basta” read committed)
- Eventuali anomalie di lettura irripetibile non alterano di molto il risultato
- READ COMMITTED livello adeguato

# ESEMPIO

- Transazioni con interazione con l’utente
- Sistema di prenotazione voli aerie

BEGIN TRANSACTION

- Trova la lista dei posti disponibili
- Parla con il cliente illustrandogli le disponibilità
- Prenota il posto scelto dal cliente

COMMIT;

- Il prezzo della serializzabilità (livello SERIALIZABLE o REPEATABLE READS) è troppo alto

- Una transazione che mantiene lock durante l’interazione con l’utente genera inutili i colli di bottiglia

- Scegliendo READ COMMITTED:
  - Si limita overhead, rinunciando a serializzabilità
  - Side effect: quando si è scelto un posto e lo si prova a prenotare, il posto è diventato indisponibile (anomalia lettura irripetibili)

# ESEMPIO

- accounts( number, branchnum, balance)
- Q: somma
  - SELECT sum(balance) AS sum FROM accounts;
- N transazioni  $T_i$ : scambiano i saldi tra due conti X e Y
  - SELECT balance INTO valX FROM accounts WHERE number=X;  
SELECT balance INTO valY FROM accounts WHERE number=Y;
  - UPDATE accounts SET balance=valX WHERE number=Y;  
UPDATE accounts SET balance=valY WHERE number=X;
- Consideriamo due tuple t1 e t2, tali che t1.number =X e t2.number=Y
- Vogliamo capire quale livello di isolamento potrebbe essere preferibile

SUM = 200

## READ COMMITTED (un possible schedule)

Q	t1.balance	t2.balance	SUM	Ti
<b>S-lock(t1)</b>				
R(t1)	<b>100</b>	200		
<b>Unlock(t1)</b>	100	200		
	100	200		<b>S-lock(t1)</b>
	100	200		<b>S-lock(t2)</b>
	<b>100</b>	200		<b>R(t1)</b>
	100	<b>200</b>		<b>R(t2)</b>
	100	200		<b>valX = t1.balance</b>
	100	200		<b>valY = t2.balance</b>
	100	200		<b>Unlock(t1)</b>
	100	200		<b>Unlock(t2)</b>
	100	200		<b>X-lock(t1)</b>
	100	200		<b>X-lock(t2)</b>
	100	200		<b>R(t1)</b>
	100	200		<b>t1.balance=valY</b>
	<b>200</b>	200		<b>W(t1)</b>
<b>S-lock(t2)</b>	200	200		
wait	200	200		<b>R(t2)</b>
wait	200	200		<b>t2.balance=valX</b>
wait	200	<b>100</b>		<b>W(t2)</b>
wait	200	100		<b>Unlock(t1)</b>
wait	200	100		<b>Unlock(t2)</b>
wait	200	100		<b>Commit</b>
R(t2)	200	<b>100</b>		
sum = t1.balance +t2.balance	200	100	<b>200</b>	
<b>Unlock(t1)</b>	200	100	<b>200</b>	
<b>Unlock(t2)</b>	200	100	<b>200</b>	
<b>Commit</b>	200	100	<b>200</b>	

# SERIALIZABLE (un possibile schedule)

SUM = 300

Q	t1.balance	t2.balance	SUM	Ti
S-lock(accounts)				
R(t1)	100	200		
R(t2)	100	200		
	100	200		S-lock(accounts)
	100	200		R(t1)
	100	200		R(t2)
	100	200		valX = t1.balance
	100	200		valY = t2.balance
	100	200		X-lock(accounts)
sum = t1.balance +t2.balance	100	200	300	wait
Unlock(accounts)	100	200	300	wait
Commit	100	200	300	wait
	100	200		R(t1)
	100	200		t1.balance=valY
	200	200		W(t1)
	200	200		R(t2)
	200	200		t2.balance=valX
	200	100		W(t2)
	200	100		Unlock(accounts)
	200	100		Commit

# LIVELLI DI ISOLAMENTO: QUALE SCEGLIERE?

- **READ COMMITTED** aumenta concorrenza ma genera errori di consistenza
- **SERIALIZABLE** risolverebbe il problema (i lock in lettura vengono rilasciati al termine della transazione) ma può essere inefficiente per query read-only lunghe
  - Se  $Q$  è molto lunga (calcolo molto complesso o molte tuple da leggere), ritarda la partenza di tutte le  $T_i$
  - Ogni  $T_i$  deve aspettare che  $Q$  finisca prima di iniziare
  - Possono verificarsi deadlock
- Le risposte corrette sono ottenute a prezzo di minore concorrenza, maggior tempo medio di risposta e minore throughput

# CONTROLLO DELLA CONCORRENZA

- Oltre alla possibilità di scegliere il livello di isolamento più opportuno, è possibile aumentare la concorrenza con alcune varianti del protocollo utilizzato
- Una di queste riguarda i **protocolli di controllo della concorrenza multiversio**n

# CONTROLLO DELLA CONCORRENZA

## MULTIVERSIONE

- I protocolli di gestione della concorrenza **multiversione**, quando un elemento viene scritto da una transazione, tengono traccia del suo vecchio valore
- Per ogni elemento X vengono quindi mantenute versioni multiple X<sub>1</sub>,...,X<sub>n</sub>
- Quando una transazione deve leggere X, se possibile, viene concesso il lock shared scegliendo la versione di X che permette di garantire la serializzabilità
- Grazie al meccanismo delle versioni, alcune operazioni in lettura che verrebbero messe in attesa dal 2PL strong possono invece essere eseguite scegliendo opportunamente la versione
- **Pro:** si riduce il numero di transazioni in attesa a causa di operazioni di lettura
- **Contro:** si aumenta il costo di gestione della concorrenza dovuto alla necessità di gestire le versioni

