ELABORAZIONE DELLE INTERROGAZIONI

Roberto Basili

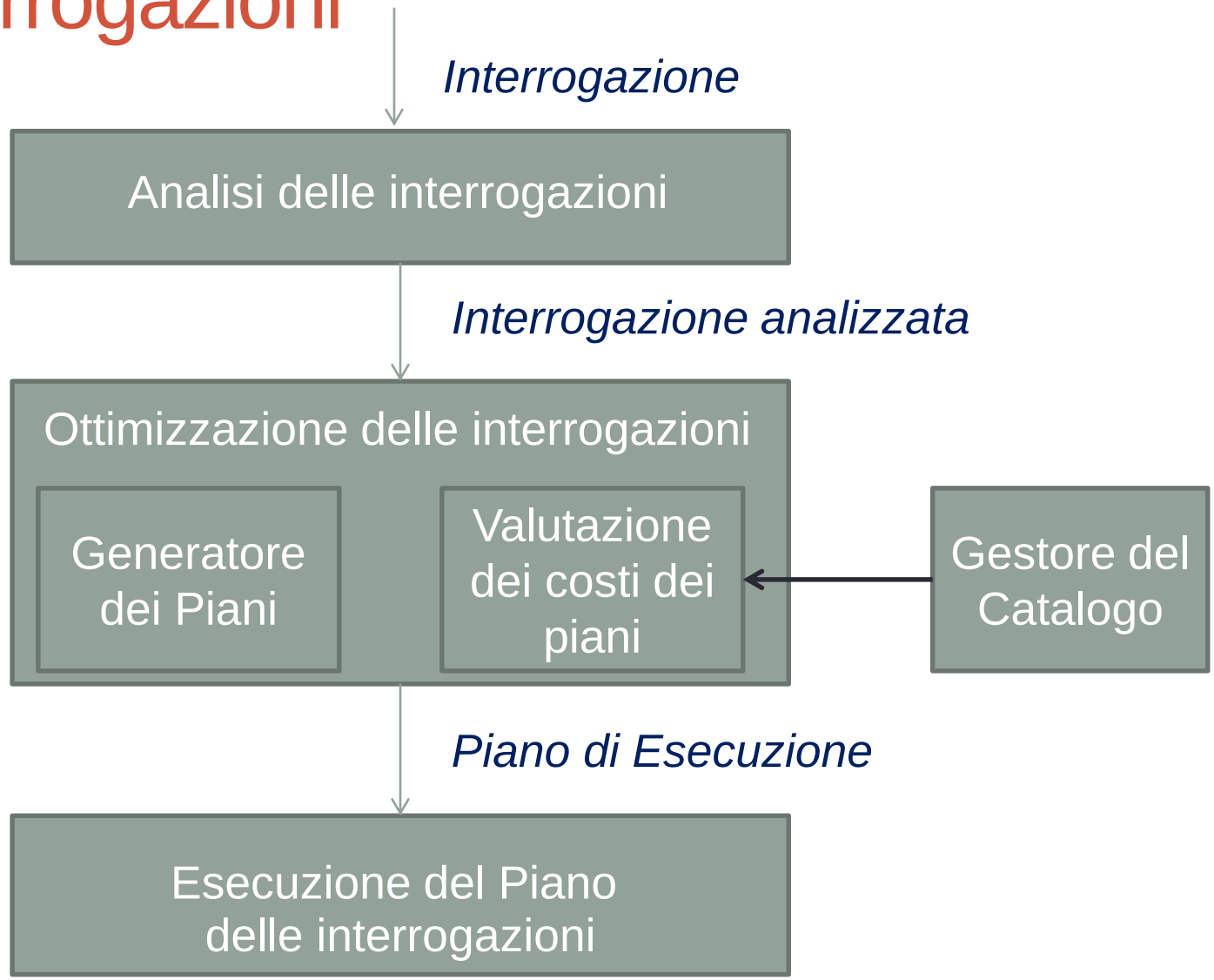
Corso di Basi di Dati

a.a. 2021/22

Introduzione alla valutazione delle interrogazioni

- Piano: albero composto da operatori dell'algebra relazionale, con scelta dell'algoritmo per ciascun operatore
  - Ciascun operatore è tipicamente implementato con una interfaccia di tipo 'pull': quando un operatore viene richiesto per la successiva tupla in uscita, esso richiama i suoi valori in ingresso ed esegue il calcolo
- Due argomenti principali nell'ottimizzazione delle interrogazioni:
  - Per una data interrogazione, quali piani vengono considerati?
    - Algoritmo per cercare nello spazio dei piani quello con il costo (stimato) minore
  - Come viene stimato il costo di un piano?
- Idealmente: vogliamo trovare il piano migliore. In pratica: evitiamo i piani peggiori!
- Studieremo l'approccio del System R

Processo di Ottimizzazione delle Interrogazioni



Alcune tecniche comuni

- Gli algoritmi per la valutazione degli operatori relazionali usano estensivamente alcune semplici idee:
  - **Indicizzazione**: usare le condizioni nella WHERE per reperire piccoli insiemi di tuple (selezioni, join)
  - **Iterazione**: a volte è più veloce la scansione di tutte le tuple anche se c'è un indice (e a volte possiamo scansionare le data entry in un indice invece che nella tabella)
  - **Partizionamento**: usando ordinamento o hashing, possiamo partizionare le tuple in ingresso e sostituire una operazione costosa con operazioni simili su un minor numero di dati

** Tenete a mente queste tecniche durante la nostra discussione sulla valutazione delle interrogazioni!*

Statistiche e catalogo

- Abbiamo bisogno di informazioni sulle relazioni e sugli indici coinvolti. I cataloghi tipicamente contengono almeno
  - **Numero di tuple** (Ntuple) e **numero di pagine** (Npagine) per ciascuna relazione
    - Numero di I/O da disco
    - Occupazione media delle pagine coinvolte
  - Numero di **valori distinti della chiave** (Nchiavi) e Numero di pagine (NpagineInd) per ciascun indice
    - Cardinalità di un indice e occupazione di memoris disco per indicie
  - **Altezza dell'indice**, **valori di chiave minimo/massimo** (Minimo/Massimo) per ciascun indice ad albero
    - Ampiezza dell'intervallo dei valori della chiave di ricerca nel caso di indici arborei

Statistiche e Catalogo (2)

- I cataloghi vengono aggiornati periodicamente
  - Aggiornare ogni volta che i dati cambiano è troppo costoso; ci sono comunque una quantità di approssimazioni, quindi una piccola inconsistenza è accettabile
- A volte vengono memorizzate informazioni più dettagliate (ad esempio istogrammi dei valori in alcuni campi)

Percorsi di accesso

- Un percorso di accesso è un metodo per reperire tuple:
  - scansione del file, o un indice che soddisfa una selezione (nell'interrogazione)
- un indice ad albero soddisfa (un concatenamento di) *termini* che coinvolgono solo attributi in un prefisso della chiave di ricerca
  - Ad esempio, l'indice ad albero su <a, b, c>

    soddisfa la selezione

    a = 5 AND b = 3, e

    a = 5 AND b > 6,

    ma non

    b = 3

Percorsi di accesso (2)

