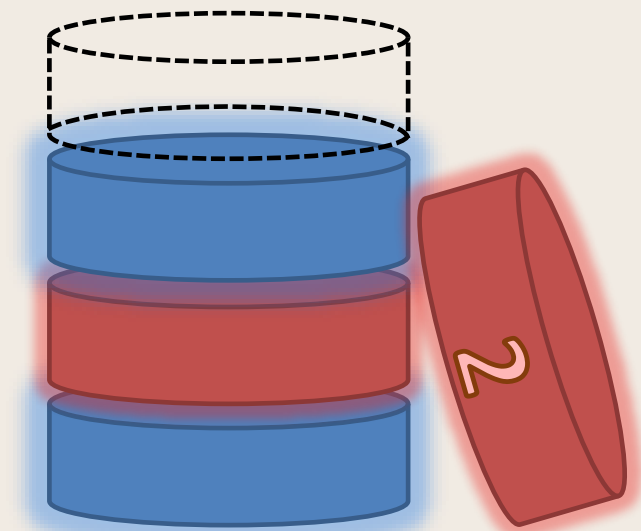


Optimizarea Interogărilor



Operatori specifici mulțimilor

- Intersecția și produsul cartezian sunt cazuri particulare de join.
- Reuniunea (*Union*) și diferența (*Except*) sunt similare
- Abordarea reuniunii bazată pe sortare:
 - Se sortează ambele tabele (folosind toate câmpurile).
 - Tabelele sortate sunt interclasate.
 - *Alternativă*: Se interclasează subșirurile sortate ale *ambelor* tabele obținute la primul pas al sortării.
- Abordarea reuniunii bazată pe funcție de dispersie:
 - Partiționarea tabelelor folosind funcția de dispersie h .
 - Pentru fiecare partiție a uneia dintre tabele, se folosește o a doua funcție de dispersie (h_2), utilizată la determinarea duplicărilor în partițiile corespunzătoare din R.

Operatori de agregare (SUM, AVG, MIN etc.)

■ Fără grupare:

- În general, necesită scanarea completă a tabelului.
- Având un index cu cheia de căutare ce include toate câmpurile din SELECT sau WHERE, se poate scana doar indexul.

■ Cu grupare:

- Sortarea atributelor din *group-by*, apoi scanarea tabelului și agregarea rezultatelor pentru fiecare grup. (Abordarea poate fi îmbunătățită prin combinarea sortării cu agregarea)
- Abordare similară bazată pe dispersie câmpurilor din *group-by*
- Având un index ce include toate câmpurile din SELECT, WHERE și GROUP BY, se poate realiza doar o scanare a sa; dacă attributele din *group-by* formează prefixul cheii de căutare a indexului, rezultatul va conține înregistrările în ordonate după valorile acestor attribute.

Impactul *Buffer*-ului

- Dacă anumite operații se execută concurent, estimarea numărului de pagini disponibile în *buffer* este dificil de făcut.
- Abordările de evaluare a operatorilor ce presupun acces repetat la câmpurile unei tabele pot interacționa cu politica *buffer*-ului de înlocuire a paginilor.
 - de exemplu, tabela internă este scanată în mod repetat la *Simple Nested Loop Join*. Dacă sunt suficiente pagini în *buffer* pentru a stoca tabela internă, politica *buffer*-ului nu afectează performanța. În caz contrar însă, MRU (*Most Recently Used*) este cea mai potrivită politică, LRU (*Least Recently Used*) fiind mai ineficientă (*sequential flooding*).

Executarea interogărilor distribuite

Interogări distribuite

```
SELECT AVG(E.age)
FROM Employees E
WHERE E.salary > 3000
      AND E.salary < 7000
```

Fragmentare orizontală:

Înregistrările cu *salary* < 5000 la Shanghai și *salary* >= 5000 la Tokyo.

- Se calculează SUM(*age*), COUNT(*age*) pe ambele servere
- Dacă WHERE conține doar *E.salary* > 6000, interogarea se poate executa pe un singur server.

Interogări distribuite

```
SELECT AVG(E.age)
FROM Employees E
WHERE E.salary > 3000
      AND E.salary < 7000
```

Fragmentare verticală:

title și *salary* la Shanghai, *ename* și *age* la Tokyo, *id* fiind prezent pe ambele servere.

- Trebuie reconstruită tabela prin intermediul unui *join* pe *id*, iar apoi se evaluează interogarea.

Replicare:

Tabela *Employees* e copiată pe ambele servere.

- Alegerea site-ului pe care se execută interogarea se face în funcție de costurile locale și costurile de transfer.

Join-uri distribuite



Employees

500 pagini,

80 înregistrări pe pagină

Reports

1000 pagini

100 înregistrări pe pagină

Fetch as Needed

- *Page-oriented nested loops, Employees* ca tabelă externă (pentru fiecare pagină din *Employees* se aduc toate paginile din *Reports* de la Paris):
 - **Cost:** $500 D + 500 * 1000 (D+S)$, unde **D** este costul de citire/salvare a paginilor; **S** costul de transfer al paginilor.
 - Dacă interogarea nu s-a lansat de la Londra, atunci trebuie adăugat costul de transfer al rezultatului către clientul care a transmis interogarea.
- Se poate utiliza și INL (*Indexed Nested Loops*) la Londra, aducând din tabela *Reports* doar înregistrările ce se potrivesc.

