

Strutture Hash

Svantaggi:

Trade-off:

- Se il numero di blocchi è troppo piccolo rispetto al Database si hanno frequenti collisioni (con catene di overflow, etc).
 - Se il numero di blocchi è troppo grande rispetto al Database si ha un fattore di riempimento dei blocchi molto basso.
-
- Struttura Hash non è efficiente per le query su un range di valori:
SELECT *
FROM STUDENTI
WHERE (MATRICOLA>10000) AND (MATRICOLA<20000)
 - Struttura Hash non è efficiente per le operazioni che coinvolgono attributi che non sono chiave.

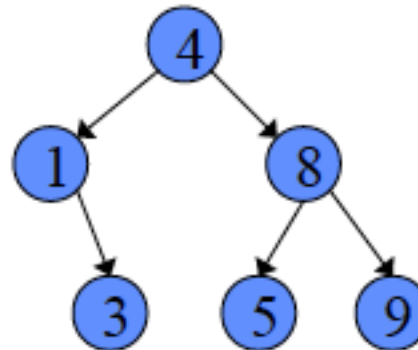
ORGANIZZAZIONE PER CHIAVE: METODO TABELLARE

- Il metodo procedurale (hash) è utile per ricerche per chiave ma non per intervallo. Per entrambi i tipi di ricerche è utile invece il **metodo tabellare**:
 - si usa un **indice**, ovvero di un **insieme ordinato** di coppie $(k, r(k))$, dove k è un valore della chiave ed $r(k)$ è un riferimento al record con chiave k .
- L'indice è gestito di solito con un'opportuna struttura albero detta **B⁺-albero**, la struttura più usata e ottimizzata dai DBMS.
- Gli indici possono essere multi-attributi.
- Indice: struttura che contiene **informazioni sulla posizione di memorizzazione delle tuple** sulla base del valore del campo **chiave**.
- La **realizzazione di indici** avviene tipicamente attraverso l'utilizzo di strutture ad **albero multi-livello**.

Albero binario di ricerca (ripasso)

Albero binario etichettato, in cui per ogni nodo,

- il sottoalbero sinistro contiene solo etichette minori di quella del nodo
- il sottoalbero destro etichette maggiori

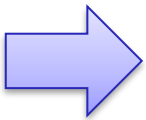
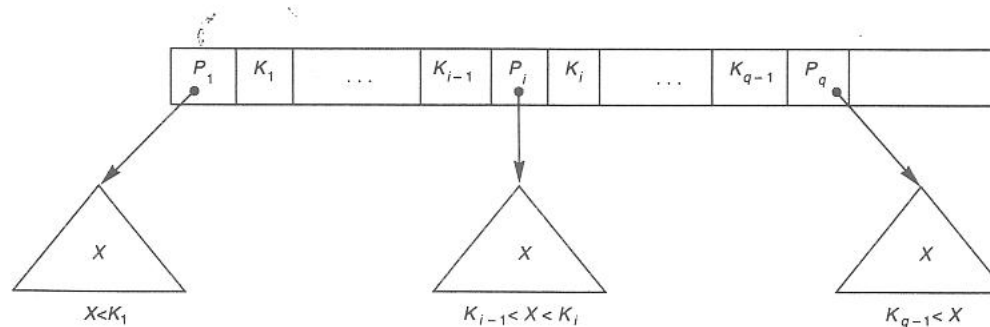


Tempo di ricerca (e inserimento), pari alla profondit.:

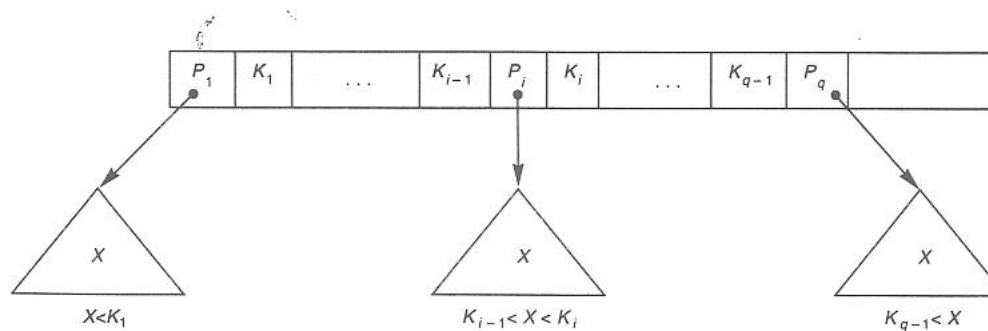
- logaritmico nel caso "medio" (assumendo un ordine di inserimento casuale)

Albero di ricerca di ordine P - Parte I (ripasso)

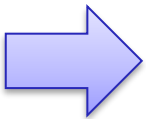
- Ogni nodo ha (fino a) P figli e (fino a) P-1 etichette, ordinate
- Un albero di ricerca di ordine p è un albero i cui nodi contengono al più p-1 search value e p puntatori nel seguente ordine
$$\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle \quad \text{con } q \leq p$$
- Ogni P_i è un puntatore ad un nodo figlio (o un puntatore nullo) e ogni K_i è un search value appartenente ad un insieme totalmente ordinato.



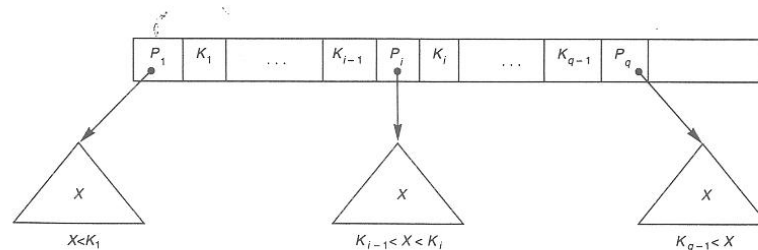
Albero di ricerca di ordine P - Parte II (ripasso)



- Ogni albero di ricerca deve soddisfare due vincoli fondamentali:
 - in ogni nodo $K_1 < K_2 < \dots < K_{q-1}$;
 - per tutti i valori di X presenti nel sottoalbero puntato da P_i , vale la seguente relazione:
 - $K_{i-1} < X < K_i$ per $1 < i < q$;
 - $X < K_i$ per $i = 1$;
 - $K_{i-1} < X$ per $i = q$.

