

# 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*).

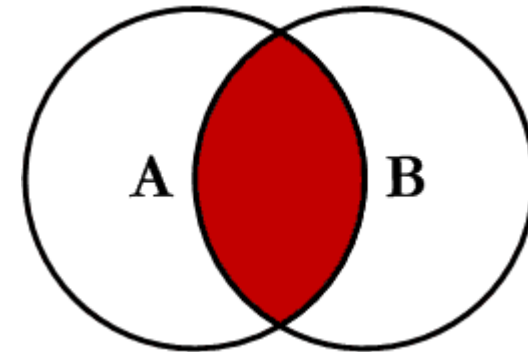
# Joins - Recapitulare

- Inner Join
- Left Join
- Right join
- Outer join

# Inner join

- Este cel mai simplu, cel mai înțeles Join și este cel mai comun.
- Interogarea va returna toate înregistrările din tabelul din stânga (tabelul A) care au o înregistrare care se potrivește în tabelul din dreapta (tabelul B).
- SQL

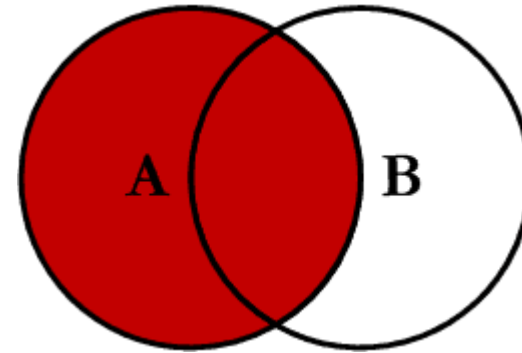
```
SELECT <select_list>
FROM Table_A A
INNER JOIN Table_B B
ON A.Key = B.Key
```



# Left Join

- Interogarea va returna toate înregistrările din tabelul din stânga (tabelul A), indiferent dacă oricare dintre aceste înregistrări are o potrivire în tabelul din dreapta (tabelul B). De asemenea, va returna orice înregistrări care se potrivesc din tabelul din dreapta.

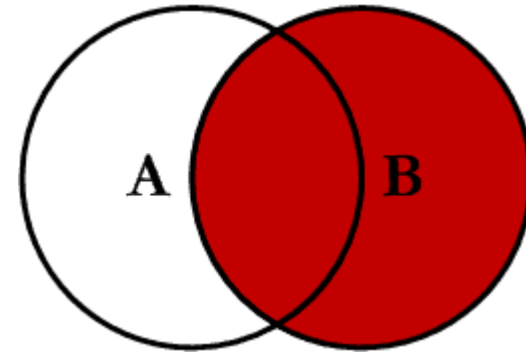
```
SELECT <select_list>  
FROM Table_A A  
LEFT JOIN Table_B B  
ON A.Key = B.Key
```



# Right Join

- Interogarea va returna toate înregistrările din tabelul din dreapta (tabelul B), indiferent dacă oricare dintre acele înregistrări are o potrivire în tabelul din stânga (tabelul A). De asemenea, va returna orice înregistrări care se potrivesc din tabelul din stânga.

```
SELECT <select_list>  
FROM Table_A A  
RIGHT JOIN Table_B B  
ON A.Key = B.Key
```

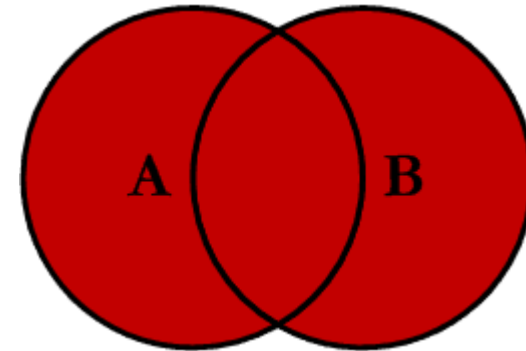




# Outer Join

- Este denumit și FULL OUTER JOIN sau FULL JOIN. Această interogare va returna toate înregistrările din ambele tabele, unind înregistrările din tabelul din stânga (tabelul A) care se potrivesc cu înregistrările din tabelul din dreapta (tabelul B).

```
SELECT <select_list>  
FROM Table_A A  
FULL OUTER JOIN Table_B B  
ON A.Key = B.Key
```



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*;  $\text{cost} = 3 \cdot (500 + 1000)$ )
  - 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*.