Ship to One Site

- Transferă tabela *Reports* la Londra
 - **Cost:** 1000 S + 4500 D (*Sort-Merge Join*; cost = $3 \cdot (500 + 1000)$)
 - Dacă dimensiunea rezultatului este mare, ambele relații ar putea fi transferate către serverul ce a inițiat interogarea iar join-ul este implementat acolo
- Transferă tabela *Employees* la Paris
 - **Cost:** 500 S + 4500 D

Semijoin

- La **Londra** se execută proiecția tablei *Employees* pe câmpul (câmpurile) folosite în join și rezultatul se transferă la Paris.
- La **Paris** se execută join între proiecția lui *Employees* și tabela *Reports*.
 - Rezultatul se numește **reducția** lui *Reports* relativ la *Employees* .
- Se transferă reducția lui *Reports* la Londra
- La **Londra**, se execută join între *Employees* și reducția lui *Reports*.

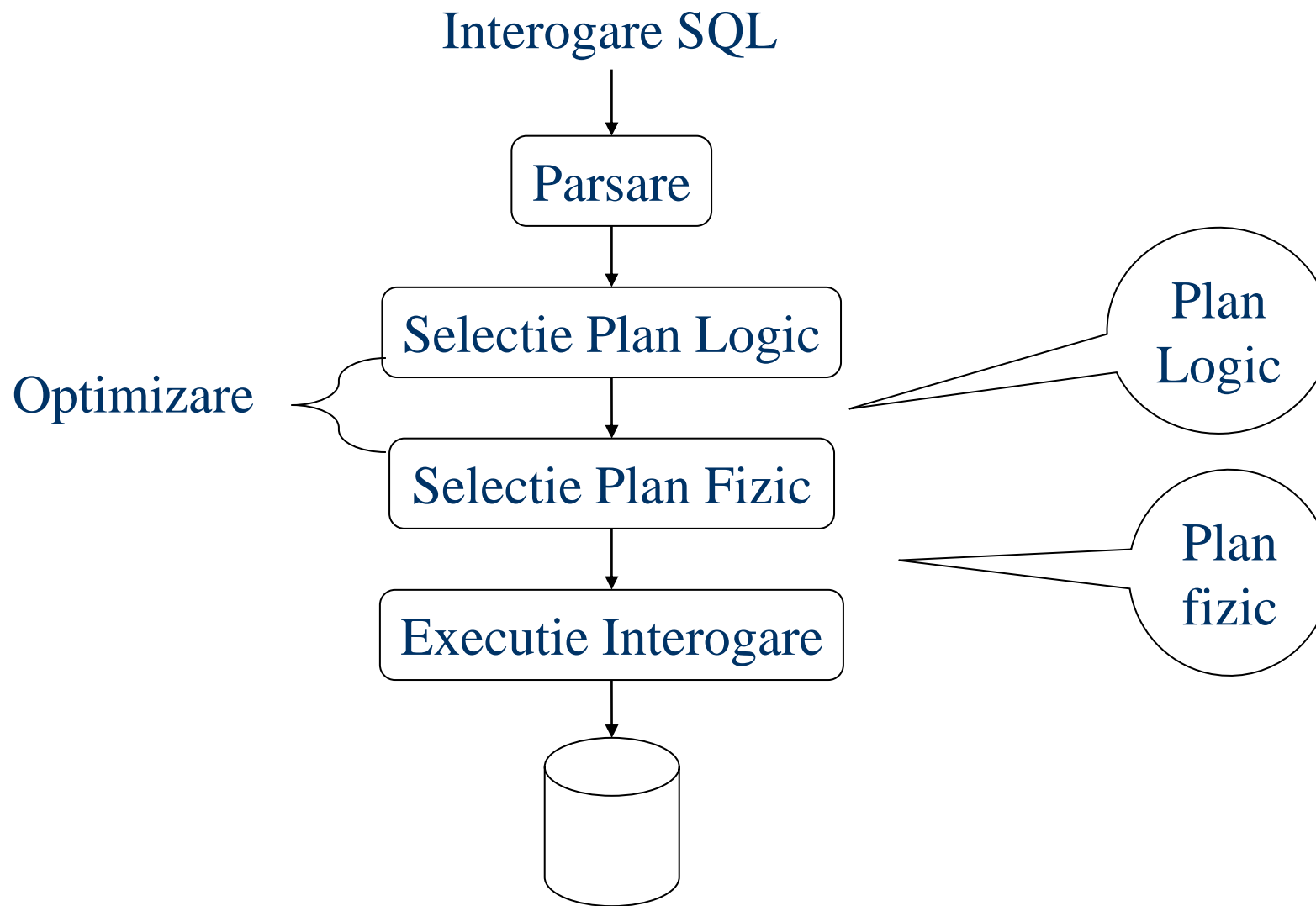
Bloomjoin

- La **Londra** se construiește un vector de biți de dimensiune k :
 - Folosind o funcție de dispersie, se împart valorile câmpului de join în partiții de la 0 la $k-1$.
 - Dacă funcția aplicată câmpului returnează i , se setează bitul i cu 1 (i de la 0 la $k-1$).
 - Se transferă vectorul de biți la *Paris*.