# CONTROLLO DELLA CONCORRENZA

## MULTIVERSIONE – 2PL MULTIVERSIONE

- Si permette a una transazione T' di **leggere** un dato X anche se è già stato rilasciato un X-lock su X a una transazione T in esecuzione (e quindi verrebbe generato un conflitto)
- Questo viene realizzato mantenendo due versioni per ogni dato X, **una delle quali deve sempre essere stata scritta dall'ultima transazione che ha eseguito il commit**
  - X1: versione scritta da una transazione che ha eseguito il **commit** (versione committed) e letta dalle altre transazioni
  - X2: creata nel momento in cui una transazione acquisisce un lock in **scrittura** (al più uno nel sistema)

# CONTROLLO DELLA CONCORRENZA

## MULTIVERSIONE – 2PL MULTIVERSIONE

- Mentre una transazione **T** detiene il lock in scrittura, le altre che vogliono leggere X **leggono** la versione committed **X1** (quindi lettura e scrittura non sono in conflitto)
- **T** se vuole **modificare** X, viene **creata e modificata** la versione **X2**, ma non può modificare la versione X1
- Quando **T** vuole fare il commit
  - deve ottenere un lock particolare, chiamato **certify lock** su ogni elemento su cui ha un lock in scrittura
  - se il **certify lock** viene **rilasciato**, la versione **X2** diventa **la nuova versione committed X1** (in quanto T ha effettuato il commit)
  - **T** rilascia il certify lock su X2

# CONTROLLO DELLA CONCORRENZA

## MULTIVERSIONE – 2PL MULTIVERSIONE

	S	X
S	OK	NO
X	NO	NO

Matrice di compatibilità 2PL

	S	X	C
S	OK	OK	NO
X	OK	NO	NO
C	NO	NO	NO

Matrice di compatibilità 2PL  
multiversione

# CONTROLLO DELLA CONCORRENZA

## MULTIVERSIONE – 2PL MULTIVERSIONE

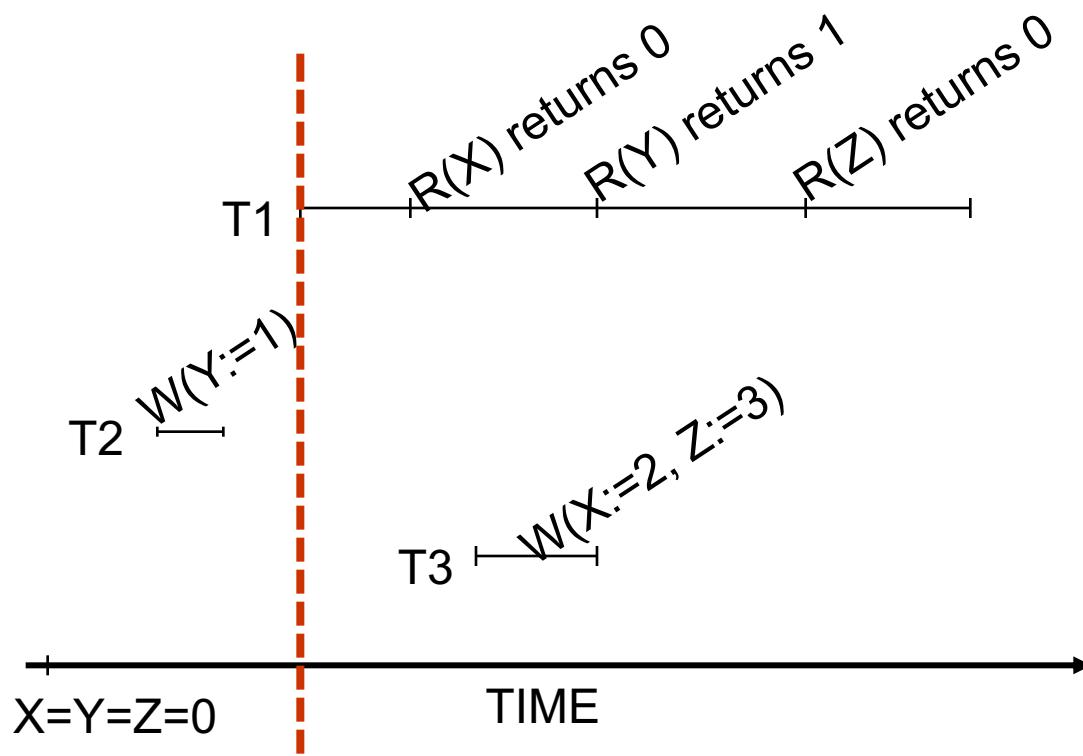
- Nel multiversion 2PL operazioni di read e write di transazioni in conflitto possono essere eseguite concorrentemente
- Questo aumenta il grado di concorrenza ma può ridurre throughput, a causa dei costi di gestione
- Come nel 2PL strong si possono verificare deadlock tra transazioni in conflitto

# CONTROLLO DELLA CONCORRENZA

## MULTIVERSIONE – SNAPSHOT ISOLATION

- Variante del protocollo 2PL multiversione
- **Nessuna transazione chiede lock in lettura**
- Ogni lettura viene eseguita sulla versione dei dati esistente al momento dell'inizio della transazione, generata da transazioni che hanno terminato con commit
- **Le scritture vengono gestire in accordo al 2PL strong**

