

LA QUALITÀ DEL SERVIZIO: L'EFFICIENZA DI UN SISTEMA DI GESTIONE DATI



QUALITÀ DEL SERVIZIO

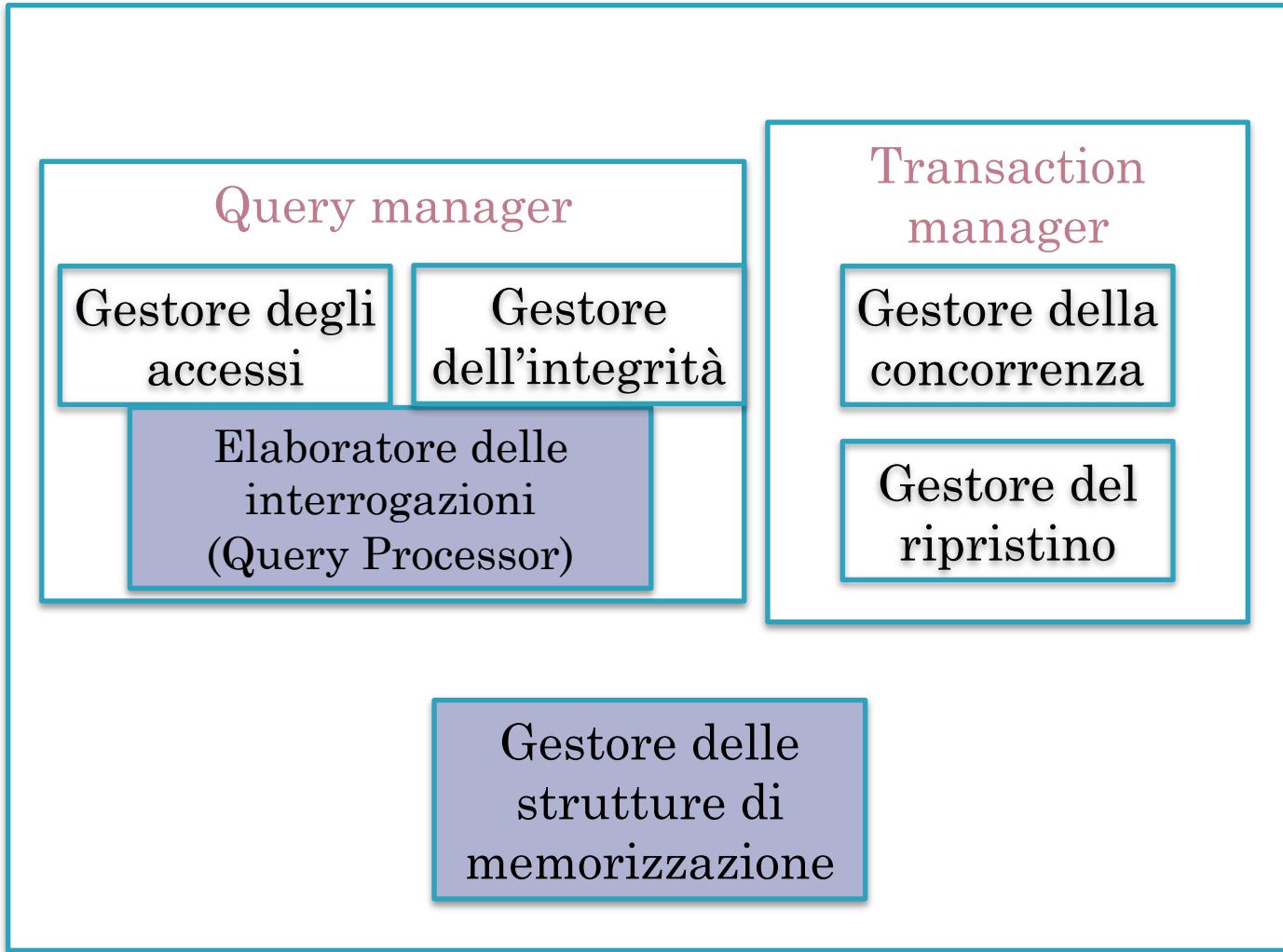
- Come migliorare le prestazioni del sistema
 - nell'eseguire le interrogazioni
 - nell'eseguire le transazioni (controllo della concorrenza e gestione del ripristino)
 - non lo vediamo
- Le prestazioni vengono analizzate in riferimento a un certo carico di lavoro – workload
- Tuning: attività con cui si analizzano le prestazioni del sistema e si cercano di adottare correttivi
- Tuning = progettazione quando il sistema non è ancora in produzione

CARICO DI LAVORO (WORKLOAD)

- **Workload**
 - quali sono le interrogazioni più importanti e con che frequenza vengono eseguite
 - quali sono gli aggiornamenti più importanti e con che frequenza vengono eseguiti
 - quali sono le prestazioni desiderate/necessarie per tali interrogazioni e aggiornamenti e per il sistema in generale
- Informazioni sul workload possono essere determinate
 - Dal documento di analisi dei requisiti
 - Dal monitoraggio del sistema
- Ci concentriamo principalmente su workload composti da sole interrogazioni

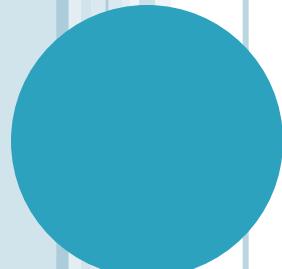
COMPONENTI DI UN DBMS

DBMS



COSA FAREMO

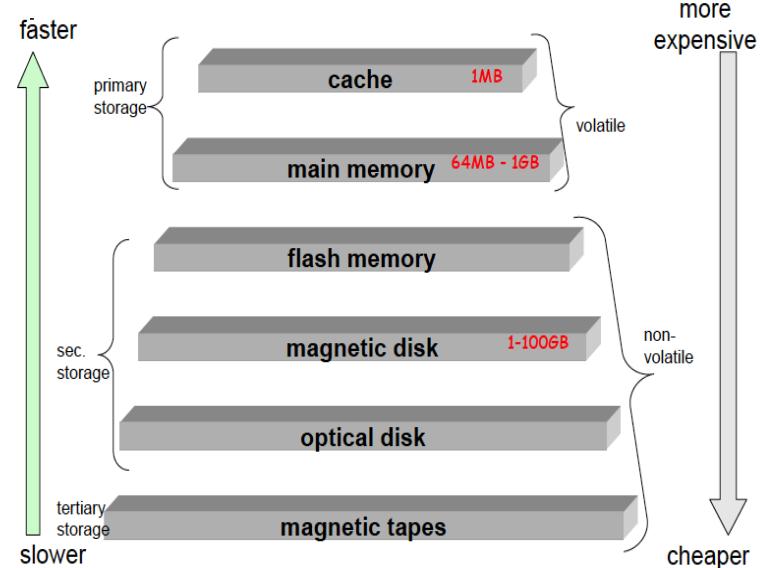
- Ci concentreremo sul gestore delle strutture di memorizzazione e sul query processor
- Ne studieremo le principali funzionalità
- Cercheremo di capire come migliorare le prestazioni del sistema nell'eseguire le interrogazioni contenute nel workload, agendo
 - a livello logico/esterno
 - a livello fisico
 - al livello di specifica di interrogazioni SQL



GESTORE DELLE STRUTTURE DI MEMORIZZAZIONE

6

GERARCHIA DELLE MEMORIE



- La memoria di un sistema di calcolo è organizzata in una **gerarchia**. Al livello più alto memorie di piccola dimensione, molto veloci, costose; scendendo lungo la gerarchia la dimensione aumenta, diminuiscono la velocità e il costo
- **Prestazioni di una memoria:**
 - dato un indirizzo di memoria, le prestazioni si misurano in termini di tempo di accesso, determinato dalla somma della
 - **latenza** (tempo necessario per accedere al primo byte), e del
 - **tempo di trasferimento** (tempo necessario per muovere i dati)

tempo di accesso = latenza +

$$\frac{\text{dimensione dati da trasferire}}{\text{velocità di trasferimento}}$$

LEGGE DI MOORE

- Gordon Moore: “*Integrated circuits are improving in many ways, following an exponential curve that doubles every 18 months*”
 - Velocità dei processori
 - Numero di bit integrabili in un chip
 - Numero di byte memorizzabili in un hard disk (HD)
- Parametri che **NON** seguono la legge di Moore:
 - **Tempo di accesso alle memorie**
 - **Velocità a cui gli HD ruotano**
- La conseguenza è che **diventa relativamente sempre più oneroso muovere i dati lungo i livelli della gerarchia**

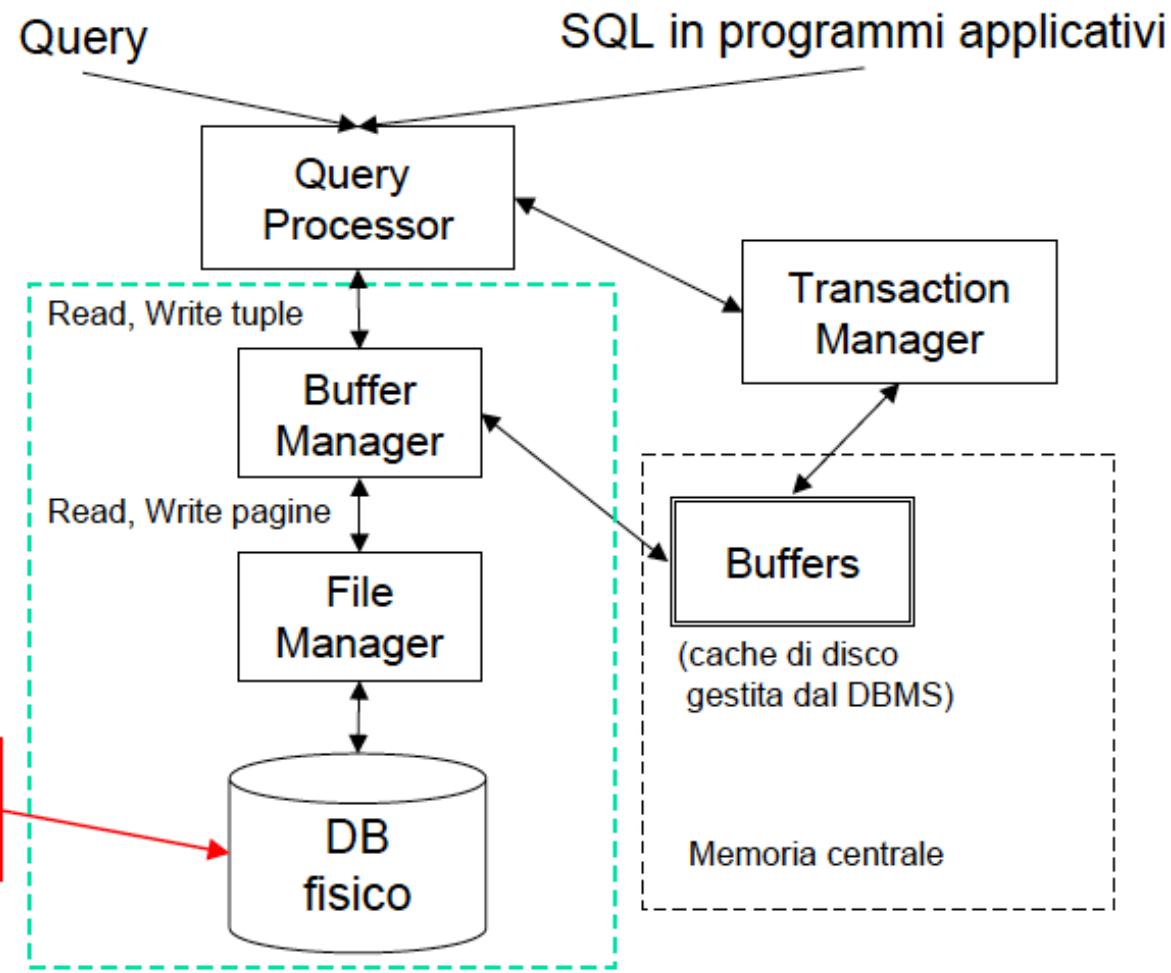
IMPLICAZIONI PER I DBMS

- Un DB, a causa della sua dimensione, risiede normalmente su dischi (eventualmente anche su altri tipi di dispositivi)
- I dati devono essere trasferiti in memoria centrale per essere elaborati dal DBMS
- Il trasferimento non avviene in termini di singole tuple, bensì di blocchi (o pagine, termine comunemente usato quando i dati sono in memoria)
- Poiché spesso le operazioni di I/O costituiscono il collo di bottiglia del sistema, si rende necessario ottimizzare l'implementazione fisica del DB, attraverso:
 - Gestore delle strutture di memorizzazione
 - Organizzazione efficiente delle tuple su disco
 - Gestione efficiente dei buffer in memoria
 - Strutture di accesso efficienti
 - Elaboratore delle interrogazioni
 - Strategie di esecuzione efficienti per le query

GESTORE DELLE STRUTTURE DI MEMORIZZAZIONE

Gestore delle strutture di memorizzazione

Il DB logico è fisicamente memorizzato su dispositivi di memoria permanente



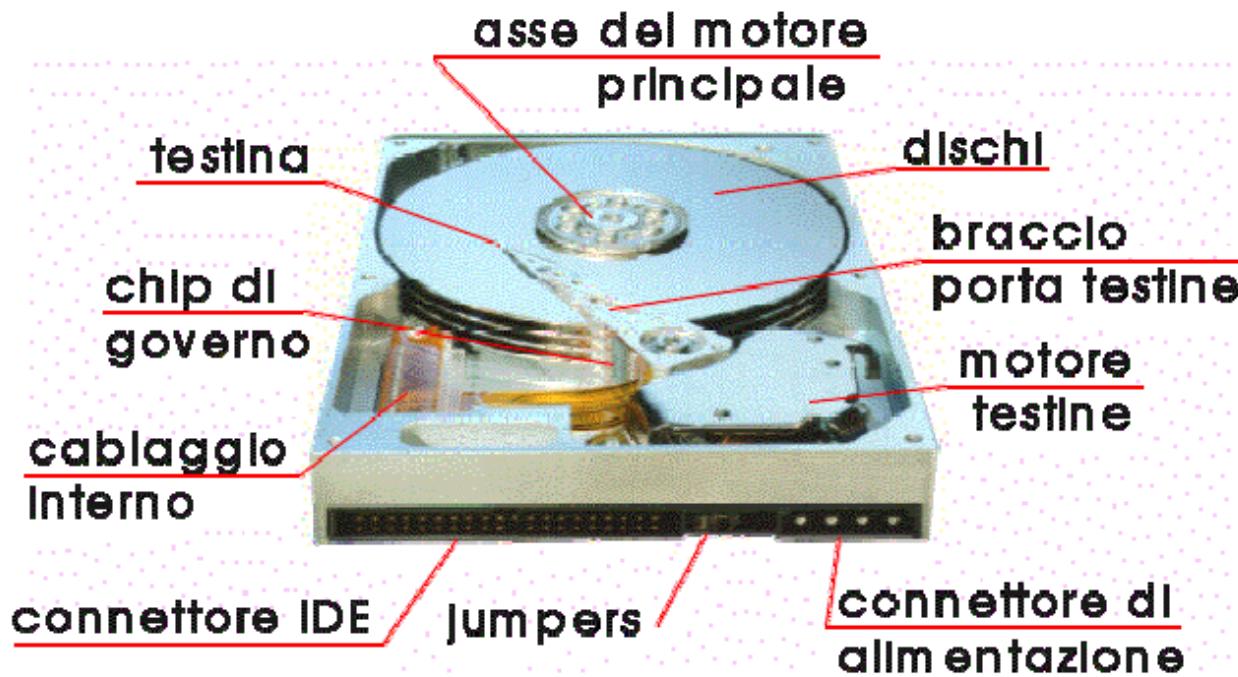
GESTORE DEI FILE

TIPI DI DISPOSITIVI

- Per il loro interesse e la loro diffusione, presentiamo i **dischi magnetici (hard disk)**
- ... anche per capire quanto può richiedere l'esecuzione di una query!

