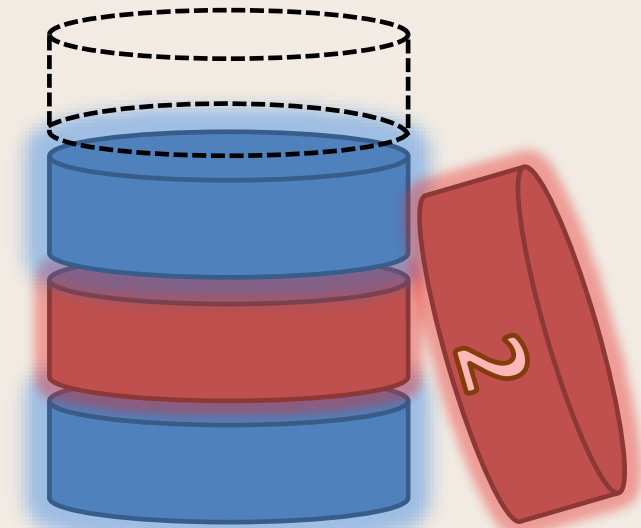


Evaluarea Operatorilor Relaționali



Operatori relaționali

- Selectie (σ) Selectează un subset de înregistrări a unei rel.
- Proiecție (π) Elimină anumite coloane ale relației.
- Join (\otimes) Permite combinarea a două relații.
- Diferență ($-$) Returnează înregistrări aflate într-o relație ce nu se găsesc în a doua.
- Reuniune (\cup) Returnează înregistrări aflate în ambele rel.
- Agregare (SUM, MIN, etc.) și grupare (GROUP BY)

Operatori relaționali

- Tehnici de implementare a operatorilor
 - Iterare
 - Indexare
 - Partiționare

Evaluarea operatorilor relaționali

- Căi de acces

 - = alternative de parcurgere a înregistrărilor

 - Scanare tabelă

 - Parcurgere index

- Selectarea căii de acces

 - Număr de pagini returnate (pagini de index sau ale tablei)

 - Se selectează calea ce minimizează costurile de acces

Structura folosită în exemple

Students (*sid*: integer, *sname*: string, *age*: integer)

Courses (*cid*: integer, *name*: string, *location*: string)

Evaluations (*sid*: integer, *cid*: integer, *day*: date, *grade*: integer)

■ *Students*:

- Fiecare înregistrare are o lungime de 50 bytes.
- 80 înregistrări pe pagină, 500 pagini.

■ *Courses*:

- Lungime înregistrare 50 bytes,
- 80 înregistrări pe pagină, 100 pagini.

■ *Evaluations*:

- Lungime înregistrare 40 bytes,
- 100 înregistrări pe pagină, 1000 pagini.

Implementare *join* bazat pe egalitatea a două câmpuri

```
SELECT  *  
FROM Evaluations R  
INNER JOIN Students S ON R.sid=S.sid
```

\equiv

$R \otimes S$

Produsul cartezian $R \times S$ este în general voluminos. Deci, implementarea prin $R \times S$ urmat de o selecție e ineficientă.

Implementare *join* bazat pe egalitatea a două câmpuri

```
SELECT  *  
FROM Evaluations R  
INNER JOIN Students S ON R.sid=S.sid
```

Notăție: M pagini în R , p_R înregistrări pe pagină, N pagini în S , p_S înregistrări pe pagină.

Implementare *join* bazat pe egalitatea a două câmpuri

```
SELECT  *  
FROM Evaluations R  
INNER JOIN Students S ON R.sid=S.sid
```

Metrica folosită: numărul de pagini citite/salvate (I/Os)

Tehnici de implementare a operatorului *Join*

- Iterare
 - *Simple/Page-Oriented Nested Loops*
 - *Block Nested Loops*
- Indexare
 - *Index Nested Loops*
- Partiționare
 - *Sort Merge Join*
 - *Hash*

