# Organizzazione fisica e gestione delle interrogazioni

**Capitolo 11 Atzeni-Ceri**


Basi di dati - 4e
Paolo Atzeni, Stefano Ceri, Piero Fraternali, Stefano Paraboschi, Riccardo Torlone

# Tecnologia delle BD: perché studiarla?

- I DBMS offrono i loro servizi in modo "trasparente":
  - ignorare molti aspetti realizzativi
  - DBMS come una "scatola nera"

- Perché caprirla?
  - capire come funziona può essere utile per un migliore utilizzo
  - alcuni servizi sono offerti separatamente

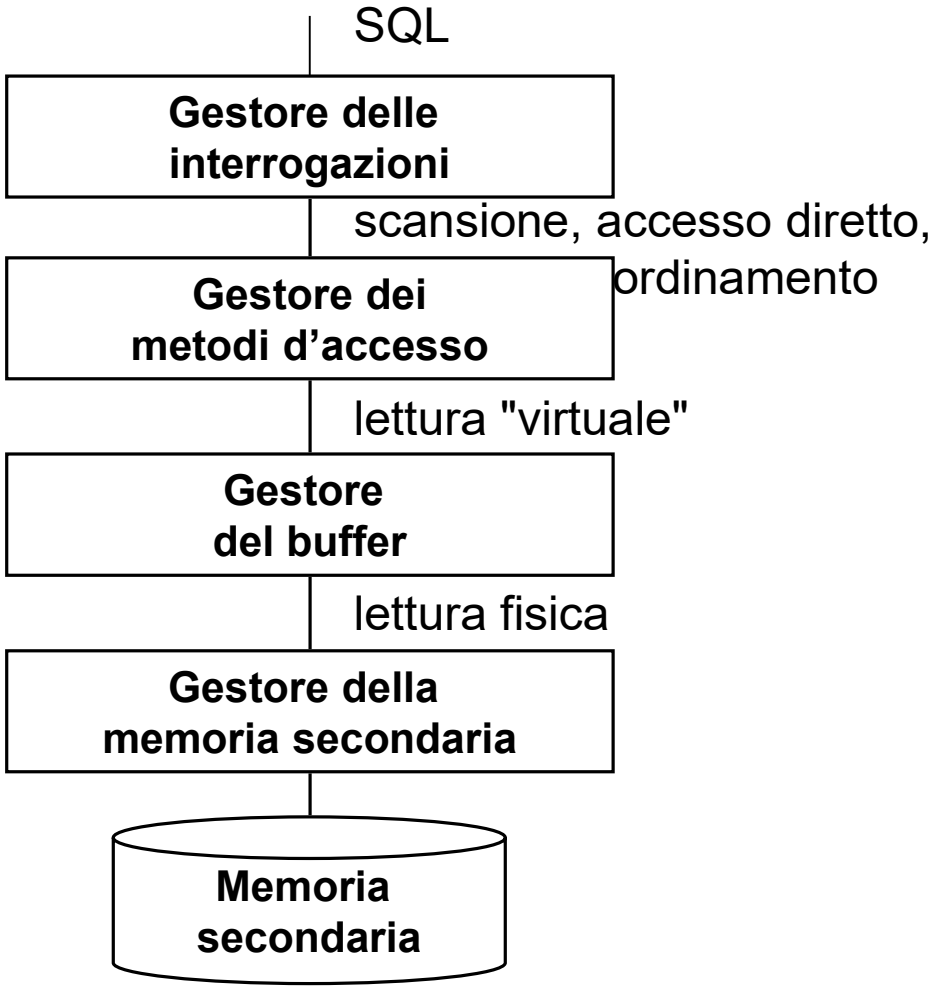
## Le basi di dati sono grandi e persistenti

- 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 …

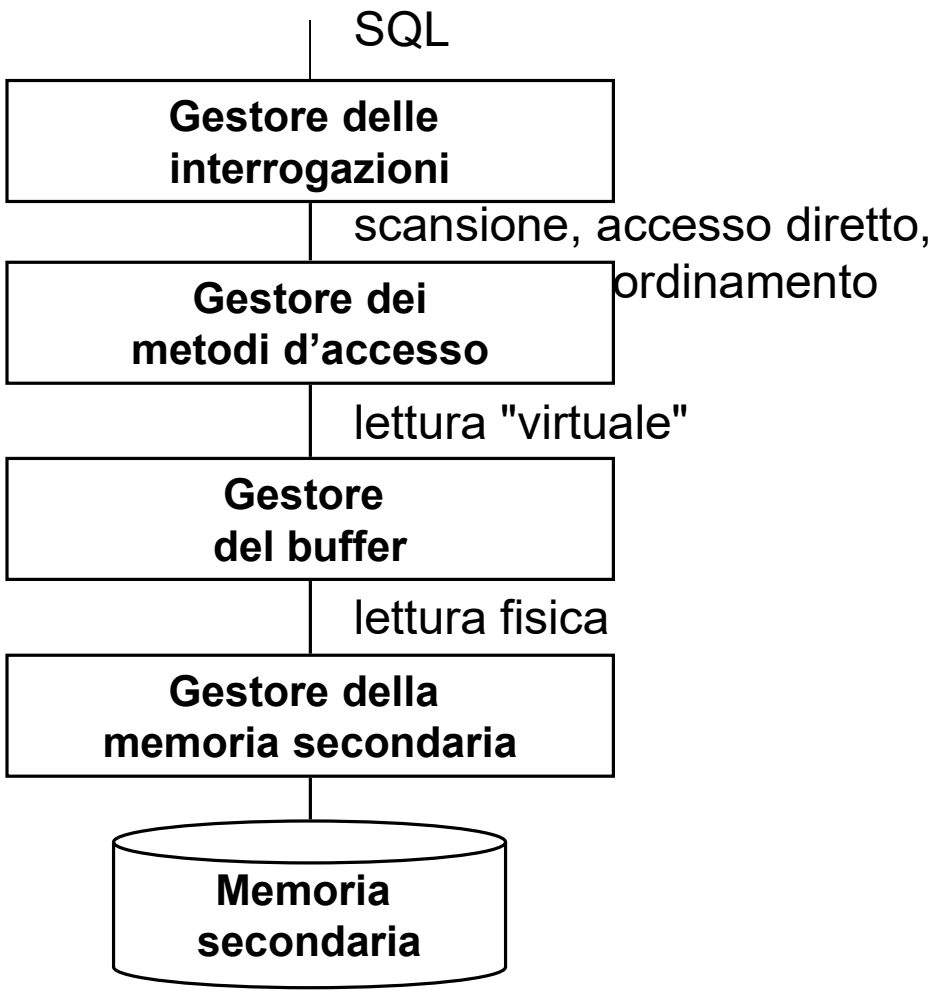
- L'affidabilità è impegnativa per via degli aggiornamenti frequenti e della necessità di gestire il buffer

# Tecnologia delle basi di dati, argomenti

- Gestione della memoria secondaria e del buffer
- Organizzazione fisica dei dati
- Gestione ("ottimizzazione") delle interrogazioni
- Controllo della affidabilità
- Controllo della concorrenza

- Architetture distribuite

# Gestore degli accessi e delle interrogazioni



- L’insieme dei dati che costituiscono una base di dati (DB) è composto da record strutturati in uno o piu` campi. Su tale insieme di dati, il DBMS deve poter eseguire con facilità operazioni di interrogazione e di aggiornamento
- Il DB `e fisicamente memorizzato su un computer mediante opportuni dispositivi di memorizzazione:
  - Primari. Consentono un rapido accesso ai dati, ma sono di dimensioni ridotte. Possono essere utilizzati direttamente dalla CPU (memoria primaria, memoria cache, etc.).
  - Secondari (e terziari). Hanno elevate capacità di memorizzazione e bassi costi, ma tempi di accesso decisamente più elevati. Comprendono dischi magnetici, dischi ottici, nastri magnetici, etc.

# 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

# File di record

I dati memorizzati su disco sono organizzati in **file di record**.

