

Basi dati II

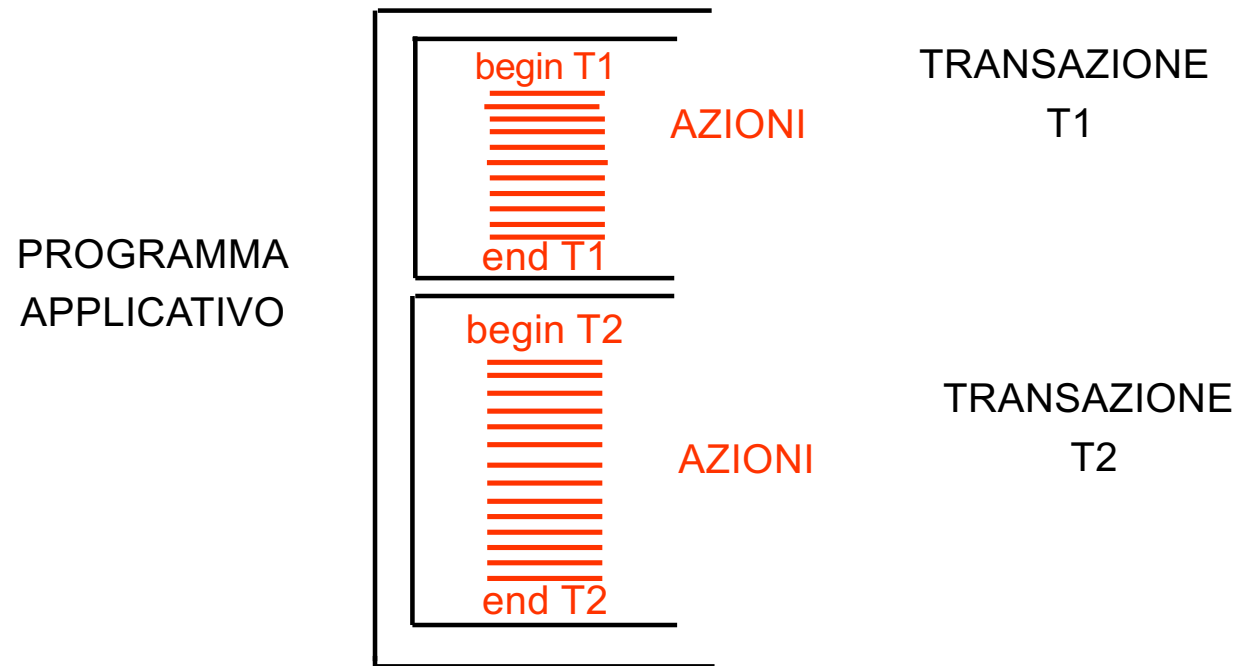
Gestione delle transazioni

Cap. 12 Basi di dati 5ed. Atzeni et al.

Definizione di transazione

- Transazione: parte di programma caratterizzata da un inizio (**begin-transaction**), una fine (**end-transaction**) che forma un'unità logica di elaborazione sulla base di dati
- All'interno della transazione deve essere eseguito una e una sola volta uno dei seguenti comandi
 - **commit work** per terminare correttamente
 - **rollback work** per abortire la transazione
- Un **sistema transazionale (OLTP)** è in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

Differenza fra applicazione e transazione



Esempio transazione

start transaction;

update ContoCorrente

set Saldo = Saldo + 10 where NumConto = 12202;

update ContoCorrente

set Saldo = Saldo – 10 where NumConto = 42177;

commit work;

Esempio transazione con varie decisioni

start transaction;

update ContoCorrente

set Saldo = Saldo + 10 where NumConto = 12202;

update ContoCorrente

set Saldo = Saldo – 10 where NumConto = 42177;

select Saldo into A

from ContoCorrente

where NumConto = 42177;

if (A>=0) then commit work

else rollback work;

Il concetto di transazione



- Una unità di elaborazione che gode delle proprietà "ACIDE"
 - Atomicità
 - Consistenza
 - Isolamento
 - Durabilità (persistenza)

Atomicità

- Una transazione è una unità **atomica** di elaborazione
- Non può lasciare la base di dati in uno stato intermedio
 - un guasto o un errore prima del commit devono causare l'annullamento (UNDO) delle operazioni svolte
 - un guasto o errore dopo il commit non deve avere conseguenze; se necessario vanno ripetute (REDO) le operazioni

Consistenza

- La transazione deve rispettare i vincoli di integrità
- Conseguenza:
 - se lo stato iniziale è corretto
 - anche lo stato finale è corretto

Isolamento

- La transazione non risente degli effetti delle altre transazioni concorrenti
 - l'esecuzione concorrente di una collezione di transazioni deve produrre un risultato che si potrebbe ottenere con una esecuzione sequenziale
- Conseguenza: una transazione non espone i suoi stati intermedi

Durabilità (Persistenza)

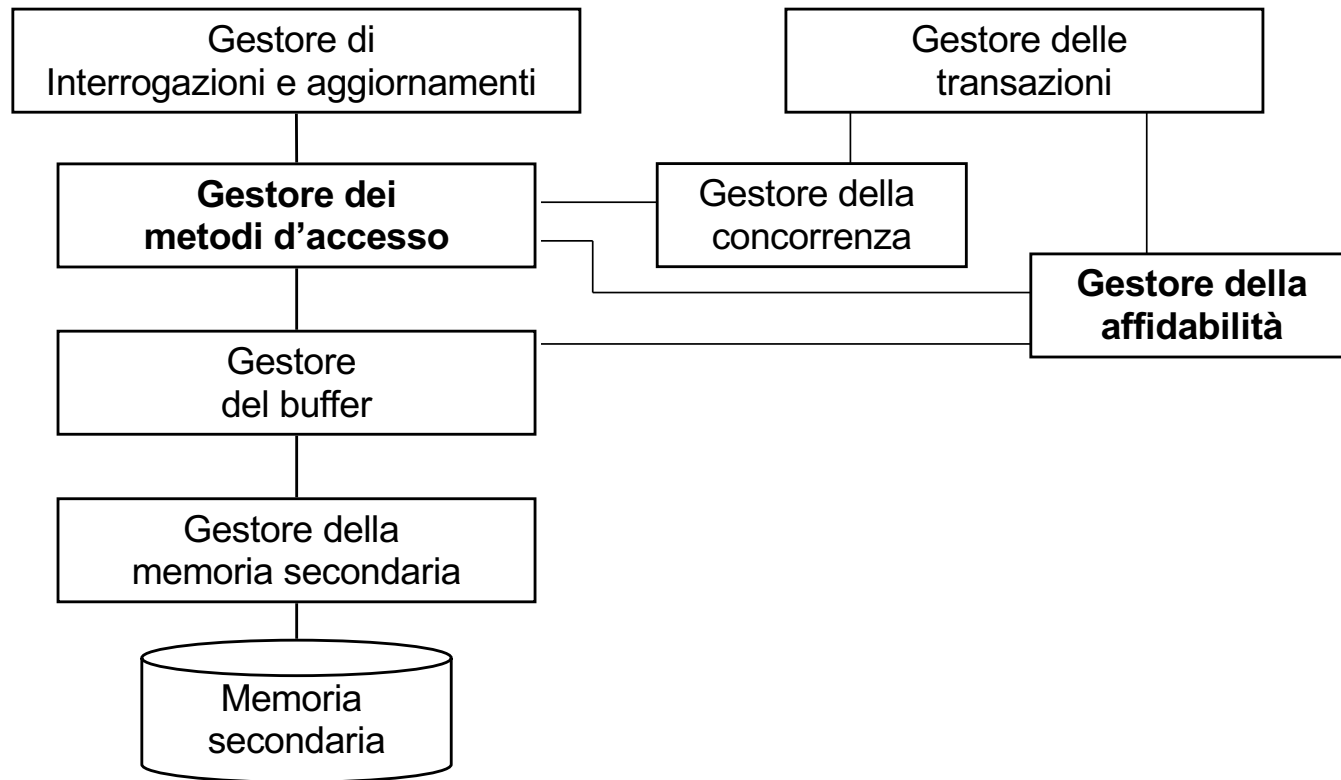
- Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti

