Gestione delle transazioni

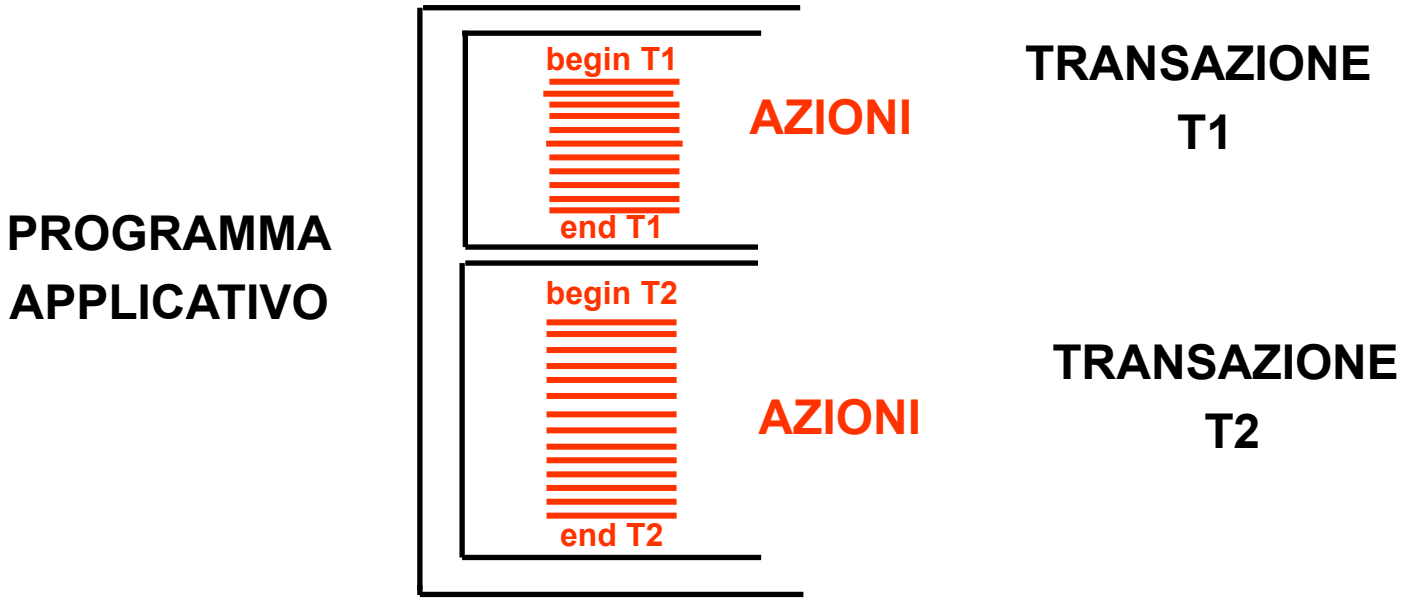
Atzeni-Ceri Capitoli 11 e 12

Definizione di transazione

- Transazione: parte di programma caratterizzata da un inizio (**begin-transaction,** `start transaction` in SQL), una fine (**end-transaction,** non esplicitata in SQL) e al cui interno 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**) e' in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti

Differenza fra applicazione e transazione



Una transazione

```
start transaction;
update ContoCorrente
  set Saldo = Saldo + 10 where NumConto =
  12202;
update ContoCorrente
  set Saldo = Saldo - 10 where NumConto =
  42177;
commit work;
```

Una 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
  - Durata (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 debbono 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
- Esito
  - Commit = caso "normale" e più frequente (99% ?)
  - Abort (o rollback)
    - richiesto dall'applicazione = suicidio
    - Richiesto dal sistema (violazione dei vincoli, concorrenza, incertezza in caso di fallimento) = omicidio

Consistenza

- La transazione rispetta i vincoli di integrita'
- Conseguenza:
  - se lo stato iniziale e' corretto
  - anche lo stato finale e' 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 potrbbe ottenerre con una esecuzione sequenziale
- Conseguenza: una transazione non espone i suoi stati intermedi
  - Si evita l' "effetto domino"

Durabilità (Persistenza)

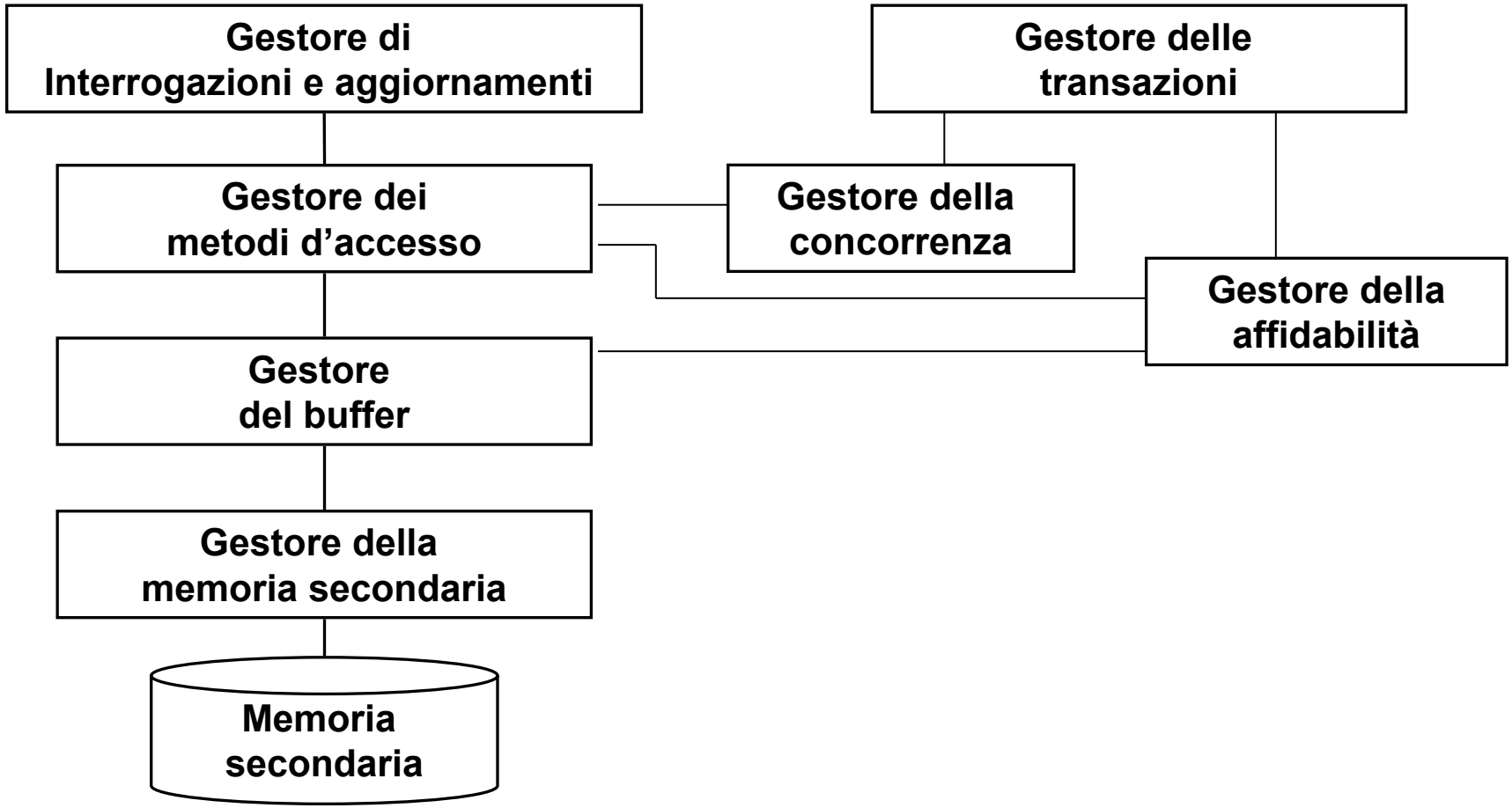
- Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti
  - Commit significa impegno

Transazioni e moduli di DBMS

- Atomicità e durabilità
  - 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

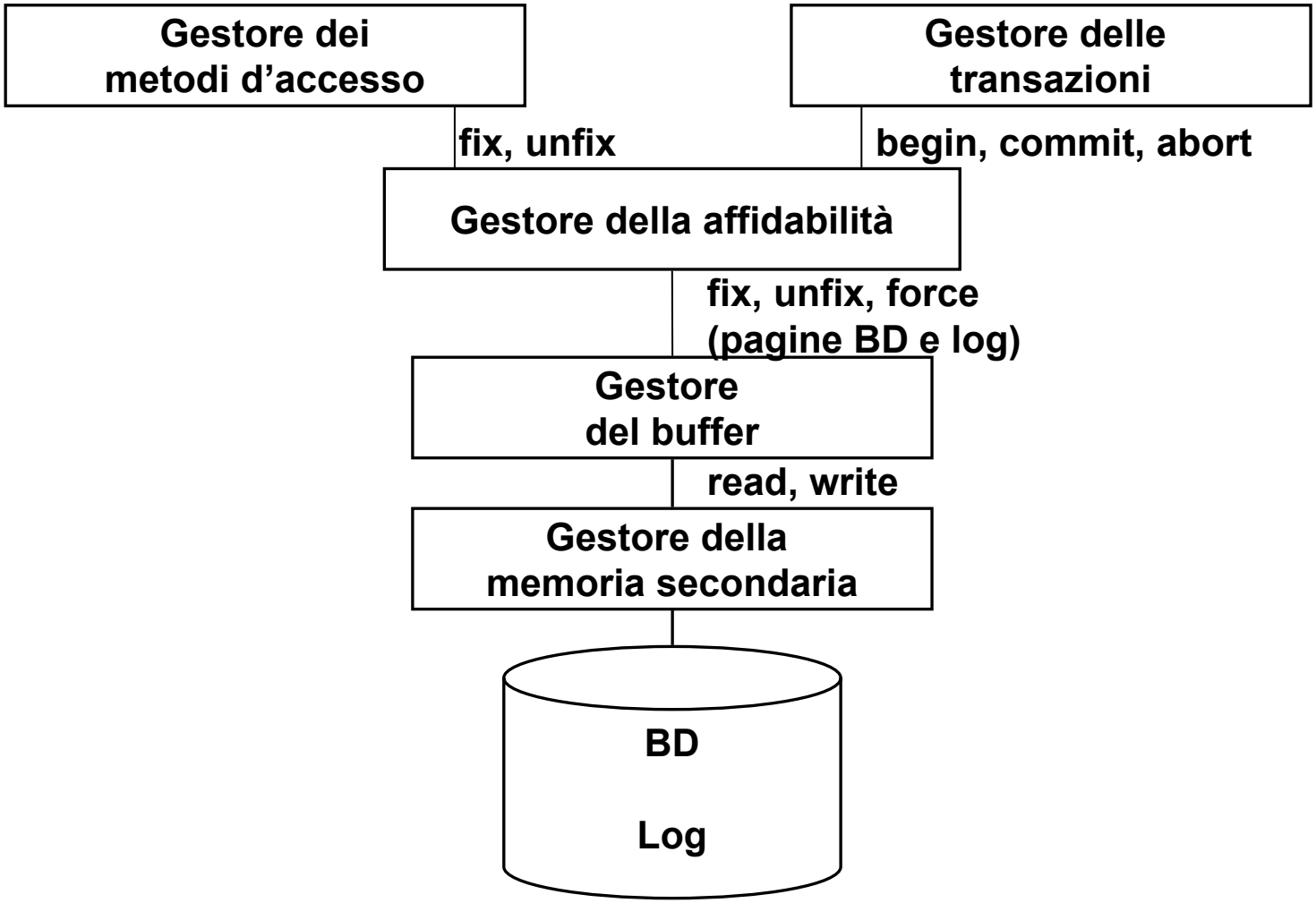


Gestore dell'affidabilità

- Gestisce l'esecuzione dei comandi transazionali
  - `start transaction` (B, begin)
  - `commit work` (C)
  - `rollback work` (A, abort)

  e le operazioni di ripristino (recovery) dopo i guasti :
  - *warm restart* e *cold restart*
- Assicura atomicità e durabilità
- Usa il **log**:
  - Un archivio permanente che registra le operazioni svolte
  - Due metafore: il filo di Arianna e i sassolini e le briciole di Hansel e Gretel

Architettura del controllore dell'affidabilità



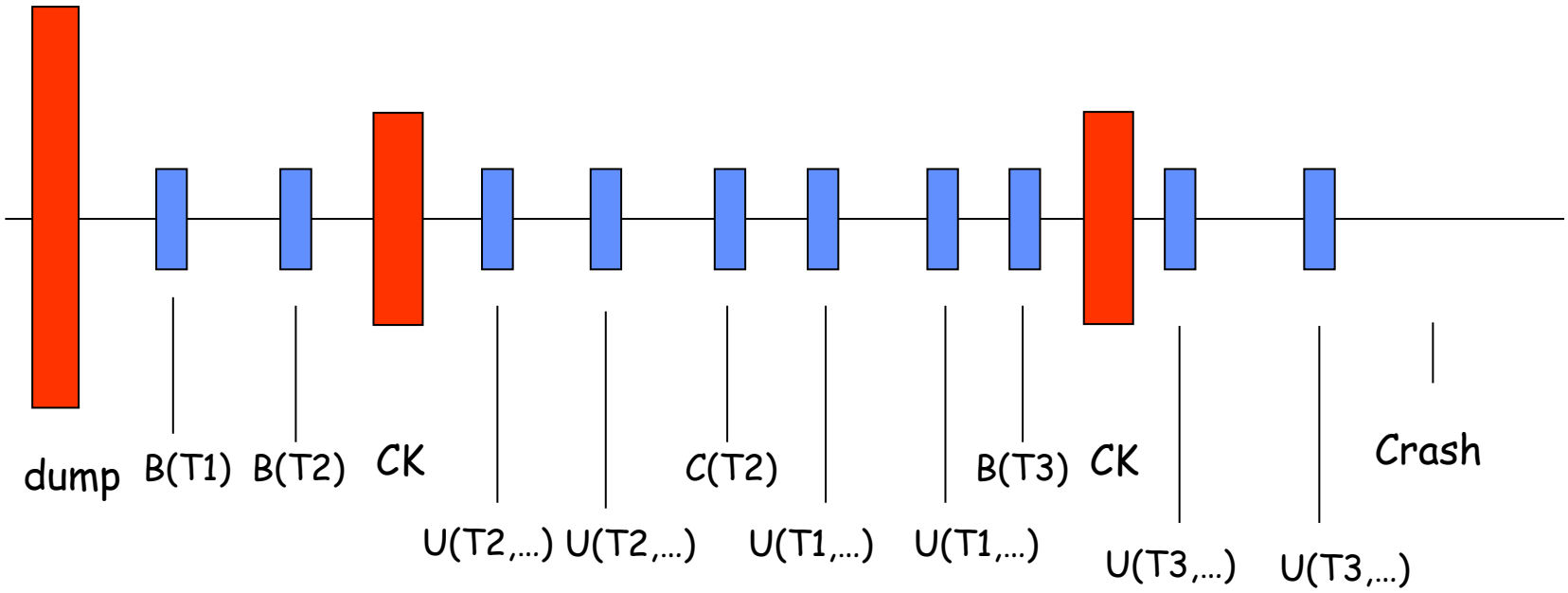
Persistenza delle memorie

- **Memoria centrale**: non e' persistente
- **Memoria di massa**: e' persistente ma puo' danneggiarsi
- **Memoria stabile**: memoria che non puo' danneggiarsi (e' 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
- "Diario di bordo": riporta tutte le operazioni in ordine
- Record nel log
  - *operazioni delle transazioni*
    - `begin`, B(T)
    - `insert`, I(T,O,AS)
    - `delete`, D(T,O,BS)
    - `update`, U(T,O,BS,AS)
    - `commit`, C(T), `abort`, A(T)
  - *record di sistema*
    - `dump`
    - `checkpoint`

Struttura del log



Log, checkpoint e dump: a che cosa servono?

