INTRODUZIONE AL LIVELLO FISICO: FILE, PAGINE, RECORD E INDICI

Roberto Basili

Corso di Basi di Dati

a.a. 2021/22

Dati su dispositivi di memorizzazione esterni

- **Dischi**: si può leggere qualunque pagina a costo fisso
- **Nastri**: si possono leggere le pagine solo in sequenza
- **Organizzazione del file**: metodo per registrare un file di record su un dispositivo di memorizzazione esterno
- **Architettura**: il gestore di buffer carica le pagine dal dispositivo di memorizzazione esterno al buffer nella memoria principale. Lo strato di gestione dei file e quello degli indici eseguono le chiamate al gestore di buffer

Disks and Files

- DBMS stores information on ("hard") disks.
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Why Not Store Everything in Main Memory?

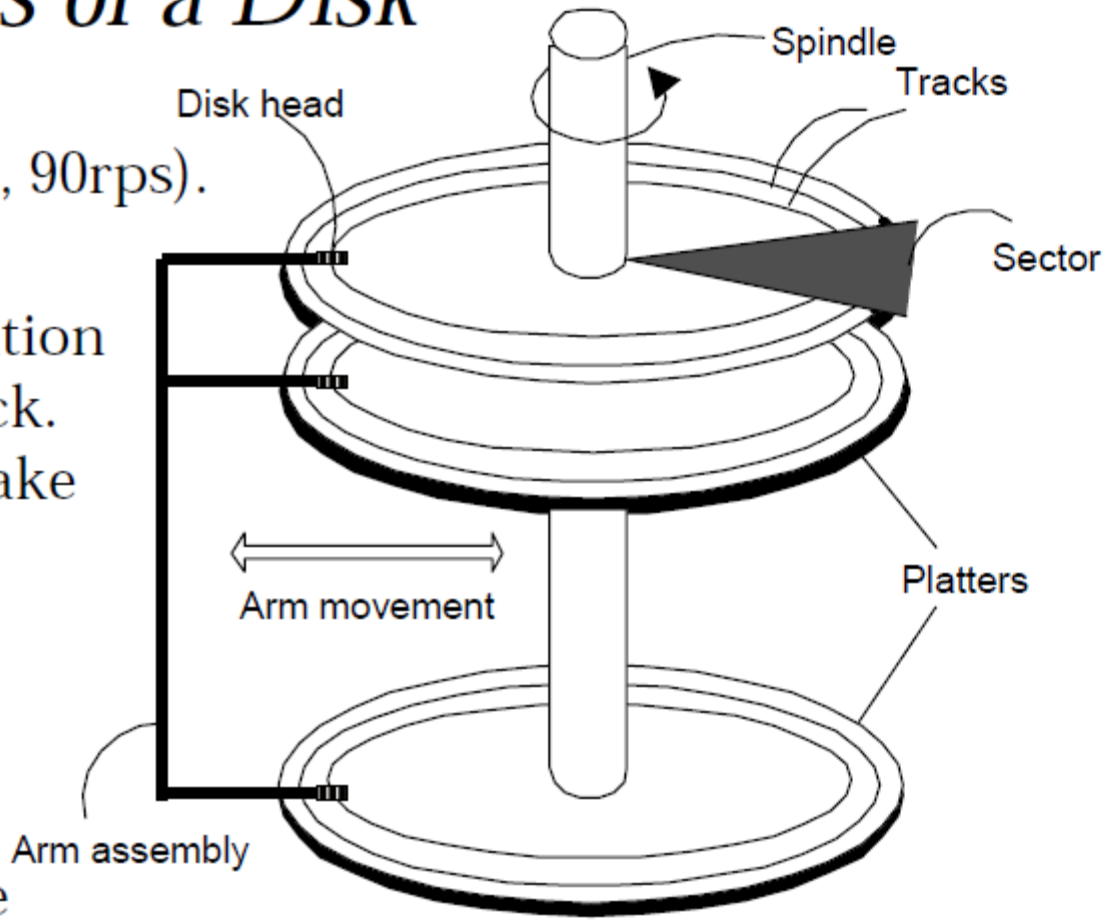
- *Costs too much.* $1000 will buy you either 128MB of RAM or 7.5GB of disk today.
- *Main memory is volatile.* We want data to be saved between runs. (Obviously!)
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

Disks

- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - Therefore, relative placement of pages on disk has major impact on DBMS performance!

Components of a Disk

- The platters spin (say, 90rps).
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
- Only one head reads/writes at any one time.
- *Block size* is a multiple of *sector size* (which is fixed).



Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

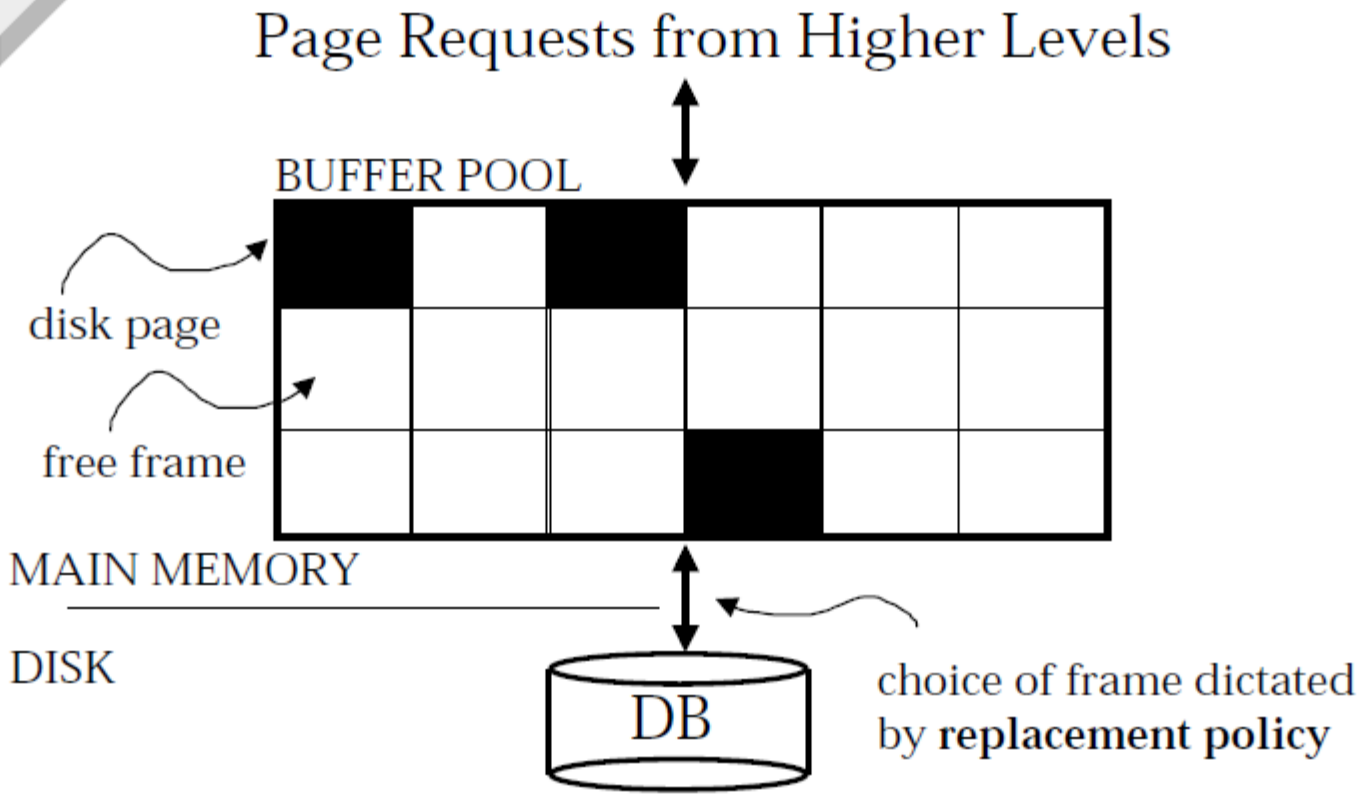
Arranging Pages on Disk

- '*Next*' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Blocks in a file should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Disk Space Management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don't need to know how this is done, or how free space is managed.

Buffer Management in a DBMS



- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*

When a Page is Requested ...

- If requested page is not in pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address.

☛ *If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!*

More on Buffer Management

- Requestor of page must unpin it, and indicate whether page has been modified:
  - *dirty* bit is used for this.
- Page in pool may be requested many times,
  - a *pin count* is used. A page is a candidate for replacement iff *pin count* = 0.
- CC & recovery may entail additional I/O when a frame is chosen for replacement. (*Write-Ahead Log* protocol; more later.)