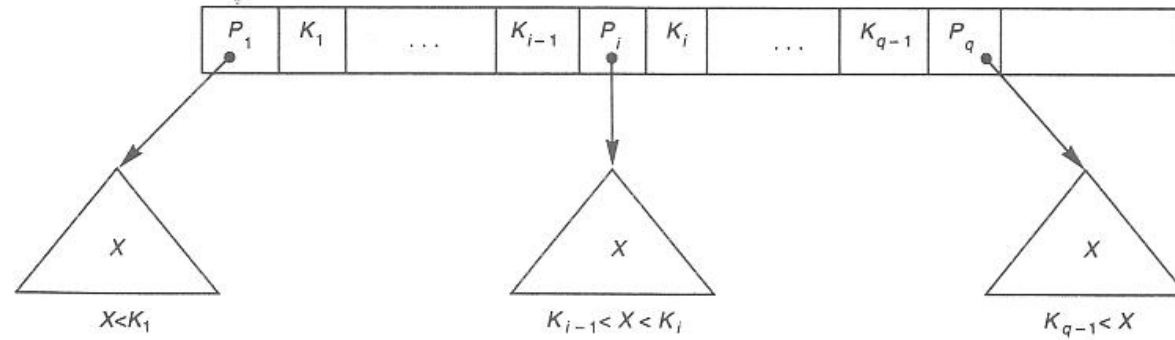


Albero di ricerca di ordine P - Parte III (ripasso)



- Un albero di ricerca può essere utilizzato per cercare record memorizzati su disco. **Ogni ricerca/modifica comporta la visita di un cammino radice foglia**
- I valori di ricerca (search value) possono essere i valori di uno dei campi del file (search field).
- Ad ogni valore di ricerca è associato un puntatore al record avente quel valore (oppure al blocco contenente quel record) nel file di dati.
- L'albero stesso può essere memorizzato su disco, assegnando ad ogni nodo dell'albero una **pagina**. Quando un nuovo record deve essere inserito, l'albero di ricerca deve essere aggiornato includendo il valore del campo di ricerca del nuovo record, col relativo puntatore al record (o alla pagina che lo contiene), nell'albero di ricerca.

Albero di ricerca di ordine P - Parte IV (ripasso)



- Per inserire (risp. cancellare) valori di ricerca nell' (risp. dall') albero di ricerca sono necessari specifici algoritmi che garantiscano il rispetto dei due vincoli fondamentali.
- In generale, tali algoritmi non garantiscono che un albero di ricerca risulti sempre bilanciato (nodi foglia tutti allo stesso livello).
- Soluzione: B-alberi e B+ -alberi.

B-tree - Parte I

- un **B-tree di ordine p**, se usato come struttura di accesso su un campo chiave per la ricerca di record in un file di dati, deve soddisfare le seguenti condizioni:
 - ogni nodo interno del B-albero ha la forma:
$$\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle \text{ con } q \leq p$$
 1. P_i è un **tree pointer** (puntatore ad un altro nodo del B-albero),
 2. K_i è la chiave di ricerca
 3. Pr_i è un **data pointer** (puntatore ad un record il cui campo **chiave di ricerca** è uguale a K_i o alla pagina che contiene tale record);
 - per ogni nodo, si ha che:
$$K_1 < K_2 < \dots < K_{q-1} ;$$
 - ogni nodo ha al più p tree pointer;



B-tree - Parte II

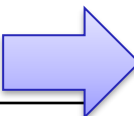
- per tutti i valori X della chiave di ricerca appartenenti al sottoalbero puntato da P_i , si ha che:

