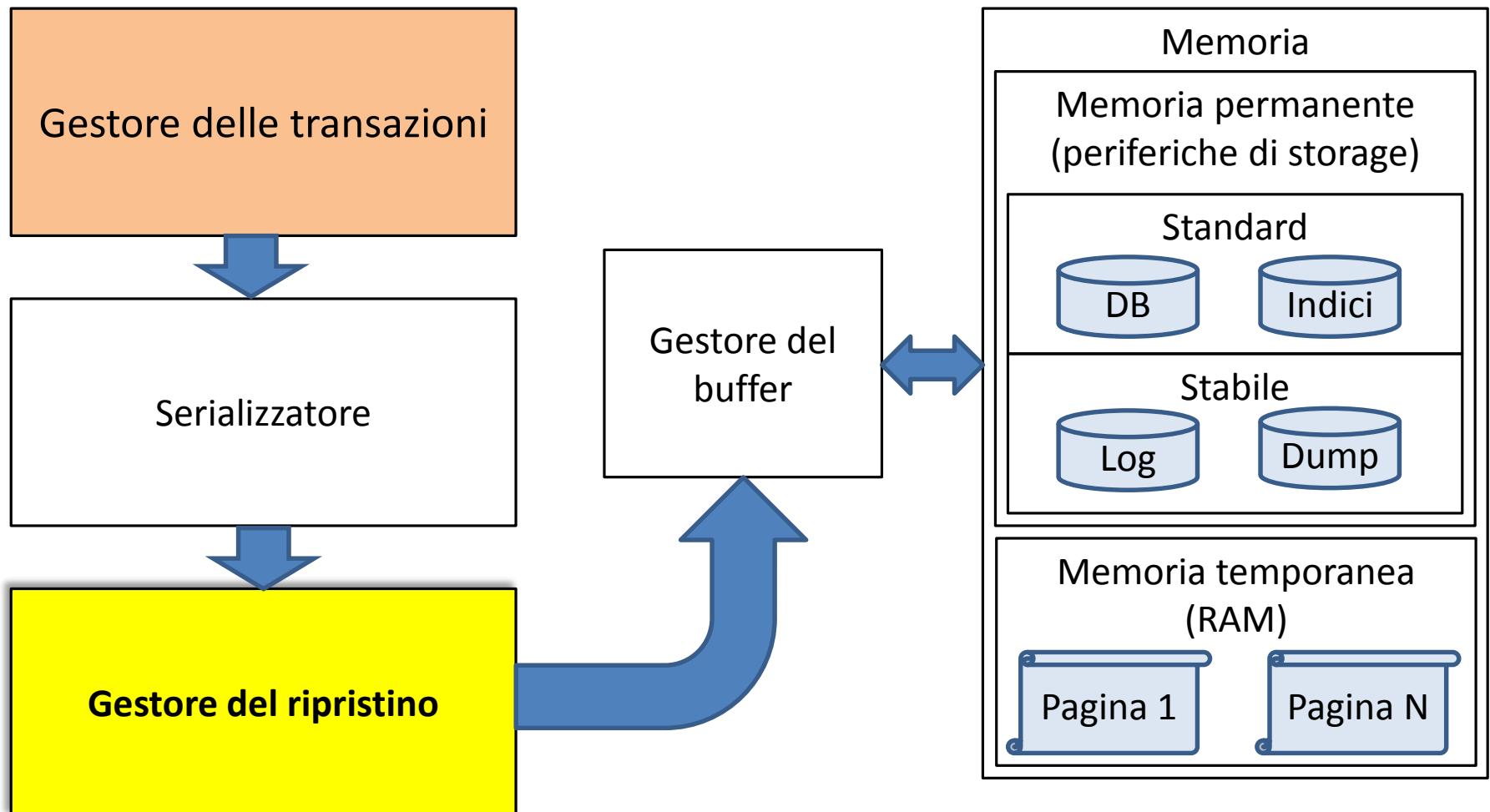


Basi di Dati
Architettura dei DBMS:
gestione del ripristino

Corso B

Architettura dei DBMS



Gestore del ripristino

Il gestore del ripristino affronta il problema dell'affidabilità

Il sistema deve garantire i dati a fronte di possibili guasti e anomalie:

- Guasti hardware dell'unità centrale
- Crash di sistemi
- Errori di programma
- Errori locali e condizioni di eccezione
- Rollback di transazioni forzate dal gestore della transazione e dal serializzatore
- Guasti delle periferiche di storage (*)
- Eventi catastrofici (**)

Gestore del ripristino

- Il gestore del ripristino è l'unità operativa che garantisce atomicità e consistenza
- Il gestore del ripristino gestisce l'affidabilità a fronte di errori tranne in caso di:
 - (*) guasti delle periferiche
 - (**) eventi catastrofici

Memoria stabile

- Abbiamo distinto la memoria delle periferiche di storage in **memoria standard** (dove si collocano DB e indici) e memoria stabile
- La memoria stabile fa da supporto al gestore del ripristino memorizzando alcune informazioni supplementari a quelle contenute alla base dati stessa
- Questi record supplementari sono chiamati **log**, registri o giornali (file strettamente sequenziali, aggiornati attraverso scritture **append**)

Memoria stabile

- C'è la necessità di una robustezza maggiore per la memoria stabile rispetto alla memoria standard
- Il concetto di memoria stabile è però teorico, in quanto non esiste memoria esente da guasto
- Si può realizzare concretamente la memoria stabile approssimando la proprietà di robustezza con la metodica della duplicazione
- Il file di log è duplicato più volte, anche in luoghi geograficamente lontani
- Per rendere efficiente la gestione della memoria stabile, le informazioni da memorizzare nei log devono essere di dimensioni contenute

Modalità di ripristino

- Il gestore del ripristino è basato su quattro approcci/algoritmi diversi
- Gli algoritmi nascono da un'analisi empirica della realtà e della tecnologia a disposizione
- Gli approcci dipendono dalle scelte progettuali legate alla modalità di gestione delle transazioni

Transazioni attive	Modifiche differite (no steal)	Modifiche <i>potenzialmente</i> immediate (steal)
Transazioni terminate	Dati lasciati nel buffer (no flush)	Dati trasferiti su periferica (flush)

Politiche di gestione del buffer

Queste modalità di gestione delle transazioni danno esito a **politiche di gestione del buffer**

- **Modifiche differite:** si sceglie come architettura generale del DBMS di registrare i dati modificati nell'area persistente in modo differito, solo dopo il **commit** della transazione (politica **no steal**)
- **Modifiche potenzialmente immediate:** si permette al buffer di sostituire pagine in memoria rendendo persistenti le modifiche della pagina rimpiazzata anche prima del commit/rollback (politiche **steal**)

Modalità di ripristino

- Il gestore del ripristino è basato su quattro approcci/algoritmi diversi
- Gli algoritmi nascono da un'analisi empirica della realtà e della tecnologia a disposizione
- Gli approcci dipendono dalle scelte progettuali legate alla modalità di gestione delle transazioni

Transazioni attive	Modifiche differite (no steal)	Modifiche <i>potenzialmente</i> immediate (steal)
Transazioni terminate	Dati lasciati nel buffer (no flush)	Dati trasferiti su periferica (flush)

Politiche di gestione del buffer

Quando una transazione, dopo aver acquisito e modificato una pagina nel buffer, raggiunge il commit, la pagina dovrebbe essere trasferita nella base di dati per rendere persistenti le modifiche