Ogni record è una collezione di dati interpretabile come un insieme di fatti relativi a entità, loro attributi e loro relazioni (ad esempio, una tupla può essere memorizzata come un record nel quale il valore di un attributo può essere memorizzato in un campo del record)

Esistono diverse tecniche di memorizzazione fisica dei record di un file su un disco magnetico:

1. **Heap file** (file non ordinato) posiziona i record sul disco in modo non ordinato
2. **Sorted file** (file sequenziale) ordina i record secondo il valore di un particolare campo
3. **Hashed file** usa una funzione di hash per determinare la posizione dei record sul disco

# 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:

$$bfr = \left\lfloor \frac{L_B}{L_R} \right\rfloor$$

Esempio: se B = 120 e R = 15, allora bfr = 8

- lo spazio residuo può essere
  - utilizzato (record "spanned" o impaccati)
  - non utilizzato ("unspanned")

# Considerazioni

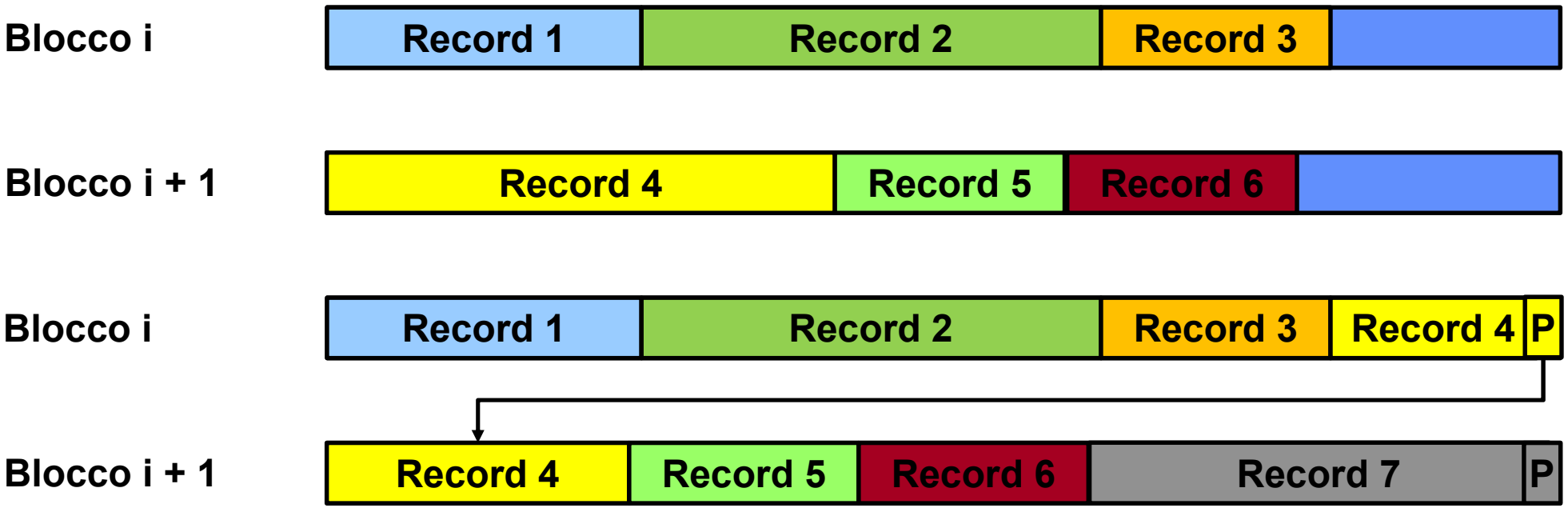
- In generale, B può non essere divisibile per R.
  - Se imponiamo che ogni record sia interamente contenuto in uno e un solo blocco, ogni blocco conterrà dello spazio inutilizzato pari a:

$$B - (bfr \cdot R) \text{ byte}$$

- Esempio:
  - se B = 120 e R = 25, allora bfr = 4 e B − (bfr · R) = 20 (più del 15% dello spazio viene sprecato)

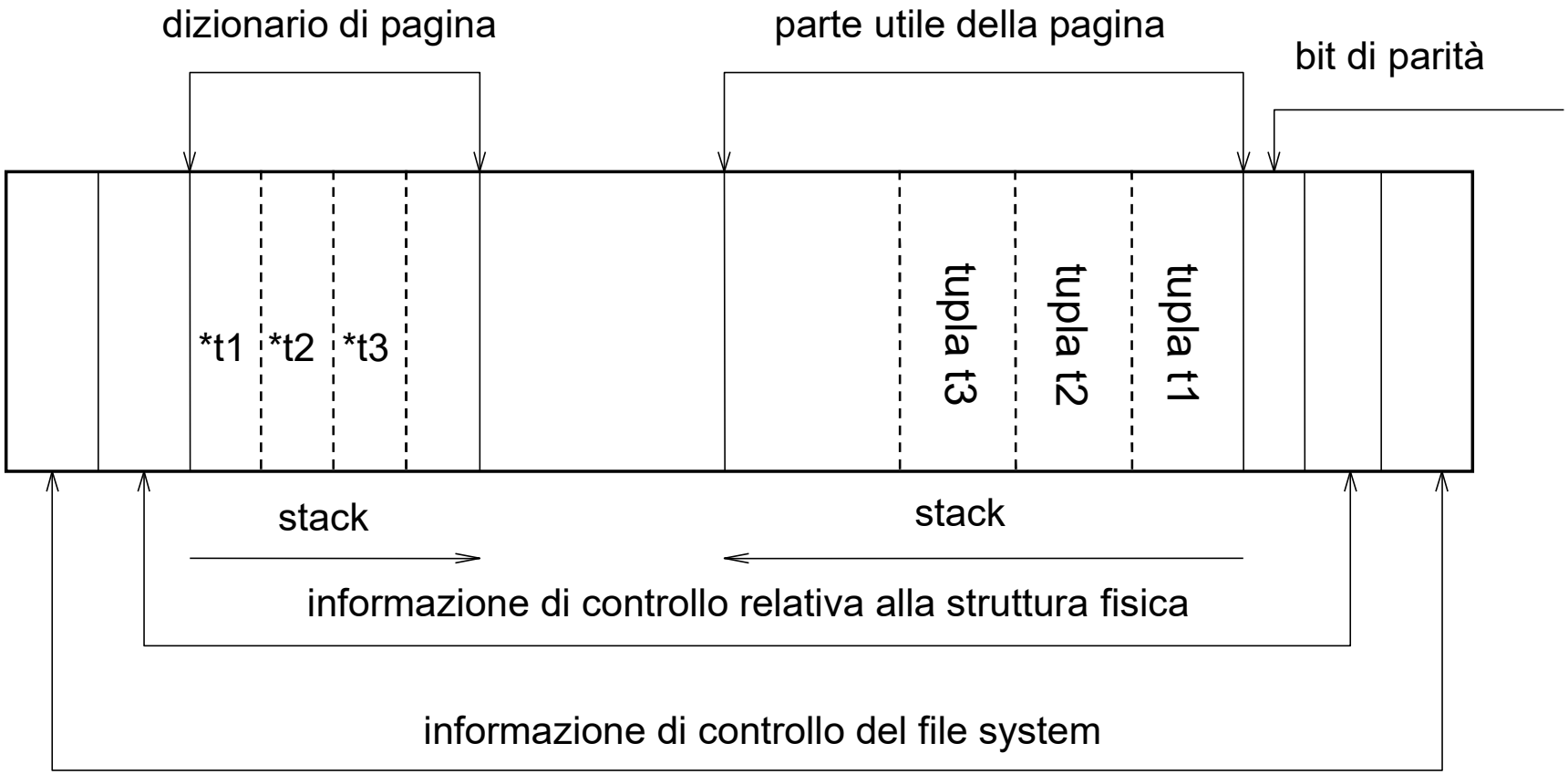
- Se la dimensione dei record è comparabile a quella del blocco (esempio, B = 120 e R = 70), il vincolo proposto risulta inaccettabile.
- Se la dimensione del record è maggiore di quella del blocco, ossia la condizione B ≥ R non risulta più soddisfatta (esempio, B = 120 e R = 125), diventa impossibile memorizzare i record.

- Una soluzione alternativa consiste nel memorizzare un record in due (o più) blocchi (spanned record):
  - Per esempio, se ogni record è distribuito su al più due blocchi, potremmo utilizzare un puntatore alla fine del primo blocco per individuare il blocco che contiene la parte rimanente del record (a meno che i due blocchi non siano uno consecutivo all'altro).
- Nel caso in cui la suddivisione di un record in più blocchi sia vietata (**unspanned** record), si parla di organizzazione di tipo unspanned.



# Organizzazione delle tuple nelle pagine

- Varie alternative legate a:
  - Metodi di accesso
  - Lunghezza delle tuple (fissa o variabile)
  - Spanning delle tuple (spanned/unspanned record)
- Esempio:



- Noti la dimensione del blocco, la dimensione del record e il numero di record contenuti nel file (tale numero può variare nel tempo; occorre, quindi, disporre di una stima attendibile del numero medio/massimo di record del file), è possibile determinare il numero *nb* di blocchi necessari per memorizzare il file.
- Assumiamo un'organizzazione dei record di tipo unspanned.
- Siano *r* e *bfr* il numero di record del file e il fattore di blocco, rispettivamente. Il numero di blocchi necessari  pari a:

$$nb = \left\lceil \frac{r}{bfr} \right\rceil$$

# Allocazione blocchi

Esistono diverse tecniche consolidate per effettuare l'allocazione dei blocchi di un file su disco.