$$K_{i-1} < X < K_i \quad \text{per } 1 < i < q;$$

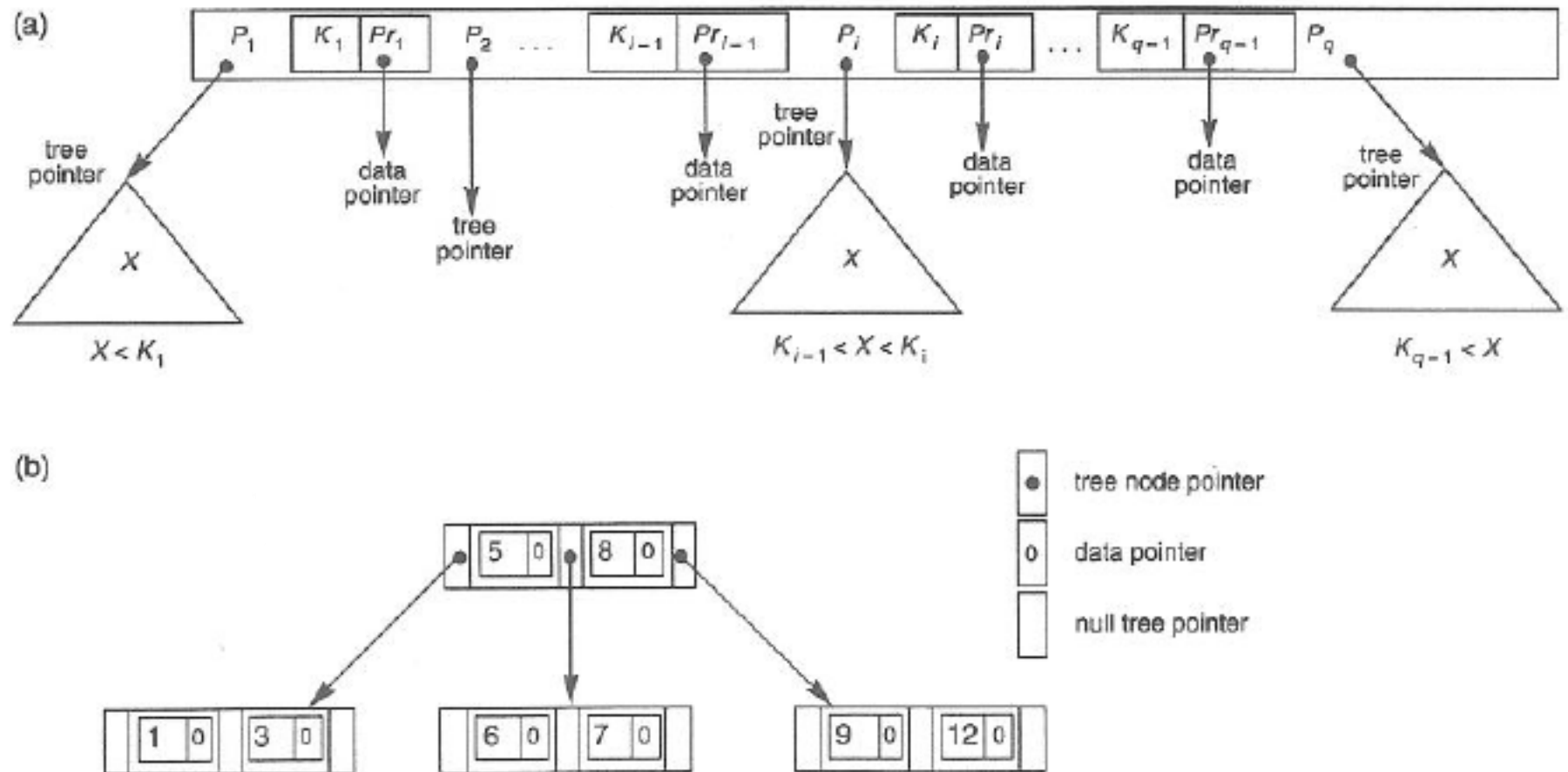
$$X < K_i \quad \text{per } i = 1;$$

$$K_{i-1} < X \quad \text{per } i = q;$$

- la **radice** ha almeno due tree pointer, a meno che non sia l'unico nodo dell'albero;
- ogni **nodo**, esclusa la radice, ha almeno $\lceil p/2 \rceil$ tree pointer;
 - un nodo con q tree pointer, $q \leq p$, ha $q-1$ campi chiave di ricerca (e $q-1$ data pointer);
- **tutti i nodi foglia sono posti allo stesso livello** (i nodi foglia hanno la stessa struttura dei nodi interni, ad eccezione del fatto che tutti i loro tree pointer P_i sono nulli)



B-alberi: struttura



B⁺-tree

- Un **B⁺-tree** è un B-albero in cui i **data pointer sono memorizzati solo nei nodi foglia dell'albero**. La struttura dei nodi foglia differisce dal B-tree quindi da quella dei nodi interni.
- **Se il campo di ricerca è un campo chiave**, i nodi foglia hanno per ogni valore del campo di ricerca una entry e un puntatore ad un record.
- **Se un campo di ricerca non è un campo chiave**, i puntatori indirizzano un blocco che contiene i puntatori ai record del file di dati, rendendo così necessario un passo aggiuntivo per l'accesso ai dati.
- **I nodi foglia di un B⁺-tree sono generalmente messi fra loro in relazione** (ciò viene sfruttato nel caso di range query). Essi corrispondono al primo livello di un indice. I nodi interni corrispondono agli altri livelli di un indice.
- Alcuni valori del campo di ricerca presenti nei nodi foglia sono ripetuti nei nodi interni per guidare la ricerca.



B⁺-tree - Parte I

La struttura dei **nodi interni** (di ordine p) di un B⁺-tree è la seguente:

(1) ogni nodo interno del B⁺-tree ha la forma:

$$\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle \text{ con } q \leq p$$

ogni **P_i** è un **tree pointer** (puntatore ad un altro nodo del B⁺-tree)

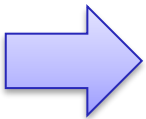
(2) per ogni nodo interno, si ha che:

$$K_1 < K_2 < \dots < K_{q-1};$$

(3) ogni nodo interno ha al più p tree pointer

(4) per tutti i valori X della search key nel sottoalbero puntato da P_i, si ha che

$X \leq K_i$	per $i = 1$;
$K_{i-1} < X \leq K_i$	per $1 < i < q$;
$K_{i-1} < X$	per $i = q$;