Buffer Replacement Policy

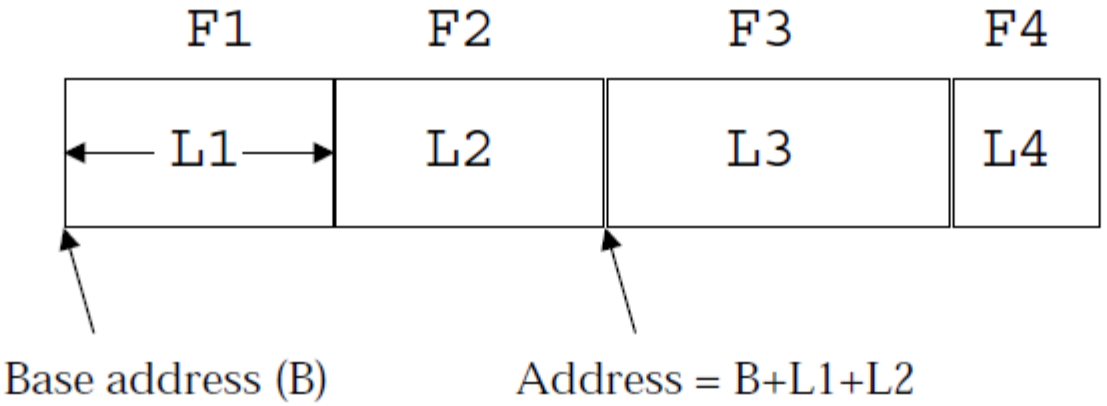
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O's; depends on the *access pattern.*
- *Sequential flooding*: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

DBMS vs. OS File System

OS does disk space & buffer mgmt: why not let OS manage these tasks?

- Differences in OS support: portability issues
- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
  - adjust *replacement policy*, and pre-fetch pages based on access patterns in typical DB operations.

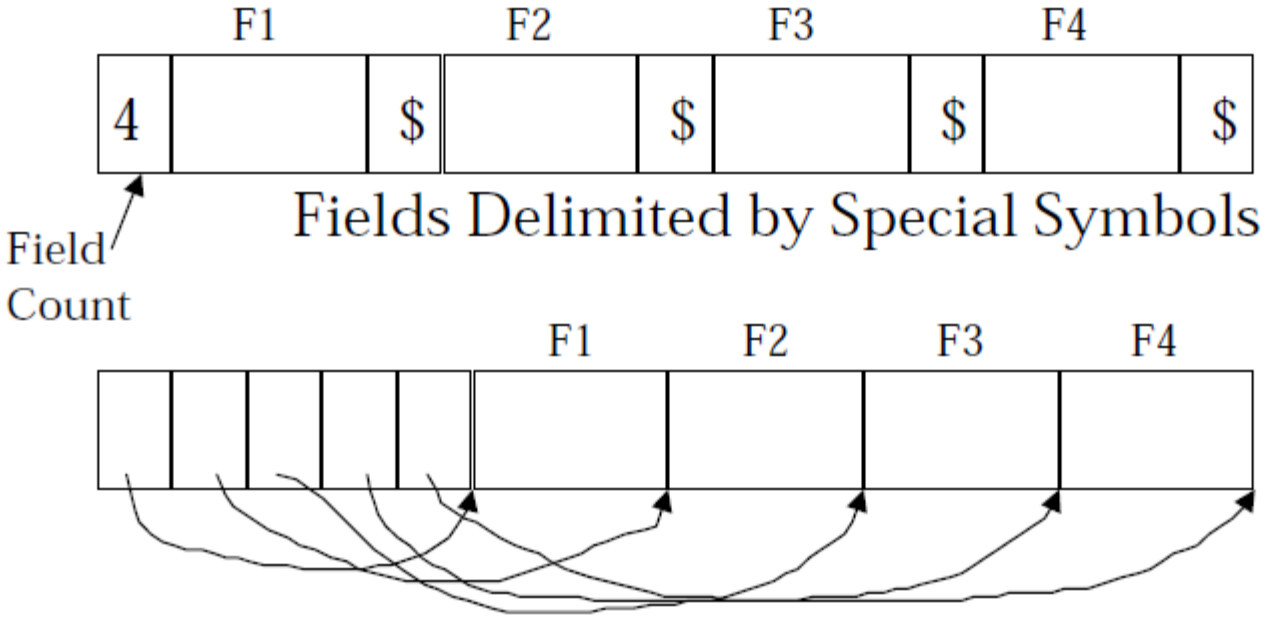
Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.

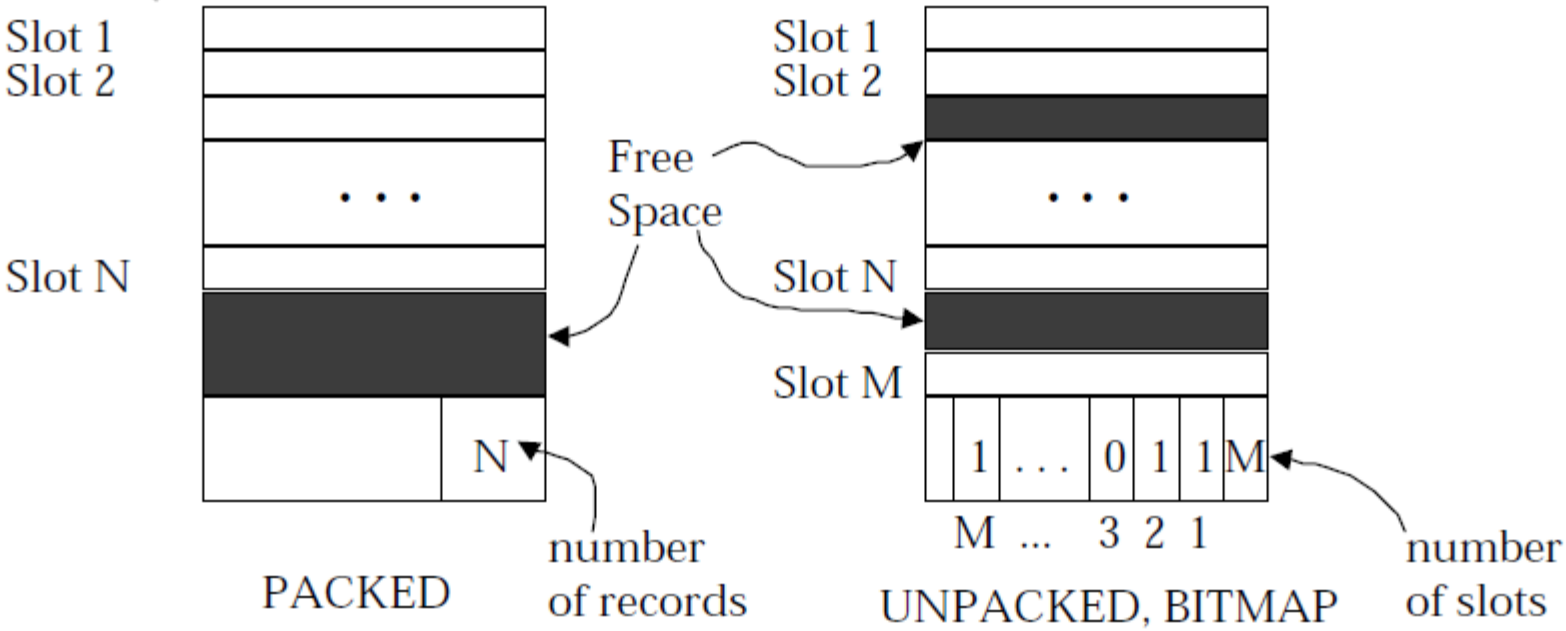
Record Formats: Variable Length

- Two alternative formats (# fields is fixed):



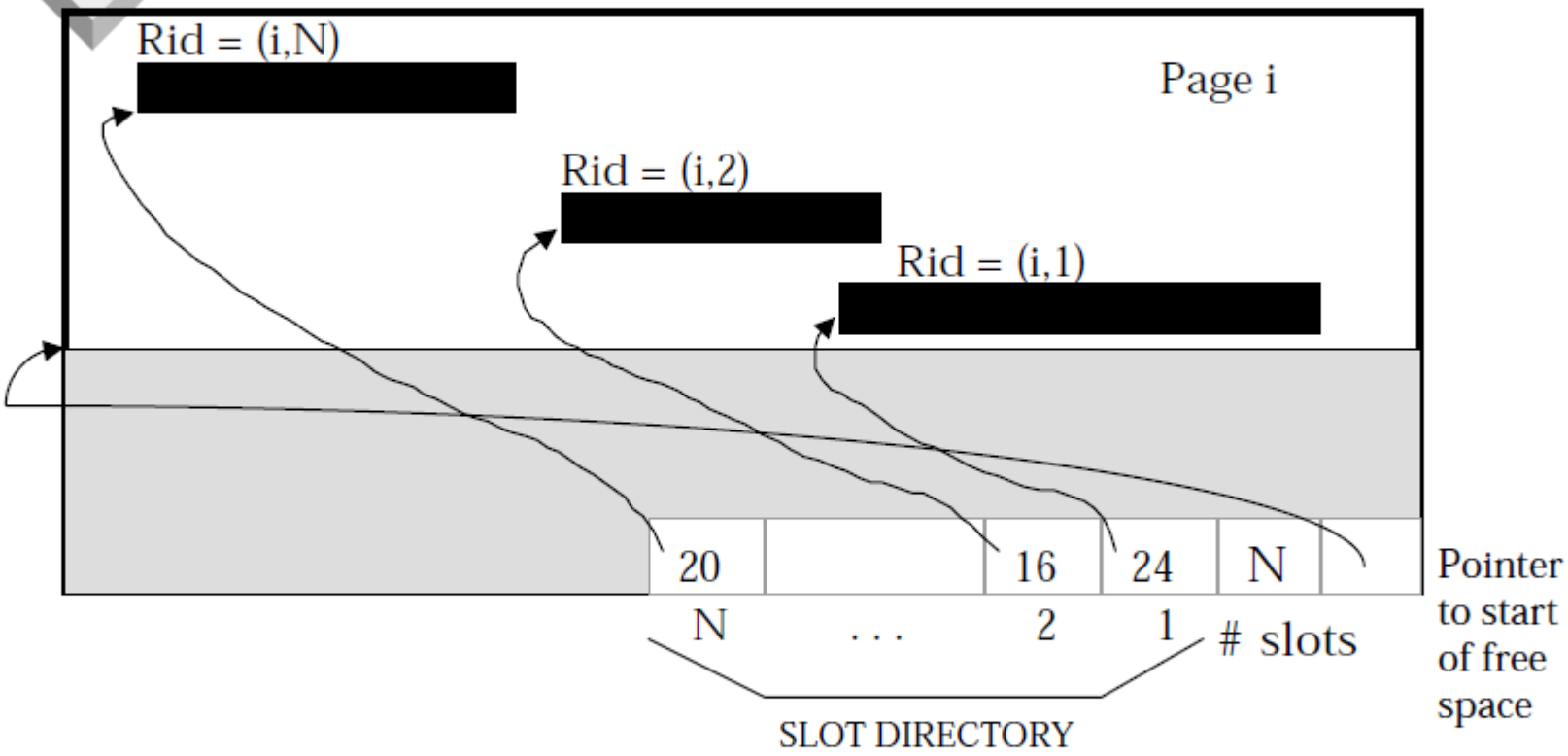
☛ Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