Transazioni e moduli di DBMS

- Atomicità e persistenza
 - Gestore dell'affidabilità (Reliability manager)
- Isolamento:
 - Gestore della concorrenza
- Consistenza:
 - Gestore dell'integrità a tempo di esecuzione (con il supporto del compilatore del DDL)

Gestore degli accessi e delle interrogazioni

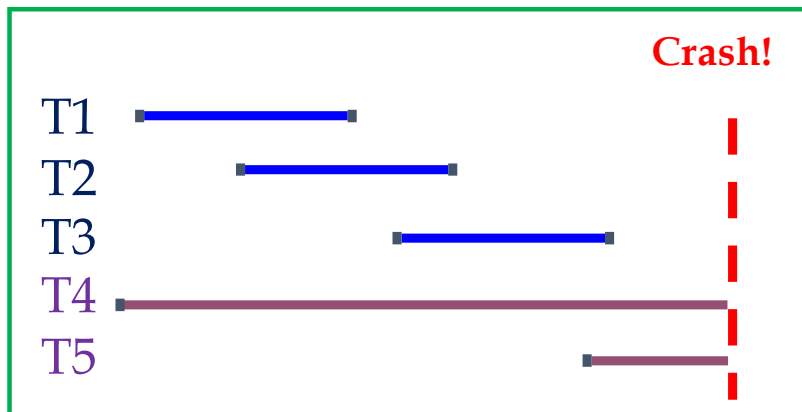
Gestore delle transazioni



Gestore dell'affidabilità

- Come garantisce l'atomicità e la persistenza in caso di guasto?

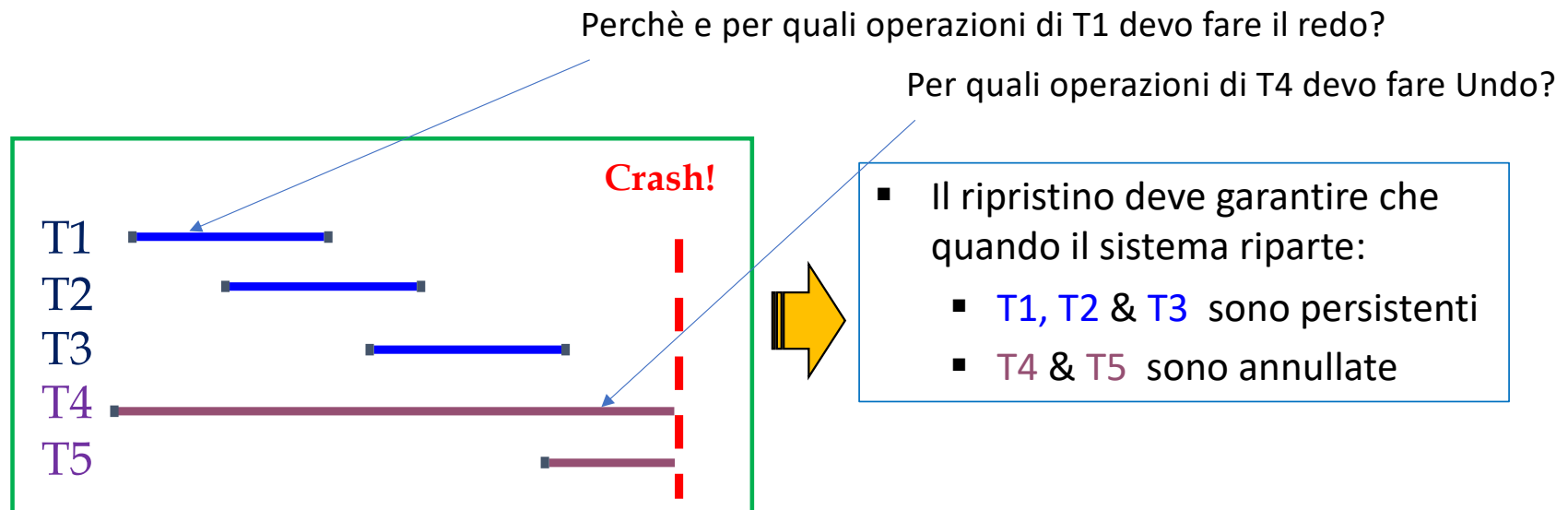
I dati sulla memoria
centrale sono persi



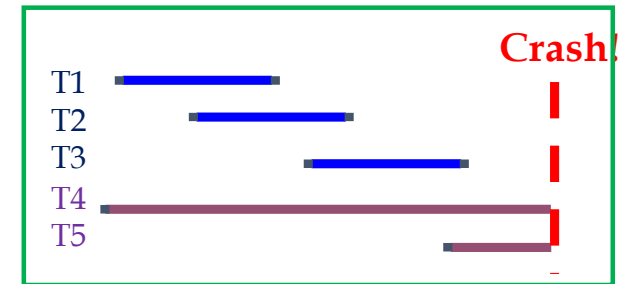
- Il ripristino deve garantire che quando il sistema riparte:
 - T1, T2 & T3 sono persistenti
 - T4 & T5 sono annullate

Gestore dell'affidabilità

- Come garantisce l'atomicità e la persistenza in caso di guasto?
 - Garantisce l'atomicità annullando (*undo*) le azioni delle transazioni che non hanno fatto il commit
 - Garantisce la persistenza rifacendo (*redo*) le azioni delle transazioni che hanno fatto il commit



Gestore dell'affidabilità



- Per capire il problema del ripristino è necessario capire cosa succede durante la normale esecuzione di una transazione
 - Q1: Le modifiche fatte da una transazione T su oggetto O (e.g., un record di una tabella) in un blocco/pagina del buffer sono scritte su memoria secondaria prima o dopo il commit di T?
 - Q2: Quando T esegue il commit, tutte le sue modifiche sono immediatamente scritte in memoria secondaria?

