Basi di Dati
Architettura dei DBMS: Introduzione

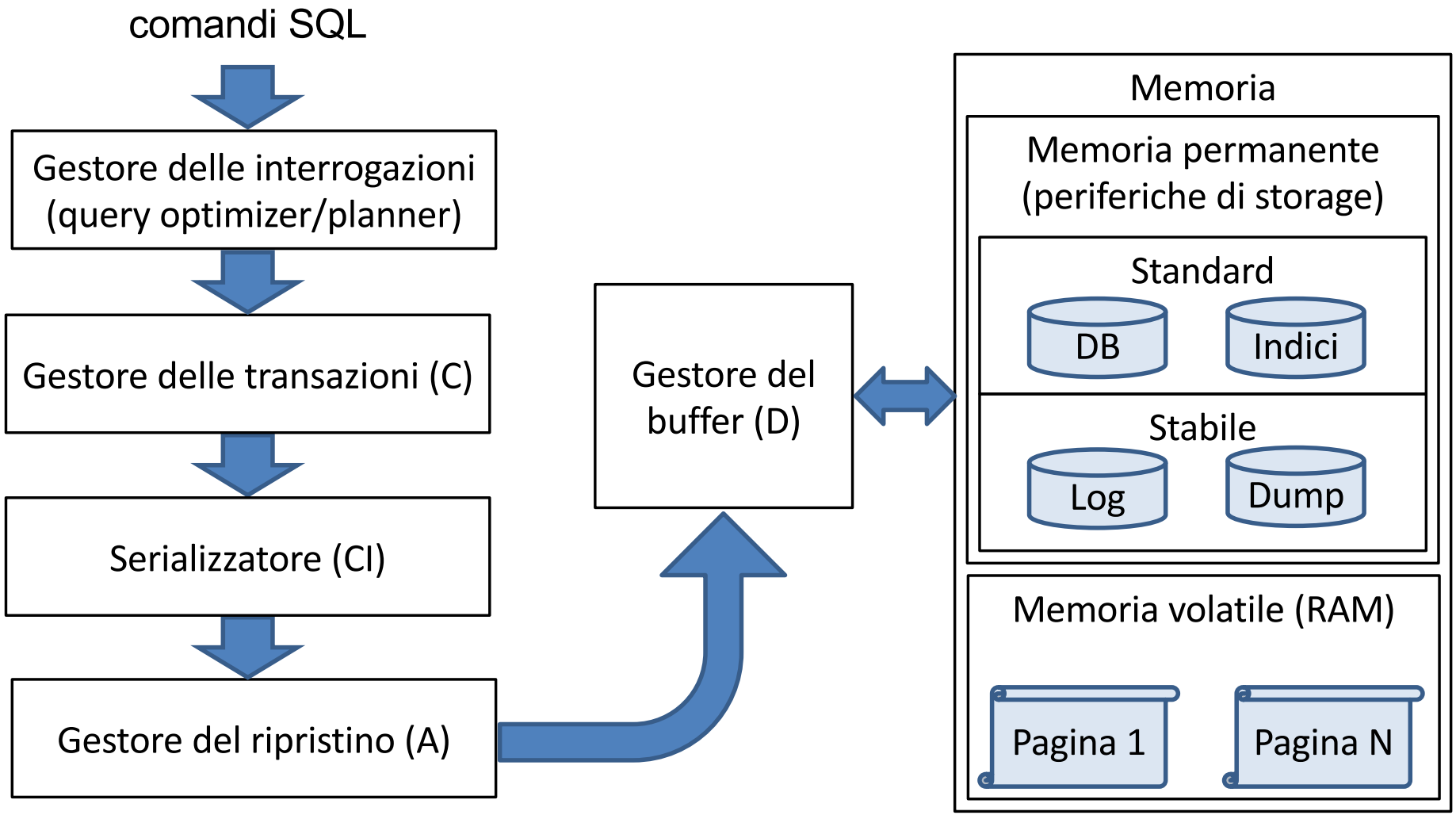
Contenuti - Roadmap

Lab (progettazione)	Corso di Teoria	Lab (SQL)
• Metodologie e modello Entità Associazioni • Progettazione concettuale e logica	• Modello Relazionale • Algebra relazionale • Ottimizzazione logica • Calcolo relazionale • La normalizzazione • **Metodi di accesso e indici** • Gestione della concorrenza • Gestione del ripristino	• Linguaggio SQL

Basi di dati come Tecnologia: perché studiarla?

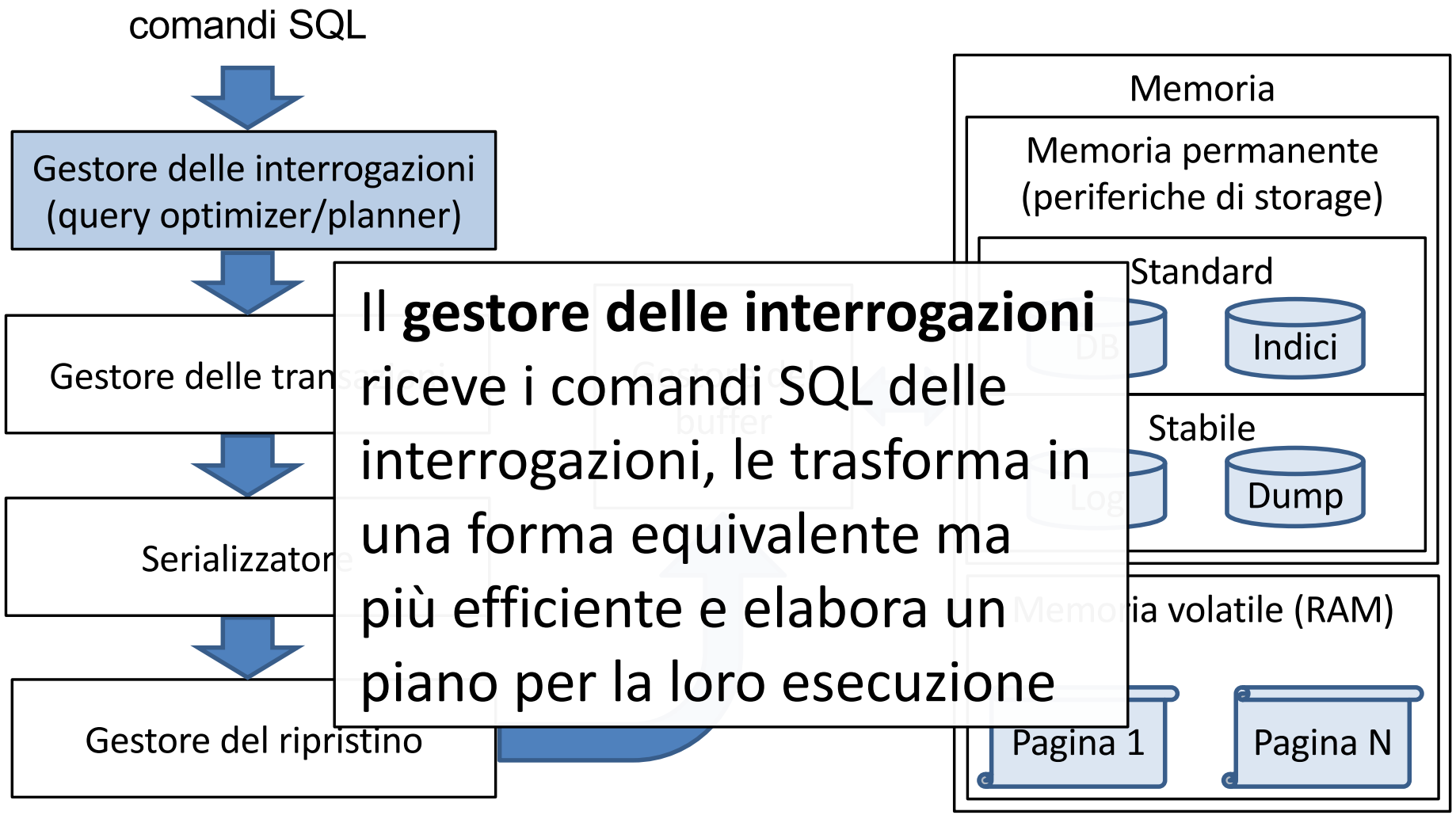
- I DBMS offrono i loro servizi in modo "trasparente":
  - per questo abbiamo potuto finora ignorare molti aspetti realizzativi
  - abbiamo considerato il DBMS come una "scatola nera"
- Perché aprire la scatola?
  - capire come funziona può essere utile per usarla al meglio
  - alcuni servizi sono offerti separatamente

Architettura dei DBMS



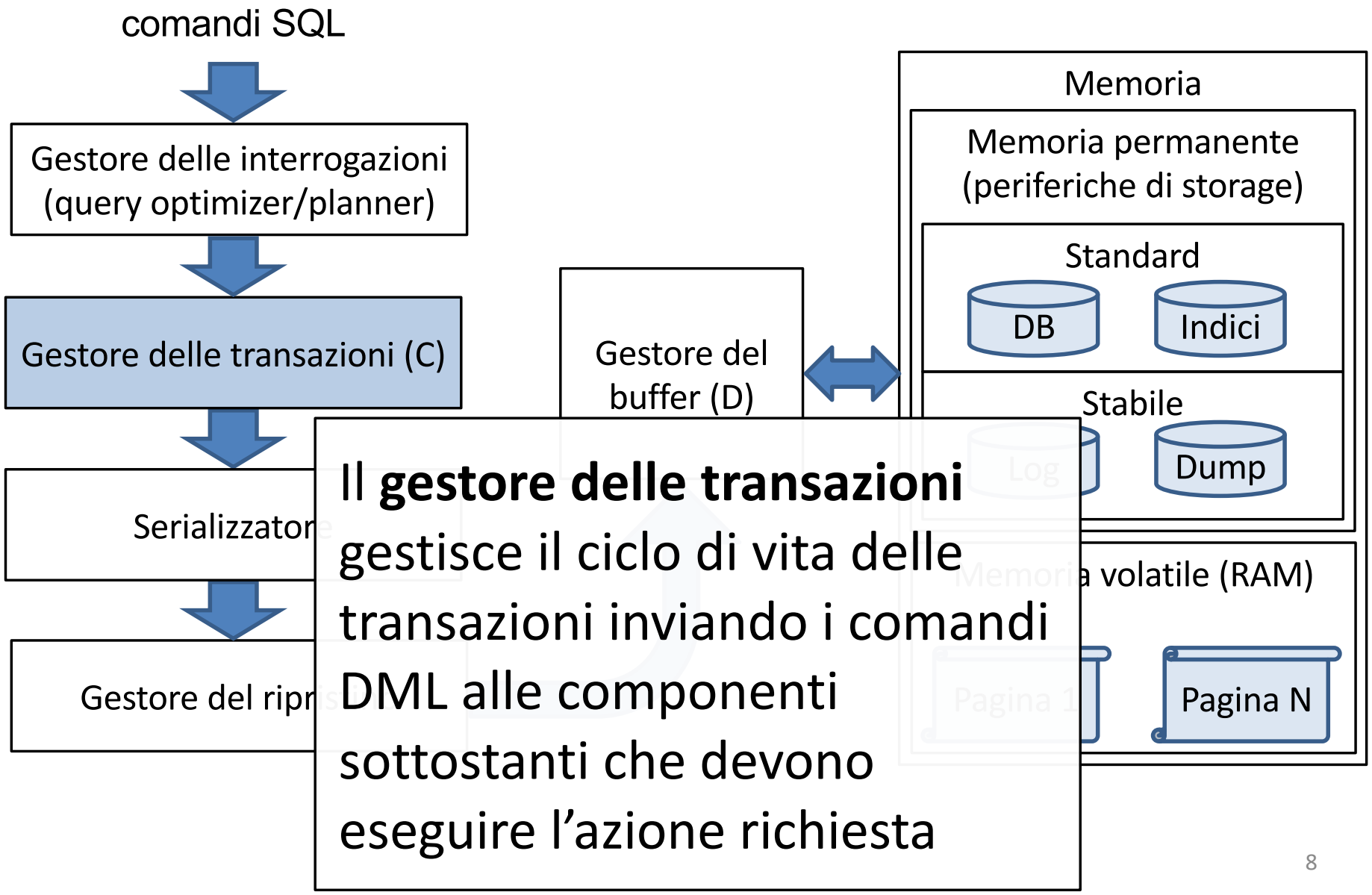
Modello semplificato. Nella realtà le funzionalità non sono così chiaramente distinte.