Page Formats: Fixed Length Records

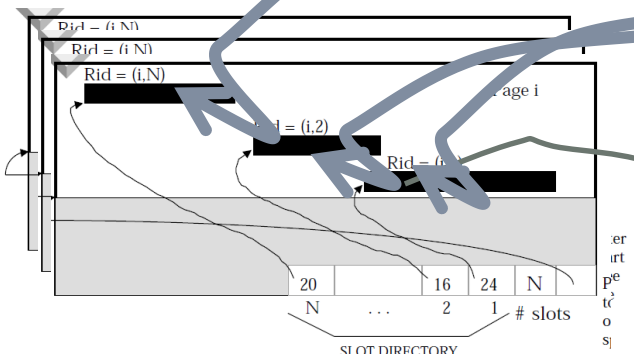


- *Record id* = <*page id, slot #*>. *In first alternative, moving records for free space management changes rid; may not be acceptable.*

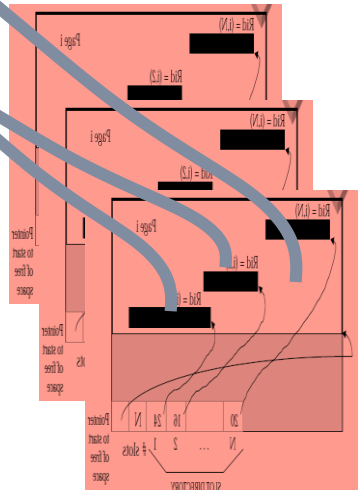
Page Formats: Variable Length Records



- *Can move records on page without changing rid; so, attractive for fixed-length records too.*



- FILE DATI



FILE INDICI

Studente

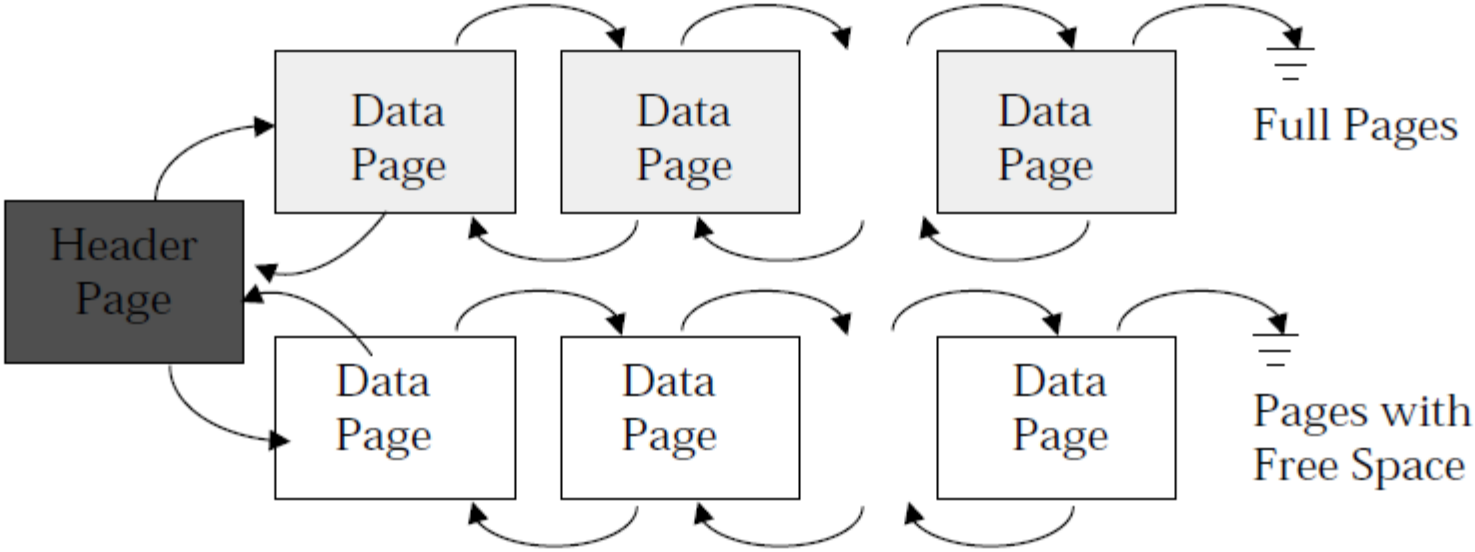
Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

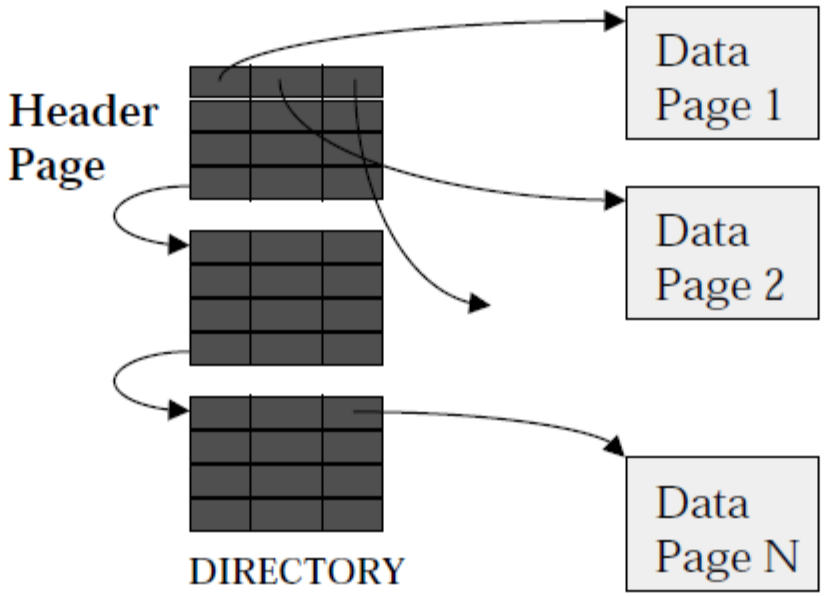
- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this.

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace.
- Each page contains 2 `pointers' plus data.

Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - *Much smaller than linked list of all HF pages*!

Indexes

- A Heap file allows us to retrieve records:
  - by specifying the *rid*, or
  - by scanning all records sequentially
- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the "CS" department
  - Find all students with a gpa > 3
- Indexes are file structures that enable us to answer such value-based queries efficiently.

System Catalogs

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

☛ *Catalogs are themselves stored as relations*!

Attr_Cat(attr_name, rel_name, type, position)

attr_name	rel_name	type	position
attr_name	Attribute_Cat	string	1
rel_name	Attribute_Cat	string	2
type	Attribute_Cat	string	3
position	Attribute_Cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