Dipende dalle politiche del buffer

Steal/No Steal, Force/No-force

Steal/no Steal: Quando si richiede una pagina al buffer e questo è pieno:

- **Approccio steal:** si seleziona una "vittima", una pagina occupata del buffer. I dati della vittima sono scritti in memoria secondaria (**flush**)
- **Approccio no-steal:** la transazione che ha richiesto la pagina viene posta in attesa

Force/no force:

- **Approccio Force:** tutte le pagine coinvolte da una transazione attiva vengono scritte in memoria di massa appena essa fa il commit
- **Approccio No-force:** ci si affida al flush per scrivere la pagine di transazioni che hanno fatto commit

Gestore dell'affidabilità



- Per capire il problema del ripristino è necessario capire cosa succede durante la normale esecuzione di una transazione
 - Q1: Le modifiche fatte da una transazione T su oggetto O (e.g., un record di una tabella) in un blocco/pagina del buffer sono scritte su memoria secondaria prima o dopo il commit di T?
 - Approccio **STEAL**: la pagina del buffer può essere memorizzata in mem. secondaria sia prima (in caso un'altra transazione richiedesse una pagina e il buffer è pieno), sia dopo il commit
 - Approccio **NO-STEAL**: No, non è consentito rubare. I dati sono memorizzati dopo il commit
 - Q2: Quando T esegue il commit, tutte le sue modifiche sono immediatamente scritte in memoria secondaria?
 - Sì, se si utilizza un approccio di **Force** per le transazioni
 - No, se si utilizza un approccio **No-force**

Steal vs. No-Steal

Force vs. No-Force

- Si hanno quattro possibili scenari:

	No-Steal	Steal
Force	Rigida, non permette al gestore del buffer ottimizzazioni.	costi alti I/O cost, Il buffer non mette in attesa le transazioni
No-Force	Bassi costi I/O Se pieno, il buffer mette in attesa le transazioni	Bassi costi I/O Il buffer non mette in attesa le transazioni



- La maggior parte dei DBMSs usano *steal, no-force*

Gestore dell'affidabilità

- Assicura atomicità e persistenza:
 - garantisce che le transazioni non vengano lasciate incomplete (con solo alcune operazioni eseguite) e che gli effetti delle transazioni chiuse con commit siano persistenti:
 - Redo, undo
- Gestisce:
 - l'esecuzione dei comandi transazionali
 - `start transaction (B, begin)`
 - `commit work (C)`
 - `rollback work (A, abort)`
 - le operazioni di ripristino (recovery) dopo i guasti :
 - *warm restart* e *cold restart*
- Per implementare correttamente redo/undo si utilizza un **log**:
 - Un archivio permanente che registra le operazioni svolte

Il log

- Il log è un file sequenziale gestito dal controllore dell'affidabilità, scritto in memoria stabile

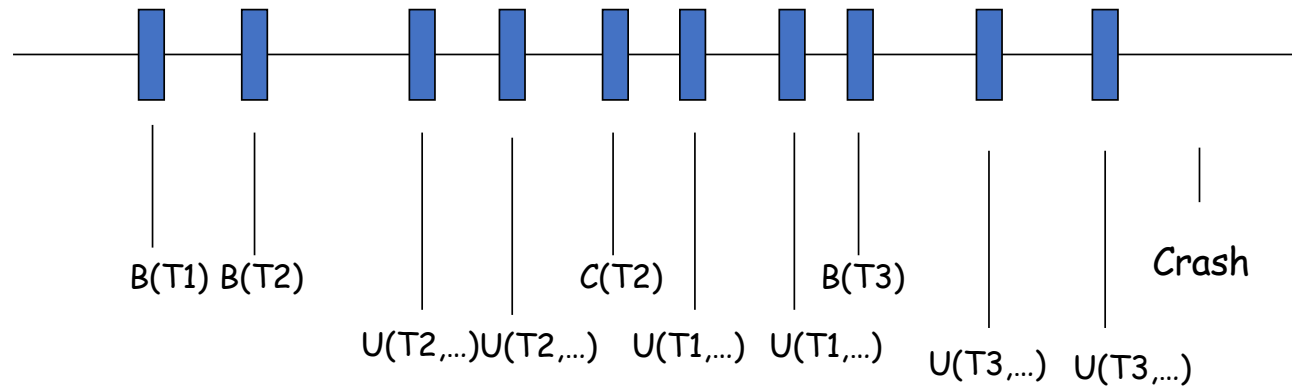
Persistenza delle memorie

- **Memoria centrale**: non è persistente
- **Memoria di massa**: è persistente ma può danneggiarsi
- **Memoria stabile**: memoria che non può danneggiarsi (è una astrazione):
 - perseguita attraverso la ridondanza:
 - dischi replicati
 - nastri
 - ...

Il log

- Il log è un file sequenziale gestito dal controllore dell'affidabilità, scritto in **memoria stabile**
- Il log memorizza tutte le operazioni eseguite in ordine sequenziale:
 - permette di ricostruire il contenuto della basi di dati a seguito di guasti
- I record nel log sono scritti per descrivere le attività di una transazione T
 - **Record Begin**, $B(T)$:
 - T identificativo transazione
 - **Record insert**, $I(T,O,AS)$:
 - T identificativo transazione,
 - O identificativo oggetto su cui avviene insert,
 - AS (After state) valore dell'oggetto O dopo l'insert
 - **Record delete**, $D(T,O,BS)$:
 - T identificativo transazione,
 - O identificativo oggetto su cui avviene delete,
 - BS (Before state) valore dell'oggetto O prima del delete
 - **Record update**, $U(T,O,BS,AS)$
 - T identificativo transazione,
 - O identificativo oggetto su cui avviene update,
 - BS (Before state) valore dell'oggetto O prima dell'update
 - AS (After state) valore dell'oggetto O dopo l'update
 - **Record commit**, $C(T)$:
 - T identificativo transazione,
 - **Record abort**, $A(T)$:
 - T identificativo transazione,

Struttura del log



Undo e redo

- Undo di una azione su un oggetto O :
 - `update`, `delete`: copia il valore del **before state** (BS) nell'oggetto O
 - `insert`: eliminare **after state** (AS) nell'oggetto O
- Redo di una azione su un oggetto O :
 - `insert`, `update`: copiare il valore dell'**after state** (AS) nell'oggetto O
 - `delete`: (ri)cancellare **before state** (AS) nell'oggetto O