1. **Allocazione contigua**. I blocchi del file sono allocati in blocchi consecutivi sul disco. Vantaggio: lettura rapida dell'intero file con la tecnica del doppio buffering. Svantaggio: espansione difficoltosa.
2. **Allocazione con collegamenti**. Ogni blocco del file contiene un puntatore al blocco successivo. Vantaggio: espansione estremamente semplice. Svantaggio: scansione/lettura dell'intero file dispendiosa/ lenta.
3. **Combinazione** delle due precedenti soluzioni. Vengono definiti e allocati cluster (detti anche segmenti o extent) di blocchi consecutivi su disco, collegati fra loro tramite puntatori.
4. **Allocazione indicizzata**. Uno o più blocchi di indici contengono i puntatori agli effettivi blocchi del file.

Di solito non è possibile scegliere come allocare i blocchi ma dipende dal filesystem e quindi dal sistema operativo.

# File header

Un **file header** (o descrittore del file) contiene le informazioni necessarie per determinare gli indirizzi sul disco dei blocchi del file.

Descrizioni relative al formato dei record:

- record di lunghezza fissa (e unspanned): lunghezza dei campi e ordine dei campi nel record;
- record di lunghezza variabile: codici tipo dei campi, caratteri di separazione, codici tipo dei record.

# Ricerca su disco

- La ricerca di un record su disco prevede di copiare uno o più blocchi nel buffer della memoria principale.
- Specifici programmi cercano all'interno dei buffer il/i record desiderati, utilizzando l'informazione contenuta nel file header.
- Se l'indirizzo del blocco che contiene il record voluto non è noto, i programmi di ricerca devono eseguire una **ricerca lineare** attraverso i blocchi dei file:
  - ogni blocco del file è copiato nel buffer, quindi si cerca nel blocco il record desiderato, se il record è stato trovato si restituisce tale record, altrimenti si passa al blocco successivo.
  - La ricerca, quindi, termina quando viene individuato il record desiderato o sono stati controllati tutti i blocchi e non è stato individuato nulla.
  - Tale ricerca risulta estremamente dispendiosa per file di notevole lunghezza.

- Le operazioni sui file possono essere suddivise in due categorie:
  - **operazioni di recupero dei dati (ricerca)**: non modificano i dati presenti nel file, ma si limitano a cercare record, esaminare, ed eventualmente elaborarne il contenuto.
  - **operazioni di aggiornamento**: modificano il file, inserendo o cancellando record o modificando il valore di alcuni dei loro campi.
- In entrambi i casi, è necessario selezionare uno o più record sulla base di un'opportuna **condizione di selezione**.

# Esempio

- Consideriamo un file IMPIEGATO con campi:
  - CF, STIPENDIO, CODICE LAVORO e DIPARTIMENTO.

- Due esempi di semplici condizioni di selezione, che utilizzano gli operatori di confronto (=, >, <, . . .) applicati ai valori di alcuni campi dei record del file, sono:

  - CF = 'FLSSRA50B28C351G'

  - DIPARTIMENTO = 'ricerca' AND STIPENDIO > 30000

# Operazioni di ricerca

- Le operazioni di ricerca su file sono generalmente basate su semplici condizioni di selezione.
- Operazioni di selezione complesse vengono decomposte in operazioni di selezione semplici che possono essere usate per localizzare i record sul disco.
- Esempio:
  - Immaginiamo di dover selezionare tutti gli impiegati del dipartimento Ricerca che percepiscono uno stipendio maggiore di 30000 euro.
  - La nostra query richiede di trovare i record per cui STIPENDIO > 30000 AND DIPARTIMENTO = 'ricerca'.
  - Possiamo decomporre la ricerca e valutare preliminarmente la condizione semplice DIPARTIMENTO = 'ricerca'.
  - Successivamente, ogni record che soddisfa tale condizione verrà analizzato per verificare se soddisfa anche l'altra condizione (STIPENDIO > 30000).

- Quando più record del file soddisfano la condizione di ricerca, si individua solo il primo record (record corrente).

- Per selezionare altri record che soddisfano la condizione, è necessario utilizzare operazioni addizionali.

- Le operazioni di ricerca successive iniziano dal record corrente.

# 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

# Heap file

- Nel caso più semplice di organizzazione, i record sono posizionati nel file nell'ordine in cui vengono inseriti
  - ogni nuovo record è inserito alla fine del file.
  - un file organizzato in questo modo è detto **heap file**.
- L'inserimento di un nuovo record può essere effettuato in modo molto efficiente:
  1. l'ultimo blocco del file (su disco) è copiato in un buffer;
  2. il nuovo record è aggiunto nel blocco contenuto nel buffer;
  3. il blocco aggiornato è riscritto su disco.

  L'indirizzo dell'ultimo blocco del file è mantenuto nel file header.

# 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

# Indici primari

- Indici primari.
  - Un indice primario è un indice costruito rispetto al campo chiave di un file di record che è fisicamente ordinato rispetto a tale campo (ordering key field).

- Si noti che:
  1. Il campo chiave è utilizzato per ordinare fisicamente i record del file sul disco (ordering key field);
  2. ogni record è identificato univocamente dal valore che assume su tale campo (ordering key field).

- Un indice primario è un file ordinato i cui record, di lunghezza prefissata, possiedono due campi:

(⟨chiave primaria⟩; ⟨indirizzo blocco⟩)

- Nel file indice esiste una index entry (o index record) *i* per ogni blocco *b* del file di dati:

⟨k(i); p(i)⟩

- dove k(i) è un valore minore (o uguale) a quello di tutte le chiavi dei record in *b*, ma maggiore del valore di tutte le chiavi del blocco che precede *b*, e p(i) è l'indirizzo del blocco b.

- Spesso si assegna a $k(i)$ il valore minimo delle chiavi presenti in $b$ (valore della chiave del primo record del blocco).

- Se $b$ e il primo blocco, conviene scegliere $k(i) = -\infty$
  - dove $-\infty$ rappresenta un valore inferiore a quello di ogni possibile chiave.

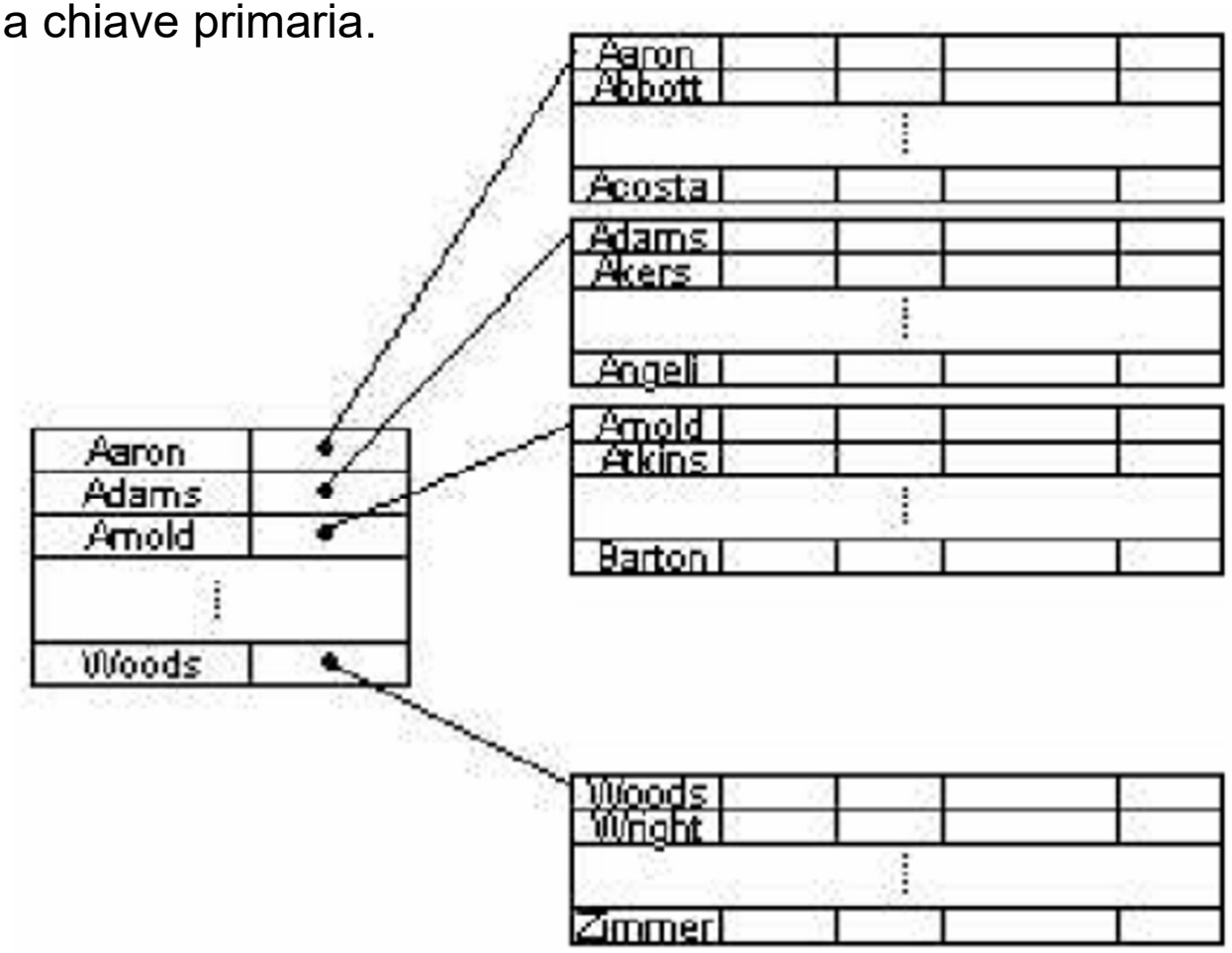
- Si noti che
  - il primo campo del file indice è una **chiave** per tale file
  - il file indice è **ordinato** rispetto ad essa.