# ESEMPIO



# ESEMPIO

- accounts( number, branchnum, balance)
- Q: somma
  - SELECT sum(balance) FROM accounts;
- N transazioni  $T_i$ : scambiano i saldi tra due conti
  - SELECT balance INTO valX FROM accounts WHERE number=X;  
SELECT balance INTO valY FROM accounts WHERE number=Y;
  - UPDATE accounts SET balance=valX WHERE number=Y;  
UPDATE accounts SET balance=valY WHERE number=X;
- Consideriamo due tuple t1 e t2, tali che t1.number =X e t2.number=Y

## SNAPSHOT (uno tra i possibili schedule)

Q	T1.balance	T2.balance	SUM	Ti
	100	200		
R(t1) ←	100	200		
	100	200		
	100	200		R(t1)
	100	200		R(t2)
	100	200		valX = t1.balance
	100	200		valY = t2.balance
	100	200		X-lock(t1)
	100	200		X-lock(t2)
	100	200		R(t1)
	100	200		t1.balance=valY
	200	200		W(t1)
	200	200		R(t2)
	200	200		t2.balance=valX
R(t2) ↓	200	200		
	200	100		W(t2)
	200	100		Unlock(t1)
	200	100		Unlock(t2)
	200	100		Commit
sum = t1.balance +t2.balance	200	100	300	
Commit	200	100	300	

SUM = 300

# ESEMPIO

- Se la query Q viene eseguita con snapshot isolation:
  - Q vede lo stato corrispondente all'inizio della sua esecuzione, anche se nel frattempo sono state effettuate modifiche
- La snapshot isolation “corregge” le query read-only senza usare locking

# CONTROLLO DELLA CONCORRENZA

## MULTIVERSIONE – SNAPSHOT ISOLATION

- La snapshot isolation può produrre schedule non serializzabili

T1	X	Y	T2
	3	17	
R(Y)	3	17	
X=Y	3	17	
	3	17	R(X)
	3	17	Y=X
X-lock(X)	3	17	
W(X)	17	17	
	17	17	X-lock(Y)
	17	3	W(Y)
Unlock(X)	17	3	
Commit	17	3	
	17	3	Unlock(Y)
	17	3	Commit

T1	X	Y	T2
	3	17	
R(Y)	3	17	
X=Y	3	17	
W(X)	17	17	
Commit	17	17	
	17	17	R(X)
	17	17	Y=X
	17	17	W(Y)
	17	17	Commit

Seriele: T1;T2

T1	X	Y	T2
	3	17	
	3	17	R(X)
	3	17	Y=X
	3	3	W(Y)
	3	3	Commit
R(Y)	3	3	
X=Y	3	3	
W(X)	3	3	
Commit	3	3	

Seriele: T2;T1

# LIVELLI DI ISOLAMENTO – ORACLE E POSTGRESQL

- Per ogni livello di isolamento, lo standard specifica solo i requisiti minimi
- I sistemi possono quindi ulteriormente restringere quanto richiesto
- In particolare, PostgreSQL e Oracle implementano **alcuni livelli di isolamento di SQL con snapshot isolation**
- Tre soli livelli di isolamento in PostgreSQL (a partire da 9.1)
  - SERIALIZABLE
  - REPEATABLE READS
  - READ COMMITTED = READ UNCOMMITTED

# LOCK ESPLICITI IN SQL

- Come visto, la richiesta di lock è **implicita**
- Poiché acquisire i lock e gestirli richiede tempo e spazio, se una transazione sa di dover elaborare molte tuple di una relazione può richiedere **esplicitamente** di porre un lock (**SHARE** o **EXCLUSIVE MODE**) sull'intera relazione

```
LOCK TABLE Studenti IN SHARE MODE ;  
SELECT *  
FROM   Studenti  
WHERE  DataNascita < '11/07/1982' ;
```

# ESEMPIO

- Database bancario
- Relazione accounts( number, branchnum, balance)
  - **Lock su tutta la relazione:** poco spazio per computazioni concorrenti
    - quasi tutte le transazioni saranno depositi/prelievi, quindi serve exclusive lock
  - **Lock su singole tuple:** maggiore concorrenza ma elevato aumento di complessità
    - Tante tuple
    - Tante operazioni
    - Tante richieste di lock
    - Tante verifiche di potenziali conflitti

# LOCK ESPLICITI IN SQL

- Lock su relazioni
  - servono pochi lock
  - basso grado di concorrenza
- Lock su tuple
  - servono più lock
  - maggior grado di concorrenza

# LOCK ESCALATION

- **Lock escalation:** quando il numero di lock ad un certo livello di granularità (ad esempio tuple) è elevato, il DBMS modifica il lock portandolo ad un livello di granularità più elevato (ad esempio, tabella)
- In questo modo il numero di lock viene ridotto ma ovviamente la concorrenza si riduce e aumenta la possibilità di generare deadlocks
- Non tutti i sistemi usano la lock escalation
  - Oracle e PostgreSQL non la supportano
  - Microsoft SQL Server la supporta

# GESTIONE DEL RIPRISTINO

# GESTIONE DEL RIPRISTINO

- Sappiamo già che ogni transazione o termina con commit o con abort
- Dopo un abort o un commit non può più “cambiare idea”
- Anche in caso di guasti:
  - Gli effetti delle transazioni **committed** devono essere permanenti (**Durability**)
  - Gli effetti delle transazioni **aborted** non devono lasciare tracce (**Atomicity**)
- Durability e Atomicity vengono assicurati da una particolare componente del Transaction Manager: il **gestore del ripristino**

# GESTIONE DEL RIPRISTINO

- La gestione del ripristino rappresenta l’“altra faccia” della gestione delle transazioni, relativa al trattamento dei guasti o malfunzionamenti (“failure”), ovvero di **tutti quegli eventi anomali che possono pregiudicare il corretto funzionamento delle transazioni**
- Il gestore del ripristino deve identificare i malfunzionamenti e ripristinare la base di dati allo stato (consistente) precedente il malfunzionamento