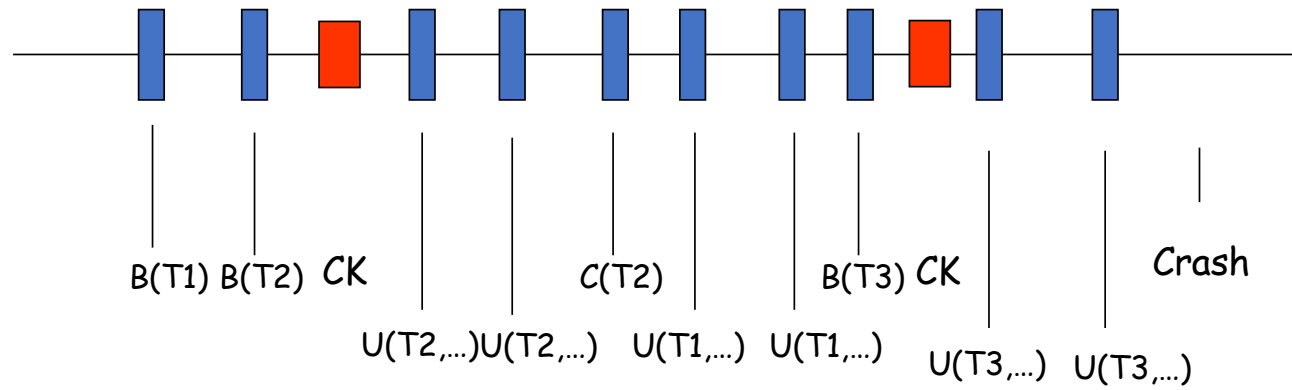
Il log

- I record di log per le attività di una transazione T permettono di svolgere le operazioni di ripristino, permettendo di ricostruire l'evoluzione della basi di dati in caso di guasti
- Per evitare di ricostruire “tutte” le operazioni del log, il gestore dell'affidabilità periodicamente esegue due operazioni:
 - Checkpoint
 - Dump

Checkpoint

- Operazione che serve a "fare il punto" della situazione, semplificando le successive operazioni di ripristino
- Ha lo scopo di registrare quali transazioni in un certo istante sono attive (non hanno ancora fatto commit)
- Esegue questi passi:
 - sospende l'accettazione di altre operazioni di scrittura, commit o abort da parte di ogni transazione
 - **trasferisce in memoria di massa** (operazione force) **tutte le pagine** del buffer su cui sono state eseguite modifiche da parte **di transazione che hanno già effettuato il commit**
 - scrive nel log un record di checkpoint che contiene gli identificativi delle transazioni attive
- Garantisce che
 - per tutte le transazioni che hanno effettuato il commit i dati sono in memoria di massa
 - le transazioni "a metà strada" sono elencate nel checkpoint

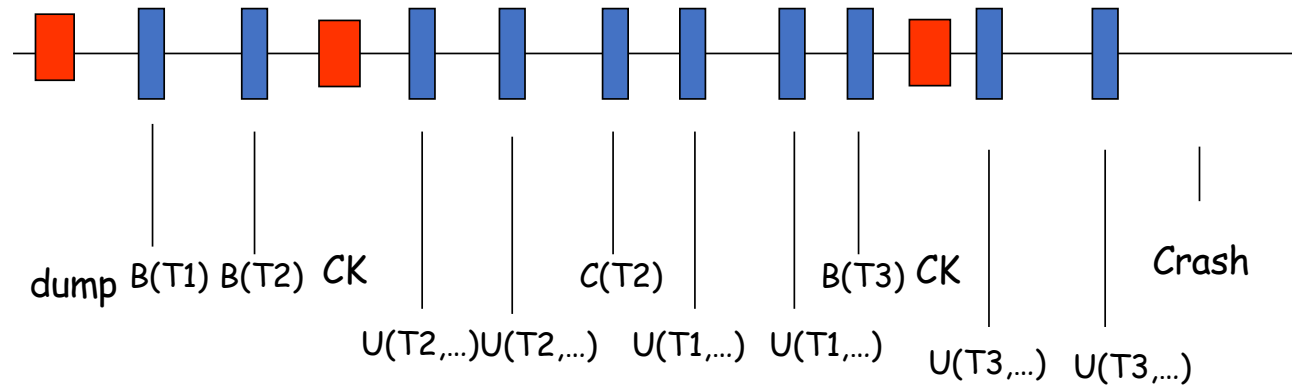
Struttura del log



Dump

- Copia completa ("di riserva", backup) della base di dati
 - Solitamente prodotta mentre il sistema non è operativo
 - Salvato in memoria stabile, tipicamente un nastro
 - Un record di `dump` nel log indica il momento in cui il log è stato effettuato (e dettagli pratici, file, dispositivo, ...)

Struttura del log



Esecuzione delle transazioni e scritture del log

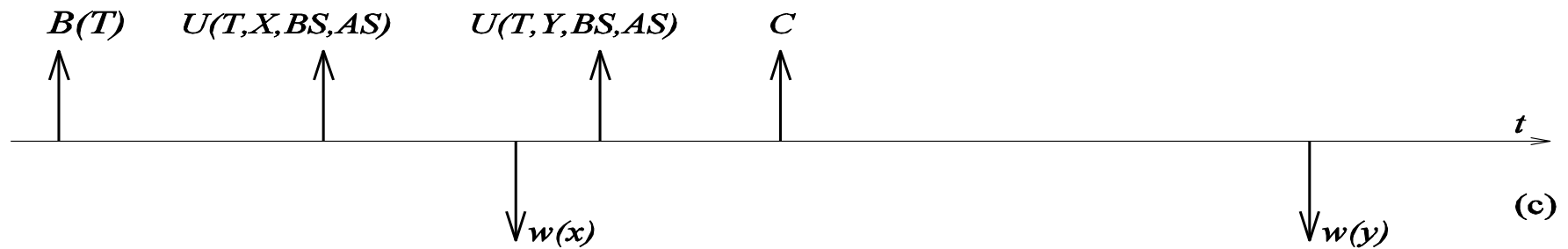
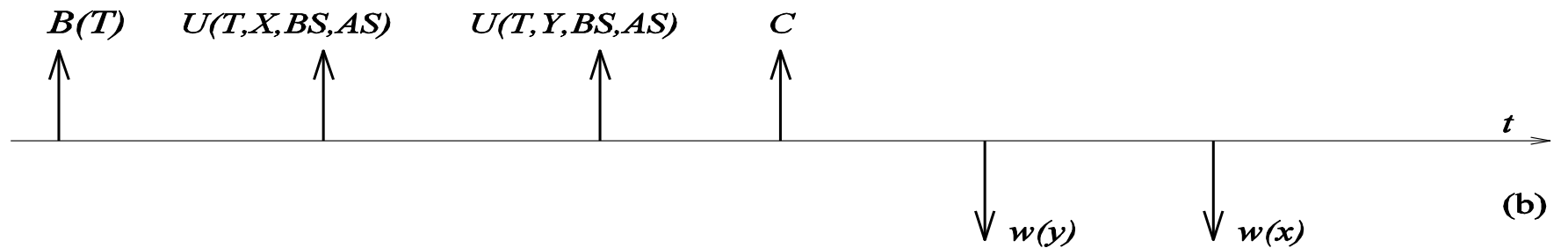
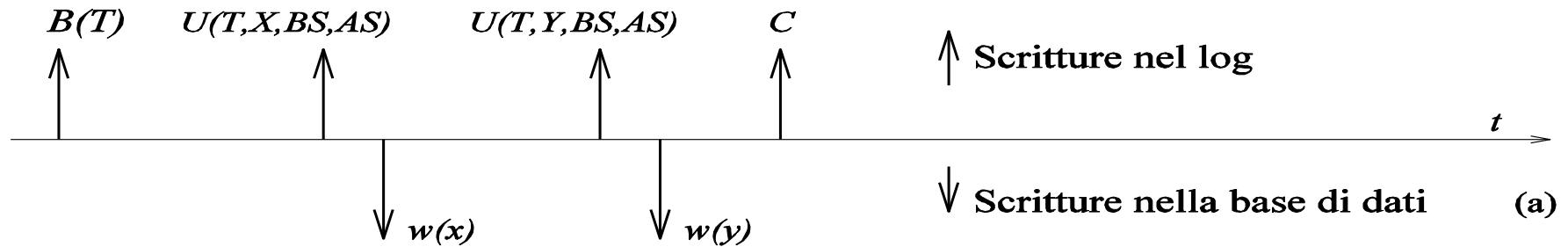
Esecuzione delle transazioni: Esito di una transazione