- Il log serve "a ricostruire" le operazioni
- Checkpoint e dump servono ad evitare che la ricostruzione debba partire dall'inizio dei tempi
  - si usano con riferimento a tipi di guasti diversi

Undo e redo

- Undo di una azione su un oggetto *O:*
  - `update`, `delete`*:* copiare il valore del before state (*BS*) nell'oggetto *O*
  - `insert`: eliminare *O*
- Redo di una azione su un oggetto *O:*
  - `insert`, `update`: copiare il valore dell' after state (*AS*) into nell'oggetto *O*
  - `Delete`: cancella *O*
- *Idempotenza* di *undo e redo:*
  - *undo*(*undo*(*A*)) = *undo*(*A*)
  - *redo*(*redo*(*A*)) = *redo*(*A*)

Checkpoint

- Operazione che serve a "fare il punto" della situazione, semplificando le successive operazioni di ripristino:
  - ha lo scopo di registrare quali transaszioni sono attive in un certo istante (e dualmente, di confermare che le altre o non sono iniziate o sono finite)
- Paragone (estremo):
  - la "chiusura dei conti" di fine anno di una amministrazione:
    - dal 25 novembre (ad esempio) non si accettano nuove richieste di "operazioni" e si concludono tutte quelle avviate prima di accettarne di nuove

Checkpoint (2)

- Varie modalità, vediamo la più semplice:
  - si sospende l'accettazione di richieste di ogni tipo (scrittura, inserimenti, …, commit, abort)
  - si trasferiscono in memoria di massa (tramite *force*) tutte le pagine sporche relative a transazioni andate in commit
  - si registrano sul log in modo sincrono (*force*) gli identificatori delle transazioni in corso
  - si riprende l'accettazione delle operazioni
- Così siamo sicuri 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

Dump

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

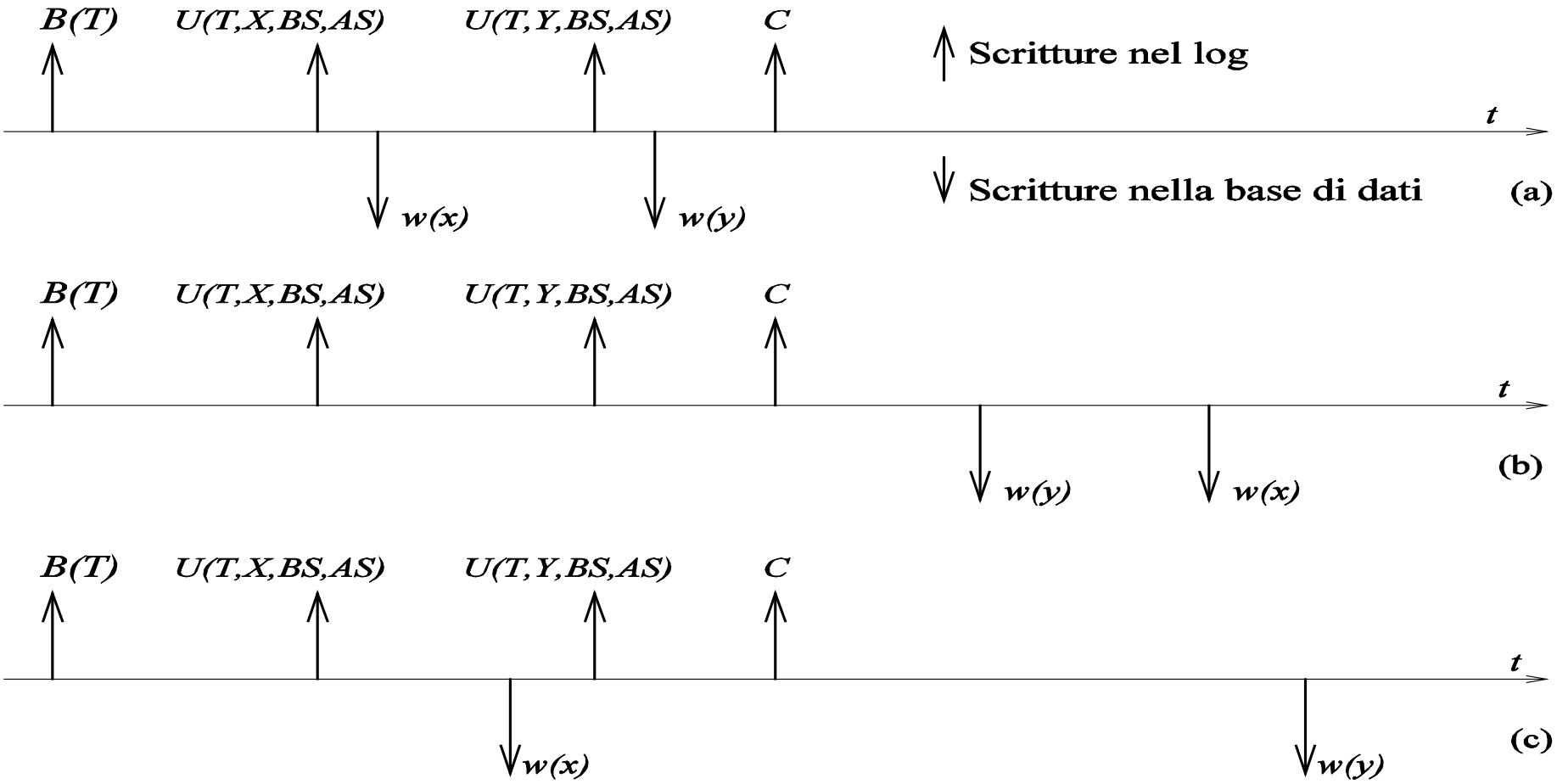
Esito di una transazione

- L'esito di una transazione è determinato irrevocabilmente quando viene scritto il record di **commit** nel log 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
- record di `abort` possono essere scritti in modo asincrono

Regole fondamentali per il log

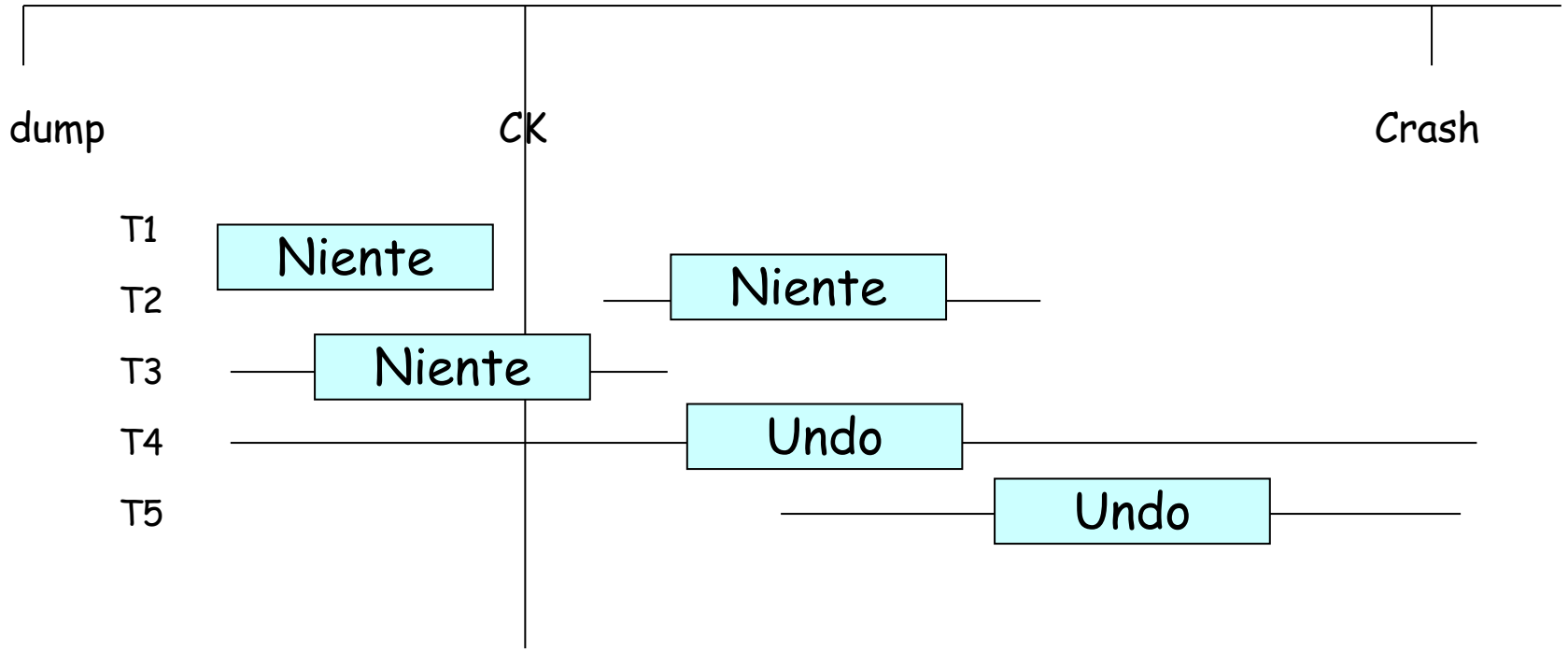
- **Write-Ahead-Log**:
  - si scrive il log (parte before) prima del database
    - consente di disfare le azioni
- **Commit-Precedenza**:
  - si scrive il log (parte after) prima del commit
    - consente di rifare le azioni
- Quando scriviamo nella base di dati?
  - Varie alternative

Scrittura nel log e nella base di dati



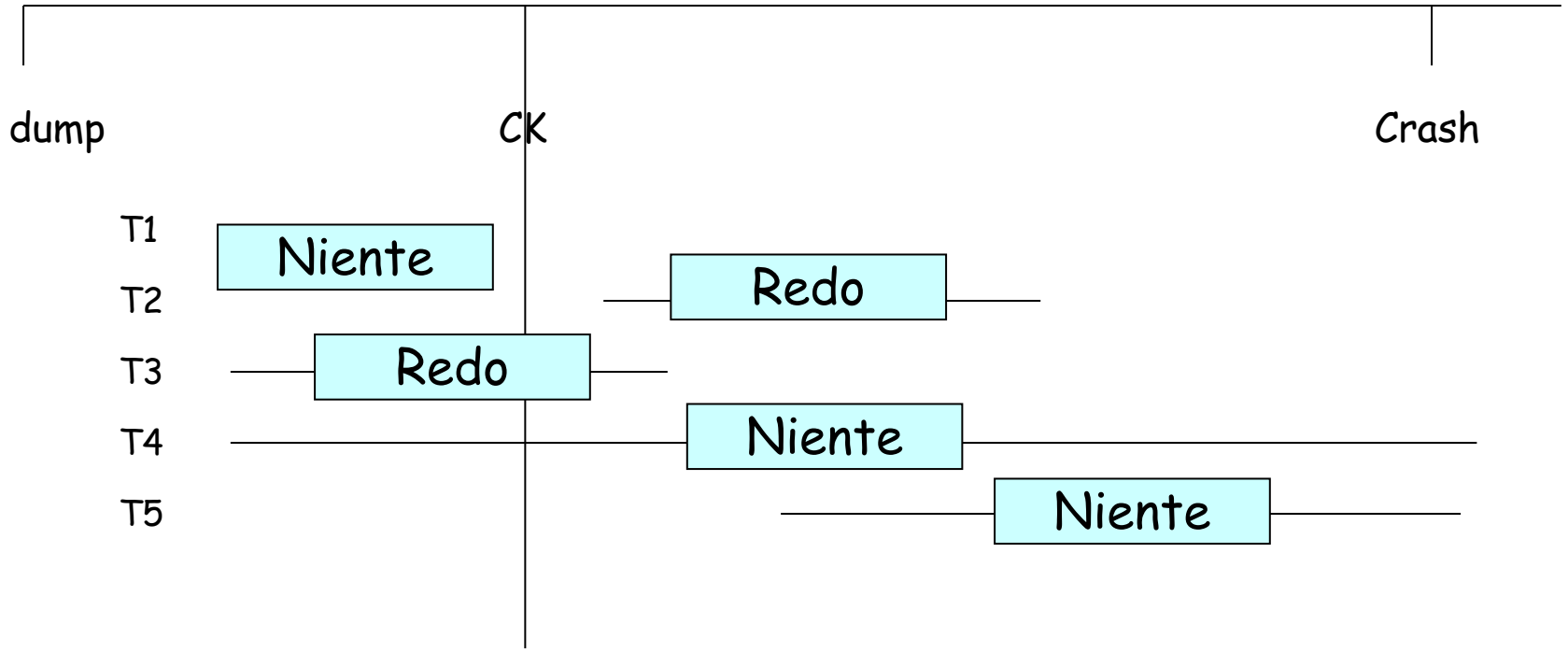
Modalità immediata

- Il DB contiene valori AS provenienti da transazioni uncommitted
- Richiede Undo delle operazioni di transazioni uncommited al momento del guasto
- Non richiede Redo



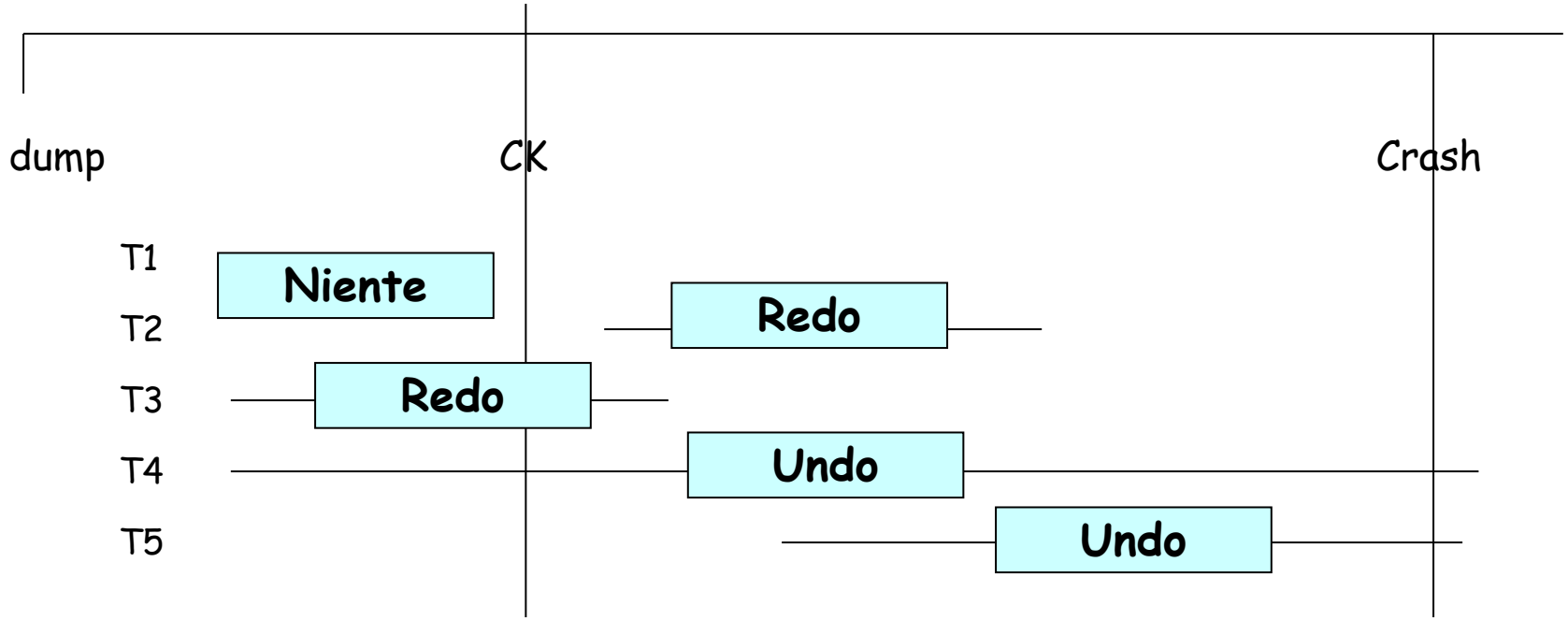
Modalità differita

- Il DB non contiene valori AS provenienti da transazioni uncommitted
- In caso di abort, non occorre fare niente
- Rende superflua la procedura di Undo. Richiede Redo