Simple Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
```

- Pentru fiecare înregistrare din tabela *externă* R, se scanează întreaga relație *internă* S.

Cost:

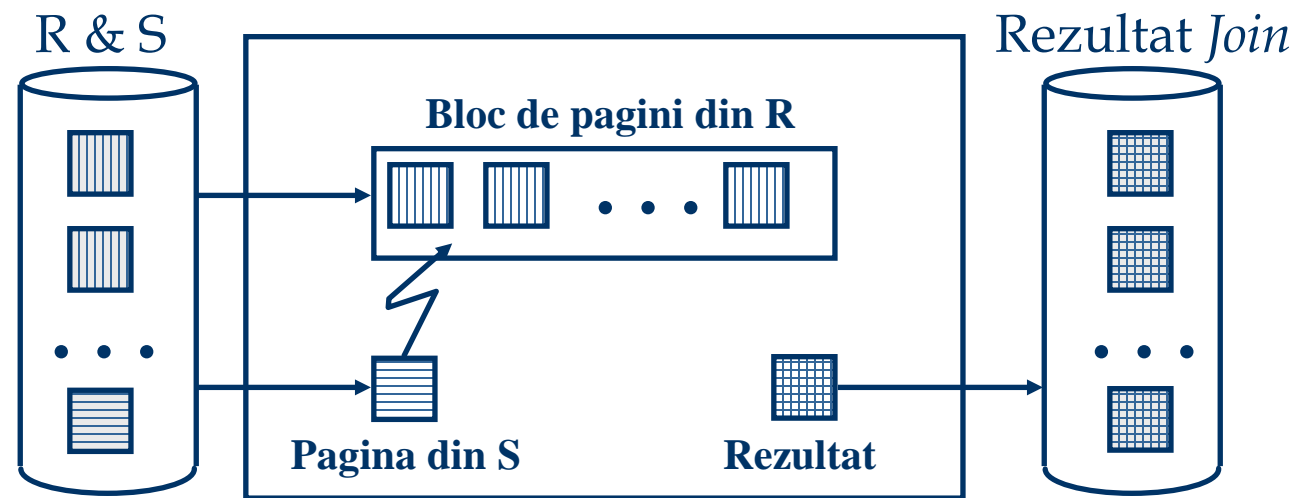
$$M + p_R * M * N = 1000 + 100 * 1000 * 500 \text{ I/Os.}$$

Page Oriented Nested Loops Join

```
foreach page in R do  
    foreach page in S do  
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  to result
```

- Pentru fiecare *pagina* din R, se citește fiecare *pagina* din S, iar perechile de înregistrări $\langle r, s \rangle$ ce verifică expresia $ri=sj$ vor salvate în pagina rezultat, unde r este din pagina lui R iar s este din pagina lui S.
- Cost: $M + M*N = 1000 + 1000*500$ I/Os
- Dacă tabela mai mică (S) este tabela externă, atunci cost = $500 + 500*1000$ I/Os

Block Nested Loops Join



Exemplu pentru Block Nested Loops

Cost:

Scan. tabelă externă + #(blocuri externe) * scan. tabelă internă

#blocuri externe = $\lceil \text{nr de pagini} / \text{dim bloc} \rceil$

- Cu *Evaluations* (R) ca tabelă externă, și bloc de 100 pagini:
 - Cost scanare R este 1000 I/Os; un total de 10 *blocuri*.
 - Pt fiecare bloc din R, se scanează *Students*: 10*500 I/Os.
 - Dacă *bufferul* avea doar 90 pagini libere, S era scanat de 12 ori.
- Cu *Students* (S) ca tabelă externă (bloc de 100 pagini):
 - Cost scanare S este 500 I/Os; un total de 5 *blocuri*.
 - Pt fiecare bloc din S, scanăm *Evaluations*; 5*1000 I/Os.

Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add <r, s> to result
```

- Dacă există un index definit pe coloana de join a unei tabele (ex. S), aceasta poate fi considerată tabelă internă și poate fi exploatat indexul.

Cost: $M + (M * p_R) * \text{cost găsiere în reg. din S}$

Index Nested Loops Join

```
foreach tuple r in R do  
    foreach tuple s in S where  $r_i == s_j$  do  
        add <r, s> to result
```