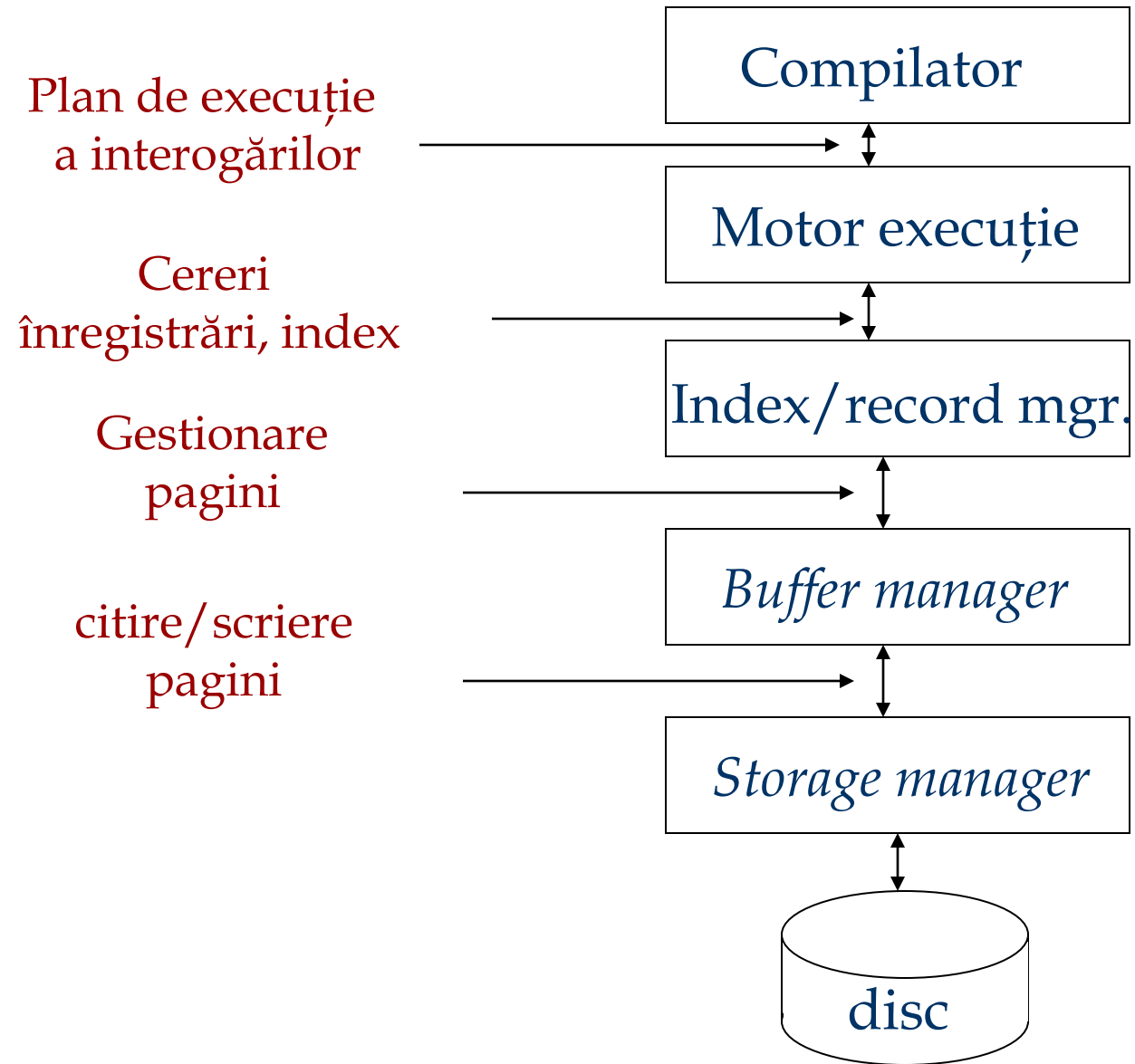
Bloomjoin (cont)

- La **Paris**, folosim similar funcția de dispersie. Dacă pentru un câmp se obține un i căruia în vector îi corespunde 0 , se elimină acea înregistrare din rezultat
 - Rezultatul se numește **reducție** a tabelii *Reports* în funcție de *Employees*.
- Se transferă reducția la Londra.
- La **Londra** se face join-ul dintre *Employees* și varianta redusă a lui *Reports*.

Optimizarea interogărilor



Executarea interogărilor



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:*

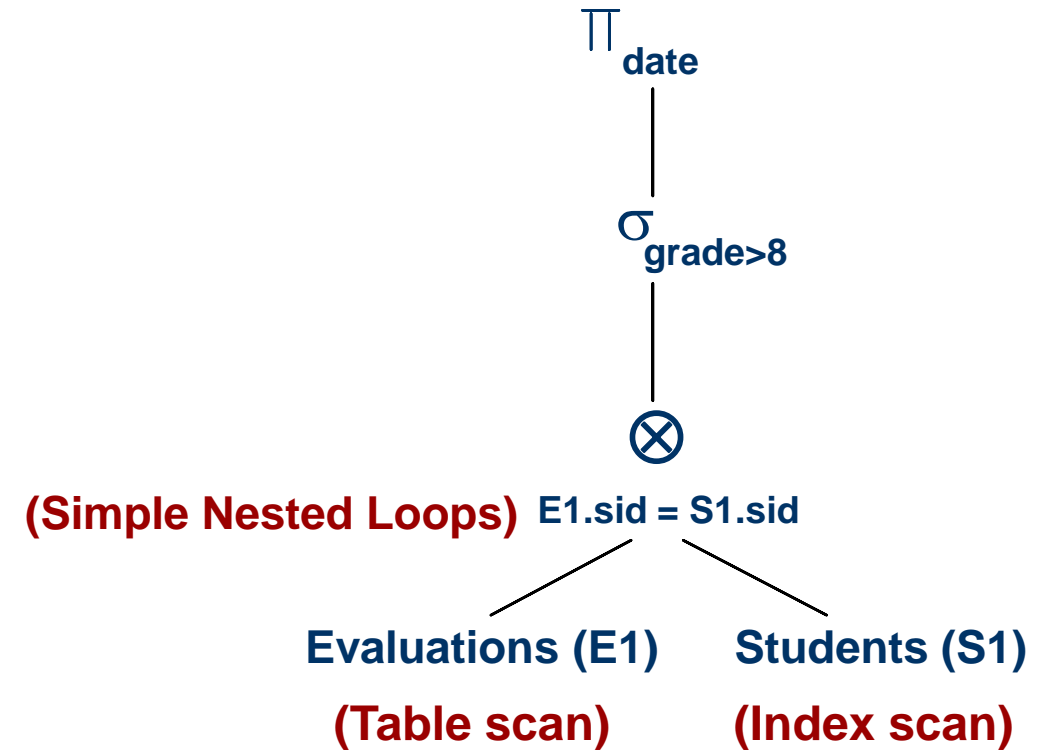
- Lungime înregistrare 40 bytes,
- 100 înregistrări pe pagină, 1000 pagini.