# Esempio

Supponiamo di avere un file di dati che rappresenti le tuple della relazione

IMPIEGATO(NOME, DATA_NASCITA, LAVORO, STIPENDIO, SESSO)

Assumiamo che NOME sia la chiave primaria.



- Il primo record di ogni blocco del file di dati è definito anchor record del blocco (**block anchor**).
- Un indice primario è un indice **non denso** in quanto include una entry per ogni blocco (un indice è detto denso se contiene una entry per ogni record del file).
- Un record con valore della chiave primaria uguale a $k$ si trova nel blocco di indirizzo $p(i)$, con

$$k(i) \leq k < k(i + 1)$$

- Per recuperare un record, noto il valore $k$ della sua chiave primaria, occorre effettuare una ricerca binaria sul file indice per cercare l'appropriata index entry $i$ e quindi recuperare il blocco il cui indirizzo è $p(i)$.

# Esempio

- Supponiamo di avere un file ordinato contenente *r* = 30000 record memorizzati su un disco con dimensioni del blocco *B* = 1024 byte.
- I record siano di lunghezza fissa *R* = 100 byte e unspanned.
- Il fattore di blocco per il file è:

$$bfr = \left\lfloor \frac{B}{R} \right\rfloor = \left\lfloor \frac{1024}{100} \right\rfloor = 10$$

- Il numero di blocchi necessari per memorizzare il file è:

$$nb = \left\lfloor \frac{r}{brf} \right\rfloor = \left\lfloor \frac{30000}{10} \right\rfloor = 3000 \textit{ blocchi}$$

- Una ricerca binaria sul file di dati richiede approssimativamente:

$$\log_2 nb = \log_2 3000 = 12 \text{ accessi}$$

# Esempio

- Supponiamo ora che il campo contenente la ordering key del file sia lungo *V* = 9 byte e che il puntatore ad un blocco sia lungo *P* = 6 byte. La dimensione di ogni index entry *Ri è*:

*Ri* = 9 + 6 = 15 byte*;*

- mentre il fattore di blocco è:

$$bfr_i = \left\lfloor \frac{B}{R_i} \right\rfloor = \left\lfloor \frac{1024}{15} \right\rfloor = 68$$

- Il numero totale di index entry *Ri* è uguale al numero di blocchi del file, che è 3000. Il numero di blocchi necessario per l'indice è:

$$nb_i = \left\lfloor \frac{r_i}{bfr_i} \right\rfloor = \left\lfloor \frac{3000}{68} \right\rfloor = 45$$

- Una ricerca binaria sul le indice richiede:

$$\lceil \log_2 nb_i \rceil = \lceil \log_2 45 \rceil = 6$$

- Conclusione:
  - per cercare un record utilizzando l'indice occorrono 6 accessi a blocchi dell'indice più 1 accesso al blocco del file contenente il record, per un totale di 6 + 1 = 7 accessi, contro i 12 accessi richiesti da una ricerca binaria sul le dei dati (senza uso dell'indice).

- Il problema principale degli indici primari (analogamente a quanto accadeva con i file ordinati) è quello dell'inserimento e della cancellazione di record, con in più l'onere di dover modificare anche le index entry.
- Per ovviare a tale problema, si può utilizzare un file di overflow non ordinato, oppure aggiungere una lista di record di overflow ad ogni blocco del file di dati.

# Indici secondari

- Un indice secondario è un file ordinato con due campi:

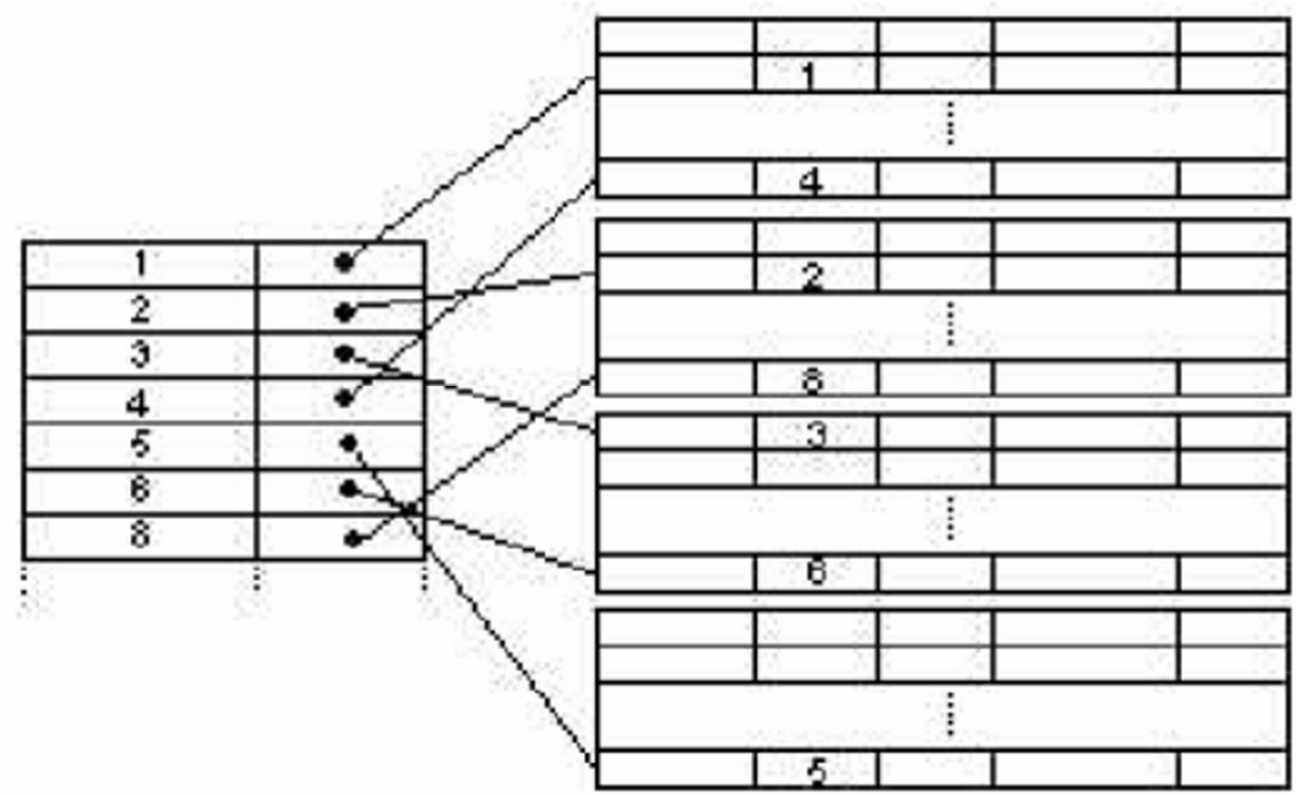
  (⟨indexing feld⟩; ⟨indirizzo blocco⟩)

- A differenza degli indici primari, è possibile associare più indici secondari (e quindi indexing field) ad uno stesso file.
- Consideriamo una struttura con accesso attraverso indici secondari su un campo chiave (campo che assume un valore diverso su ogni record del file di dati - secondary key).
- Vi sarà una index entry per ogni record del file di dati, contenente il valore della chiave secondaria per il record ed un puntatore al blocco in cui il record è memorizzato (puntatore al blocco anziché al record..).

# Indici secondari

- Un indice secondario su un key feld è un indice denso (a differenza degli indici primari, gli indici secondari contengono una entry per ogni record, e non per ogni blocco, del file di dati)
  - poiché i record del file di dati non sono fisicamente ordinati rispetto al valore del secondary key field, non è possibile utilizzare i block anchor.
  - quando un blocco è trasferito in memoria primaria, bisogna effettuare una ricerca del record desiderato.
- Un indice secondario richiede molto più spazio ed un tempo di ricerca maggiore di un indice primario in ragione dell'elevato numero di entry (indice denso).
- Tuttavia, la riduzione del tempo di ricerca di un record sulla base del valore da esso assunto sull'indexing field è molto significativa (in questi casi, disponendo del solo indice primario, occorre eseguire una scansione lineare).