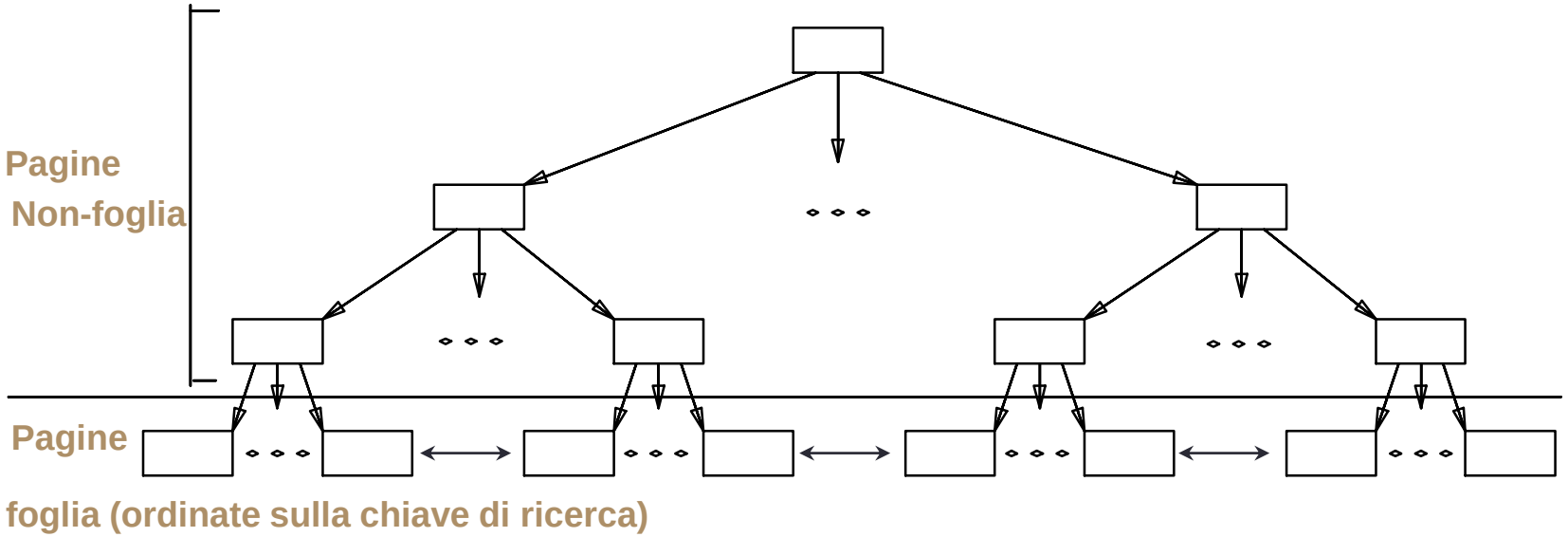
Organizzazioni alternative dei file

- Esistono molte alternative, ciascuna ideale per qualche situazione e non molto buona in altre :
  - **file heap** (ordine casuale): va bene quando l'accesso tipico è una scansione di file per leggere tutti i record
  - **file ordinato**: ottimale quando i record devono essere letti in qualche ordine, o quando si ha bisogno solo di un "intervallo" di record
  - **indici**: strutture di dati per organizzare i record in **alberi** o tramite **hashing**
    - come i file ordinati, essi velocizzano le ricerche di sottoinsiemi di record, basate su valori in certi campi ("chiavi di ricerca")
    - gli aggiornamenti sono molto più veloci che nei file ordinati

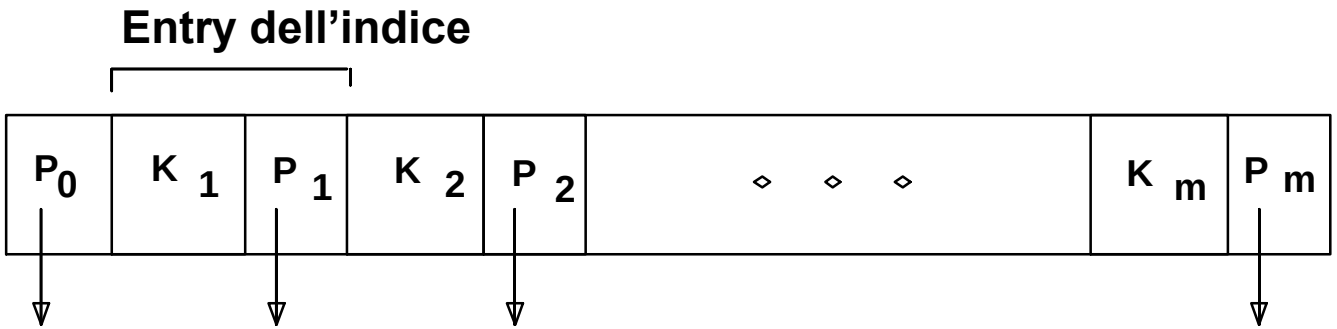
Indici

- Un indice su un file velocizza le selezioni sui campi che compongono la **chiave di ricerca per l'indice**
  - Qualunque sottoinsieme dei campi di una relazione può essere la chiave di ricerca per un indice sulla relazione
  - **La chiave di ricerca non è la chiave** (insieme minimo di campi che identificano univocamente un record in una relazione)
- Un indice contiene una collezione di *data entry*, e supporta il reperimento efficiente di tutte le data entry *k** con un dato valore *k* della chiave
  - Avendo la data entry *k** possiamo trovare i record con chiave *k* in al più un I/O di disco (fra poco i dettagli...)

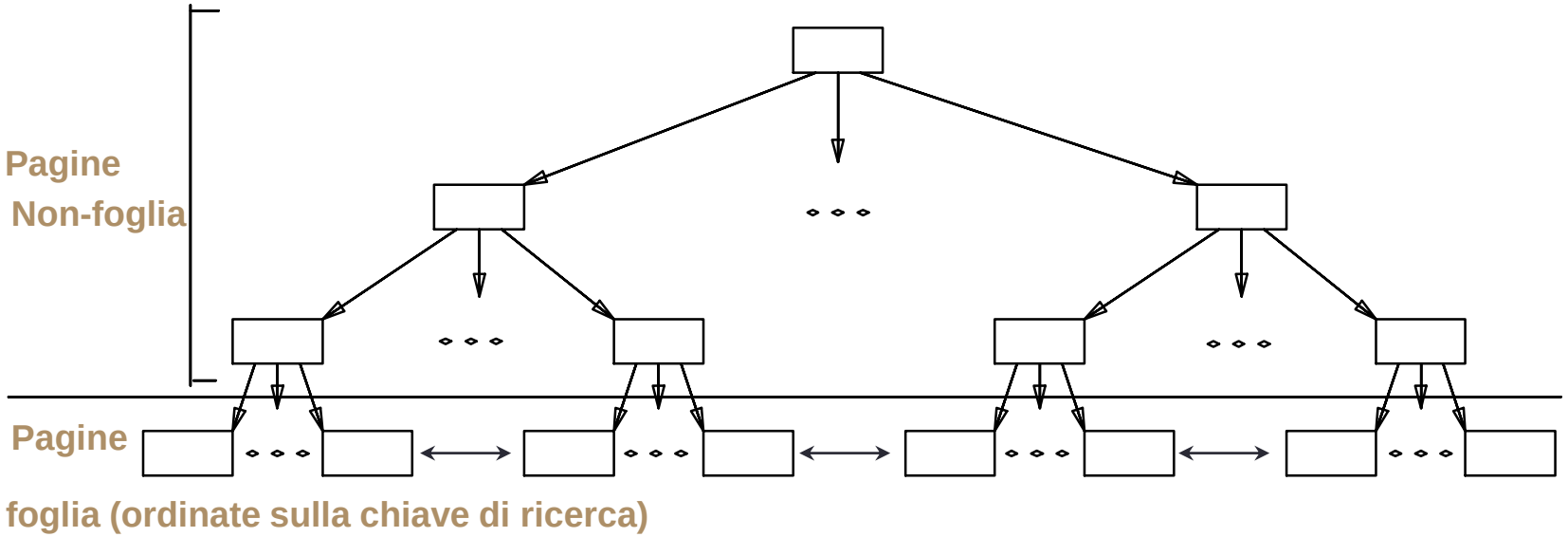
Indici ad albero B+



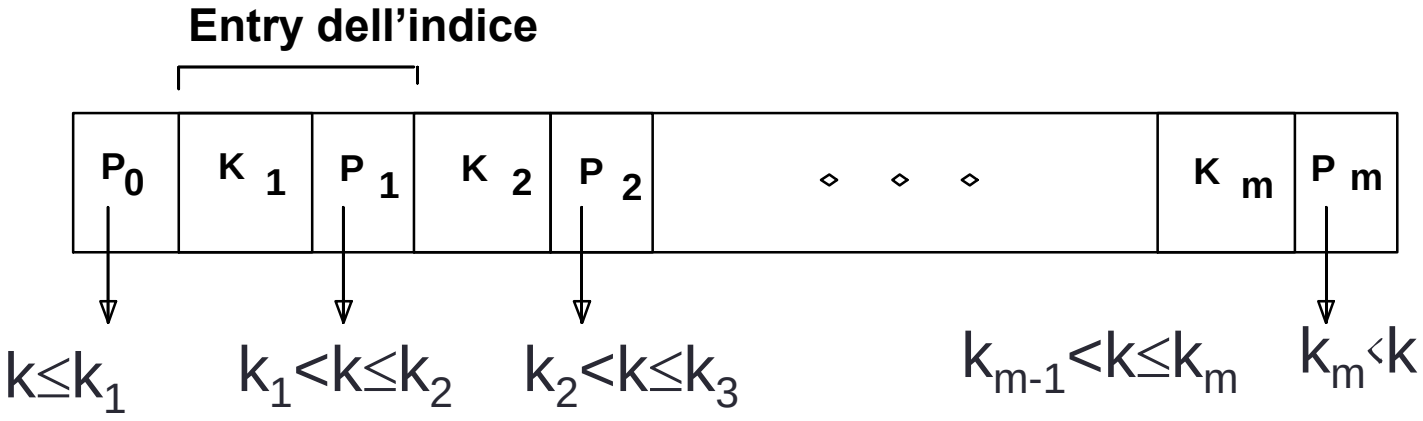
- Le pagine foglia *contengono data record*, e sono collegate (prec & succ)
- Le altre pagine contengono *data entry*, usate solo per guidare le ricerche