- Un indice hash soddisfa (un concatenamento di) termini con un termine attributo = valore per ciascun attributo nella chiave di ricerca dell'indice
  - Ad esempio, un indice hash su <a, b, c> soddisfa

    a = 5 AND b = 3 AND c = 5;

    ma non soddisfa

    b = 3,

    oppure

    a = 5 AND b = 3,

    oppure

    a > 5 AND b = 3 AND c = 5

Schema per gli esempi

- Simile al vecchio schema; è stato aggiunto **`vnome`** per le variazioni
- Prenota:
  - ogni tupla è lunga 40 bytes, 100 tuple per pagina, 1000 pagine
- Velisti:
  - ogni tupla è lunga 50 byte, 80 tuple per pagina, 500 pagine

```
Velisti(vid:integer, vnome:string,
                esperienza:integer, età:real)

Prenota(vid:integer, bid:integer, giorno:dates,
                                        pnome:string)
```

Nota sulle selezioni complesse

(giorno < 8/9/94 AND rnome = 'Paul') OR bid = 5 OR vid = 3

- Le condizioni di selezione sono prima convertite in forma normale congiuntiva (conjunctive normal form, CNF)

  (giorno < 8/9/94 OR bid = 5 OR vid = 3) AND

  (rnome = 'Paul' OR bid = 5 OR vid = 3)

- Noi tratteremo solo casi senza OR; si rimanda al testo per il caso generale

Un approccio alle selezioni

- Trovare il percorso di accesso più selettivo, usarlo per leggere le tuple e applicare ogni termine rimanente che non soddisfa l'indice:
  - **percorso di accesso più selettivo**: un indice o una scansione di file che stimiamo richiedere il minor numero di I/O di pagina
  - I termini che soddisfano questo indice riducono il numero di tuple restituito: gli altri termini sono usati per scartare alcune delle tuple restituite, ma non influenzano il numero di tuple/pagine lette

Approccio alle Selezioni (2)

- Consideriamo

  giorno < 8/9/94 AND bid = 5 AND vid = 3.

- Si deve usare un indice ad albero B+ su giorno; poi, bid = 5 e vid = 3 devono essere controllati su ciascuna tupla restituita (*on the fly*).

- Analogamente, si potrebbe usare un indice hash su <bid, vid>; si dovrà allora controllare giorno < 8/9/94

Usare un indice per le selezioni

- Il costo dipende dal numero delle tuple compatibili e dal clustering
  - Il costo della ricerca delle data entry compatibili (tipicamente piccolo) più il costo della lettura dei record (potrebbe essere grande in assenza di clustering)
  - Nel nostro esempio, ipotizzando la distribuzione uniforme dei nomi, circa il 10% delle tuple è compatibile:
    - Corrispondono dunque a 100 pagine con ca. 10.000 tuple
  - Con un indice clustered dunque il costo è poco più di 100 I/O ma …
  - … senza cluster, possiamo arrivare sino a 10.000 I/O!

```
SELECT *
FROM Prenota P
WHERE P.rnome < 'C%'
```

Proiezione

```
SELECT DISTINCT P.vid, P.bid
FROM Prenota P
```

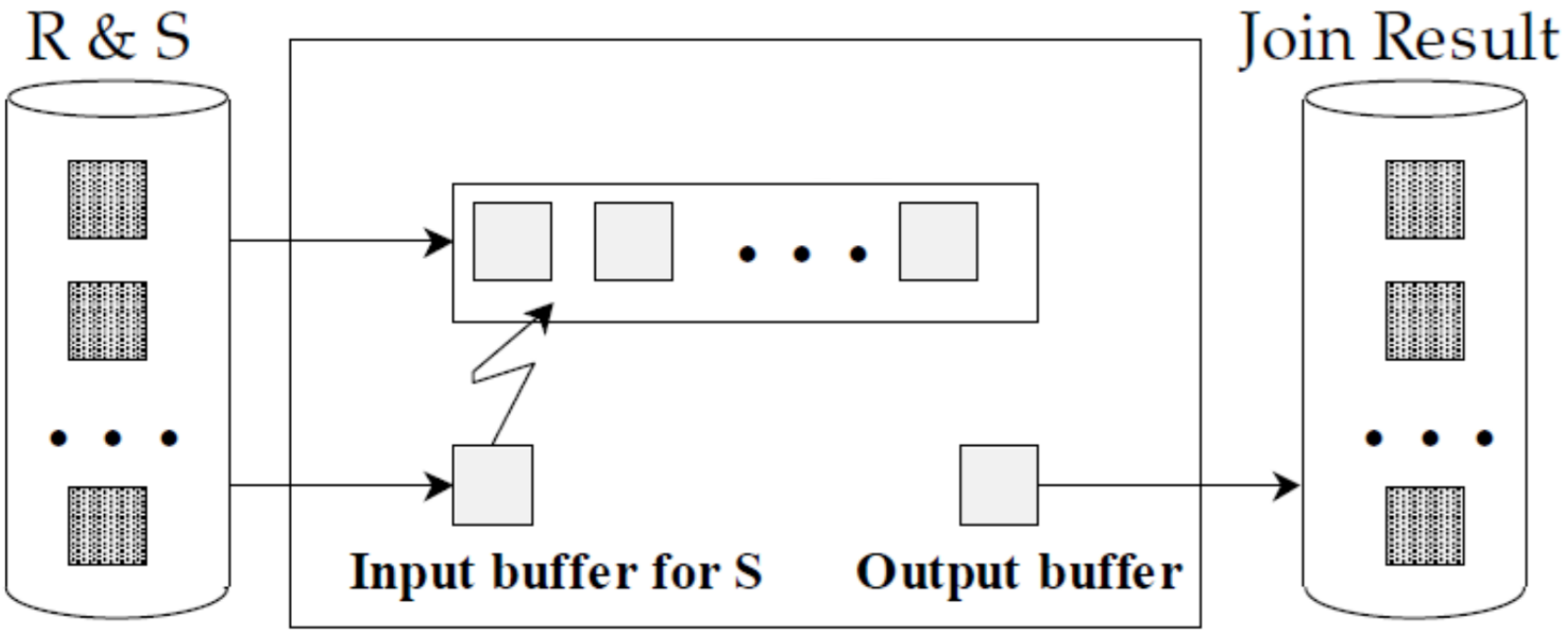
- La parte costosa è la rimozione dei duplicati
  - I sistemi SQL non rimuovono i duplicati a meno che la parola chiave DISTINCT sia specificata nell'interrogazione.
- Approccio con ordinamento: ordinare su <vid, bid> e rimuovere i duplicati (si può ottimizzare eliminando durante l'ordinamento le informazioni non desiderate)
- Approccio hashing: costruire un hash su <vid, bid> per creare partizioni. Caricare le partizioni in memoria una alla volta, costruire in memoria una struttura hash ed eliminare i duplicati
- Se c'è un indice che ha nella chiave di ricerca sia P.vid che P.bid, può essere più economico ordinare le data entry!