Essite una terza modalità: modalità mista

- La scrittura puo' avvenire in modalita' sia immediata che differita
- Consente l'ottimizzazione delle operazioni di flush
- Richiede sia Undo che Redo



Guasti

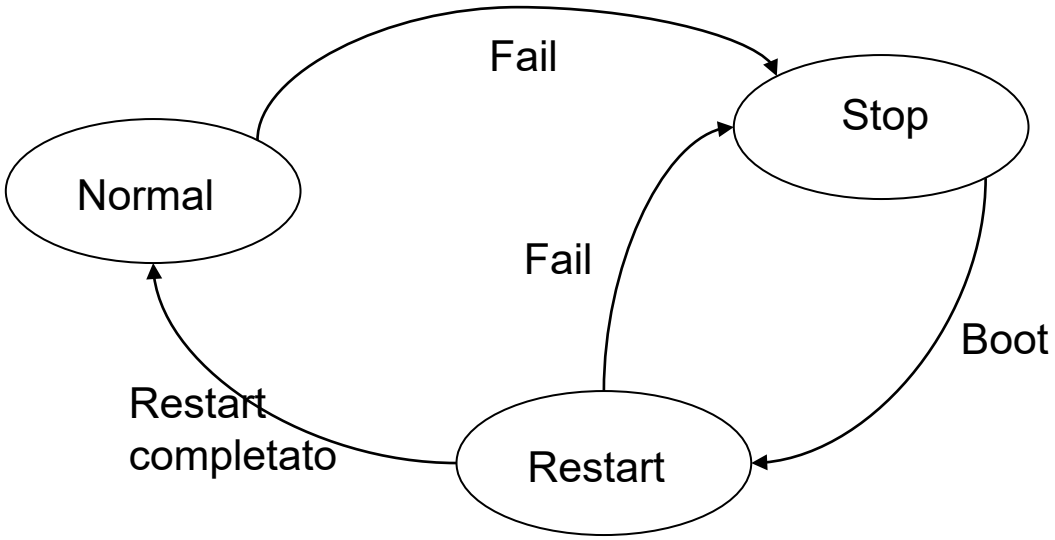
- **Guasti "soft":** errori di programma, crash di sistema, caduta di tensione
  - si perde la memoria centrale
  - non si perde la memoria secondaria

  warm restart, ripresa a caldo

- **Guasti "hard":** 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

Modello "fail-stop"



Processo di restart

- Obiettivo: classificare le transazioni in
  - completate (tutti i dati in memoria stabile)
  - in commit ma non necessariamente completate (può servire redo)
  - senza commit (vanno annullate, undo)

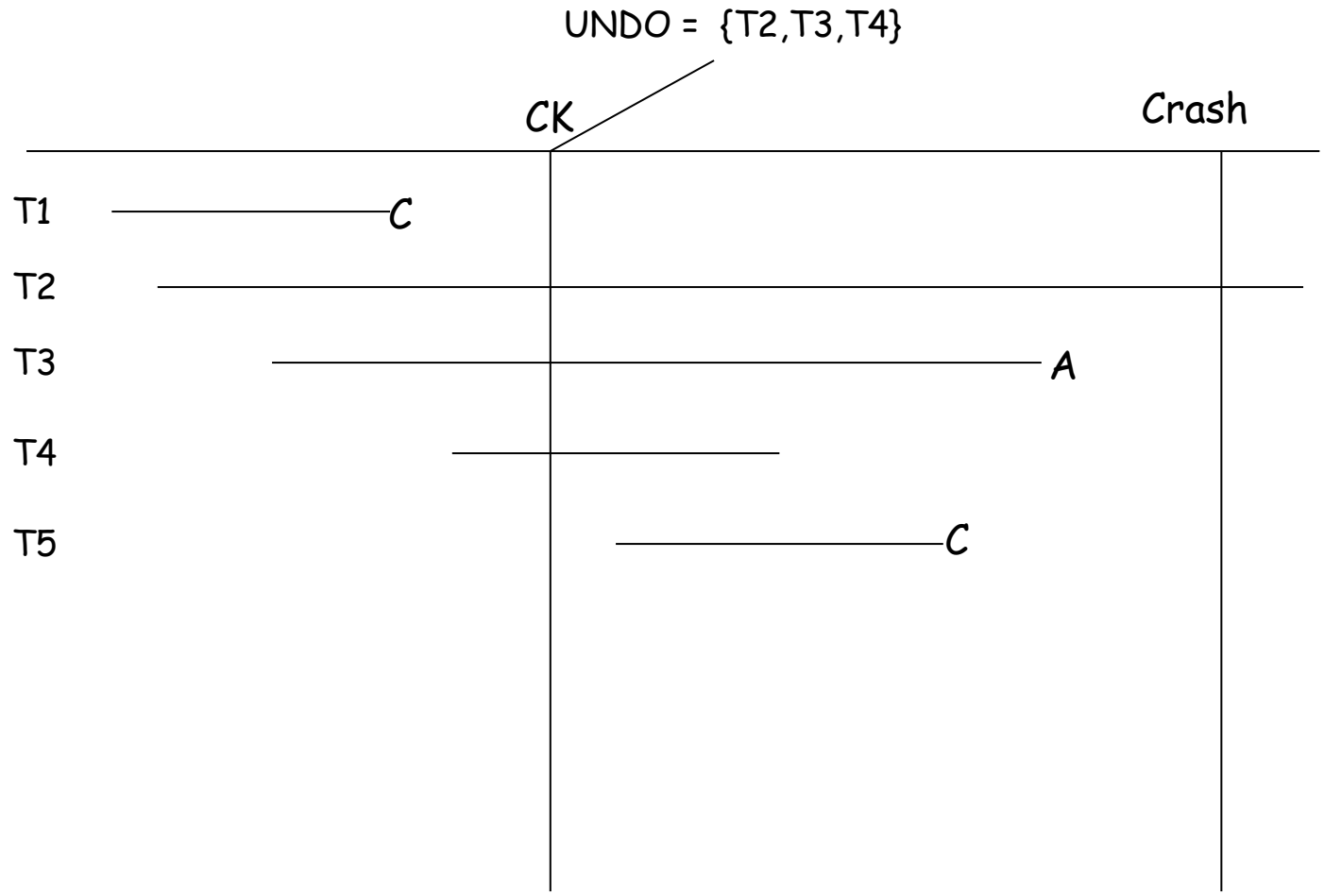
Ripresa a caldo

Quattro fasi:

- trovare l'ultimo checkpoint (ripercorrendo il log a ritroso)
- costruire gli insiemi *UNDO* (transazioni da disfare) e *REDO* (transazioni da rifare)
- ripercorrere il log all'indietro, fino alla più vecchia azione delle transazioni in *UNDO e REDO*, disfacendo tutte le azioni delle transazioni in *UNDO*
- ripercorrere il log in avanti, rifacendo tutte le azioni delle transazioni in *REDO*

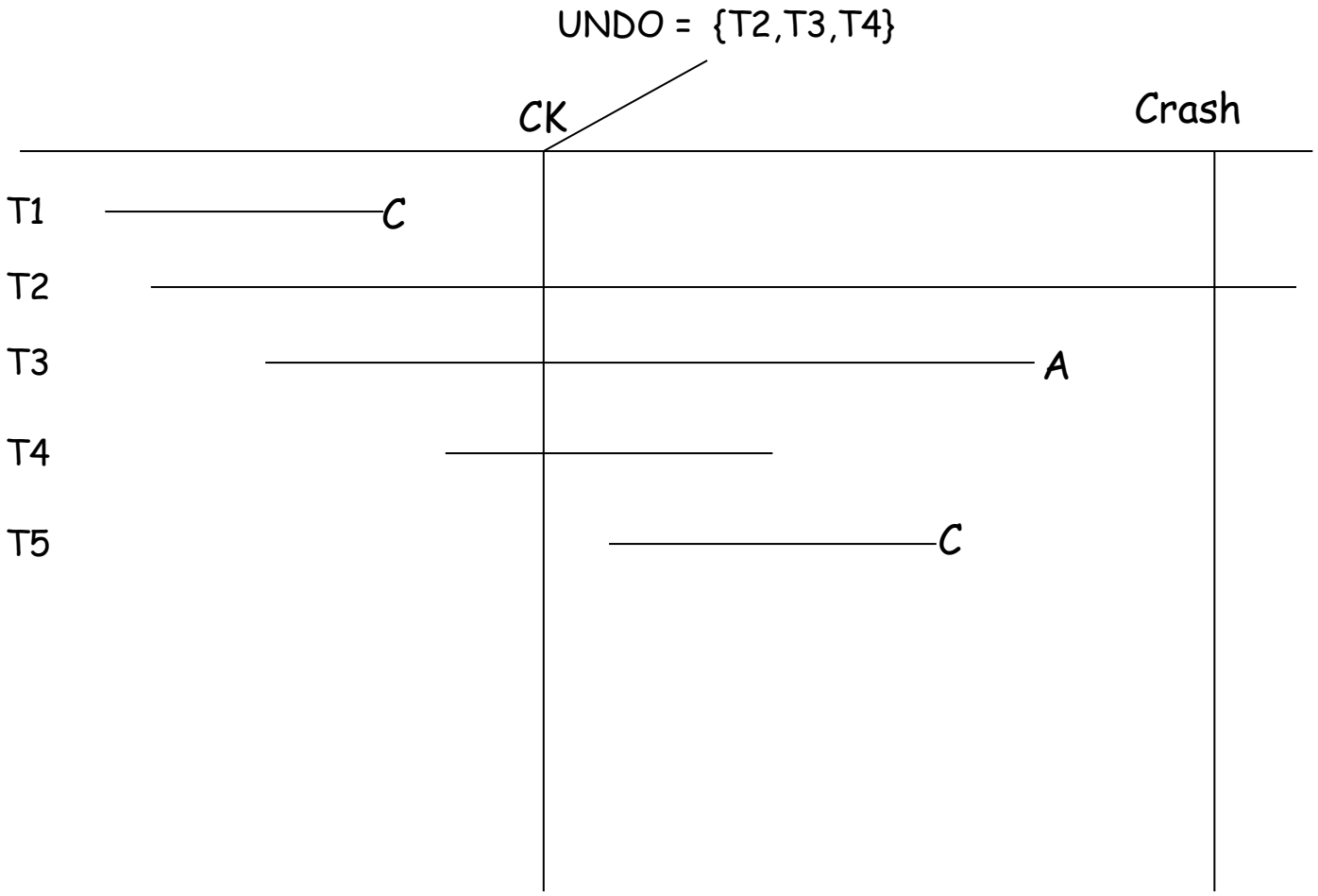
Esempio di warm restart

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

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)



2. Costruzione degli insiemi UNDO e REDO

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

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Setup

3. Fase UNDO

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

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Setup

---

4. D(O6)

5. O5 =B7

6. O3 = B5

7. O2 =B3

8. O1=B1

Undo

4. Fase REDO

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

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Setup

---

4. D(O6)

5. O5 =B7

6. O3 = B5

7. O2 =B3

8. O1=B1

Undo

---

9. O3 = A4

10. O4 = A6

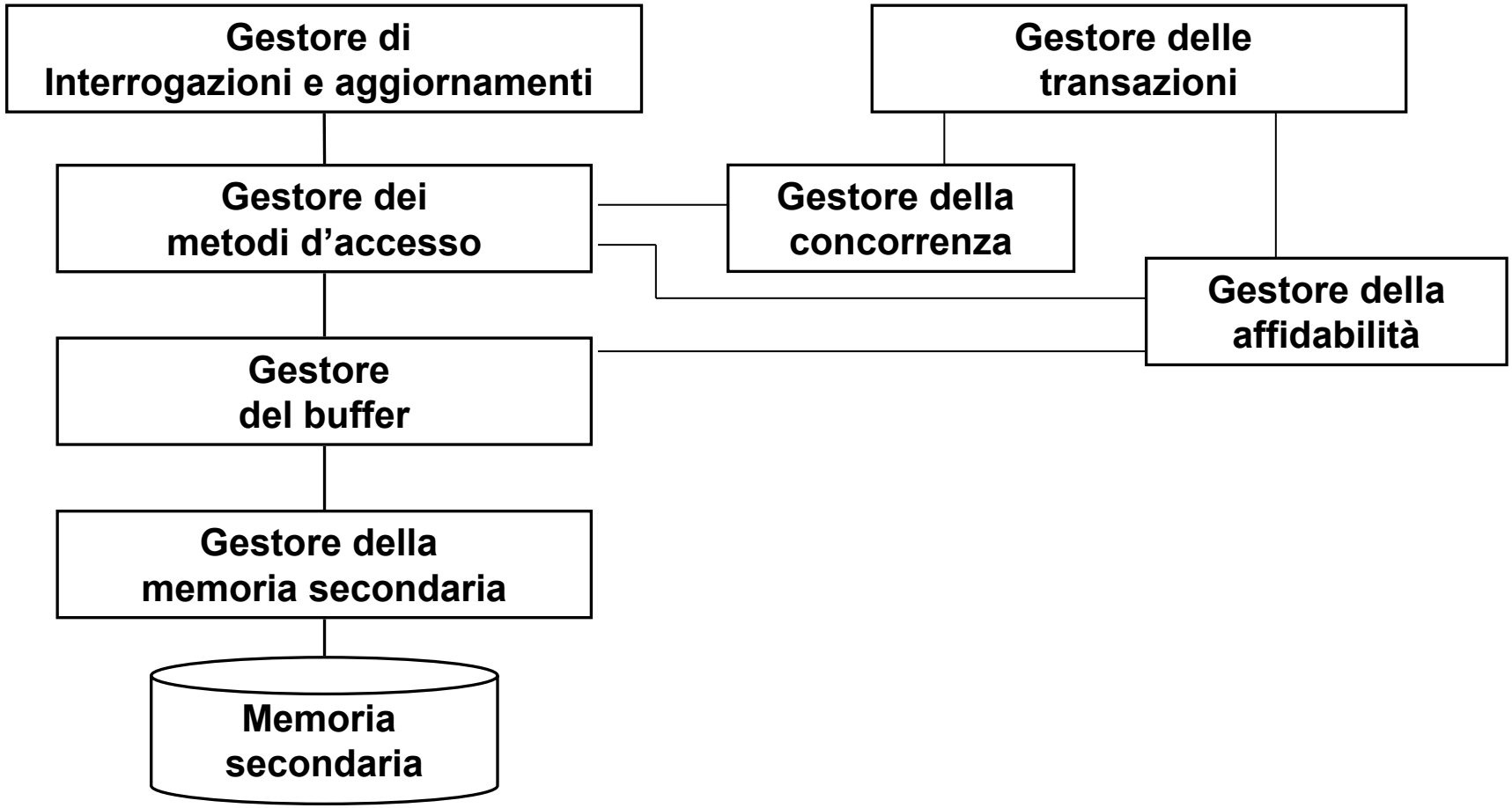
Redo

Ripresa a freddo

- Si ripristinano i dati a partire dal backup
- Si eseguono le operazioni registrate sul giornale fino all'istante del guasto
- Si esegue una ripresa a caldo

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



Controllo di concorrenza

- La concorrenza è fondamentale: decine o centinaia di transazioni al secondo, non possono essere seriali
- Esempi: banche, prenotazioni aeree

Modello di riferimento

- Operazioni di input-output su oggetti astratti *x*, *y*, *z*

Problema

- Anomalie causate dall'esecuzione concorrente, che quindi va governata

Perdita di aggiornamento

- Due transazioni identiche:
  - *t* 1 : *r(*x), *x* = *x* + 1, *w(*x)
  - *t* 2 : *r(*x), *x* = *x* + 1, *w(*x)
- Inizialmente *x*=2; dopo un'esecuzione seriale *x*=4
- Un'esecuzione concorrente:

t_1	t_2
bot	
$r_1(x)$	
$x = x + 1$	
	bot
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
commit	
	$w_2(x)$
	commit

- Un aggiornamento viene perso: *x*=3

Lettura sporca

t_1	t_2
bot	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	bot
	$r_2(x)$
abort	
	commit

- Aspetto critico: t_2 ha letto uno stato intermedio ("sporco") e lo può comunicare all'esterno

Letture inconsistenti

- t_1 legge due volte:

t_1	t_2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

- t_1 legge due valori diversi per x !