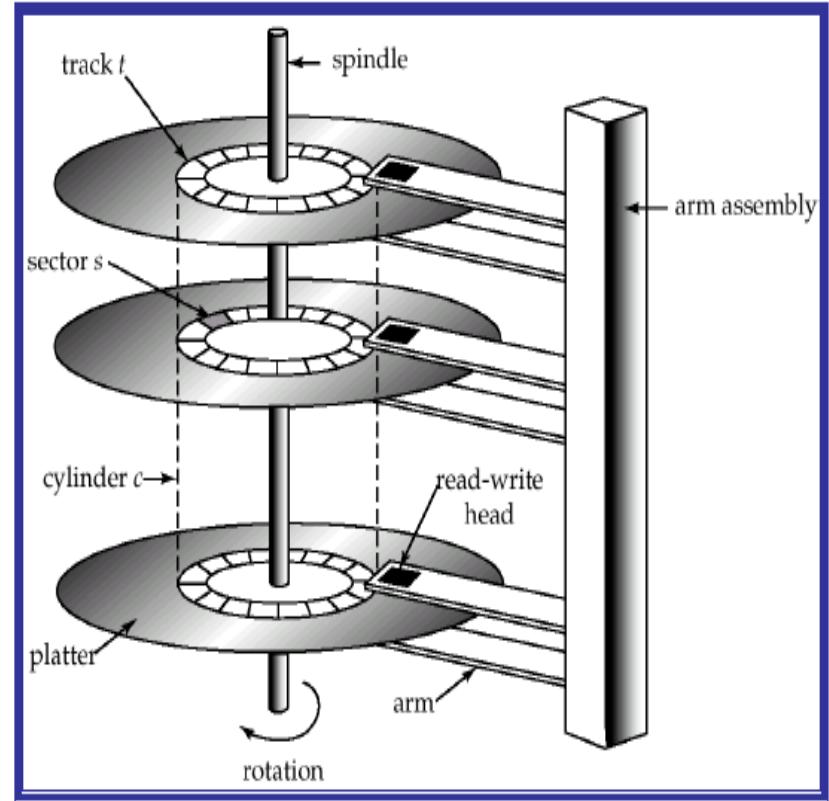
DISCHI MAGNETICI

- Un hard disk (HD) è un **dispositivo elettro-meccanico** per la conservazione di informazioni sotto forma magnetica, su supporto rotante a forma di piatto su cui agiscono delle testine di lettura/scrittura



DISCHI MAGNETICI

- L'informazione memorizzata su una superficie del disco in cerchi concentrici di piccola ampiezza detti **tracce**
 - anche > 16.000
- Ogni traccia suddivisa in **settori**
 - **più piccola unità** di dati che può essere letta o scritta
 - spesso 512 kb
 - 100-1000 settori per traccia
- Per leggere/scrivere un settore:
 - il braccio si posiziona sulla traccia
 - i dati vengono letti/scritti quando il settore passa sotto la testina
- Dischi a piatti multipli
 - una testina per superficie
 - un unico braccio
- Le tracce con lo stesso diametro sulle varie superfici sono dette **cilindro**
- **Dati memorizzati su uno stesso cilindro possono essere recuperati molto più velocemente che non dati distribuiti su diversi cilindri**



DISCHI MAGNETICI - PRESTAZIONI

- Le prestazioni possono essere classificate in:
 - **Interne**: dipendono da
 - Caratteristiche meccaniche
 - Tecniche di memorizzazione e codifica dei dati
 - Disk controller (interfaccia tra l'HW del disco e il sistema di calcolo)
 - Misurate in tempo di latenza e tempo di trasferimento
 - **Esterne**: dipendono da
 - Tipo di interfaccia
 - Architettura del sottosistema di I/O

tempo di accesso = latenza +

$$\frac{\text{dimensione dati da trasferire}}{\text{velocità di trasferimento}}$$

DISCHI MAGNETICI – TEMPO DI LATENZA

- Tempo impiegato per accedere al primo byte di interesse

Seek Time(Ts): tempo impiegato dal braccio a posizionarsi sulla traccia desiderata (min 2-10 msec, max 2-20 msec)

Command Overhead Time: tempo necessario a impartire comandi al drive (circa 0.5 msec, trascurabile)

Rotational Latency (Tr): tempo di attesa del primo settore da leggere (da 2 a 11 msec)

Component	Best-Case Figure (ms)	Worst-Case Figure (ms)
Command Overhead	0.5	0.5
Seek Time (Ts)	2.2	15.5
Settle Time	<0.1	<0.1
Rotational Latency (Tr)	0.0	8.3
Total	2.8	28.4

IBM 34GXP drive

Settle Time: tempo richiesto per la stabilizzazione del braccio

Tempo di latenza = qualche decina di msec nel caso peggiore

DISCHI MAGNETICI – LATENZA ROTAZIONALE

RPM =
rounds per minute

Spindle Speed (RPM)	Worst-Case Latency (Full Rotation) (ms)	Average Latency (Half Rotation) (ms)
3,600	16.7	8.3
4,200	14.2	7.1
4,500	13.3	6.7
4,900	12.2	6.1
5,200	11.5	5.8
5,400	11.1	5.6
7,200	8.3	4.2
10,000	6.0	3.0
12,000	5.0	2.5
15,000	4.0	2.0

DISCHI MAGNETICI - TEMPO DI TRASFERIMENTO

- Tempo per trasferire un certo numero di byte
- Dipende dalla **velocità di trasferimento (o transfer rate)**: la velocità massima alla quale il drive può leggere o scrivere i dati
 - Tipicamente dell'ordine di qualche decina di MB/sec
- Si riferisce alla velocità con cui si trasferiscono bit dai (sui) piatti sulla (dalla) cache del controller
- Si può stimare come: (dim dati su una traccia) / tempo di rotazione

$$\frac{(\text{bytes/sector}) \times (\text{sectors/track})}{\text{rotation time}}$$

BLOCCHI

- I dati sono trasferiti tra il disco e la memoria principale in unità chiamate **blocchi/pagine**
- Un blocco (o pagina) è una sequenza contigua di settori su una traccia, e costituisce l'unità di I/O per il trasferimento di dati da/per la memoria principale
- La dimensione del blocco dipende dal sistema operativo
 - La dimensione tipica di una pagina è di qualche KB (4 -64 KB)
 - Pagine piccole comportano un maggior numero di operazioni di I/O
 - Pagine grandi tendono ad aumentare la frammentazione interna (pagine parzialmente riempite) e richiedono più spazio in memoria per essere caricate

BLOCCHI

- Il **tempo di trasferimento di un blocco** (**Tt**) è il tempo impiegato dalla testina per trasferire un blocco nel buffer, una volta posizionata all'inizio del blocco
- tale tempo è molto più breve del tempo di seek e dipende dalla dimensione del blocco (**P**) e dalla velocità di trasferimento (**Tr**):
 - $Tt = P / Tr$
- Esempio:
 - transfer rate: 21.56 MB/sec
 - dimensione blocco $P = 4 \text{ KB}$
 - tempo di trasferimento $Tt = 4 / (21.56 * 1024) = 0.18 \text{ msec}$
 - dimensione blocco $P = 64 \text{ KB}$
 - tempo di trasferimento $Tt = 64 / (21.56 * 1024) = 2.9 \text{ msec}$

Tempo di trasferimento = upper bound 1msec per blocchi da 4KB

DICHI MAGNETICI - PRESTAZIONI

- **Tempo di latenza**
 - qualche decina di msec nel caso peggiore
- **Tempo di trasferimento**
 - circa 1msec per blocchi da 4KB
- Il gestore delle strutture di memorizzazione deve cercare di ridurre il tempo di latenza

IL DB FISICO (LIVELLO FISICO)

- A livello fisico un DB consiste di un insieme di file, ognuno dei quali viene visto come una collezione di pagine, di dimensione fissa (es: 4 KB)
- Ogni pagina/blocco memorizza più record (corrispondenti alle tuple logiche)
- A sua volta un record consiste di più campi, di lunghezza fissa e/o variabile, che rappresentano gli attributi
- Casi limite:
 - Ogni relazione del DB è memorizzata in un proprio file
 - Tutto il DB è memorizzato in un singolo file
- In pratica ogni DBMS a livello fisico adotta soluzioni specifiche più articolate e flessibili

IL DB FISICO IN POSTGRESQL

- L'insieme dei tutti i **database** gestiti da un server PostgreSQL si chiama **database cluster**
- Ogni database contiene **tabelle** e altri oggetti
- **Ogni tabella è memorizzata in un file separato**
- PostgreSQL organizza lo spazio fisico in **tablespace**, ciascuno corrispondente a una posizione su disco in cui vengono memorizzati i file corrispondenti a oggetti dei database
- Il file corrispondente ad una tabella è memorizzato in un singolo tablespace, ma un tablespace può contenere più relazioni
- **L'uso dei tablespace permette al sistema di decidere dove memorizzare i vari oggetti**
 - L'uso dei tablespace permette di raggiungere le migliori prestazioni nel caso di DB di grandi dimensioni

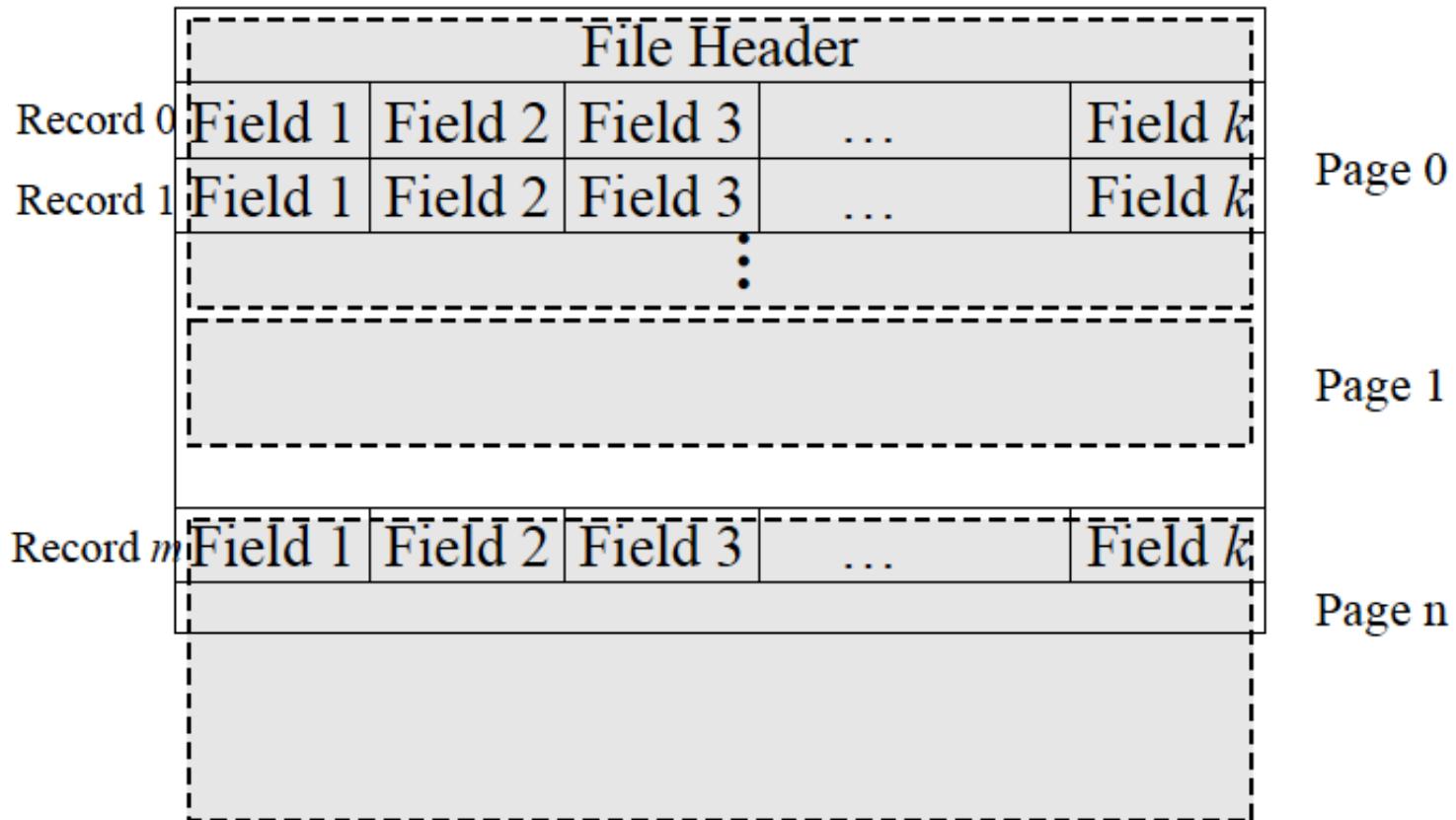
PERCHÉ NON AFFIDARSI SEMPRE AL FILE SYSTEM?

- Le prestazioni di un DBMS dipendono fortemente da come i dati sono organizzati su disco
- Intuitivamente, l'allocazione dei dati dovrebbe mirare a ridurre i tempi di accesso ai dati, e per far questo bisogna sapere come (logicamente) i dati dovranno essere elaborati e quali sono le relazioni (logiche) tra i dati
- Tutte queste informazioni non possono essere note al file system
- Esempi:
 - Se due relazioni contengono dati tra loro correlati (mediante join) può essere una buona idea memorizzarle in cilindri vicini, in modo da ridurre i tempi di seek (certamente nello stesso tablespace)

ORGANIZZAZIONE DEI DATI NEI FILE

Schema di riferimento (semplificato)

File



RECORD

- I dati sono generalmente memorizzati in forma di **record**
 - un record è costituito da un insieme di valori collegati
 - ogni valore è formato da uno o più byte e corrisponde ad un campo del record
- Una collezione di nomi di campi a cui sono associati i tipi corrispondenti costituisce un **tipo di record**
- Per ogni tipo di record nel DB deve essere definito uno schema (fisico) che permetta di interpretare correttamente il significato dei byte che costituiscono il record

RECORD RAPPRESENTAZIONE DEI VALORI

- CHAR (n): n byte
- VARCHAR(n): m + p byte
 - m numero caratteri (= byte) nella stringa
 - p numero di byte per rappresentare m (p = 1 per m < $2^8 = 256$)
- DATE, TIME: stringhe di lunghezza fissa
 - DATE: 10 caratteri YYYY-MM-DD
 - TIME: 8 caratteri HH:MM:SS

ESEMPIO

LIVELLO LOGICO

Relazione (tabella)

Tupla

CREATE TABLE Film

```
(titolo  VARCHAR(30),  
regista  VARCHAR(20),  
anno    DECIMAL(4) NOT NULL,  
genere   CHAR(15) NOT NULL,  
valutaz  NUMERIC(3,2),  
PRIMARY KEY (titolo,regista))
```

LIVELLO FISICO

File

Record

```
Struct Film {char titolo [30],  
            char regista [20],  
            int anno,  
            char genere [15],  
            real valutaz};
```

- Ogni record memorizzato in 73 byte
 - $30+20+15=65$ per le stringhe
 - 4 per l'intero
 - 4 per il reale

FILE

- un **file** è una sequenza di record
- un file è detto **file con record a lunghezza fissa** se tutti i record memorizzati nel file hanno la stessa dimensione (in byte)
- altrimenti, parliamo di **file con record a lunghezza variabile**

FILE CON RECORD A LUNGHEZZA FISSA

- ogni record ha gli stessi campi e la lunghezza dei campi è fissa
- si possono identificare
 - la posizione di partenza del record
 - la posizione di partenza di ogni campo rispetto alla posizione di partenza del record
- l'accesso ai campi del record è quindi facilitato

```
Struct Film {char titolo [30],  
            char regista [20],  
            int anno,  
            char genere [15],  
            real valutaz}
```

titolo	regista	anno	genere	valutaz
		50	54	69

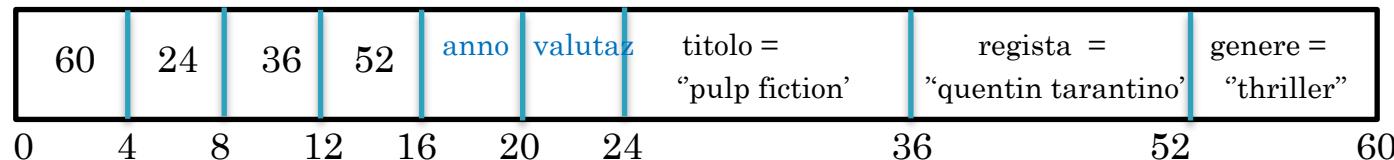
FILE CON RECORD A LUNGHEZZA VARIABILE

- Un file può contenere record a **lunghezza variabile** per varie ragioni:
 - **campi di dimensione variabile** (ad esempio valori di tipo VARCHAR)
 - **campi opzionali**
 - tipo di record Film: la valutazione può non essere specificata
 - **file eterogeneo**: il file contiene record di tipi differenti, e quindi di dimensioni differenti
 - es: le informazioni di un cliente memorizzate nello stesso file vicino ai noleggi da lui effettuati
 - ...