Architettura dei DBMS



Il **gestore delle interrogazioni** riceve i comandi SQL delle interrogazioni, le trasforma in una forma equivalente ma più efficiente e elabora un piano per la loro esecuzione

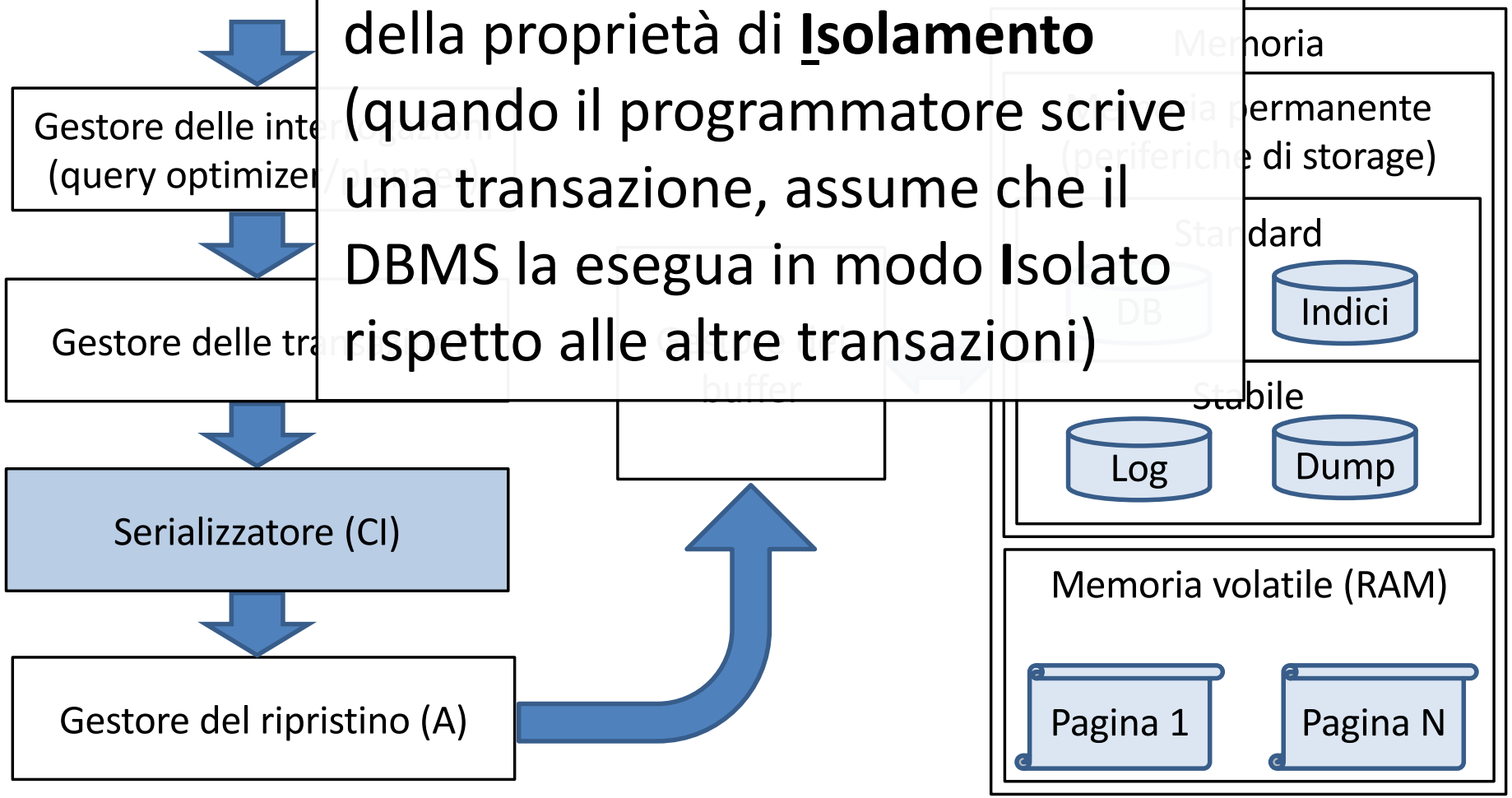
Architettura dei DBMS



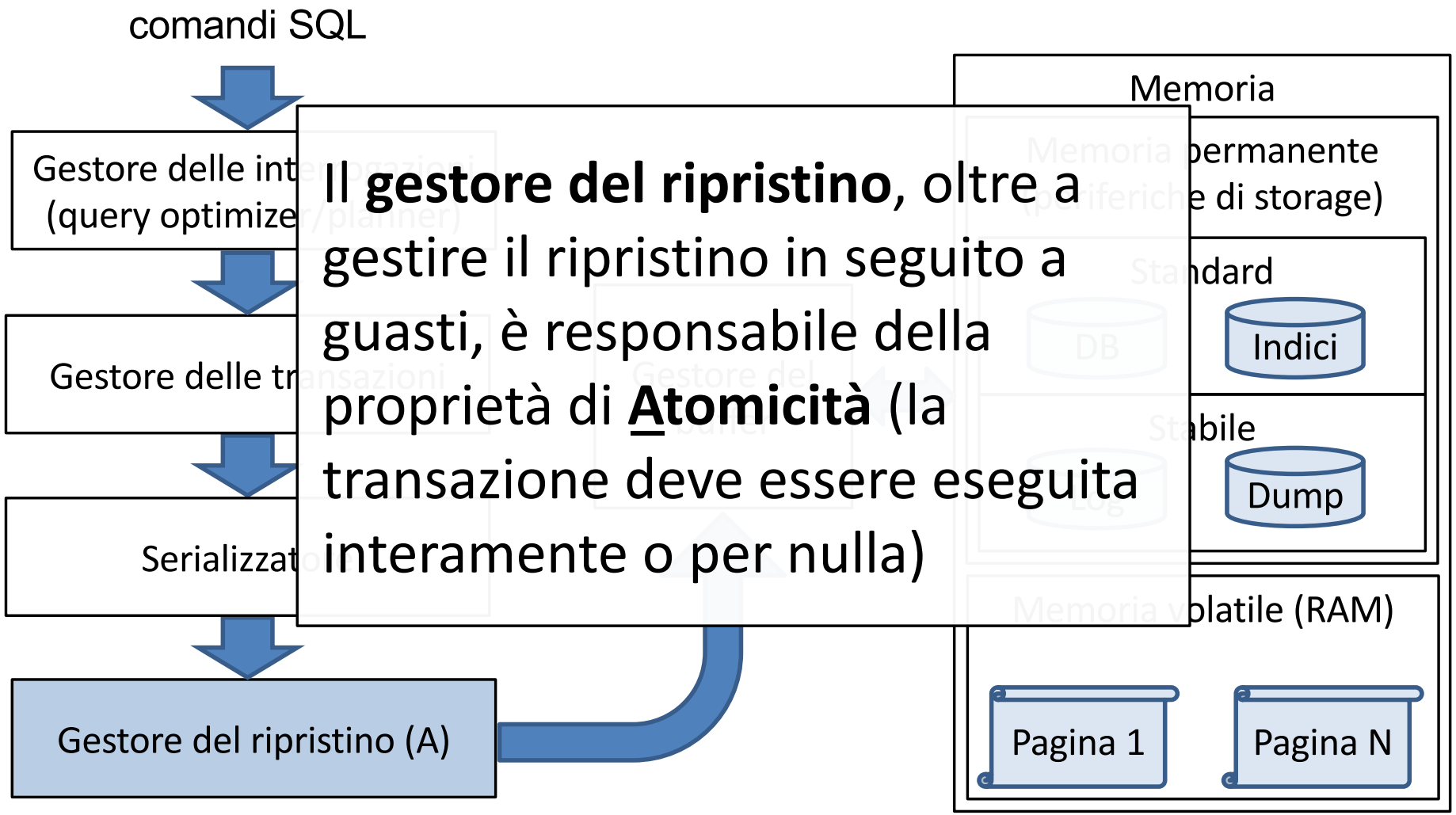
Il **gestore delle transazioni** gestisce il ciclo di vita delle transazioni inviando i comandi DML alle componenti sottostanti che devono eseguire l'azione richiesta

Architettura dei DBMS

Il **serializzatore** è responsabile della proprietà di **I**solamento (quando il programmatore scrive una transazione, assume che il DBMS la esegua in modo **I**solato rispetto alle altre transazioni)