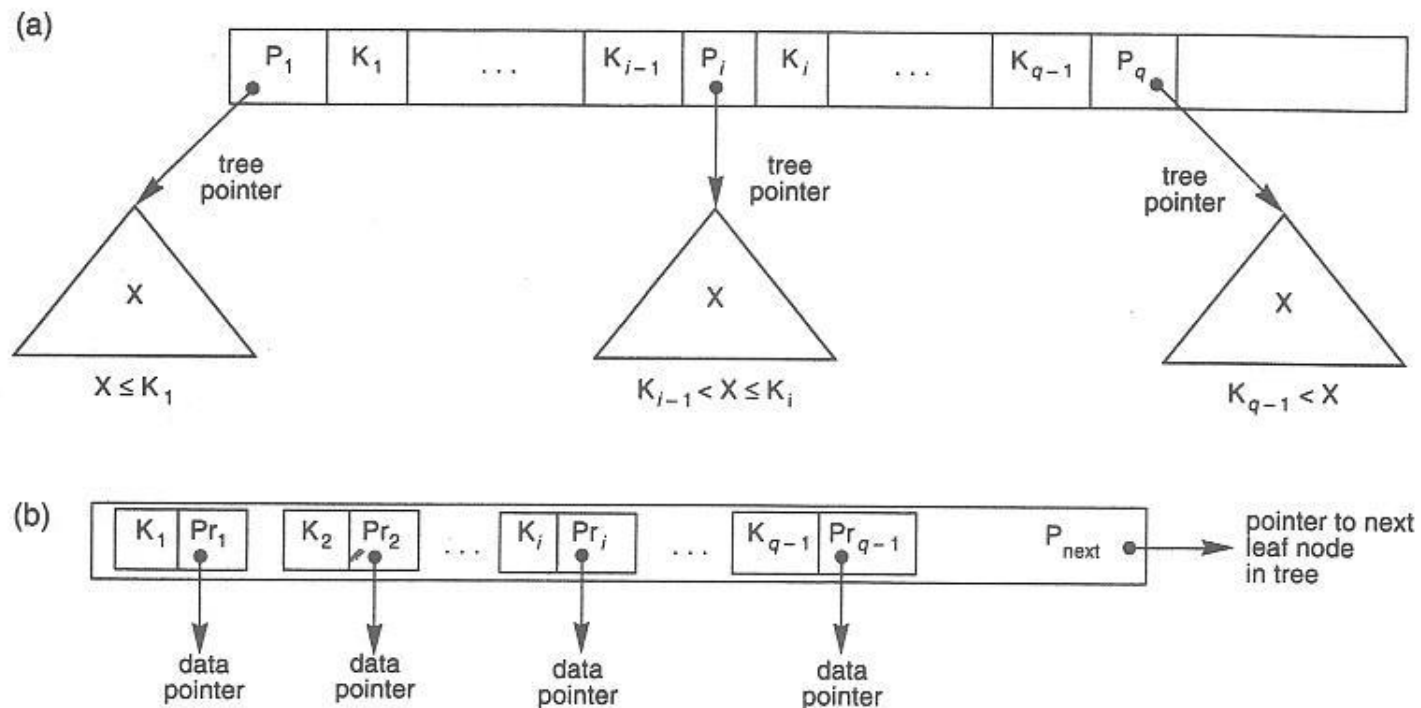


B⁺-tree - Parte II

(5) ogni nodo interno, esclusa la radice, ha almeno $\lceil p/2 \rceil$ tree pointer.

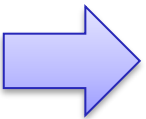
La radice ha almeno 2 tree pointer se è un nodo interno.

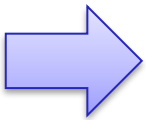
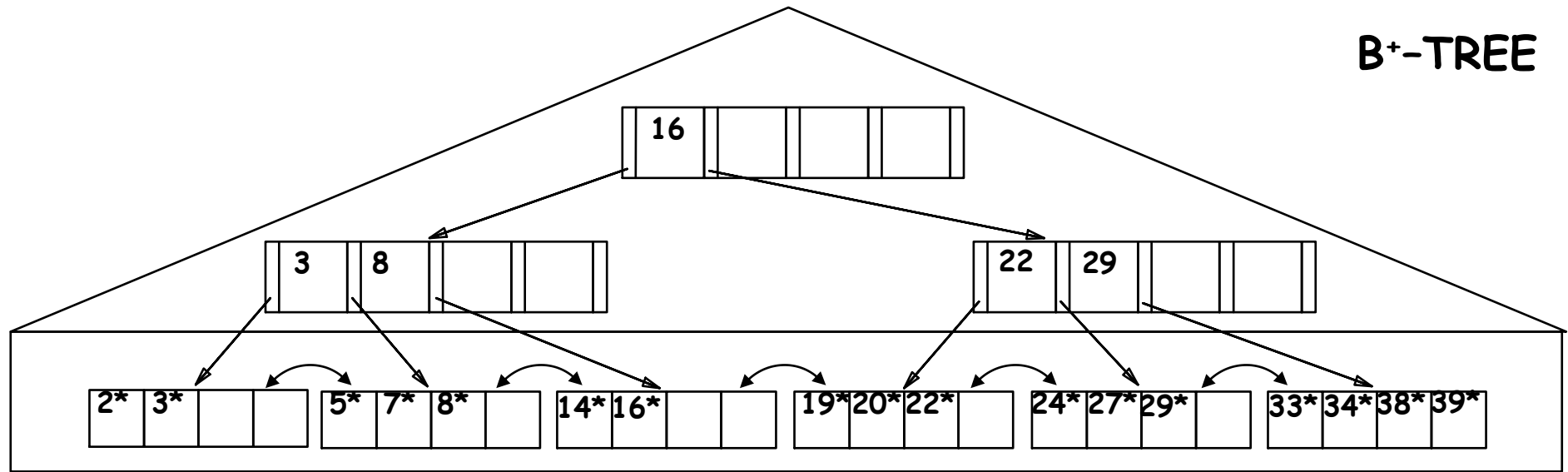
(6) un nodo interno con q tree pointer, con $q \leq p$, ha $q-1$ campi di ricerca.



La struttura dei nodi **foglia** (di ordine p_{leaf}) di un B⁺-albero è la seguente:

- (1) ogni nodo foglia è della forma: $\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_q, Pr_q \rangle P_{next} \rangle$, dove $q \leq p_{leaf}$ e per ogni nodo, si ha che: $K_1 < K_2 < \dots < K_q$, e
 - P_{next} è un tree pointer punta al successivo nodo foglia del B⁺-tree
 - ogni Pr_i è un data pointer che punta al record con valore del campo di ricerca uguale a K_i o ad un blocco contenente tale record (o ad un blocco di puntatori ai record con valore del campo di ricerca uguale a K_i , se il campo di ricerca non è una chiave)
- (4) ogni nodo foglia ha almeno $\lceil p_{leaf}/2 \rceil$ valori
- (5) tutti i nodi foglia sono dello stesso livello





- Di solito un **indice è organizzato a B⁺-albero** per permettere di trovare con pochi accessi, a partire da un valore v , i record di R in cui il valore di A è in una relazione specificata con v .
- Due tipologie di indici ad albero:
 - Indici **statici**: La struttura ad albero viene creata sulla base dei dati presenti nel DB, e non più modificata (o modificata periodicamente).
 - Indici **dinamici**: La struttura ad albero viene aggiornata ad ogni operazione sulla base di dati di inserimento o di cancellazione, memorizzati, preservando le prestazioni senza bisogno di riorganizzazioni.

Indici

- **Indice**: struttura che contiene informazioni sulla posizione di memorizzazione delle tuple sulla base del valore del campo chiave.
- Tali strutture velocizzano l'accesso casuale via chiave di ricerca.
- Possiamo distinguere:
 - **INDICE PRIMARIO**: in questo caso la chiave di ordinamento del file sequenziale coincide con la **chiave di ricerca dell'indice**.
 - **INDICE SECONDARIO**: in questo caso invece la **chiave di ordinamento** e la chiave di ricerca sono diverse.
- Un **indice** può essere **definito su di un insieme A_1, \dots, A_n di attributi**.
 - In questo caso, l'indice contiene un record per ogni combinazione di valori assunti dagli attributi A_1, \dots, A_n nella relazione, e può essere utilizzato per rispondere in modo efficiente ad interrogazioni che specifichino un valore per ciascuno di questi attributi.