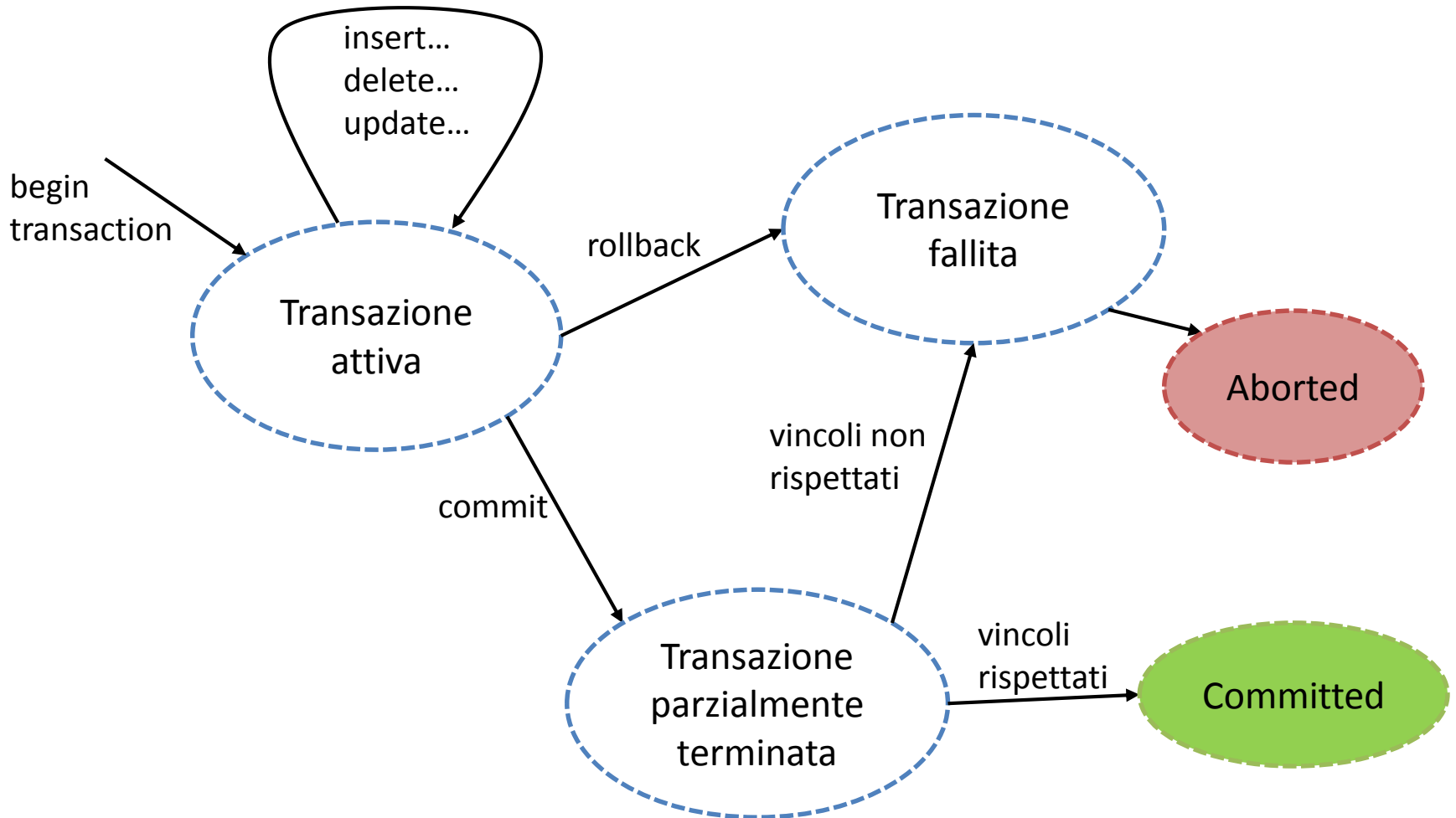
- **Dati lasciati nel buffer:** alcune architetture DBMS scelgono di non trasferire subito le pagine su periferica al fine di migliorare l'efficienza generale del sistema
In questo modo si dà la possibilità alle transazioni successive di trovare i dati già nel buffer (politiche **no flush**, diffuse nelle basi dati distribuite)
- **Dati trasferiti su periferica:** la pagina modificata dalla transazione che ha raggiunto il commit viene trasferita immediatamente su periferica (politica **flush**)

Modalità di gestione del ripristino

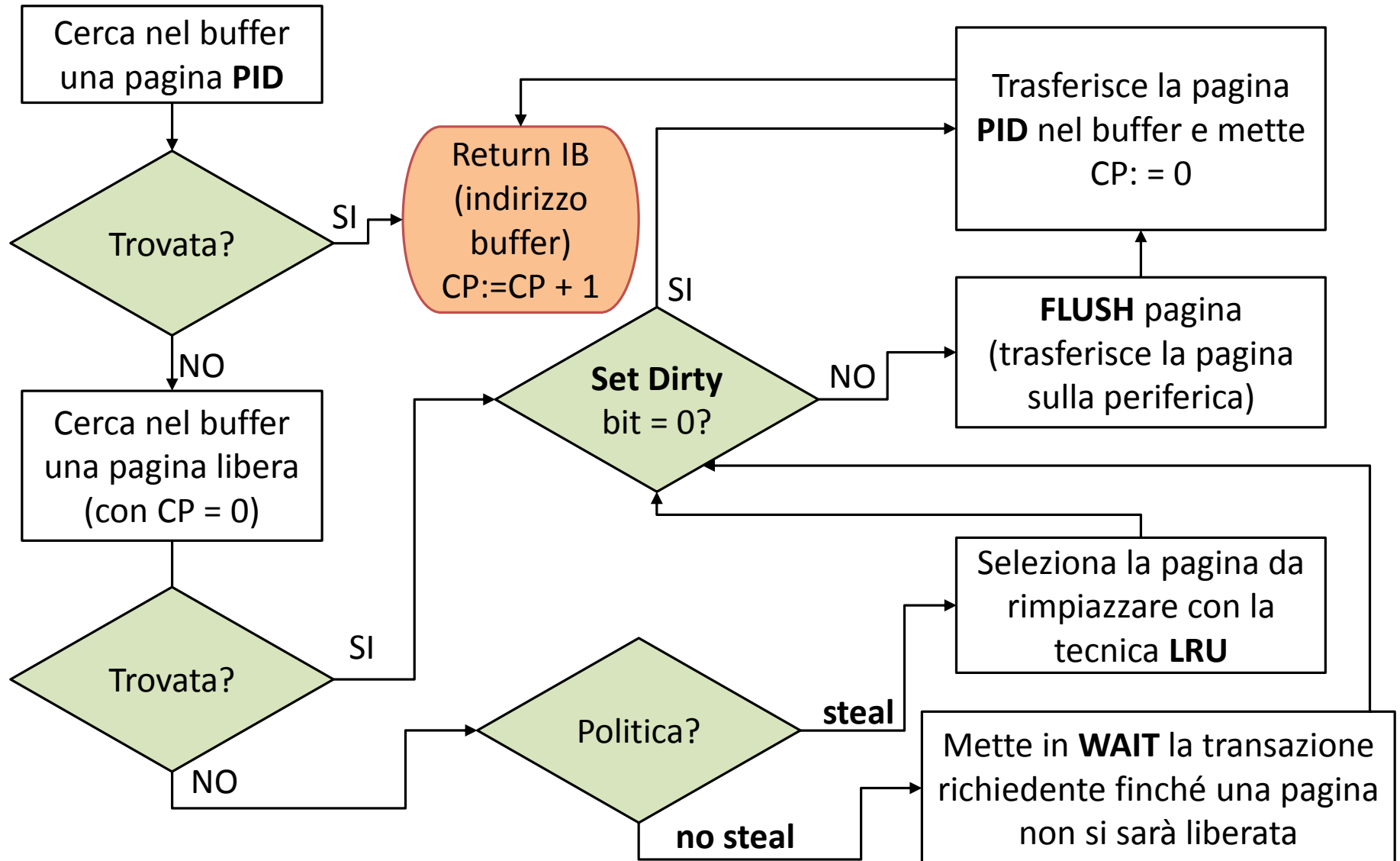
Le modalità di gestione del ripristino sono date dalle quattro combinazioni delle politiche di gestione del buffer:

1. steal/no flush
2. no steal/no flush
3. steal/flush
4. no steal/flush

Ciclo di vita di una transazione



Algoritmo FIX(PID)



Caso 1: steal/no flush

- Politiche **steal**: persistenza dei dati modificati da transazioni di cui non si conosce ancora l'esito
- Politiche **no flush**: le pagine di una transazione terminata in commit sono lasciate nel buffer

C'è bisogno di un formato di giornale (log) per recuperare tutte le situazioni:

- transazione in rollback con alcune modifiche rese persistenti dalla politica steal
- crash del sistema con transazione in commit ma modifiche non ancora persistenti