# Indici secondari



# Esempio

- Supponiamo di avere un file ordinato con *r* = 30000 record, memorizzati su un disco con dimensione del blocco pari a *B* = 1024 byte. I record siano di lunghezza fissa *R* = 100 byte e unspanned. Come visto in precedenza, il fattore di blocco e *bfr* = 10 record per blocco, e il numero di blocchi necessari per memorizzare il le e *nb* = 3000.
- Una ricerca lineare sul le di dati richiede mediamente

$$\frac{nb}{2} = \frac{3000}{2} = 1500 \text{ accessi}$$

- Assumiamo che la dimensione di ogni index entry *Ri* sia uguale a 9 + 6 = 15 byte ed il fattore di blocco *bfri* sia di 68 entry per blocco.

- Trattandosi di indice denso, il numero totale di index entry *ri* e uguale al numero di record del file, che e 30000. Ne consegue che il numero di blocchi necessario per l'indice è:

$$nb_i = \left\lceil \frac{r_i}{bfr_i} \right\rceil = \left\lceil \frac{30000}{68} \right\rceil = 442$$

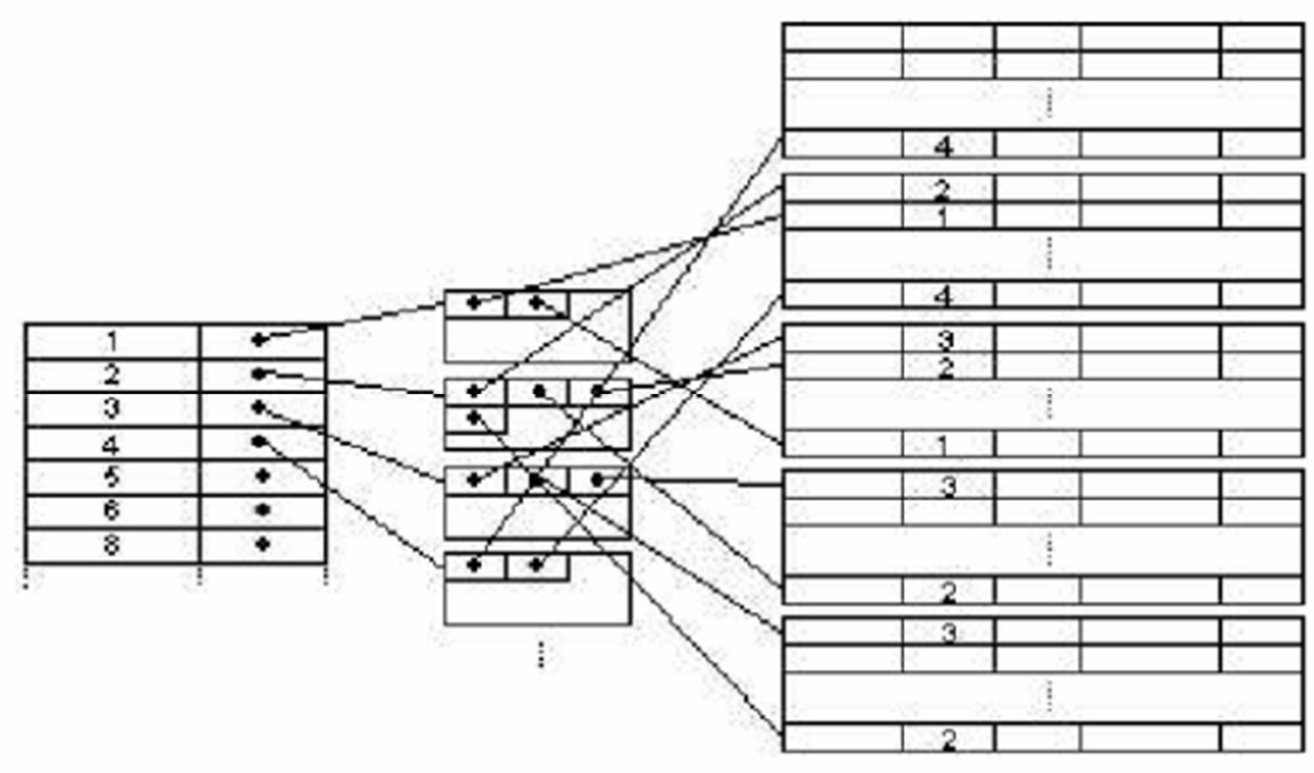
- contro i 45 blocchi necessari nel caso di indice primario.
- Una ricerca binaria sul le indice richiede:

$$\lceil \log_2 nb_i \rceil = \lceil \log_2 442 \rceil = 9$$

accessi a blocco.

- Per cercare un record utilizzando l'indice (secondario), occorre 1 accesso addizionale ad un blocco del le per un totale di 9 + 1 = 10 accessi, contro i 1500 accessi necessari in media per una ricerca lineare (poco più dei 7 accessi richiesti dalle ricerche che sfruttano l'indice primario).

- E’ possibile creare un indice secondario su un campo non chiave. Diversi record possono assumere lo stesso valore sul campo non 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)

# Indici multilivello

- L'idea di utilizzare indici multilivello e nata dalla necessita di ridurre ad ogni passo la parte dell'indice che rimane da analizzare di un fattore pari a *bfri* (che e abitualmente > 2). Il valore *bfri* è detto fan-out (*fo*) dell'indice multilivello.

- Un indice multilivello considera il file indice (primo livello) come un file ordinato con un valore distinto per ogni $k(i)$ e crea un indice primario per il primo livello (**secondo livello**).
- Poiché il secondo livello è un indice primario, si possono in realtà usare dei block anchor in modo che il secondo livello abbia una entry per ogni blocco (e non per ogni entry) del primo livello.
- Il *bfri* per il secondo livello (e per tutti i livelli successivi) è lo stesso dell'indice del primo livello, poiché tutte le index entry hanno la stessa dimensione (ognuna ha un ⟨field value⟩ e un ⟨indirizzo blocco⟩).

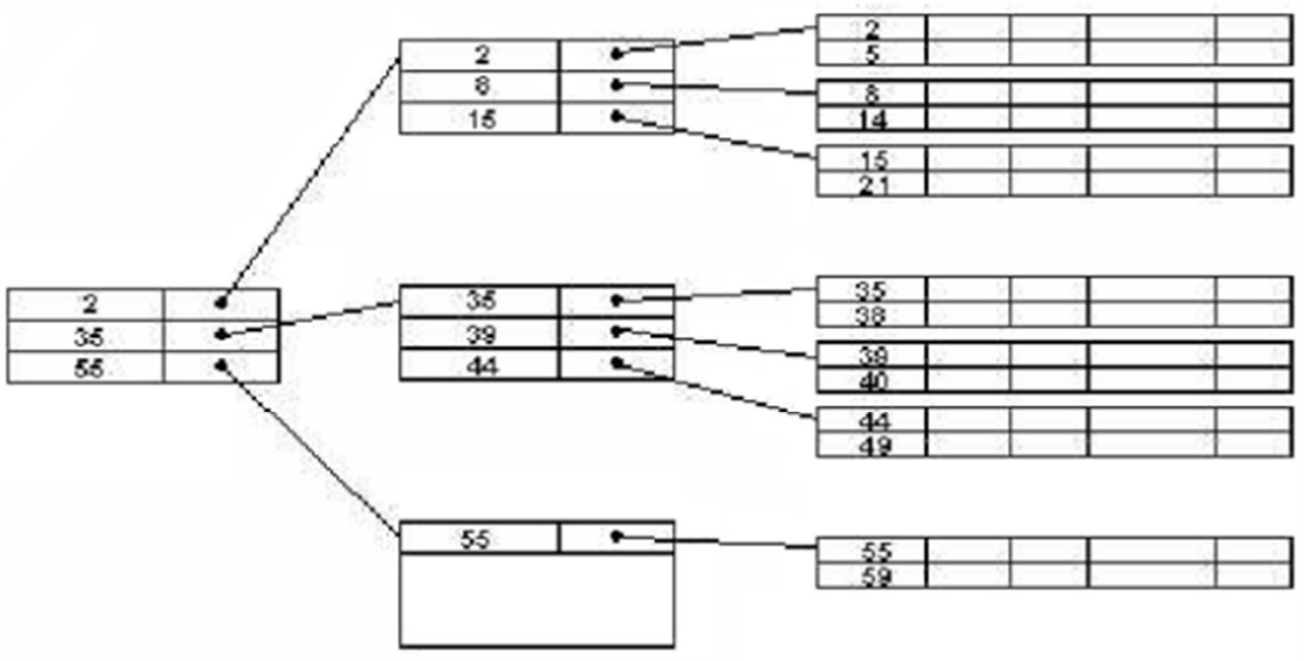
- Se il primo livello ha *r*1 entry e *bfri* = *fo*, allora il primo livello ha bisogno di $\left\lceil \frac{r1}{fo} \right\rceil$blocchi, che è il numero di entry *r*2 necessarie al secondo livello dell'indice.
- Si può ripetere questo processo per il secondo livello. Il **terzo livello**, che è un indice primario per il secondo livello, ha una entry per ogni blocco di secondo livello.
- Il numero di entry al terzo livello e: *r*3 = $\left\lceil \frac{r2}{fo} \right\rceil$