# MALFUNZIONAMENTI

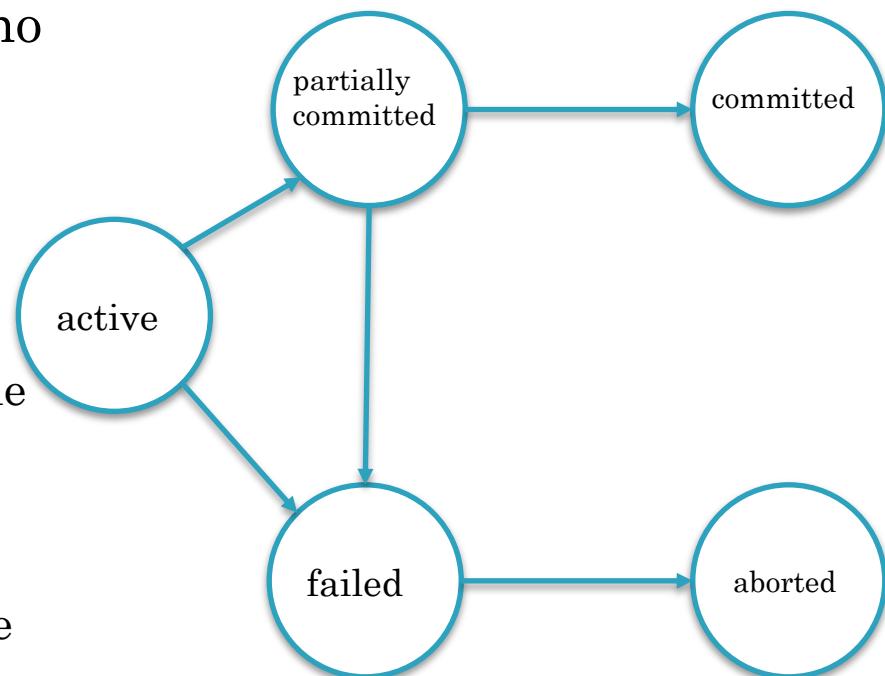
- II tipi di malfunzionamenti sono essenzialmente 3:
  - **Transaction failure:** è il caso in cui una transazione abortisce; gli effetti della transazione sul DB devono essere annullati
  - **System failure:** il sistema ha un guasto hardware o software che provoca l'interruzione di tutte le transazioni in esecuzione, senza però danneggiare la memoria permanente (dischi); spesso dovuto a problemi a memoria volatile (centrale)
  - **Media (o device) failure:** in questo caso il contenuto (persistente) della base di dati viene danneggiato

# CLASSIFICAZIONE DELLE MEMORIE

- Per analizzare i meccanismi di ripristino utilizzati dai DBMS, è conveniente iniziare da una classificazione delle memorie in gioco:
  - **memoria volatile:** le informazioni contenute vengono perse in caso di cadute di sistema
    - esempi: memoria principale e cache
    - coinvolta in **system failures**
  - **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
    - coinvolta in **media failures**
  - **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
    - teoricamente non è mai coinvolta in **failures**

# MODELLO ASTRATTO DI ESECUZIONE

- La gestione del ripristino si basa su un 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 (tutto è stato reso persistente)



# MODELLO ASTRATTO DI ESECUZIONE - OSSERVAZIONI

- In caso di Abort, eventuali scritture esterne osservabili (cioè scritture che non possono essere "cancellate", ad es. su terminale o stampante) eseguite dalla transazioni non possono essere eliminate
- 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 (system failures or media failures)
  - **eliminare la transazione** se si verificano transaction failures che possono essere corretti solo riscrivendo il programma applicativo

# ESEMPIO

- T transazione che trasferisce 10K Euro dal conto A al conto B, con A = 50K Euro, B = 150K Euro
- 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 → stato (inconsistente) in cui A = 30K e B = 160K
  - non si riesegue T → stato corrente (inconsistente) in cui A = 40K e B = 150K
- 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 avrebbe terminato con successo

# RECOVERY CON LOG

- Le attività di ripristino eseguite a valle del verificarsi di un **transaction o di un system failure** vengono genericamente indicate con il termine **ripresa a caldo**
- 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
- **il file di log si assume memorizzato su memoria stabile**
- **tramite il log, il sistema può gestire transaction failures e system failures**

# RECOVERY CON LOG

- **Log:** lista ordinata di informazioni che permettono di effettuare operazioni di REDO/UNDO di transazioni in caso di failure
- Le scritture nel log sono sequenziali
- Nel log vengono scritte le informazioni minimali
  - più aggiornamenti stanno nella stessa pagina di log

# RECOVERY CON LOG

- Se una pagina P del DB viene modificata dalla transazione T, il Log contiene un record del tipo  
 $(\underline{\text{LSN}}, \text{T}, \text{PID}, \text{before}(P), \text{after}(P), \text{prevLSN})$ 
  - LSN = Log Sequence Number (n. progressivo del record nel Log)
  - T = identificatore della transazione
  - PID = identificatore della pagina modificata
  - before(P) = è la cosiddetta “*before image*” di P, ovvero il contenuto di P prima della modifica
  - after(P) = è l’“*after image*” di P, ossia il contenuto di P dopo la modifica
  - prevLSN = LSN del precedente record del Log relativo a T

Per semplicità nel seguito assumiamo che un blocco corrisponda a un singolo record

# RECOVERY CON LOG

- Il Log contiene anche record che specificano l'inizio (**BEGIN**) di una transazione e la sua terminazione (**COMMIT** o **ROLLBACK**)

LSN	T	PID	before(P)	after(P)	prevLSN
...					
235	T1	BEGIN			-
236	T2	BEGIN			-
237	T1	P15	(abc, 10)	(abc, 20)	235
238	T2	P18	(def, 13)	(ghf, 13)	236
239	T1	COMMIT			237
240	T2	P19	(def, 15)	(ghf, 15)	238
241	T3	BEGIN			-
242	T2	P19	(ghf, 15)	(ghf, 17)	240
243	T3	P15	(abc, 20)	(abc, 30)	241
244	T2	ROLLBACK			242
245	T3	COMMIT			243
...					