- L'esito di una transazione è determinato irrevocabilmente quando viene scritto il record di **commit** nel log (memoria stabile).
- Questa scrittura avviene in modo sincrono, con una **force**
 - Un guasto prima di tale istante porta ad un UNDO di tutte le azioni, per ricostruire lo stato originario della base di dati
 - un guasto successivo non deve avere conseguenze: lo stato finale della base di dati deve essere ricostruito, con REDO se necessario

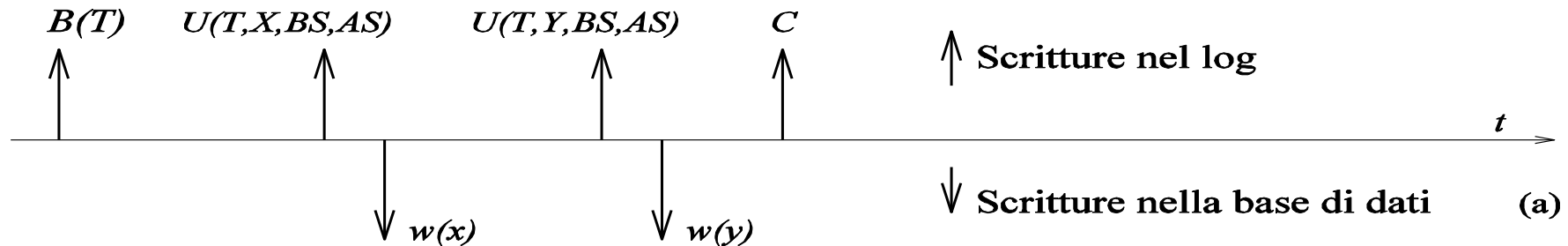
Esecuzione delle transazioni: scritture del log

- Il controllore dell'affidabilità deve garantire due regole che consentono di ripristinare la correttezza della basi di dati
- **Regola Write-Ahead-Log:** la parte before state dei record di un log deve essere scritta nel log (in memoria stabile) **prima di effettuare la corrispondente operazione sulla base di dati**
 - Questa regola ci permette di disfare (UNDO) le scritture già effettuate in memoria di massa che non hanno ancora fatto il commit
- **Commit-Precedenza:** La parte after state dei record di un log deve essere scritta nel log (in memoria stabile) **prima di effettuare il commit**
 - Questa regola consente di rifare le scritture (REDO) già decise da una transazioni che ha fatto commit, ma le cui pagine nel buffer non sono state scritte in memoria secondaria
- Quando scriviamo nella base di dati?
 - Varie alternative

Scrittura nel log e nella base di dati



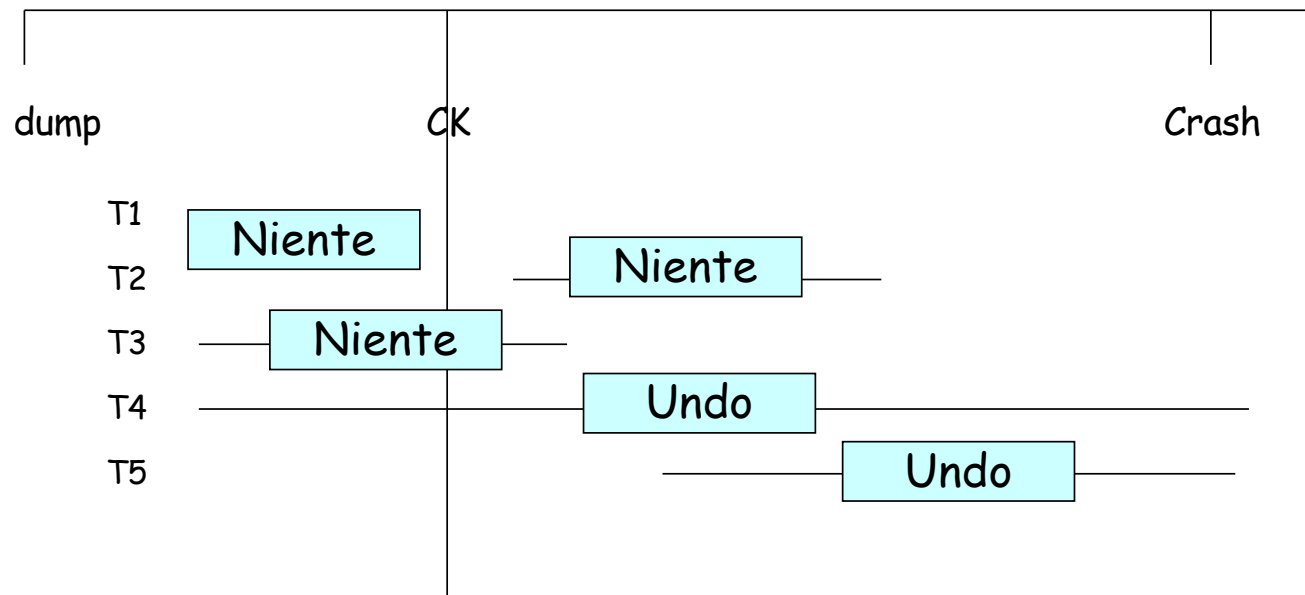
Scrittura nel log e nella base di dati: prima del commit