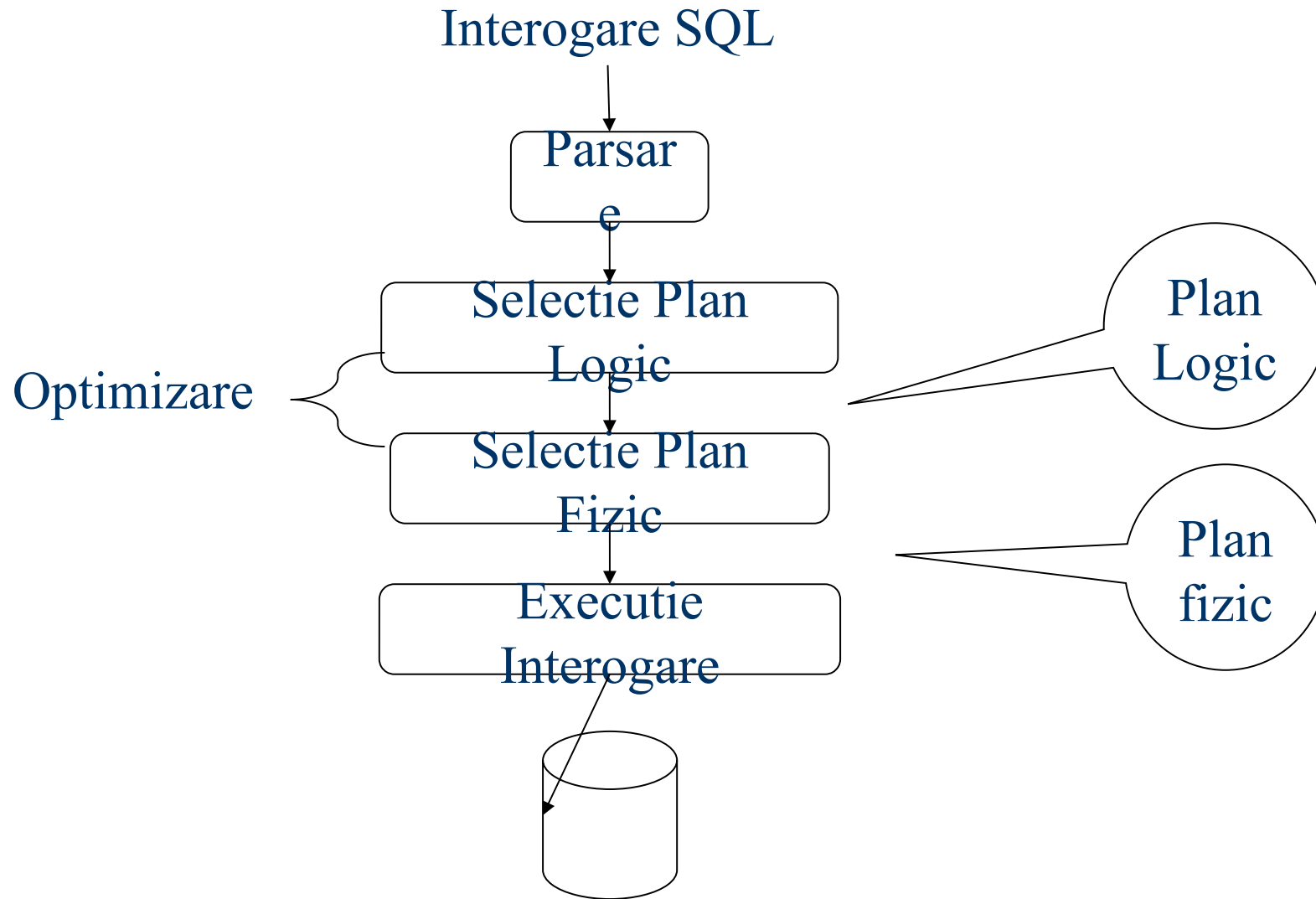
# Bloomjoin

- 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*.