Caso 1: steal/no flush

Il file di log è un file di tipo sequenziale con record registrati durante l'esecuzione delle transazioni

- **<T_i, start>**: inizio della transazione T_i
- **<T_i, X, BS(X), AS(X)>**: ogni volta che la transazione modifica un dato, si registra l'identificativo della transazione, l'oggetto X (per esempio, la tupla) modificato, il **Before State** di X e l'**After State** di X (stato di X prima e dopo la modifica)

Le quadruple così generate entrano nel file di log nell'ordine con cui vengono prodotte le modifiche nel corso dell'elaborazione, ordine che riflette la storia delle azioni delle varie transazioni

Il **log** è quindi una **storia serializzabile corretta** di modifiche della base di dati

Caso 1: steal/no flush

Possiamo immaginare due situazioni:

- Transazione T_i fallita (rollback deciso dalla transazione stessa, dal serializzatore...): si avrà quindi, nella storia, un'azione detta **UNDO(T_i)** che disfa tutte le azioni precedentemente compiute dalla transazione T_i , e nel log verrà memorizzato il record **< T_i , abort>** seguito dalla direttiva **FORCE LOG**
 - La gestione del log avviene come per tutti gli altri dati attraverso il buffer
 - La **FORCE LOG** richiede al gestore del ripristino un'attività di forzatura della scrittura in memoria stabile di **tutte le pagine del log** presenti nel buffer, al fine di garantire la coerenza della sequenza storica

Caso 1: steal/no flush

Possiamo immaginare due situazioni:

- Transazione T_i parzialmente terminata (commit): il gestore delle transazione esegue la **verifica dei vincoli di integrità**
 - Se un vincolo non è soddisfatto, la transazione viene abortita e si eseguono le operazioni previste per il log
 - Se i vincoli sono soddisfatti, dato che siamo in politica no flush, non dobbiamo rendere persistenti le pagine modificate, ma per evitare la perdita delle modifiche, si crea il record $\langle T_i, \text{commit} \rangle$ e si esegue un **FORCE LOG** (trasferimento in memoria stabile di tutte i record del log presenti nel buffer, fino al commit incluso)

Ripristino

Quando cade il sistema e si riprende l'attività, alcune transazioni $LA = \{T_i \dots T_h\}$ possono trovarsi in uno stadio di elaborazione non terminato, altre transazioni $LC = \{T_j \dots T_k\}$ invece possono risultare terminate

In prima approssimazione, il ripristino in seguito ad un crash, avviene con due operazioni:

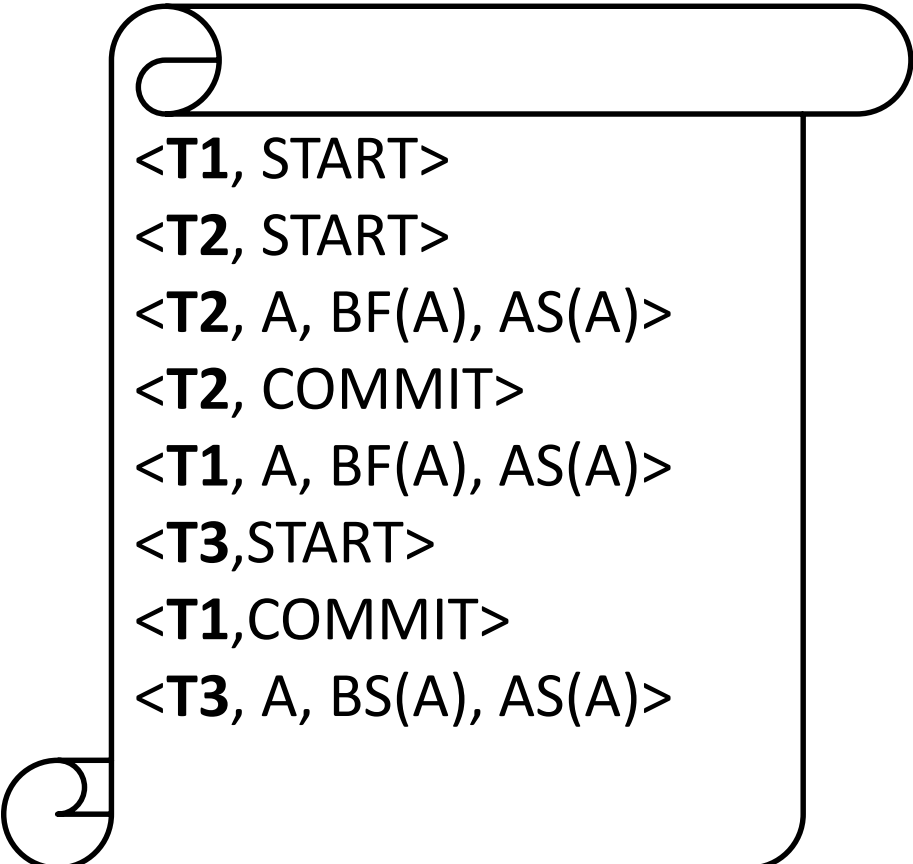
- **UNDO(LA): disfacimento** delle transazioni non terminate per garantire la proprietà di atomicità
- **REDO(LC): rifacimento** delle transazioni in commit, a causa delle modifiche potenzialmente perse per via della politica **no flush**

Esempio di file di log

File di log presente in memoria stabile

Il sistema va in crash dopo l'ultima operazione registrata nel log

Vediamo come ricostruire lo stato corretto del DB



<T1, START>
<T2, START>
<T2, A, BF(A), AS(A)>
<T2, COMMIT>
<T1, A, BF(A), AS(A)>
<T3, START>
<T1, COMMIT>
<T3, A, BS(A), AS(A)>

Esempio

Ipotizziamo che T2 sia la prima transazione che modifica l'oggetto A

La transazione T2 va in **commit**, ma se c'è commit nel file di log significa che le azioni di quella transazione le ritrovo interamente nel file di log

Quando T1 acquisisce il **lock** su A, sicuramente il suo BS(A) sarà uguale all'AS(A) di T2

La transazione T3 entra nel sistema, poi la T1 va in **commit** e la T3 modifica l'oggetto A, ma non riesce ad andare in **commit** a causa del crash

REDO(T_1, \dots, T_k)

Si esplora il file di log in **avanti**

Per ogni $\langle T_i, X, BS(X), \mathbf{AS(X)} \rangle \wedge T_i \in \{ T_1, \dots, T_k \}$

/* rendere persistente AS(X) cioè: */

Leggere la pagina che contiene X

Modificare nel buffer la pagina

FORCE della pagina

Esempio

Rifare significa rendere persistenti gli After State

Dato che non sappiamo se le modifiche fatte nel buffer sono state trasferite nella memoria permanente, occorre fare il REDO delle transazioni andate in commit, nel nostro esempio:

REDO(T2,T1)

<T1, START>

<T2, START>

<T2, A, BF(A), AS(A)>

*<T2, COMMIT>

<T1, A, BF(A), AS(A)>

<T3, START>

*<T1, COMMIT>

<T3, A, BS(A), AS(A)>

Lettura in avanti

- Bisogna esplorare il log in avanti perché il rifacimento deve lasciare sulla base dati l'ultima modifica
- Il log dice che l'esecuzione seriale della transazione è stata T2 e poi T1
- Il REDO rispetta esattamente quest'ordine se il file di log viene letto in avanti

UNDO(T_1, \dots, T_k)

Si esplora il file di log a ritroso

Per ogni $\langle T_i, X, \mathbf{BS}(X), AS(X) \rangle \wedge T_i \in \{ T_1, \dots, T_k \}$