Aggiornamento fantasma

- Assumere ci sia un vincolo $y + z = 1000$;

t_1	t_2
bot	
$r_1(y)$	
	bot
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = y + z$	
commit	

- $s = 1100$: il vincolo sembra non soddisfatto, t_1 vede un aggiornamento non coerente

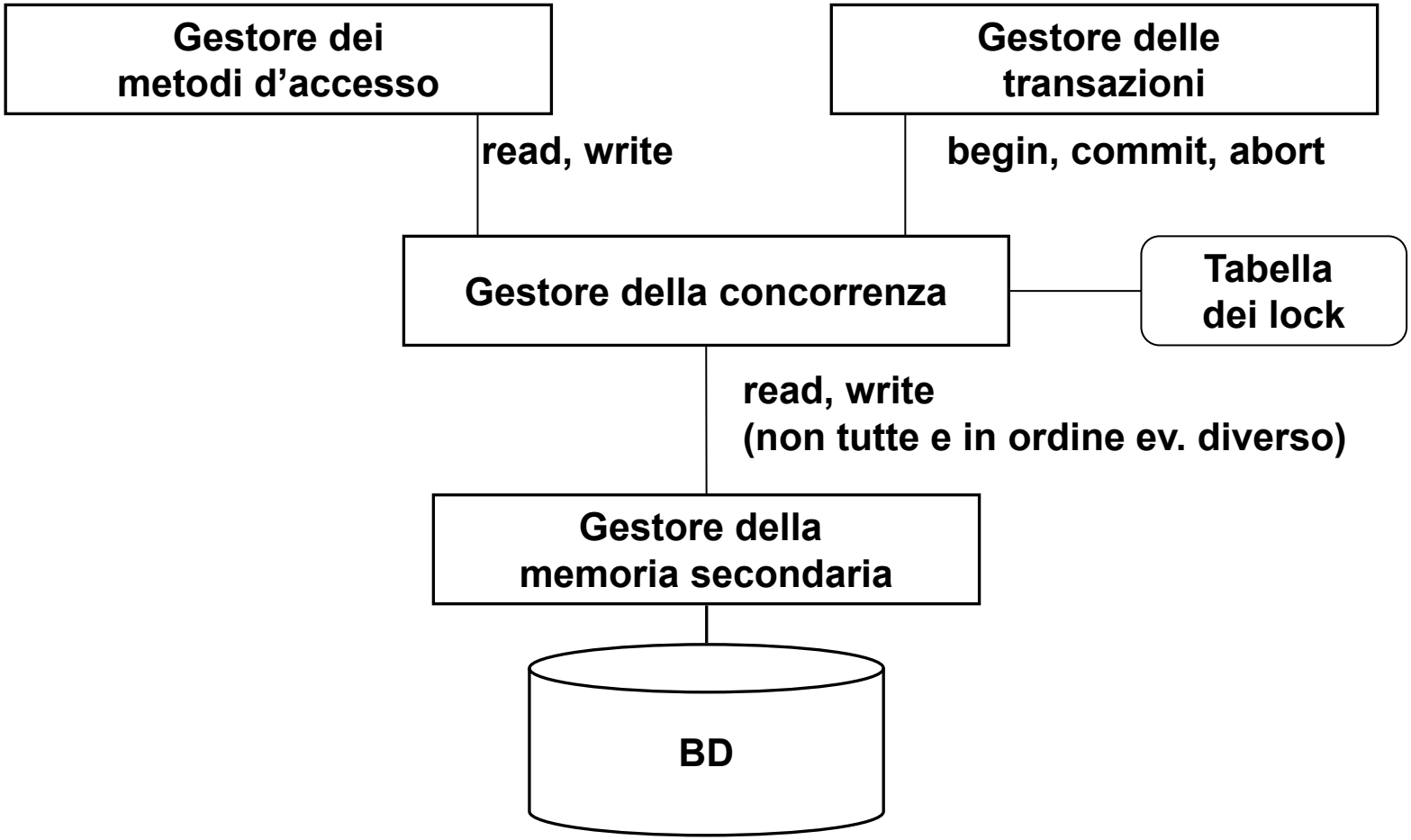
Inserimento fantasma

t_1	t_2
`bot`	
"legge gli stipendi degli impiegati del dip A e calcola la media"	
	`bot`
	"inserisce un impiegato in A"
	`commit`
"legge gli stipendi degli impiegati del dip A e calcola la media"	
`commit`	

Anomalie

- Perdita di aggiornamento W-W
- Lettura sporca R-W (o W-W) con abort
- Letture inconsistenti R-W
- Aggiornamento fantasma R-W
- Inserimento fantasma R-W su dato "nuovo"

Gestore della concorrenza (ignorando buffer e affidabilità)



Schedule

- Sequenza di operazioni di input/output di transazioni concorrenti
- Esempio:

$$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$$

- Ipotesi semplificativa:
  - consideriamo la **commit-proiezione** e ignoriamo le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule

Controllo di concorrenza

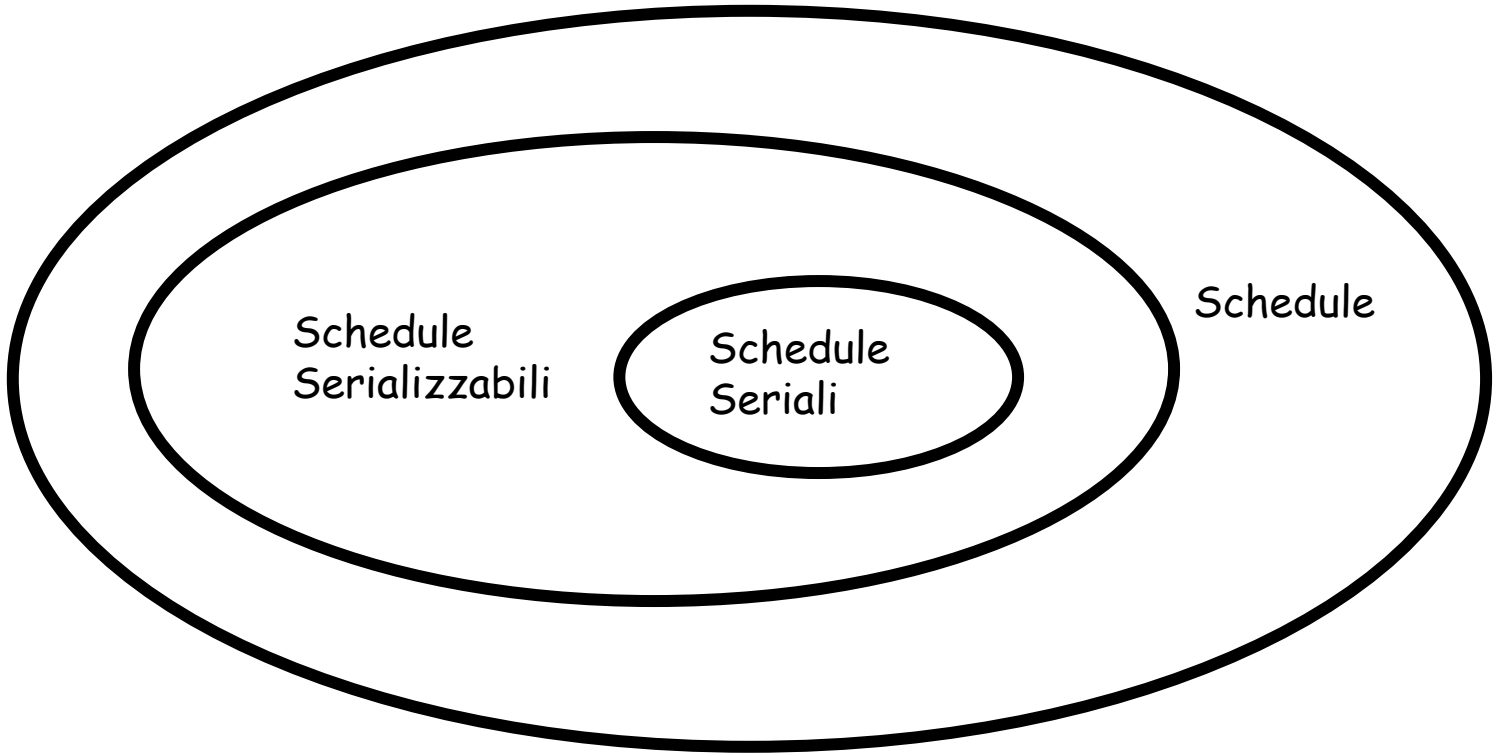
- *Obiettivo*: evitare le anomalie
- *Scheduler*: un sistema che accetta o rifiuta (o riordina) le operazioni richieste dalle transazioni
- *Schedule seriale:* le transazioni sono separate, una alla volta

$$S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$$

- *Schedule serializzabile:* produce lo stesso risultato di uno schedule seriale sulle stesse transazioni
  - Richiede una nozione di equivalenza fra schedule

Idea base

- Individuare classi di schedule serializzabili che siano sottoclassi degli schedule possibili, siano serializzabili e la cui proprietà di serializzabilità sia verificabile a costo basso



View-Serializzabilità

- Definizioni prelilminari:
  - $r_i(x)$ ***legge-da*** $w_j(x)$ in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è $w_k(x)$ fra $r_i(x)$ e $w_j(x)$ in S
  - $w_i(x)$ in uno schedule S è ***scrittura finale*** *se* è l'ultima scrittura dell'oggetto x in S
- Schedule ***view-equivalenti*** *(*$S_i \approx_V S_j$): hanno la stessa relazione *legge-da* e le stesse scritture finali
- Uno schedule è ***view-serializzabile*** se è view-equivalente ad un qualche schedule seriale
- L'insieme degli schedule view-serializzabili è indicato con **VSR**

"Nello studio della View-Serializzabilità si assume che una transazione **scriva solo una volta** un oggetto X" (Garcia-Molina-Ullman-Widom, Database Systems The Complete Book).

View serializzabilità: esempi

- S_3 : $w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  S_4 : $w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
  S_5 : $w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
  S_6 : $w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$
  - S_3 è view-equivalente allo schedule seriale S_4 (e quindi è view-serializzabile)
  - S_5 non è view-equivalente a S_4, ma è view-equivalente allo schedule seriale S_6, e quindi è view-serializzabile
- S_7 : $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$ (perdita di aggiornamento)
  S_8 : $r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$ (letture inconsistenti)
  S_9 : $r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$ (aggiornamento fantasma)
  - S_7, S_8, S_9 non view-serializzabili

View serializzabilità

- Complessità:
  - la verifica della view-equivalenza di due dati schedule:
    - polinomiale
  - decidere sulla View serializzabilità di uno schedule:
    - problema NP-completo
- Non è utilizzabile in pratica

Conflict-serializzabilità

- Definizione preliminare:
  - Un'azione a_i è in *conflitto* con a_j ($i \neq j$), se operano sullo stesso oggetto e almeno una di esse è una scrittura. Due casi:
    - conflitto *read-write* (*rw* o *wr*)
    - conflitto *write-write* (*ww*).
- *Schedule conflict-equivalenti* ($S_i \approx_C S_j$): includono le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi
- Uno schedule è *conflict-serializable* se è conflict-equivalente ad un qualche schedule seriale
- L'insieme degli schedule conflict-serializzabili è indicato con **CSR**

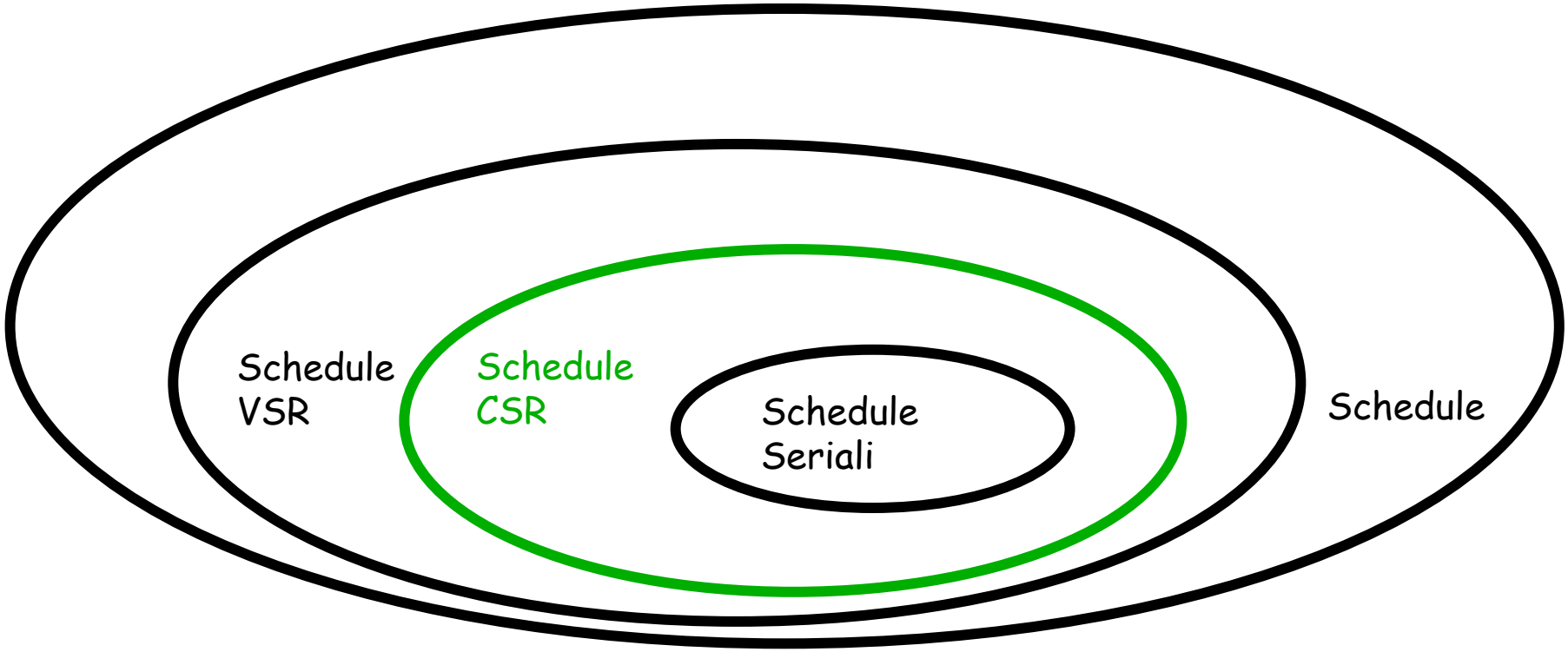
CSR e VSR

- Ogni schedule conflict-serializzabile è view-serializzabile, ma non necessariamente viceversa
- Controesempio per la non necessità:

- r1(x) w2(x) w1(x) w3(x)

  - view-serializzabile: view-equivalente a r1(x) w1(x) w2(x) w3(x)
  - non conflict-serializzabile

CSR e VSR



Verifica di conflict-serializzabilità

- Per mezzo del **grafo dei conflitti**:
  - un nodo per ogni transazione t_i
  - un arco (orientato) da t_i a t_j se c'è almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j
- Teorema
  - **Uno schedule è in CSR se e solo se il grafo è aciclico**

CSR e aciclicità del grafo dei conflitti

- Se uno schedule S è CSR allora è $\approx_C$ ad uno schedule seriale.
  Supponiamo le transazioni nello schedule seriale ordinate secondo il TID: t_1, t_2, …, t_n . Poiché lo schedule seriale ha tutti i conflitti nello stesso ordine dello schedule S, nel grafo di S ci possono essere solo archi (i,j) con i<j e quindi il grafo non può avere cicli, perché un ciclo richiede almeno un arco (i,j) con i>j.
- Se il grafo di S è aciclico, allora esiste fra i nodi un "ordinamento topologico" (cioè una numerazione dei nodi tale che il grafo contiene solo archi (i,j) con i<j). Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è equivalente a S, perché per tutti i conflitti (i,j) si ha sempre i<j.