Join: Index Nested Loop

Per ogni tupla p in P
    Per ogni tupla v in V dove p_i == v_j
        Aggiungi <p, v> al risultato

- Se c'è un indice sulla colonna di join di una relazione (diciamo V), si può utilizzarla come relazione interna (inner) e sfruttare l'indice
  - Costo: M + ((M*pP) * costo della ricerca delle tuple di V che soddisfano la condizione)
  - M = numero di pagine di P, pP = numero delle tuple di P per pagina

JOIN



Join: Index Nested Loop (2)

Per ogni tupla p in P
    Per ogni tupla v in V dove p_i == v_j
        Aggiungi <p, v> al risultato

- Per ciascuna tupla di P, il costo per sondare l'indice V è circa
  - 1.2 per gli indici hash,
  - 2-4 per gli alberi B+.
- Il costo quindi per trovare le tuple di V (assumendo l'Alternativa (2) o la (3) per le data entry) dipende dal *clustering*
  - Indice clustered: 1 I/O in tutto (tipicamente),
  - non clustered: fino a 1 I/O per ogni tupla compatibile di V

Esempi di Index Nested Loop

- Indice hash (Alternativa 2) su vid di Velisti (come inner)
  - Scansione di Prenota: 1000 I/O di pagina, 100*1000 tuple
  - Per ciascuna tupla di Prenota:
    - 1.2 I/O per leggere la data entry nell'indice, più …
    - … 1 I/O per leggere (l'unica) tupla di Velisti che soddisfa le condizioni.
  - Totale: **220.000 I/O**
- Indice hash (Alternativa 2) su vid di Prenota (come inner)
  - Scansione di Velisti: 500 I/O di pagina, 80*500 tuple
  - Per ciascuna tupla di Velisti: 1.2 I/O per trovare la pagine dell'indice con le data entry, più il costo per la lettura delle tuple rilevanti di Prenota. Ipotizzando una distribuzione uniforme, 2.5 prenotazioni per velista (100.000/40.000). Il costo per leggerle è 1 oppure 2.5 I/O a seconda che l'indice sia o meno *clustered* (148.500 I/O)

Join: Sort-Merge ($P \underset{i=j}{\bowtie} V$)

- Ordinare P e V sulla colonna di join, poi scansionarle per eseguire un “merge” (sulla colonna di join) e produrre le tuple risultato.
  1. Far avanzare la scansione di P fino a quando la P-tupla corrente ≥ tupla corrente di V, poi …
  2. far avanzare la scansione di V fino a quando la V-tupla corrente ≥ tupla corrente di P;
  3. Ripetere 1 e 2 fino a quando la tupla corrente di P = tupla corrente di V
  4. A questo punto, tutte le tuple di P con lo stesso valore in Pi (gruppo corrente di P) e tutte le tuple di V con lo stesso valore in Vj (gruppo corrente di V) sono uguali:
     - produrre <p, v> per tutte le coppie di tali tuple
  5. Quindi continuare con la scansione di P e V
- P viene scansionata una volta; ciascun gruppo di V viene scansionato una volta per ogni tupla rilevante di R (per trovare le pagine necessarie nel buffer sono probabili scansioni multiple dei gruppi in V)

```
proc smjoin(R, S, 'R_i = S'_j)

if R not sorted on attribute i, sort it;
if S not sorted on attribute j, sort it;

Tr = first tuple in R;                                  // ranges over R
Ts = first tuple in S;                                  // ranges over S
Gs = first tuple in S;                                  // start of current S-partition

while Tr ≠ eof and Gs ≠ eof do {

    while Tr_i < Gs_j do
        Tr = next tuple in R after Tr;                  // continue scan of R

    while Tr_i > Gs_j do
        Gs = next tuple in S after Gs                   // continue scan of S

    Ts = Gs;                                            // Needed in case Tr_i ≠ Gs_j
    while Tr_i == Gs_j do {                             // process current R partition
        Ts = Gs;                                        // reset S partition scan
        while Ts_j == Tr_i do {                         // process current R tuple
            add ⟨Tr, Ts⟩ to result;                     // output joined tuples
            Ts = next tuple in S after Ts;}             // advance S partition scan
        Tr = next tuple in R after Tr;                  // advance scan of R
        }                                               // done with current R partition

    Gs = Ts;                                            // initialize search for next S partition

}
```

Esempio di join di tipo sort-merge

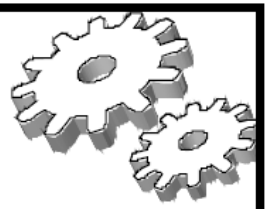
vid	vnome	esperienza	età
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

vid	bid	giorno	pnome
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Costo: M log M + N log N + (M + N)
  - Il costo della scansione, (M + N), potrebbe essere M*N (molto improbabile!)
- Con 35, 100 o 300 pagine di buffer, sia Prenota che Velisti possono essere ordinate in 2 passi; costo totale del join: 7500 I/Os

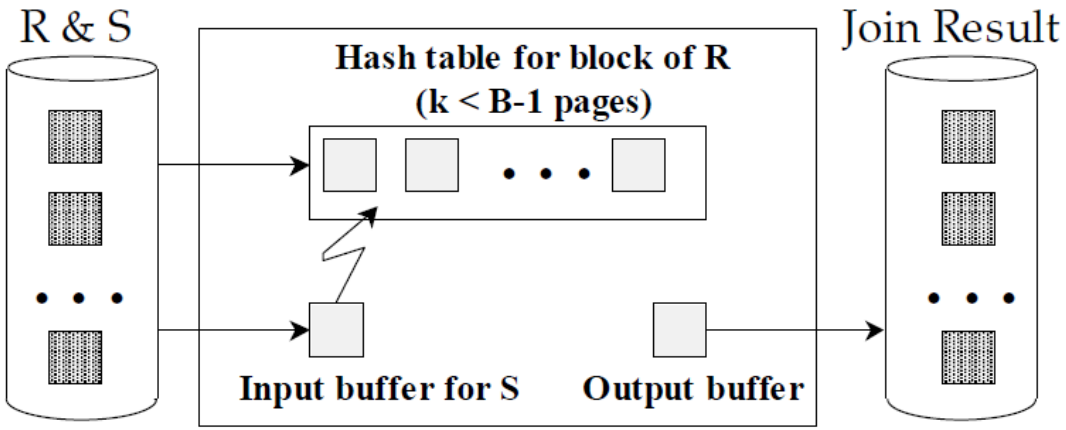
(Il costo corrispondente del BNL è tra 2500 e 15000 I/Os)