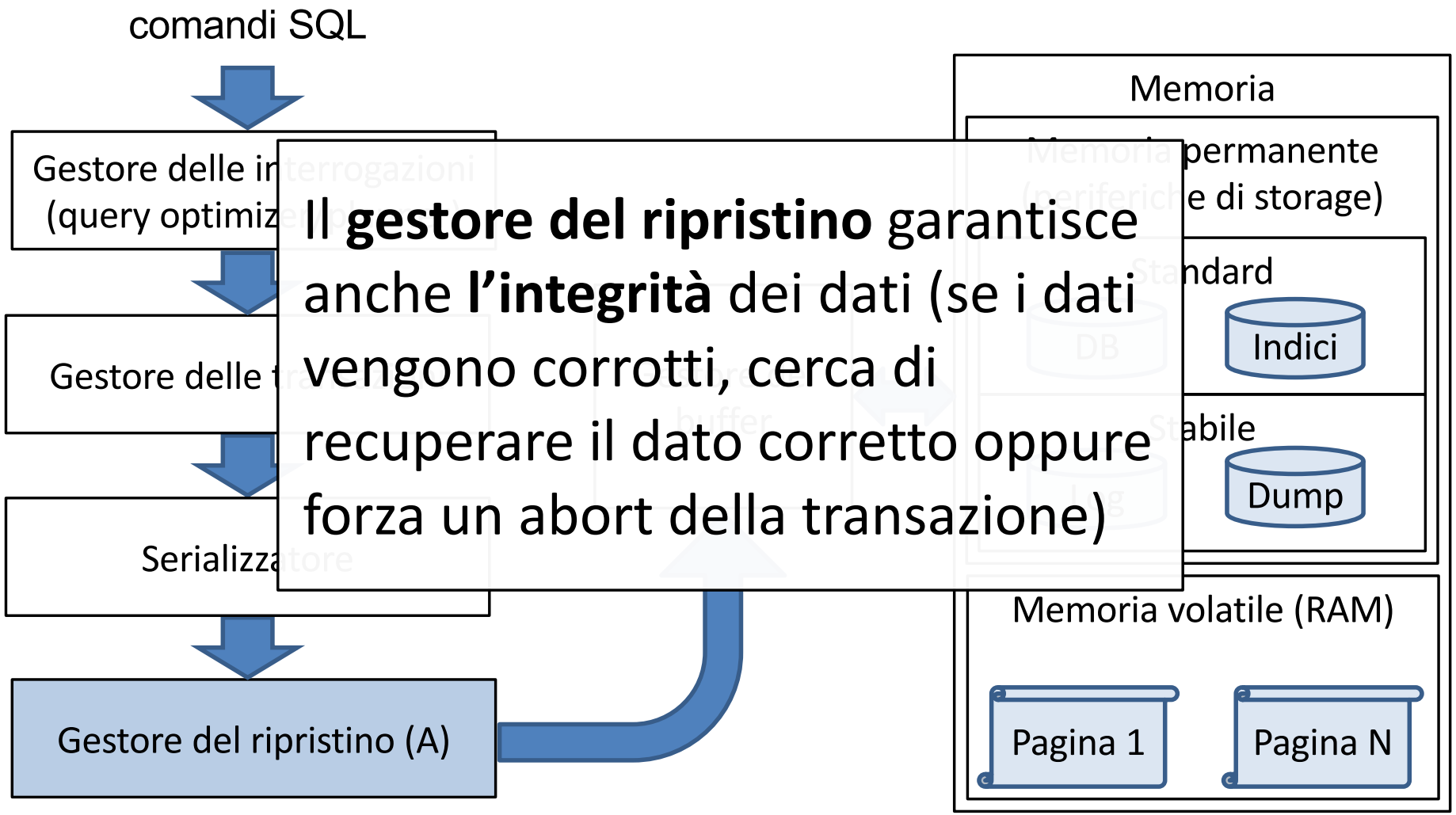


Architettura dei DBMS



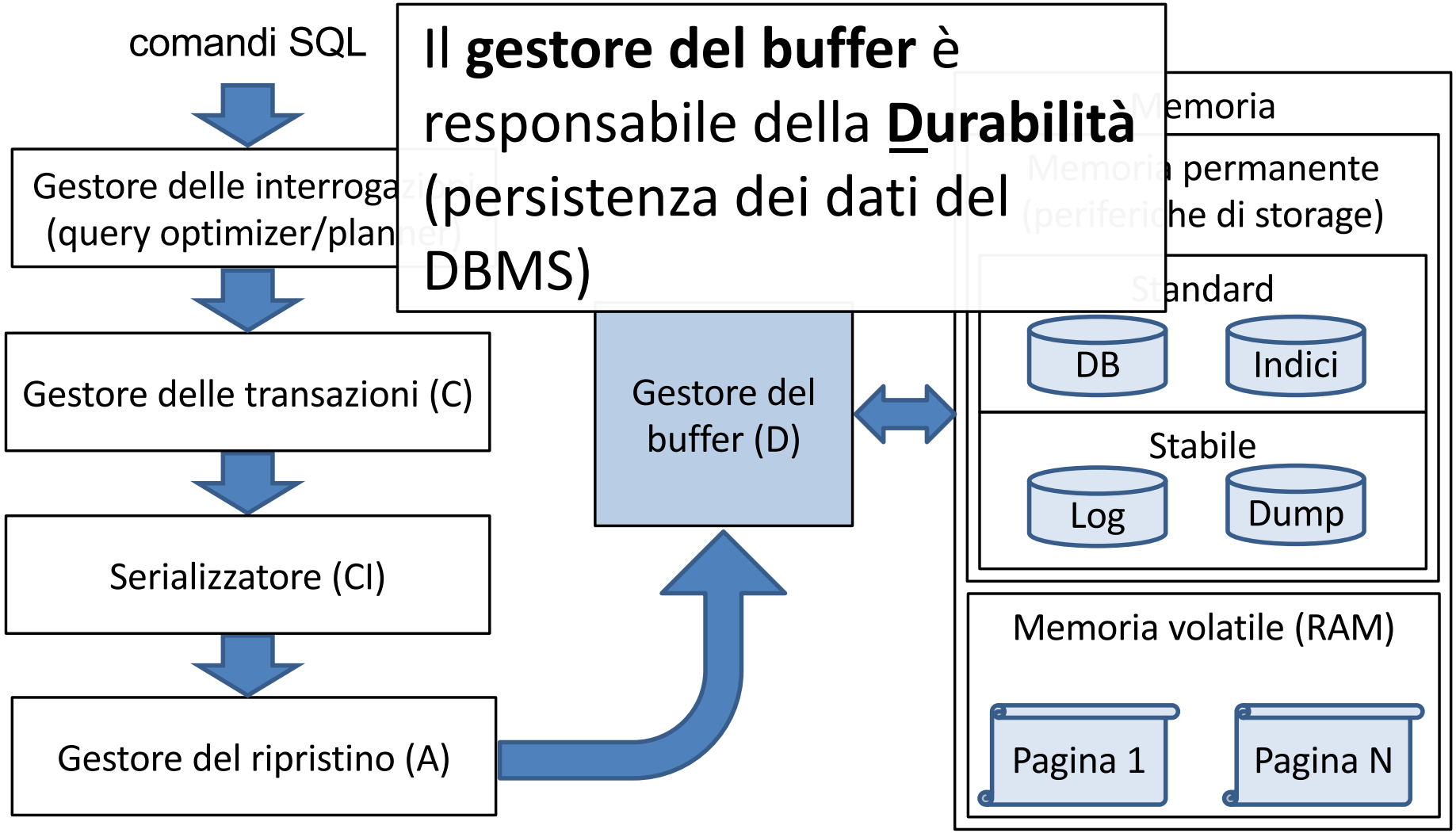
Il **gestore del ripristino**, oltre a gestire il ripristino in seguito a guasti, è responsabile della proprietà di **Atomicità** (la transazione deve essere eseguita interamente o per nulla)

Architettura dei DBMS

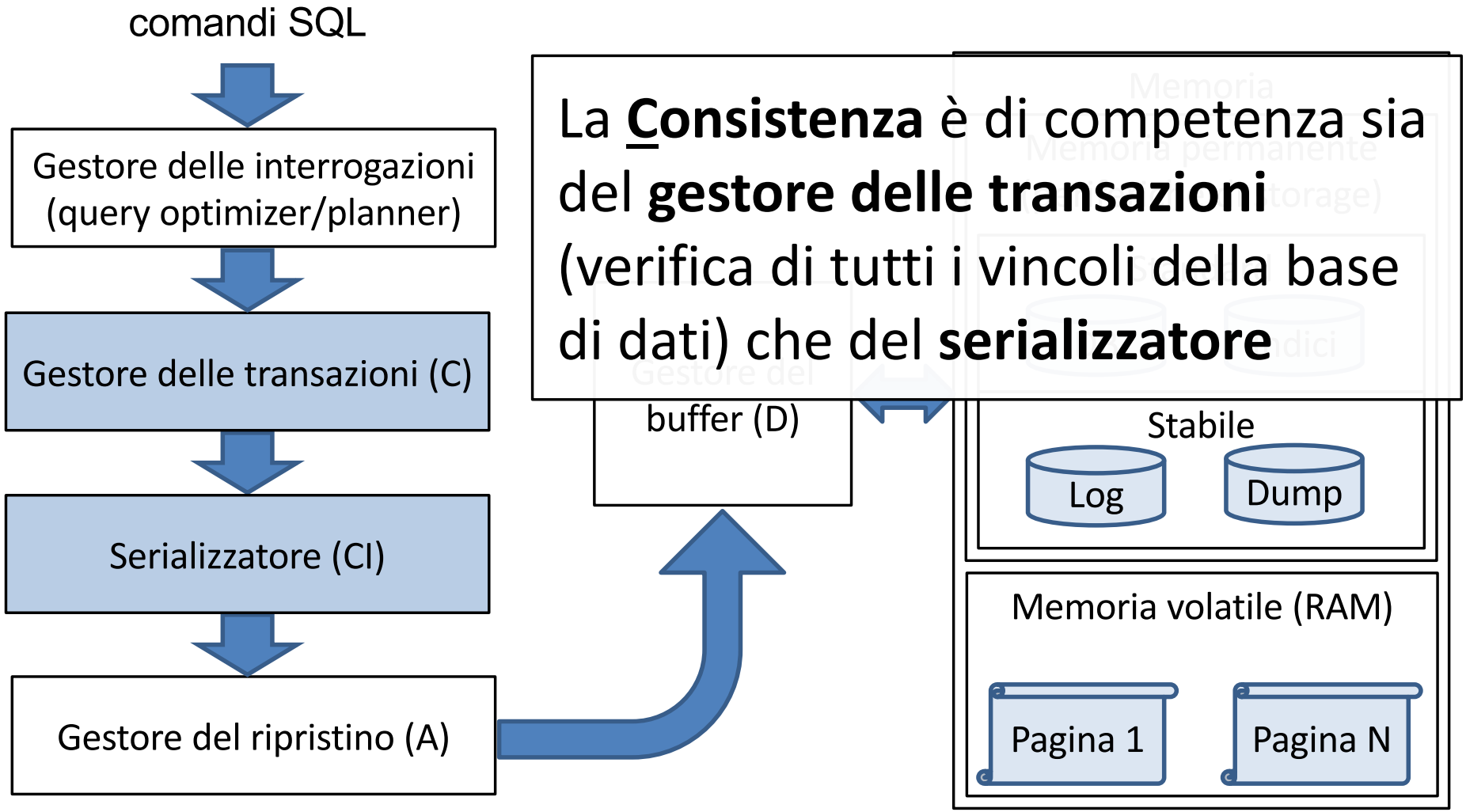


Il **gestore del ripristino** garantisce anche **l'integrità** dei dati (se i dati vengono corrotti, cerca di recuperare il dato corretto oppure forza un abort della transazione)

Architettura dei DBMS



Architettura dei DBMS



Architettura dei DBMS: argomenti

- Gestore del buffer (blocco slide I)
- Strutture per l'organizzazione dei file (blocco slide I)
- Ottimizzatore fisico (non ne parliamo)
- Gestore della concorrenza (serializzatore) (blocco di slide II)
- Gestore del ripristino (blocco di slide III)

Basi di Dati
Architettura dei DBMS:
Gestore del buffer

Outline

- **Gestore del buffer**
- Strutture primarie per l'organizzazione dei file
  - Heap
  - Struttura ordinata
- Strutture secondarie (indici)
  - B-tree e B+-tree
  - Caratterizzazione degli indici
  - Euristiche/consigli

Memoria principale e secondaria

- I programmi possono fare riferimento solo a dati in memoria principale

- Quindi i dati in memoria secondaria possono essere utilizzati solo se prima trasferiti in memoria principale (questo spiega i termini "principale" e "secondaria")

Le basi di dati sono grandi e persistenti

- La *persistenza* richiede di memorizzare i dati in memoria secondaria
- La *grandezza* richiede che tale gestione sia sofisticata (non possiamo caricare tutto in memoria principale e poi riscaricare)

Le pagine

- Tutti i dati del database (tabelle, indici, log, dump) sono **organizzati in pagine**.
- Le pagine hanno dimensioni che dipendono dal sistema (anche variabili).
- **I record sono contenuti nelle pagine**.
- Se la transazione ha bisogno di lavorare su un determinato record (tupla), il gestore del buffer cerca una pagina *pid* che contiene il record desiderato.
- Il gestore del buffer, cioè, riceve una richiesta chiamata **fix pid**, che significa: metti a disposizione della transazione la pagina *pid* (pid = page identification).