FILE CON RECORD A LUNGHEZZA VARIABILE

- Nel caso di record a lunghezza variabile si hanno diverse alternative, che devono considerare anche i problemi legati agli aggiornamenti che modificano la lunghezza dei campi (e quindi dei record)
- Una soluzione consolidata consiste nel
 - memorizzare prima tutti i campi a lunghezza fissa, e quindi tutti quelli a lunghezza variabile
 - per ogni campo a lunghezza variabile si ha un “**prefix pointer**” che riporta l’indirizzo del primo byte del campo

```
Struct Film {  varchar titolo [30],  
              varchar regista [20],  
              int anno,  
              varchar genere [15],  
              real valutaz}
```

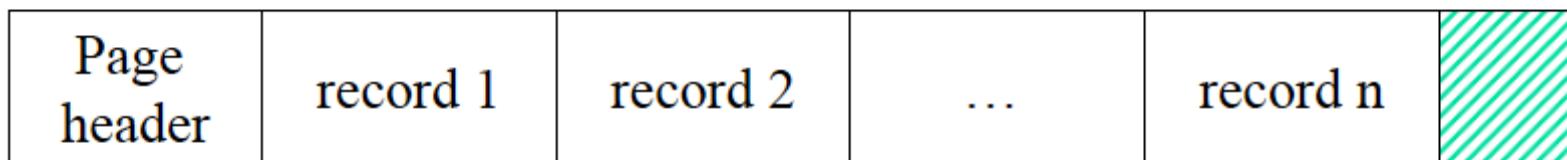


RECORD HEADER

- In generale ogni record include un **header** che, oltre alla lunghezza del record, può contenere:
 - L'**identificatore della relazione** cui il record appartiene
 - L'**identificatore univoco RID** del record nel DB
 - Un **timestamp** che indica quando il record è stato inserito o modificato l'ultima volta
- Il formato specifico dell'header ovviamente varia da un DBMS all'altro

ORGANIZZAZIONE DEI RECORD IN BLOCCHI

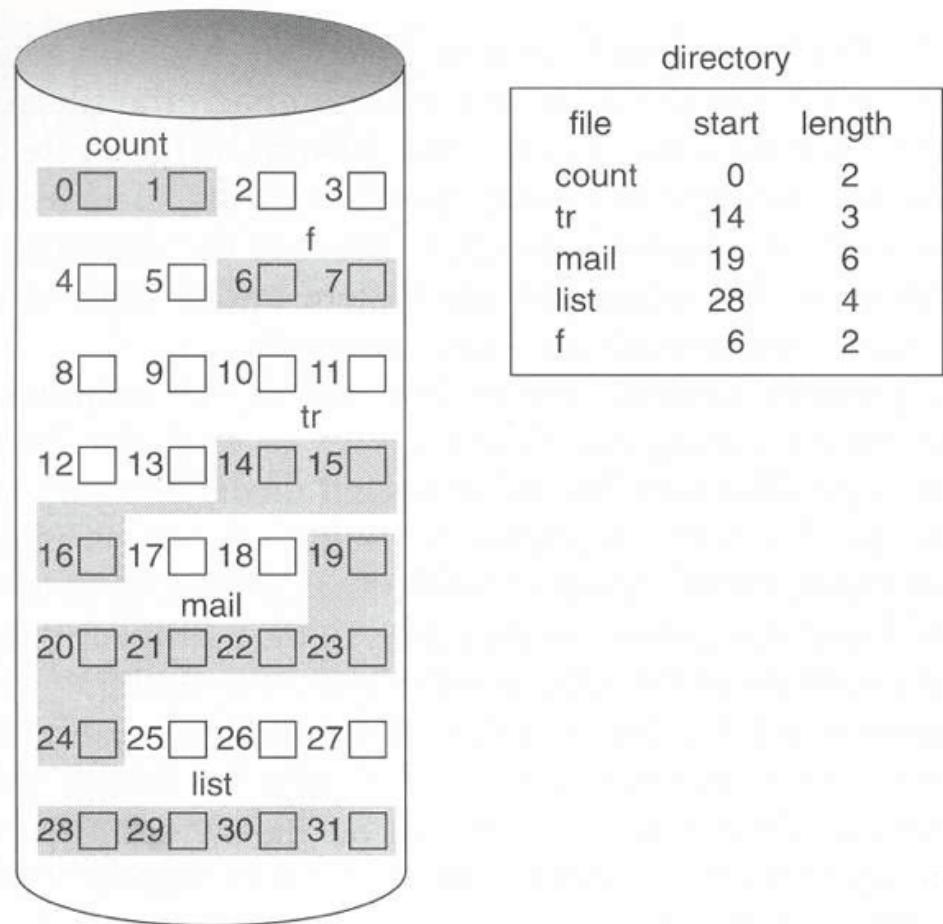
- Normalmente la dimensione di un record è (molto) minore di quella di una pagina
 - Esistono tecniche particolari (che qui non vediamo) per gestire il caso di “long tuples”, la cui dimensione eccede quella di una pagina
- Nel caso di record a lunghezza fissa l'organizzazione in una pagina si potrebbe presentare così:



- Il **page header** mantiene informazioni quali:
 - ID della pagina nel DB, timestamp che indica quando la pagina è stata modificata l'ultima volta, relazione cui le tuple nella pagina appartengono, ecc.
- Normalmente un record è contenuto interamente in una pagina
 - Quindi si può avere dello spazio sprecato

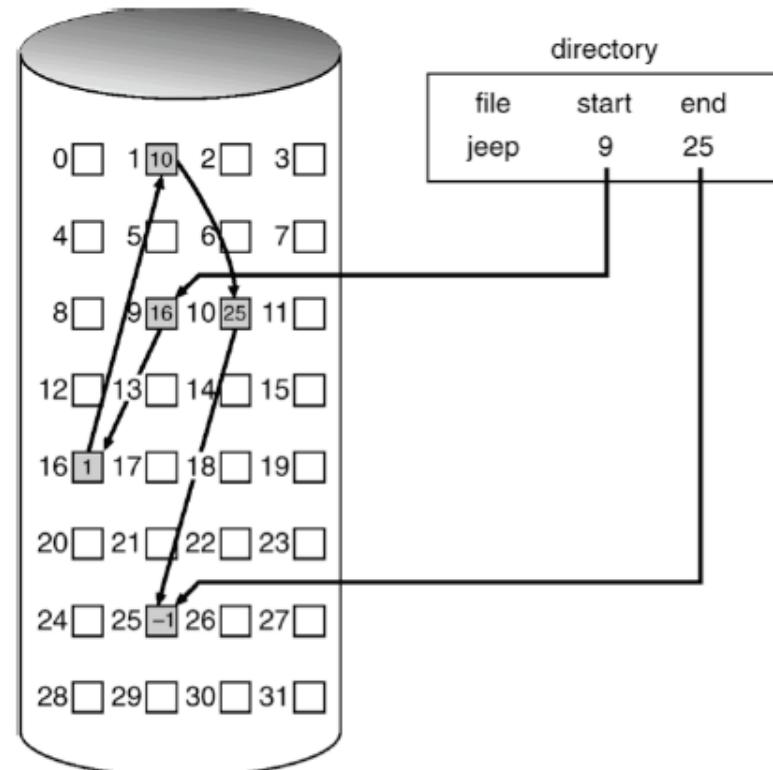
ORGANIZZAZIONE DEI BLOCCHI SU DISCO: ALLOCAZIONE CONTINUA

- Tipico problema affrontato anche dai sistemi operativi
- **Allocazione contigua:** i blocchi del file sono allocati in blocchi di disco contigui
- lettura dell'intero file molto efficiente
- espansione del file difficile



ORGANIZZAZIONE DEI BLOCCHI SU DISCO: ALLOCAZIONE CONCATENATA

- **Allocazione concatenata:** ogni blocco di un file contiene un puntatore al successivo blocco del file
- espansione del file facile
- lettura dell'intero file lenta



ORGANIZZAZIONE DEI BLOCCHI SU DISCO: ALLOCAZIONE CONCATENATA

- Utilizzo di **bucket** (cioè un insieme di blocchi, non necessariamente contigui ma ‘vicini’, possibilmente nello stesso cilindro) per gruppi di record tra loro collegati (ad esempio, tutti i noleggi di un certo cliente)

ORGANIZZAZIONE DEI RECORD NEI FILE

- I file che contengono i record dei dati costituiscono l'**organizzazione primaria dei dati**
- Il modo con cui i record vengono organizzati nei file incide sull'efficienza delle operazioni e sull'occupazione di memoria
 - Poiché i dati sono trasferiti in blocchi tra la memoria secondaria e la memoria principale, è importante assegnare i record ai blocchi in modo tale che uno stesso blocco contenga record tra loro interrelati
 - Poiché la lettura di blocchi vicini riduce il tempo di seek, è importante memorizzare blocchi contenenti record interrelati il "più vicino possibile" su disco
- Le informazioni sull'organizzazione primaria dei dati sono contenute in **cataloghi di sistema**, che dipendono dallo specifico DBMS considerato e in generale contengono per ogni relazione
 - Numero tuple
 - Numero blocchi nel file
 - Altre statistiche (ci torneremo in seguito)
 - Informazioni su strutture ausiliarie per l'accesso ai dati

ESEMPIO

```
SELECT *  
FROM Noleggio;
```

- L'ordine di memorizzazione dei record nel file è ininfluente a fini prestazionali perché devono essere restituite tutte le tuple

ESEMPIO

```
SELECT *
FROM Noleggio
WHERE dataNol = '15-mar-2006';
```

- Se i record fossero memorizzati ordinati nel file, si potrebbe procedere sequenzialmente, fino al primo record con `dataNol = '15-mar-2006'` e leggere fino al primo record con `dataNol > '15-mar-2006'`
- Si riduce il numero di blocchi letti
- Si possono comunque leggere blocchi che non contengono risultati
 - I blocchi contenenti i film con `dataNol < '15-mar-2006'` (precedono i blocchi contenenti i record di nostro interesse se il file è ordinato rispetto a `dataNol`)

ESEMPIO

```
SELECT *
FROM Noleggio
WHERE dataNol = '15-mar-2006';
```

- Se si potesse ‘ricordare’ la posizione del primo blocco contenente record con `dataNol = '15-mar-2006'`, si ridurrebbe il numero di blocchi ‘inutili’ letti

ORGANIZZAZIONE DEI RECORD NEI FILE

- A seconda dell'organizzazione primaria dei dati scelta il file che contiene i record dei dati può essere
 - **file heap** (o file pila): i record dati vengono memorizzati uno dopo l'altro in ordine di inserimento
 - **file ordinato** (o file sequenziale o file **clusterizzato**): i record dati sono memorizzati mantenendo l'ordinamento su uno o più campi
 - **file hash**: i record che condividono lo stesso valore per uno o più campo sono memorizzati «vicini» (in genere nello stesso bucket)
- Le organizzazioni ordinate e hash si possono ottenere usando opportune strutture ausiliarie di accesso (le vedremo in seguito)

ESEMPIO

File heap

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?
1116	21-Mar-2006	6610	?

File ordinato su dataNol

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

File hash su dataNol

colloc	dataNol	codCli	dataRest
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

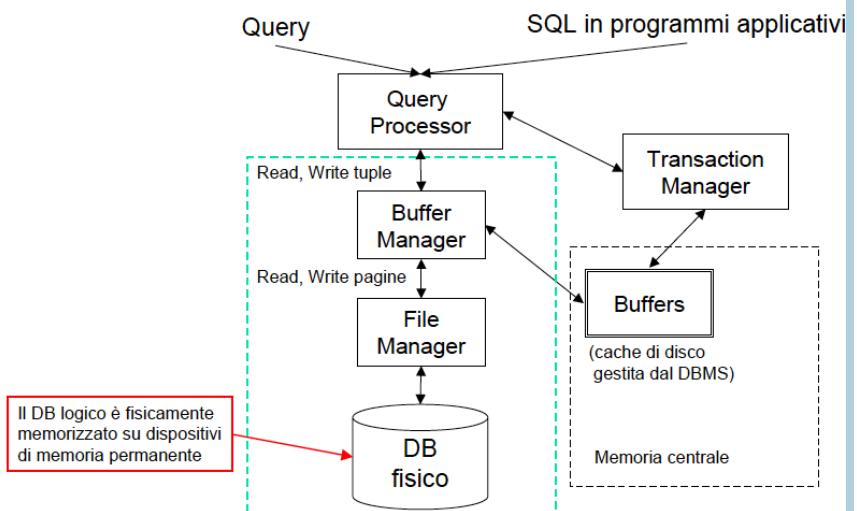
GESTORE DEL BUFFER

PROBLEMA

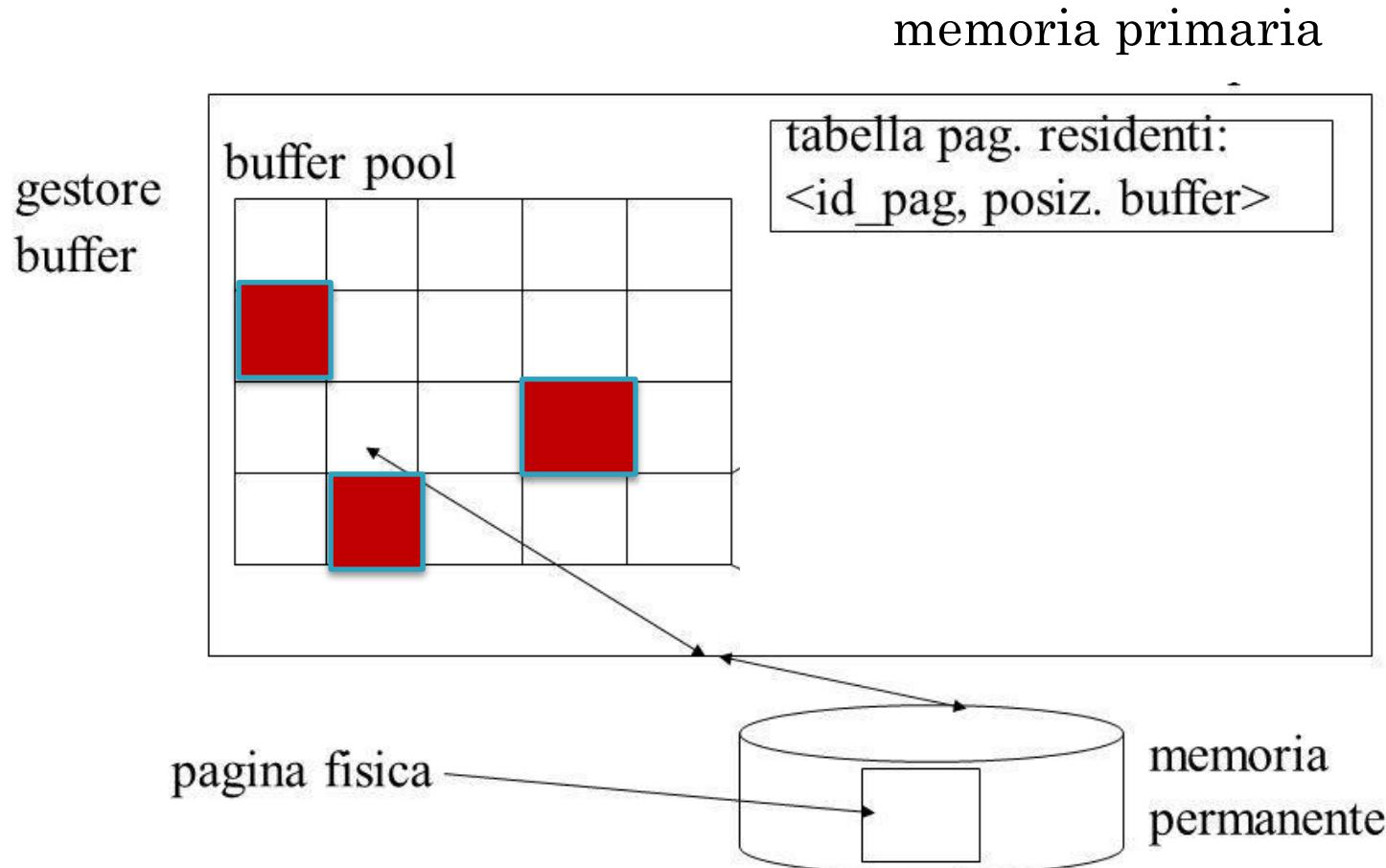
- L'obiettivo principale delle strategie di memorizzazione è di minimizzare gli accessi a disco
- Un altro modo per minimizzare gli accessi a disco consiste nel mantenere più blocchi possibile in memoria principale
- Ma dove e come?