Block Nested Loop

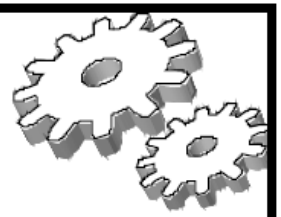


Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



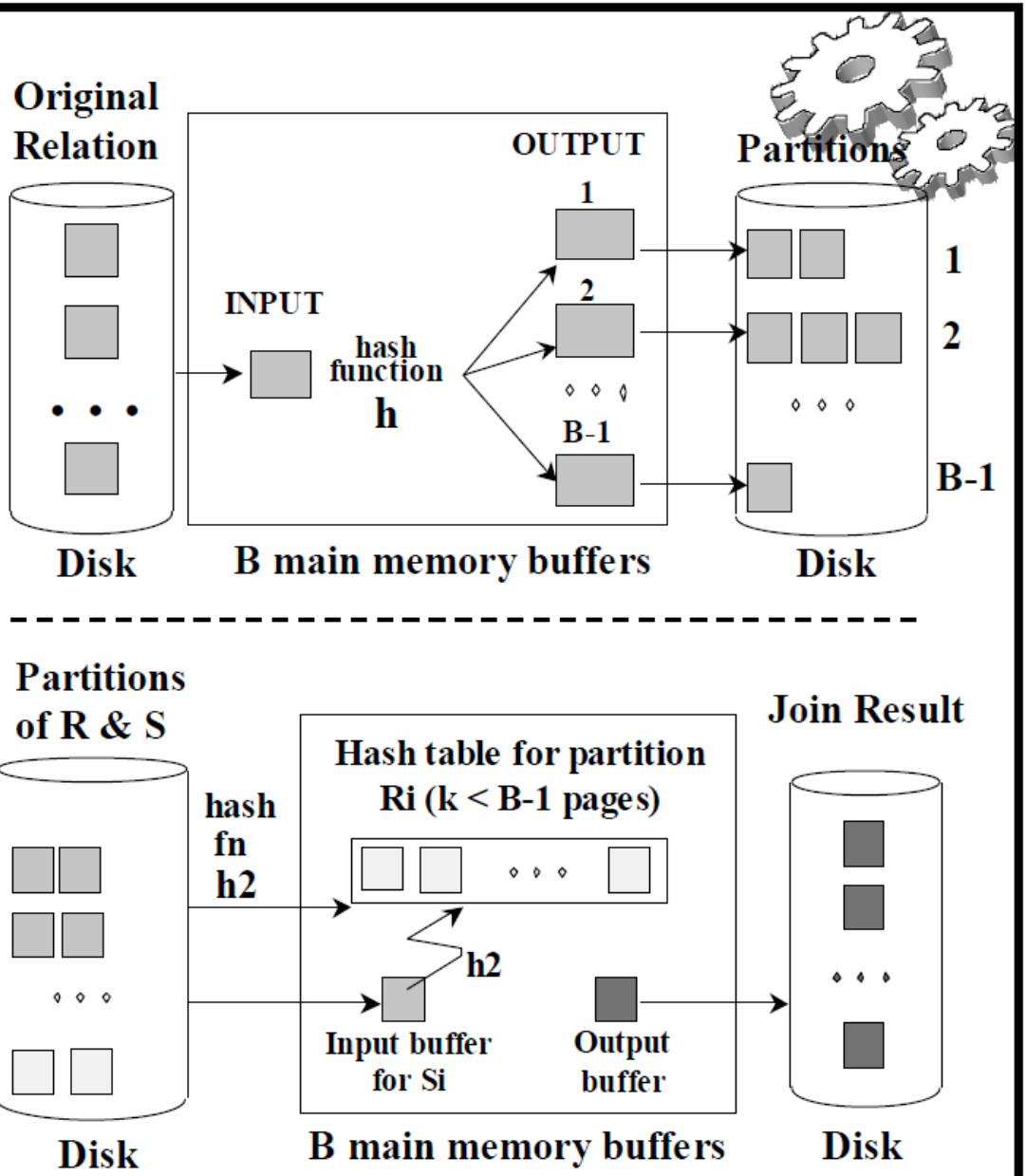
Examples of Block Nested Loops



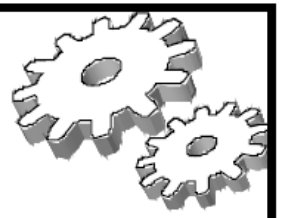
- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \#\ of\ pages\ of\ outer\ /\ blocksize \rceil$
- With Reserves (R) as outer, and 100 pages of R:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.
- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



Cost of Hash-Join



- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

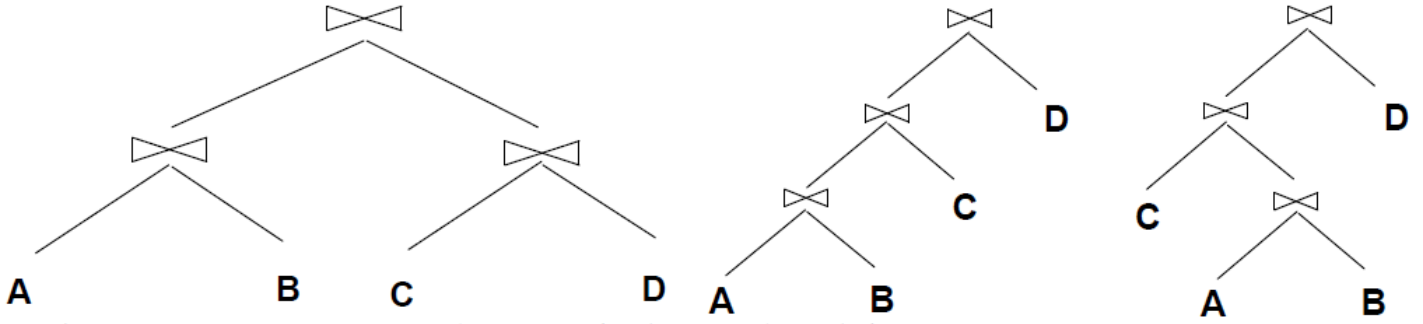
Principali caratteristiche dell'Ottimizzatore System R

- Impatto:
  - Largamente usato correntemente: lavora bene per un numero di join < 10
- Stima del costo: approssimato al meglio
  - Statistiche, mantenute nei cataloghi di sistema, usate per stimare il costo delle operazioni e le dimensioni del risultato
  - Prende in considerazione combinazioni dei costi della CPU e di I/O
- Spazio dei piani: troppo grande, deve essere ridotto
  - Vengono considerati solo i piani left-deep
    - I piani left-deep consentono all'uscita di ciascun operatore di essere utilizzata dall'operatore successivo senza dover essere memorizzata in una relazione temporanea
  - Il prodotto cartesiano viene evitato