- **Osservazione**. E' richiesto un secondo livello solo se il primo livello richiede più di un blocco di spazio su disco; è richiesto un terzo livello solo se il secondo richiede più di un blocco, e cos via.
- Si ripete il procedimento fino a quando tutte le entry di un certo index level *t* (**top index level**) stanno in un singolo blocco.
- Ogni livello riduce il numero di entry del livello precedente di un fattore *fo*. Il top index level *t* si può calcolare a partire dal vincolo:

$$1 \geq r_1 / fo^t$$

- Un indice multilivello con *r*1 entry al primo livello avrà approssimativamente *t* livelli, dove

$$t = \lceil \log_{fo} r_1 \rceil$$


2
35
55
2
8
15
35
39
44
55
2
5
8
14
15
21
35
38
39
40
44
49
55
59

- Supponiamo di avere un file non ordinato con *r* = 30000 record, memorizzati su un disco con dimensione del blocco pari a *B* = 1024 byte. I record siano di lunghezza ssa *R* = 100 byte e unspanned.
- Come visto in precedenza, il blocking factor e *bfr* = 10 record per blocco e il numero di blocchi necessari per memorizzare il file è *nb* = 3000.
- Come in precedenza, assumiamo che la dimensione di ogni index entry *Ri* sia uguale a 9 + 6 = 15 byte ed il blocking factor *bfri* sia di 68 entry per blocco.
- Vogliamo ora convertire tale file con un indice secondario denso in un **indice multilivello**. Il blocking factor *bfri* è anche il fan-out dell'indice multilivello. Il numero di blocchi di primo livello è *b*1 = 442, mentre quello dei blocchi di secondo livello e:

$$b_2 = \left\lceil \frac{b_1}{fo} \right\rceil = \left\lceil \frac{442}{68} \right\rceil = 7\ blocchi$$

Quello di terzo livello è

$$b_3 = \left\lceil \frac{b_2}{fo} \right\rceil = \left\lceil \frac{7}{68} \right\rceil = 1\ blocco$$

- Il terzo livello e il top level dell'indice e $t = 3$.
- Accesso con indice secondario:

$$\lceil \log_2 nb_i \rceil = \lceil \log_2 442 \rceil = 9$$

- più un accesso a un blocco del file di dati.
- Per accedere ad un record utilizzando l'indice multilivello, bisogna accedere ad un blocco per ogni livello più l'accesso al blocco del file di dati. Ne segue che il numero totale di accessi a blocchi e pari a:

$$t + 1 = 3 + 1 = 4$$

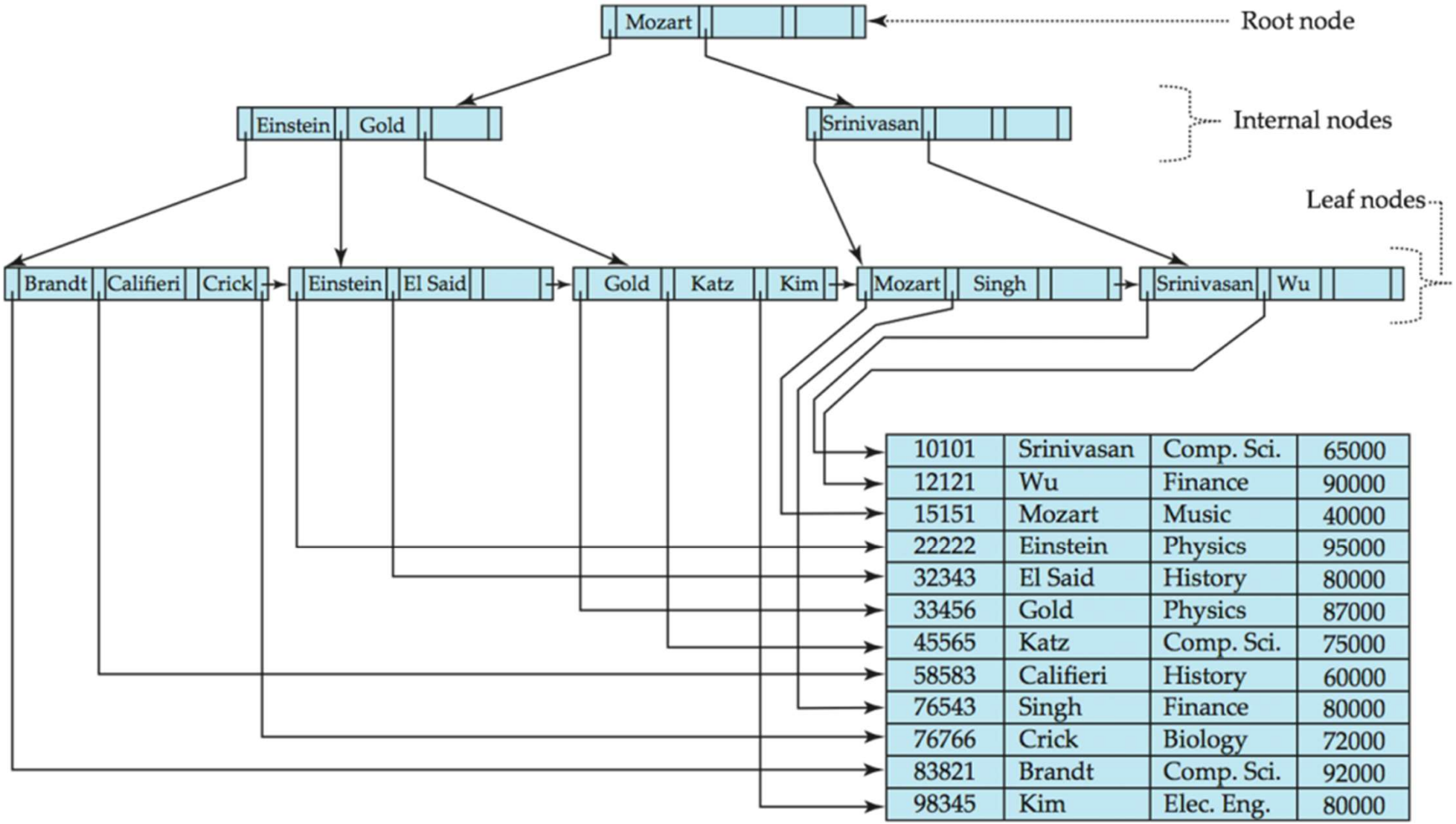
# Strutture fisiche nei DBMS relazionali

- 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)

# B-tree

- I database moderni usano indici basati sui B-tree.
- I B-tree sono alberi di ricerca (similarmente agli alberi binari) che sono stati pensati per lavorare in memoria secondaria (ridurre I/O al minimo):
  - Ogni nodo dell'albero ha la dimensione di una pagina
  - Ogni nodo può contenere più elementi
  - Tutti i nodi (tranne la radice) devono avere almeno 2 e fino a k elementi (chiamati chiavi)
  - Ogni nodo contiene fino a k+1 puntatori a nodi figli
  - Gli elementi all'interno di ogni nodo sono ordinati e anche i figli sono ordinati come avviene per gli alberi binari
  - Per ovviare ai problemi degli alberi binari, i B-tree implementano algoritmi di bilanciamento automatico in inserimento e cancellazione.
- I B-tree possono essere utilizzati per memorizzare sia gli indici secondari che gli indici primari e gli interi record (per implementare i file ordinati di cui abbiamo parlato prima - SQLlite)

# B-tree




| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 80000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 58583 | Califieri | History | 60000 |
| 76543 | Singh | Finance | 80000 |
| 76766 | Crick | Biology | 72000 |
| 83821 | Brandt | Comp. Sci. | 92000 |
| 98345 | Kim | Elec. Eng. | 80000 |