Controllo della concorrenza in pratica

- Anche la conflict-serializabilità, pur più rapidamente verificabile (l'algoritmo, con opportune strutture dati richiede tempo lineare), è inutilizzabile in pratica
- La tecnica sarebbe efficiente se potessimo conoscere il grafo dall'inizio, ma così non è: uno scheduler deve operare "incrementalmente", cioè ad ogni richiesta di operazione decidere se eseguirla subito oppure fare qualcos'altro;
  non è praticabile mantenere il grafo, aggiornarlo e verificarne l'aciclicità ad ogni richiesta di operazione
- Inoltre, la tecnica si basa sull'ipotesi di commit-proiezione
- In pratica, si utilizzano tecniche che
  - garantiscono la conflict-serializzabilità senza dover costruire il grafo
  - non richiedono l'ipotesi della commit-proiezione

Lock

- Principio:
  - Tutte le letture sono precedute da *r_lock* (lock condiviso) e seguite da *unlock*
  - Tutte le scritture sono precedute da *w_lock* (lock esclusivo) e seguite da *unlock*
- Quando una transazione prima legge e poi scrive un oggetto, può:
  - richiedere subito un lock esclusivo
  - chiedere prima un lock condiviso e poi uno esclusivo (*lock escalation*)
- Il *lock manager* riceve queste richieste dalle transazioni e le accoglie o rifiuta, sulla base della tavola dei conflitti

Gestione dei lock

- Basata sulla tavola dei conflitti

Richiesta	Stato della risorsa		
	free	*r_locked*	*w_locked*
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends	OK / free

  - Un contatore tiene conto del numero di "lettori"; la risorsa è rilasciata quando il contatore scende a zero
- Se la risorsa non è concessa, la transazione richiedente è posta in attesa (eventualmente in coda), fino a quando la risorsa non diventa disponibile
- Il lock manager gestisce una tabella dei lock, per ricordare la situazione

Locking a due fasi (2PL)

- Usato da quasi tutti i sistemi
- Garantisce "a priori" la conflict-serializzabilità
- Basata su due regole:
  - "proteggere" tutte le letture e scritture con lock
  - un vincolo sulle richieste e i rilasci dei lock:
    - **una transazione, dopo aver rilasciato un lock, non può acquisirne altri**

2PL e CSR

- Ogni schedule 2PL e' anche conflict serializzabile, ma non necessariamente viceversa
- Controesempio per la non necessita':

$$r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$$

  - Viola 2PL
  - Conflict-serializzabile

- Sufficienza: vediamo

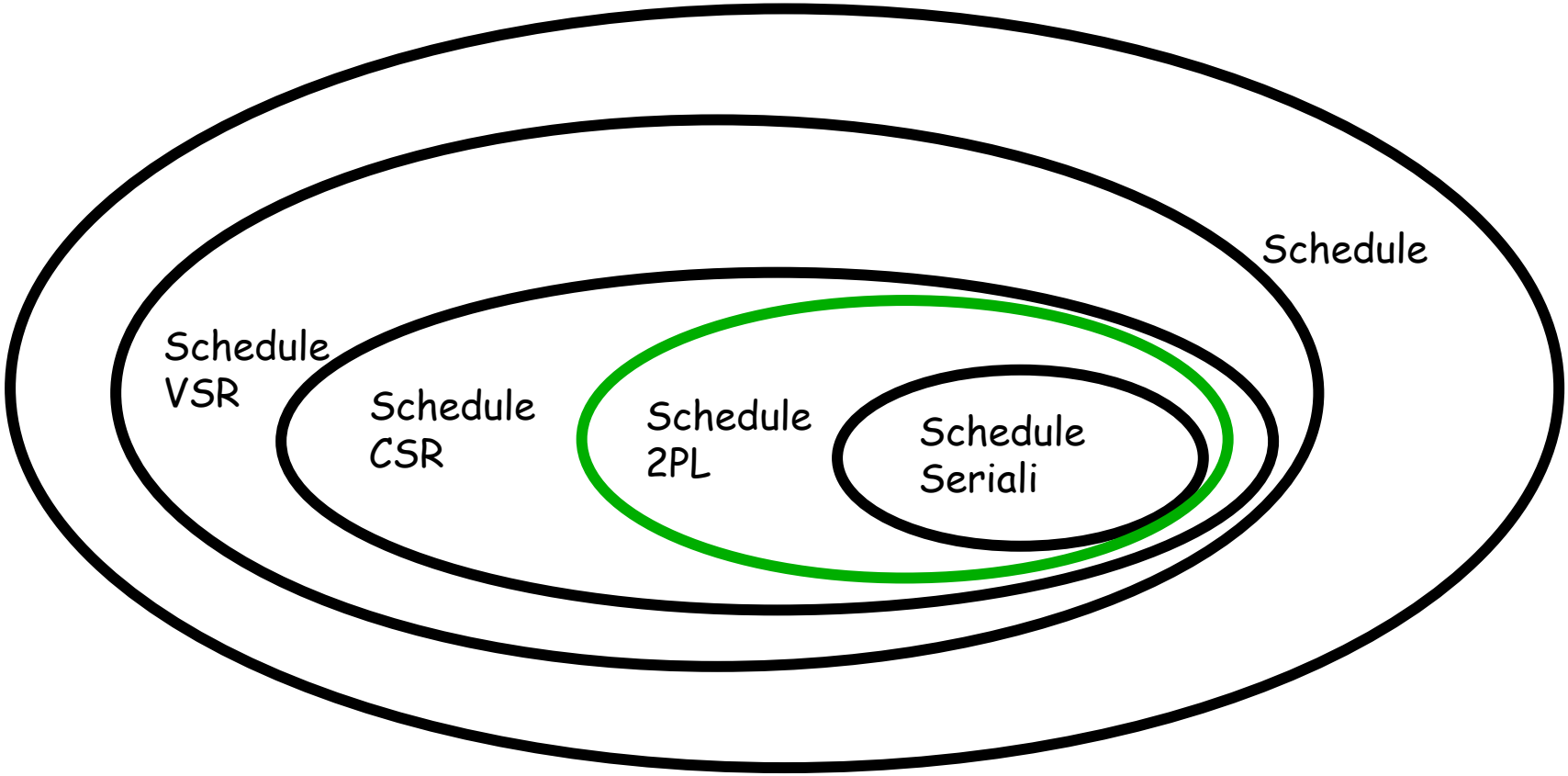
2PL implica CSR

- S schedule 2PL
- Consideriamo per ciascuna transazione l'istante in cui ha tutte le risorse e sta per rilasciare la prima
- Ordiniamo le transazioni in accordo con questo valore temporale e consideriamo lo schedule seriale corrispondente
- Vogliamo dimostrare che tale schedule è equivalente ad S:
  - allo scopo, consideriamo un conflitto fra un'azione di t_i e un'azione dei t_j con i<j; è possibile che compaiano in ordine invertito in S? no, perché in tal caso t_j dovrebbe aver rilasciato la risorsa in questione prima della sua acquisizione da parte di t_i

Locking a due fasi stretto

- Condizione aggiuntiva:
  - **I lock possono essere rilasciati solo dopo il commit o abort**
- Supera la necessità dell'ipotesi di commit-proiezione (ed elimina il rischio di letture sporche)

CSR, VSR e 2PL


Schedule
Schedule VSR
Schedule CSR
Schedule 2PL
Schedule Seriali

Stallo (deadlock)

- Attese incrociate: due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra
- Esempio:
  - t_1: *read*(x), *write*(y)
  - t_2: *read*(y), *write*(x)
  - Schedule:

    $r\_lock_1(x)$, $r\_lock_2(y)$, $read_1(x)$, $read_2(y)$ $w\_lock_1(y)$, $w\_lock_2(x)$

Risoluzione dello stallo

- Uno stallo corrisponde ad un ciclo nel grafo delle attese (nodo=transazione, arco=attesa)
- Tre tecniche
  1. Timeout (problema: scelta dell'intervallo, con trade-off)
  2. Rilevamento dello stallo
  3. Prevenzione dello stallo
- Rilevamento: ricerca di cicli nel grafo delle attese
- Prevenzione: uccisione di transazioni "sospette" (può esagerare)

Livelli di isolamento in SQL:1999 (e JDBC)

- Le tranzazioni possono essere definite **`read-only`** (non possono richiedere lock esclusivi)
- Il livello di isolamento può essere scelto per ogni transazione
  - **`read uncommitted`** permette letture sporche, letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`read committed`** evita letture sporche ma permette letture inconsistenti, aggiornamenti fantasma e inserimenti fantasma
  - **`repeatable read`** evita tutte le anomalie esclusi gli inserimenti fantasma
  - **`serializable`** evita tutte le anomalie
- Nota:
  - la perdita di aggiornamento è sempre evitata

Livelli di isolamento: implementazione

- Sulle scritture si ha sempre il 2PL stretto (e quindi si evita la perdita di aggiornamento)
- **`read uncommitted`**:
  - nessun lock in lettura (e non rispetta i lock altrui)
- **`read committed`**:
  - lock in lettura (e rispetta quelli altrui), ma senza 2PL
- **`repeatable read`**:
  - 2PL anche in lettura, con lock sui dati
- **`serializable`**:
  - 2PL con lock di predicato

Lock di predicato

- Caso peggiore:
  - sull'intera relazione
- Se siamo fortunati:
  - sull'indice

Transazioni in MySQL

- I comandi **COMMIT RELEASE** o **ROLLBACK RELEASE**, oltre a chiudere la transazione chiudono anche la connessione al server.

- Con l'istruzione **SET AUTOCOMMIT=0** possiamo disattivare l'autocommit: in questo caso non è più necessario avviare le transazioni con START TRANSACTION, e tutti gli aggiornamenti rimarranno sospesi fino all'uso di COMMIT o ROLLBACK.

DB transazionali: InnoDB

- È altamente sconsigliato ovviamente l'utilizzo di tabelle MyISAM nelle transazioni dato che su di esse non è possibile effettuare il ROLLBACK; con questo engine, le modifiche richieste sono immediatamente effettive.

- Usare solo engine InnoDB o DBD.

Transazioni: savepoint

All'interno di una transazione è anche possibile stabilire dei **savepoint**, cioè degli stati intermedi ai quali possiamo ritornare con una ROLLBACK, invece di annullare interamente la transazione.

```
START TRANSACTION
    ...istruzioni di aggiornamento (1)...
SAVEPOINT sp1;
    ...istruzioni di aggiornamento (2)...
ROLLBACK TO SAVEPOINT sp1;
    ...istruzioni di aggiornamento (3)...
COMMIT
```

Transazioni: savepoint

L'istruzione *ROLLBACK TO SAVEPOINT sp1* fa sì che "ritorniamo" alla situazione esistente quando abbiamo creato il savepoint.

Solo il secondo blocco di aggiornamenti viene annullato, e la transazione rimane aperta; una semplice ROLLBACK invece avrebbe annullato tutto e chiuso la transazione.

La COMMIT effettuata dopo il terzo blocco fa sì che vengano consolidati gli aggiornamenti effettuati nel primo e nel terzo blocco.

Transazioni: FOR UPDATE

La clausola FOR UPDATE stabilisce un lock su tutte le righe lette che impedirà ad altri utenti di leggere le stesse righe fino al termine della nostra transazione:

SELECT … FOR UPDATE;

si utilizza quando leggiamo un dato con l'intenzione di aggiornarlo.

Transazioni: LOCK IN SHARE MODE

La clausola LOCK IN SHARE MODE invece stabilisce un lock che impedisce solo gli aggiornamenti, garantendoci che il contenuto della riga rimarrà invariato per la durata della nostra transazione.

SELECT … LOCK IN SHARE MODE;

Transazioni: livelli di isolamento

Quando viene avviato il server mysqld, è possibile definire il livello di isolamento:

c:\> mysqld --transaction-isolation=livello

Transazioni: livelli di isolamento

I livelli di isolamento sono quattro:

READ UNCOMMITTED: a questo livello sono visibili gli aggiornamenti effettuati da altri utenti **anche se non consolidati.** Questo livello può dare ovviamente seri problemi di consistenza dei dati; si usa quando occorre velocizzare le letture.

Transazioni: livelli di isolamento

READ COMMITTED: a questo livello gli aggiornamenti diventano visibili solo dopo il consolidamento.

REPEATEABLE READ (Default): in questo caso affinché un aggiornamento diventi visibile deve essere consolidato e la transazione che legge deve essere terminata; in pratica, la stessa lettura ripetuta all'interno di una transazione darà sempre lo stesso risultato a prescindere dalle modifiche subite.

Transazioni: livelli di isolamento

SERIALIZABLE: come nel caso precedente, ma in più, la semplice lettura di un dato provoca il blocco degli aggiornamenti fino al termine della transazione; in sostanza è come se ogni SELECT venisse effettuata con la clausola LOCK IN SHARE MODE.

Transazioni: livelli di isolamento

Per conoscere il livello corrente di isolamento usare il comando **SELECT @@tx_isolation.**

Per modificare il livello corrente di isolamento usare il comando SET:

SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL

{ READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }

Transazioni: livelli di isolamento

Se omettiamo le clausole GLOBAL e SESSION la modifica è valida solo per la transazione successiva; con SESSION impostiamo il valore per l'intera connessione, mentre con GLOBAL modifichiamo il valore per il server: tale valore verrà quindi adottato su tutte le connessioni aperte successivamente (**non** su quelle già aperte); in quest'ultimo caso è necessario il privilegio SUPER.

Organizzazione fisica e gestione delle interrogazioni

Capitolo 11 Atzeni-Ceri

DataBase Management System — DBMS

Sistema (prodotto software) in grado di gestire collezioni di dati che siano (anche):

- **grandi** (di dimensioni (molto) maggiori della memoria centrale dei sistemi di calcolo utilizzati)
- **persistenti** (con un periodo di vita indipendente dalle singole esecuzioni dei programmi che le utilizzano)
- **condivise** (utilizzate da applicazioni diverse)

garantendo **affidabilità** (resistenza a malfunzionamenti hardware e software) e privatezza (con una disciplina e un controllo degli accessi). Come ogni prodotto informatico, un DBMS deve essere **efficiente** (utilizzando al meglio le risorse di spazio e tempo del sistema) ed efficace (rendendo produttive le attività dei suoi utilizzatori).