/* ripristinare $BS(X)$ cioè: */

Leggere la pagina che contiene X

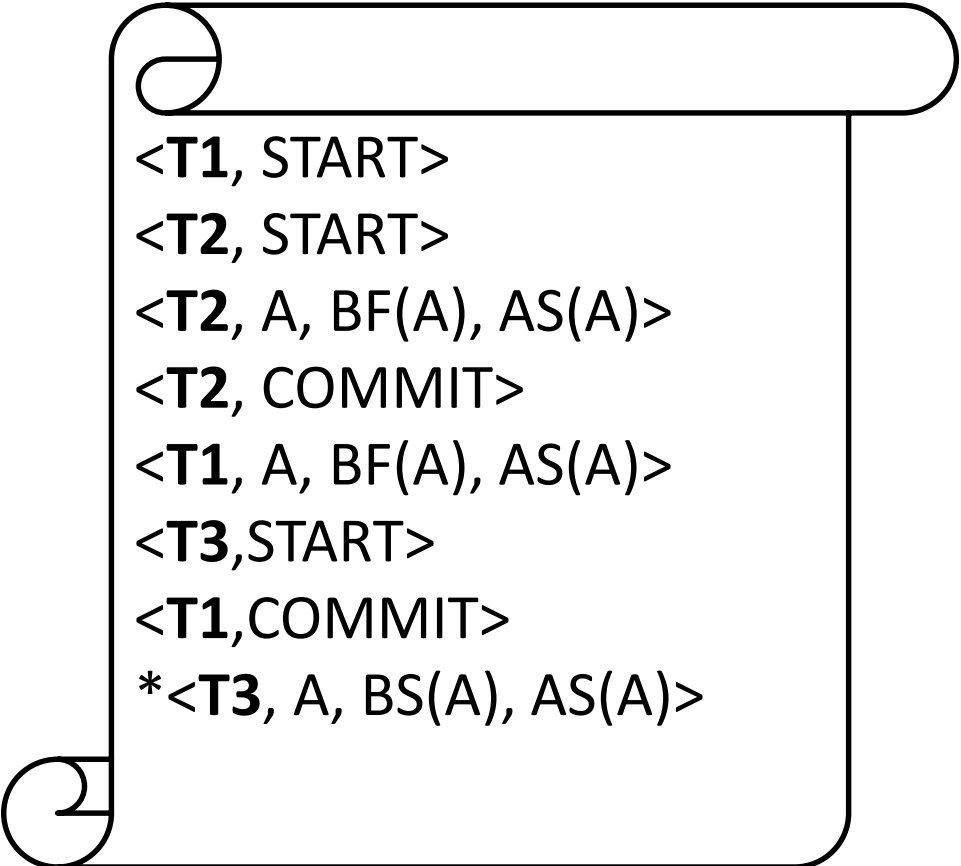
Modificare nel buffer la pagina

FORCE della pagina

Lettura a ritroso

Leggendo a ritroso il file di log si può ricostruire sulla periferica di storage lo stato iniziale della base di dati precedente alle modifiche delle transazioni non andate a buon fine, nel nostro esempio:

UNDO(T3)



<T1, START>
<T2, START>
<T2, A, BF(A), AS(A)>
<T2, COMMIT>
<T1, A, BF(A), AS(A)>
<T3, START>
<T1, COMMIT>
*<T3, A, BS(A), AS(A)>

Algoritmo di ripristino 1 (steal/no flush)

Quando avviene un crash il sistema cade e poi dovrà ripartire a freddo, senza nessuna transazione in corso

A questo punto avviene il ripristino che prevede i seguenti passi:

1. **LA**: = lista delle transazioni non andate a buon fine
2. **LC**: = lista delle transazioni che hanno raggiunto il commit
3. **UNDO(LA)**
4. **REDO(LC)** (sempre dopo la UNDO)

Il protocollo a due fasi garantisce che la sequenza di azioni sia equivalente ad una storia seriale

Problema

- L'utilizzo delle versioni date di REDO e UNDO creano problemi di ridondanza e quindi di efficienza
- La REDO considera il log dall'inizio, apporta tutte le modifiche, ma ciò che resta nella base dati, per ogni oggetto, è l'ultima modifica
- La UNDO considera il log dalla fine, apporta tutte le modifiche, ma ciò che resta nella base dati, per ogni oggetto, è lo stato prima della prima modifica

REDO(T_1, \dots, T_k) ottimizzato

LER: = \emptyset /* LER: Lista degli Elementi Rifatti */

Si esplora il file di log a **ritroso** /* **attenzione** */

Per ogni $\langle T_i, X, BS(X), \mathbf{AS(X)} \rangle \wedge T_i \in \{ T_1, \dots, T_k \}$