IL BUFFER

- La lettura di una tupla richiede che la pagina corrispondente sia prima portata in memoria, in un'area gestita dal DBMS detta **buffer**
- La gestione del buffer è fondamentale dal punto di vista prestazionale ed è demandata al **gestore del buffer** (Buffer Manager, BR)
- Il buffer è organizzato in pagine, che hanno la stessa dimensione delle pagine/blocchi su disco
- Terminologia alternativa
 - Buffer → buffer pool
 - Pagina nel buffer → buffer
- Già introdotto quando abbiamo parlato di transazioni



IL BUFFER



GESTIONE DEL BUFFER

- A fronte di una richiesta di una pagina, ad esempio durante una operazione di lettura (**SELECT**) il Buffer Manager (BM) opera come segue:
 - **Se la pagina è già nel buffer**, viene fornito al programma chiamante l'indirizzo della pagina corrispondente
 - **Se la pagina non è in memoria**:
 - Il BM seleziona una pagina nel buffer per la pagina richiesta
 - Se la pagina prescelta dal buffer è occupata, questa viene riscritta su disco solo se è stata modificata e non ancora salvata su disco e se nessuno la sta usando
 - A questo punto il BM può leggere la pagina da disco e copiarla nella pagina del buffer selezionata, rimpiazzando così quella prima presente

GESTIONE DEL BUFFER

- Quando una pagina disco è presente nel buffer, il DBMS può effettuare le sue operazioni di lettura e scrittura direttamente su di essa
- Importante
 - I tempi di accesso a **disco** sono dell'ordine di **millesimi** di secondo
 - I tempi di accesso al **buffer** sono dell'ordine di **milionesimi** di secondo
 - Accedere alle pagine nel buffer invece che alle corrispondenti pagine su disco influenza notevolmente le prestazioni

GESTIONE DEL BUFFER

- Il buffer manager di un DBMS usa alcune politiche di gestione che sono più sofisticate delle politiche usate nei SO
- SO in genere si basano su politiche LRU (Least Recently Used)
 - Le pagine utilizzate meno, di recente, vengono sovrascritte
- Le politiche di LRU non sempre sono le più adatte per i DBMS
 - per motivi legati alla gestione del recovery in alcuni casi un blocco non può essere trasferito su disco
 - per motivi legati alla gestione del recovery in alcuni casi è necessario forzare un blocco su disco
 - un DBMS è in grado di predire meglio di un SO il tipo dei futuri riferimenti

GESTIONE DEL BUFFER

- Come vedremo, esistono algoritmi di join che scandiscono N volte le tuple di una relazione
- In questo caso la politica migliore sarebbe la **MRU** (Most Recently Used)
 - Si rimpiazza la pagina usata più di recente
 - Verrà infatti usata solo dopo avere letto tutte quelle successive

STRUTTURE AUSILIARIE DI ACCESSO

PERCHÉ GLI INDICI

- Le organizzazioni dei file viste (heap, sequenziale, hash) permettono di migliorare le prestazioni di ricerche rispetto a condizioni su attributi che sono quelli utilizzati per organizzare i record nel file
- Possono però portare ad accedere blocchi inutili
- Inoltre non sono efficaci per ricerche con condizioni su altri attributi

ESEMPIO

```
SELECT *
FROM Noleggi
WHERE dataNol = '15-Mar-2006'
```

efficiente
(ma accediamo blocchi inutili)

```
SELECT *
FROM Noleggi
WHERE colloc = 1122
```

non efficiente
(dobbiamo scandire tutto il file)

File ordinato su **dataNol**

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

ESEMPIO

```
SELECT *
FROM Noleggi
WHERE dataNol = '15-Mar-2006'
```

File ordinato
rispetto a
dataNol

Indice con
chiave di
ricerca
dataNol

01-Mar-2006
01-Mar-2006
02-Mar-2006
02-Mar-2006
...
15-Mar-2006
15-Mar-2006
15-Mar-2006
...

1111
1115
...
1122
1122
...

```
SELECT *
FROM Noleggi
WHERE colloc = 1122
```

File ordinato su **dataNol**

colloc	dataNol	codCli	dataRest
1111	01-Mar-2006	6635	02-Mar-2006
1115	01-Mar-2006	6635	02-Mar-2006
1117	02-Mar-2006	6635	06-Mar-2006
1118	02-Mar-2006	6635	06-Mar-2006
1111	04-Mar-2006	6642	05-Mar-2006
1119	08-Mar-2006	6635	10-Mar-2006
1120	08-Mar-2006	6635	10-Mar-2006
1116	08-Mar-2006	6642	09-Mar-2006
1118	10-Mar-2006	6642	11-Mar-2006
1121	15-Mar-2006	6635	18-Mar-2006
1122	15-Mar-2006	6635	18-Mar-2006
1113	15-Mar-2006	6635	18-Mar-2006
1129	15-Mar-2006	6635	20-Mar-2006
1119	15-Mar-2006	6642	16-Mar-2006
1126	15-Mar-2006	6610	16-Mar-2006
1112	16-Mar-2006	6610	18-Mar-2006
1114	16-Mar-2006	6610	17-Mar-2006
1128	18-Mar-2006	6642	20-Mar-2006
1124	20-Mar-2006	6610	21-Mar-2006
1115	20-Mar-2006	6610	21-Mar-2006
1124	21-Mar-2006	6642	22-Mar-2006
1116	21-Mar-2006	6610	?
1117	21-Mar-2006	6610	?
1127	22-Mar-2006	6635	?
1125	22-Mar-2006	6635	?
1122	22-Mar-2006	6642	?
1113	22-Mar-2006	6642	?

File non ordinato rispetto a **colloc**
Indice con chiave di ricerca **colloc**

INDICI

- Un indice su una relazione R è un insieme di coppie del tipo (k_i, r_i) dove:
 - k_i è un valore per la chiave di ricerca su cui l'indice è costruito
 - r_i puo` assumere diverse forme, per il momento assumiamo che sia un puntatore a un record con valore di chiave k_i
 - è quindi un RID o, al limite, un PID
 - L'indice contiene una coppia per ogni tupla di R
 - possono esistere più coppie con lo stesso valore k_i
 - esiste esattamente una coppia per un certo valore r_i

INDICI

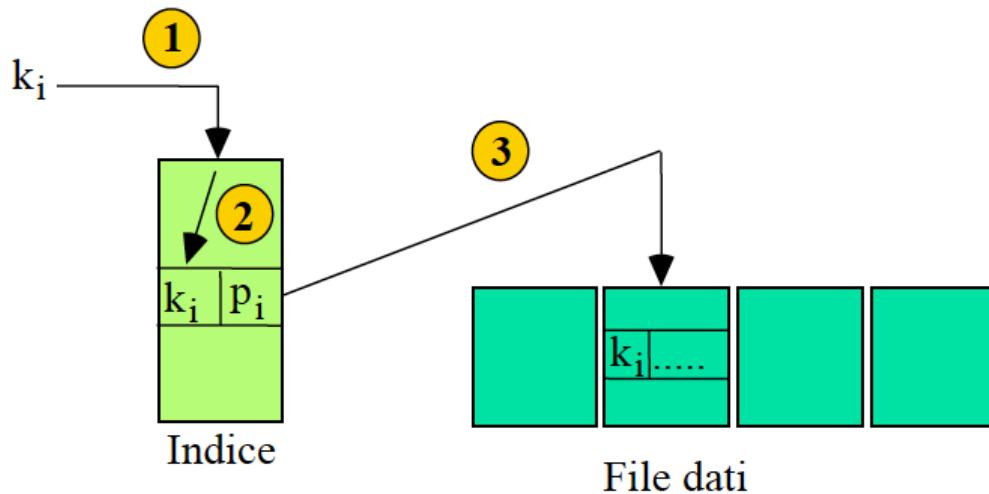
- Il vantaggio di usare un indice nasce dal fatto che la chiave è solo parte dell'informazione contenuta in un record. Pertanto, l'indice occupa uno spazio minore rispetto al file dati

ACCESSO CON INDICE: SCHEMA GENERALE

1. Accedere all'indice
2. Ricercare la coppia (k_i, p_i)
3. Accedere alla pagina dati relativa

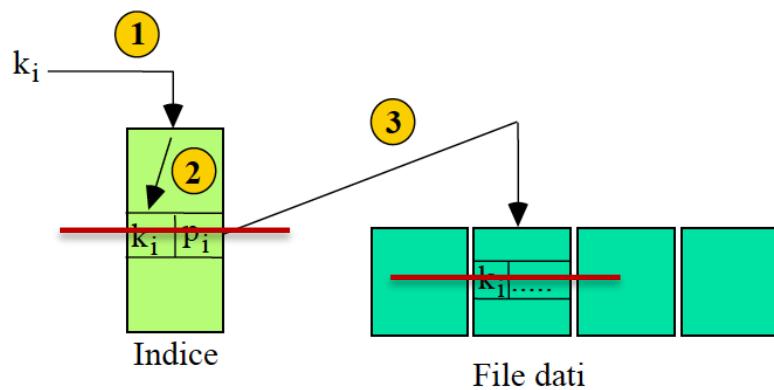
```
SELECT *
FROM R
WHERE A = ki
```

A chiave di ricerca



INDICI E AGGIORNAMENTI

- L'uso di indici rende l'esecuzione delle interrogazioni più efficiente, ma rende più costosi gli aggiornamenti
- Un'interrogazione non è mai rallentata dalla presenza di indici, al più non vengono utilizzati per la sua esecuzione
- A seguito di un aggiornamento dei dati tutti gli indici devono essere aggiornati



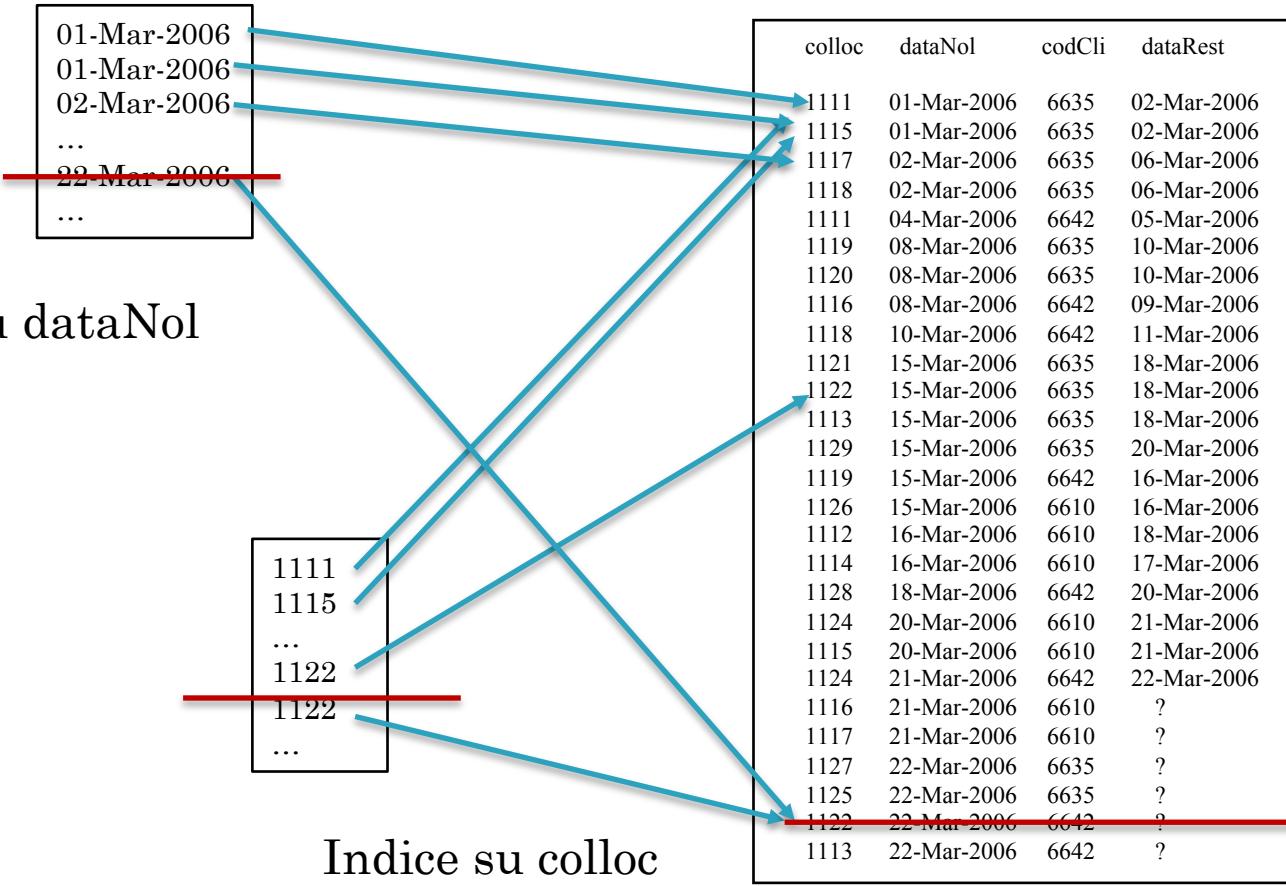
DELETE FROM R
WHERE A = k_i

A chiave di ricerca

ESEMPIO

```
DELETE FROM Noleggio  
WHERE dataNol = '22-Mar-2006'  
AND colloc = 1122
```

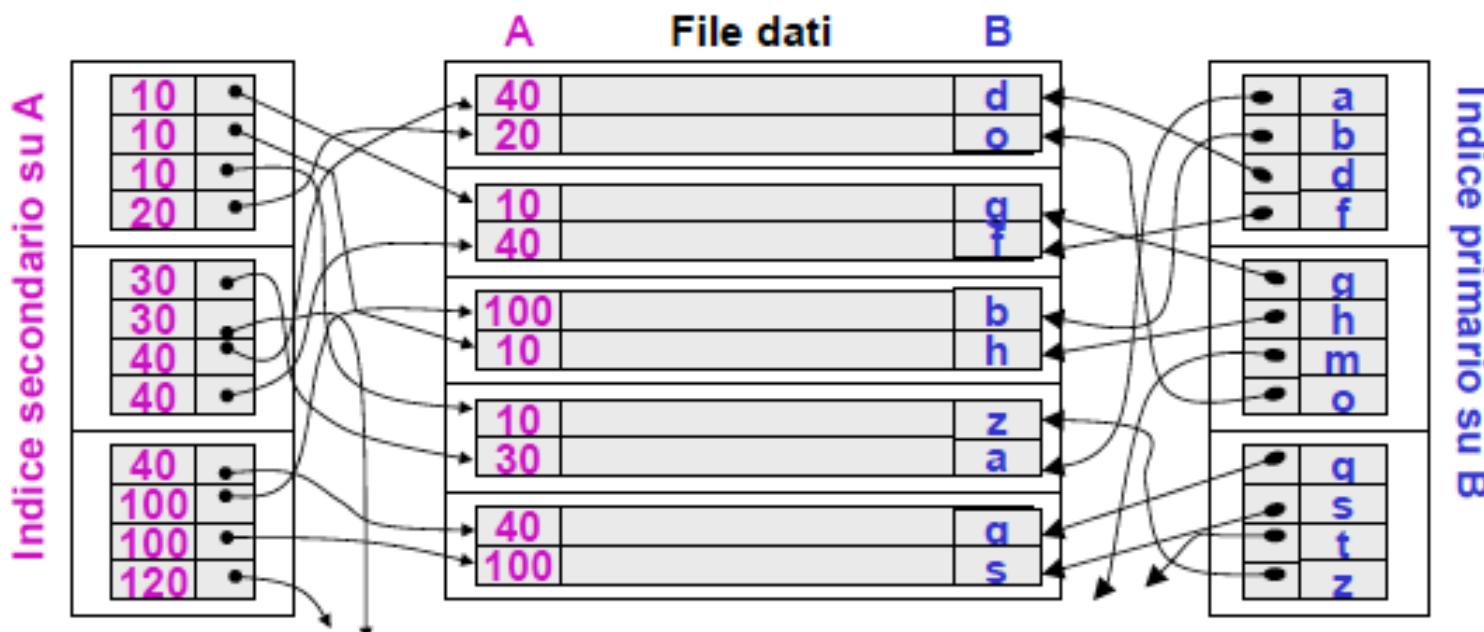
File ordinato su **dataNol**



INDICI PRIMARI E SECONDARI

Indice primario:

- La chiave di ricerca è **chiave** per la relazione
- Più coppie nell'indice **non possono** condividere lo stesso valore k per la chiave di ricerca