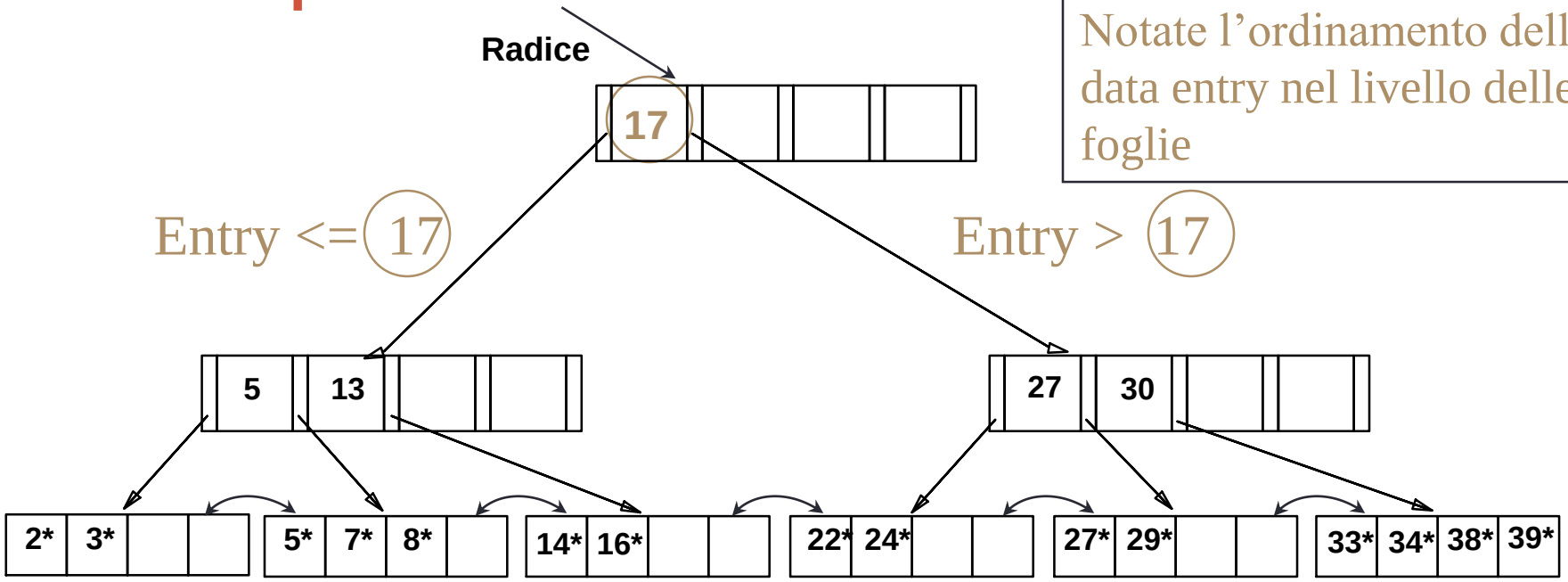
Indici ad albero B+



- Le pagine foglia contengono *data record*, e sono collegate (prec & succ)
- Le altre pagine contengono *data entry*, usate solo per guidare le ricerche



Esempio di albero B+



Notate l'ordinamento delle data entry nel livello delle foglie

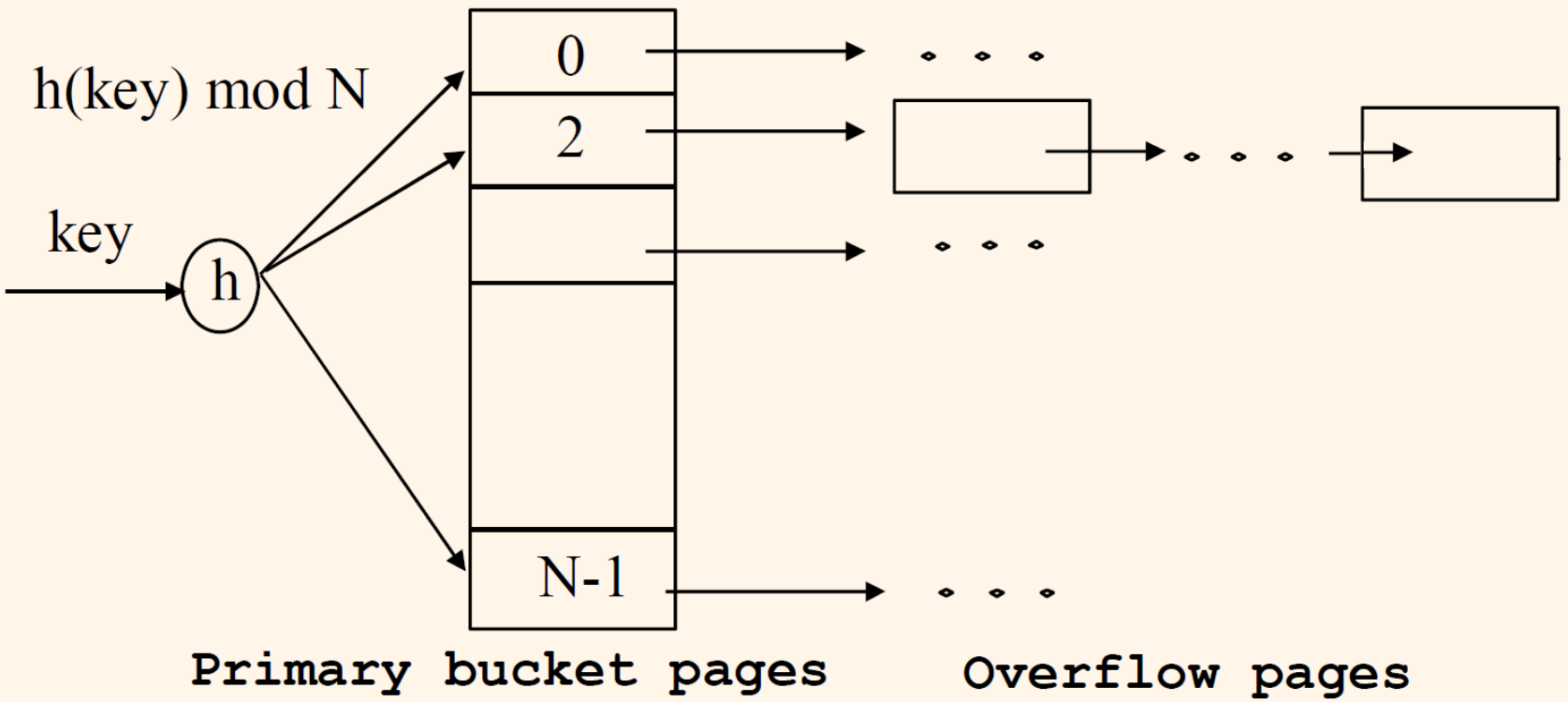
- Trovare 28*? 29*? Tutte >15* e <30*
- Inserimento/cancellazione: trovare la data entry nella foglia, poi cambiarla. A volte bisogna aggiustare i nodi genitori
  - E qualche volta la modifica si propaga a tutto l'albero

Indici basati su *hash*

- Buoni per selezioni basate su **uguaglianze**
- L'indice è una collezione di **bucket**
  - **Bucket** = pagina primaria più zero o più pagine addizionali
  - I **bucket** contengono le **data entry**
- Funzione hash

  h: h(r) = bucket cui appartiene (la data entry per) il record r.
- h si applica ai campi della chiave di ricerca di r
  - In questo schema non c'è bisogno di index entry

Indici Hash: struttura delle pagine



da Database Management Systems, R. Ramakrishnan and J. Gehrke

Alternative per le data entry k* nell'indice

- In una data entry k* possiamo memorizzare:
  1. record di dati con valore di chiave K, oppure
  2. <*k, rid* dei record di dati con valore della chiave di ricerca k>, oppure
  3. <*k, lista dei rid* dei record di dati con chiave di ricerca k>

- La scelta dell'alternativa per le data entry è ortogonale alla tecnica di indicizzazione usata per localizzare le data entry con un dato valore di chiave K
  - esempi di tecnica di indicizzazione: alberi B+, strutture basate su hash
  - tipicamente, l'indice contiene informazioni aggiuntive che guidano le ricerche verso le data entry desiderate

Alternative per le data entry (segue)

- Alternativa 1:
  - se si usa questa alternativa, la struttura a indice è una organizzazione di file per i record di dati (piuttosto che un file heap o un file ordinato)
  - al più un indice su una data collezione di record di dati può usare l'Alternativa 1 (altrimenti, i record di dati sono duplicati, il che porta a memorizzazione ridondante e potenziale inconsistenza)
  - se i record di dati sono molto grandi, il numero di pagine contenente le data entry è alto. Tipicamente, ciò implica che la dimensione delle informazioni aggiuntive nell'indice è anch'essa molto grande

Alternative per le data entry (segue)