Se $X \notin \text{LER}$

/* rendere persistente $AS(X)$ cioè: */

Leggere la pagina che contiene X

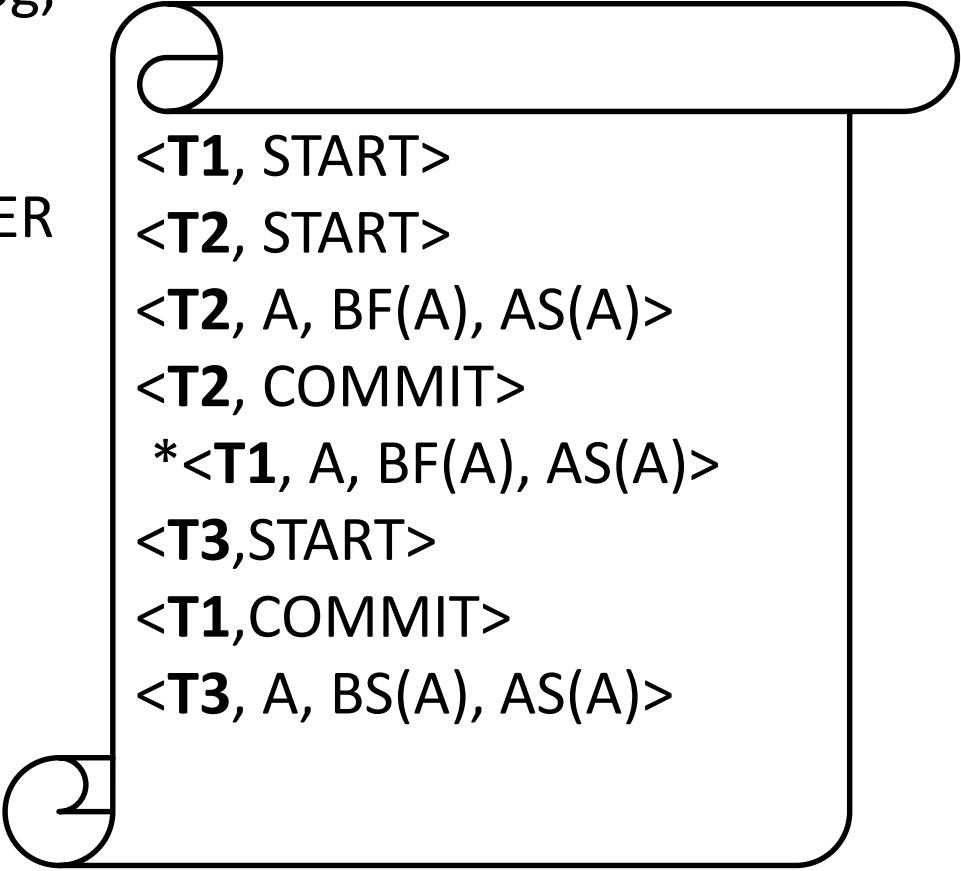
Modificare nel buffer la pagina

FORCE della pagina

LER: = LER $\cup \{X\}$

Lettura a ritroso

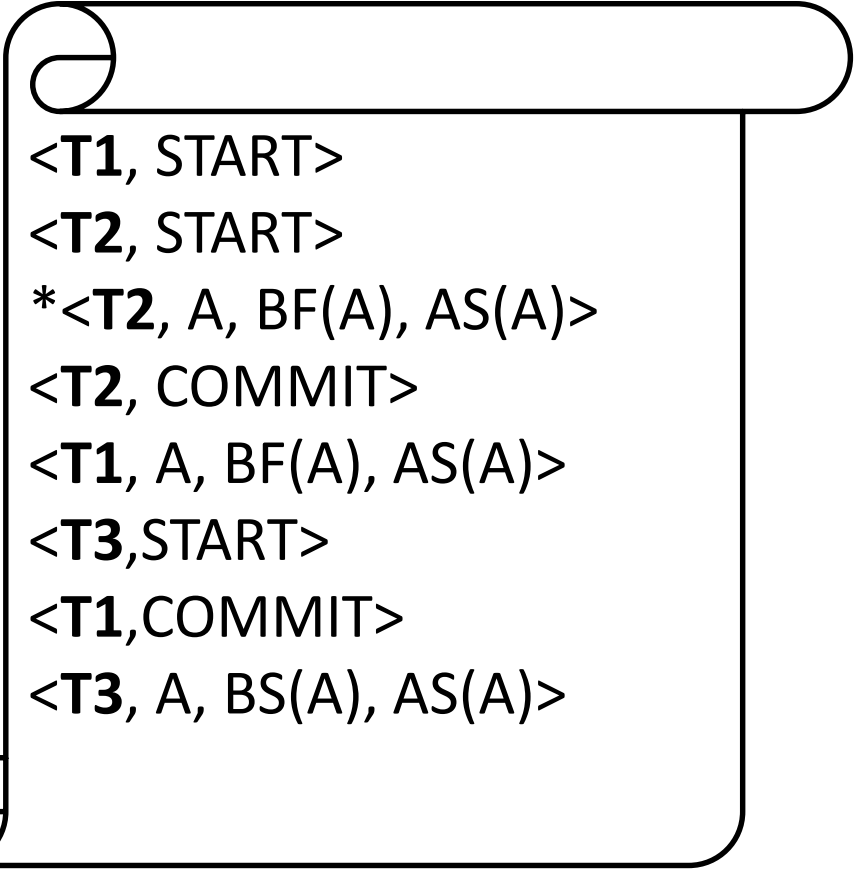
- Leggendo a ritroso il file di log, il REDO(T1,T2) trova la modifica effettuata da T1 sull'oggetto A che non è in LER (all'inizio LER è vuota) e la rende persistente



<**T1**, START>
<**T2**, START>
<**T2**, A, BF(A), AS(A)>
<**T2**, COMMIT>
*<**T1**, A, BF(A), AS(A)>
<**T3**, START>
<**T1**, COMMIT>
<**T3**, A, BS(A), AS(A)>

Lettura a ritroso

- Leggendo a ritroso il file di log, il REDO(T1,T2) trova la modifica effettuata da T1 sull'oggetto A che non è in LER (all'inizio LER è vuota) e la rende persistente
- Poi trova la modifica di T2, che però avviene su A che ora è nella lista degli oggetti rifatti, quindi la ignora



<**T1**, START>
<**T2**, START>
*<**T2**, A, BF(A), AS(A)>
<**T2**, COMMIT>
<**T1**, A, BF(A), AS(A)>
<**T3**, START>
<**T1**, COMMIT>
<**T3**, A, BS(A), AS(A)>

UNDO(T1,...,Tk) ottimizzato

LER: = \emptyset /* LER: Lista degli Elementi Rifatti */

Si esplora il file di log **in avanti** /* **attenzione** */

Per ogni $\langle T_i, X, \mathbf{BS(X)}, AS(X) \rangle \wedge T_i \in \{ T_1, \dots, T_k \}$

Se $X \notin \text{LER}$

/* ripristinare BS(X) cioè: */

Leggere la pagina che contiene X

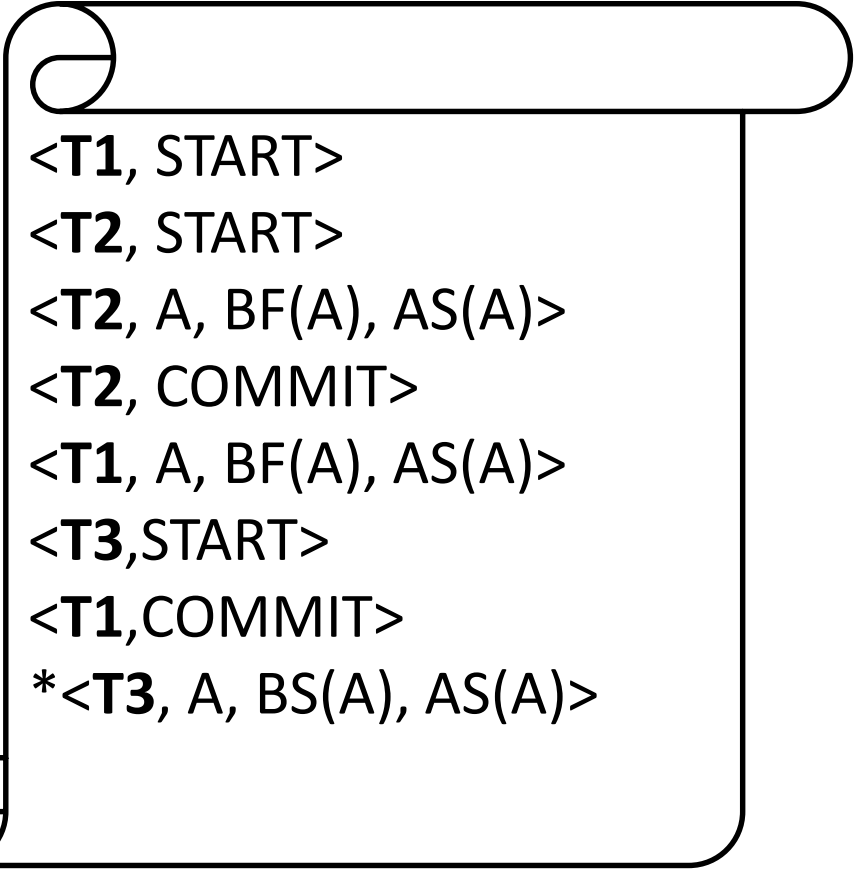
Modificare nel buffer la pagina

FORCE della pagina

LER: = LER $\cup \{X\}$

Lettura a ritroso

- Leggendo in avanti il file di log, la UNDO(T3) trova la modifica effettuata da T3 sull'oggetto A che non è in LER (all'inizio LER è vuota) e la rende persistente
- Se in seguito ci fosse un'altra modifica ad A da parte di una transazione non a buon fine, la modifica verrebbe ignorata (A è già nella LER)



<**T1**, START>
<**T2**, START>
<**T2**, A, BF(A), AS(A)>
<**T2**, COMMIT>
<**T1**, A, BF(A), AS(A)>
<**T3**, START>
<**T1**, COMMIT>
*<**T3**, A, BS(A), AS(A)>

Ripristino delle transazioni in abort

Nel ripristino non si parla mai delle transazioni in abort

Nel ripristino le transazioni in abort vengono ignorate

Infatti, se c'è un **abort**, come visto in precedenza l'UNDO è già stato fatto, ma l'UNDO agisce sulla periferica grazie alla chiamata alla primitiva FORCE

Politica steal

Ricordiamo che, quando il gestore del buffer adotta una politica steal, se una transazione richiede il caricamento di una pagina (**FIX PID**), il gestore del buffer può "rubare" una pagina da una transazione, renderla persistente e lasciare spazio alla nuova pagina

Questa politica richiede sempre due passi:

- **FORCE LOG**: anche il gestore del buffer, prima di copiare una pagina dati sulla periferica, deve prevenire eventuali fenomeni di crash
- **FORCE pagina**: trasferimento su periferica della pagina rubata

Ciclo operativo del sistema e log

- Nei sistemi informativi reali, c'è un momento in cui il file di log è vuoto, la base di dati si considera corretta, ed inizia un ciclo di vita dell'attività transazionale
- Il ciclo di vita dura circa 24 ore
- Quando l'attività è minima (intorno alla mezzanotte) si interrompe l'attività, si rendono persistenti tutte le modifiche delle transazioni committate, si disfano le altre e si azzera il file di log

Ripristino problematico

- Nei grossi sistemi informativi, in un ciclo di vita il file di log può contenere anche centinaia di migliaia di record
- Se c'è una crash di sistema nel pomeriggio, quando il file di log contiene già migliaia e migliaia di tuple, il ripristino diventa estremamente costoso
- Anche se i crash non sono frequenti, possono avvenire in qualsiasi momento, **anche in pieno orario di attività** e bisogna ridurre al minimo i tempi necessari al ripristino
- Il ripristino richiede il blocco dell'attività transazionale, comportando, ad esempio, il blocco di tutti gli sportelli del bancomat

Checkpoint

La soluzione al problema consiste nell'introduzione, nel file di log, di un passo detto checkpoint

<T1, START>

<T2, START>

<T2, A, BF(A), AS(A)>

<T2, COMMIT>

<T1, A, BF(A), AS(A)>

<T3, START>

...