# 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

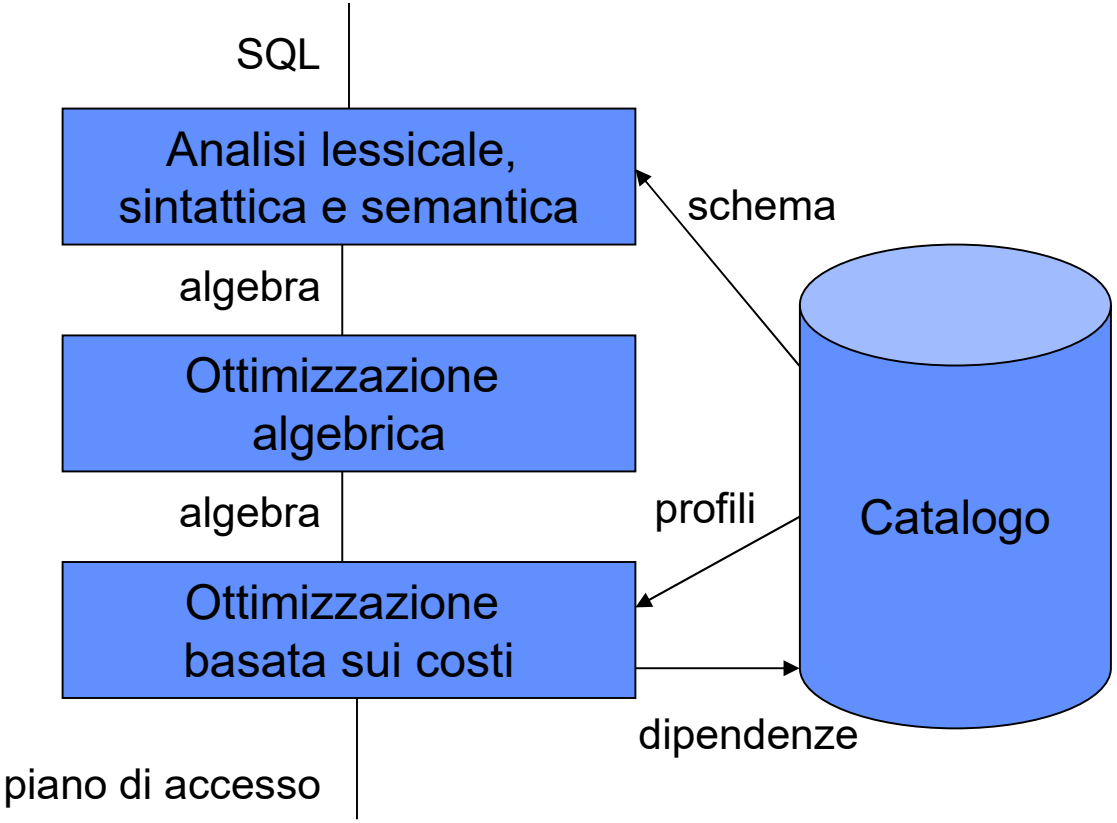
# Definizione degli indici in SQL

- 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

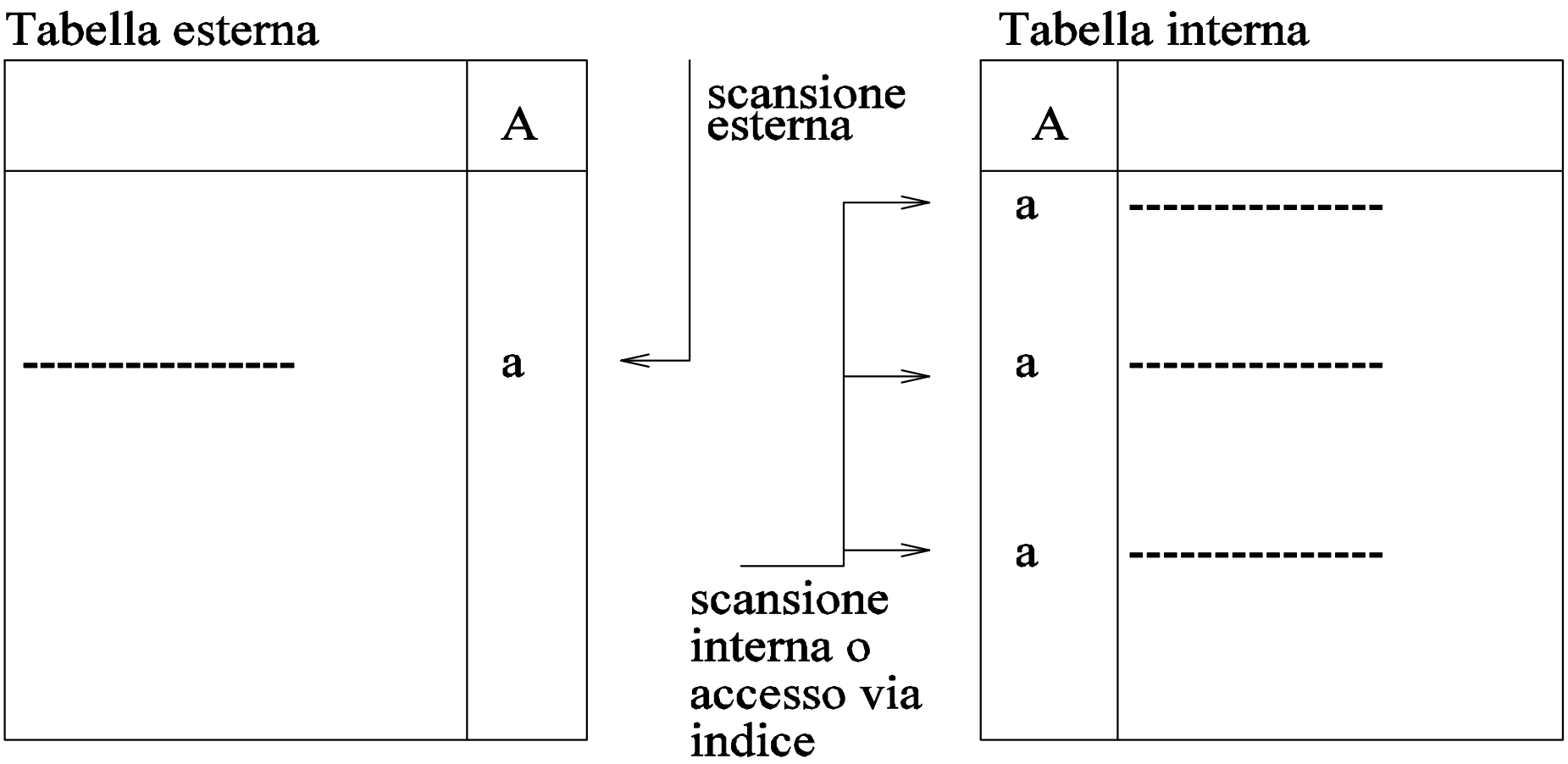


Tabella esterna
A
a
Tabella interna
A
a
a
a
scansione esterna
scansione interna o accesso via indice