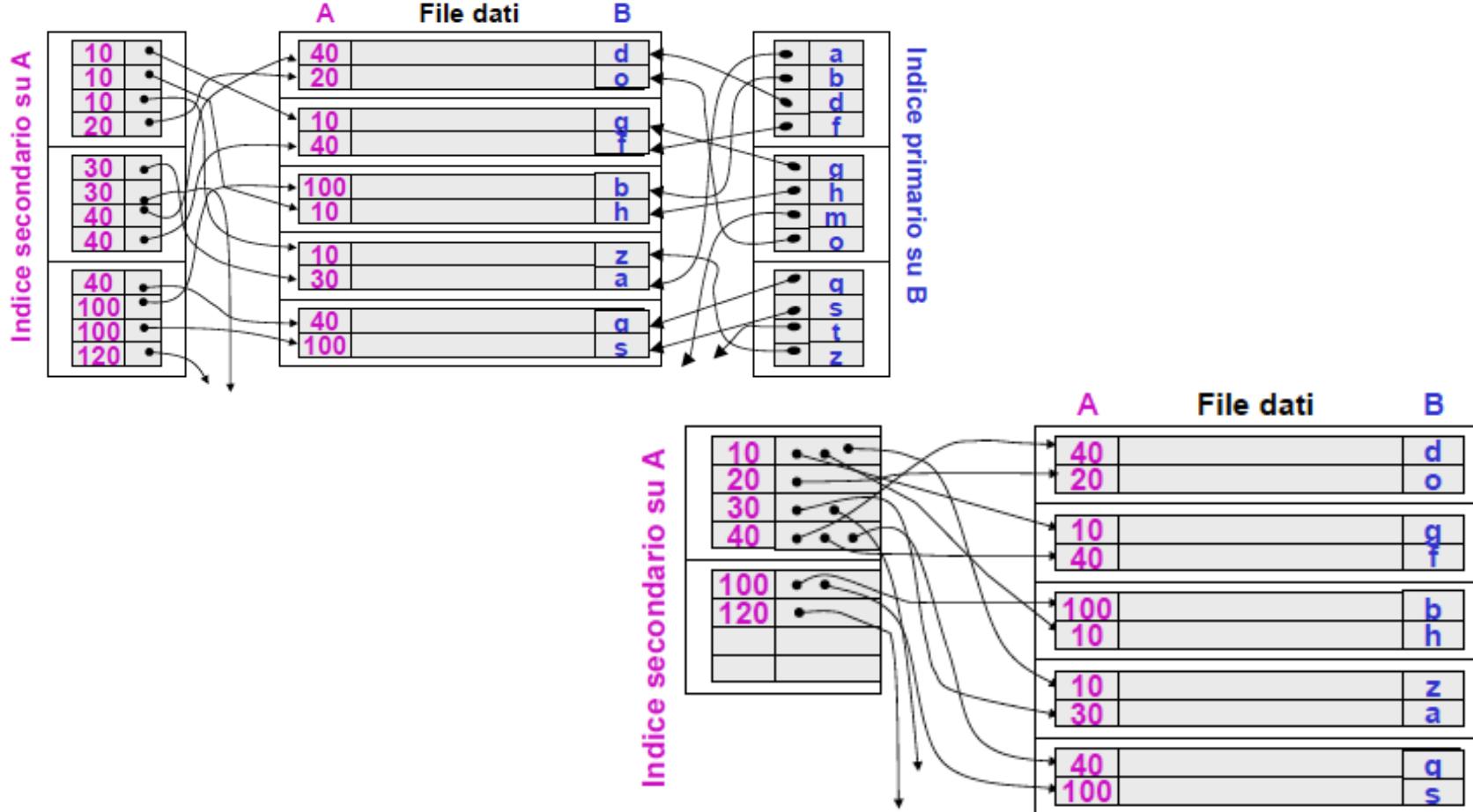


Indice secondario:

- La chiave di ricerca **non è chiave** per la relazione
- Più coppie nell'indice **possono** condividere lo stesso valore k per la chiave di ricerca

INDICI SECONDARI A LISTE DI PUNTATORI

- Per evitare di replicare inutilmente i valori di chiave, la soluzione più comunemente adottata per gli indici secondari consiste nel raggruppare tutte le coppie con lo stesso valore di chiave in una lista di puntatori



TIPI DI INDICI

- Le diverse tecniche di indice differiscono nel modo in cui organizzano l'insieme di coppie (k_i, r_i)
- Indici ordinati (anche chiamati **indici ad albero**): le coppie (k_i, r_i) vengono mantenute esplicitamente
 - Le coppie vengono salvate in un file, ordinate rispetto ai valori di chiave k_i
- Indici hash: esiste una funzione hash h per cui, se una coppia (k_i, r_i) appartiene all'indice allora $h(k_i) = r_i$
 - Le coppie non vengono memorizzate su un file apposito ma calcolate attraverso l'uso di una funzione hash

INDICI ORDINATI/AD ALBERO

INDICI ORDINATI/AD ALBERO

- Gli indici ordinati sono indici in cui l'insieme delle coppie (k_i, r_i) vengono mantenute **memorizzate in un file su disco, ordinate rispetto ai valori della chiave di ricerca k_i**
- Se l'indice è piccolo, può essere tenuto in memoria principale
- Molto spesso, però, è necessario tenerlo su disco e la scansione dell'indice può richiedere parecchi trasferimenti di blocchi
- Viene quindi utilizzata una **struttura multilivello** che permette di accedere più velocemente alle coppie dell'indice
- L'indice assume una **struttura ad albero** in cui ogni nodo dell'albero corrisponde a un blocco disco

INDICI AD ALBERO

- Gli indici ad albero sono organizzazioni ad albero bilanciato, utilizzate come **strutture di indicizzazione per dati in memoria secondaria**
- Requisiti fondamentali per indici per memoria secondaria:
 - **bilanciamento**: l'indice deve essere bilanciato rispetto ai blocchi e non ai singoli nodi (è il numero di blocchi acceduti a determinare il costo I/O di una ricerca)
 - **occupazione minima**: è importante che si possa stabilire un limite inferiore all'utilizzazione dei blocchi, per evitare un sotto-utilizzo della memoria
 - **efficienza di aggiornamento**: le operazioni di aggiornamento devono avere costo limitato

B+-TREE – CARATTERISTICHE GENERALI

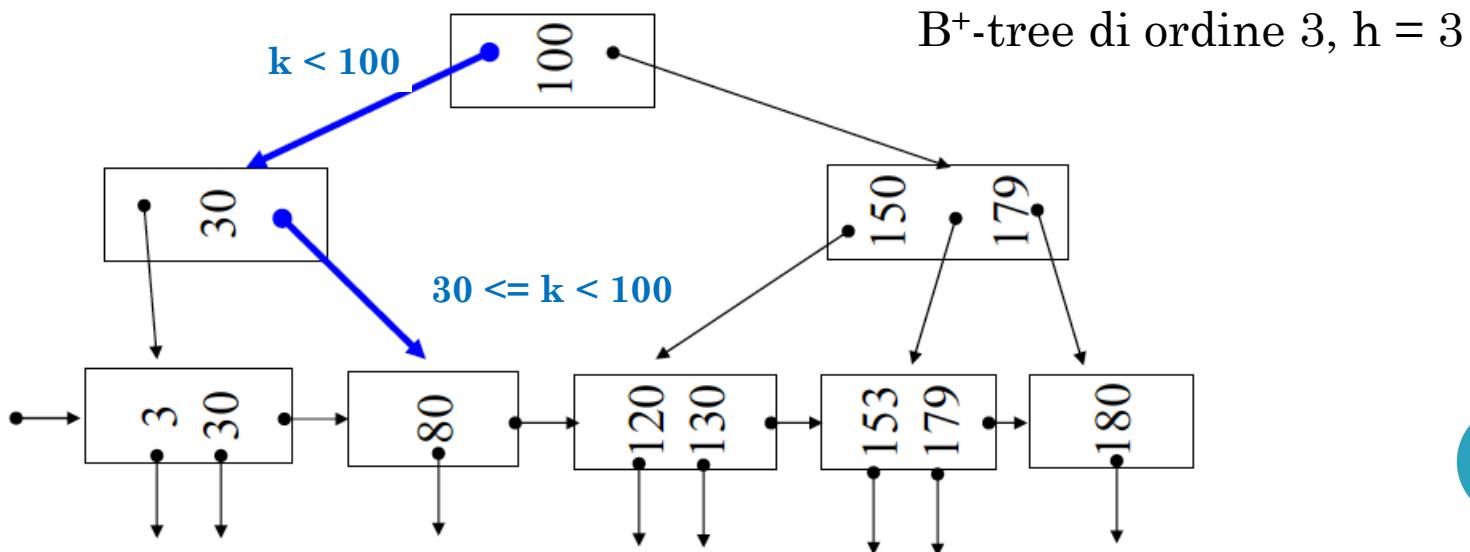
- Esistono diverse strutture ad albero, la struttura più utilizzata dai DBMS commerciali è il **B⁺-tree**
- Ogni **nodo** corrisponde ad un **blocco**
- Le operazioni di ricerca, inserimento e cancellazione hanno **costo**, nel caso peggiore
 - **lineare nell'altezza dell'albero h**
 - logaritmico nel numero di valori distinti N della chiave di ricerca, in quanto si può provare che h è in $O(\log N)$
- il **numero massimo m-1 di elementi memorizzabili** in un nodo è l'unico parametro dipendente dalle caratteristiche della memoria, cioè dalla dimensione del blocco
- garantiscono un'occupazione di memoria almeno del 50% (almeno metà di ogni blocco allocato è effettivamente occupato)

B⁺-TREE – CARATTERISTICHE GENERALI

- Le coppie (k_i, r_i) sono tutte contenute in nodi (quindi blocchi) foglia e le foglie sono collegate a lista mediante puntatori (PID) per favorire ricerche di tipo range
- **La ricerca di una chiave deve individuare una foglia**
- I nodi interni memorizzano dei separatori (anch'essi valori della chiave di ricerca k_i) la cui funzione è determinare il giusto cammino nella ricerca di una chiave
 - parziale duplicazione delle chiavi (perché alcune sono presenti in nodi foglia e nodi interni)

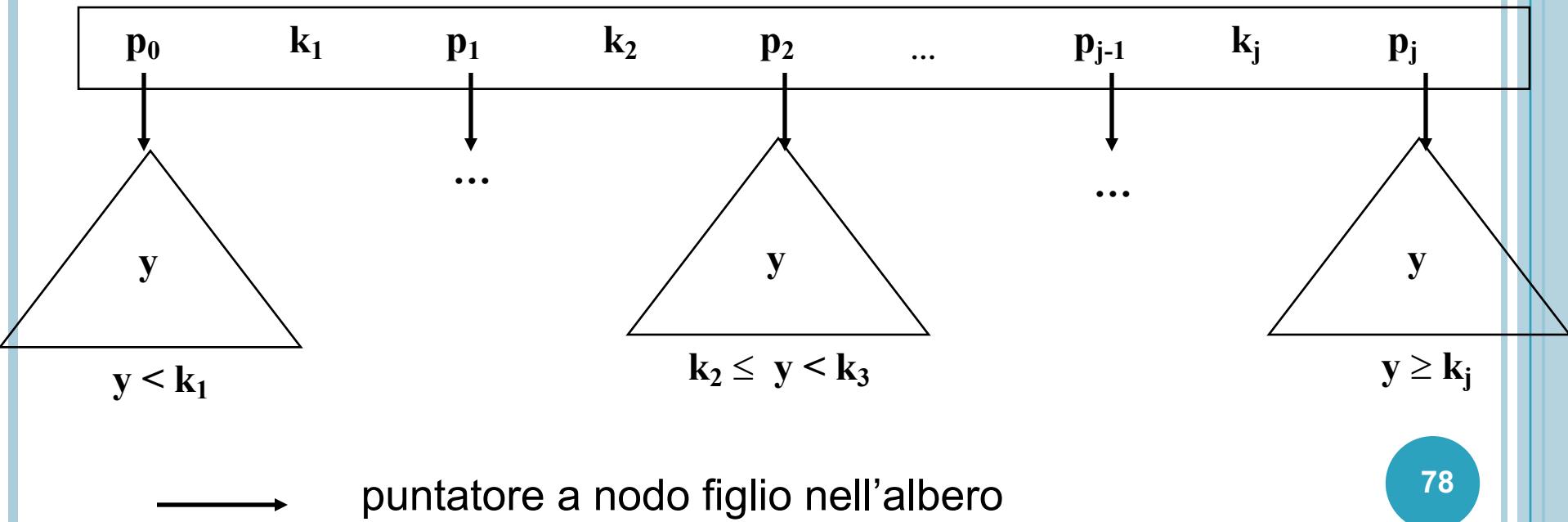
B⁺-TREE - DEFINIZIONE FORMALE

- Un B⁺-albero di ordine m ($m \geq 3$) è un albero bilanciato che soddisfa le seguenti proprietà:
 - ogni nodo contiene al più $m - 1$ elementi
 - ogni nodo, tranne la radice, contiene almeno $\lceil m/2 \rceil - 1$ elementi, la radice può contenere anche un solo elemento
 - ogni nodo non foglia contenente j elementi ha $j + 1$ figli



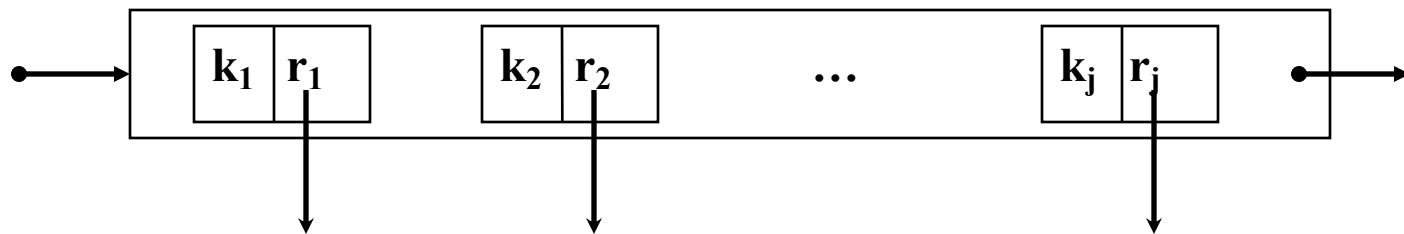
B⁺-TREE – NODI INTERNI

- Valori chiave ordinati



B⁺-TREE – NODI FOGLIA

- Valori chiave ordinati



- puntatore a nodo foglia successivo nell'albero
- puntatore al file dei dati
(organizzazione primaria)

B⁺-TREE

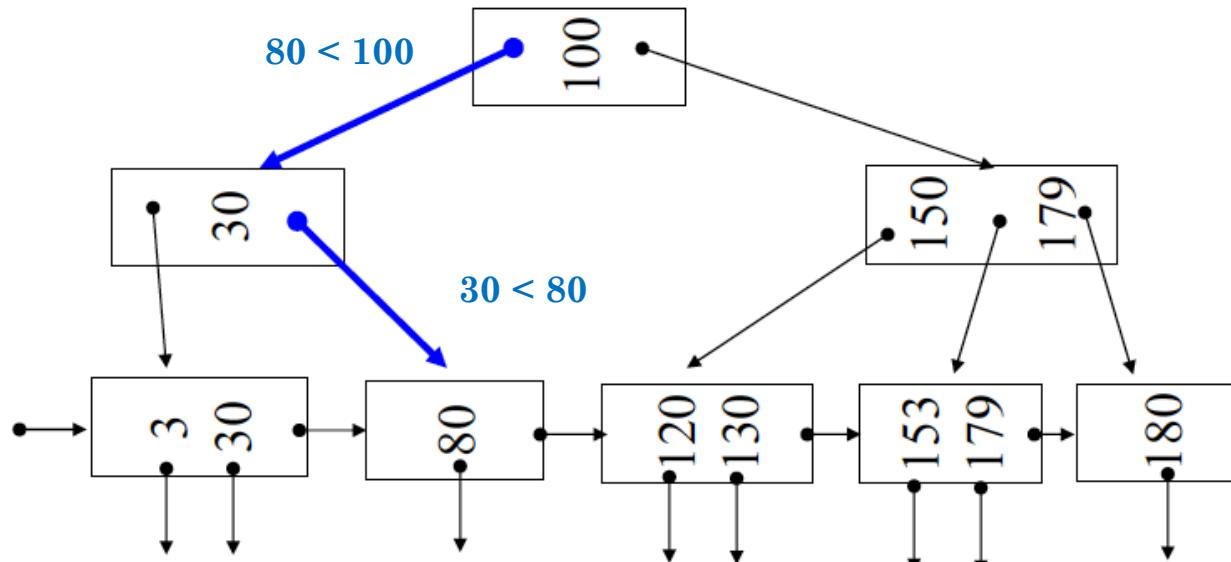
- Il **sottoalbero sinistro** di un separatore contiene valori di chiave **minori** del separatore, quello **destro** valori di chiave **maggiori** od uguali al separatore
- Nel caso di chiavi alfanumeriche facendo uso di separatori di lunghezza ridotta, tramite tecniche di compressione, si risparmia spazio

B⁺-TREE - RICERCA DI UN ELEMENTO

- Si trasferisce la radice in memoria e si esegue la ricerca tra le chiavi contenute per determinare se scendere nel sottoalbero sinistro o destro
- Una volta raggiunta una foglia, o la chiave cercata è presente in tale foglia o non è presente nell'albero
- Il costo della ricerca di una chiave nell'indice è il numero di nodi letti, sempre pari all'altezza dell'albero h
- Due casi
 - Ricerca per uguaglianza
 - Ricerca per intervallo (range)