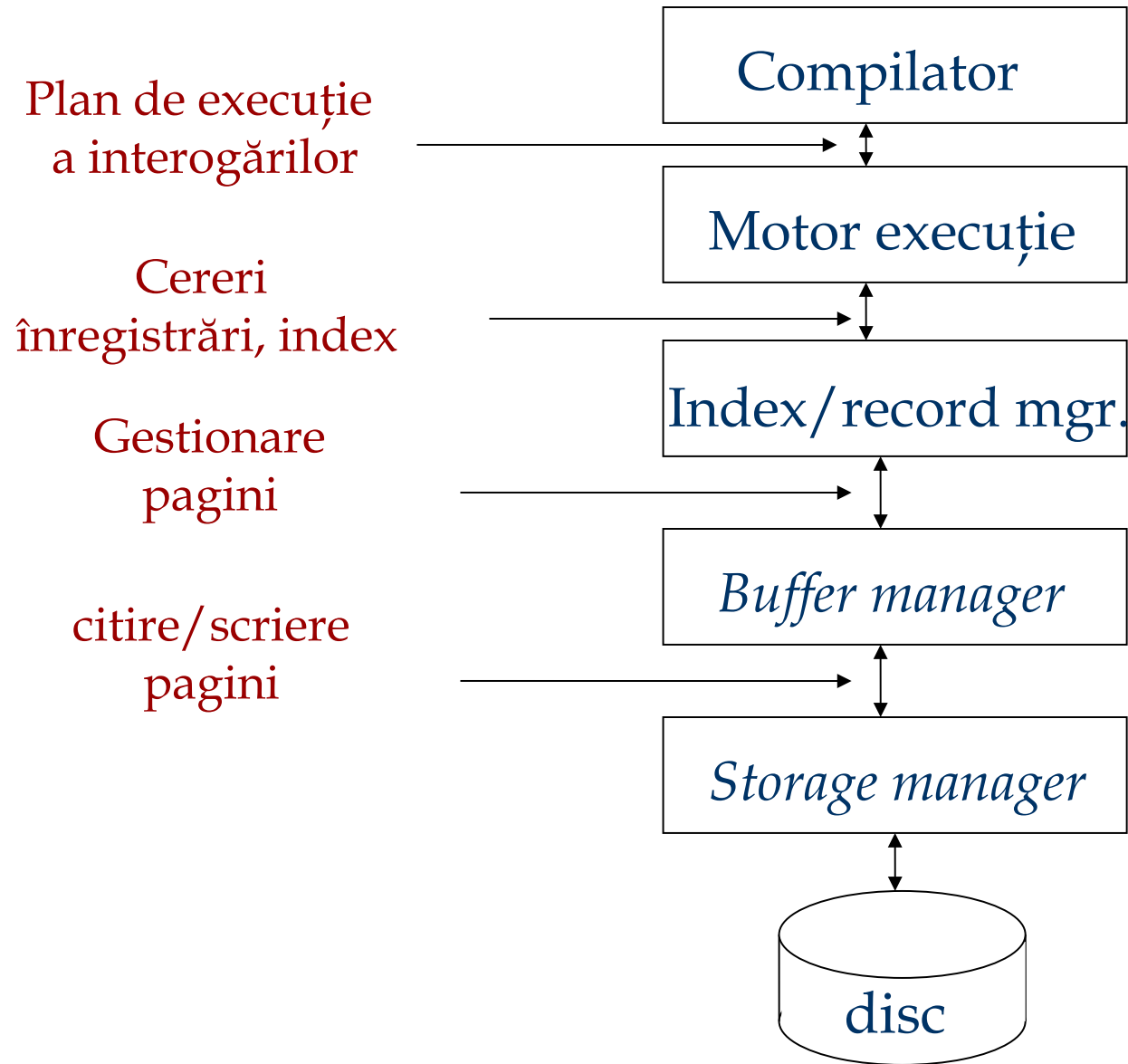
## Bloomjoin (cont)

- La **Paris**, folosim similar funcția de dispersie. Dacă pentru un câmp se obține un  $i$  căruia în vector ii 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:*