Indice primario

- Un indice **primario** è un file ordinato i cui record sono di lunghezza fissa e sono costituiti da due campi
 - Il primo campo è dello stesso tipo del **campo chiave di ordinamento** (Chiave primaria)
 - Il secondo campo è un **puntatore a un blocco del disco**
- Esiste un record nel file dell'indice per ogni blocco nel file di dati

$\langle K(i), RID(i) \rangle$

RID	Matr	Prov	An
1	106	MI	1972
2	102	PI	1970
3	107	PI	1971
4	104	FI	1968
5	100	MI	1970
6	103	PI	1972

Indice secondario

- Un indice **secondario** può essere definito su un **campo non chiave** che è una chiave **candidata** e ha valori univoci, o su un campo non chiave con **valori duplicati**
- Il **primo** campo è dello stesso tipo del campo che non viene usato per ordinare il file ed è chiamato **campo di indicizzazione**
- Il **secondo** campo è un **puntatore al blocco** oppure un **puntatore al record (RID)**

Matr	RID
100	5
102	2
103	6
104	4
106	1
107	3

Indice su Matr



ESEMPI DI INDICI PER ATTRIBUTO CHIAVE O NON CHIAVE

- Tabella:

RID	Matr	Prov	An
1	106	MI	1972
2	102	PI	1970
3	107	PI	1971
4	104	FI	1968
5	100	MI	1970
6	103	PI	1972

- Indici

Matr	RID
100	5
102	2
103	6
104	4
106	1
107	3

Indice su Matr

An	RID
1968	4
1970	2
1970	5
1971	3
1972	1
1972	6

Indice su An

PARTE III

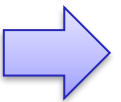
ORDINAMENTO DI ARCHIVI

- Perché è importante l'ordinamento di archivi
 - Risultato di interrogazioni ordinato (order by)
 - Per eseguire alcune operazioni relazionali (join, select distinct, group by)
- Algoritmo sort-merge: costo $N \cdot \log(N)$



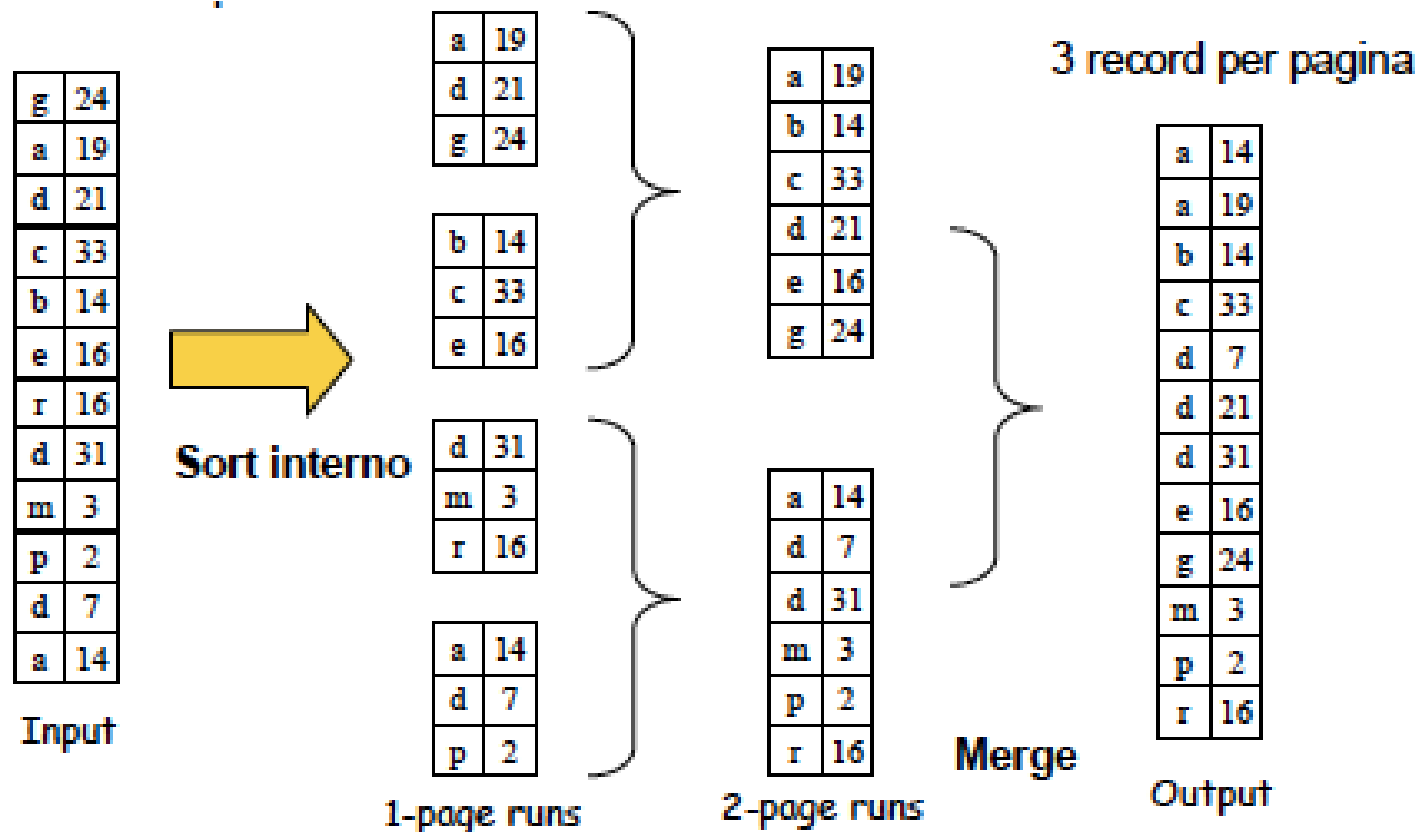
Sort: implementazione

- L'algoritmo più comunemente utilizzato dai DBMS è quello detto di **Merge Sort** a Z vie (**Z-way Sort-Merge**)
- Supponiamo di dover ordinare un input che consiste di un file di **NP pagine** e di avere a disposizione solo **$NB < NP$ buffer** in memoria centrale
- L'algoritmo opera in 2 fasi:
 - **Sort interno**: si leggono una alla volta le pagine del file; i record di ogni pagina vengono ordinati facendo uso di un algoritmo di sort interno (es. Quicksort); **ogni pagina così ordinata, detta anche "run"**, viene scritta su disco in un file temporaneo
 - **Merge**: operando uno o più passi di fusione, le **run vengono fuse**, fino a produrre un'unica run