B⁺-TREE – RICERCA PER UGUAGLIANZA

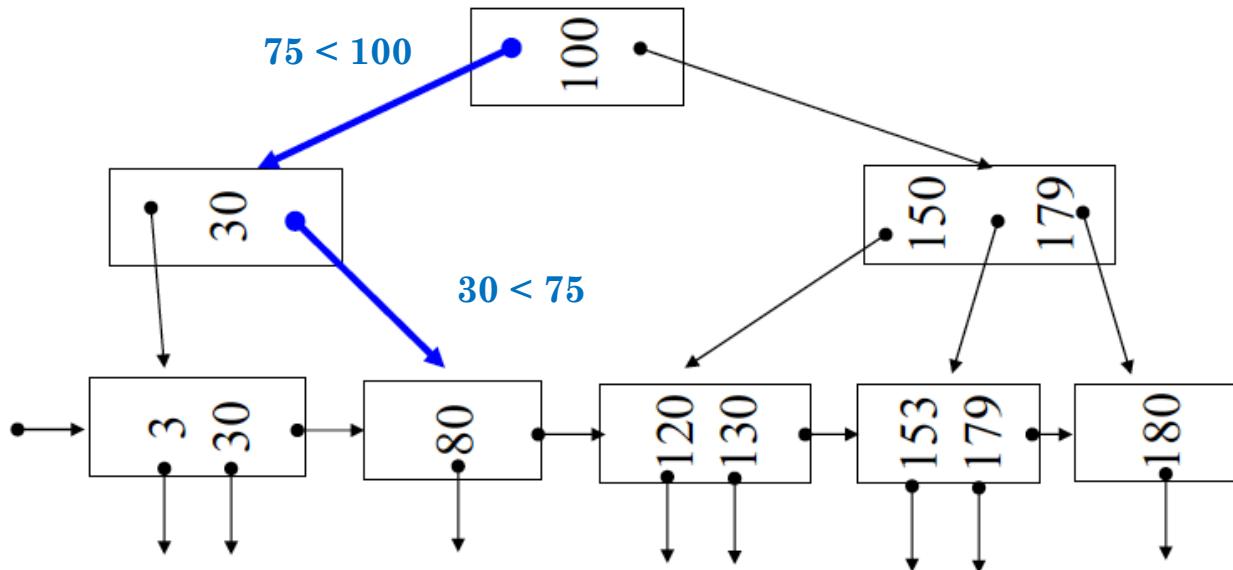
- R(K,B,C)
- Chiave di ricerca per indice primario: **K**
- Ricerca tuple con **K = 80**



Trovato, si naviga il puntatore al file dei dati e
si recupera la tupla
(una sola, indice primario, un solo puntatore)

B⁺-TREE – RICERCA PER UGUAGLIANZA

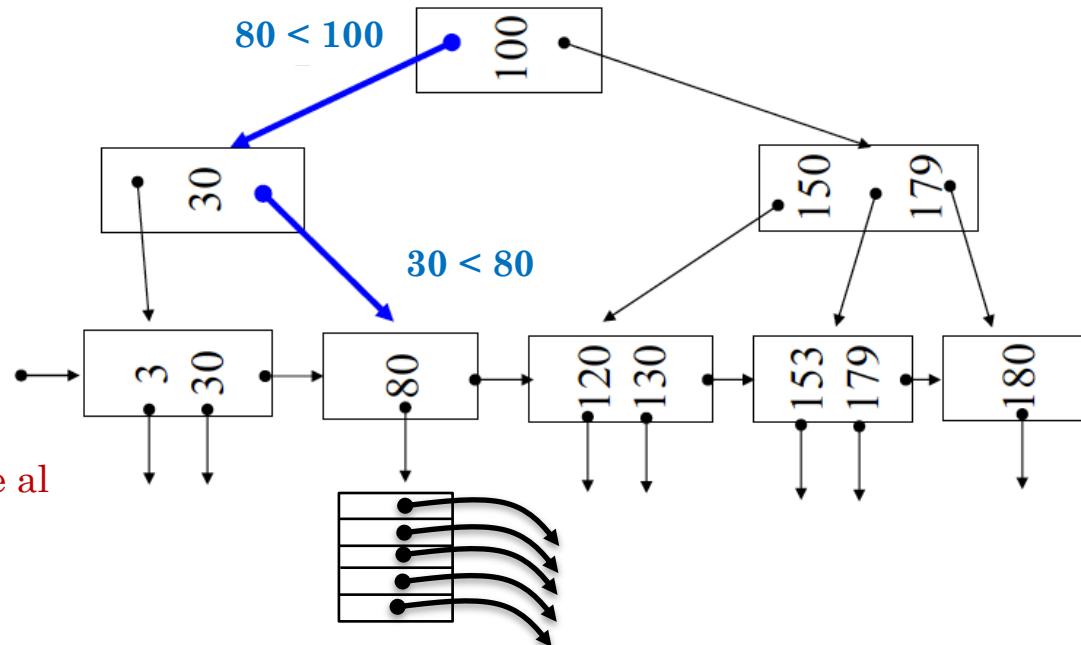
- R(\underline{K} ,B,C)
- Chiave di ricerca per indice **primario**: $\textcolor{green}{K}$
- Ricerca tuple con $\textcolor{blue}{K} = 75$



Non trovato: il valore non esiste nell'indice,
quindi non esistono tuple che soddisfano la condizione
non si accede il file dei dati

B⁺-TREE – RICERCA PER UGUAGLIANZA

- R(K, B, C)
- Chiave di ricerca per indice **secondario**: K
- Ricerca tuple con **K = 80**



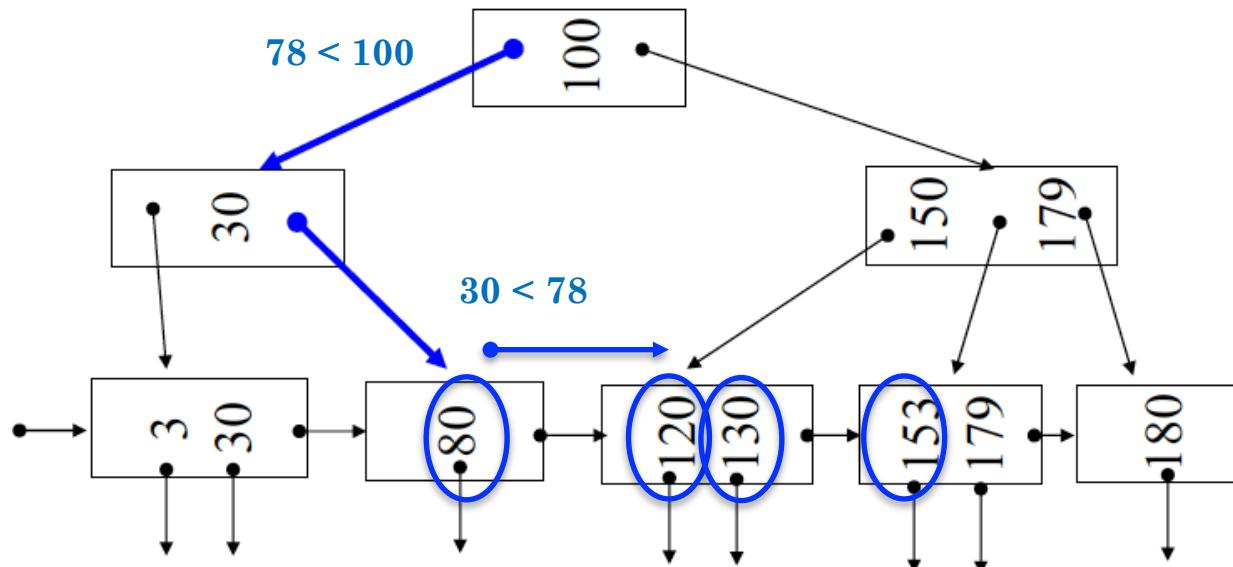
Trovato, si naviga il puntatore al
file dei dati e
si recuperano le **tuple**
**(anche più di una, indice
secondario,
numero arbitrario di puntatori,
inseriti in un blocco puntato dal
puntatore nell'indice)**

B⁺-TREE - RICERCA PER INTERVALLO

- Si esegue la ricerca dell'estremo sinistro dell'intervallo
- Una volta raggiunta una foglia, ci si sposta nella lista dei nodi foglia fino a superare l'estremo destro dell'intervallo
- La ricerca per intervallo è quindi molto efficiente

B⁺-TREE – RICERCA PER INTERVALLO

- R(\underline{K} ,B,C)
- Chiave di ricerca per indice primario: K
- Ricerca tuple con $78 \leq K \leq 154$



Si navigano i puntatori di tutti i valori
che soddisfano la condizione

B+-TREE - INSERIMENTO E CANCELLAZIONE

- Inserimento e cancellazione richiedono prima di tutto un'operazione di ricerca
- Possono richiedere ulteriori aggiustamenti dell'albero per mantenere la proprietà di bilanciamento ed i vincoli sul numero minimo e massimo di elementi
- Idea chiave di inserimento e cancellazione:
 - le modifiche partono sempre dalle foglie e l'albero cresce o si accorcia verso l'alto
- Anche il costo di inserimento e cancellazione è lineare in h , cioè nell'altezza dell'albero, quindi nel logaritmo del numero di valori memorizzati

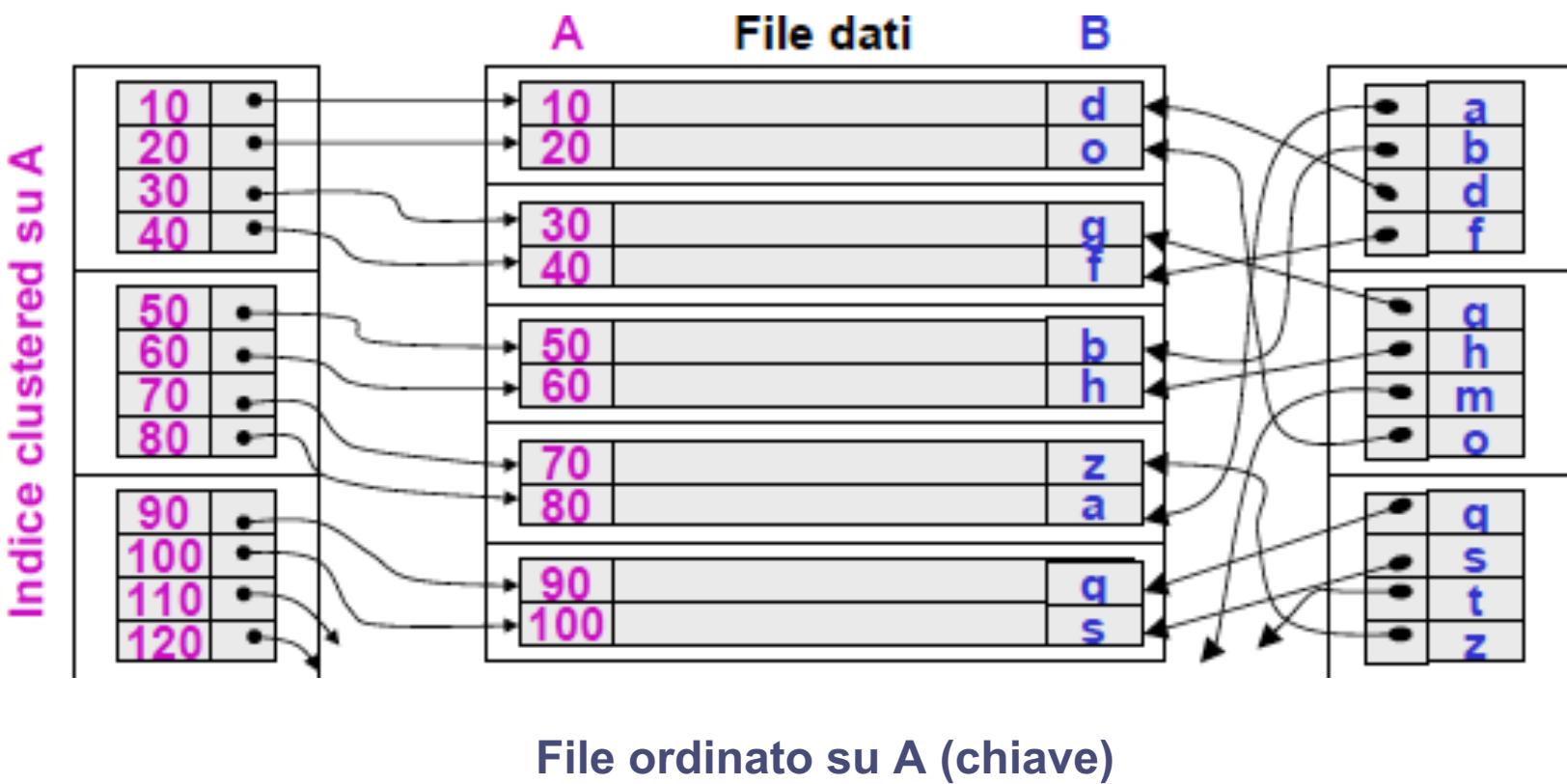
B⁺-TREE - VARIANTI

- **B-tree**
 - Variante in cui le entrate dell'indice sono associate alle chiavi anche nei nodi interni dell'albero, senza duplicazione delle chiavi
 - Migliora occupazione di memoria
 - La ricerca di una singola chiave è più costosa in media in un B⁺-tree (si deve necessariamente raggiungere sempre la foglia per ottenere il puntatore ai dati)
 - B⁺-tree meglio per ricerche di intervalli di valori

INDICI AD ALBERO CLUSTERIZZATI E NON CLUSTERIZZATI

- Un indice ad albero è **clusterizzato** se il file dei dati è ordinato rispetto alla chiave di ricerca, in caso contrario si definisce **non clusterizzato**
- Un file dei dati ordinato è sempre associato a un indice ad albero clusterizzato: si sfrutta ordinamento delle foglie dell'indice per ordinare il file dei dati
 - le operazioni di inserimento, cancellazione e aggiornamento nel file ordinato sono facilitate dalla presenza dell'indice
- Al più un indice clusterizzato per tabella (indipendentemente dal tipo)

ESEMPIO – INDICE CLUSTERIZZATO PRIMARIO



ESEMPIO – INDICE CLUSTERIZZATO SECONDARIO

Basta un solo
puntatore per ogni
valore della chiave di
ricerca

01-Mar-2006	1111	01-Mar-2006	6635	02-Mar-2006
02-Mar-2006	1115	01-Mar-2006	6635	02-Mar-2006
02-Mar-2006	1117	02-Mar-2006	6635	06-Mar-2006
04-Mar-2006	1118	02-Mar-2006	6635	06-Mar-2006
08-Mar-2006	1111	04-Mar-2006	6642	05-Mar-2006
08-Mar-2006	1119	08-Mar-2006	6635	10-Mar-2006
08-Mar-2006	1120	08-Mar-2006	6635	10-Mar-2006
10-Mar-2006	1116	08-Mar-2006	6642	09-Mar-2006
10-Mar-2006	1118	10-Mar-2006	6642	11-Mar-2006
10-Mar-2006	1121	15-Mar-2006	6635	18-Mar-2006
12-Mar-2006	1122	15-Mar-2006	6635	18-Mar-2006
13-Mar-2006	1113	15-Mar-2006	6635	18-Mar-2006
13-Mar-2006	1129	15-Mar-2006	6635	20-Mar-2006
14-Mar-2006	1119	15-Mar-2006	6642	16-Mar-2006
14-Mar-2006	1126	15-Mar-2006	6610	16-Mar-2006
15-Mar-2006	1112	16-Mar-2006	6610	18-Mar-2006
15-Mar-2006	1114	16-Mar-2006	6610	17-Mar-2006
16-Mar-2006	1128	18-Mar-2006	6642	20-Mar-2006
16-Mar-2006	1124	20-Mar-2006	6610	21-Mar-2006
18-Mar-2006	1115	20-Mar-2006	6610	21-Mar-2006
18-Mar-2006	1124	21-Mar-2006	6642	22-Mar-2006
20-Mar-2006	1116	21-Mar-2006	6610	?
20-Mar-2006	1117	21-Mar-2006	6610	?
21-Mar-2006	1127	22-Mar-2006	6635	?
21-Mar-2006	1125	22-Mar-2006	6635	?
21-Mar-2006	1122	22-Mar-2006	6642	?
22-Mar-2006	1113	22-Mar-2006	6642	?

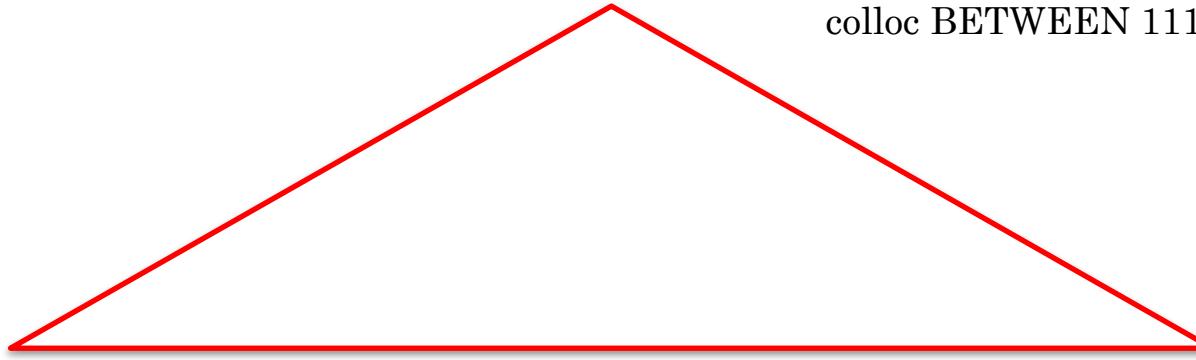
File ordinato su DataNol

INDICI AD ALBERO SU SINGOLO ATTRIBUTO E MULTI-ATTRIBUTO

- Indice su singolo attributo
 - indice la cui chiave di ricerca è costituita da un solo attributo
- Indice multiattributo
 - indice la cui chiave di ricerca è costituita da più di un attributo
- Esempio: per Noleggio:
 - indice su (colloc,dataNol) è indice multiattributo
 - indice su codCli è indice a singolo attributo

ESEMPIO

- Non permette di determinare direttamente le tuple che soddisfano
colloc = 1112 oppure
colloc BETWEEN 1116 AND 1122



(01-Mar-2006, 1111) (01-Mar-2006, 1111)

(02-Mar-2006, 1117) (02-Mar-2006, 1118)

(15-Mar-2006, 1121) (15-Mar-2006, 1122)

...