Planurile de execuție ale interogărilor

```
SELECT E1.date
FROM Evaluations E1, Students S1
WHERE E1.sid=S1.sid AND
      E1.grade > 8
```

Planul interogării:

- arbore logic
- se specifică o decizie de implementare la fiecare nod
- planificarea operațiilor



Frunzele planului de execuție: scanări

- **Table scan**: iterează prin înregistrările tabelului.
- **Index scan**: accesează înregistrările index-ului tabelului
- **Sorted scan**: accesează înregistrările tabelului după ce aceasta a fost sortată în prealabil.
- Cum se combină operațiile?
 - **Modelul iterator**. Fiecare operație e implementată cu 3 funcții:
 - *Open*: inițializări / pregătește structurile de date
 - *GetNext*: returnează următoarea înregistrare din rezultat
 - *Close*: finalizează operația / eliberează memoria

=> permite lucrul în pipeline!
 - **Modelul materializat (*data-driven*)**
 - Uneori modul de operare a ambelor modele e identic (exemplu: *sorted scan*).

Proces de optimizare a interogărilor

- Se transformă interogarea SQL într-un arbore logic:
 - identifică blocurile distincte (*view-uri*, sub-interogări).
- Se rescrie interogarea:
 - se aplica **transformări algebrice** pentru a obține un plan mai puțin costisitor.
 - se unesc blocuri de date și/sau se mută predicate între blocuri.
- Se optimizează fiecare bloc: **secvențele de execuție a join-urilor**.

Proces de optimizare a interogărilor

- Plan: *Arbore format din operatori algebrici relaționali*
 - Pentru fiecare operator este identificat un algoritm de execuție
 - Fiecare operator are (în general) implementată o interfață `pull`.
- Probleme:
 - Ce planuri se iau în considerare?
 - Cum se estimează costul unui plan?
- Se implementează algoritmi de identificare a planurilor cele mai puțin costisitoare:
 - **Ideal**: se dorește obținerea celui mai bun plan.
 - **Practic**: se elimină planurile cele mai costisitoare!

System R Optimizer

- Impact:
 - Cel mai utilizat algoritm;
 - Functionează bine pentru < 10 *join*-uri.
- **Estimare cost:** aproximări, aproximări, aproximări...
 - Statistici, actualizate în catalogul bazei de date, folosite la estimarea costului operațiilor și a dimensiunii rezultatelor.
 - Combinație între costul CPU și costurile de citire/scriere.

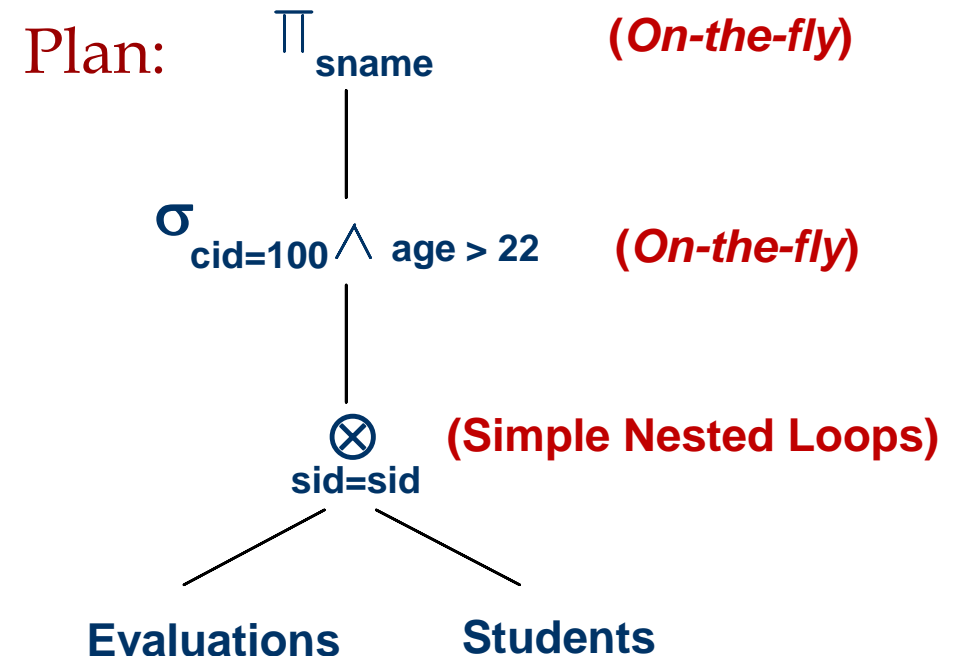
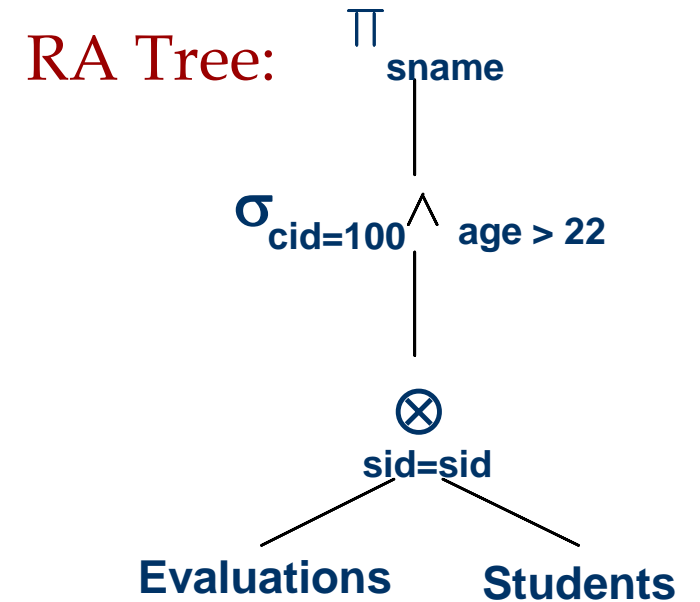
System R Optimizer