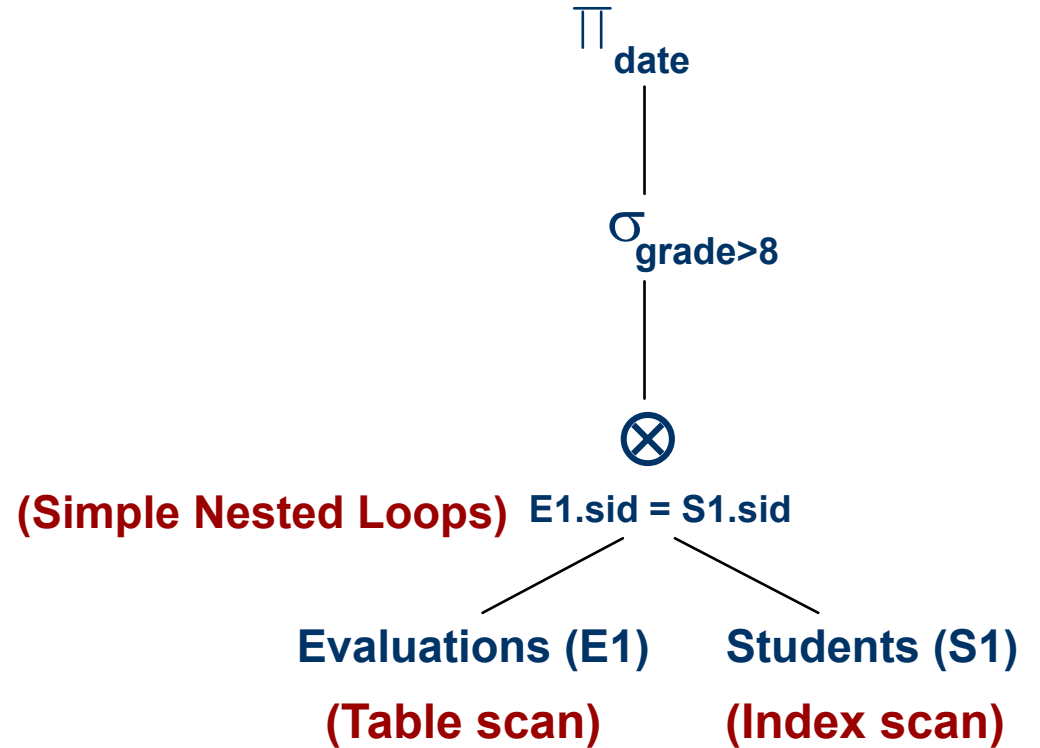
- Lungime înregistrare 40 bytes,
- 100 înregistrări pe pagină, 1000 pagini.

# 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.

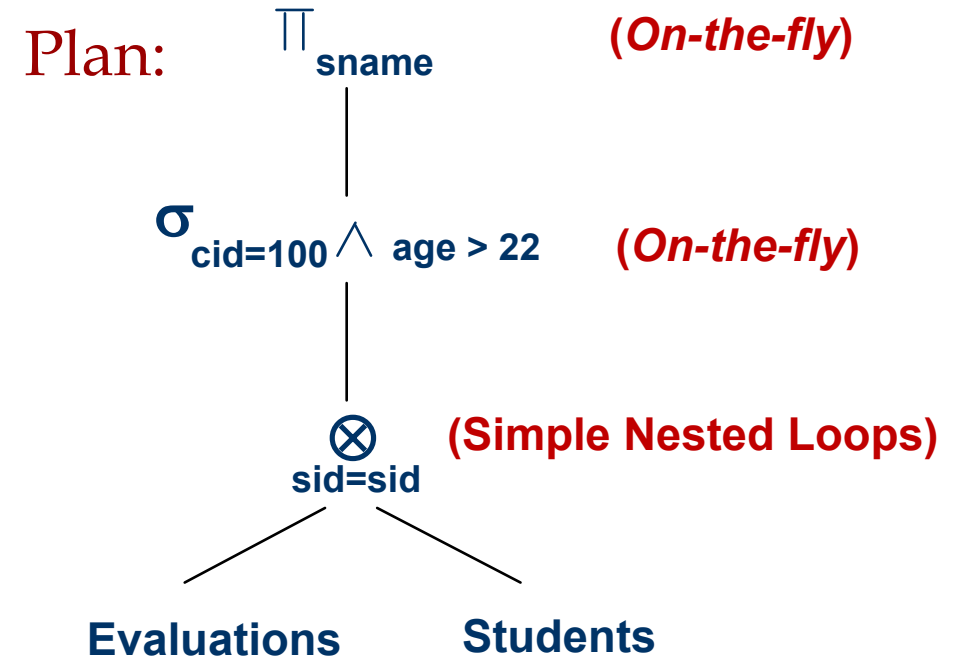
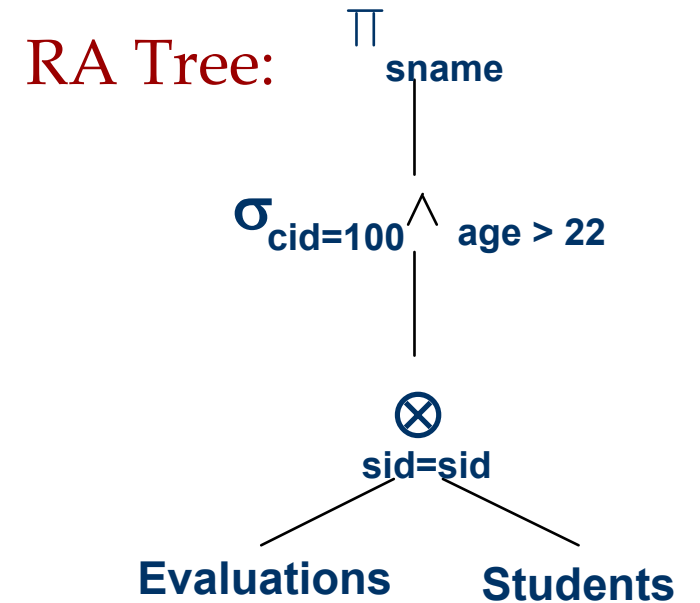
# System R Optimizer