(15-Mar-2006, 1113) (15-Mar-2006, 1118)

- Permette di determinare direttamente le tuple che soddisfano

dataNol = '15-Mar-2006' oppure

dataNol = '15-Mar-2006' AND colloc = 1122

...

INDICI AD ALBERO SU SINGOLO ATTRIBUTO E MULTI-ATTRIBUTO

- Un indice multiattributo è definito su una lista di attributi
 - diversi ordinamenti degli attributi chiave corrispondono a diversi indici
 - indice su (dataNol,colloc) è diverso da indice su (colloc,dataNol)
- Un indice multi-attributo su A₁, A₂, ..., A_n permette di determinare direttamente
 - le tuple che soddisfano condizioni di uguaglianza su tutti gli attributi A₁,..., A_n o sui primi i < n
 - le tuple che soddisfano condizioni di uguaglianza sui primi i attributi A₁,..., A_i e condizioni di intervallo sugli attributi A_{i+1},..., A_j, j < n

INDICI MULTI-ATTRIBUTO

- Indice multiattributo può supportare più interrogazioni di un indice a singolo attributo
- Deve però essere aggiornato più di frequente ed è di dimensione maggiore rispetto ad indice su singolo attributo

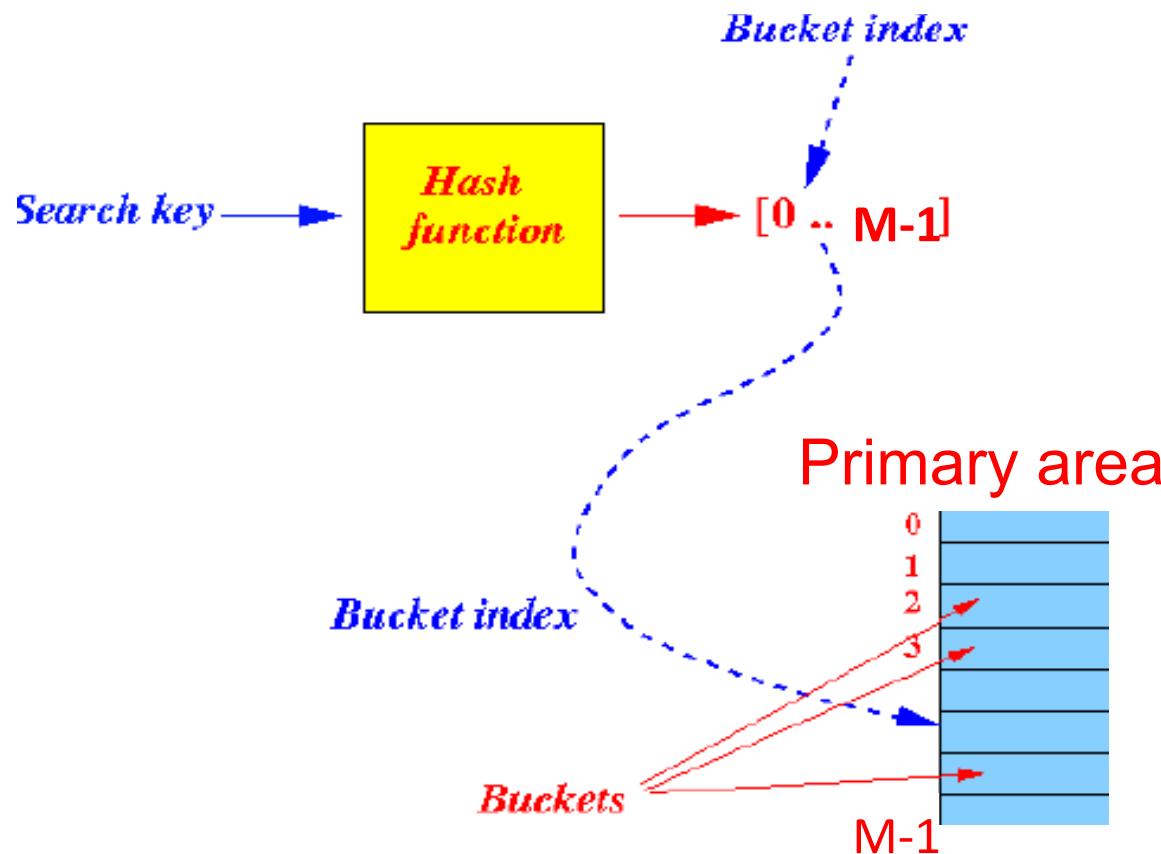
INDICI HASH

INDICI HASH

- L'uso di indici ad albero ha lo svantaggio di richiedere la scansione di una **struttura dati, memorizzata su disco**, per localizzare i dati
- Questo perché le associazioni (k_i, r_i) vengono mantenute in forma **esplicita**, come record in un file
- Gli indici **hash** al contrario mantengono le associazioni (k_i, r_i) in modo **implicito**, tramite l'uso di una **funzione hash**, definita sul dominio della chiave di ricerca

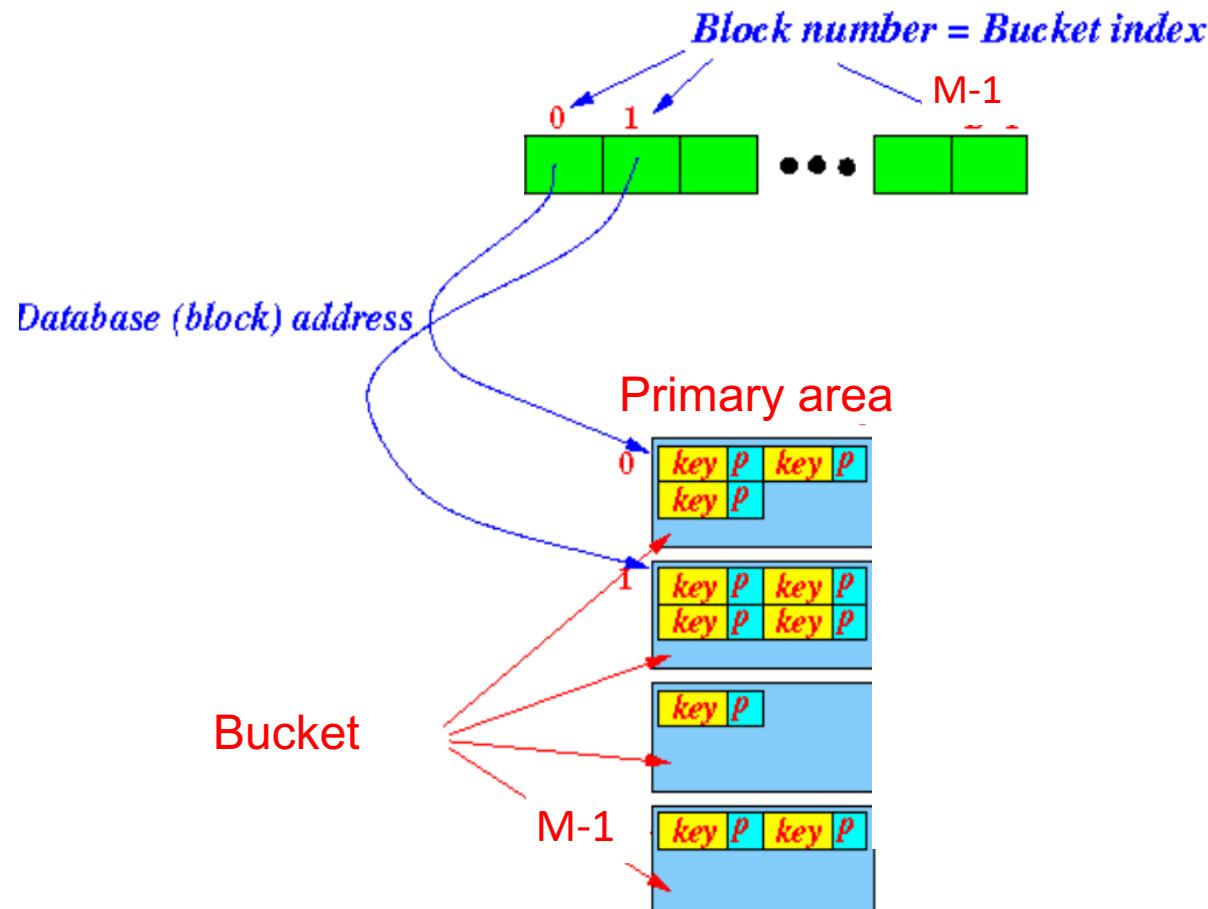
INDICI HASH – CARATTERISTICHE GENERALI

- Ad ogni valore della funzione hash corrisponde un indirizzo in area primaria



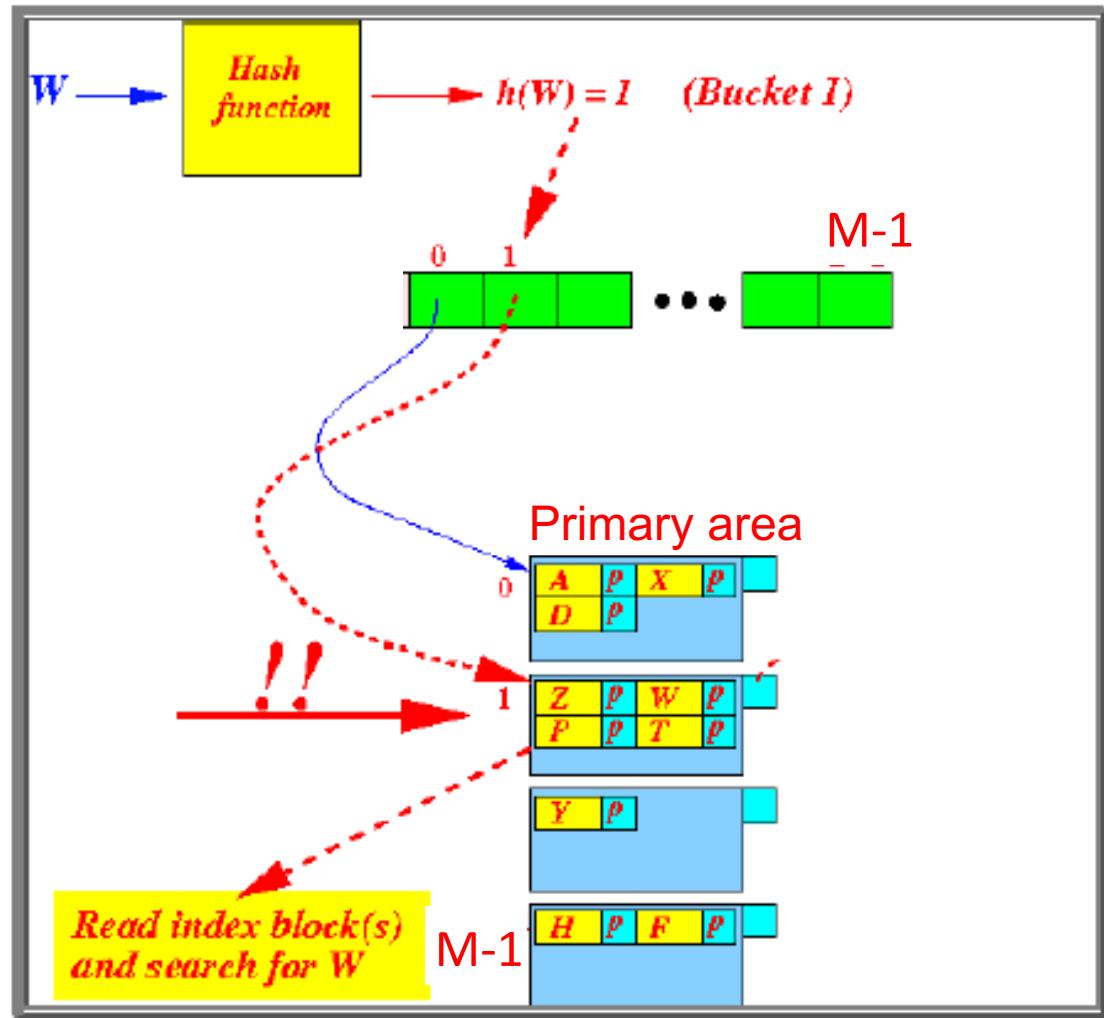
INDICI HASH – CARATTERISTICHE GENERALI

- L'associazione tra i valori della funzione Hash e gli indirizzi su disco avviene tramite un **bucket index**



INDICI HASH - RICERCA PER UGUAGLIANZA

1. Supponiamo di cercare le tuple della relazione $R(K, B, C)$ con $B = w$, con indice hash su attributo B
2. Si calcola $H(w)$
3. Se $H(w) = 1$, tramite il bucket index, si determina l'indirizzo corrispondente al bucket (supponiamo 1)
4. Si accede il bucket e, record per record, si cercano le tuple t con $t.B = w$



INDICI HASH – RICERCA PER INTERVALLO

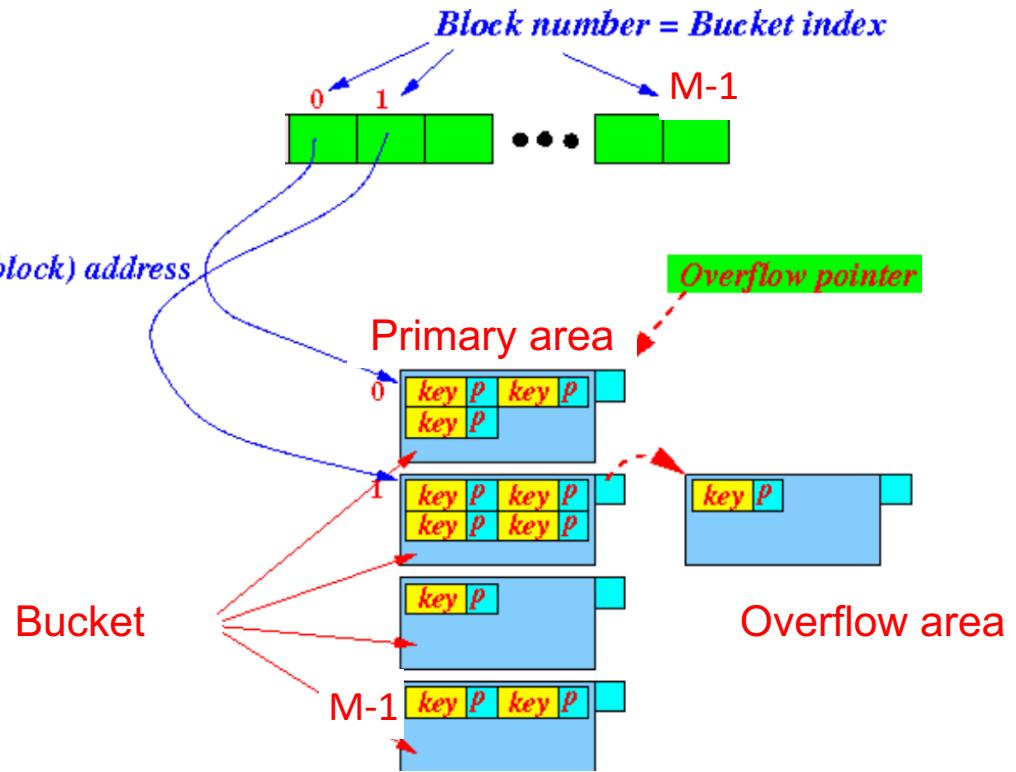
- Un indice hash non supporta ricerche per intervallo
- La funzione hash infatti non mantiene l'ordine
 - Tuple con valori contigui per la chiave di ricerca possono essere memorizzati in bucket diversi

INDICI HASH – INSERIMENTO E TRABOCCHI

- Per inserire un record, si procede come per la ricerca per uguaglianza, identificando il bucket in cui il record deve essere inserito
- Il numero di record che possono essere allocati nello stesso bucket determina la **capacità c** dei bucket
- I bucket hanno in genere la stessa capacità
- Se il bucket individuato per l'inserimento ha ancora **spazio** (la sua capacità è inferiore a c), si inserisce il record nel bucket
- Se il bucket non ha più spazio (la sua capacità è pari a c), si genera un **trabocco (overflow)**
- La presenza di overflow può richiedere l'uso di un'area di memoria separata, detta **area di overflow**