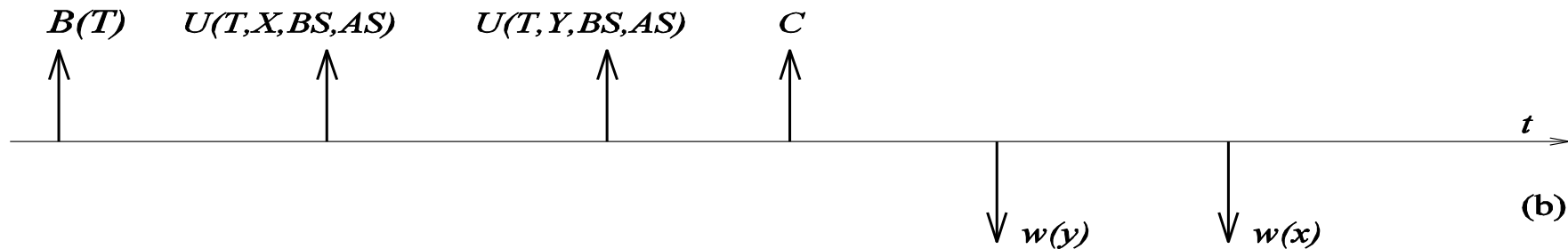
- Prima del commit:
 - il buffer scrive gli aggiornamenti sulla base di dati (i.e., $w(y)$) prima del commit (in modo sincrono (force) o asincrono (flush))

Scrittura nel log e nella base di dati: prima del commit

- Richiede Undo delle operazioni di transazioni uncommitted al momento del guasto
- Non richiede Redo



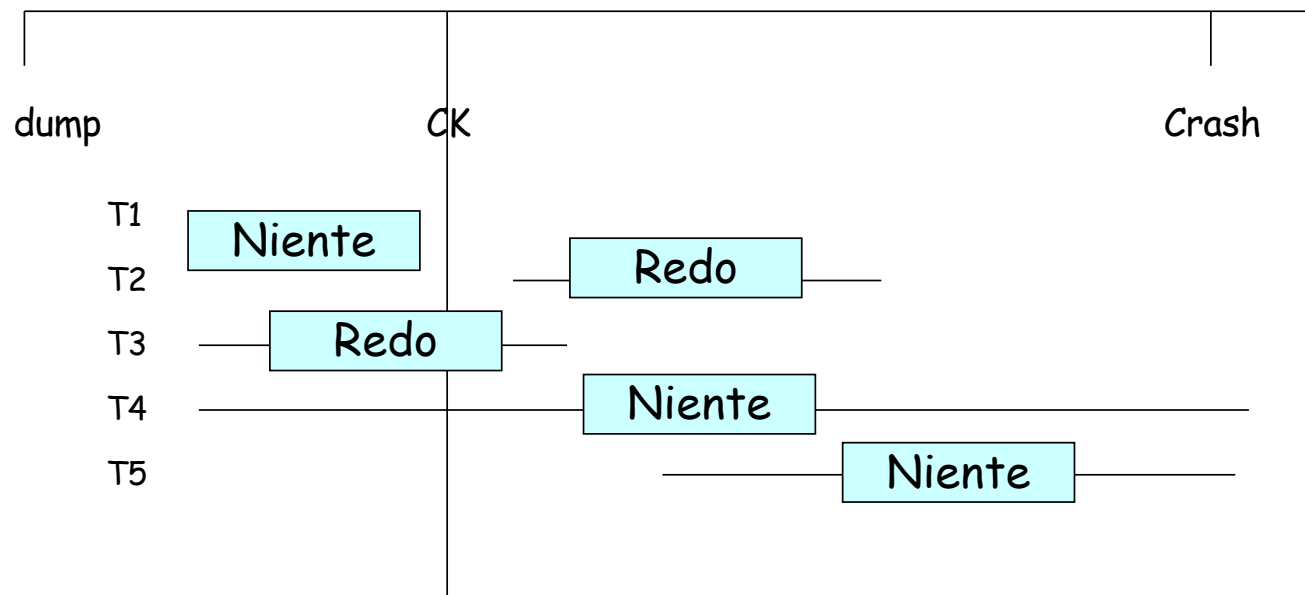
Scrittura nel log e nella base di dati: dopo il commit



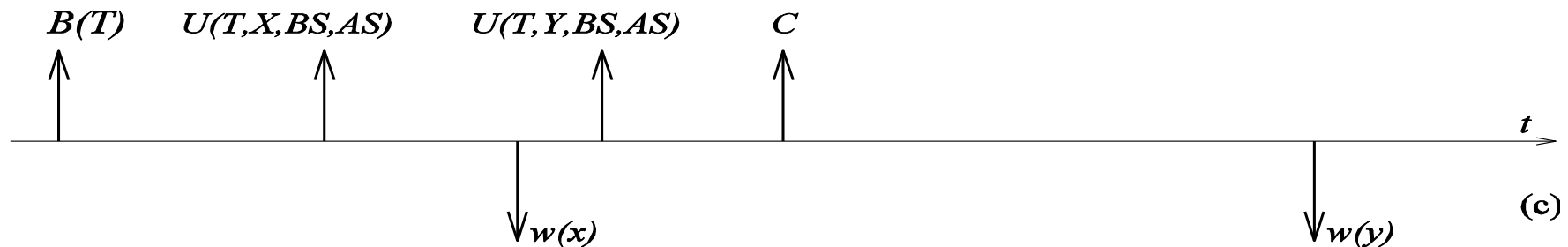
- Dopo il commit:
 - il buffer scrive gli aggiornamenti sulla base di dati (i.e., $w(y)$) dopo il commit (in modo sincrono, force).

Scrittura nel log e nella base di dati: dopo il commit

- Rende superflua la procedura di Undo.
- Richiede Redo



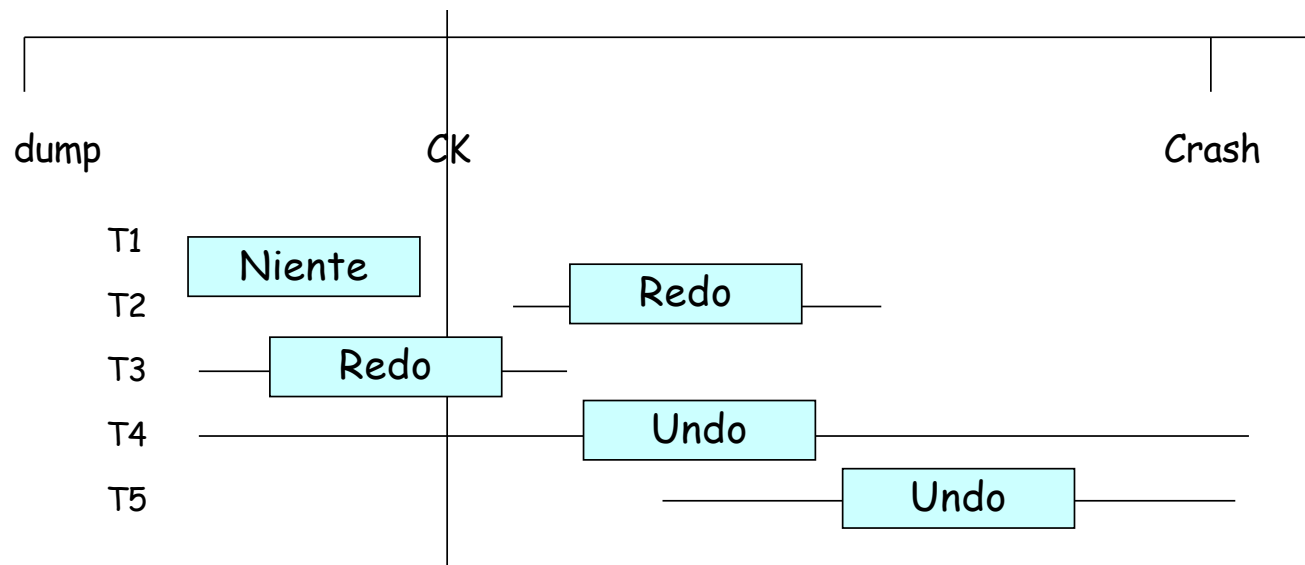
Scrittura nel log e nella base di dati: mista