- 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

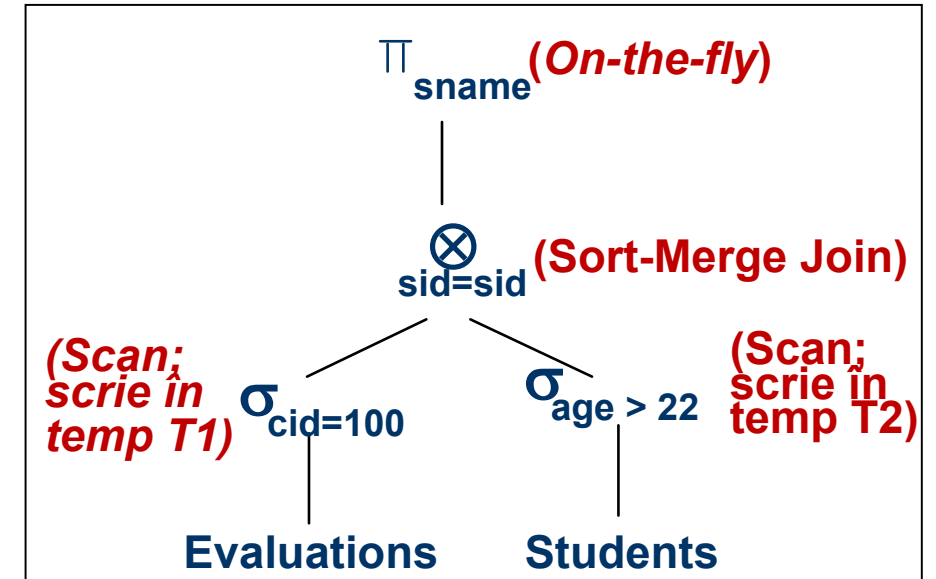
- **Diferența esențială:**

poziția operatorilor de selecție.

- **Costul planului**

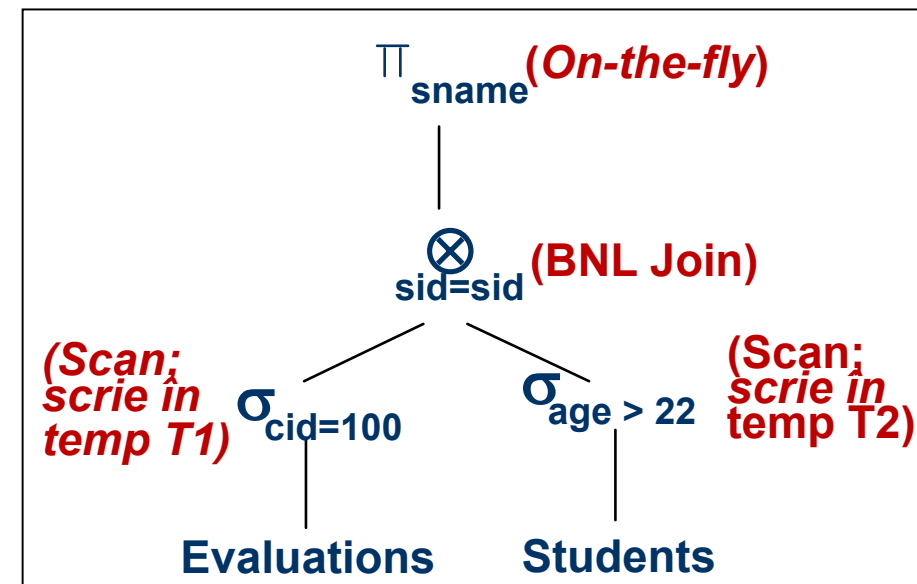
(presupunem că sunt 5 pagini în buffer):

- 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\*2\*10), sortare T2 (2\*4\*250), interclasare (10+250) – *total 2300 I/Os*
- Total: 4060 pagini I/Os.