# PROTOCOLLO WAL - WRITE-AHEAD LOGGING

- Affinché il Log possa essere utilizzato per ripristinare lo stato del DB a fronte di malfunzionamenti, è importante che venga applicato il cosiddetto protocollo **WAL** (“Write-ahead Logging”):

prima di scrivere su disco una pagina P modificata,  
il corrispondente Log record deve essere già stato scritto nel Log

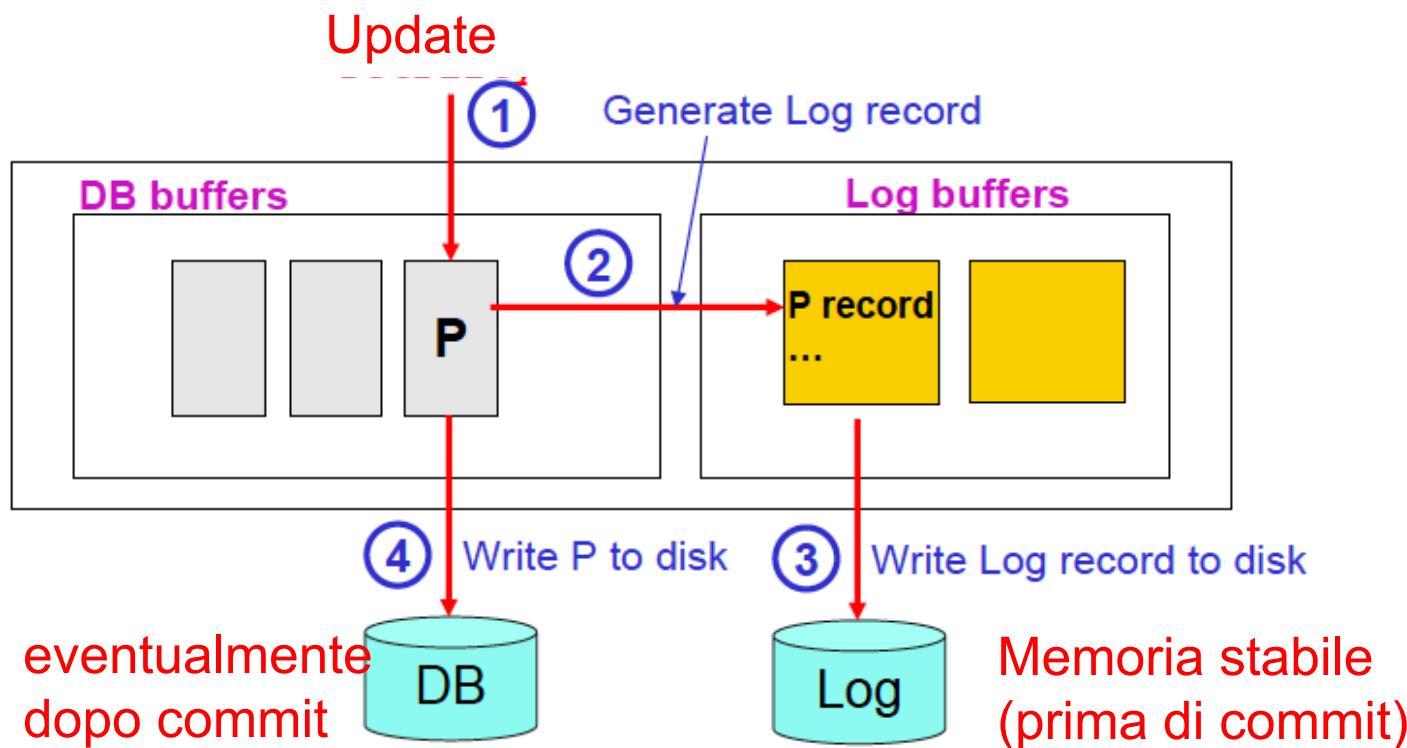
- Intuitivamente, se il protocollo WAL non viene rispettato è possibile che
  - Una transazione T modifichi il DB aggiornando una pagina P
  - Si verifichi un system failure prima che il Log record relativo alla modifica di P sia stato scritto nel Log
- In questo caso è evidente che non ci sarebbe alcun modo di riportare il DB allo stato iniziale

# PROTOCOLLO WAL - WRITE-AHEAD LOGGING

- Write-Ahead Logging Protocol:
  - Il **record di log** per un aggiornamento deve essere forzato su memoria stabile prima che la corrispondente pagina dati arrivi su disco
    - garantisce l'atomicità
  - **Tutti i record di log** per una transazione devono essere forzati su memoria stabile prima del commit
    - garantisce la persistenza
- Quindi una transazione si considera committed solo quando tutti i suoi record di log sono stati salvati su memoria stabile!