- Nu se estimeaza toate planurile!
 - Sunt considerate doar planurile *left-deep join*
 - Aceste planuri permit ca rezultatul unui operator sa fie transferat în *pipeline* către următorul operator fără stocarea temporară a relației.
 - Se exclude produsul cartezian

Exemplu

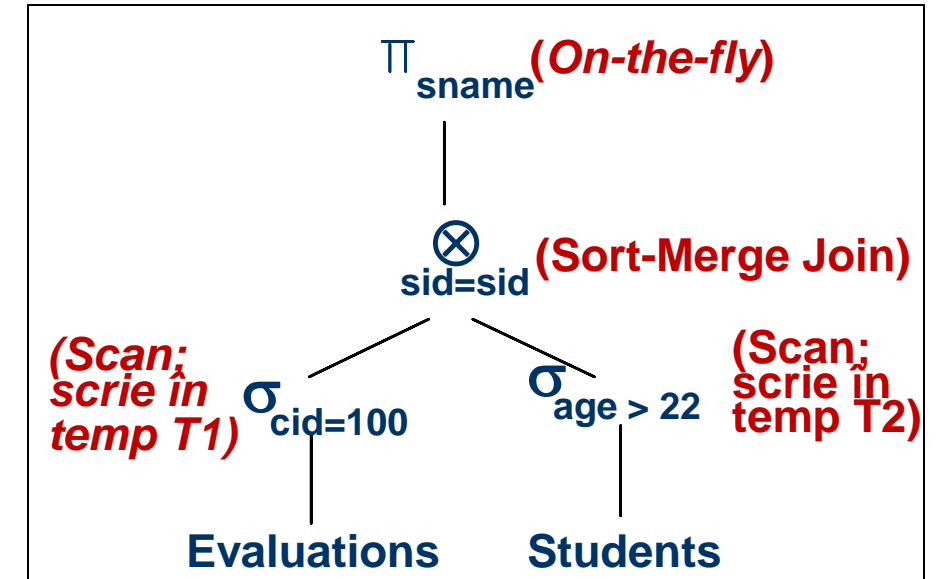
```
SELECT S.sname
FROM Evaluations E, Students S
WHERE E.sid=S.sid AND
      E.cid=100 AND S.age>22
```

- Cost: 500+500*1000 I/Os
- Plan inadecvat, cost f mare!
- *Scopul optimizarii*: căutarea de planuri mai eficiente ce calculează același răspuns.



Plan alternativ 1

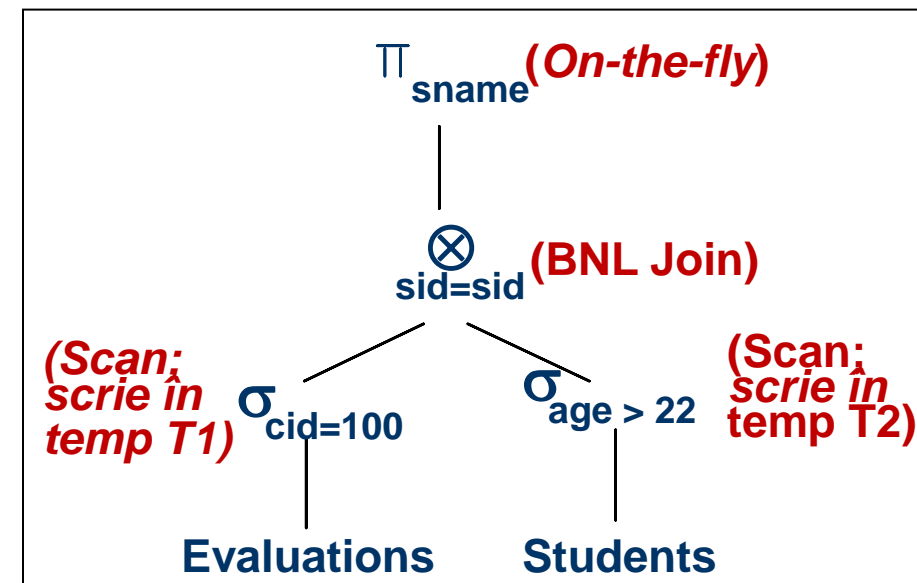
- *Diferența esențială:*
poziția operatorilor de selecție.