Cost găsiere înregistrare =
Cost căutare în index +
Cost citire înregistrări

Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add <r, s> to result
```

Cost căutare in index

- aproximativ 1.2 (pentru index cu acces direct),
- 2-4 pentru B-arbore.

Cost citire înregistrări

- Depinde de clusterizare:
 - Index grupat: 1 I/O (*tipic*)
 - Index negrupat: 1 I/O per înregistrare din S (*în cel mai rău caz*)

Exemplu pentru Index Nested Loops

- Index cu acces direct pt. *sid* din *Students*:
 - Scanare *Evaluations*: 1000 pagini I/Os, 100×1000 înreg.
 - Pentru fiecare înreg din *Evaluations*: 1.2 I/Os pentru a localiza intrarea în index, plus 1 I/O pentru a citi (exact o) înreg. din *Students* \Rightarrow cost 220,000. Total: 221,000 I/Os.

Exemplu pentru Index Nested Loops

- Index cu acces direct pt. *sid* din *Evaluations*:
 - Scanare *Students*: 500 pagini I/Os, 80×500 înreg.
 - Pentru fiecare înreg din *Students*: 1.2 I/Os pentru a localiza intrarea în index, plus costul citirii înreg. din *Evaluations*. Presupunem o distribuție uniformă a notelor, deci 2.5 note per student ($100,000 / 40,000$). Costul citirii lor e 1 sau 2.5 I/Os (index grupat sau nu). Total: de la 88,500 la 148,500 I/Os

Sort-Merge Join ($R \bowtie_{i=j} S$)

- Ordonare R și S după câmpurile ce apar în condiția de join, apoi scanare pentru identificarea perechilor.
 - Scanarea lui R avansează până r_i curent $>$ s_j curent, apoi se avansează cu scanarea lui S până s_j curent $>$ r_i curent; până când r_i curent $=$ s_j curent.
 - La acest punct toate perechile posibile între înregistrările din R cu aceeași valoare r_i și toate înregistrările din S cu aceeași valoare s_j sunt salvate în pagina specială pentru rezultat.
 - Apoi se reia scanarea lui R și S.
- R este scanat o dată; fiecare grup de înregistrări din S este scana pentru fiecare înregistrare “potrivită” din R .

Exemplu pentru Sort-Merge Join