# IMPLEMENTAZIONE DEL PROTOCOLLO WAL

- La responsabilità di garantire il rispetto del protocollo WAL è del **Buffer Manager**, che gestisce, oltre ai buffer del DB, anche quelli del Log
- In figura viene riportato l'ordine in cui si succedono le varie operazioni relative alla modifica di una pagina P



# IMPLEMENTAZIONE DEL PROTOCOLLO WAL

**Stato corrente della base di dati =  
Stato corrente della base di dati su disco**  
+  
**Log su disco**

- Stato corrente della base di dati
  - riflette tutte e sole le azioni delle transazioni committed
- Stato corrente della base di dati su disco
  - riflette solo le azioni delle transazioni effettuate su pagine che sono già state scaricate su disco
  - può non riflettere tutte le azioni di transazioni committed (alcuni blocchi devono ancora essere forzati su disco)
  - può riflettere azioni di transazioni non committed (alcuni blocchi sono stati forzati su disco ma devono essere riportati allo stato iniziale)

# IMPLEMENTAZIONE DEL PROTOCOLLO WAL

## - GESTIONE DEL BUFFER

Per implementare il protocollo WAL, bisogna determinare quando eseguire step (4)

- Quando una transazione T modifica una pagina P, il Buffer Manager ha 2 possibilità:
  - **Politica No-steal:** Mantenere la pagina P nel buffer, e attendere che T abbia eseguito **COMMIT** prima di scriverla su disco
  - **Politica Steal:** Scrivere P quando “più conviene” (per liberare il buffer o per ottimizzare le prestazioni di I/O), eventualmente anche prima della terminazione di T
- E’ evidente che nel primo caso non ci sarà mai bisogno di eseguire l’**“UNDO”** di transazioni che abortiscono, ma si rischia di esaurire lo spazio a disposizione in memoria centrale (in quanto una transazione non può “rubare” i buffer ad altre transazioni ancora in esecuzione)

Attenzione: la politica no-steal non significa che tutte le pagine nel buffer verranno copiate su disco **al momento del commit**, ma **dal momento del commit** (eventualmente anche successivamente)

# IMPLEMENTAZIONE DEL PROTOCOLLO WAL

## - ESECUZIONE DEL COMMIT

Per implementare il protocollo WAL, bisogna anche determinare come comportarsi con blocchi dati nel buffer non ancora forzati su disco al momento del commit

- Quando una transazione esegue **COMMIT** si hanno ancora 2 possibilità:
  - **Politica Force**: prima di scrivere il record di **COMMIT** sul Log, che “ufficializza” la conclusione della transazione, si forza la scrittura su disco di tutte le pagine modificate da T
  - **Politica No-force**: si scrive subito il record di **COMMIT** sul Log; quindi, quando T termina, alcune delle sue modifiche possono ancora non essere state rese persistenti su disco
- La politica **Force** può generare **molti I/O inutili**; ad esempio, se una pagina P è frequentemente modificata (“**hot spot**”), deve essere scritta su disco ad ogni **COMMIT**. Con la politica No-force ciò non è necessario, e P viene scritta su disco solo se deve essere rimpiazzata nel buffer, secondo la politica adottata dal Buffer Manager

# TRANSACTION FAILURE

- Con la politica **Steal**, se una transazione T abortisce è possibile che alcune pagine da essa modificate siano già state scritte su disco
- Per annullare (**UNDO**) queste modifiche si scandisce il Log a ritroso (usando i prevLSN) e si ripristinano nel DB le **before image** delle pagine modificate da T

LSN	T	PID	before(P)	after(P)	prevLSN
...					
236	T2	<b>BEGIN</b>			-
237	T1	P15	(abc, 10)	(abc, 20)	235
238	T2	P18	(def, 13) ← (ghf, 13)		236
239	T1	<b>COMMIT</b>			237
240	T2	P19	(def, 15) ← (ghf, 15)		238
241	T3	<b>BEGIN</b>			-
242	T2	P19	(ghf, 15) ← (ghf, 17)		240
243	T3	P15	(abc, 20)	(abc, 30)	241
244	T2	<b>ROLLBACK</b>			242

# TRANSACTION FAILURE

- Con la politica **No Steal**, se una transazione abortisce, nessun blocco è stato aggiornato su disco
- Quindi non ci sono operazioni da eseguire

LSN	T	PID	before(P)	after(P)	prevLSN
...					
235	T1	BEGIN			-
236	T2	BEGIN			-
237	T1	P15	(abc, 10)	(abc, 20)	235
238	T2	P18	(def, 13)	(ghf, 13)	236
239	T1	COMMIT			237
240	T2	P19	(def, 15)	(ghf, 15)	238
241	T3	BEGIN			-
242	T2	P19	(ghf, 15)	(ghf, 17)	240
243	T3	P15	(abc, 20)	(abc, 30)	241
244	T2	ROLLBACK			242

# SYSTEM FAILURE

- Al momento di un **system failure**, una transazione potrebbe avere già scritto **COMMIT** nel file di log oppure no
- Se ha già scritto **COMMIT**
  - Se la politica è **No-force**, non è detto che tutte le modifiche operate da una transazione T che ha eseguito COMMIT siano state riportate su disco; pertanto T va “rifatta”(REDO), riscrivendo le after image che si trovano sul Log
  - Se la politica è **Force**, tutte le modifiche sono già state salvate su disco prima di scrivere COMMIT su log e quindi non c’è nulla da fare
- Se non ha scritto **COMMIT**
  - Si prosegue come per **Transaction failure** in quanto la transazione deve essere disfatta

LSN	T	PID	before(P)	after(P)	prevLSN
...					
235	T1	BEGIN			-
236	T2	BEGIN			-
237	T1	P15	(abc, 10) → (abc, 20)		235
238	T2	P18	(def, 13)	(ghf, 13)	236
239	T1	COMMIT			237
...					