- **Costul planului**
(presupunem că sunt 5 pagini în buffer):
 - Scan *Evaluations* (1000) + memorează temp T1 (10 pag, dacă avem 100 cursuri și distribuție uniformă). – *total 1010 I/Os*
 - Scan *Students* (500) + memorează temp T2 (250 pag, dacă avem 10 vârste). – *total 750 I/Os*
 - Sortare T1 ($2 \cdot 2 \cdot 10$), sortare T2 ($2 \cdot 4 \cdot 250$), interclasare ($10 + 250$) – *total 2300 I/Os*
 - Total: 4060 pagini I/Os.

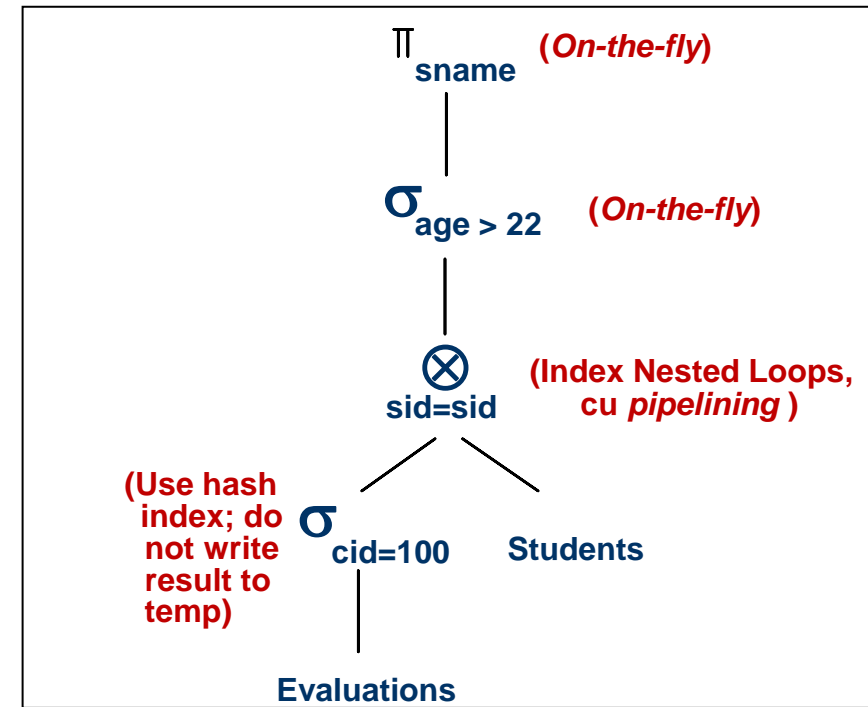
Plan alternativ 1

- *Diferența esențială:*
poziția operatorilor de selecție.
- Costul planului
(presupunem că sunt 5 pagini în buffer):
- Dacă se folosește **BNL join**:
 - cost join = $10 + 4 * 250$,
 - cost total = 2770.
- Dacă `împingem` proiecțiile:
 - T1 rămâne cu *sid*, T2 rămâne cu *sid* și *sname*:
 - T1 încapă în 3 pagini, costul BNL este sub 250 pagini, total < 2000.



Plan alternativ 2

- Cu index grupat cu access direct pe *cid* din *Evaluations*, avem
 $100,000/100 = 1000$ tupluri în
 $1000/100 = 10$ pagini.
 - INL cu *pipelining* (rezultatul nu e materializat).
 - Se elimină câmpurile inutile din output.
 - Coloana *sid* e cheie pentru *Students*.
 - Cel mult o "potrivire", index grupat pe *sid* e OK.
 - Decizia de a nu „împinge” selecția *age>22* mai repede e dată de disponibilitatea indexului pe *sid* al *Students*.
- Cost:** Selecție pe *Evaluations* (10 I/Os); pentru fiecare obținem înregistrările din *Students* ($1000 \cdot (1.2 + 1)$); total **2210 I/Os**.



Unitatea de optimizare: *bloc Select*

- O interogare SQL este descompusă într-o colecție de *blocuri Select* care sunt optimizate separat.
- *Blocurile Select* imbricate sunt de obicei tratate ca apeluri de subrutine (câte un apel pentru fiecare înregistrare din blocul *Select* extern)

```
SELECT S.sname  
FROM Students S  
WHERE S.age IN  
      (SELECT MAX (S2.age)  
       FROM Students S2  
       GROUP BY S2.sname)
```

Bloc extern *Bloc imbricat*

Pentru fiecare bloc, planurile considerate sunt:

- Toate metodele disponibile de acces, pt fiecare tabelă din FROM
- Toate planurile *left-deep join trees* (adică, toate modurile secvențiale de *join* ale tabelelor, considerând toate permutările de tabele posibile)

Estimarea costului

- Se estimează costul fiecărui plan considerat:
 - Trebuie *estimat costul* fiecărui operator din plan
 - Depinde de cardinalitatea tabelelor de intrare
 - Modul de estimarea al costurilor a fost discutat în cursurile precedente (scanare tabele, join-uri, etc.)
 - Trebuie *estimată dimensiunea rezultatului* pentru fiecare operație a arborelui!
 - Se utilizează informații despre relațiile de intrare
 - Pentru selecții și join-uri, se consideră predicatele ca fiind independente.
- Algoritmul **System R**
 - Inexact, dar cu rezultate bune în practică.
 - În prezent există metode mai sofisticate