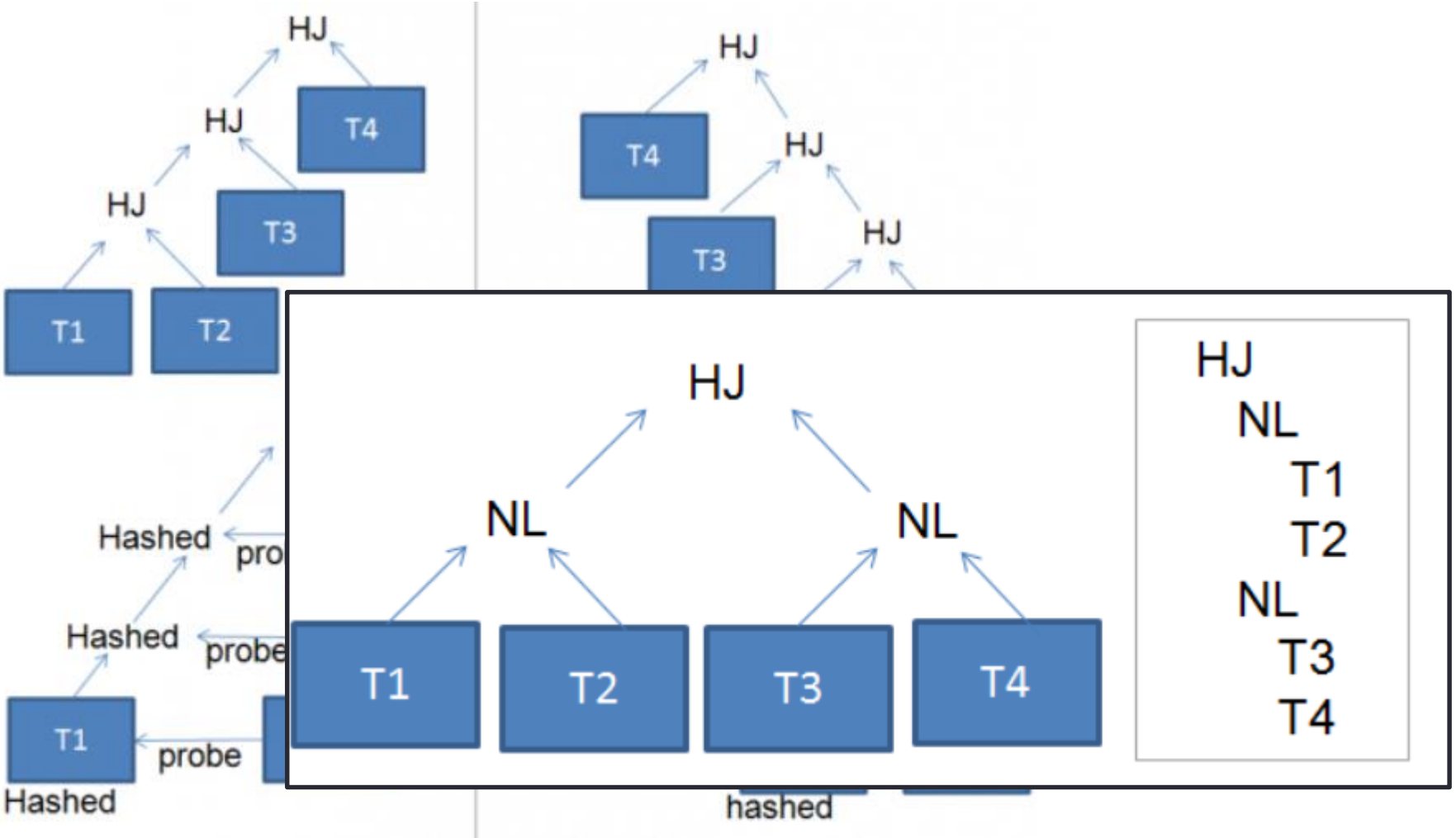
Piani Left-deep

Queries Over Multiple Relations

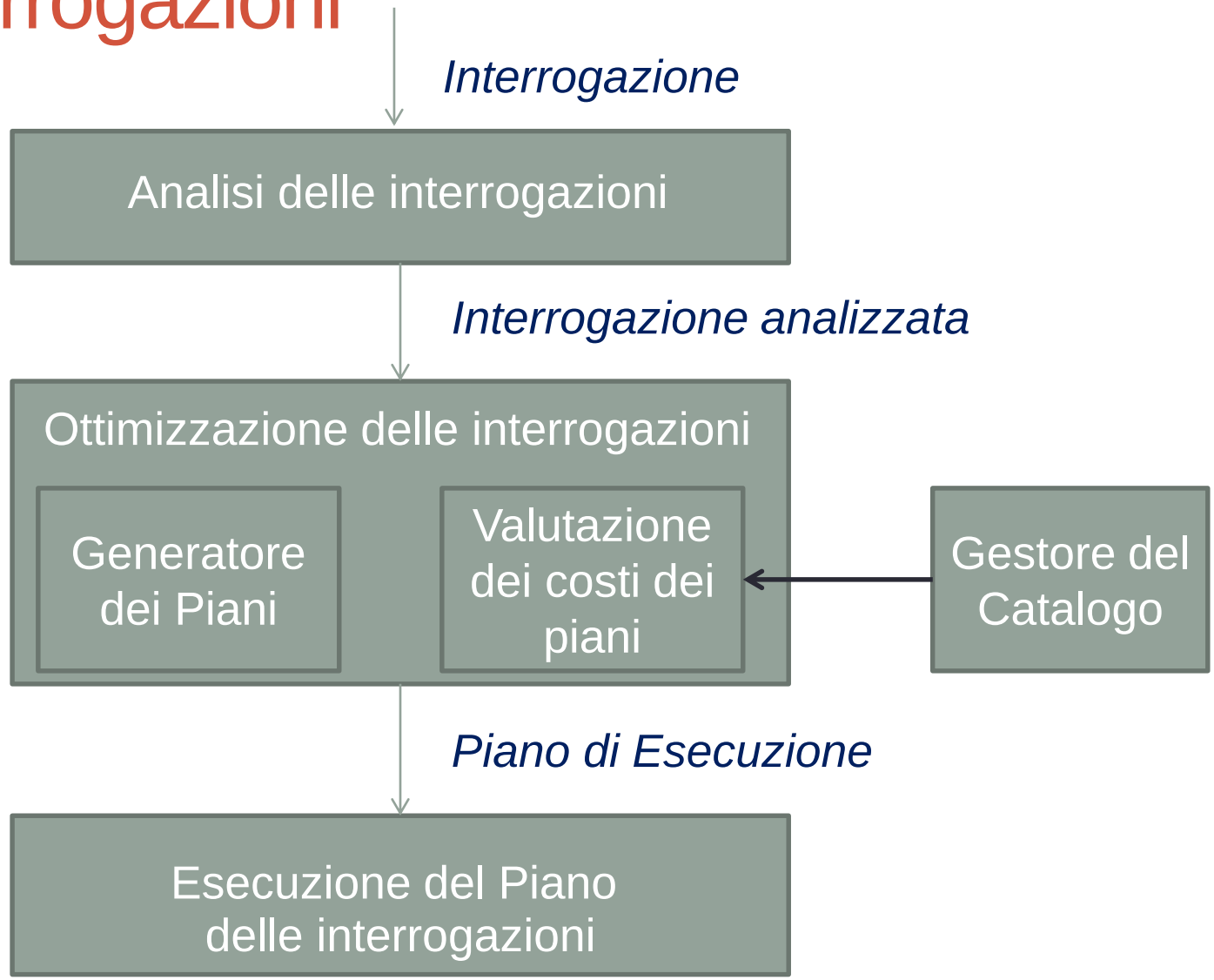
- Fundamental decision in System R: *only left-deep join trees* are considered.
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict the search space.*
  - Left-deep trees allow us to generate all *fully pipelined* plans.
    - Intermediate results not written to temporary files.
    - Not all left-deep trees are fully pipelined (e.g., SM join).