INDICI HASH - TRABOCCHI

- Supponiamo che ogni bucket abbia capacità $c = 4$ (può contenere 4 record)
- Supponiamo adesso di inserire un nuovo record con chiave k tale che $H(k) = 1$
- Il bucket 1 è già pieno, quindi è necessario usare un blocco in area di overflow per memorizzare il nuovo record



INDICI HASH – AREA PRIMARIA E AREA DI OVERFLOW

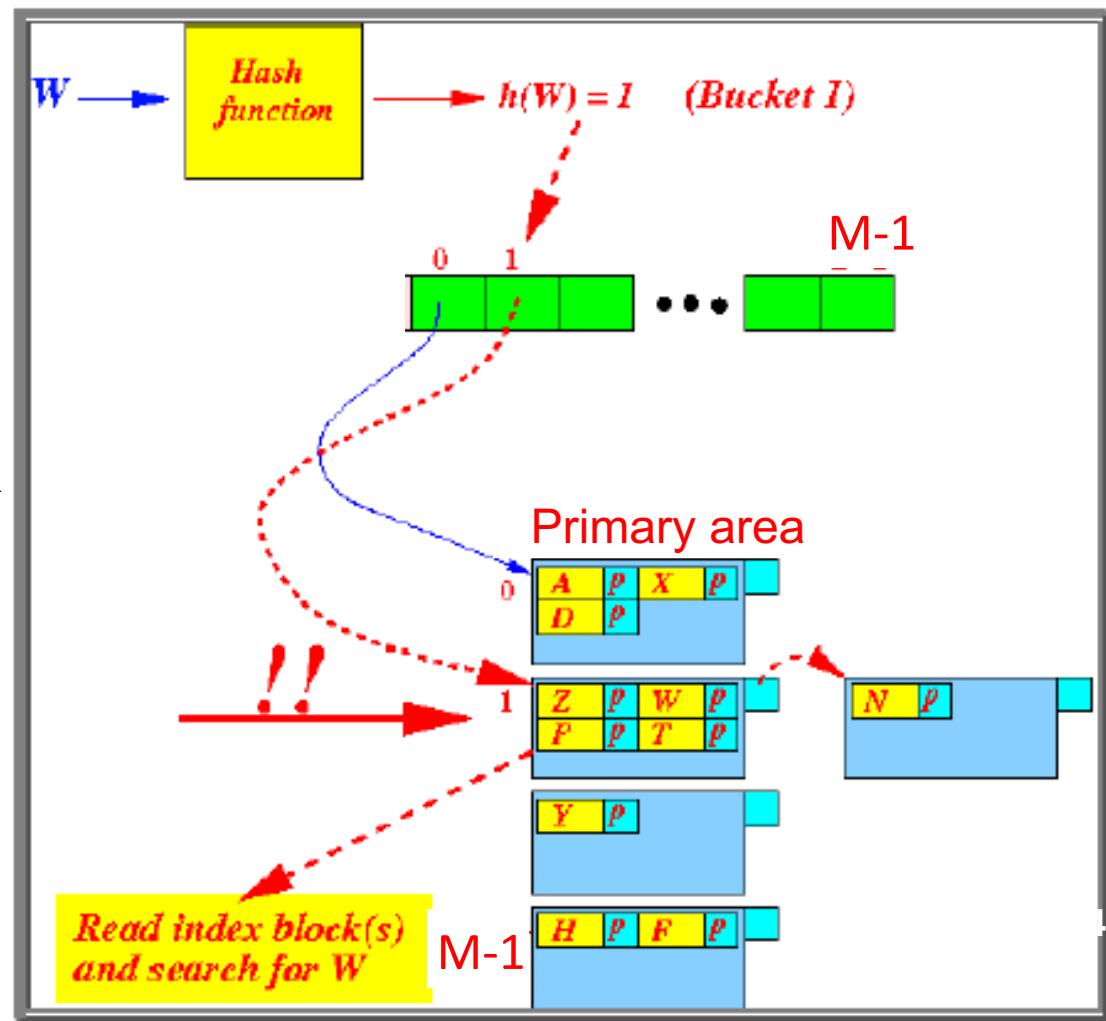
- L'area primaria viene allocata al momento della creazione dell'indice hash, l'area di overflow successivamente, quando ce n'è bisogno
- Il sistema ottimizza l'allocazione dell'area primaria: se un bucket contiene più blocchi in area primaria, questi sono memorizzati in modo possibilmente contiguo su disco (e certamente sullo stesso cilindro)
 - Tempo di latenza minimizzato per accesso ad area primaria
- Il sistema non è in grado di ottimizzare l'allocazione dell'area di overflow: i blocchi dell'area di overflow vengono memorizzati dove si può
 - Maggiore tempo di latenza per accesso ad area di overflow

INDICI HASH PERFETTA

- Una funzione hash è detta **perfetta** se per un certo numero di record non produce trabocchi
- Una funzione perfetta può sempre essere definita disponendo di un'area primaria con capacità complessiva pari al numero dei record da memorizzare

INDICI HASH – DI NUOVO RICERCA PER UGUAGLIANZA

1. Supponiamo di cercare le tuple della relazione $R(K, B, C)$ con $B = w$, con indice hash su attributo B
2. Si calcola $H(w)$
3. Se $H(w) = 1$, tramite il bucket index, si determina l'indirizzo corrispondente al bucket (supponiamo 1)
4. Si accede il bucket e, record per record, si cercano le tuple t con $t.B = w$
5. Si deve analizzare sia la parte del bucket in area primaria sia quella in area di overflow



INDICI HASH - CANCELLAZIONE

- Si cercano i record da cancellare, come descritto per l'operazione di ricerca per uguaglianza
- Si elimina il record dal bucket
- La cancellazione di un record da un'area di overflow può determinare la cancellazione dell'area di overflow (se è l'ultimo record nell'area)

INDICI HASH – COSTI

- Un indice hash supporta in modo efficiente ricerche per uguaglianza, inserimenti e cancellazioni
 - Non è necessario accedere un file come per gli indici ordinati
- In assenza di overflow, il costo di accesso a indice è costante
 - Costo di accesso al bucket index se non è già in memoria (1)
 - Costo di accesso al bucket (1 se corrisponde a un singolo blocco)
 - Costo di scrittura, per inserimento e cancellazione (1)
- In presenza di overflow, le prestazioni non sono facilmente determinabili
 - Quanti blocchi contiene l'area di overflow per il bucket acceduto?
 - Dove sono memorizzati?

INDICI HASH – COME CREARLI

- La creazione di un indice hash richiede di specificare:
 - la funzione H per la trasformazione della chiave
 - il metodo per la gestione dei trabocchi
 - il fattore di caricamento d
 - valore tra 0 e 1 che indica quanto si intendono mantenere “pieni” i bucket
- Il DBA ha al più la possibilità di agire sulla funzione, ma non in tutti i DBMS si può specificare (in PostgreSQL non si può)

INDICI HASH – LA FUNZIONE

- Una **funzione hash** è un'applicazione surgettiva H dall'insieme delle possibili chiavi all'insieme $0, \dots, M - 1$ dei possibili indirizzi che verifichi le seguenti proprietà:
 - **distribuzione uniforme** delle chiavi nello spazio degli indirizzi (ogni indirizzo deve essere generato con la stessa probabilità)
 - **distribuzione casuale** delle chiavi (eventuali correlazioni tra i valori delle chiavi non devono tradursi in correlazioni tra gli indirizzi generati)
- tali proprietà dipendono dall'insieme delle chiavi su cui si opera e quindi non esiste una funzione universale ottima

INDICI HASH – LA FUNZIONE

- In genere le funzioni hash operano su **insiemi di chiavi intere**
- Se i valori delle chiavi sono stringhe alfanumeriche, si può associare in modo univoco ad ogni chiave un numero intero, prima di applicare la trasformazione

INDICI HASH – LA FUNZIONE

- **Metodo della divisione**
- la chiave numerica viene divisa per M e l'indirizzo è ottenuto considerando il resto:

$$H(k) = k \bmod M$$

dove mod indica il resto della divisione intera

- Affinché H distribuisca bene, M deve essere primo oppure non primo con nessun fattore primo minore di 20
- Test sperimentali eseguiti con file con caratteristiche molto diversificate mostrano che, in generale, il metodo della divisione è il più adattabile ed è quello più utilizzato dai DBMS (incluso PostgreSQL)

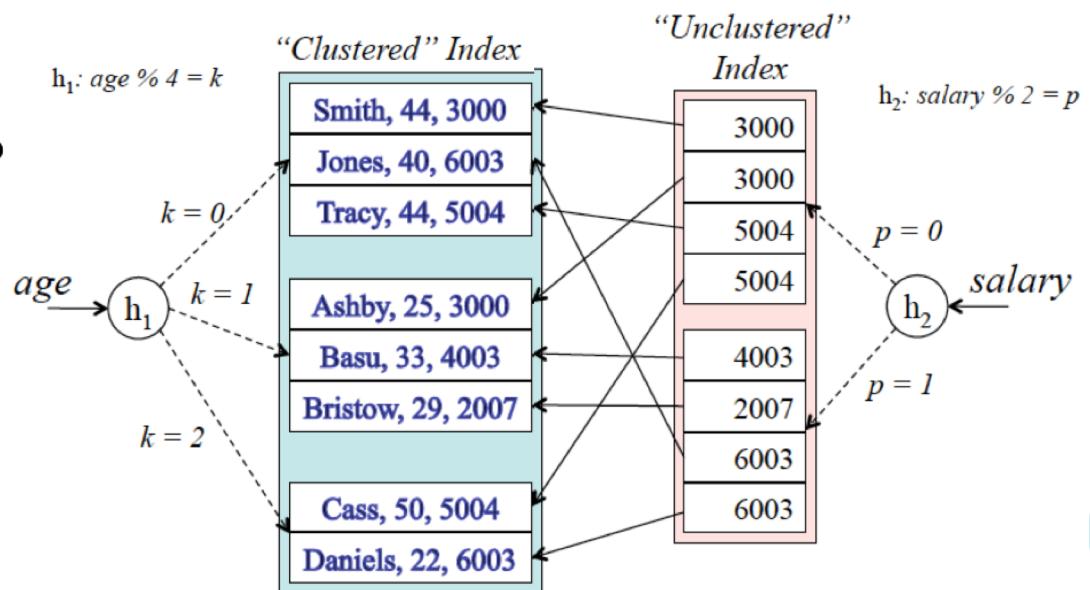
INDICI HASH CLUSTERIZZATI E NON CLUSTERIZZATI

- Un indice hash è **clusterizzato** se i record che condividono lo stesso valore per la chiave di ricerca sono memorizzati in posizioni adiacenti nel file dei dati
- In caso contrario, l'indice è **non clusterizzato**
- Un file dei dati di tipo hash è sempre associato a un indice hash clusterizzato:
 - le operazioni di inserimento, cancellazione e aggiornamento nel file ordinato sono facilitate dall'uso dell'organizzazione hash
- In presenza di un indice hash clusterizzato (e quindi in presenza di un file organizzato a hash) l'**organizzazione primaria** corrisponde ai **record memorizzati nell'area primaria + i record memorizzati nell'area di overflow**
- Al più un indice clusterizzato per tabella (indipendentemente dal tipo)

INDICI HASH CLUSTERIZZATI E NON CLUSTERIZZATI

- Employee(Name, **age**, **salary**)
- Indice hash rispetto a **age clusterizzato**
- struttura discussa finora
- tutti i record che corrispondono allo stesso valore hash sono memorizzati vicini, cioè nello stesso bucket
- tutti i record con lo stesso valore per la chiave di ricerca sono memorizzati vicini, cioè nello stesso bucket

- Indice hash rispetto a **salary non clusterizzato**
- I record con lo stesso valore per la chiave di ricerca o, più in generale, che corrispondono allo stesso valore hash, possono essere memorizzati in bucket diversi
- Serve un livello in più → **indice multilivello**



INDICI HASH SU SINGOLO ATTRIBUTO E MULTI-ATTRIBUTO

- Un indice hash multiattributo è definito su una lista di attributi
 - diversi ordinamenti degli attributi chiave potrebbero corrispondere a diverse organizzazioni (**dipende dalla funzione hash**)
 - indice su (dataNol,colloc) potrebbe essere diverso da indice su (colloc,dataNol)
- Un indice multi-attributo su A₁, A₂, ..., A_n permette di determinare direttamente le tuple che soddisfano condizioni di uguaglianza su tutti gli attributi A₁,...,A_n o sui primi i < n
-

CONFRONTO TRA INDICI AD ALBERO E HASH

- Per selezioni con **condizioni di uguaglianza** del tipo

```
SELECT A1, A2, ..... An  
FROM R  
WHERE Ai=C
```

un indice hash su A_i è preferibile

- Per selezioni con **condizioni di tipo range** del tipo

```
SELECT A1, A2, ..... An  
FROM R  
WHERE C1 < Ai < C2
```

un indice ad albero su A_i è preferibile

- Gli indici hash infatti non mantengono l'ordine

- Infatti:
 - la scansione di un indice ha un costo proporzionale al logaritmo del numero di valori in R per A_i
 - in una struttura hash il tempo di ricerca è indipendente dalla dimensione della base di dati

INDICI IN POSTGRESQL

INDICI IN SQL

- Lo standard SQL fornisce primitive che permettono al progettista della base di dati di definire le strutture ausiliarie di accesso e, conseguentemente, l'organizzazione dei record nei file
- I DBMS relazionali non sempre sono aderenti allo standard, ma la sintassi non differisce molto nei vari sistemi per quanto riguarda i comandi principali
 - Vedremo sintassi PostgreSQL
- In SQL la definizione di indici avviene mediante lo statement CREATE INDEX
- **Se non si specifica altrimenti, viene creato un indice ordinato**

INDICI IN SQL - POSTGRESQL

- `CREATE INDEX nome_index
ON nome_tabella(nomi_attributi)
[DESC| ASC];`
- `CREATE INDEX nome_index
ON nome_tabella
USING HASH (nome_attributo);`
- `DROP INDEX nome_index;`
- Crea un indice **ordinato** (in modo ascendente o descendente), **non clusterizzato**, **denso**, di nome `nome_index` sugli attributi `nomi_attributi` della tabella `nome_tabella`
- L'indice è **multi-attributo** se compare più di un attributo in `nomi_attributi`
- Crea un indice **hash**, **non clusterizzato**, **denso**, di nome `nome_index` sugli attributi `nomi_attributi` della tabella `nome_tabella`
- L'indice è sempre **singolo-attributo**
- Cancella indice `nome_index`

INDICI IN SQL - POSTGRESQL

- ```
CREATE UNIQUE INDEX nome_index
ON nome_tabella(nome_attributi)
[DESC| ASC];
```
- Crea un indice **ordinato** (in modo ascendente o discendente) **primario**, **non clusterizzato**, **denso**, di nome `nome_index` sugli attributi `nome_attributi` della tabella `nome_tabella`
- Non sono quindi ammessi duplicati tra i valori della chiave dell'indice
- Anche se non si specifica **UNIQUE**, l'indice è creato come primario primario se `nome_attributi` rappresentano una chiave (primaria o alternativa) per `nome_tabella`
- **Non applicabile a indici hash**

# INDICI SQL - POSTGRESQL

- `CLUSTER nome_tabella USING nome_indice;`
  - Rende l'indice di nome `nome_indice clusterizzato`, clusterizzando il file della tabella corrispondente rispetto alla chiave di ricerca dell'indice
- `CLUSTER nome_tabella;`
  - Ri-clusterizza una tabella (si considera implicitamente l'indice utilizzato durante la prima clusterizzazione)
  - Se la tabella viene aggiornata, la clusterizzazione non viene mantenuta
  - E' quindi necessario invocare periodicamente il comando per mantenere la clusterizzazione
- `CLUSTER;`
  - Ri-clusterizza tutte le tavelle, clusterizzate in precedenza

# ESEMPI

- Indice ordinato secondario sulla colonna codCli della relazione Noleggio:

```
CREATE INDEX idxCli ON Noleggio (codCli);
```

```
CREATE INDEX idxCli ON Noleggio (codCli)
DESC;
```

- Indice hash secondario sulla colonna codCli della relazione Noleggio:

```
CREATE INDEX idxCliH ON Noleggio
```

```
USIN HASH(codCli);
```

- Cancellazione indice idxCli

```
DROP INDEX idxCli;
```

# ESEMPI

- Indice ordinato primario sulla colonna colloc della relazione Video

```
CREATE INDEX idxColloc ON Video (colloc);
```

```
CREATE UNIQUE INDEX idxCollocU ON Video
(colloc);
```

- Clusterizzazione indice idxColloc su Video

```
CLUSTER Video USING idxColloc;
```

- Clusterizzazione indici idxCli e idxCliH su Noleggi

```
CLUSTER Noleggi USING idxCli;
```

```
CLUSTER Noleggi USING idxCliH;
```