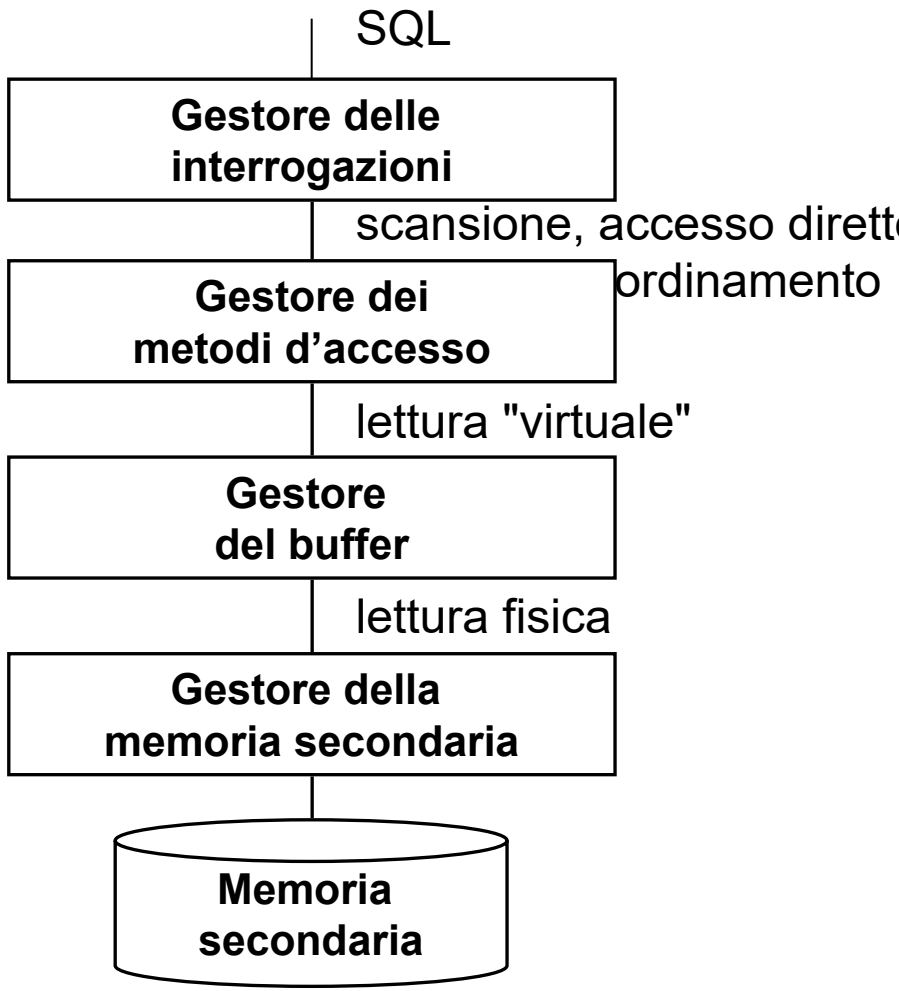
Le basi di dati sono grandi e persistenti

- La persistenza richiede una gestione in memoria secondaria
- La grandezza richiede che tale gestione sia sofisticata (non possiamo caricare tutto in memoria principale e poi riscaricare)

Le basi di dati vengono interrogate …

- Gli utenti vedono il modello logico (relazionale)
- I dati sono in memoria secondaria
- Le strutture logiche non sarebbero efficienti in memoria secondaria:
  - servono strutture fisiche opportune
- La memoria secondaria è molto più lenta della memoria principale:
  - serve un'interazione fra memoria principale e secondaria che limiti il più possibile gli accessi alla secondaria
- Esempio: una interrogazione con un join

Gestore degli accessi e delle interrogazioni



Le basi di dati sono affidabili

- Le basi di dati sono una risorsa per chi le possiede, e debbono essere conservate anche in presenza di malfunzionamenti

- Esempio:
  - un trasferimento di fondi da un conto corrente bancario ad un altro, con guasto del sistema a metà

- Le transazioni debbono essere
  - atomiche (o tutto o niente)
  - definitive: dopo la conclusione, non si dimenticano

Le basi di dati vengono aggiornate …

- L'affidabilità è impegnativa per via degli aggiornamenti frequenti e della necessità di gestire il buffer

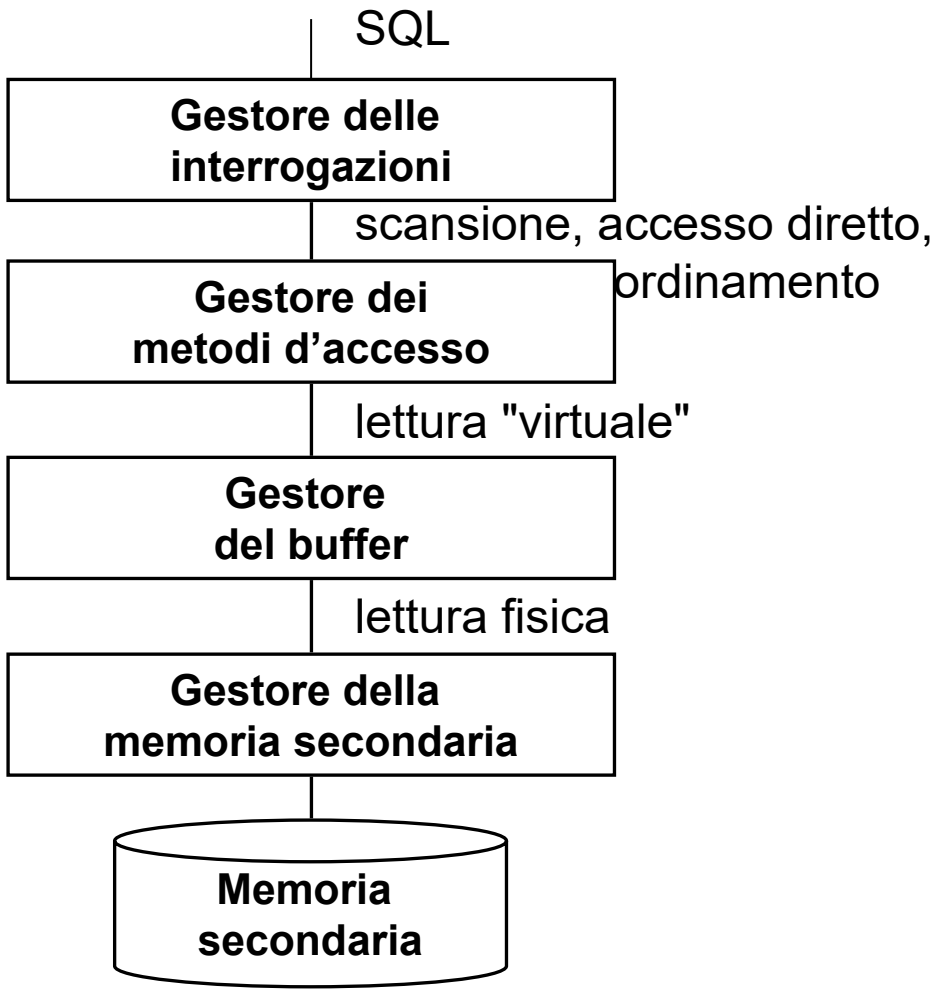
Le basi di dati sono condivise

- Una base di dati è una risorsa **integrata**, **condivisa** fra le varie applicazioni
- conseguenze
  - Attività diverse su dati in parte condivisi:
    - meccanismi di autorizzazione
  - Attività multi-utente su dati condivisi:
    - controllo della **concorrenza**

Aggiornamenti su basi di dati condivise …

- Esempi:
  - due prelevamenti (quasi) contemporanei sullo stesso conto corrente
  - due prenotazioni (quasi) contemporanee sullo posto
- Intutitivamente, le transazioni sono corrette se **seriali** (prima una e poi l'altra)
- Ma in molti sistemi reali l'efficienza sarebbe penalizzata troppo se le transazioni fossero seriali:
  - il **controllo della concorrenza** permette un ragionevole compromesso

Gestore degli accessi e delle interrogazioni



Memoria principale e secondaria

- I programmi possono fare riferimento solo a dati in memoria principale
- Le basi di dati debbono essere (sostanzialmente) in memoria secondaria per due motivi:
  - Dimensioni
  - Persistenza
- I dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale (questo spiega i termini "principale" e "secondaria")

Memoria principale e secondaria, 2

- I dispositivi di memoria secondaria sono organizzati in **blocchi** di lunghezza (di solito) **fissa** (ordine di grandezza: alcuni KB)
- Le uniche operazioni sui dispositivi sono la lettura e la scrittura di una **pagina**, cioè dei dati di un blocco (cioè di una stringa di byte);
- per comodità consideriamo **blocco** e **pagina** sinonimi

Memoria principale e secondaria, 3

- Accesso a memoria secondaria:
  - tempo di posizionamento della testina (10-50ms)
  - tempo di latenza (5-10ms)
  - tempo di trasferimento (1-2ms)

  in media non meno di 10 ms
- Il costo di un accesso a memoria secondaria è quattro o più ordini di grandezza maggiore di quello per operazioni in memoria centrale
- Perciò, nelle applicazioni "I/O bound" (cioè con molti accessi a memoria secondaria e relativamente poche operazioni) il costo dipende esclusivamente dal numero di accessi a memoria secondaria
- Inoltre, accessi a blocchi “vicini” costano meno (contiguità)

Buffer management

- **Buffer**:
  - area di memoria centrale, gestita dal DBMS (preallocata) e condivisa fra le transazioni
  - organizzato in **pagine** di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (1KB-100KB)
  - è importantissimo per via della grande differenza di tempo di accesso fra memoria centrale e memoria secondaria

Scopo della gestione del buffer

- Ridurre il numero di accessi alla memoria secondaria
  - In caso di lettura, se la pagina è già presente nel buffer, non è necessario accedere alla memoria secondaria
  - In caso di scrittura, il gestore del buffer può decidere di differire la scrittura fisica (ammesso che ciò sia compatibile con la gestione dell'affidabilità – vedremo più avanti)
- La gestione dei buffer e la differenza di costi fra memoria principale e secondaria possono suggerire algoritmi innovativi.
- Esempio:
  - File di 10.000.000 di record di 100 byte ciascuno (1GB)
  - Blocchi di 4KB
  - Buffer disponibile di 20M

  Come possiamo fare l'ordinamento?
  - Merge-sort "a più vie"

Dati gestiti dal buffer manager

- Il buffer
- Un direttorio che per ogni pagina mantiene (ad esempio)
  - il file fisico e il numero del blocco
  - due variabili di stato:
    - un contatore che indica quanti programmi utilizzano la pagina
    - un bit che indica se la pagina è “sporca”, cioè se è stata modificata

Funzioni del buffer manager

- Intuitivamente:
  - riceve richieste di lettura e scrittura (di pagine)
  - le esegue accedendo alla memoria secondaria solo quando indispensabile e utilizzando invece il buffer quando possibile
  - esegue le primitive
    - ***fix***, ***unfix***, ***setDirty, force***.

- Le politiche sono simili a quelle relative alla gestione della memoria da parte dei sistemi operativi;

  - "località dei dati": è alta la probabilità di dover riutilizzare i dati attualmente in uso

  - "legge 80-20" l'80% delle operazioni utilizza sempre lo stesso 20% dei dati

Interfaccia offerta dal buffer manager

- ***fix:*** richiesta di una pagina; richiede una lettura solo se la pagina non è nel buffer (incrementa il contatore associato alla pagina)
- ***setDirty:*** comunica al buffer manager che la pagina è stata modificata
- ***unfix:*** indica che la transazione ha concluso l'utilizzo della pagina (decrementa il contatore associato alla pagina)
- ***force***: trasferisce in modo sincrono una pagina in memoria secondaria (su richiesta del gestore dell'affidabilità, non del gestore degli accessi)

Esecuzione della fix

- Cerca la pagina nel buffer;
  - se c'è, restituisce l'indirizzo
  - altrimenti, cerca una pagina libera nel buffer (contatore a zero);
    - se la trova, restituisce l'indirizzo
    - altrimenti, due alternative
      - "**steal**": selezione di una "vittima", pagina occupata del buffer; I dati della vittima sono scritti in memoria secondaria; viene letta la pagina di interesse dalla memoria secondaria e si restituisce l'indirizzo
      - "**no-steal**": l'operazione viene posta in attesa

Commenti

- Il buffer manager richiede scritture in due contesti diversi:
  - in modo **sincrono** quando è richiesto esplicitamente con una force
  - in modo **asincrono** quando lo ritiene opportuno (o necessario); in particolare, può decidere di anticipare o posticipare scritture per coordinarle e/o sfruttare la disponibilità dei dispositivi

DBMS e file system

- Il file system è il componente del sistema operativo che gestisce la memoria secondaria
- I DBMS ne utilizzano le funzionalità, ma in misura limitata, per creare ed eliminare file e per leggere e scrivere singoli blocchi o sequenze di blocchi contigui.
- L'organizzazione dei file, sia in termini di distribuzione dei record nei blocchi sia relativamente alla struttura all'interno dei singoli blocchi è gestita direttamente dal DBMS.