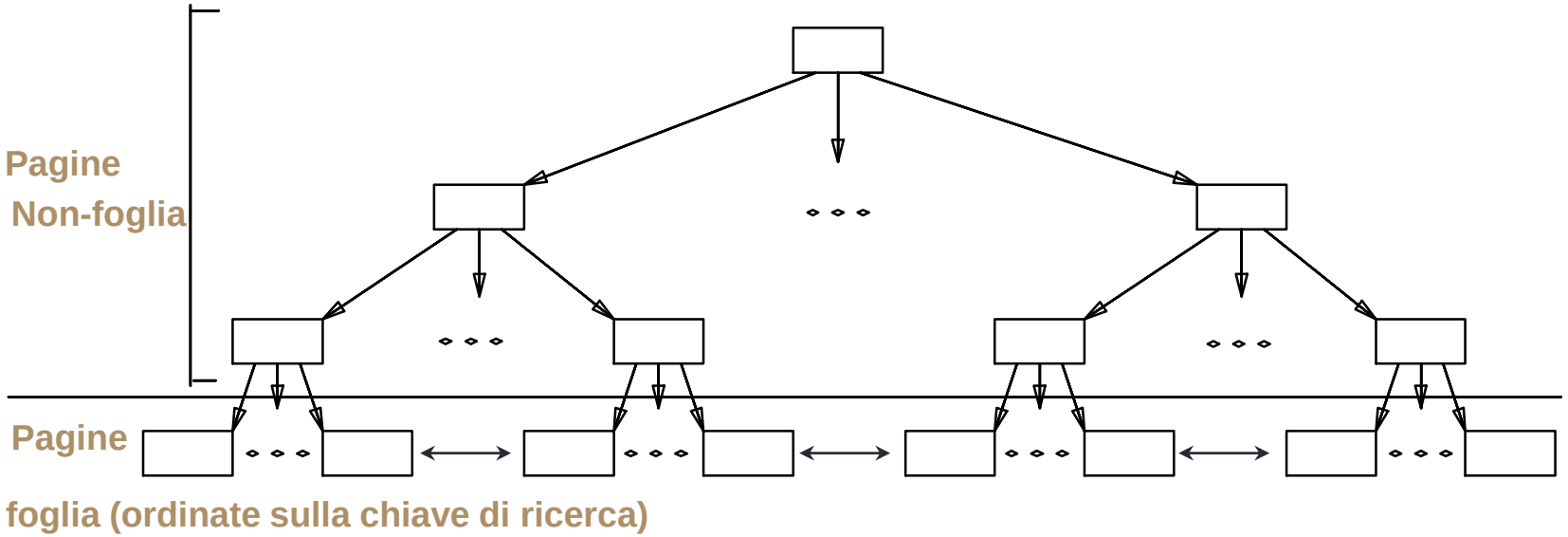
Differenze tra i piani



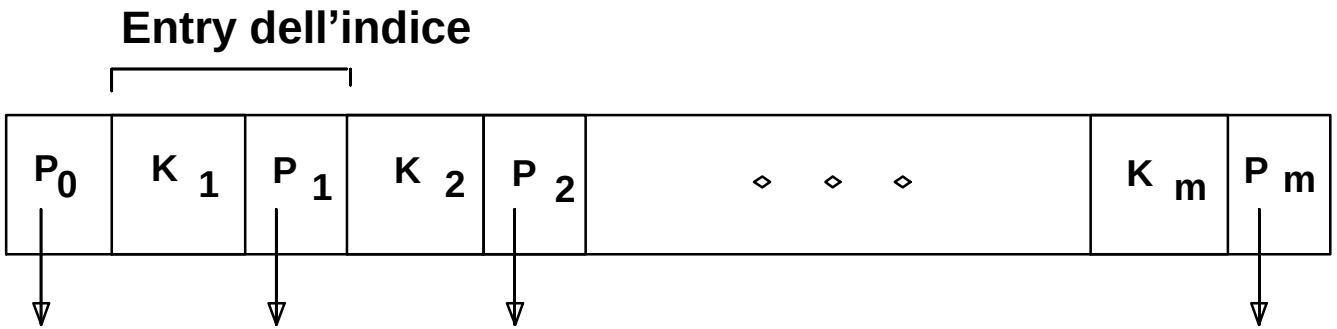
Processo di Ottimizzazione delle Interrogazioni



Indici ad albero B+

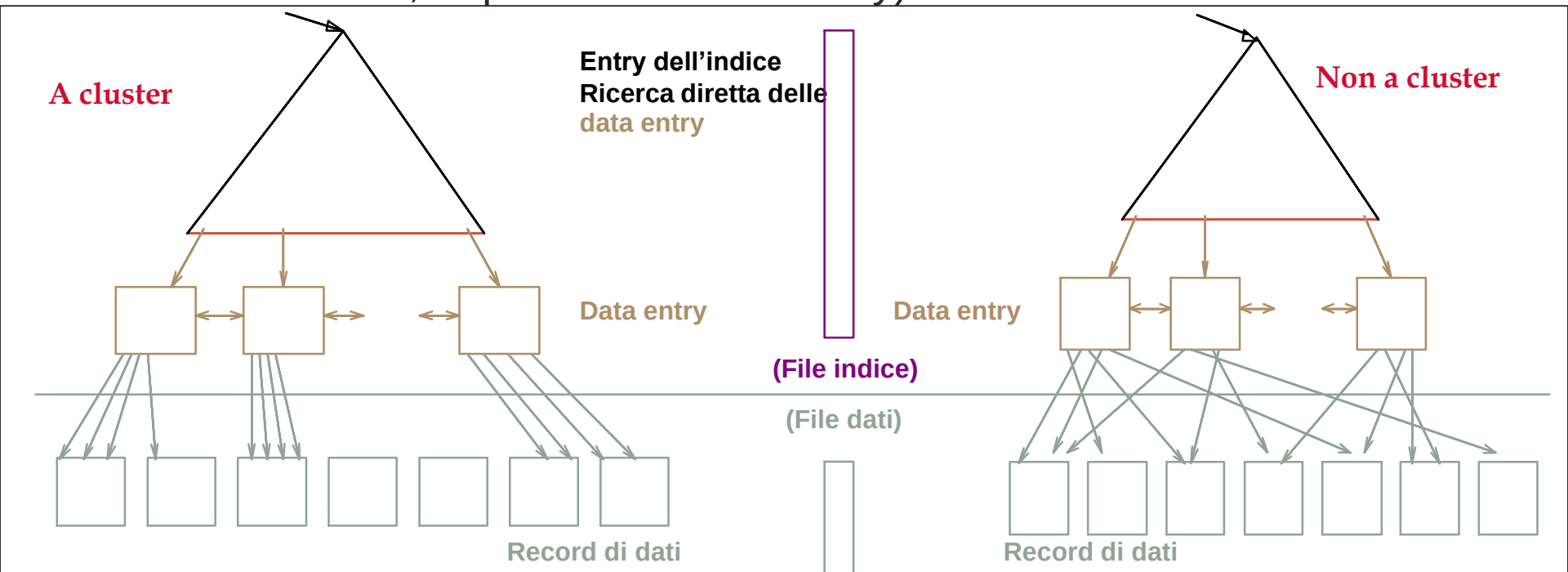


- Le pagine foglia contengono *data record*, e sono collegate (prec & succ)
- Le altre pagine contengono *data entry*, usate solo per guidare le ricerche