# QUINDI IN CASO DI FAILURE...

- Se la politica è **STEAL**
  - Log a modifiche immediate
  - Serve procedura di **UNDO**
- Se la politica è **NO STEAL**
  - Log a modifiche differite
  - Inutile mantenere anche la versione vecchia dei dati perché gli aggiornamenti non dovranno mai essere disfatti
- Se la politica è **NO FORCE**
  - Serve procedura di **REDO**
- Se la politica è **FORCE**
  - Non serve procedura di **REDO** perché al commit i dati saranno già tutti copiati su disco (e se una transazione non è arrivata al commit deve solo essere disfatta)

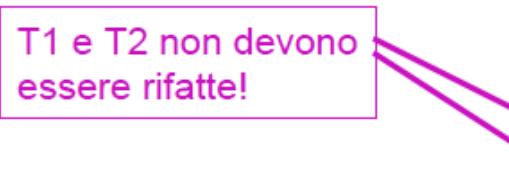
**REDO e UNDO** devono essere **idempotenti**, per non generare risultati errati in presenza di failure in cascata

# IN SINTESI

- In generale, il recovery con log, anche a modifiche immediate (**steal**), ha un costo
  - scrivendo immediatamente i dati su disco:
    - ogni pagina richiede un seek
    - I/O risultante sarebbe random e con cattive prestazioni
- Tuttavia, la politica **steal** è quella più utilizzata perché non richiede di mantenere nel buffer necessariamente tutti i blocchi modificati da transazioni che non hanno effettuato il commit
- La politica **force** è quella meno utilizzata a causa dei costi
- **Quindi la combinazione steal-no force è a più comune**
- La politica **steal-no force** ha però lo svantaggio di eseguire una procedura di recovery per ciascuna transazione eseguita durante la procedura di restart in caso di failure
  - **undo** se non è arrivata al commit
  - **redo** se è arrivata al commit

# CHECKPOINT

- La procedura di **restart** è quella che si occupa di riportare il DB in uno stato consistente a fronte di system failure; per ridurre i tempi di restart, periodicamente si può eseguire un “**checkpoint**”, ovvero una **scrittura forzata su disco delle pagine modificate**
  - L'esecuzione del checkpoint viene registrata scrivendo sul Log un record **CKP** (checkpoint)
- In questo modo se T ha eseguito **COMMIT** prima del checkpoint si è sicuri che T non deve essere rifatta



LSN	T	PID	before(P)	after(P)	prevLSN
237	T3	P15	...	...	...
238	T2	P18	...	...	...
239	T1	P17	...	...	...
240	T1	COMMIT			...
241	T2	COMMIT			...
242		CKP			
243	T3	P19	...	...	...
244	T3	COMMIT			...

La registrazione del checkpoint sul log viene effettuata dopo avere completato la scrittura forzata su disco delle pagine modificate, per far fronte ad eventuali malfunzionamenti durante l'operazione di checkpoint

# CHECKPOINT

- Il checkpoint riduce i tempi relativi alla procedura di restart
- il sistema **periodicamente** (al checkpoint):
  - forza tutte le **pagine di log** residenti in memoria principale su **memoria stabile**
  - forza tutte le **pagine dati** di buffer su **disco**
  - forza il record <CKP> (o <checkpoint>) sul log in memoria stabile
- in questo modo gli effetti delle transazioni che hanno effettuato il commit prima del checkpoint sono memorizzati in modo permanente nella base di dati e i relativi blocchi nel buffer possono essere rilasciati
- In caso di failure, lo stato della base di dati verrà ripristinato considerando i record di log seguenti all'ultimo checkpoint

# MEDIA FAILURE

- I file di log permettono di far fronte a malfunzionamenti software (transaction failure) e della memoria volatile (system failure)
- In questo caso la procedura di restart viene anche detta **ripresa a caldo**
- per fare fronte a **malfunzionamenti della memoria non volatile**, è necessario utilizzare ulteriori meccanismi
- la procedura di restart in questo caso viene detta **ripresa a freddo**
- **dump**: copia completa della base di dati, normalmente creata quando il sistema non è operativo (nessuna transazione in esecuzione)
- la copia è memorizzata in **memoria stabile**, ed è chiamata **backup**
- un record <dump> nel log segnala la presenza di un backup effettuato in un certo momento ed identifica il file o il device su cui è stato effettuato il dump

## MEDIA FAILURE - RIPRESA A FREDDO

- Al restart, **si accede al dump e si ripristina il contenuto della base di dati** su memoria non volatile
- **Si effettua una ripresa a caldo:** si accede il file di log e si esegue il ripristino come discusso in precedenza

# ESEMPIO

# POLITICA NO STEAL, NO FORCE (LOG A MODIFICHE DIFFERITE)

- Nell'esempio usiamo log semplificato
- Supponiamo inoltre che ogni record corrisponda ad un blocco in modo da semplificare ulteriormente la rappresentazione delle informazioni nel log
- Durante l'esecuzione di una transazione Ti:
  - prima che Ti cominci la propria esecuzione viene scritto nel log buffer il record  
 $\langle Ti \text{ start} \rangle$
  - ogni operazione di scrittura  $\text{write}[x]$  produce un record di log buffer della forma:  
 $\langle Ti; x; \text{nuovo valore} \rangle$
  - quando Ti entra in stato partially committed, viene scritto nel log buffer il record  
 $\langle Ti \text{ commit} \rangle$