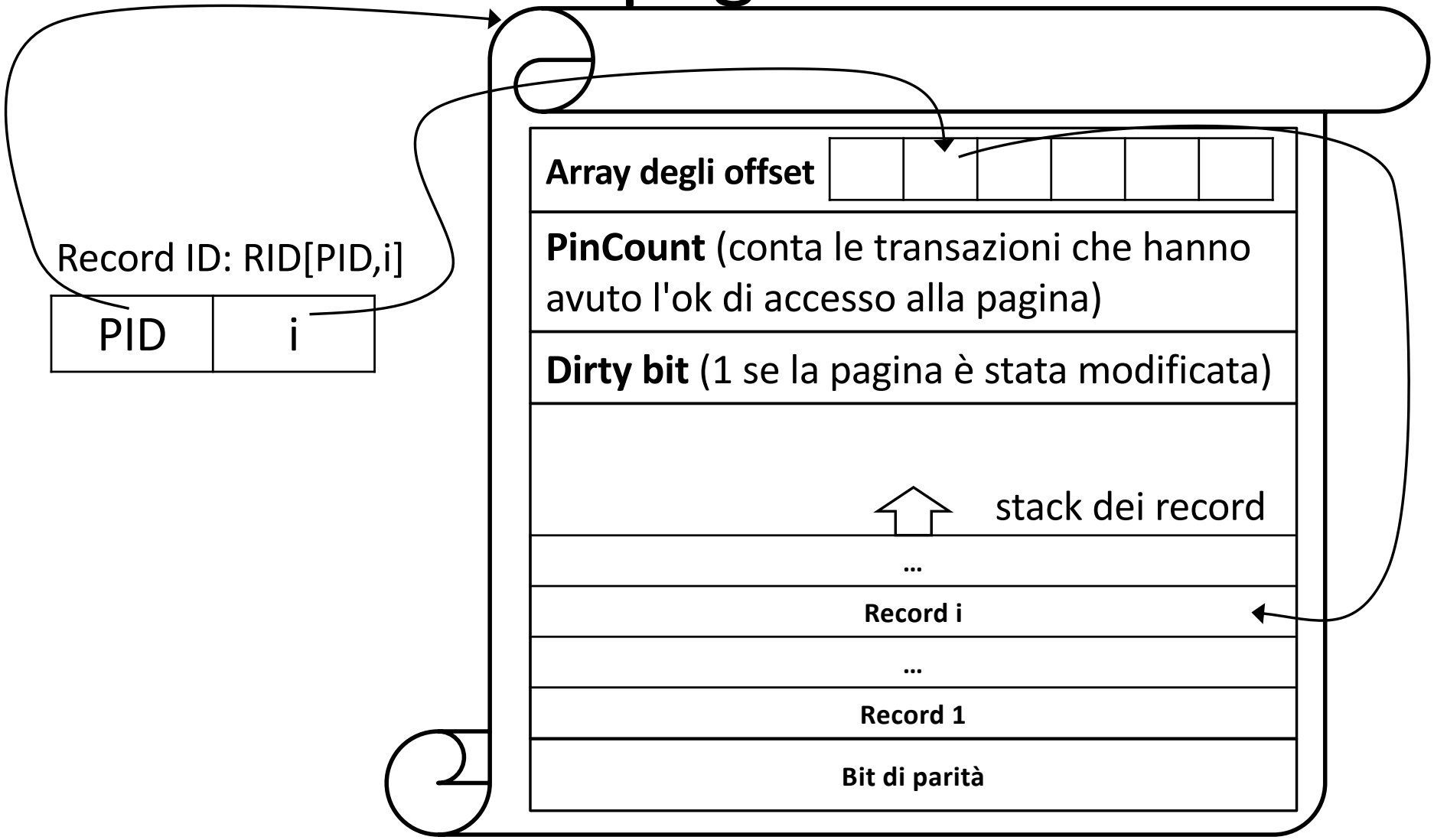
Caricamento delle pagine

- Quando riceve la richiesta di una pagina, il **gestore del buffer** caricherà la pagina nel buffer stesso prendendola dalla periferica di storage se non è già presente nel buffer:
  - Il gestore del buffer legge i dati memorizzati in memoria secondaria e li trasferisce in cache,
  - Dalla cache vengono poste nel buffer della memoria centrale,
  - Il gestore del buffer risponde alla transazione inviandole l'indirizzo della memoria centrale in cui trovare la pagina.

Tempi di trasferimento

- Ricordiamo che i tempi di trasferimento di una pagina tra periferiche di **storage** e RAM (e viceversa) sono dell'ordine dei 5-20 millisecondi **(10^{-3} secondi)**
- I tempi di trasferimento di pagine in **RAM** sono nell'ordine dei 50-70 nanosecondi (**10^{-9} secondi**)
- Abbiamo un fattore di **10^{6} di differenza**
- Quindi il costo temporale grava soprattutto sulla movimentazione delle pagine da periferica a memoria centrale (e viceversa)
- Nonostante i progressi tecnologici, il divario ha ancora lo stesso ordine di grandezza

Organizzazione delle tuple nelle pagine



DBMS e file system

- L'interazione tra il DBMS (**buffer manager**) e il sistema operativo (**file system**) non è banale.
- Il DBMS usa alcune (poche) funzionalità del file system creando, però, una propria astrazione dei file che consente di garantire **efficienza** (tramite l'uso del buffer e una gestione di basso livello delle strutture fisiche) e **transazionalità** (attraverso il gestore dell'affidabilità, che assicura le proprietà di atomicità e persistenza).

DBMS e file system

- Il DBMS crea file di grandi dimensioni che utilizza per memorizzare diverse relazioni (al limite, l'intero database).
- La struttura dei file, sia all'interno dei singoli blocchi sia nell'organizzazione dei dati a partire dai blocchi, è gestita direttamente dal DBMS.
- Quindi i DBMS utilizzano le funzionalità del file system per creare e eliminare file e per leggere e scrivere singoli blocchi o sequenze di blocchi contigui.
- Talvolta, vengono creati file in tempi successivi:
  - è possibile che un file contenga i dati di più relazioni e che i vari record di una tabella siano in file diversi.

Blocchi e record

- 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.
- Per comodità consideriamo **blocco** e **pagina** sinonimi.

Fattore di blocco (blocking factor)

- numero di record in un blocco $\lfloor L_B / L_R \rfloor$
  - L_B: dimensione di un blocco
  - L_R: dimensione media di un record
  - se $L_B > L_R$, possiamo avere più record in un blocco
  - Es. L_B = 4096 B, L_R = 100 B, fattore di blocco = $\lfloor 4096 / 100 \rfloor$ = 40 record per blocco
- lo spazio residuo (nell'es. 96 B) può essere
  - utilizzato per altre tabelle (record "spanned" o impaccati)
  - non utilizzato ("unspanned")

Ottimizzazione fisica

- Ricordiamo che l'ottimizzatore fisico prende in input un albero di parsificazione modificato dall'ottimizzatore logico e corrispondente a un'interrogazione DML
- L'ottimizzatore fisico si basa innanzitutto sui **metodi di accesso**, cioè deve scegliere come accedere ai dati richiesti
- I DBMS possono presentare più modalità di accesso ai dati che dipendono dai **metodi usati per organizzare i record su storage**

Outline

- Gestore del buffer
- **Strutture primarie per l'organizzazione dei file**
  - **Heap**
  - Struttura ordinata
- Strutture secondarie (indici)
  - B-tree e B+-tree
  - Caratterizzazione degli indici
  - Euristiche/consigli

Strutture primarie per l'organizzazione dei file

Specificano come i record e le pagine sono fisicamente organizzati su file.

- File di record **non ordinati o a heap** (struttura **seriale**)
- File di record **ordinati** (struttura **sequenziale**)
- Strutture ad accesso **calcolato o a hash** (che non vediamo)

Record a heap (struttura seriale)

- Chiamata anche:
  - "Entry sequenced", file heap, file non ordinato
- I record vengono inseriti nei blocchi nell'ordine con cui si presentano al sistema
- È **molto diffusa** nelle basi di dati relazionali, spesso associata a strutture **secondarie** di accesso come indici
- Gli **inserimenti** vengono effettuati in modo molto efficiente:
  - in coda oppure al posto di record cancellati
- La **cancellazione** lascia spazio inutilizzato nei blocchi e richiede un compattamento periodico

Metodo di accesso seriale

Per quanto riguarda la **ricerca**, l'ottimizzatore fisico deve valutare ***a priori*** il costo dell'uso di questo metodo per confrontarlo con altri.

Immaginiamo una selezione sulla chiave primaria:

$$\sigma_{MATR=333}(studenti)$$

Consideriamo un'organizzazione dei dati secondo un accesso seriale con **N pagine**, da 1 a N.

Senza strutture secondarie, il DBMS non può fare altro che leggere le pagine una dopo l'altra.

Costo dell'accesso seriale

Una ricerca con insuccesso richiede di leggere tutte le pagine (devo scorrere tutti i record per verificare che un valore non sia presente).

Una ricerca con successo, invece, in media richiede di leggere metà delle pagine.

Più formalmente consideriamo una **distribuzione di probabilità uniforme** in cui il record ha la **stessa probabilità** $\frac{1}{N}$ di trovarsi in una qualsiasi pagina.

Costo dell'accesso seriale

Se il record è nella prima pagina (probabilità $\frac{1}{N}$), leggerò in tutto 1 pagina.

Se il record invece è nella seconda pagina (probabilità $\frac{1}{N}$), leggerò in tutto 2 pagine…

Il costo medio è

$$1 \cdot \frac{1}{N} + 2 \cdot \frac{1}{N} + \cdots + N \cdot \frac{1}{N} =$$

$$\sum_{i=1}^{N} i \frac{1}{N} = \frac{1}{N} \cdot \frac{N(N+1)}{2} = \frac{N+1}{2}$$

Outline

- Gestore del buffer
- **Strutture primarie per l'organizzazione dei file**
  - Heap
  - **Struttura ordinata**
- Strutture secondarie (indici)
  - B-tree e B+-tree
  - Caratterizzazione degli indici
  - Euristiche/consigli

Struttura ordinata (o sequenziale)

Se manteniamo, invece di un heap, un'**organizzazione ordinata** in cui i record nei file sono ordinati secondo un attributo (di solito chiamato «*chiave*»(*)), abbiamo vantaggi?

L'**inserimento** è costoso perché richiede di spostare pagine.

La **cancellazione** si può attuare contrassegnando un record come cancellato e attuando una riorganizzazione periodica.

La **ricerca con insuccesso** sull'attributo «chiave» ha lo stesso costo della ricerca con successo perché non ho bisogno di scorrere tutte le pagine (mi posso **arrestare** quando supero il valore richiesto): $\frac{N+1}{2}$

- Le ricerche dicotomiche spesso non sono possibili perché i record non sono memorizzati in blocchi consecutivi.

(*) Tra virgolette per non confonderlo con le chiavi propriamente dette.

Costo dell'accesso sequenziale

Cosa succede se effettuo una selezione su un attributo non «chiave»?

$$\sigma_{INDIRIZZO='TO'}(studenti)$$

- In caso di insuccesso, devo scorrere tutte le pagine: costo *N*

Stesso discorso per le ricerche di range (<, >, between)

Questi aspetti non verranno trattati

Osservazione

Gli accessi a heap e a struttura ordinata hanno costi che sono dell'ordine del numero delle pagine (nel miglior caso: metà delle pagine).

Questa organizzazione è troppo costosa se pensiamo a sistemi informativi con migliaia di pagine per le tabelle.