Recap: *Indici clustered e non clustered*

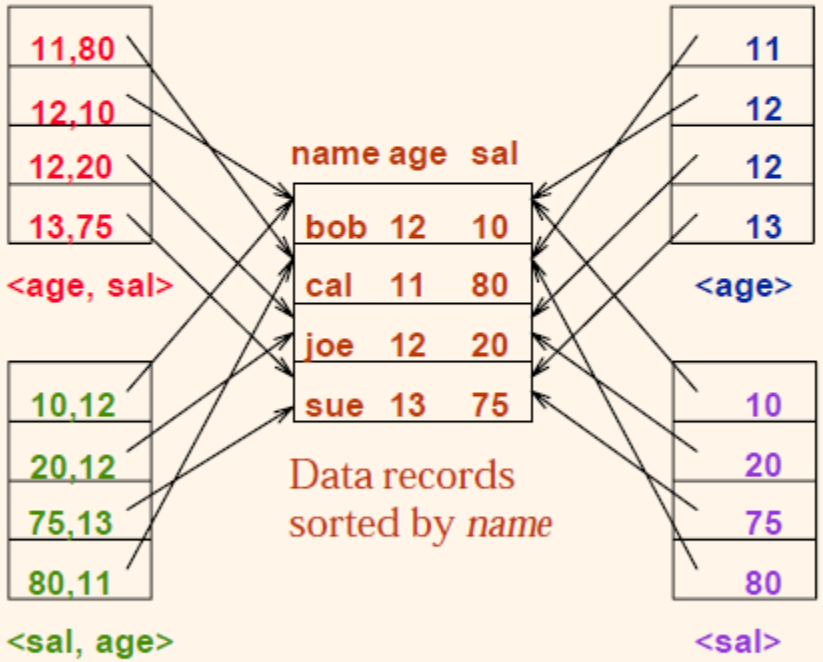
- Supponiamo che venga usata l'Alternativa 2 per le data entry, e che i record di dati siano memorizzati in un file heap
  - Per costruire un indice clustered, prima ordiniamo il file heap (con dello spazio libero su ciascuna pagina per gli inserimenti futuri)
  - Potrebbero essere necessarie delle pagine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



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.



Recap: *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!*

Stima dei costi

- Per ciascun piano considerato, bisogna stimarne il costo:
  - si deve stimare il costo di ciascuna operazione nell'albero del piano
    - Dipende dalla cardinalità dei dati in ingresso
    - Abbiamo già discusso di come stimare il costo delle operazioni (scansione sequenziale, scansione dell'indice, join, etc)
- Si deve anche stimare la dimensione del risultato per ciascuna operazione dell'albero!
  - Usare le informazioni sulle relazioni in ingresso
  - Per le selezioni i per i join, ipotizzare l'indipendenza dei predicati

Stima delle dimensioni e fattori di riduzione

- Interrogazione:

```
SELECT lista-attributi
FROM lista-relazioni
WHERE term1 AND ... AND termk
```

- Massimo numero di tuple nel risultato = prodotto delle cardinalità delle relazioni nella FROM
- Il fattore di riduzione (FR) associato con ciascuno dei term riflette l'impatto del term nella riduzione della dimensione del risultato.
- Cardinalità del risultato = Max num. tuple * $\prod_{i=1, \ldots, k} FR_i$
  - Assunzione implicita che i term siano indipendenti!
  - Il termine *col = valore* ha FR *1/Nchiavi(I)*, dato indice I su *col*
  - Il termine *col1 = col2* ha FR *1 / max(Nchiavi(I1), Nchiavi(I2))*
  - Il termine *col > valore* ha FR

    (max(I) – valore) / (max(I) – min(I))

Schema per gli esempi

- Simile al vecchio schema; è stato aggiunto **`vnome`** per le variazioni
- Prenota:
  - ogni tupla è lunga 40 bytes, 100 tuple per pagina, 1000 pagine
- Velisti:
  - ogni tupla è lunga 50 byte, 80 tuple per pagina, 500 pagine

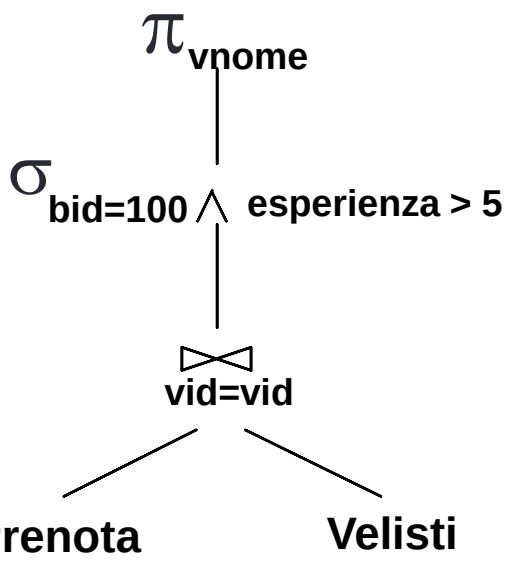
```
Velisti(vid:integer, vnome:string,
                esperienza:integer, età:real)

Prenota(vid:integer, bid:integer, giorno:dates,
                                        pnome:string)
```