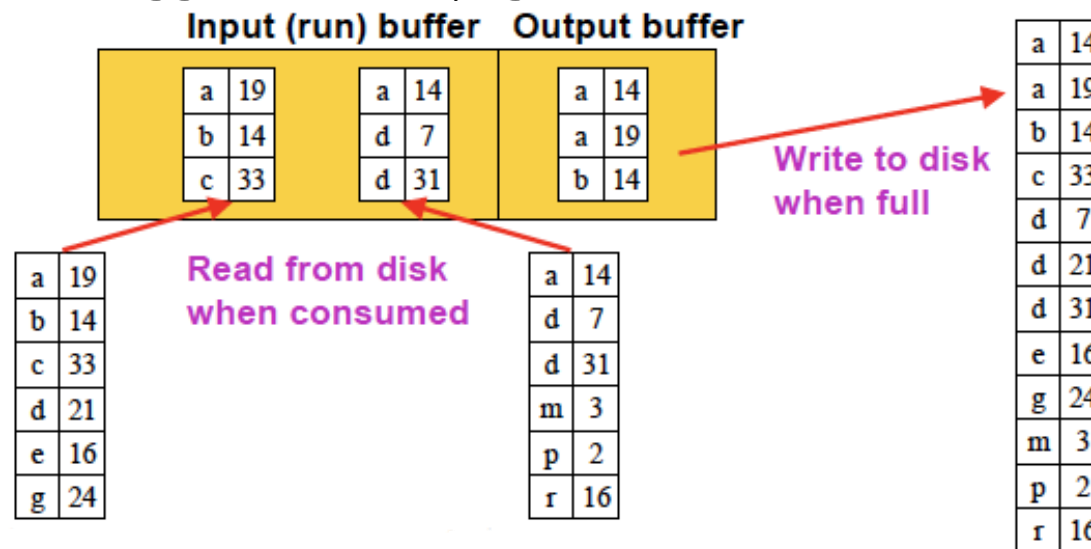
Z-way Merge Sort: caso base

- Per semplicità consideriamo il caso base a **Z = 2 vie**, e supponiamo di avere a disposizione solo NB = 3 buffer in memoria centrale



Fusione delle run

- Nel caso base $Z = 2$ si fondono 2 run alla volta
- Con $NB = 3$, si associa un buffer a ognuna delle run, il terzo buffer serve per produrre l'output, 1 pagina alla volta
- Si legge la prima pagina da ciascuna run e si può quindi determinare la prima pagina dell'output; quando tutti i record di una pagina di run sono stati consumati, si legge un'altra pagina della run

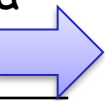


Caso base - Complessità

- Consideriamo per semplicità solo il numero di operazioni di I/O
- Nel caso base $Z = 2$ (e $NB = 3$) si può osservare che:
 - Nella fase di sort interno si leggono e si riscrivono NP pagine
 - Ad ogni passo di merge si leggono e si riscrivono NP pagine ($2 * NP$)
 - Il numero di passi fusione è pari a $\lceil \log_2 NP \rceil$, in quanto ad ogni passo il numero di run si dimezza
 - Il costo complessivo è pertanto pari a $2 * NP * (\lceil \log_2 NP \rceil + 1)$

Esempio: per ordinare $NP = 8000$ pagine sono necessarie circa 224.000 ($2 * 8000 * (\lceil \log_2 8000 \rceil + 1)$) operazioni di I/O; se ogni I/O richiede 20 msec, il sort richiede 4.480 secondi, ovvero circa 1h 15 minuti!

- In realtà se NP non è una potenza di 2 il numero effettivo di I/O è leggermente minore di quello calcolato, in quanto in uno o più passi di fusione può capitare che una run non venga fusa con un'altra



Z-way Sort-Merge: caso generale

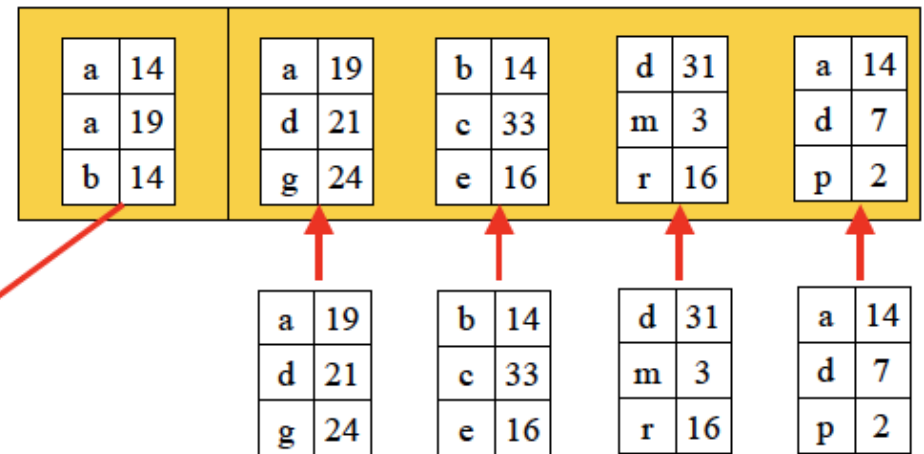
- Una prima osservazione è che nel passo di sort interno, avendo a disposizione **NB buffer**, si possono ordinare **NP pagine** alla volta (anziché una sola), il che porta a un costo di $2 * NP * (\lceil \log_2 (NP/NB) \rceil + 1)$
- Esempio: con $NP = 8000$ pagine e $NB = 3$ si hanno ora 208.000 I/O
 - Miglioramenti sostanziali si possono ottenere se, avendo $NB > 3$ buffer a disposizione, **si fondono NB - 1 run alla volta** (1 buffer è per l'output)
 - In questo caso il numero di passi di fusione è logaritmico in $NB - 1$, ovvero è pari a $2 * NP * (\lceil \log_{NB-1} (NP/NB) \rceil + 1)$