DBMS e file system, 2

- Il DBMS gestisce i blocchi dei file allocati come se fossero un unico grande spazio di memoria secondaria e costruisce, in tale spazio, le strutture fisiche con cui implementa le relazioni.
- Il DBMS crea file di grandi dimensioni che utilizza per memorizzare diverse relazioni (al limite, l'intero database)
- Talvolta, vengono creati file in tempi successivi:
  - è possibile che un file contenga i dati di più relazioni e che le varie tuple di una relazione siano in file diversi.
- Spesso, ma non sempre, ogni blocco è dedicato a tuple di un'unica relazione

Blocchi e record

- I blocchi (componenti "fisici" di un file) e i record (componenti "logici") hanno dimensioni in generale diverse:
  - la dimensione del blocco dipende dal file system
  - la dimensione del record (semplificando un po') dipende dalle esigenze dell'applicazione, e può anche variare nell'ambito di un file

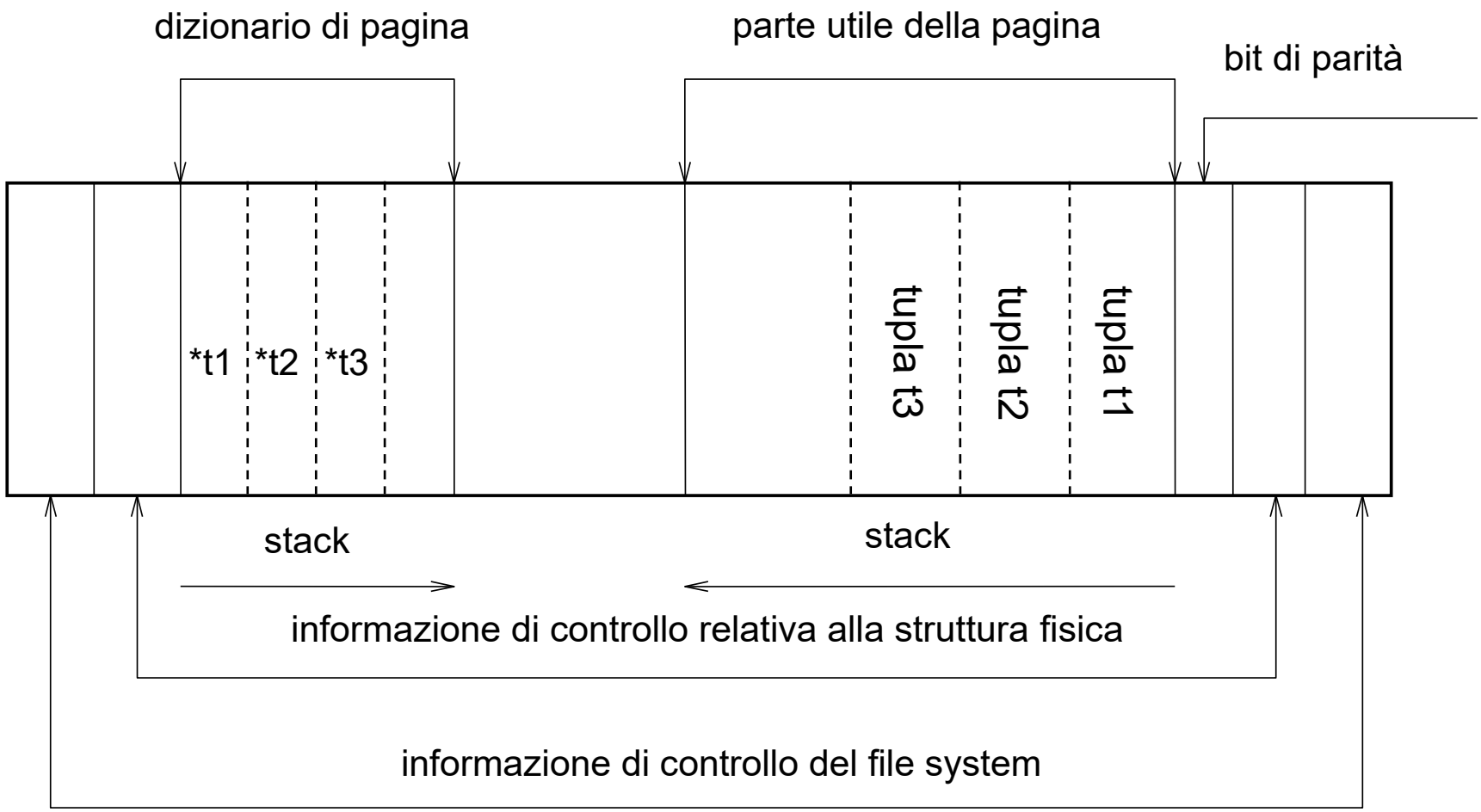
Fattore di blocco

- numero di record in un blocco
  - L_R: dimensione di un record (per semplicità costante nel file: "record a lunghezza fissa")
  - L_B: dimensione di un blocco
  - se $L_B > L_R$, possiamo avere più record in un blocco: $\lfloor L_B / L_R \rfloor$
- lo spazio residuo può essere
  - utilizzato (record "spanned" o impaccati)
  - non utilizzato ("unspanned")

Organizzazione delle tuple nelle pagine

- Ci sono varie alternative, anche legate ai metodi di accesso; vediamo una possibilità
- Inoltre:
  - se la lunghezza delle tuple è fissa, la struttura può essere semplificata
  - alcuni sistemi possono spezzare le tuple su più pagine (necessario per tuple grandi)

Organizzazione delle tuple nelle pagine



Strutture sequenziali

- Esiste un ordinamento fra le tuple, che può essere rilevante ai fini della gestione
  - **seriale***:* ordinamento fisico ma non logico
  - **array***:* posizioni individuate attraverso indici
  - **ordinata**: l'ordinamento delle tuple coerente con quello di un campo

Struttura seriale

- Chiamata anche:
  - "Entry sequenced"
  - file heap
  - file disordinato
- È molto diffusa nelle basi di dati relazionali, associata a indici secondari
- Gli inserimenti vengono effettuati
  - in coda (con riorganizzazioni periodiche)
  - al posto di record cancellati

Strutture ordinate

- Permettono ricerche binarie, ma solo fino ad un certo punto (ad esempio, come troviamo la "metà del file"?)
- Nelle basi di dati relazionali si utilizzano quasi solo in combinazione con indici (file ISAM o file ordinati con indice primario)

File hash

- Permettono un accesso diretto molto efficiente (da alcuni punti di vista)
- La tecnica si basa su quella utilizzata per le tavole hash in memoria centrale

File hash, osservazioni

- È l'organizzazione più efficiente per l'accesso diretto basato su valori della chiave con condizioni di uguaglianza (accesso puntuale): costo medio di poco superiore all'unità (il caso peggiore è molto costoso ma talmente improbabile da poter essere ignorato)
- Le collisioni (overflow) sono di solito gestite con blocchi collegati
- Non è efficiente per ricerche basate su intervalli (né per ricerche basate su altri attributi)
- I file hash "degenerano" se si riduce lo spazio sovrabbondante: funzionano solo con file la cui dimensione non varia molto nel tempo

Indici di file

- Indice:
  - struttura ausiliaria per l'accesso (efficiente) ai record di un file sulla base dei valori di un campo (o di una "concatenazione di campi") detto chiave (o, meglio, pseudochiave, perché non è necessariamente identificante);
- Idea fondamentale: l'indice analitico di un libro: lista di coppie (termine,pagina), ordinata alfabeticamente sui termini, posta in fondo al libro e separabile da esso
- Un indice I di un file f è un altro file, con record a due campi: chiave e indirizzo (dei record di f o dei relativi blocchi), ordinato secondo i valori della chiave

Tipi di indice

- indice primario:
  - su un campo sul cui ordinamento è basata la memorizzazione
- indice secondario
  - su un campo con ordinamento diverso da quello di memorizzazione
- indice denso:
  - contiene un record per ciascun valore del campo chiave
- indice sparso:
  - contiene un numero di record inferiore rispetto al numero di valori diversi del campo chiave

Tipi di indice, commenti

- Un indice primario può essere sparso, uno secondario deve essere denso
- Esempio, sempre rispetto ad un libro
  - indice generale
  - indice analitico
- I benefici legati alla presenza di indici secondari sono molto più sensibili
- Ogni file può avere al più un indice primario e un numero qualunque di indici secondari (su campi diversi).

Caratteristiche degli indici

- Accesso diretto (sulla chiave) efficiente, sia puntuale sia per intervalli
- Scansione sequenziale ordinata efficiente
  - Tutti gli indici (in particolare quelli secondari) forniscono un **ordinamento logico** sui record del file; con numero di accessi pari al numero di record del file (a parte qualche beneficio dovuto alla bufferizzazione)
- Modifiche della chiave, inserimenti, eliminazioni inefficienti (come nei file ordinati)
  - tecniche per alleviare i problemi:
    - file o blocchi di overflow
    - marcatura per le eliminazioni
    - riempimento parziale
    - blocchi collegati (non contigui)
    - riorganizzazioni periodiche

Indici secondari, due osservazioni

- Si possono usare, come detto, puntatori ai blocchi oppure puntatori ai record
  - I puntatori ai blocchi sono più compatti
  - I puntatori ai record permettono di semplificare alcune operazioni (effettuate solo sull'indice, senza accedere al file se non quando indispensabile)

Indici

- Tutte le strutture di indice viste finora sono basate su strutture ordinate e quindi sono poco flessibili in presenza di elevata dinamicità
- Gli indici utilizzati dai DBMS sono più sofisticati:
  - indici dinamici multilivello: B-tree (alberi di ricerca bilanciati)

Strutture fisiche nei DBMS relazionali

- Struttura primaria:
  - disordinata (heap, "unclustered")
  - ordinata ("clustered"), anche su una pseudochiave
  - hash ("clustered"), anche su una pseudochiave, senza ordinamento
  - clustering di più relazioni
- Indici (densi/sparsi, semplici/composti):
  - ISAM (statico), di solito su struttura ordinata
  - B-tree (dinamico)

Strutture fisiche in alcuni DBMS

- Oracle:
  - struttura primaria
    - file heap
    - "hash cluster" (cioè struttura hash)
    - cluster (anche plurirelazionali) anche ordinati (con B-tree denso)
  - indici secondari di vario tipo (B-tree, bit-map, funzioni)
- DB2:
  - primaria: heap o ordinata con B-tree denso
  - indice sulla chiave primaria (automaticamente)
  - indici secondari B-tree densi
- SQL Server:
  - primaria: heap o ordinata con indice B-tree sparso
  - indici secondari B-tree densi
- MySQL
  - primaria: heap o ordinata con B-tree denso o hash
  - indice sulla chiave primaria (automatico)
  - indici secondari B-tree densi o hash

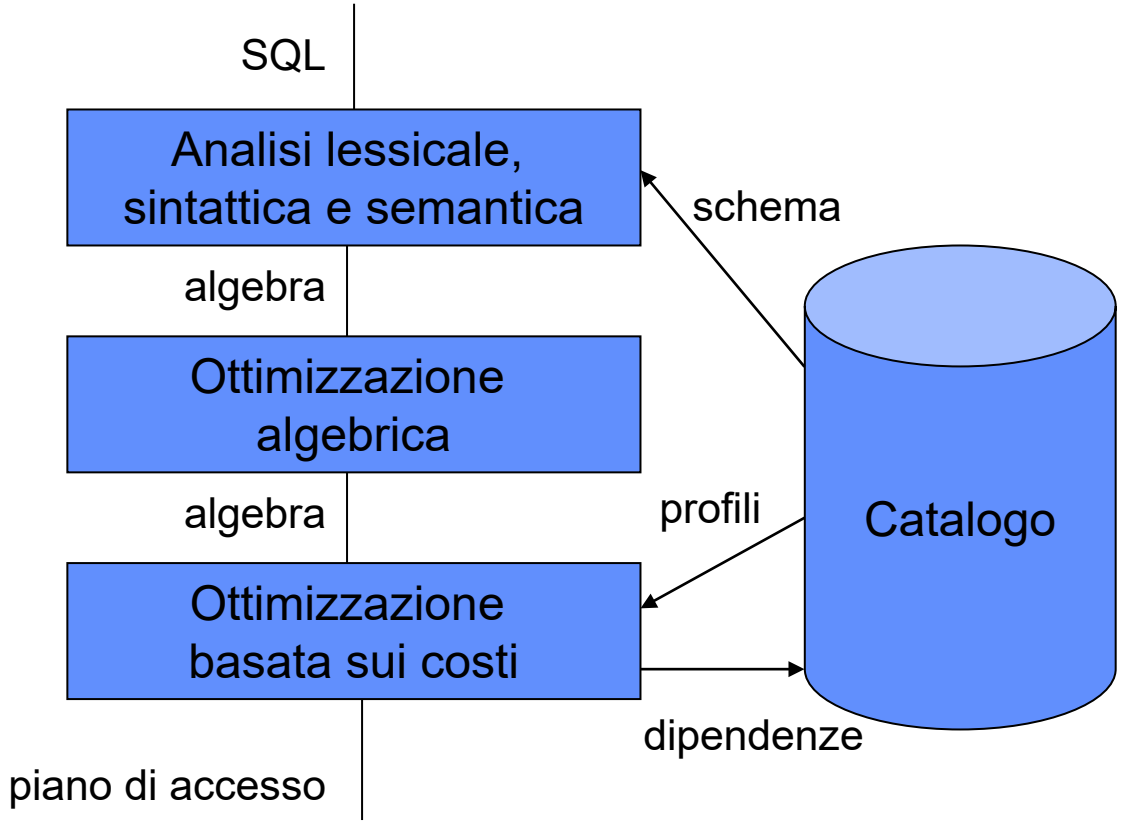
Definizione degli indici in SQL

- Non è standard, ma presente in forma simile nei vari DBMS
  - `create` [`unique`] `index` *IndexName* `on` *TableName*(*AttributeLis*t)
  - `drop index` *IndexName*

Esecuzione e ottimizzazione delle interrogazioni

- ***Query processor*** (o ***Ottimizzatore***): un modulo del DBMS
- Più importante nei sistemi attuali che in quelli "vecchi" (gerarchici e reticolari):
  - le interrogazioni sono espresse ad alto livello (ricordare il concetto di **indipendenza dei dati**):
    - insiemi di tuple
    - poca proceduralità
  - l'ottimizzatore sceglie la strategia realizzativa (di solito fra diverse alternative), a partire dall'istruzione SQL

Il processo di esecuzione delle interrogazioni



"Profili" delle relazioni

- Informazioni quantitative:
  - cardinalità di ciascuna relazione
  - dimensioni delle tuple
  - dimensioni dei valori
  - numero di valori distinti degli attributi
  - valore minimo e massimo di ciascun attributo
- Sono memorizzate nel "catalogo" e aggiornate con comandi del tipo `update statistics`
- Utilizzate nella fase finale dell'ottimizzazione, per stimare le dimensioni dei risultati intermedi

Esecuzione delle operazioni

- I DBMS implementano gli operatori dell'algebra relazionale (o meglio, loro combinazioni) per mezzo di operazioni di livello abbastanza basso, che però possono implementare vari operatori "in un colpo solo"
- Operatori fondamentali:
  - scansione
  - accesso diretto
- A livello più alto:
  - ordinamento
- Ancora più alto
  - join

Accesso diretto

- Può essere eseguito solo se le strutture fisiche lo permettono
  - indici
  - strutture hash

Accesso diretto basato su indice

- Efficace per interrogazioni (sulla "chiave dell'indice)
  - "puntuali" ($A_i = v$)
  - su intervallo ($v_1 \leq A_i \leq v_2$)
- Per predicati congiuntivi
  - si sceglie il più selettivo per l'accesso diretto e si verifica poi sugli altri dopo la lettura (e quindi in memoria centrale)
- Per predicati disgiuntivi:
  - servono indici su tutti, ma conviene usarli se molto selettivi e facendo attenzione ai duplicati

Accesso diretto basato su hash

- Efficace per interrogazioni (sulla "chiave dell'indice)
  - "puntuali" ($A_i = v$)
  - NON su intervallo ($v_1 \leq A_i \leq v_2$)
- Per predicati congiuntivi e disgiuntivi, vale lo stesso discorso fatto per gli indici

Indici e hash su più campi

- Indice su cognome e nome
  - funziona per accesso diretto su cognome?
  - funziona per accesso diretto su nome?
- Hash su cognome e nome
  - funziona per accesso diretto su cognome?
  - funziona per accesso diretto su nome?