Echivalențe în algebra relațională

- Permite alegerea unei ordini diferite a join-urilor și 'împingerea' selecțiilor și proiecțiilor în fața join-urilor.
- Selectii: $\sigma_{c1 \wedge \dots \wedge cn}(R) \equiv \sigma_{c1}(\dots(\sigma_{cn}(R)))$ (Cascadă)
 $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (Comutativitate)
- Proiecții: $\pi_{a1}(R) \equiv \pi_{a1}(\dots(\pi_{an}(R)))$ (Cascadă)
- Join:
 $R \otimes (S \otimes T) \equiv (R \otimes S) \otimes T$ (Asociativitate)
 $(R \otimes S) \equiv (S \otimes R)$ (Comutativitate)
 $\rightarrow R \otimes (S \otimes T) \equiv (T \otimes R) \otimes S$

Alte echivalențe

- A proiecție se comută doar cu o selecție ce utilizează câmpurile ce apar în proiecție.
- Selecția dintre câmpurile ce aparțin tabelelor implicate într-un produs cartezian convertește produsul cartezian într-un *join*.
- O selecție doar pe attributele lui R comută cu $R \otimes S$. (adică, $\sigma(R \otimes S) \equiv \sigma(R) \otimes S$)
- Similar, dacă o proiecție urmează unui join $R \otimes S$, putem să o ‘împingem’ în fața join-ului păstrând doar câmpurile lui R (și S) care sunt necesare pentru join sau care apar în lista proiecției.

Enumerarea planurilor alternative

- Sunt luate în considerare două cazuri:
 - Planuri cu o singură tabelă
 - Planuri cu tabele multiple
- Pentru interogările ce implică o singură tabelă, planul conține o combinație de selecturi, proiecții și operatori de agregare:
 - Sunt considerate toate metodele de acces și este păstrată cea cu cel mai mic cost estimat.
 - Mai mulți operatori sunt executați deodată (în *pipeline*)

Estimări de cost pentru planuri bazate pe o tabelă

- Indexul I pt cheia primară implicată într-o selecție:
 - Costul e $Height(I)+1$ pt un arbore B+, sau $1.2+1$ pt hash index.
- Index grupat I pe câmpurile implicate în una sau mai multe selecții:
 - $(NPages(I)+NPages(R)) * produs\ al\ FR\ pt\ fiecare\ selecție$
- Index negrupat I pe câmpurile implicate în una sau mai multe selecții:
 - $(NPages(I)+NTuples(R)) * produs\ al\ FR\ pt\ fiecare\ selecție.$
- Scanare secvențială a tablei:
 - $NPages(R).$

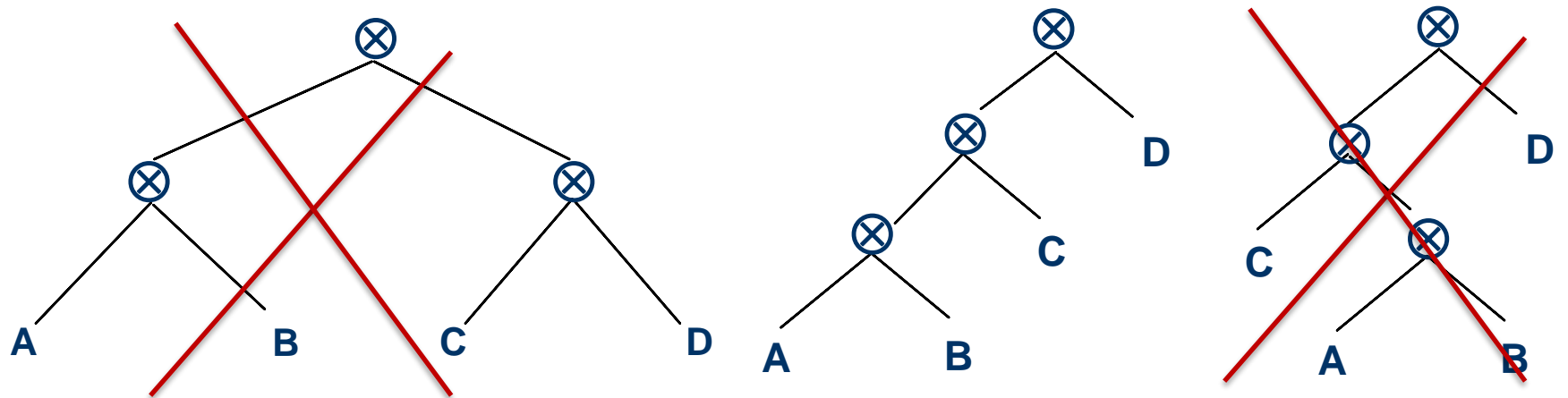
Exemplu

```
SELECT S.sid  
FROM Students S  
WHERE S.age=20
```

- Dacă există **index** pt. *age*:
 - $(1/NKeys(I)) * NTuples(R) = (1/10) * 40000$ înreg. returnate
 - **Index grupat**: $(1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500)$ pagini returnate.
 - **Index negrupat**: $(1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000)$ pagini returnate.
- Dacă se **scanează** tabela :
 - Sunt citite toate paginile (500).