Si possono usare organizzazioni più complesse e soprattutto si associano alle strutture primarie a heap e ordinate strutture secondarie.

Outline

- Gestore del buffer
- Strutture primarie per l'organizzazione dei file
  - Heap
  - Struttura ordinata
- **Strutture secondarie (indici)**
  - **B-tree e B+-tree**
  - Caratterizzazione degli indici
  - Euristiche/consigli

Gli indici

- L'indice è una **struttura secondaria**, cioè è separata dall'area che contiene le pagine dati
- In altre parole, i record stessi non sono contenuti negli indici; gli indici contengono «puntatori» (i **RID**) ai record che sono memorizzati nelle strutture primarie
- Prima di introdurre gli indici, studieremo brevemente la struttura dati **B-tree** (o B-Albero)
- È la struttura utilizzata nella maggior parte dei sistemi che **implementano indici**

B-tree

Un **B-tree** è una generalizzazione degli alberi binari di ricerca in cui ogni nodo può avere *m* figli (*branching factor*):

- ogni sottoalbero di *sinistra* di una chiave *k* ha chiavi di ricerca strettamente *inferiori* alla chiave *k*,
- ogni sottoalbero di *destra* ha chiavi *strettamente* superiori a *k.*

I B-tree sono bilanciati.

N.B. Qui per "chiave" intendiamo non chiave relazionale, ma "chiave di ricerca", cioè un attributo qualsiasi su cui si effettua una ricerca

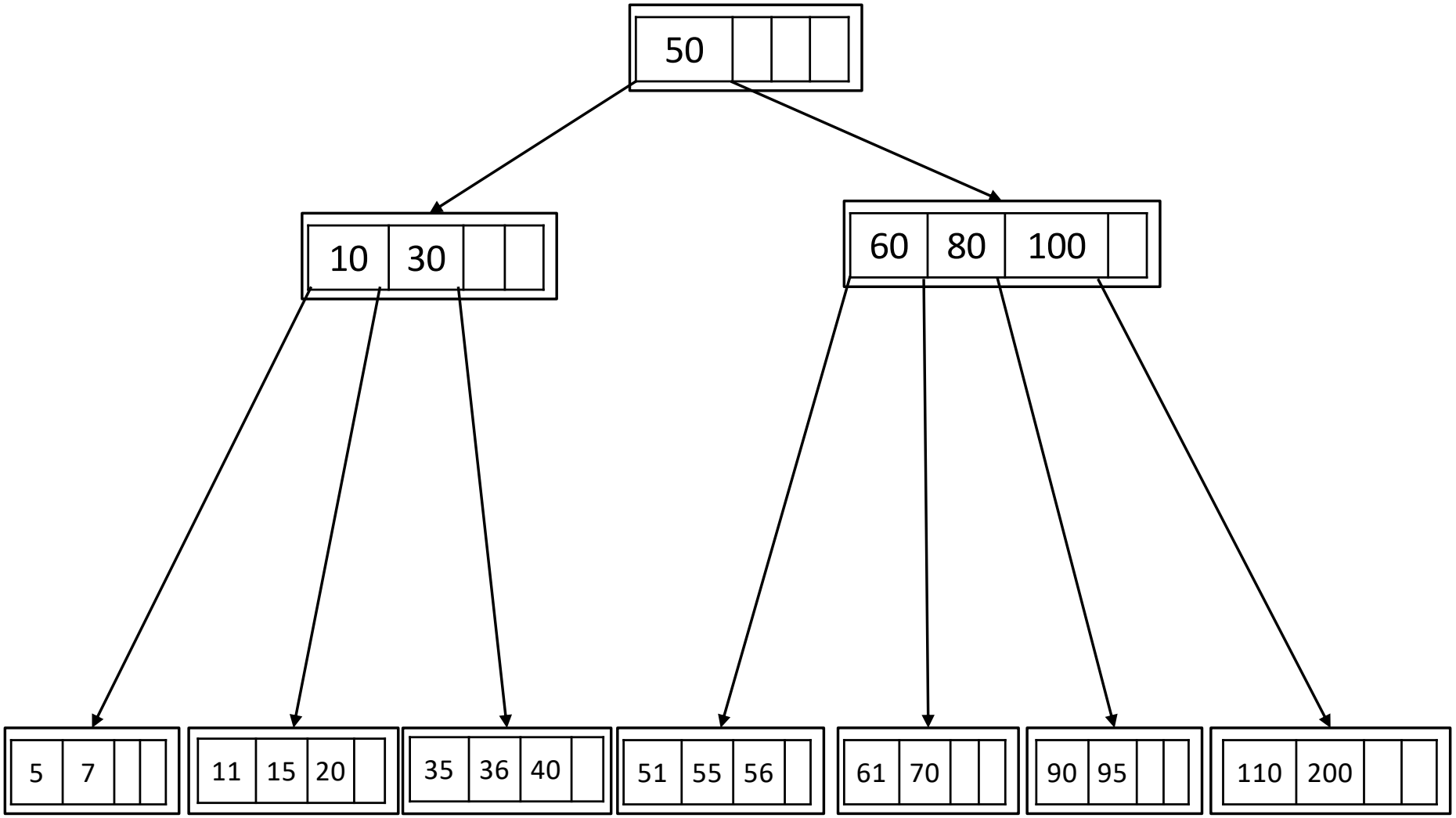
B-tree

Un B-tree con branching factor *m* deve rispettare le seguenti proprietà (ricordiamo che un nodo interno è un nodo non-foglia):

1. **Ogni nodo** ha ***al massimo*** $\boldsymbol{m-1}$ **chiavi**.
2. **Ogni nodo** a eccezione della radice ha ***almeno*** $\boldsymbol{\lceil m/2 \rceil - 1}$ **chiavi**.
3. Se il B-tree non è vuoto, la radice ha almeno una chiave.
4. Tutte le foglie sono allo stesso livello.
5. Un nodo interno che ha *k* chiavi ha *k+1* figli.

- Ad es. in un B-tree non vuoto con m=5:
  - Ogni nodo ha da 2 a 4 chiavi, tranne la radice che ha da 1 a 4 chiavi
- Ad es. in un B-tree non vuoto con m=100:
  - Ogni nodo ha da 49 a 99 chiavi, tranne la radice che ha da 1 a 99 chiavi

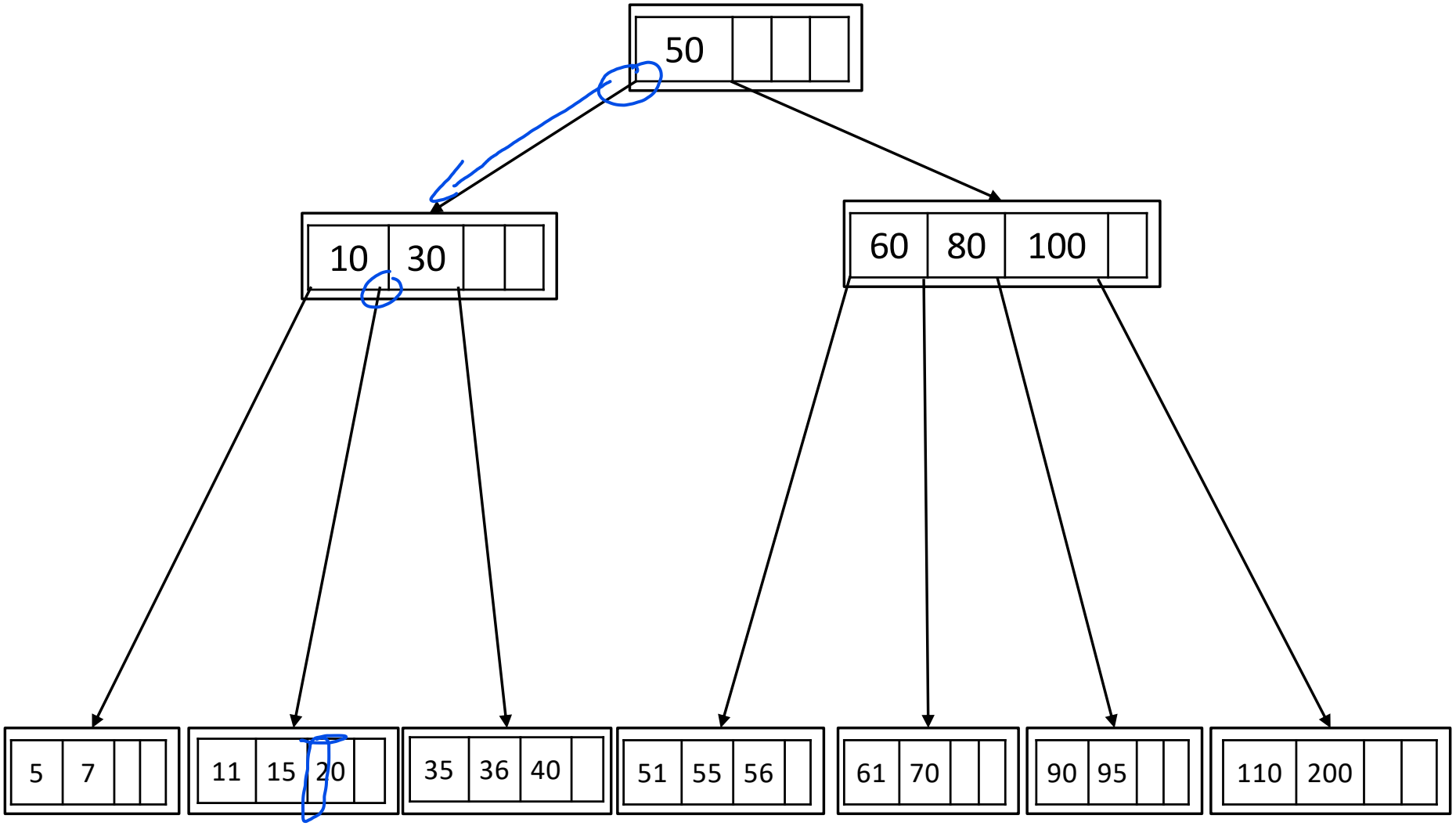
Esempio di B-tree (m=5)



Rispetta tutte le proprietà: ogni nodo ha al massimo 4 chiavi; ogni nodo eccetto la radice ha almeno 2 chiavi; la radice ha almeno 1 chiave; tutte le foglie sono allo stesso livello; ogni nodo non foglia se ha k chiavi ha k+1 figli.

Ricerca in un B-tree

Ricerca del valore 17:



Ricerca in un B-tree

- Cerco il valore 17 all'interno dell'albero.
- Inizio dalla radice.
- Confronto la chiave di ricerca con le chiavi della radice (17 < 50) e scendo nel sottoalbero di sinistra.
- Confronto la chiave di ricerca con le chiavi del nodo e, dato che 10 < 17 < 30, entro nel secondo sottoalbero, che è una foglia.
- Dato che 17 non è contenuto nella foglia, il valore 17 non è contenuto nel B-tree.

Analisi quantitativa

Il dato quantitativo importante per i DBMS è il **numero di livelli** (per es. la radice, un livello intermedio e le foglie: 3 livelli)

Si può dimostrare che un B-tree con *N* chiavi totali e branching factor *m*

ha un numero di livelli *L* tale che

$$\log_m(N+1) \leq L \leq \log_{\lceil \frac{m}{2} \rceil}\left(\frac{N+1}{2}\right) + 1$$

SKIP

Stima del numero *minimo* di livelli

Ogni nodo contiene al massimo m-1 chiavi.