Join

- L'operazione più costosa
- Vari metodi; i più noti:
  - *nested-loo*p, *merge-scan* and *hash-based*

Nested-loop

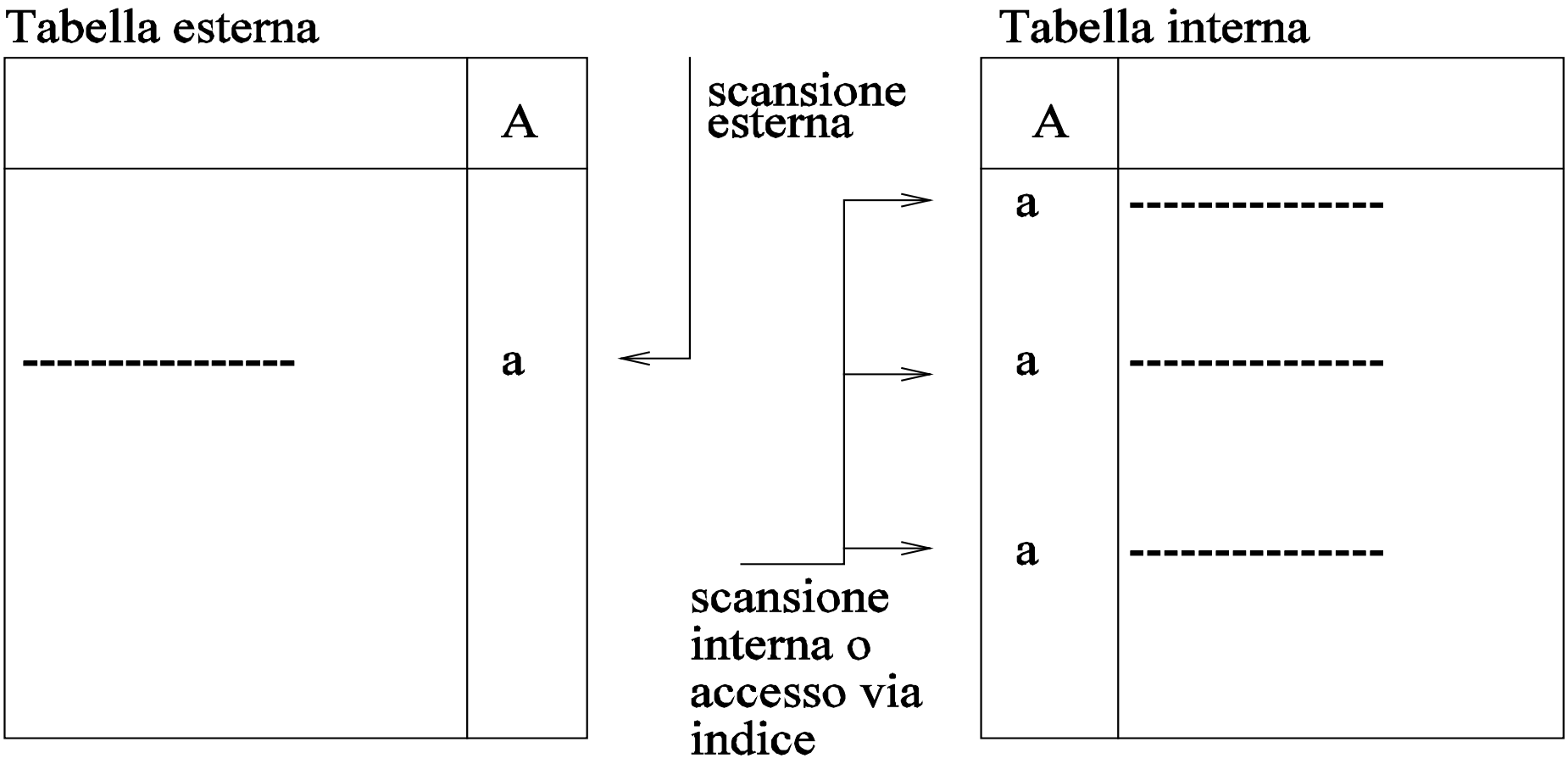


Tabella esterna
A
a
Tabella interna
A
a
a
a
scansione esterna
scansione interna o accesso via indice

Merge-scan

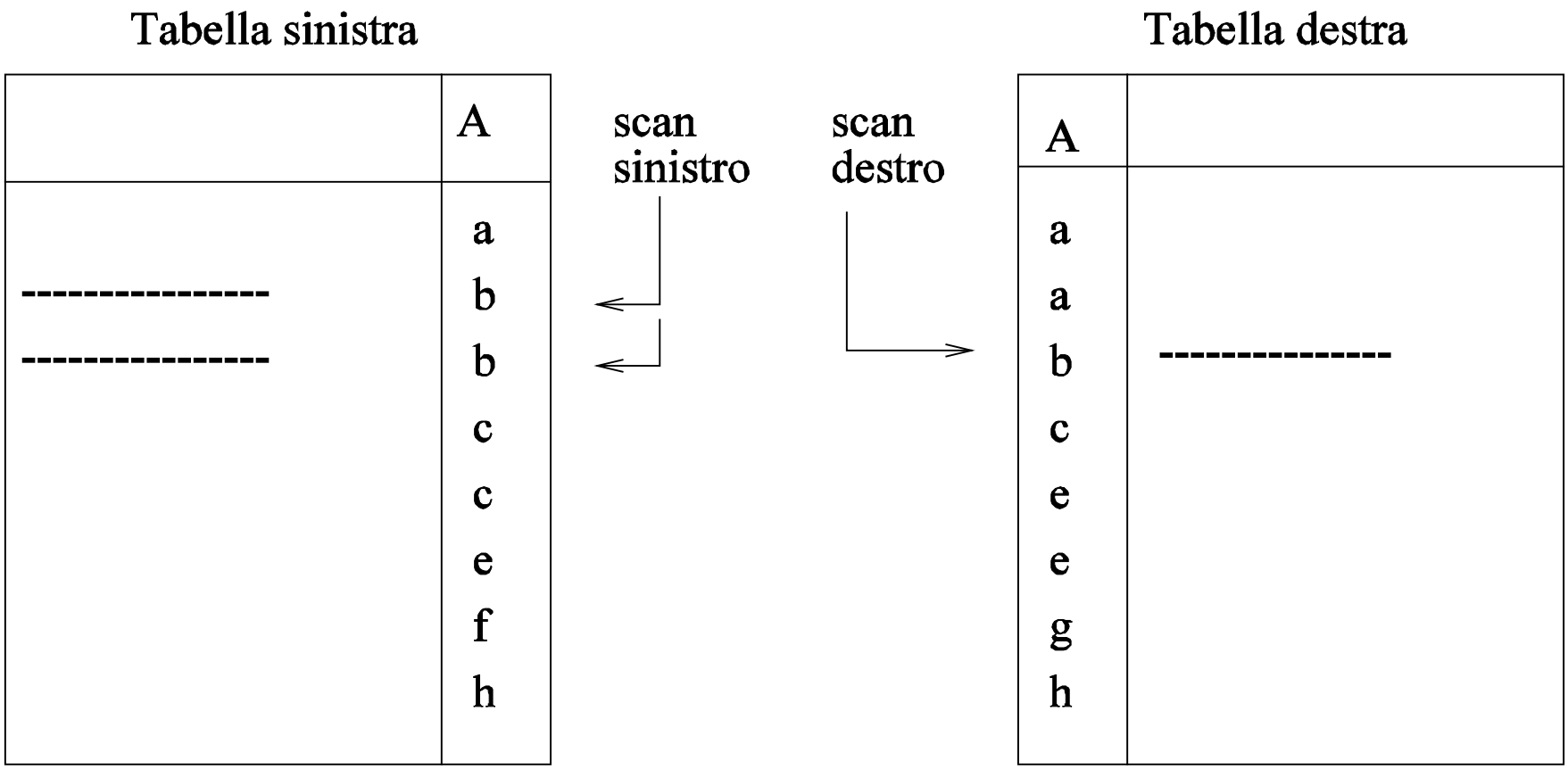
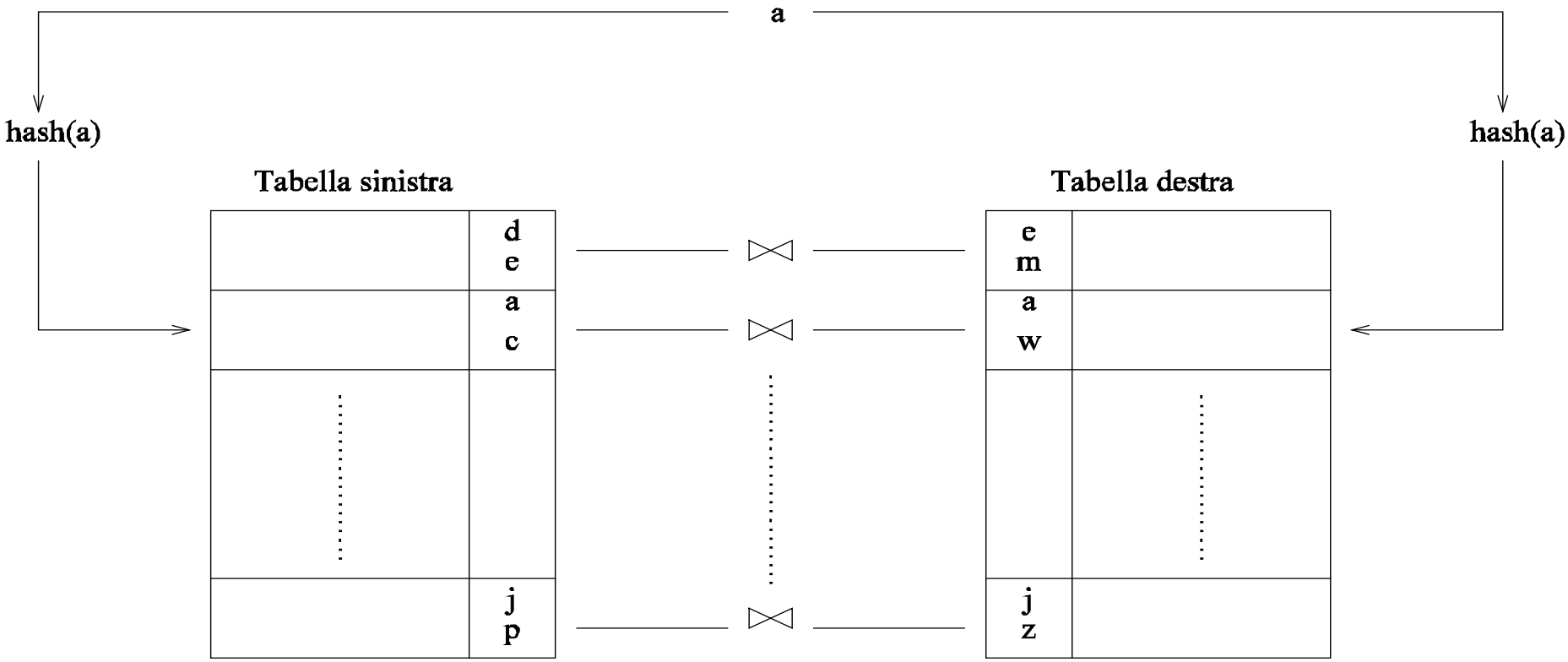


Tabella sinistra
A
a
b
b
c
c
e
f
h
scan
sinistro
scan
destro
Tabella destra
A
a
a
b
c
e
e
g
h

Hash join


a
hash(a)
hash(a)
Tabella sinistra
Tabella destra
d
e
a
c
j
p
e
m
a
w
j
z

Ottimizzazione basata sui costi

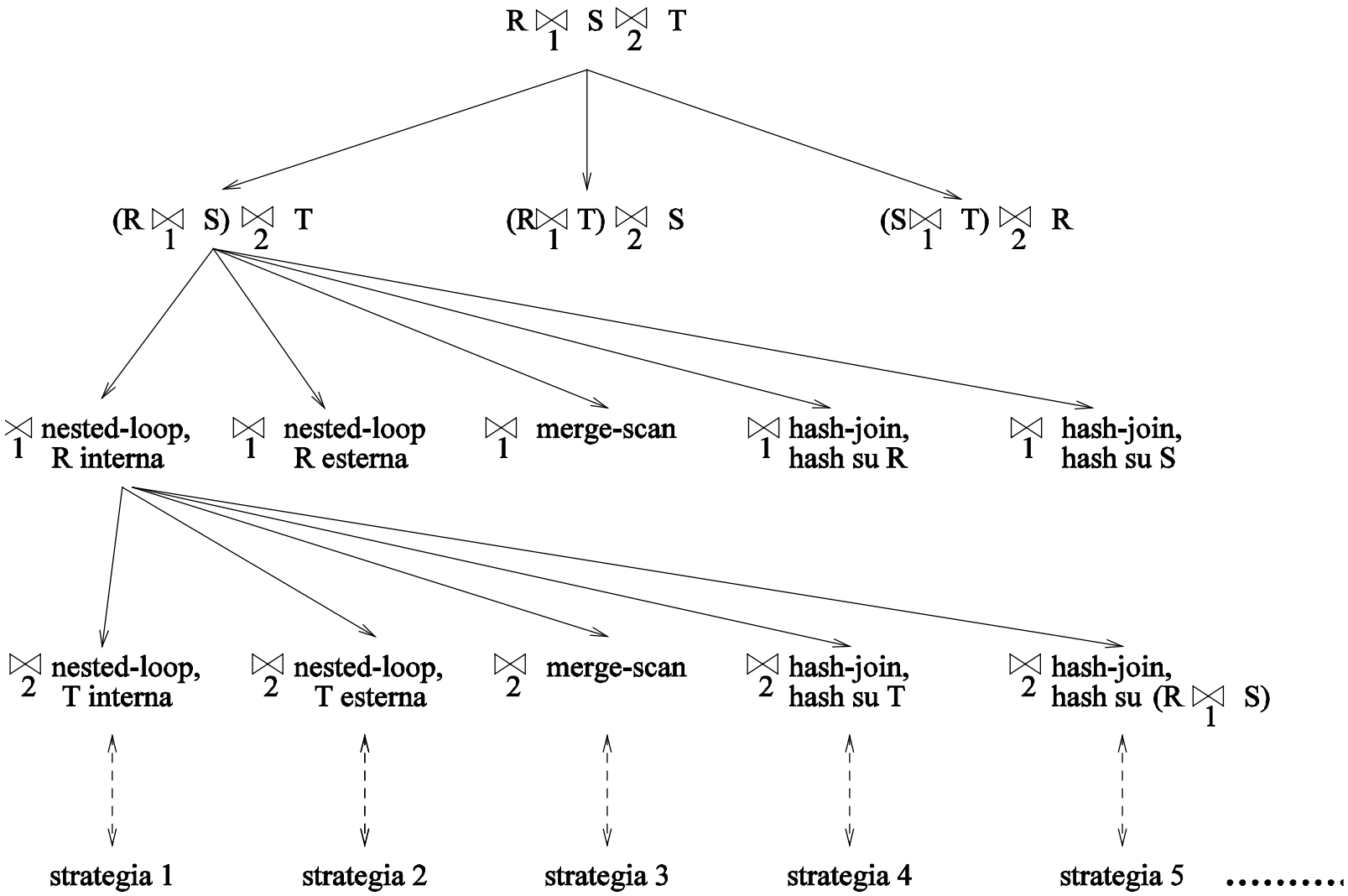
- Un problema articolato, con scelte relative a:
  - operazioni da eseguire (es.: scansione o accesso diretto?)
  - ordine delle operazioni (es. join di tre relazioni; ordine?)
  - i dettagli del metodo (es.: quale metodo di join)

Il processo di ottimizzazione

- Si costruisce un albero di decisione con le varie alternative ("**piani di esecuzione**")
- Si valuta il costo di ciascun piano
- Si sceglie il piano di costo minore

- L'ottimizzatore trova di solito una "buona" soluzione, non necessarimante l"ottimo"

Un albero di decisione



Progettazione fisica

- La fase finale del processo di progettazione di basi di dati
- input
  - lo schema logico e informazioni sul carico applicativo
- output
  - schema fisico, costituito dalle definizione delle relazioni con le relative strutture fisiche (e molti parametri, spesso legati allo specifico DBMS)

Progettazione fisica nel modello relazionale

- La caratteristica comune dei DBMS relazionali è la disponibilità degli indici:
  - la progettazione fisica spesso coincide con la scelta degli indici (oltre ai parametri strettamente dipendenti dal DBMS)
- Le chiavi (primarie) delle relazioni sono di solito coinvolte in selezioni e join: molti sistemi prevedono (oppure suggeriscono) di definire indici sulle chiavi primarie
- Altri indici vengono definiti con riferimento ad altre selezioni o join "importanti"
- Se le prestazioni sono insoddisfacenti, si "tara" il sistema aggiungendo o eliminando indici
- È utile verificare se e come gli indici sono utilizzati con il comando SQL `show plan (explain in MySQL)`