- Mista (più comunemente usato):
 - il buffer scrive gli aggiornamenti sulla base di dati (i.e., $w(y)$) in qualunque momento rispetto alla scrittura del record di commit sul log.
 - Permette al buffer manager di ottimizzare le operazioni di flush

Scrittura nel log e nella base di dati: mista

- Richiede sia Undo che Redo

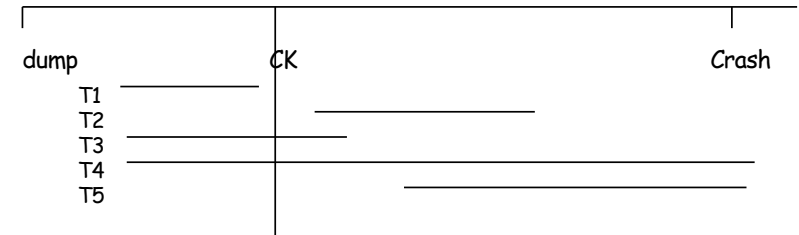


Gestione ripristino: tipo di guasti

- **Guasti di sistema:** errori di programma, crash di sistema, caduta di tensione
 - si perde la memoria centrale
 - non si perde la memoria secondaria**warm restart, ripresa a caldo**
- **Guasti di dispositivo:** sui dispositivi di memoria secondaria
 - si perde anche la memoria secondaria
 - non si perde la memoria stabile (e quindi il log)**cold restart, ripresa a freddo**

Ripresa a caldo

Quattro fasi:



1) Si accede all'ultimo blocco del log e si ripercorre all'indietro fino al più recente record di checkpoint CK

2) Si inizializzano due insiemi:

- *UNDO_SET*, insieme delle transazioni da disfare, inizializzato con le transazioni attive al checkpoint CK
- *REDO_SET*, inizializzato vuoto

Si ripercorre il log in avanti:

- per ogni commit, si toglie la corrispondente transazione dall'insieme UNDO_SET e lo si inserisce nell'insieme REDO_SET
- Per ogni begin transaction, si inserisce il corrispondente identificativo nell'insieme UNDO_SET

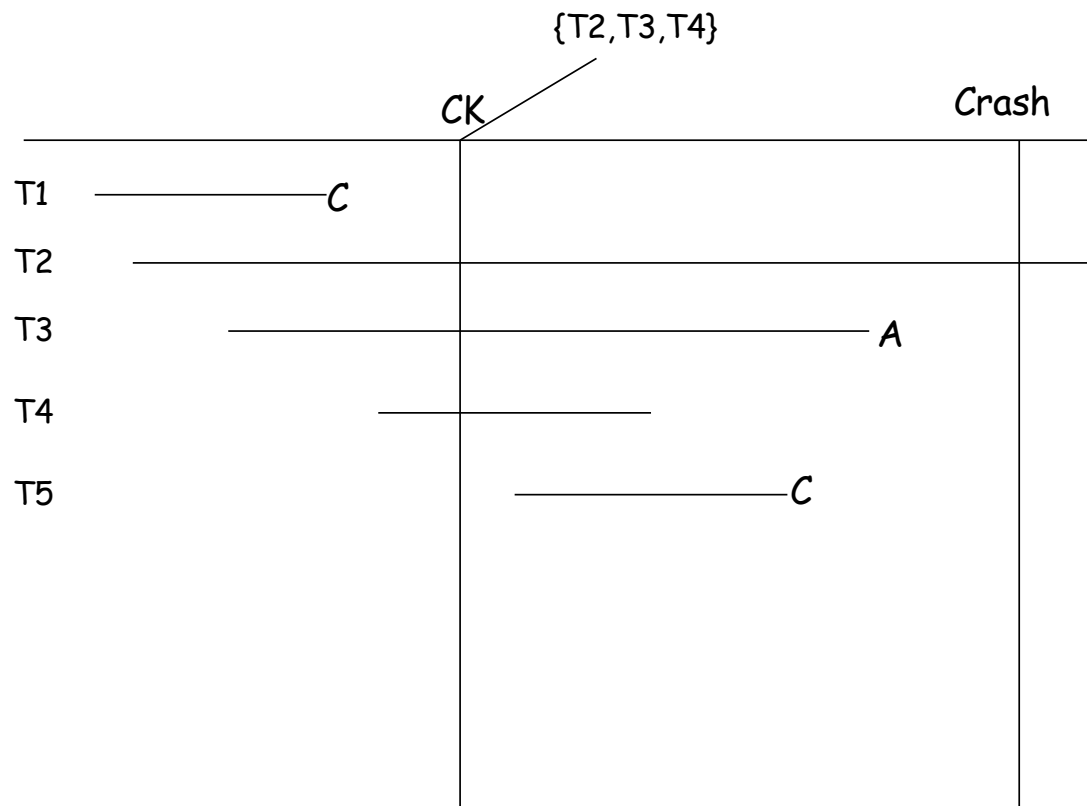
3) Si ripercorre il log all'indietro annullando ogni operazioni di transazione contenuta nell'insieme UNDO_SET

NB: una transazione in UNDO_SET potrebbe avere operazioni anche oltre al punto di checkpoint CK