----- checkpoint -----

...

<T1, COMMIT>

<T3, A, BS(A), AS(A)>

Checkpoint

Periodicamente avviene un processo di checkpoint che aggiunge un record al file di log

- Il checkpoint **sospende tutte** le transazioni
- Viene costruito il record di checkpoint contenente l'elenco delle transazioni che in quel momento sono attive col relativo puntatore alla posizione dello start nel file di log
- Si esegue un **FORCE LOG**
- Si esegue un **FORCE** delle pagine delle transazioni in commit
- Viene aggiunto un flag di **OK** nel record di checkpoint e si esegue un nuovo **FORCE LOG**
- Vengono riavviate le transazioni sospese

Record di checkpoint: esempio

- **T1** e **T3** sono transazioni iniziate ma non terminate, quindi vengono aggiunte alla lista di checkpoint con i relativi puntatori
- **T2** è iniziata e terminata, quindi non viene aggiunta al record
- Il record è reso persistente con una **FORCE LOG**
- Le modifiche di **T2** vengono rese persistenti con un **FORCE pagine**
- Si aggiunge il flag **OK**
- Si esegue un nuovo **FORCE LOG**

<T1, START>
<T2, START>
<T2, A, BF(A), AS(A)>
<T2, COMMIT>
<T1, A, BF(A), AS(A)>
<T3, START>

cp:	T1, p	T3, p	OK
-----	-------	-------	----

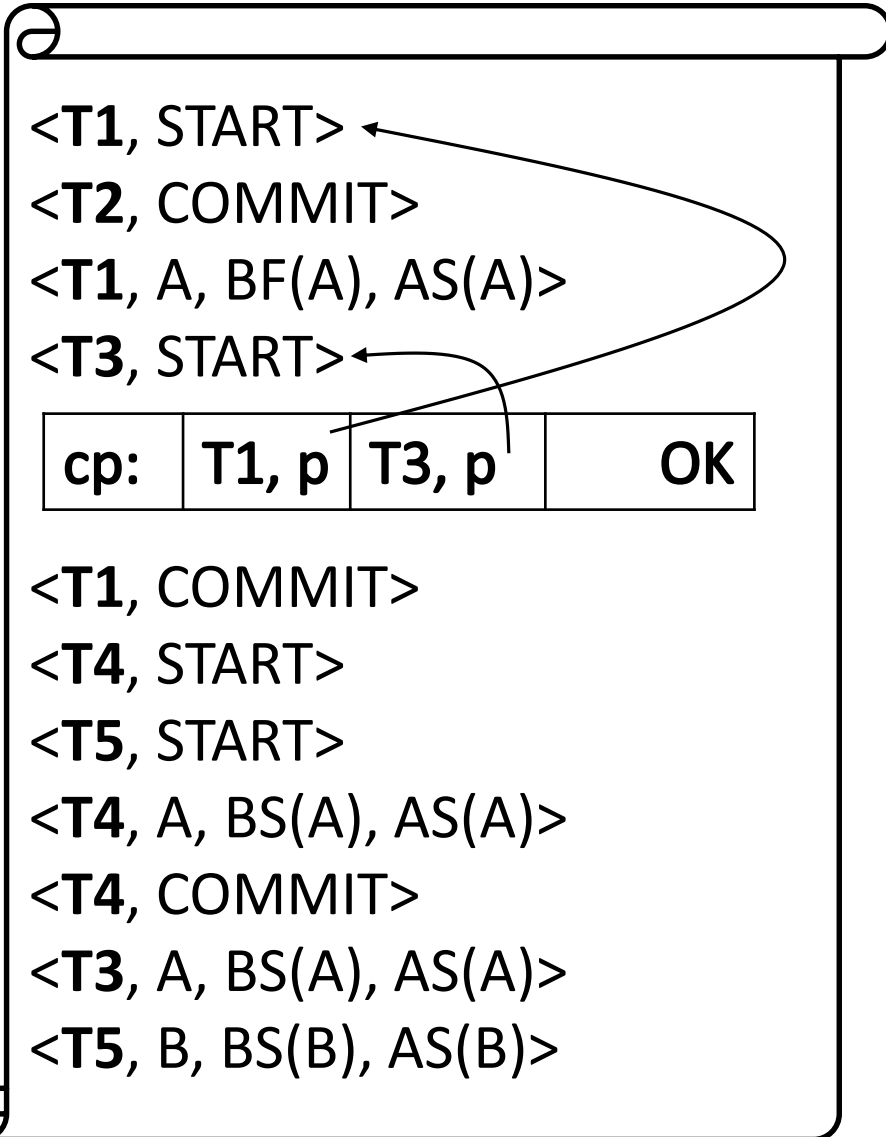
<T1, COMMIT>
<T3, A, BS(A), AS(A)>

Periodicità del checkpoint

Il tempo che intercorre tra un checkpoint ed un altro varia in base alle tecnologie, attualmente 10/15 minuti

Ripristino con checkpoint

- Il ripristino cerca l'ultimo checkpoint
- L'ultimo checkpoint dà l'elenco delle transazioni attive in quel momento (**T1** e **T3**)
- Le transazioni che hanno raggiunto il commit **prima** del checkpoint (**T2**) sono già sistemate



Ripristino con checkpoint

Ci possono essere:

- transazioni iniziate prima del checkpoint e committate dopo il checkpoint (**T1**)
- Transazioni iniziate dopo il checkpoint e committate prima del crash (**T4**)
- Transazioni iniziate prima del checkpoint e non terminate (**T3**)
- Transazioni iniziate dopo il checkpoint e non terminate (**T5**)

<**T1**, START>
<**T2**, COMMIT>
<**T1**, A, BF(A), AS(A)>
<**T3**, START>

cp:	T1 , p	T3 , p	OK
-----	---------------	---------------	----

<**T1**, COMMIT>
<**T4**, START>
<**T5**, START>
<**T4**, A, BS(A), AS(A)>
<**T4**, COMMIT>
<**T3**, A, BS(A), AS(A)>
<**T5**, B, BS(B), AS(B)>

Ripristino con checkpoint

- Il gestore del ripristino recupera la lista **LC** delle transazioni che hanno raggiunto il commit **dopo** l'ultimo checkpoint
- Il gestore del ripristino recupera la lista **LA** delle transazioni ancora **attive** durante il crash
- A questo punto avviene il ripristino che prevede sempre **prima l'UNDO(LA) e poi il REDO(LC)**

Ripristino con checkpoint

Lista delle transazioni terminate dopo il checkpoint:

LC = {T1, T4}

Lista delle transazioni attive dopo il checkpoint:

LA = {T3, T5}

Ripristino DB:

1. **UNDO(LA)**
2. **REDO(LC)**

<T1, START>
<T2, COMMIT>
<T1, A, BF(A), AS(A)>
<T3, START>

cp:	T1, p	T3, p	OK
-----	-------	-------	----

<T1, COMMIT>
<T4, START>
<T5, START>
<T4, A, BS(A), AS(A)>
<T4, COMMIT>
<T3, A, BS(A), AS(A)>
<T5, B, BS(B), AS(B)>

Caso 2: no steal/no flush

- Politica **no steal**: il gestore del buffer non può rubare pagine alle transazioni
- Politica **no flush**: dopo il commit, il gestore del buffer lascia le pagine nel buffer
- In questo caso, nel file di log, è sufficiente memorizzare l'**after state**
- Siamo in politica **no flush**: serve l'**AS** per il REDO delle transazioni che hanno raggiunto il **commit**
- Siamo in politica **no steal**: non serve il **BS** perché tutte le azioni fatte dalle transazioni non committate hanno modificato solo le pagine nel buffer (in memoria centrale)

Caso 2: no steal/no flush

Nel file di log troveremo

- **<T_i, start>**: inizio della transazione T_i
- **<T_i, X, AS(X)>**: ogni volta che la transazione modifica un dato, si registra l'identificativo della transazione, l'oggetto X (per esempio, la tupla) modificato e l'**After State** di X (stato di X dopo la modifica)

Al termine, possiamo immaginare due situazioni:

- Transazione T_i fallita (rollback deciso dalla transazione stessa, dal serializzatore...): ci sarà solo il **rilascio del buffer**, e nel log verrà memorizzato il record **<T_i, abort>** seguito dalla direttiva **FORCE LOG**