# Merge-scan

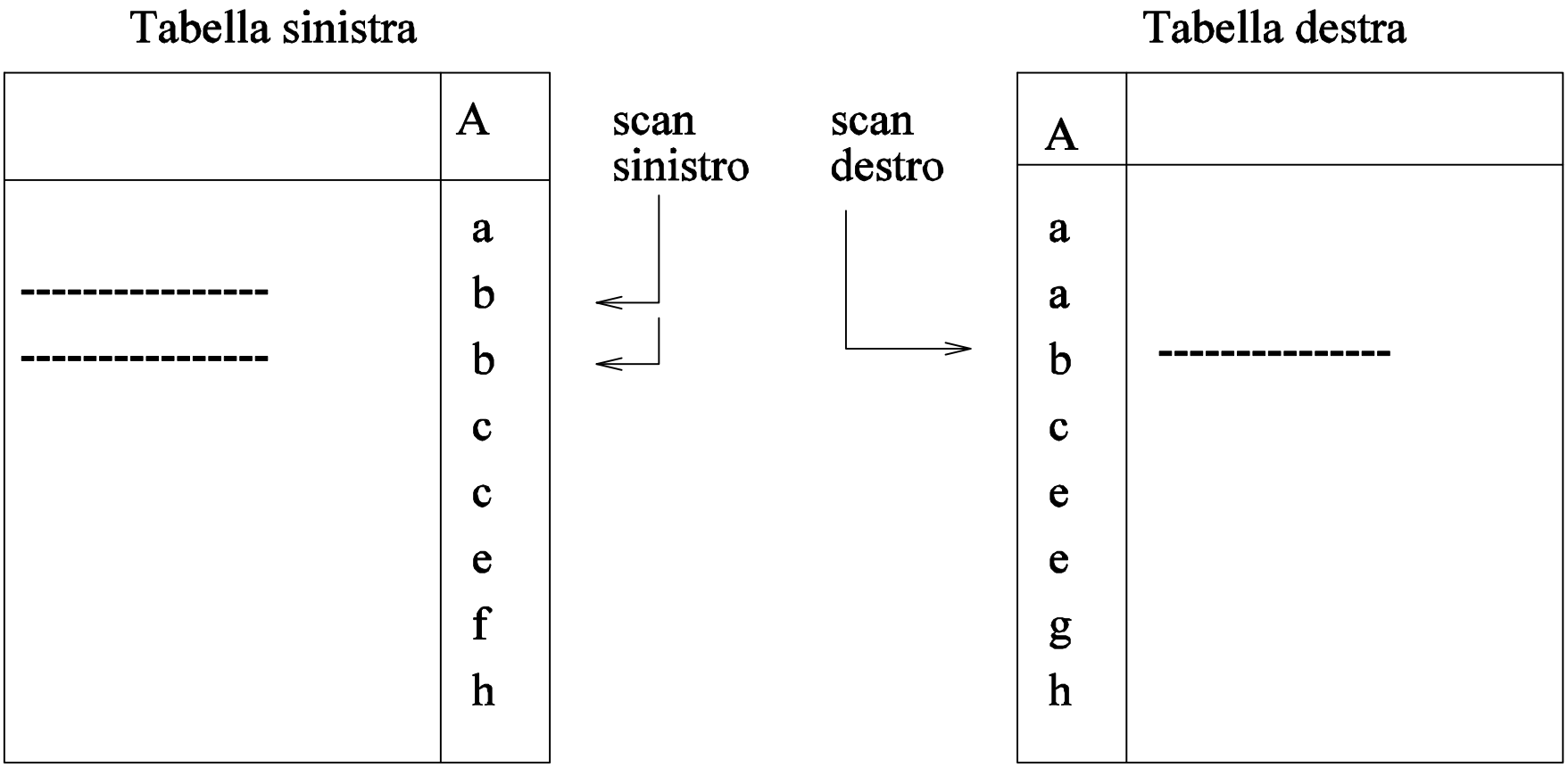
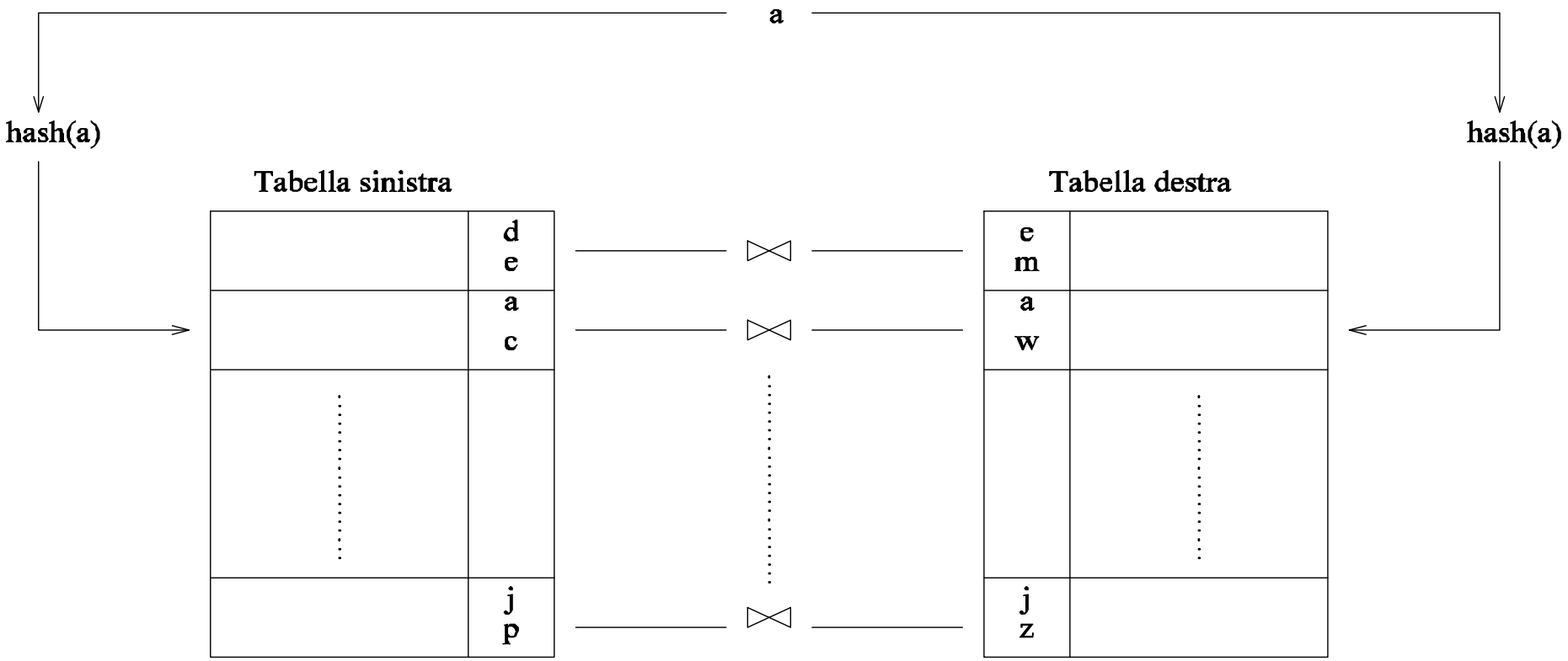


Tabella sinistra
A
a
b
b
c
c
e
f
h
scan
sinistro
scan
destro
Tabella destra
A
a
a
b
c
e
e
g
h

# 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

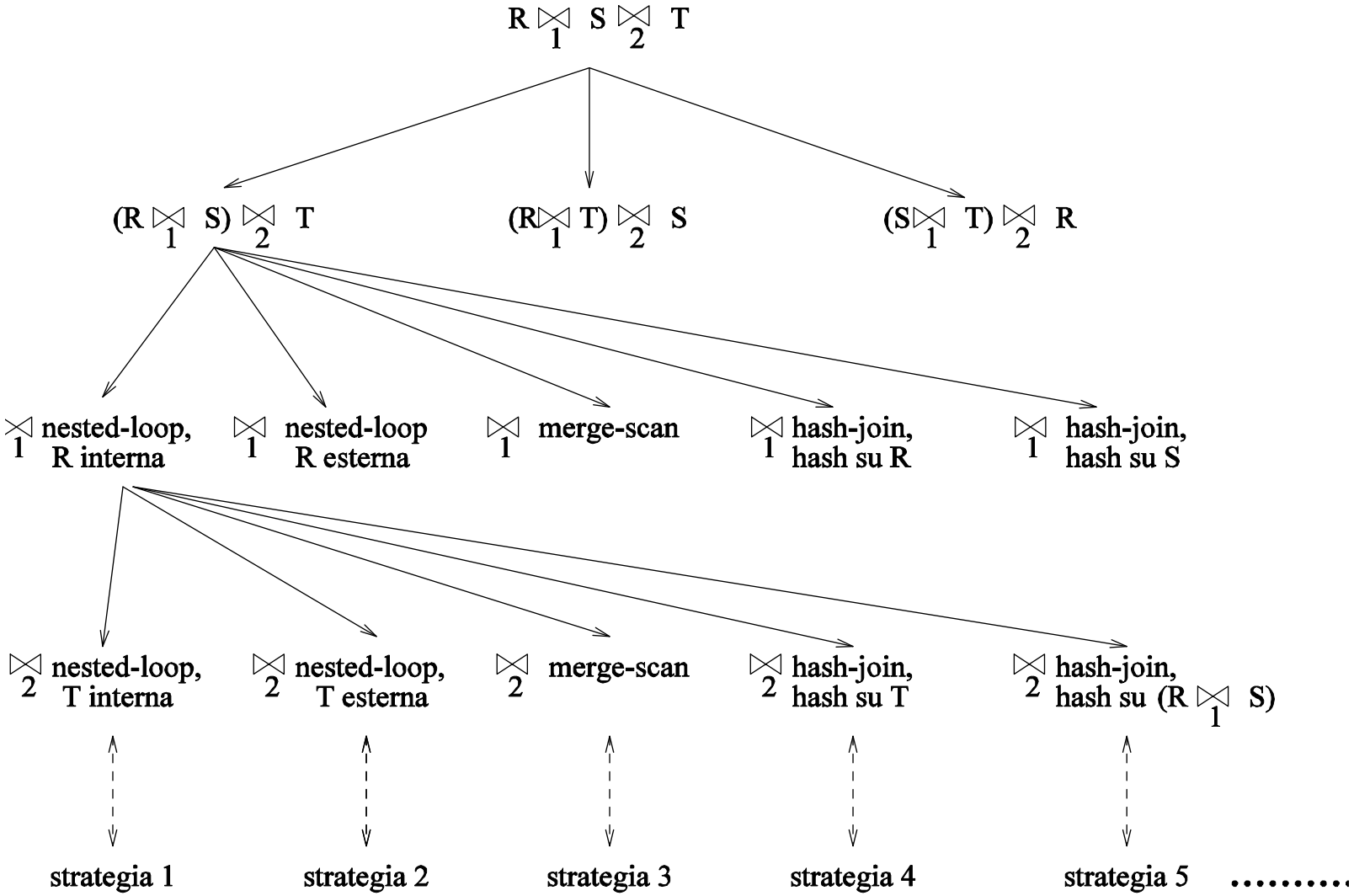
- Un problema articolato, con scelte relative a:
  - operazioni da eseguire (es.: scansione o accesso diretto?)
  - ordine delle operazioni (es. join di tre relazioni; ordine?)
  - i dettagli del metodo (es.: quale metodo di join)

# 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)`