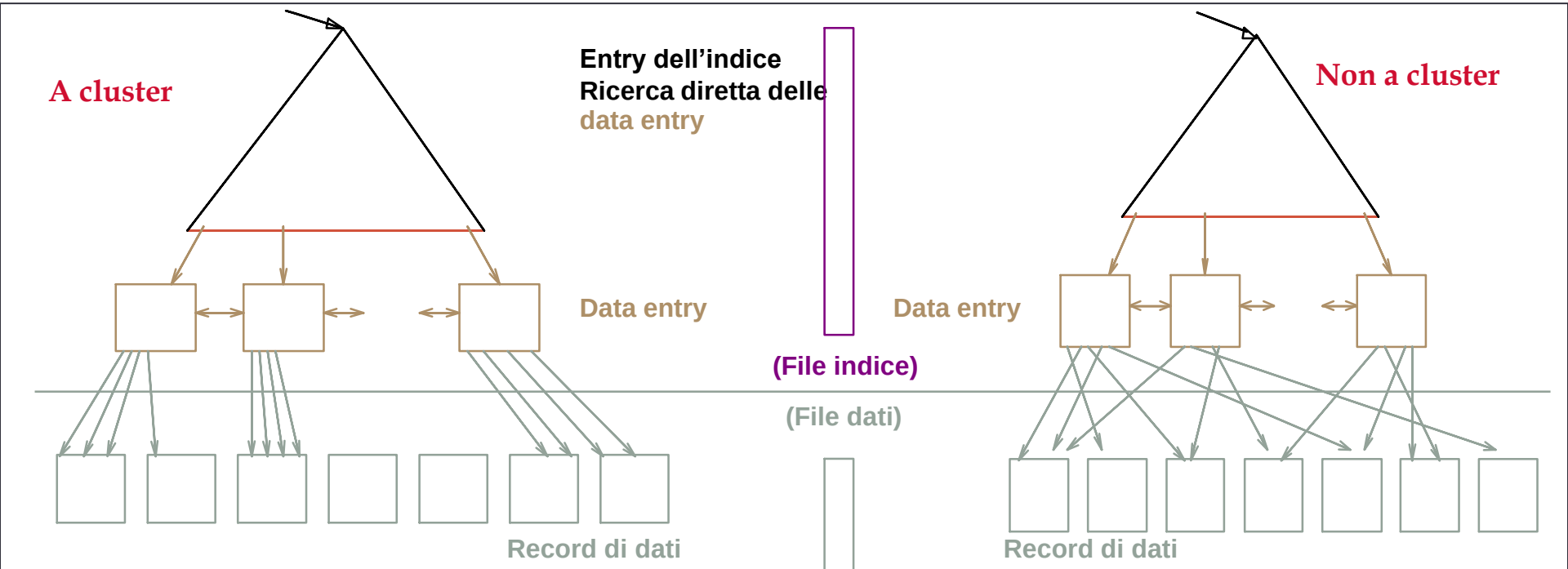
- Alternative 2 e 3:
  - Le data entry tipicamente sono molto più piccole dei record di dati. Quindi, meglio dell'Alternativa 1 con record di dati grandi, specialmente se le chiavi di ricerca sono piccole (la porzione della struttura ad indice usata per dirigere la ricerca, che dipende dalla dimensione delle data entry, è molto più piccola rispetto all'Alternativa 1)
  - L'alternativa 3 è più compatta dell'Alternativa 2, ma porta a data entry di dimensioni variabili; anche se le chiavi di ricerca sono di lunghezza fissa

Classificazione degli indici

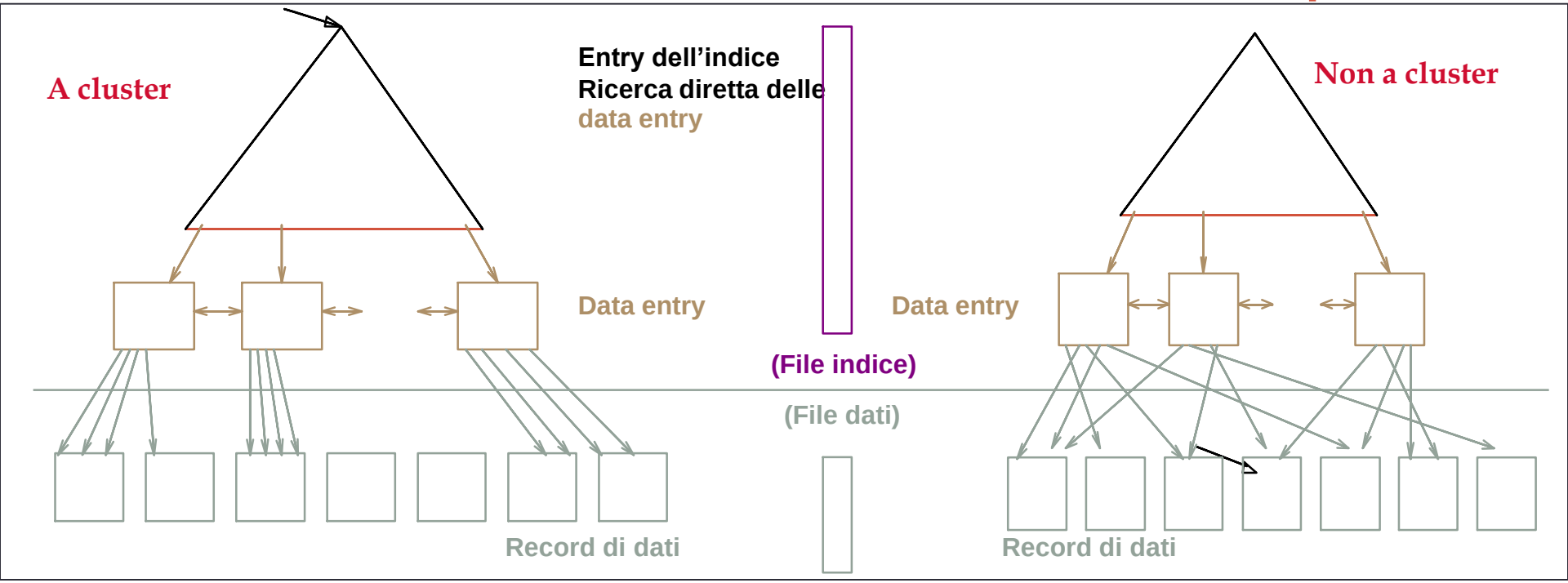
- **Primario** o **secondario**: se la chiave di ricerca contiene la chiave primaria, allora è chiamato indice primario
  - Indice univoco: la chiave di ricerca contiene una chiave candidata
- **Clustered** e **non clustered**: se l'ordine dei record di dati è lo stesso, o "quasi", delle data entry, allora è chiamato indice clustered
  - L'Alternativa 1 implica un indice *clustered*; nella pratica, anche gli indici clustered implicano l'Alternativa 1 (poiché i file ordinati sono rari)
  - Un file può essere *clustered* su al più una chiave di ricerca
  - Il costo della lettura dei record di dati usando l'indice varia fortemente in base al fatto che l'indice sia clustered oppure no!

Indici clustered e non clustered

- Supponiamo che venga usata l'Alternativa 2 per le data entry, e che i record di dati siano memorizzati in un file heap
  - Per costruire un indice clustered, prima ordiniamo il file heap (con dello spazio libero su ciascuna pagina per gli inserimenti futuri)
  - Potrebbero essere necessarie delle pagine aggiuntive per gli inserimenti (quindi, l'ordine dei record di dati è "quasi" uguale, ma non identico, a quello delle data entry)



Indici clustered e non clustered esempio

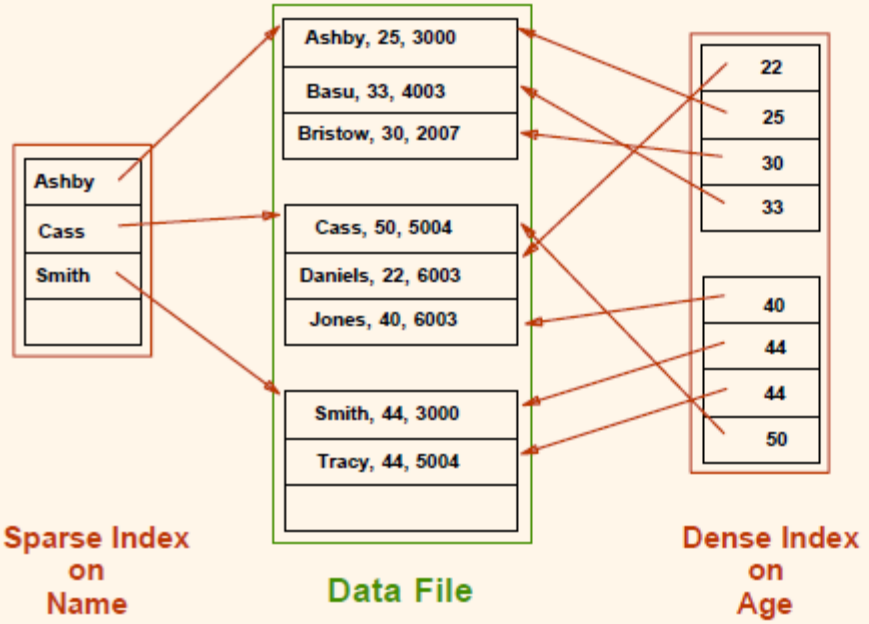


- Data Record: relazione Studenti - Stesso file (i.e. Record di) dati a dx e sx
- Albero sx: clustered su chiave di ricerca *Cognome*
- Albero dx su chiave di ricerca: Età

⇒ L'Indice diviene immediatamente *unclustered*

Index Classification (Contd.)