<i>sid</i>	<i>sname</i>	<i>age</i>
22	dustin	20
28	yuppy	21
31	johnny	20
44	guppy	22
58	rusty	21

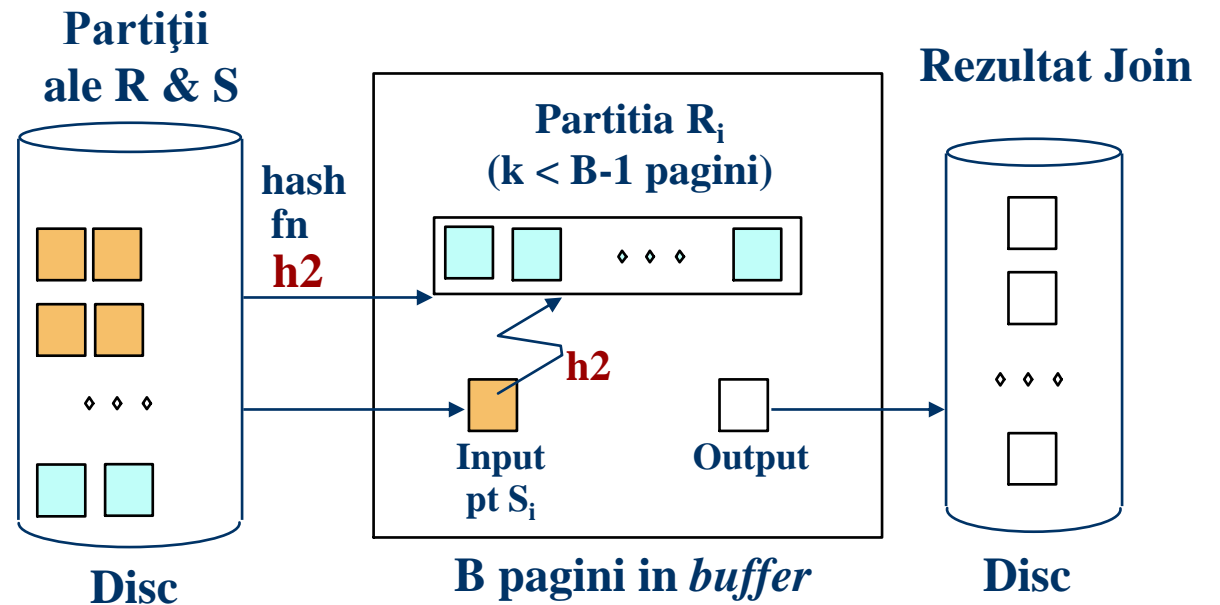
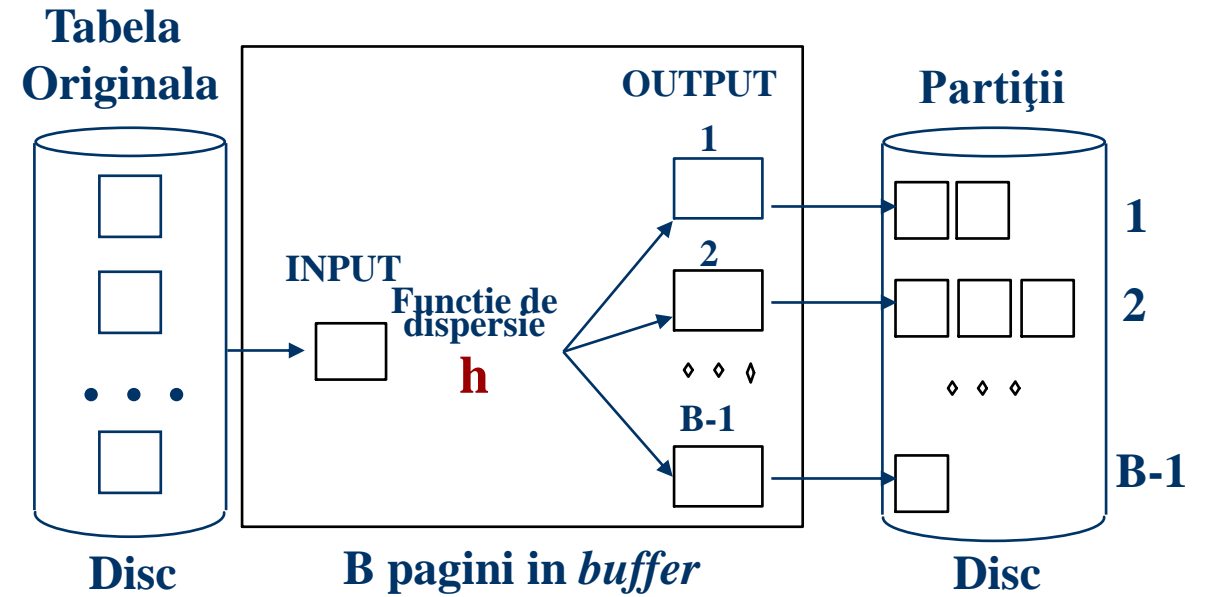
<i>sid</i>	<i>cid</i>	<i>day</i>	<i>grade</i>
28	101	15/6/04	8
28	102	22/6/04	8
31	101	15/6/04	9
31	102	22/6/04	10
31	103	30/6/04	10
58	101	16/6/04	7

- Cost: $M \log_2 M + N \log_2 N + (M+N)$
 - Costul scanării este $M+N$ (poate fi $M*N$ – f rar!)
- Cu 35, 100 sau 300 pagini în *buffer*, *Evaluations* și *Students* pot fi sortate în 2 treceri. Cost total: 7500.

Rafinare algoritm Sort-Merge Join

- Se poate combina faza de interclasare din *sortarea* lui R și S cu faza de scanare pentru join.
 - Având $B > \sqrt{L}$, unde L este numărul de pagini a celei mai mari tabele, și folosind optimizarea algoritmului de sortare (ce produce subșiruri inițiale sortate de lungime $2B$), numărul de subșiruri pentru fiecare relație este $< B/2$.
 - Alocând o pagină pentru câte un subșir al fiecărei relații, se va verifica expresia de join dintr-o singură trecere.
 - **Cost**: citire+salvare fiecare tabelă la Pas 0 + citire fiecare tabelă o dată pentru comparare (+ scriere rezultat).
 - În exemplu, costul coboară de la 7500 la 4500 I/Os.
- În practică, costul alg. *sort-merge join*, (la fel ca cel al sortării externe), este *liniar*.

Hash-Join



Observații asupra Hash-Join

- Vrem ca numărul de partiții $k < B-1$, și $B-2 >$ dimensiunea celei mai mari partiții.
 - dacă $B > \sqrt{M}$ condiția este îndeplinită
- Tabelă de dispersie (performanță)
- Dacă sunt partiții ce nu încap în memoria internă \rightarrow *hash-join* recursiv

Costul Hash-Join

- $3(M+N)$ I/Os.
- Sort-Merge Join vs. Hash Join:
 - *Hash Join* e superior dacă dimensiunea tablelor diferă f mult si este paralelizabil.
 - *Sort-Merge Join* e mai puțin sensibil la modificări de dimensiune a datelor; rezultatul este sortat.