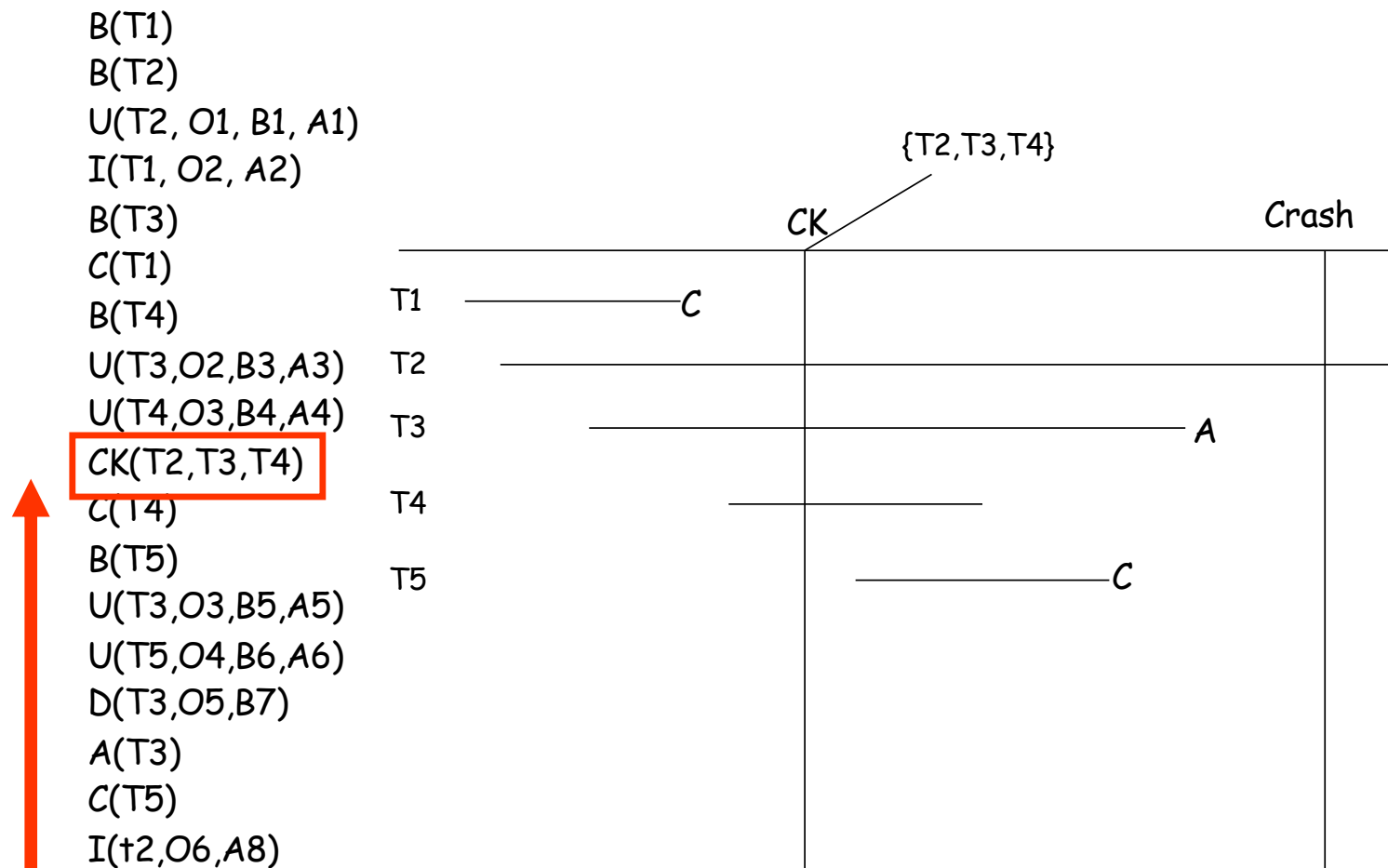
4) Si ripercorre il log in avanti, rifacendo tutte le azioni delle transazioni in *REDO_SET*

Esempio di warm restart

B(T1)
 B(T2)
 U(T2, O1, B1, A1)
 I(T1, O2, A2)
 B(T3)
 C(T1)
 B(T4)
 U(T3, O2, B3, A3)
 U(T4, O3, B4, A4)
 CK(T2, T3, T4)
 C(T4)
 B(T5)
 U(T3, O3, B5, A5)
 U(T5, O4, B6, A6)
 D(T3, O5, B7)
 A(T3)
 C(T5)
 I(T2, O6, A8)



1. Ricerca dell'ultimo checkpoint



2. Costruzione degli insiemi UNDO_SET e REDO_SET


	B(T1)	0. $UNDO_SET = \{T2, T3, T4\}$. $REDO_SET = \{\}$	
	B(T2)		
	U(T2, O1, B1, A1)	1. $C(T4) \rightarrow UNDO_SET = \{T2, T3\}$. $REDO_SET = \{T4\}$	
	I(T1, O2, A2)		
	B(T3)	2. $B(T5) \rightarrow UNDO_SET = \{T2, T3, T5\}$. $REDO_SET = \{T4\}$	Setup
	C(T1)	3. $C(T5) \rightarrow UNDO_SET = \{T2, T3\}$. $REDO_SET = \{T4, T5\}$	
	B(T4)		
	U(T3, O2, B3, A3)		
	U(T4, O3, B4, A4)		
	CK(T2, T3, T4)		
1.	C(T4)		
2.	B(T5)		
	U(T3, O3, B5, A5)		
	U(T5, O4, B6, A6)		
	D(T3, O5, B7)		
	A(T3)		
3.	C(T5)		
	I(T2, O6, A8)		

3. Fase UNDO



	B(T1)	0. UNDO_SET = {T2,T3,T4}. REDO_SET = {}	
	B(T2)		
8.	U(T2, O1, B1, A1)	1. C(T4) → UNDO_SET = {T2, T3}. REDO_SET = {T4}	
	I(T1, O2, A2)	2. B(T5) → UNDO_SET = {T2,T3,T5}. REDO_SET = {T4}	Setup
	B(T3)	3. C(T5) → UNDO_SET = {T2,T3}. REDO_SET = {T4, T5}	
	C(T1)		
	B(T4)		
7.	U(T3,O2,B3,A3)	4. D(O6)	
	U(T4,O3,B4,A4)	5. O5 =B7	
	CK(T2,T3,T4)		
1.	C(T4)	6. O3 = B5	Undo
2.	B(T5)		
6.	U(T3,O3,B5,A5)	7. O2 =B3	
	U(T5,O4,B6,A6)	8. O1=B1	
5.	D(T3,O5,B7)		
	A(T3)		
3.	C(T5)		
4.	I(T2,O6,A8)		

4. Fase REDO



	B(T1)	0. UNDO_SET = {T2,T3,T4}. REDO_SET = {}	
	B(T2)		
8.	U(T2, O1, B1, A1)	1. C(T4) → UNDO_SET = {T2, T3}. REDO_SET = {T4}	
	I(T1, O2, A2)	2. B(T5) → UNDO_SET = {T2,T3,T5}. REDO_SET = {T4}	Setup
	B(T3)		
	C(T1)	3. C(T5) → UNDO_SET = {T2,T3}. REDO_SET = {T4, T5}	
	B(T4)		
7.	U(T3,O2,B3,A3)	4. D(O6)	
9.	U(T4,O3,B4,A4)	5. O5 = B7	
	CK(T2,T3,T4)		
1.	C(T4)	6. O3 = B5	Undo
2.	B(T5)		
6.	U(T3,O3,B5,A5)	7. O2 = B3	
10.	U(T5,O4,B6,A6)	8. O1=B1	
5.	D(T3,O5,B7)		
	A(T3)	9. O3 = A4	
3.	C(T5)		Redo
4.	I(T2,O6,A8)	10. O4 = A6	

Ripresa a freddo

- 1) Si ripristinano i dati a partire dal backup più recente (dump)
- 2) Si ripercorre in avanti il log (dal dump), eseguono le operazioni registrate nel log fino all'istante del guasto
- 3) Si esegue una ripresa a caldo dall'istante del guasto