Caso 2: no steal/no flush

Al termine, possiamo immaginare due situazioni:

- Transazione T_i parzialmente terminata (commit): il gestore delle transazione esegue la **verifica dei vincoli di integrità**
 - Se un vincolo non è soddisfatto, la transazione viene abortita e si eseguono le operazioni previste per il log
 - Se i vincoli sono soddisfatti, dato che siamo in politica no flush, non dobbiamo rendere persistenti le pagine modificate, ma per evitare la perdita delle modifiche, si crea il record $\langle T_i, \text{commit} \rangle$ e si esegue un **FORCE LOG** (trasferimento in memoria stabile di tutte i record del log presenti nel buffer, fino al commit incluso)

Siamo nella stessa situazione dell'algoritmo 1

Algoritmo di ripristino 2 (no steal/no flush)

In caso di crash il ripristino avviene nello stesso modo visto in precedenza per l'algoritmo steal/no flush con un'unica differenza:

non si esegue l'UNDO

Si deve costruire la sola lista **LC** delle transazioni terminate **dopo** il checkpoint

Caso 3: steal/flush

- Politica **steal**: il gestore del buffer può rubare pagine alle transazioni
- Politica **flush**: dopo il commit, il gestore del buffer trasferisce le modifiche in memoria persistente
- Siamo in politica **steal**: abbiamo bisogno dell'informazione **before state** (il gestore potrebbe aver reso persistenti modifiche di transazioni non terminate per liberare spazio nel buffer)
- Siamo in politica **flush**: non abbiamo bisogno dell'informazione **after state** (il gestore ha sicuramente reso persistenti le modifiche delle transazioni che hanno raggiunto il commit)

Caso 3: steal/flush

Nel file di log troveremo

- **<T_i, start>**: inizio della transazione T_i
- **<T_i, X, BS(X)>**: ogni volta che la transazione modifica un dato, si registra l'identificativo della transazione, l'oggetto X (per esempio, la tupla) modificato e il **Before State** di X (stato di X prima la modifica)

Al termine, possiamo immaginare due situazioni:

- Transazione T_i fallita (rollback deciso dalla transazione stessa, dal serializzatore...): si avrà quindi, nella storia, un **UNDO(T_i)** che disfa tutte le azioni precedentemente compiute dalla transazione T_i, e nel log verrà memorizzato il record **<T_i, abort>** seguito dalla direttiva **FORCE LOG**

Caso 3: steal/flush

Al termine, possiamo immaginare due situazioni:

- Transazione T_i parzialmente terminata (commit): il gestore delle transazione esegue la **verifica dei vincoli di integrità**
 - Se un vincolo non è soddisfatto, la transazione viene abortita e si eseguono le operazioni previste per il log
 - Se i vincoli sono soddisfatti, dato che siamo in politica flush, dobbiamo rendere persistenti le pagine modificate:
 1. si esegue un **FORCE LOG**
 2. si esegue un **FORCE delle pagine dati** di T_i
 3. si crea il record **$\langle T_i, \text{commit} \rangle$**
 4. si esegue un **FORCE LOG**

Algoritmo di ripristino 2 (steal/flush)

In caso di crash il ripristino avviene nello stesso modo visto in precedenza per l'algoritmo steal/no flush con un'unica differenza:

non si esegue il REDO

Si deve costruire la sola lista **LA** delle transazioni terminate **dopo** il checkpoint

Caso 4: no steal/flush

- Politica **no steal**: il gestore del buffer non può rubare pagine alle transazioni
- Politica **flush**: dopo il commit, il gestore del buffer trasferisce le modifiche in memoria persistente

In linea di principio questa combinazione di politiche non richiede file di log, infatti:

- Se T_i fallisce, si rilascia solo il buffer
- Se T_i termina e i vincoli sono soddisfatti, il gestore del buffer rende persistenti le modifiche

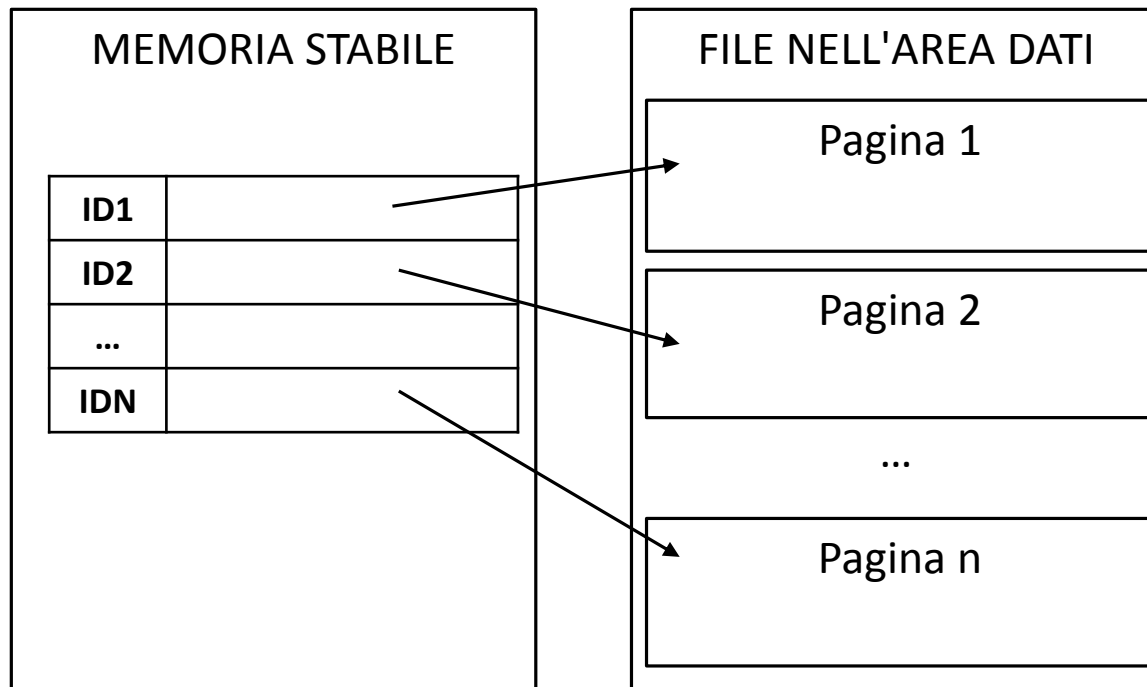
Caso 4: fase critica

- Se durante il trasferimento delle pagine modificate dalle transazioni andate a buon fine avviene un crash del sistema, non si può sapere quali pagine nella memoria permanente sono modificate e quali no
- Di conseguenza occorre che ogni trasferimento di pagina venga preceduto da una memorizzazione nella **memoria stabile** delle sole pagine modificate, poi si inizia un procedimento di trasferimento di pagine dalla memoria stabile al **DB** vero e proprio