- Esempio:

con $NP = 8000$ pagine e $NB = 11$

si hanno 64000 I/O,

per un tempo stimato pari a 1280 sec



Utilità del Sort

- Oltre che per ordinare le tuple, il **Sort** può essere utilizzato per:
 - Query in cui compare l'opzione DISTINCT, ovvero per eliminare i duplicati
 - Query che contengono la clausola GROUP BY

REALIZZAZIONE DEGLI OPERATORI RELAZIONALI

- Si considerano i seguenti operatori:
 - *Proiezione*
 - *Selezione*
 - *Raggruppamento*
 - *Join*

```
SELECT  *  
        Provincia  
FROM    Studenti R
```

```
SELECT  DISTINCT  
        Provincia  
FROM    Studenti R
```

- Caso di DISTINCT: approccio basato sull'ordinamento (non è l'unico!):
 - Si legge R e si scrive T che contiene solo gli attributi della SELECT
 - Si ordina T su tutti gli attributi
 - Si eliminano i duplicati

RESTRIZIONE CON CONDIZIONE SEMPLICE

```
SELECT *  
FROM   Studenti R  
WHERE  R.Provincia = 'PI'
```

- Senza indice e dati disordinati:
Npag(R).
- Con indice (B⁺-albero): CI + CD

OPERAZIONI DI AGGREGAZIONE

- **Senza GROUP BY**
 - Si visitano i dati e si calcolano le funzioni di aggregazione.
 - `Select count(*) from Persone`
- **Con GROUP BY**
 - Approccio basato sull'ordinamento (non è l'unico!):
 - Si ordinano i dati sugli attributi del `GROUP BY`, poi si visitano i dati e si calcolano le funzioni di aggregazione per ogni gruppo.
 - `Select cognome, count(cognome) from Persone group by cognome`

GIUNZIONE


```
SELECT *  
FROM   Studenti S, Esami E  
WHERE  S.Matricola=E.Matricola
```

- $R \times S$ è grande; pertanto, $R \times S$ seguito da una restrizione è inefficiente.

R	A	B
	22 a	
	87 s	
	45 h	
	32 b	

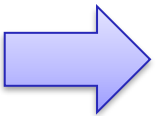
PJ: R.A = S.A

S	A	C	D
	22	z	8
	45	k	4
	22	s	7
	87	s	9
	32	c	3
	45	h	5
	32	g	6



A	C	D	B
22	z	8	a
22	s	7	a
87	s	9	s
45	k	4	h
45	h	5	h
32	c	3	b
32	g	6	b

Anche se logicamente il Join sia commutativo, dal punto di vista fisico vi è una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro (o "esterno", "outer") e operando destro (o "interno", "inner")



NESTED LOOPS

- Il costo di esecuzione dipende dallo spazio a disposizione in memoria centrale

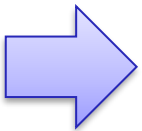
```
foreach record r in R do
  foreach record s in S do
    if ri = sj then
      aggiungi <r, s> al risultato
```

- Nel caso base in cui vi sia 1 buffer per R e 1 buffer per S, bisogna leggere 1 volta R e Nrec(R) volte S, ovvero tante volte quante sono le tuple della relazione esterna, per un totale di

$$N_{\text{pag}}(R) + N_{\text{rec}}(R) * N_{\text{pag}}(S) \text{ I/O}$$

- Se è possibile allocare Npag(S) buffer per la relazione interna il costo si riduce a

$$N_{\text{pag}}(R) + N_{\text{pag}}(S)$$



Nested Loops: esempio (relazione esterna vs interna)

- Se si fa il join tra Dipartimenti e Impiegati ($\text{DipImp} = \text{NumDip}$) e si ha:
$$\text{Npag}(\text{Dipartimenti}) = 20 \quad \text{Nrec}(\text{Dipartimenti}) = 100$$
$$\text{Npag}(\text{Impiegati}) = 1000 \quad \text{Nrec}(\text{Impiegati}) = 10000$$

e considerando il caso base (1 buffer per ciascuna relazione):
- **Dipartimenti esterna:**
$$\text{Npag}(\text{Dipartimenti}) + \text{Nrec}(\text{Dipartimenti}) * \text{Npag}(\text{Impiegati}) = 100.020 \text{ I/O}$$
- **Impiegati esterna:**
 - $$\text{Npag}(\text{Impiegati}) + \text{Nrec}(\text{Impiegati}) * \text{Npag}(\text{Dipartimenti}) = 201.000 \text{ I/O}$$

NESTED LOOPS

- Per ogni record della relazione esterna R, si visita tutta la relazione interna S.