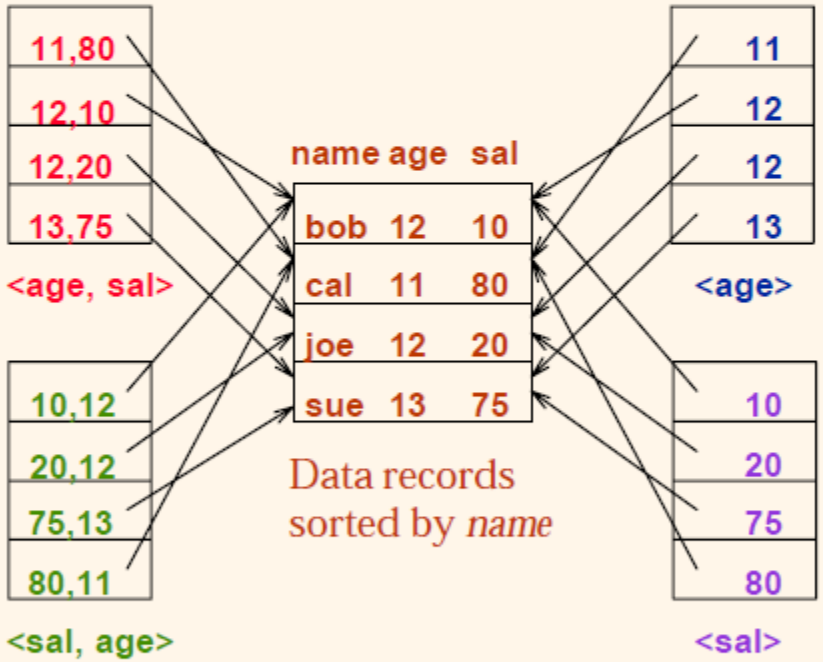
- *Dense* vs. *Sparse*: If there is at least one data entry per search key value (in some data record), then dense.
  - Alternative 1 always leads to dense index.
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



Index Classification (Contd.)

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.
  - Lexicographic order, or
  - Spatial order.

Examples of composite key indexes using lexicographic order.



Modello di costo per la nostra analisi

- Tralasciamo i costi della CPU, per semplicità:
- B: il numero di pagine di dati
- R: il numero di record per pagina
- D: tempo (medio) per leggere o scrivere una pagina su disco
- La misura del numero di I/O di pagina tralascia il guadagno dovuto al precaricamento di una sequenza di pagine; quindi, anche il costo di I/O è solo approssimato
- Analisi del caso medio; basata su diverse assunzioni semplificatrici

➢ *Abbastanza buona per mostrare la tendenza generale !*

Confronto tra le organizzazioni di file

- **File heap** (ordine casuale; inserimento alla fine del file)
- **File ordinati** su <età, sal>
- **File clustered ad albero B+,** Alternativa (1), chiave di ricerca <età, sal>
- **File heap con indice hash non clustered** su chiave di ricerca <età, sal>

Operazioni da confrontare

- **Scansione**: lettura di tutti i record dal disco
- **Ricerca con selezione di uguaglianza**
- **Ricerca con selezione di intervallo**
- **Inserimento** di un record
- **Cancellazione** di un record

Ipotesi nella nostra analisi

- File heap:
  - selezione di uguaglianza sulla chiave; un solo risultato
- File ordinati:
  - file compattati dopo le cancellazioni
- Indici:
  - Alternative (2), (3): dimensione delle data entry = 10% della dimensione dei record
  - Hash: niente bucket aggiuntivi
    - 80% di occupazione delle pagine => dimensione del file = 1.25 volte la dimensione dei dati
  - Albero: 67% di occupazione (valore tipico)
    - Implica una dimensione di file = 1.5 volte la dimensione dei dati

Ipotesi (segue)

- **Scansioni**:
  - i livelli foglia di un indice ad albero sono collegati
  - le data entry dell'indice più il file vero e proprio vengono scansionati per indici non clustered
- **Ricerche con selezione di intervallo**:
  - usiamo gli indici ad albero per restringere l'insieme di record di dati restituito, ma tralasciamo gli indici hash

Costo delle operazioni

	(a) Scansione	(b) Uguaglianza	(c) Intervallo	(d) Inserimento	(e) Cancellazione
(1) Heap					
(2) Ordinato					
(3) Clustered					
(4) Indice ad albero non *clustered*					
(5) Indice *hash* non *clustered*					

- *Diverse ipotesi alla base di queste (grossolane) stime!*

Costo delle operazioni

	(a) Scansione	(b) Uguaglianza	(c) Intervallo	(d) Inserimento	(e) Cancellazione
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Ricerca +D**

- *Diverse ipotesi alla base di queste (grossolane) stime!*

Costo delle operazioni

	(a) Scansione	(b) Uguaglianza	(c) Intervallo	(d) Inserimento	(e) Cancellazione
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Ricerca +D**
(2) Ordinato	**BD**	**Dlog 2B**	**D(log 2 B + Numero di pagine con *record* compatibili)**	**Ricerca + BD**	**Ricerca +BD**

- *Diverse ipotesi alla base di queste (grossolane) stime!*

Costo delle operazioni

	(a) Scansione	(b) Uguaglianza	(c) Intervallo	(d) Inserimento	(e) Cancellazione
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Ricerca +D**
(2) Ordinato	**BD**	**Dlog 2B**	**D(log 2 B + Numero di pagine con *record* compatibili)**	**Ricerca + BD**	**Ricerca +BD**
(3) *Clustered*	**1.5BD**	**Dlog $_F$ 1.5B**	**D(log $_F$ 1.5B + Numero di pagine con *record* compatibili)**	**Ricerca + D**	**Ricerca +D**

- *Diverse ipotesi alla base di queste (grossolane) stime!*

Costo delle operazioni

	(a) Scansione	(b) Uguaglianza	(c) Intervallo	(d) Inserimento	(e) Cancellazione
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Ricerca +D**
(2) Ordinato	**BD**	**Dlog 2B**	**D(log 2 B + Numero di pagine con *record* compatibili)**	**Ricerca + BD**	**Ricerca +BD**
(3) *Clustered*	**1.5BD**	**Dlog $_F$ 1.5B**	**D(log $_F$ 1.5B + Numero di pagine con *record* compatibili)**	**Ricerca + D**	**Ricerca +D**
(4) Indice ad albero non *clustered*	**BD(R+0.15)**	**D(1 + log $_F$ 0.15B)**	**D(log $_F$ 0.15B + Numero di pagine con *record* compatibili)**	**Ricerca + 2D**	**Ricerca + 2D**

- *Diverse ipotesi alla base di queste (grossolane) stime!*

Costo delle operazioni

	(a) Scansione	(b) Uguaglianza	(c) Intervallo	(d) Inserimento	(e) Cancellazione
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Ricerca +D**
(2) Ordinato	**BD**	**Dlog 2B**	**D(log 2 B + Numero di pagine con *record* compatibili)**	**Ricerca + BD**	**Ricerca +BD**
(3) *Clustered*	**1.5BD**	**Dlog $_F$ 1.5B**	**D(log $_F$ 1.5B + Numero di pagine con *record* compatibili)**	**Ricerca + D**	**Ricerca +D**
(4) Indice ad albero non *clustered*	**BD(R+0.15)**	**D(1 + log $_F$ 0.15B)**	**D(log $_F$ 0.15B + Numero di pagine con *record* compatibili)**	**Ricerca + 2D**	**Ricerca + 2D**
(5) Indice *hash* non *clustered*	**BD(R+0.125)**	**2D**	**BD**	**Ricerca + 2D**	**Ricerca + 2D**

- *Diverse ipotesi alla base di queste (grossolane) stime!*

Costo delle operazioni

	(a) Scansione	(b) Uguaglianza	(c) Intervallo	(d) Inserimento	(e) Cancellazione
(1) Heap	**BD**	**0.5BD**	**BD**	**2D**	**Ricerca +D**
(2) Ordinato	**BD**	**Dlog 2B**	**D(log 2 B + Numero di pagine con *record* compatibili)**	**Ricerca + BD**	**Ricerca +BD**
(3) *Clustered*	**1.5BD**	**Dlog F 1.5B**	**D(log F 1.5B + Numero di pagine con *record* compatibili)**	**Ricerca + D**	**Ricerca +D**
(4) Indice ad albero non *clustered*	**BD(R+0.15)**	**D(1 + log F 0.15B)**	**D(log F 0.15B + Numero di pagine con *record* compatibili)**	**Ricerca + 2D**	**Ricerca + 2D**
(5) Indice *hash* non *clustered*	**BD(R+0.125)**	**2D**	**BD**	**Ricerca + 2D**	**Ricerca + 2D**

- *Diverse ipotesi alla base di queste (grossolane) stime!*

Comprendere il carico di lavoro

- Per ciascuna interrogazione nel carico di lavoro:
  - A quali relazioni accede?
  - Quali attributi vengono restituiti?
  - Quali attributi sono coinvolti nelle condizioni di selezione/join? Quanto selettive sono, probabilmente, tali condizioni?
- Per ciascun aggiornamento nel carico di lavoro:
  - Quali attributi sono coinvolti nelle condizioni di selezione/join? Quanto selettive sono, probabilmente, tali condizioni?
  - Il tipo di aggiornamento (INSERT, DELETE, UPDATE) e gli attributi che vengono modificati

Scelta degli indici

- Quali indici dovremmo creare?
  - Quali relazioni dovrebbero avere indici?
  - Quali campi dovrebbero costituire la chiave di ricerca?
  - Dovremmo creare più di un indice?
- Per ciascun indice, che tipo di indice dovremmo usare?
  - Clustered?
  - Hash/albero?

Scelta degli indici (segue)

- Un approccio
  - Consideriamo a turno le interrogazioni più importanti.
  - Consideriamo il miglior piano usando gli indici correnti, e vediamo se è possibile un piano migliore con un indice aggiuntivo. Se sì, creiamolo.