Gestione dei dati con la tecnica delle pagine ombra

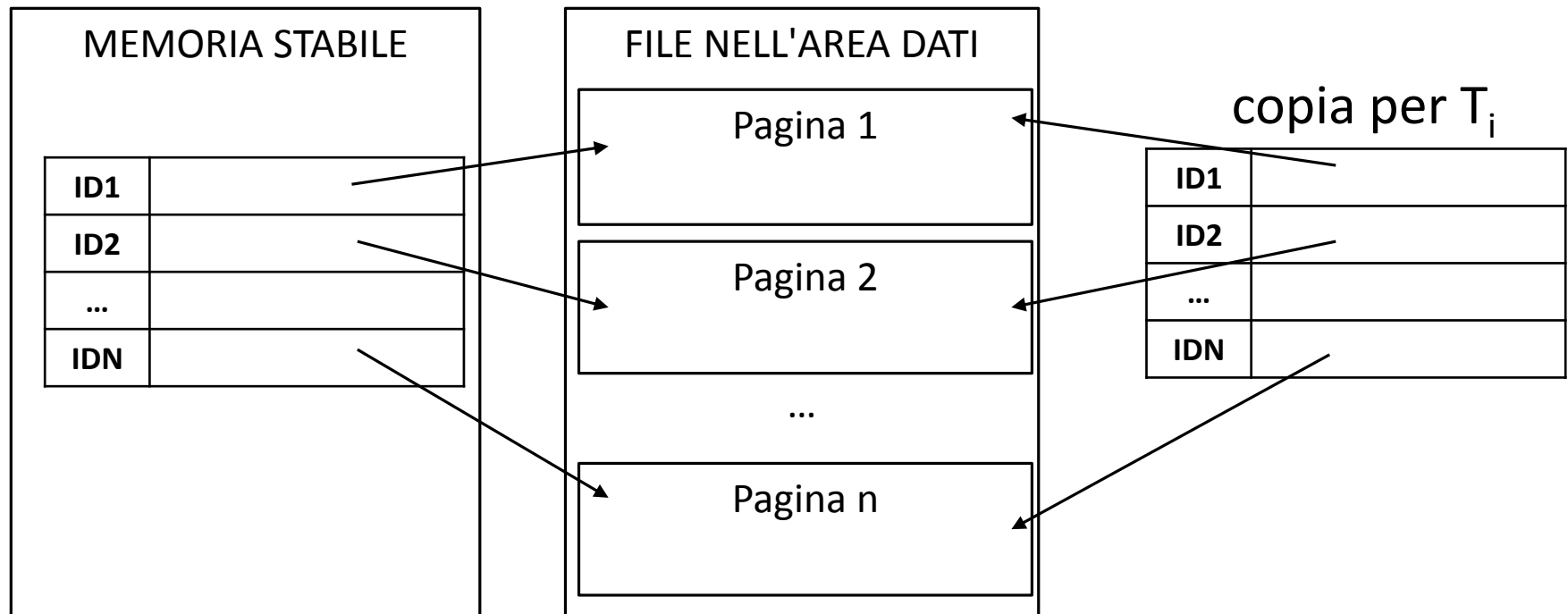
Consideriamo un file organizzato in pagine

In memoria stabile abbiamo una **tabella** che contiene un **identificatore** di una delle pagine dati e la sua **posizione** nel file



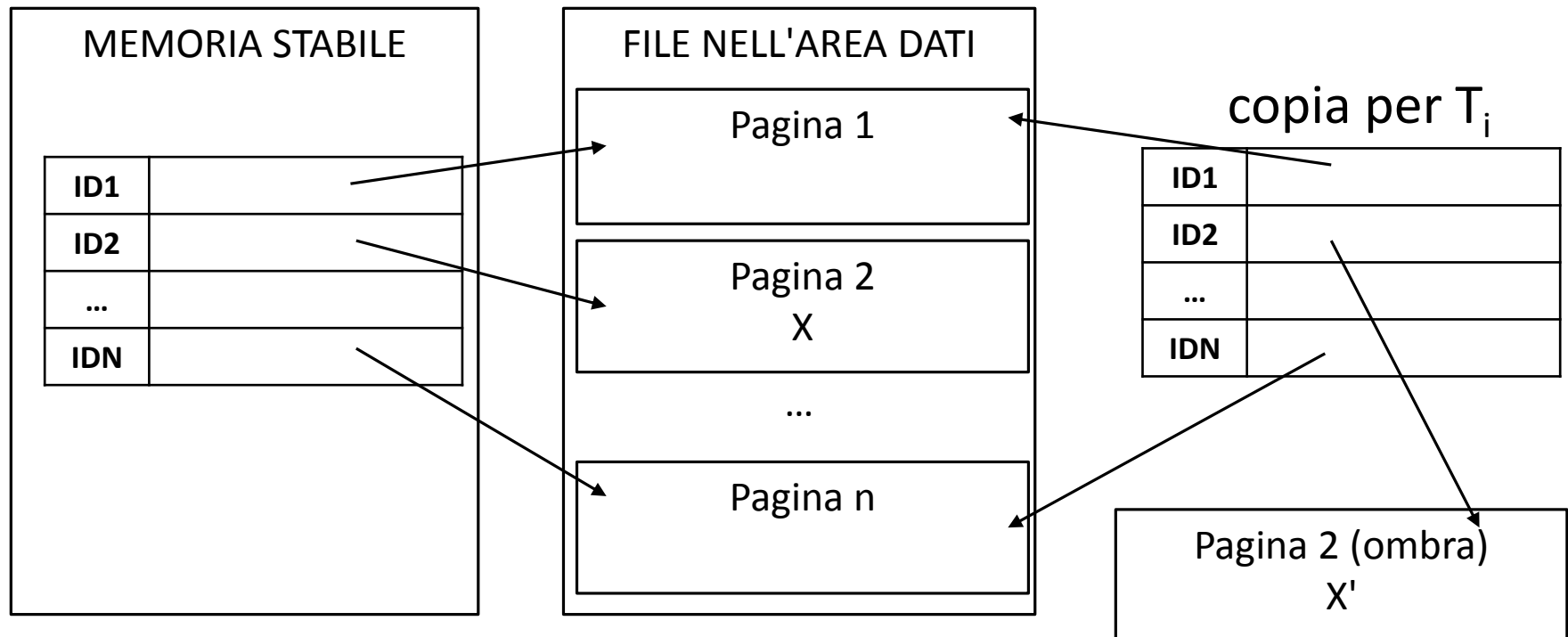
Tecnica delle pagine ombra

- Quando una transazione T_i vuole lavorare su dati così organizzati, acquisisce dal DBMS la copia della tabella delle pagine



Tecnica delle pagine ombra

- Supponiamo che T_i voglia modificare una pagina (pagina 2)
- Il sistema fa una copia della pagina richiesta e la T_i modifica il suo puntatore per farlo puntare alla copia della pagina



Tecnica delle pagine ombra

- Sia la copia che la pagina originaria sono in lock exclusive a T_i
- Quando la T_i termina in **rollback**, è sufficiente rilasciare la pagina ombra e rilasciare il lock della pagina originale
- Quando la T_i termina in **commit**:
 1. la tabella delle pagine (detta **tabella delle pagine ombra**) della T_i viene copiata in **memoria stabile**
 2. si sostituisce la tabella originaria (master) con la tabella proveniente dalla transazione T_i

Crash di sistema

- In caso di crash di sistema, se la caduta avviene prima del trasferimento della tabella delle pagine ombra in memoria stabile, la transazione semplicemente fallisce (si perdono le modifiche)
- Se invece la caduta di sistema avviene quando la tabella delle pagine ombra della transazione T_i è già copiata in memoria stabile, al **ripristino** si completa l'opera non portata a termine prima della caduta del sistema

Vantaggi

- Questa tecnica viene utilizzata in modo generale nella casistica no steal/flush
- E' però utilizzata soprattutto nei **DB multimediali** dove le tuple dei file di dati contengono campi molto pesanti (testi, filmati, audio)
- In questo caso si evita la riscrittura di BS e AS su campi molto pesanti

Guasti di periferiche

- I guasti fin qui presentati riguardano la memoria centrale
- C'è poi il problema dei guasti di periferiche di storage
- Abbiamo periferiche destinate a mantenere la persistenza dei dati ed altre che devono mantenere la persistenza dei log e dei dump
- Queste ultime sono in memoria stabile, i dati sono quindi duplicati (esempio tecniche **RAID**)

Tecnica di dump restore

- La base dati viene copiata (**dump**) periodicamente su memoria stabile (ad esempio nastri) e il log è vuoto
- Le transazioni modificano lo stato della base dati e scrivono il file di log con relativi **after state**
- Se avviene un guasto alla periferica che contiene la base di dati, il recupero avviene in due passi:
 1. Copia del dump sulla nuova periferica (**restore**)
 2. Si legge **tutto il log** e si effettua il **REDO(LC)**

Conclusione del corso di BD

Vantaggi del DBMS	Argomenti del corso
Integrazione	ER + Modello relazionale (schemi concettuali e logici)
Standardizzazione	
Integrità	Vincoli
Consistenza	Normalizzazione + Transazione ACID + Ripristino
Affidabilità	Ripristino
Indipendenza logica	Architettura ANSI/SPARC (viste di SQL)
Indipendenza fisica	Metodi di accesso / indici
Facilità d'uso	SQL
Concorrenza	Serializzatore / lock a due fasi
Sicurezza	Grant con SQL (non trattato nel corso)