Quindi un B-tree avrà al massimo:

- un nodo radice al livello 1 con m-1 chiavi e m figli,
- m nodi al livello 2 ognuno con m-1 chiavi e m figli (in tutto m(m-1) chiavi),
- m^2 nodi al livello 3 ognuno con m-1 chiavi e m figli (in tutto m^2(m-1) chiavi) e così via fino a…
- m^{L-1} nodi al livello L ognuno con m-1 chiavi (in tutto m^{L-1}(m-1) chiavi)

Il numero totale N chiavi sarà al massimo
$m - 1 + m(m-1) + m^2(m-1) + \cdots + m^{L-1}(m-1)$,
cioè $N \leq (m-1)\sum_{i=1}^{L} m^{i-1}$.

Stima del numero *minimo* di livelli

Il numero totale N chiavi sarà al massimo
$m - 1 + m(m-1) + m^2(m-1) + \cdots + m^{L-1}(m-1)$,
cioè $N \leq (m-1)\sum_{i=1}^{L} m^{i-1}$.

Questa è una serie geometrica per cui si ha $\sum_{i=1}^{L} m^{i-1} = \frac{1-m^L}{1-m}$.

Quindi $N \leq -(1-m)\frac{1-m^L}{1-m}$, cioè $N + 1 \leq m^L$.

Applicando il logaritmo a entrambi i membri si ottiene
$\log_m(N+1) \leq \log_m m^L$, cioè $\log_m(N+1) \leq L$.

Stima del numero *massimo* di livelli

La radice di un B-tree contiene almeno 1 chiave. Tutti gli altri nodi contengono almeno $\lceil m/2-1 \rceil$ chiavi.

Un B-tree con L livelli contiene:

- 1 nodo radice a livello 1 con 1 chiave,
- almeno 2 nodi a livello 2 con $\lceil m/2-1 \rceil$ chiavi,
- almeno $2\lceil m/2 \rceil$ nodi a livello 3 con $\lceil m/2-1 \rceil$ chiavi,
- almeno $2\lceil m/2-1 \rceil^2$ a livello 4 con $\lceil m/2-1 \rceil$ chiavi e così via fino a…
- almeno $2\lceil m/2-1 \rceil^{L-2}$ a livello L con $\lceil m/2-1 \rceil$ chiavi.

Quindi il numero totale di chiavi è almeno

$$1 + \left\lceil \frac{m}{2} - 1 \right\rceil \cdot \left(2 + 2\left\lceil \frac{m}{2} \right\rceil + 2\left\lceil \frac{m}{2} \right\rceil^2 + \cdots + 2\left\lceil \frac{m}{2} \right\rceil^{L-2}\right),$$

cioè $N \geq 1 + \left\lceil \frac{m}{2} - 1 \right\rceil \cdot \sum_{i=1}^{L-1} 2\left\lceil \frac{m}{2} \right\rceil^{i-1}$

Stima del numero *massimo* di livelli

È una serie geometrica per cui si ha $\sum_{i=1}^{L-1} \left\lceil \frac{m}{2} \right\rceil^{i-1} = \frac{1-\left\lceil \frac{m}{2} \right\rceil^{L-1}}{1-\left\lceil \frac{m}{2} \right\rceil}$.

Quindi da $N \geq 1 + \left\lceil \frac{m}{2} - 1 \right\rceil \cdot \sum_{i=1}^{L-1} 2 \left\lceil \frac{m}{2} \right\rceil^{i-1}$ si ha

$$N \geq 1 + 2\left\lceil \frac{m}{2} - 1 \right\rceil \cdot \frac{1-\left\lceil \frac{m}{2} \right\rceil^{L-1}}{1-\left\lceil \frac{m}{2} \right\rceil} = 1 + 2\left\lceil \frac{m}{2} - 1 \right\rceil \cdot \frac{\left\lceil \frac{m}{2} \right\rceil^{L-1} - 1}{\left\lceil \frac{m}{2} - 1 \right\rceil} =$$

$$= 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{L-1} - 1.$$

Perciò $\left\lceil \frac{m}{2} \right\rceil^{L-1} \leq \frac{N+1}{2}$.

Applicando il logaritmo a entrambi i membri abbiamo

$$L \leq \log_{\left\lceil \frac{m}{2} \right\rceil} \frac{N+1}{2} + 1$$

[CVD]

Approssimazione della formula

Dal momento che abbiamo valori alti del numero di chiavi *N* e piuttosto alti del branching factor *m*, la formula che stima il numero di livelli *L*

$$\log_m(N+1) \leq L \leq \log_{\lceil \frac{m}{2} \rceil}\left(\frac{N+1}{2}\right) + 1$$

si può approssimare in questo modo:

$$\log_m(N) \leq L \leq \log_m(N) + 1$$

Dato che $L \simeq \log_m N$,

$$N \simeq m^L$$

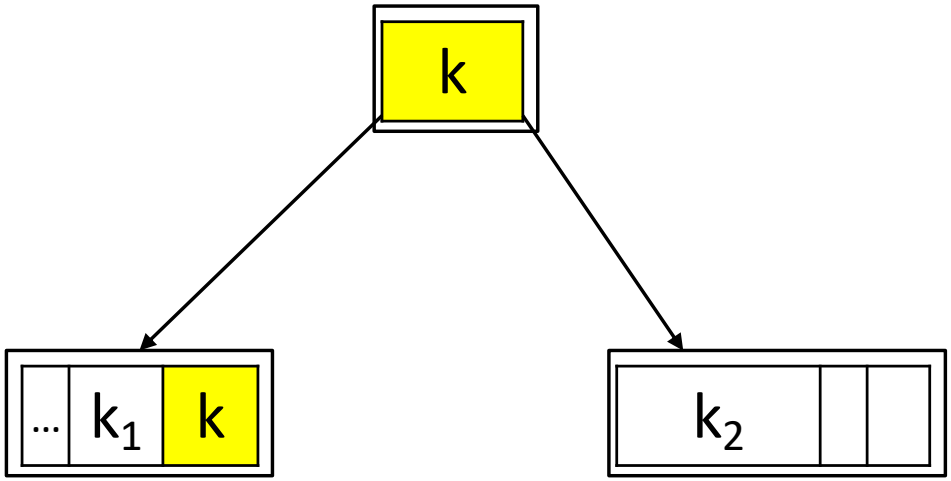
Esempio

- In ogni nodo memorizziamo, oltre alle chiavi di ricerca, anche i RID dei relativi record.
- Assumiamo che la coppia <chiave di ricerca,RID> occupi 40 byte.
- Se una pagina ha una dimensione di 4 KB, ogni pagina può contenere 4 KB / 40 B, cioè circa 99 chiavi di ricerca.
- Considerando che un nodo occupa una pagina, abbiamo che m=100.
- Con soli tre livelli possiamo memorizzare $m^L=100^3$, cioè un milione di chiavi.
- Per accedere a una chiave, se ogni nodo è contenuto in una pagina diversa, «muovo» al più **3 pagine**!
- Nei DBMS, gli indici reggono bene un carico del 70% della capacità massima (cioè con 3 livelli si reggono bene 700 mila chiavi).

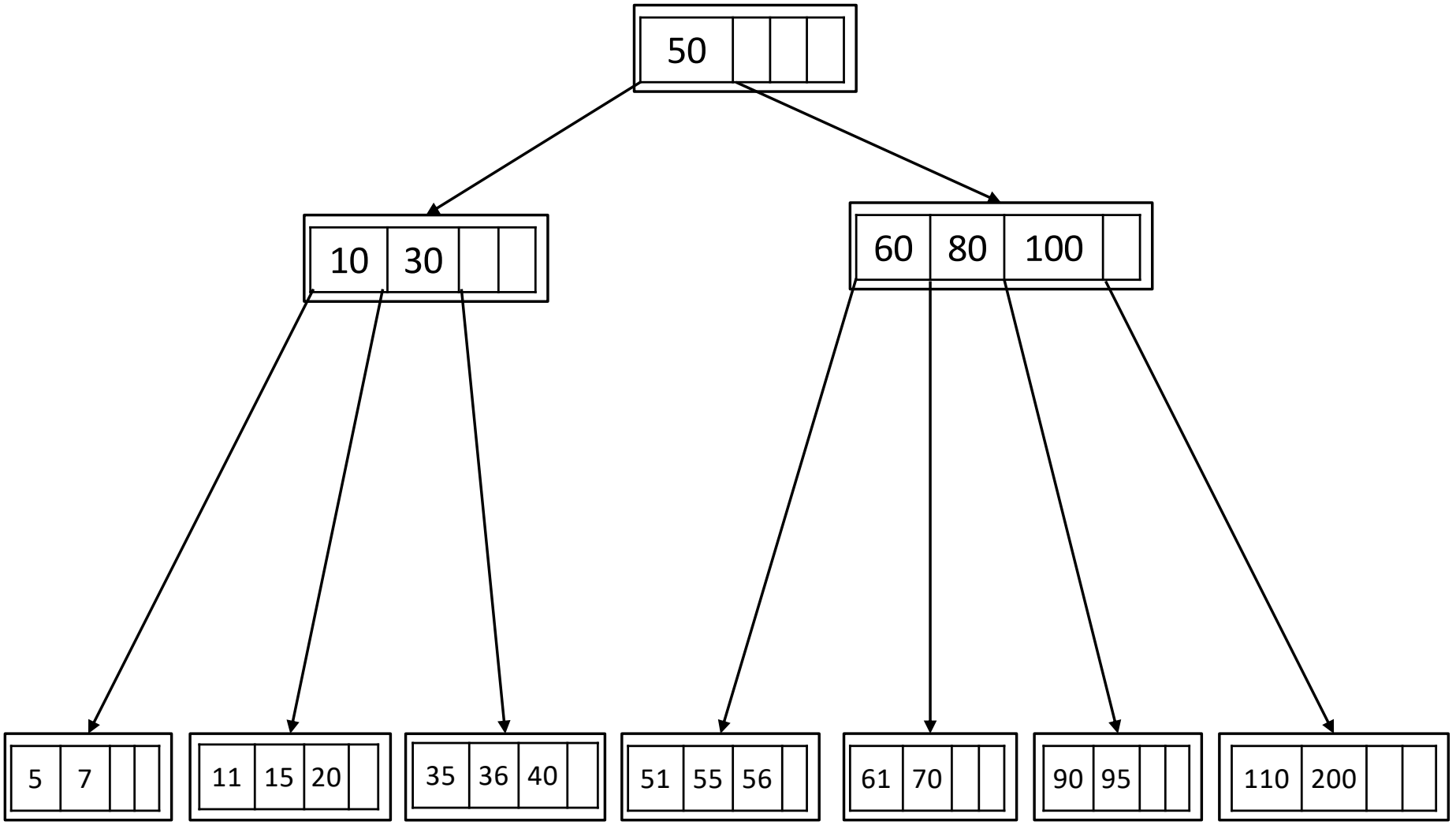
B+-tree

- Viene usata spesso una variazione dei B-tree chiamata **B+-tree**.
- **Idea**: per cercare di tenere in memoria principale i nodi usati più spesso, cioè i **nodi interni**, li rendiamo **più leggeri**.
- Le coppie **<chiavi di ricerca, RID> vengono memorizzate solo nelle foglie**. Nei **nodi interni memorizziamo solo le chiavi di ricerca**.
- Quindi le **chiavi di ricerca vengono duplicate nelle foglie**.
- Questo nuovo modo di organizzare i B-tree non cambia le proprietà dei B+-tree rispetto a quelle dei B-tree.