- Costo: $N_{\text{pag}}(R) + N_{\text{rec}}(R) * N_{\text{pag}}(S)$

$$\approx N_{\text{pag}}(R) * \frac{N_{\text{rec}}(R)}{N_{\text{pag}}(R)} * N_{\text{pag}}(S)$$

```
foreach record r in R do
  foreach record s in S do
    if r_i = s_j then
      aggiungi <r, s> al risultato
```

$$\frac{N_{\text{rec}}(R)}{N_{\text{pag}}(R)} < \frac{N_{\text{rec}}(S)}{N_{\text{pag}}(S)}$$

con S esterna:

- Costo: $N_{\text{pag}}(S) + N_{\text{rec}}(S) * N_{\text{pag}}(R)$

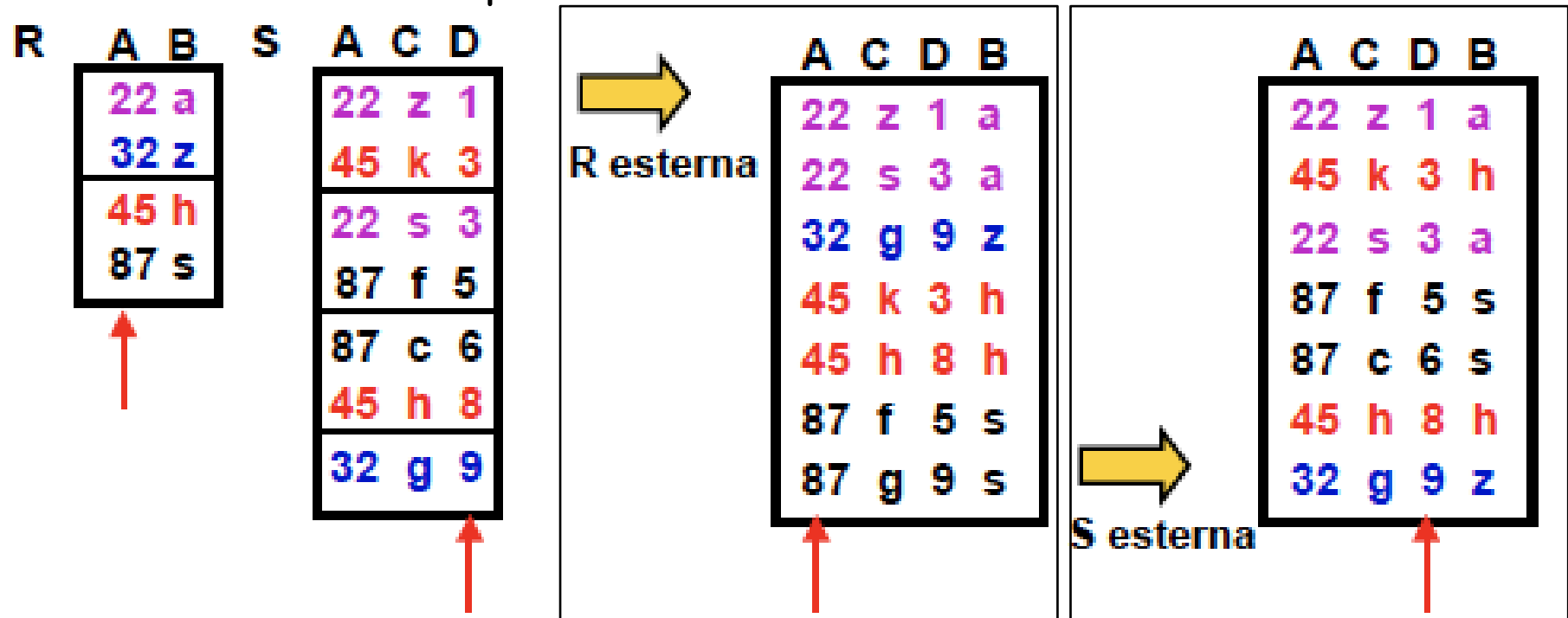
$$\approx N_{\text{pag}}(S) * \frac{N_{\text{rec}}(S)}{N_{\text{pag}}(S)} * N_{\text{pag}}(R)$$

si sceglierà R come esterna e S come interna se
 $N_{\text{rec}}(R) * N_{\text{pag}}(S) < N_{\text{rec}}(S) * N_{\text{pag}}(R)$
che corrisponde a dire che le tuple di R sono più grandi di quelle di S

- Come esterna conviene la relazione con record più lunghi/grandi

Nested loops: cammini di accesso

- L'ordine con cui vengono generate le tuple del risultato coincide con l'ordine eventualmente presente nella relazione esterna



Pertanto se l'ordine che si genera è "**interessante**", ad esempio perché la query contiene **ORDER BY R.A**, la scelta della relazione esterna può **risultarne influenzata**

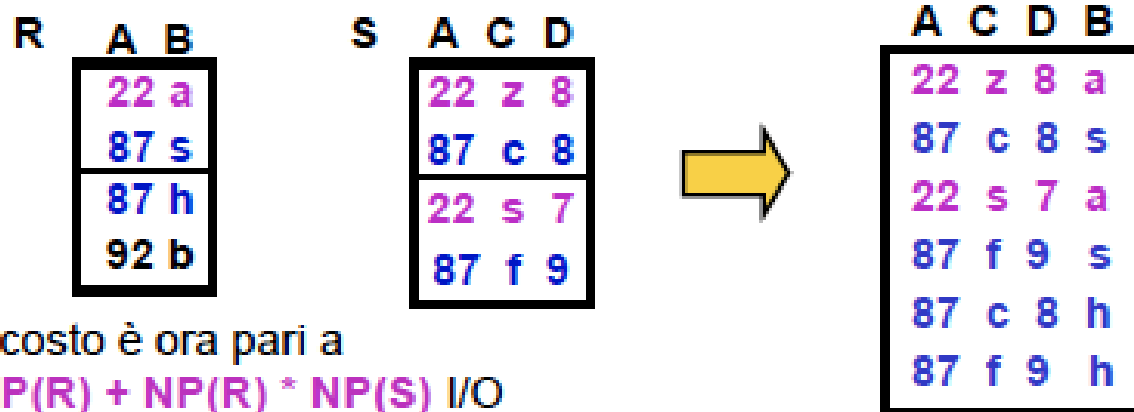
Nested loop a pagine (*PageNestedLoop*)

- Molti DBMS usano una variante del Nested Loops, detta **Nested loop a pagine**, che, rinunciando a preservare l'ordine della relazione esterna, risulta più efficiente in quanto **esegue il join di tutte le tuple in memoria prima di richiedere nuove pagine della relazione interna**

Per ogni pagina p_R di R:

{ Per ogni pagina p_S in S:

{ esegui il join di tutte le tuple in p_R e p_S } }



Il costo è ora pari a
 $NP(R) + NP(R) * NP(S)$ I/O

La strategia si
estende anche al
caso in cui a R siano
assegnati più buffer