Esempio motivante

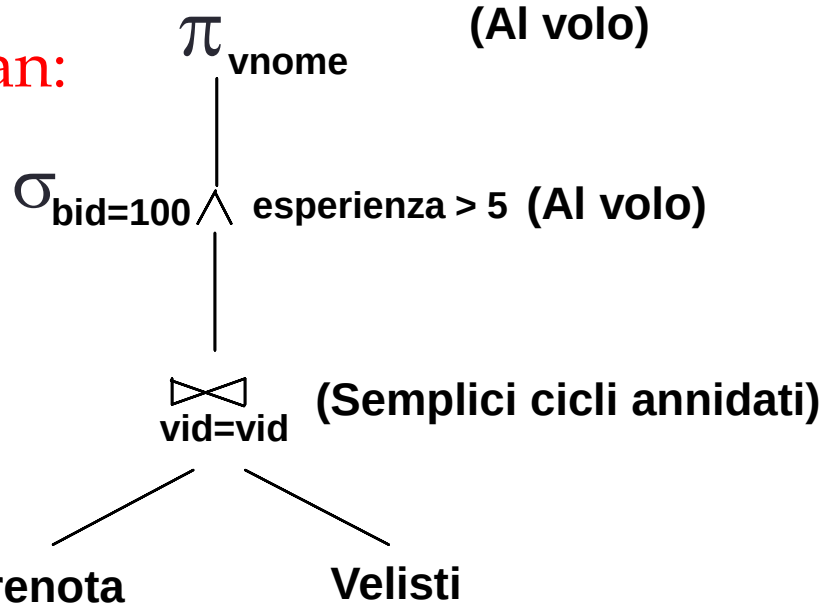
```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

- Costo: 500 + 500 * 1000 I/O
- Assolutamente non è il piano peggiore!
- Non sfrutta diverse opportunità: le selezioni avrebbero potuto essere “spinte” prima, non viene fatto alcun uso degli indici esistenti, etc.
- Scopo dell’ottimizzazione: trovare piani più efficienti per calcolare la stessa risposta

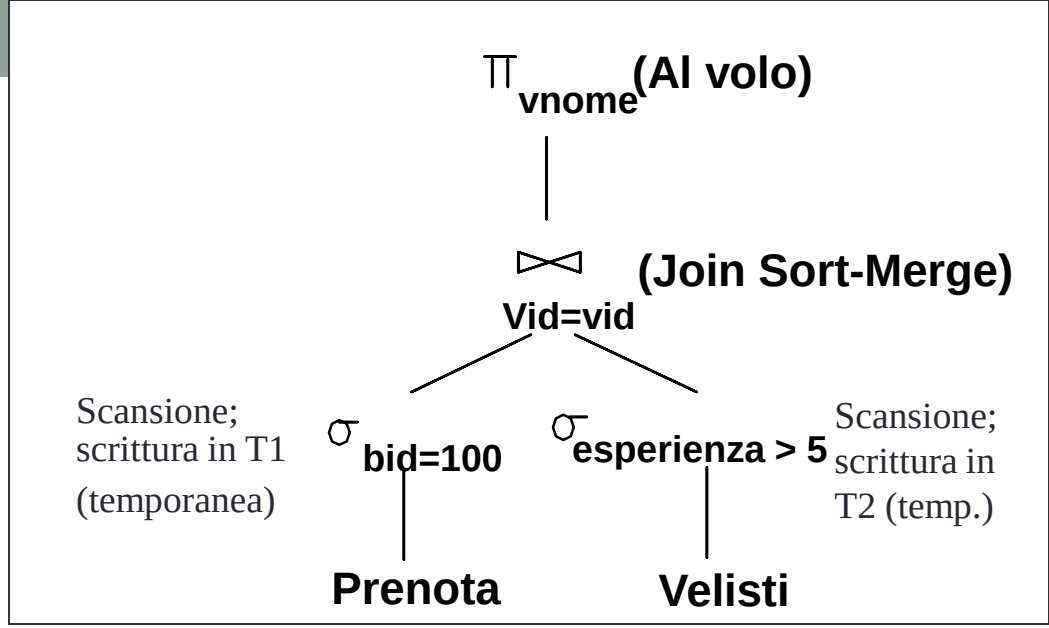
Albero RA:



Plan: (Al volo)



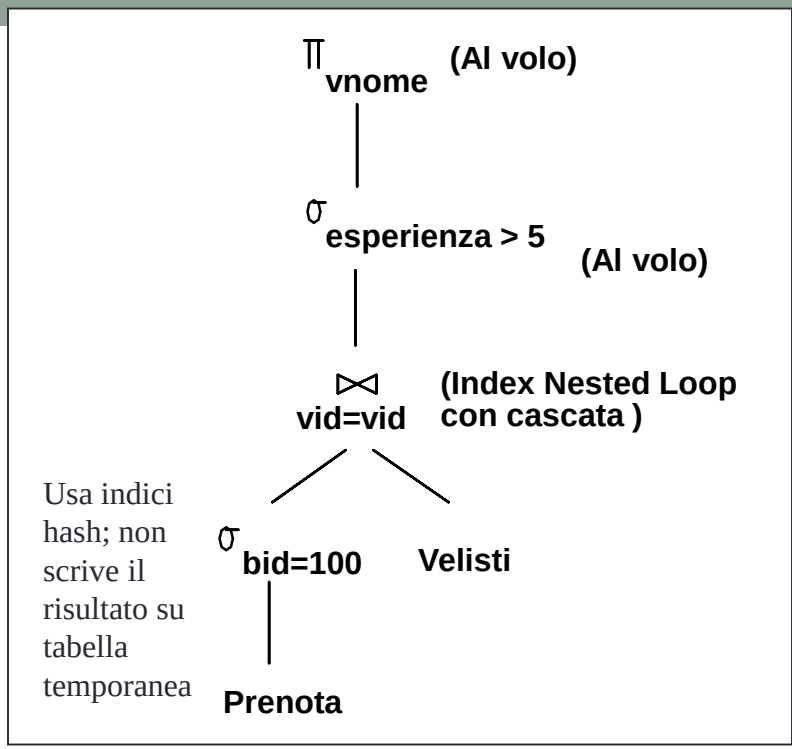
Piani alternativi 1 (niente indici)



- Principale differenza: anticipare le selezioni
- Con 5 buffer, costo del piano:
  - **Scansione di Prenota** (1000) + scrittura temporanea di T1 (10 pagine, se abbiamo 100 barche, distribuzione uniforme)
  - **Scansione di Velisti** (500) + scrittura temporanea di T2 (250 pagine, se abbiamo 10 livelli di esperienza)
  - **Ordinamento di T1** (2 * 2 * 10), **Ordinamento di T2** (2 * 4 * 250),
  - **Fusione** (10 + 250)
  - Totale: **1010**+**750**+**40**+**2000**+**260** = **3560** I/O di pagina
- Se usassimo join BNL, costo del join = **10 + 4 * 250**, costo totale = **1760**+**1010**=**2770**
- Se anticipiamo le proiezioni, T1 ha solo vid, T2 solo vid e vnome:
  - T1 entra in 3 pagine, costo del BNL scende sotto le 250 pagine, totale < **2000**

Piani alternativi 2 con indici

- Con indice *clustered* su bid di *Prenota*, otteniamo 100.000 / 100 barche =
  1000 tuple su 1000 / 100 = 10 pagine
- INL con passaggio in cascata
  (la rel. esterna non viene materializzata)
  - Proiettare via i campi non necessari della relazione esterna non serve
- La colonna di join vid è una chiave per Velisti
  - Al più una tupla rilevante, un indice non clustered su vid è OK
- La decisione di non anticipare esperienza > 5 prima del join è basata sulla disponibilità dell'indice su vid di Velisti
- Costo: Selezione delle tuple di Prenota (10 I/O); per ciascuna, bisogna leggere le corrispondenti tuple di Velisti (1000 * 1.2); totale **1210** I/O



Sommario

- Ci sono diversi algoritmi di valutazione alternativi per ciascun operatore relazionale
- Una interrogazione è valutata convertendola in un albero di operatori e valutando gli operatori dell'albero
- Bisogna capire l'ottimizzazione delle interrogazioni per comprendere pienamente l'impatto sulle prestazioni di un dato progetto di base di dati (relazioni, indici) su un carico di lavoro (insieme di interrogazioni)

Sommario (2)

- Due criteri per ottimizzare una interrogazione:
  - considerare un insieme di piani alternativi
    - bisogna ridurre lo spazio di ricerca; tipicamente, solo piani left-deep
  - stimare il costo di ciascun piano che viene considerato
    - bisogna stimare la dimensione del risultato e il costo per ciascun nodo del piano
    - concetti chiave: statistiche, indici, implementazione degli operatori