Interogări pe tabele multiple

- System R consideră doar arborii left-deep join.
 - Pe măsură ce numărul de join-uri crește , numărul de planuri alternative este tot mai semnificativ; *este necesară restricționarea spațiului de căutare*.
 - Arborii *left-deep* ne permit generarea tuturor planurilor ce suportă *pipeline* complet.
 - Rezultatele intermediare nu sunt salvate în tabele temporare.
 - Nu toți arborii *left-deep* suportă *pipeline* complet (ex. *SM join*).



Enumerarea planurilor *left-deep*

- Planurile *left-deep* diferă prin ordinea tabelelor, metoda de acces a fiecărei table și metoda utilizată pentru implementarea fiecărui *join*.
- N pași de dezvoltare a planurilor cu N table:
 - **Pas 1:** Găsirea celui mai bun plan cu o tabelă pentru fiecare tabelă.
 - **Pas 2:** Găsirea celei mai bune variante de join al rezultatului unui plan cu o tabelă și altă tabelă (*Toate planurile bazate pe 2 table*)
 - **Pas N:** Găsirea celei mai bune variante de join al rezultatului unui plan cu N-1 table și altă tabelă . (*Toate planurile bazate pe N table*)

Enumerarea planurilor *left-deep*

- Pentru fiecare submulțime de relații, se reține:
 - Cel având costul cel mai redus, plus
 - Planul cu cel mai mic cost pentru fiecare *ordonare interesantă* a înregistrărilor.
- **ORDER BY**, **GROUP BY** și **agregările** sunt tratate la final, folosind planurile ordonate sau un operator de sortare adițional.
- Un plan bazat pe N-1 tabele nu se combină cu o altă tabelă dacă nu există o condiție de join între acestea (și dacă mai există predicate nefolosite în clauza WHERE)
 - adică se **evită produsul cartezian**, dacă se poate.
- În ciuda restrângerii mulțimii de planuri considerate, numărul acestora crește **exponențial** cu numărul tabelelor implicate

Exemplu

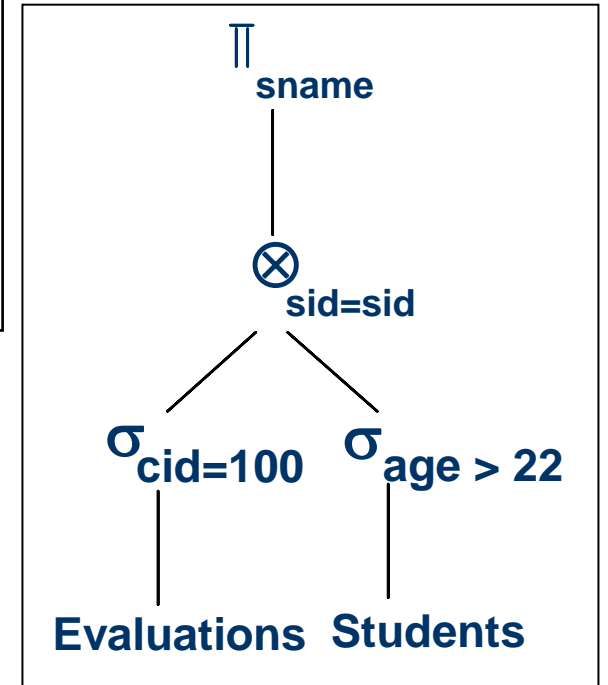
Students:

Arbore B+ pt *age*

Hash pt *sid*

Evaluations:

Arbore B+ pt *cid*



Pas 1:

- *Students*: Utilizarea arborelui B+ pentru selecția *age > 22* este cea mai puțin costisitoare. Totuși, dacă rezultatul va avea multe înregistrări și indexul nu este clusterizat, scanarea tabelului poate fi mai avantajoasă.

- Se preferă arborele B+ (deoarece rezultatul e ordonat după *age*).

- *Evaluations*: Utilizare arborelui B+ pentru selecția *cid=100* este cea mai avantajoasă.

Pas 2:

- Se consideră fiecare plan rezultat la Pas 1, și se combină cu cea de-a doua tabelă. (ex: se folosește indexul Hash pe *sid* pentru selectarea înregistrărilor din *Students* ce satisfac condiția de join)

Interogări imbricate

- *Blocurile Select imbricate sunt optimizate independent, înregistrarea curentă a blocului Select extern furnizând date pentru o condiție de selecție.*
- Blocul extern e optimizat ținând cont de costul `apelului' blocului imbricat.
- Ordonarea implicită a acestor blocuri implică faptul că anumite strategii nu vor fi considerate. *Versiunile neimbricate ale unei interogări sunt (de obicei) optimizate mai bine.*

```
SELECT S.sname
FROM Students S
WHERE EXISTS
  (SELECT *
   FROM Evaluations E
   WHERE E.cid=103
   AND E.sid=S.sid)
```

Bloc Select de optimizat:

```
SELECT *
FROM Evaluations E
WHERE E.cid=103
AND S.sid= valoare ext
```

Interogare simplă echivalentă :

```
SELECT S.sname
FROM Students S,
Evaluations E
WHERE S.sid=E.sid
AND E.cid=103
```

Optimizarea interogărilor distribuite

- Abordare bazată pe cost; similară optimizării centralizate, se consideră toate planurile, alegându-se cel mai ieftin.
 - Diferență 1: Trebuie considerate costurile de transfer.
 - Diferență 2: Trebuie respectată autonomia locală a siteului.
 - Diferență 3: Se considera metode noi de join in context distribuit.
- Este creat un plan global, cu planurile local sugerate de fiecare site.
 - Dacă un site poate îmbunătăți planul local sugerat, este liber să o facă.