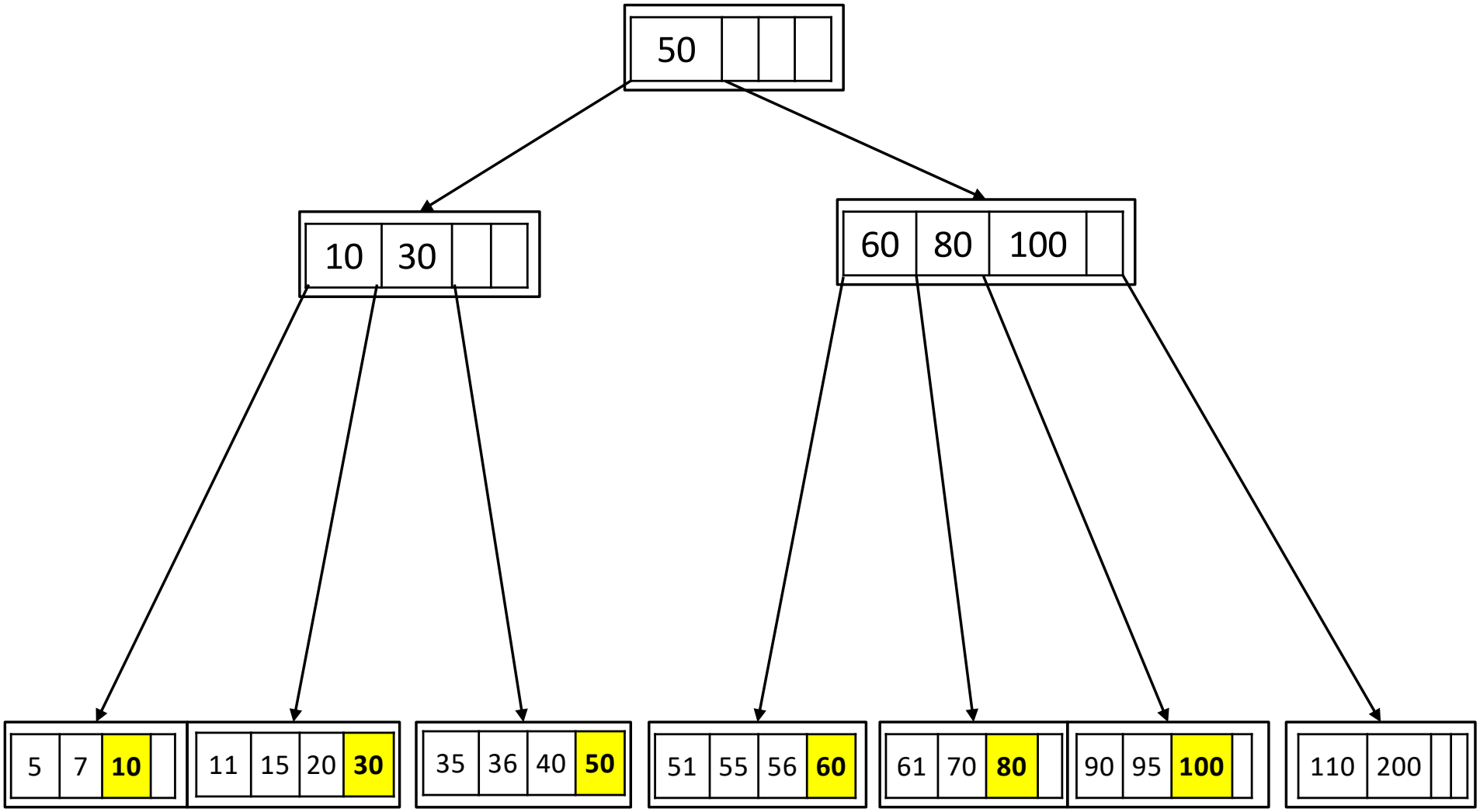
B+-tree



B-tree

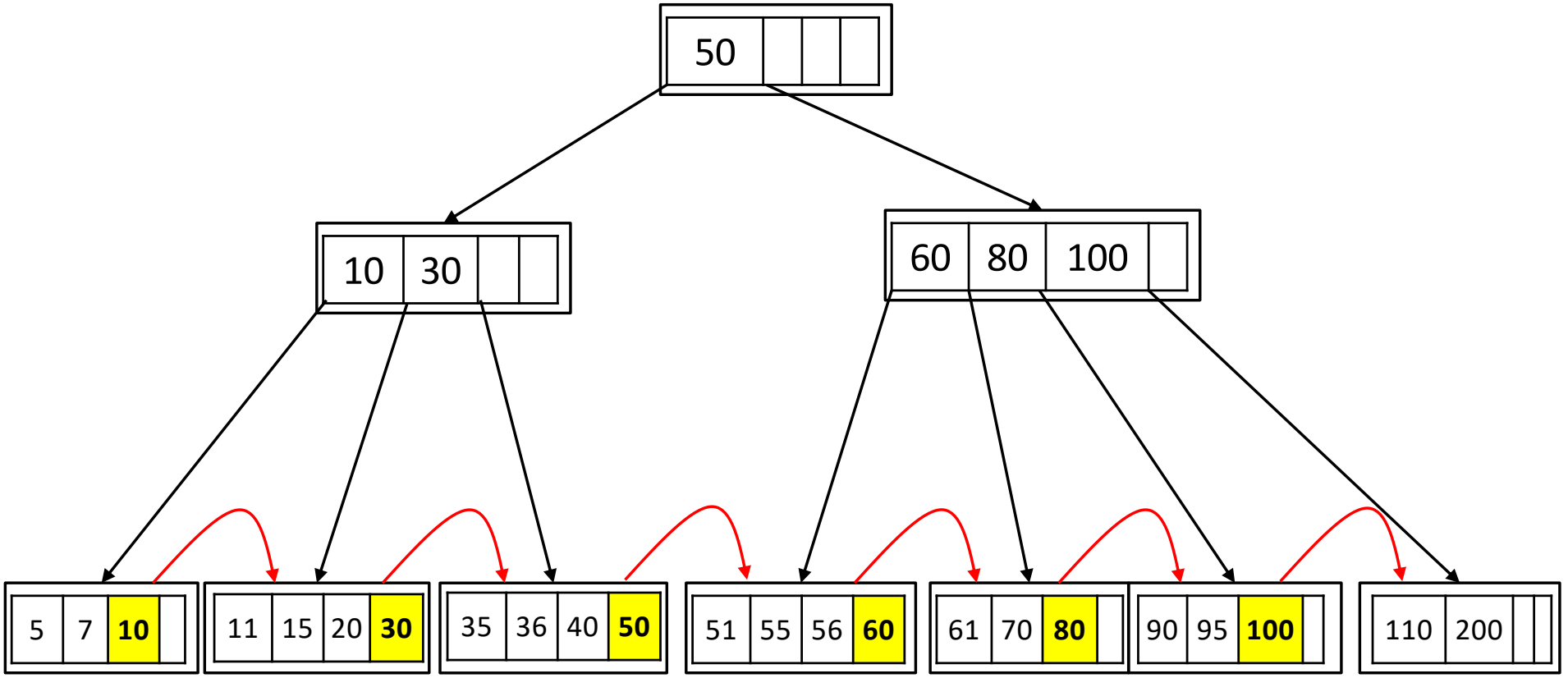


B+-tree


50
10
30
60
80
100
5
7
10
11
15
20
30
35
36
40
50
51
55
56
60
61
70
80
90
95
100
110
200

Collegamenti tra foglie nei B+-tree

Inoltre le foglie sono tutte linkate tra di loro in modo da potere accedere in modo efficiente a sequenze di chiavi



Collegamento con l'area primaria

- Immaginiamo di usare un B+-tree per indicizzare le matricole degli studenti in un tabella studenti (che può essere memorizzata ad esempio come heap).
- I nodi interni hanno le matricole/chiavi di ricerca *k*.
- Le foglie hanno tutte le matricole/chiavi di ricerca abbinate ai rispettivi RID: *k*= <k,RID>*.
- Per la chiave di ricerca 35, per esempio, avrò il RID che punta al record corrispondente allo studente con matricola 35.

Tipi diversi di data entry K*

Le foglie del B+-tree possono contenere come data entry k*:

- **<k,tupla>**: il data entry è il record stesso (cioè in questo caso il B+-tree è usato non come indice ma come struttura di memorizzazione primaria).
- **<k,RID>**: il data entry è puntatore al record nell'area primaria.
- **<k,lista_di_RID>**: il data entry è una lista di puntatori a record che hanno tutti la stessa chiave k.

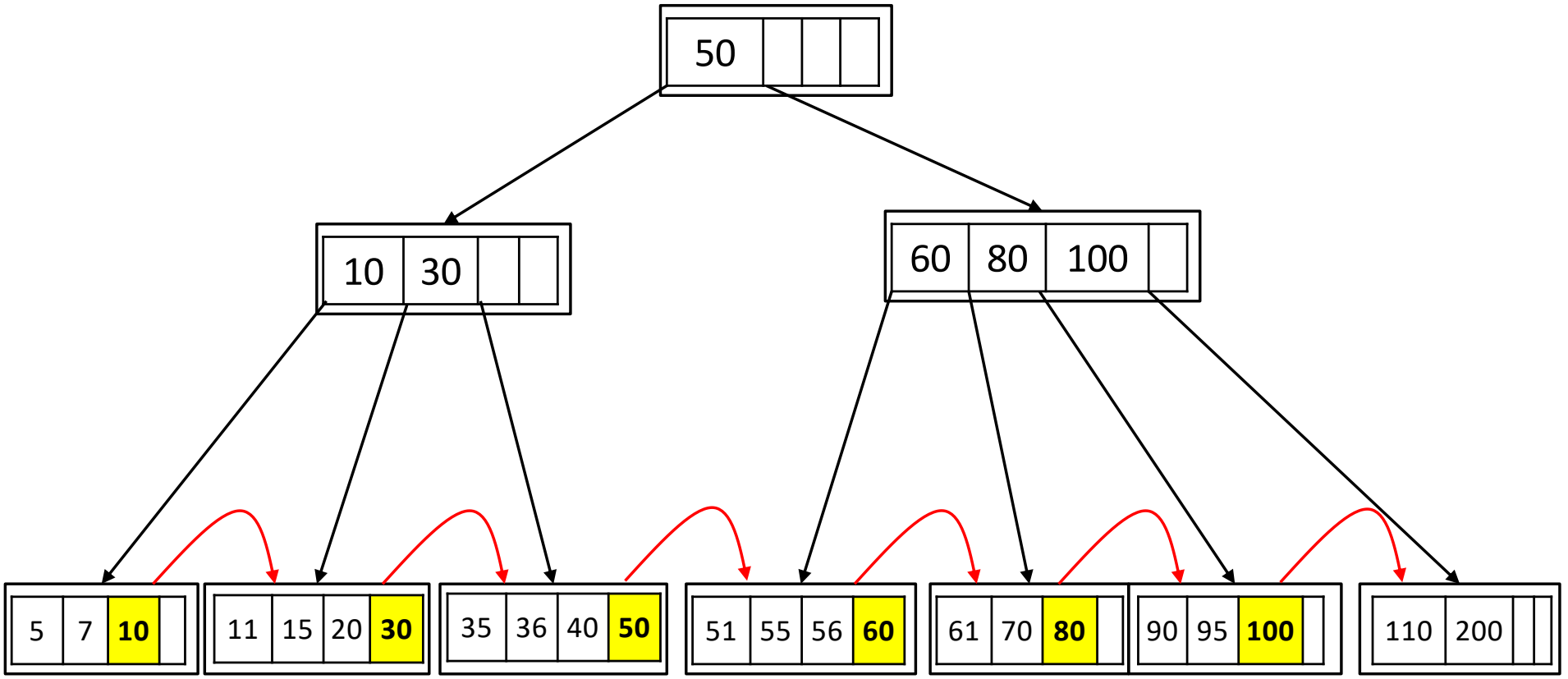
Variante <*k,lista_di_RID*>

Serve quando un progettista crea un indice su un attributo che non è chiave relazionale (cioè non è univoco).