# POLITICA NO STEAL, NO FORCE (LOG A MODIFICHE DIFFERITE)

- a seguito di un system failure:
  - se per una transazione  $T_i$  il log contiene entrambi i record  $\langle T_i \text{ start} \rangle$  e  $\langle T_i \text{ commit} \rangle$ ,  $T_i$  viene rieseguita, cioè viene effettuato redo( $T_i$ )
  - se per una transazione  $T_i$  il log contiene il record  $\langle T_i \text{ start} \rangle$  ma non il record  $\langle T_i \text{ commit} \rangle$ , non si deve eseguire alcuna azione

# POLITICA NO STEAL, NO FORCE (LOG A MODIFICHE DIFFERITE)

T1

Read lr

$lr = lr - 1.500$

Write lr

Read cc1

$cc1 = cc1 + 1.500$

Write cc1

Commit

T2

Read cc2

$cc2 = cc2 - 20.000$

Write cc2

Commit

- viene eseguita prima T1 e poi T2
- inizialmente
  - $lr = 50.000$
  - $cc1 = 60.000$
  - $cc2 = 80.000$

## POLITICA NO STEAL, NO FORCE (LOG A MODIFICHE DIFFERITE)

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

< T1, start >

< T1, lr,48.500 >

< T1, cc1,61.500 >

< T1, commit >

< T2, start >

< T2, cc2, 60.000 >

< T2, commit >

## POLITICA NO STEAL, NO FORCE (LOG A MODIFICHE DIFFERITE)

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write cc1", lo stato del log al momento del crash è il seguente:

< T1, start >

< T1, lr,48.500 >

< T1, cc1,61.500 >

- non si esegue alcuna operazione di redo

## POLITICA NO STEAL, NO FORCE (LOG A MODIFICHE DIFFERITE)

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write cc2", lo stato del log al momento del crash è il seguente:

```
< T1, start >
< T1, lr,48.500 >
< T1, cc1,61.500 >
< T1, commit >
< T2, start >
< T2, cc2, 60.000 >
```

- viene effettuato redo(T1) e lo stato della base di dati diventa: lr = 48.500 cc1 = 61.500 cc2 = 80.000

# POLITICA NO STEAL, NO FORCE (LOG A MODIFICHE DIFFERITE)

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Commit T2 ", lo stato del log al momento del crash è il seguente:

```
< T1, start >
< T1, lr,48.500 >
< T1, cc1,61.500 >
< T1, commit >
< T2, start >
< T2, cc2, 60.000 >
< T2, commit >
```

- vengono effettuate le operazioni redo(T1) e redo(T2) e lo stato della base di dati diventa: lr = 48.500 cc1 = 61.500 cc2 = 60.000

# POLITICA STEAL, NO FORCE (LOG A MODIFICHE IMMEDIATE)

- Log semplificato ai fini dell'esempio
- prima che Ti cominci la propria esecuzione viene scritto nel log il record <Ti start>
- ogni operazione di scrittura write[x] è preceduta dalla scrittura un record di log della forma:  
< Ti; x; vecchio valore, nuovo valore >
- quando Ti entra in stato partially committed,  
viene scritto nel log buffer il record  
<Ti commit>

# POLITICA STEAL, NO FORCE (LOG A MODIFICHE IMMEDIATE)

- a seguito di un system failure:
  - se per una transazione  $T_i$  il log contiene entrambi i record  $\langle T_i \text{ start} \rangle$  e  $\langle T_i \text{ commit} \rangle$ ,  $T_i$  viene rieseguita, cioè viene effettuato redo( $T_i$ )
  - se per una transazione  $T_i$  il log contiene il record  $\langle T_i \text{ start} \rangle$  ma non il record  $\langle T_i \text{ commit} \rangle$ ,  $T_i$  viene disfatta, cioè viene effettuato undo( $T_i$ )

## POLITICA STEAL, NO FORCE (LOG A MODIFICHE IMMEDIATE)

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

< T1, start >

< T1, lr, 50.000, 48.500 >

< T1, cc1, 60.000, 61.500 >

< T1, commit >

< T2, start >

< T2, cc2, 80.000, 60.000 >

< T2, commit >

## POLITICA STEAL, NO FORCE (LOG A MODIFICHE IMMEDIATE)

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write cc1", lo stato del log al momento del crash è il seguente:

< T1, start >

< T1, lr, 50.000, 48.500 >

< T1, cc1, 60.000, 61.500 >

- si esegue l'azione di undo(T1), il nuovo stato dei dati è:  
 $lr = 50.000$   
 $cc1 = 60.000$   
 $cc2 = 80.000$

# POLITICA STEAL, NO FORCE (LOG A MODIFICHE IMMEDIATE)

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Write cc2", lo stato del log al momento del crash è il seguente:

```
< T1, start >
< T1, lr, 50.000, 48.500 >
< T1, cc1, 60.000, 61.500 >
< T1, commit >
< T2, start >
< T2, cc2, 80.000, 60.000 >
```

- viene effettuato redo(T1) e undo(T2), lo stato della base di dati diventa:  
 $lr = 48.500$   
 $cc1 = 61.500$   
 $cc2 = 80.000$

# POLITICA STEAL, NO FORCE (LOG A MODIFICHE IMMEDIATE)

- se si verifica un crash subito dopo aver scritto su memoria stabile il record di log per l'operazione "Commit T2", lo stato del log al momento del crash è il seguente:

```
< T1, start >
< T1, lr, 50.000, 48.500 >
< T1, cc1, 60.000, 61.500 >
< T1, commit >
< T2, start >
< T2, cc2, 80.000, 60.000 >
< T2, commit >
```

- vengono effettuate le operazioni redo(T1) e redo(T2) e lo stato della base di dati diventa:
  - lr = 48.500
  - cc1 = 61.500
  - cc2 = 60.000