- Ovviamente, ciò implica la comprensione di come un DBMS valuta le interrogazioni e crea i piani di valutazione delle interrogazioni!
  - Per ora discutiamo semplici interrogazioni di una tabella

Scelta degli indici (segue)

- Prima di creare un indice, dobbiamo anche considerare l'impatto sugli aggiornamenti nel carico di lavoro!
  - Compromesso: gli indici possono velocizzare le interrogazioni, rallentare gli aggiornamenti. Anche richiedere spazio su disco

Linee guida per la selezione degli indici

- Gli attributi nella clausola WHERE sono candidati come chiavi dell'indice
- Chiavi di ricerca multi-attributo dovrebbero essere prese in considerazione quando una clausola WHERE contiene diverse condizioni
- Provare a scegliere gli indici che supportano quante più interrogazioni possibile. Poiché solo un indice per ogni relazione può essere *clustered*, sceglierlo in base alle interrogazioni importanti che più traggono beneficio dal *clustering*

Esempi di indici clustered

- Un indice ad albero B+ su I.età può essere usato per selezionare le tuple
  - Quanto è selettiva la condizione?
  - L'indice è clustered?
- Consideriamo l'interrogazione con GROUP BY
  - Se molte tuple hanno I.età > 10, usare l'indice su I.età e ordinare le tuple restituite può essere costoso
  - L'indice clustered I.rnum può essere migliore!
- Interrogazioni con selezione di uguaglianza e duplicati
  - Il clustering su I.hobby aiuta!

```
SELECT I.rnum
FROM Imp I
WHERE I.età  > 40
```

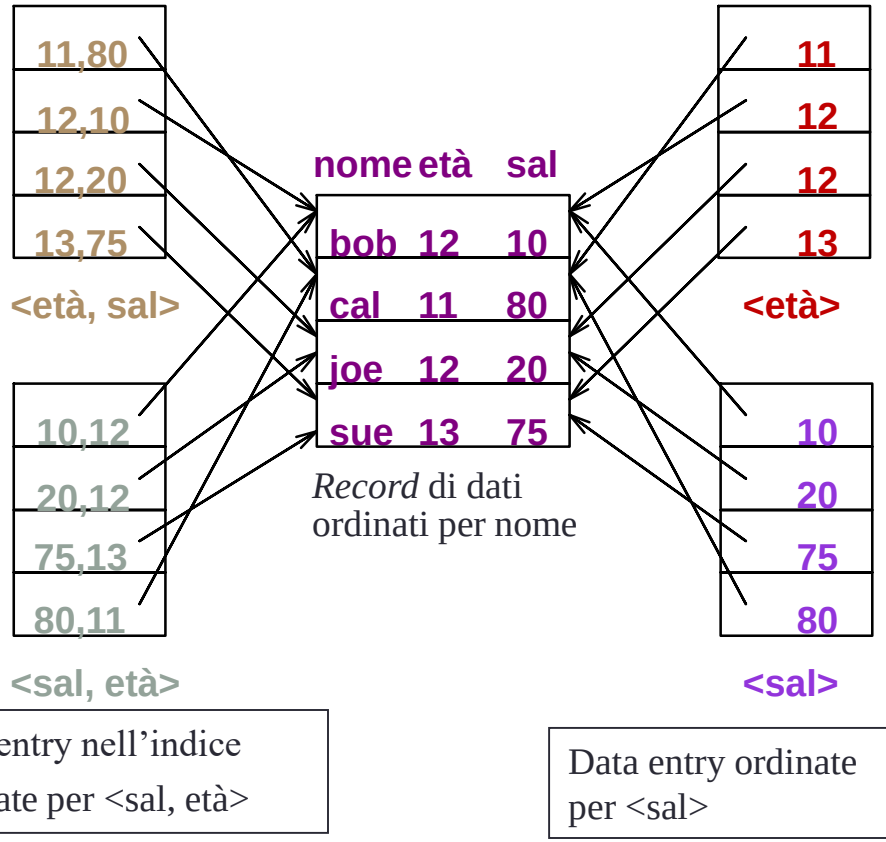
```
SELECT I.rnum, COUNT(*)
FROM Imp I
WHERE I.età > 10
GROUP BY I.rnum
```

```
SELECT I.rnum
FROM Imp I
WHERE
I.hobby='Francobolli'
```

Indici con chiavi di ricerca composite

- Chiavi di ricerca composite: ricerca su una combinazione di campi
  - Interrogazioni con selezione di uguaglianza: il valore di ogni campo è uguale a un valore costante. Ad esempio, con riferimento all'indice <sal, età>:
    - età = 20 and sal = 75
  - Interrogazioni con selezione di intervallo: il valore di alcuni campi non è una costante. Ad esempio:
    - età = 20 and sal > 10
- le data entry nell'indice sono ordinate per chiave di ricerca per supportare le interrogazioni con selezione di intervallo
  - ordine lessicografico, oppure
  - ordine spaziale

Esempi di indice con chiave composita che usa l'ordine lessicografico



Chiavi di ricerca composite

- Per trovare i record di Imp con età = 30 AND sal = 4000, un indice su <età, sal> sarebbe meglio di un indice su età o di un indice su sal
  - La scelta della chiave dell'indice è ortogonale al clustering etc.
- Se la condizione è 20 < età < 30 AND 3000 < sal < 5000
  - È meglio un indice clustered ad albero su <età, sal> o su <sal, età>
- Se la condizione è età = 30 AND 3000 < sal < 5000
  - Un indice clustered <età, sal> è molto meglio di un indice su <sal, età>
- Gli indici compositi sono più grandi, e aggiornati più spesso

Piani basati solo sull'indice

- Se è disponibile un indice opportuno, a molte interrogazioni si può rispondere senza leggere alcuna tupla da alcuna delle relazioni coinvolte

<I.rnum>

```
SELECT I.rnum, COUNT(*)
FROM Imp I
GROUP BY I.rnum
```

<I.rnum, I.sal>
Indice ad albero!

```
SELECT I.rnum, MIN(I.sal)
FROM Imp I
GROUP BY I.rnum
```

<I.età, I.sal>
oppure
<I.sal, I.età>
Indice ad albero!

```
SELECT AVG(I.sal)
FROM Imp I
WHERE I.età = 25 AND
E.sal BETWEEN 3000 AND 5000
```

Piani basati solo sull'indice (segue)

- Piani basati solo sull'indice sono possibili se la chiave è <rnum, età> o se abbiamo un indice ad albero con chiave <età, rnum>

```
SELECT I.rnum, COUNT(*)
FROM Imp I
WHERE I.età = 30
GROUP BY I.rnum
```

  - Quale è migliore?
  - Che succede se consideriamo la seconda interrogazione?

```
SELECT I.rnum, COUNT(*)
FROM Imp I
WHERE I.età > 30
GROUP BY I.rnum
```

Piani basati solo sull'indice (segue)

- Si possono trovare piani basati solo sull'indice anche per interrogazione che coinvolgono più di una tabella: ne parleremo più avanti

<I.rnum>

```
SELECT R.mgr
FROM Rep R, Imp I
WHERE R.rnum = I.rnum
```

<I.rnum, I.iid>

```
SELECT R.mgr, I.iid
FROM Rep R, Imp I
WHERE R.rnum = I.rnum
```

Sommario

- Esistono molte organizzazioni di file alternative, ciascuna appropriata in certe situazioni
- Se le interrogazioni con selezione di intervallo sono frequenti, l'ordinamento del file o la costruzione di un indice sono importanti
  - Indici basati su hash sono buoni solo per ricerche con selezione di uguaglianza
  - File ordinati e indici basati su alberi sono migliori per ricerche con selezione di intervallo; buoni anche per ricerche con selezione di uguaglianza (nella pratica i file sono raramente mantenuti ordinati: è meglio un indice ad albero B+)
- Un indice è una collezione di data entry più un modo per trovare rapidamente le data entry quando vengono dati dei valori per la chiave.

Sommario (segue)

- Le data entry possono in realtà essere record di dati, coppie <chiave, rid> oppure coppie <chiave, lista-di-rid>
  - La scelta è ortogonale alla tecnica di indicizzazione usata per localizzare le data entry con un dato valore per la chiave
- Si possono avere diversi indici su un dato file di record di dati, ciascuno con una diversa chiave di ricerca
- Gli indici possono essere classificati in clustered e non clustered, primari e secondari, densi o sparsi. Le differenze hanno conseguenze importanti sull'utilità e sulle prestazioni

Sommario (segue)

- Capire la natura del carico di lavoro per l'applicazione, e gli obiettivi prestazionali, è essenziale per sviluppare un buon progetto
  - Quali sono le interrogazioni e gli aggiornamenti importanti? Quali attributi/relazioni sono coinvolti?

Sommario (segue)

- Gli indici devono essere scelti per velocizzare le interrogazioni importanti (e magari qualche aggiornamento!)
  - Il mantenimento degli indici incide sugli aggiornamenti dei campi chiave
  - Scegliete indici che possono aiutare molte interrogazioni, se possibile
  - Costruite indici per supportare strategie basate solo sull'indice
  - L'uso del clustering è una decisione importate; soltanto un indice per ogni relazione può essere clustered!
  - L'ordine dei campi nella chiave di un indice composito può essere importante