Ad esempio, se nel B+-tree precedente le chiavi di ricerca corrispondono all'attributo "crediti superati", ogni valore può fare riferimento a più record di studenti (cioè tutti gli studenti che hanno superato quel numero di crediti).

Ricerca puntuale nell'indice

$\sigma_{MATR=56}(studenti)$



Ricerca puntuale nell'indice

Immaginiamo questa selezione

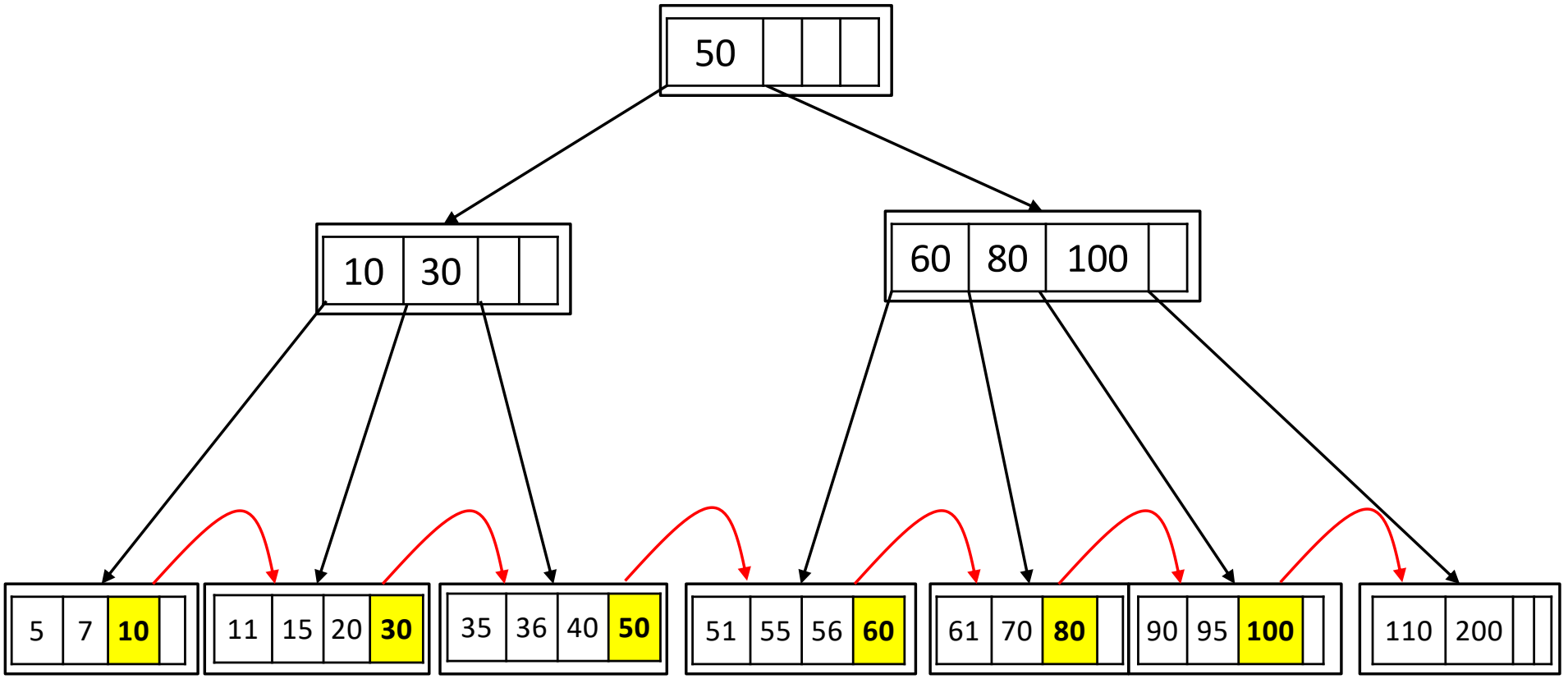
$$\sigma_{MATR=56}(studenti)$$

- Il DBMS sa che esiste l'indice sul campo MATR e usa il corrispettivo B+-tree: prende il valore 56 e lo confronta con la radice; dato che 56 > 50 scende nel sottoalbero di destra e successivamente, dato che 56 < 60, nel sottoalbero di sinistra.
- Trova il valore nella foglia e utilizza il RID per accedere al record.
- In tutto sono stati necessari *L + 1* accessi.

Nei sistemi informativi reali, con centinaia di migliaia di chiavi di ricerca, in genere gli indici hanno 2/3 livelli.

Ricerca di range di valori

$\sigma_{53 \leq MATR \leq 61}(studenti)$



Ricerca di range di valori

Immaginiamo questa selezione

$$\sigma_{53 \leq MATR \leq 61}(studenti)$$

La strategia è la seguente:

1. Si parte dal valore più basso: la ricerca di 53 viene effettuata come nel caso di ricerca puntuale.
2. Non trovo 53 ma trovo 55 che è il primo valore utile maggiore di 53.
3. Ora basta scandire sequenzialmente le foglie sino a che non si raggiunge la foglia che può contenere il valore più alto, 61, sfruttando i link alle foglie contigue.

Outline

- Gestore del buffer
- Strutture primarie per l'organizzazione dei file
  - Heap
  - Struttura ordinata
- **Strutture secondarie (indici)**
  - B-tree e B+-tree
  - **Caratterizzazione degli indici**
  - Euristiche/consigli

Perché gli indici?

- L'indice è una struttura separata rispetto alle strutture di memorizzazione primaria

- Posso costruire diversi indici sulla stessa struttura di memorizzazione primaria:
  - Indice su matricola
  - Indice su città
  - Indici su più attributi (ad esempio coppie)

- Un DB administrator crea un indice quando il carico di lavoro in termini di interrogazioni è tale per cui il costo di accesso all'indice riduce notevolmente i tempi di risposta rispetto alla scansione sequenziale

Indici su tutto?

- È vero che l'indice rende molto più efficienti le interrogazioni, ma appesantisce le modifiche:
  - Gli inserimenti e le cancellazioni modificano gli indici
  - Le modifiche possono modificare gli indici
- Può non essere conveniente, ad esempio, definire indici su attributi che vengono cambiati molto spesso dagli update

Caratterizzazione degli indici

- Indice **primario**
  - indice definito sullo stesso ***attributo chiave relazionale*** su cui sono ***ordinati fisicamente*** i record.
- Indice **clusterizzato**
  - indice su un ***attributo non chiave relazionale*** (quindi sono possibili valori ripetuti) su cui i record sono ***ordinati fisicamente***.
- Indici **secondari**
  - indici su un ***attributo qualunque*** su cui i record ***non sono ordinati fisicamente***.
- Una tabella ha al più un indice primario o clusterizzato (ma non entrambi) e zero o più indici secondari.

Outline

- Introduzione all'architettura dei DBMS
- Gestore del buffer
- Strutture primarie per l'organizzazione dei file
  - Heap
  - Struttura ordinata
- **Strutture secondarie (indici)**
  - B-tree e B+-tree
  - Caratterizzazione degli indici
  - **Euristiche/consigli**

Quando definire gli indici?

Consideriamo un esempio

STUDENTI(MATR,Nome,DataNascita,Genere,Indirizzo)

ESAMI(MATR,Corso,Voto,DataEsame)

Ricordiamo che, se è vero che un indice favorisce le interrogazioni, ha comunque un costo di mantenimento per gli inserimenti, cancellazioni e modifiche.

Esistono delle euristiche generali per capire se conviene o meno definire un indice.

Euristiche/consigli

1. Evitare gli indici su tabelle di poche pagine
2. Evitare indici su attributi volatili (ad esempio, Saldo del conto corrente)
3. Evitare indici su chiavi poco selettive
4. Evitare indici su chiavi con valori sbilanciati
5. Limitare il numero di indici
6. Definire indici su chiavi relazionali ed esterne
7. Gli indici velocizzano le scansioni ordinate
8. Conoscere a fondo il DBMS

1. Evitare indici su tabelle piccole

Una tabella relazionale di poche pagine (~10 pagine) non ha esigenza di indici, perché probabilmente potrà essere caricata interamente nei buffer.

2. Evitare indici su attributi volatili

Un attributo volatile come il saldo del conto corrente viene modificato più spesso di quanto venga letto, quindi l'overhead del mantenimento dell'indice non giustifica il guadagno di prestazioni in lettura.

3. Selettività di un indice

La selettività f_s di un indice su un attributo A in una tabella T è data dal rapporto tra il numero di pagine restituite dalla ricerca di un valore di A e il numero di pagine in T.

Empiricamente si suggerisce di definire un indice su un attributo se

$$f_s < 20\%$$

3. Selettività dell'indice

STUDENTI(MATR,Nome,DataNascita,Genere,Indirizzo)

ESAMI(MATR,Corso,Voto,DataEsame)

- Un indice sulla chiave MATR ha un'altissima selettività, quindi va bene.
- Può avere senso mettere un indice sulle date di nascita.
- Non ha nessun senso mettere un indice sul genere (solo due valori possibili): il DBMS non userebbe mai questo indice.

4. Evitare indici su attributi sbilanciati

Bisogna fare attenzione alla distribuzione dei valori:

- Se c'è un grosso squilibrio nella distribuzione dei valori di una chiave, gli indici su quella chiave saranno poco efficienti.

- Infatti gli indici funzionano bene quando la distribuzione è uniforme.

5. Quanti indici definire?

- Non ci sono indicazioni di carattere generale.
- Gli indici selettivi favoriscono le interrogazioni ma la loro manutenzione costa.
- Un’indicazione di massima è di limitarsi al massimo a 4/5 indici per tabelle corpose.

6. Indici sulle chiavi

Un indice su una chiave è sempre consigliato (infatti in genere non serve crearli perché i DBMS creano automaticamente indici sugli attributi unique).

Analogamente si consiglia di definire indici su chiavi esterne perché possono venire usati dai join.

Esempio:

$$studenti \bowtie_{studenti.MATR=esami.MATR} esami$$

In ESAMI, MATR non è chiave relazionale (la chiave è MATR,Corso) e un indice su ESAMI.MATR potrebbe, dato uno studente, ottenere i suoi esami velocemente.

7. Scansione ordinata

- Un indice è utile anche per reperire i record secondo l'ordine della chiave indicizzata.
- Infatti quando consideriamo le foglie di un B+-tree, i valori delle chiavi sono sempre ordinati per definizione.
- Di conseguenza, se si vuole esplorare l'area dati secondo l'ordine della chiave indicizzata *k*, è sufficiente eseguire una scansione sequenziale delle foglie.

8. Conoscenza del DBMS

Si raccomanda di scegliere gli indici conoscendo a fondo le strategie di ottimizzazione del DBMS.

È l'ottimizzatore del DBMS a scegliere se utilizzare o meno l'indice e ogni DBMS ha una propria strategia.