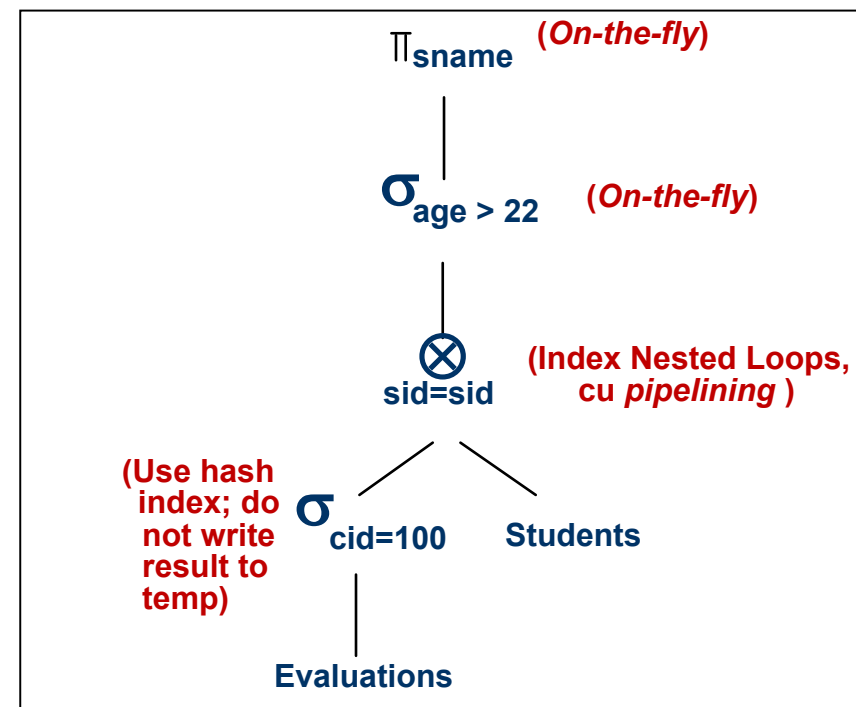
# Plan alternativ 1

- **Diferența esențială:**  
poziția operatorilor de selecție.
- Costul planului  
(presupunem că sunt 5 pagini în buffer):
- Dacă se folosește BNL join:
  - cost join =  $10 + 4 \cdot 250$ ,
  - 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

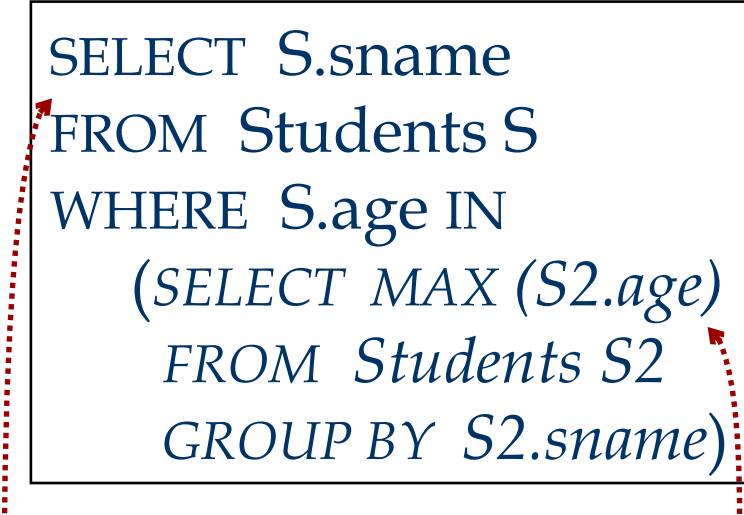
- Cu index grupat cu access direct pe *cid* din *Evaluations*, avem  
 $100,000/100 = 1000$  tupluri în  
 $1000/100 = 10$  pagini.
  - INL cu *pipelining* (rezultatul nu e materializat).
    - Se elimină câmpurile inutile din output.
  - Coloana *sid* e cheie pentru *Students*.
    - Cel mult o "potrivire", 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.
- Selecții:  $\sigma_{c_1 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\dots(\sigma_{c_n}(R)))$  (Cascadă)  
 $\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$  (Comutativitate)
- Proiecții:  $\pi_{a_1}(R) \equiv \pi_{a_1}(\dots(\pi_{a_n}(R)))$  (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:
  - $NPages(R).$

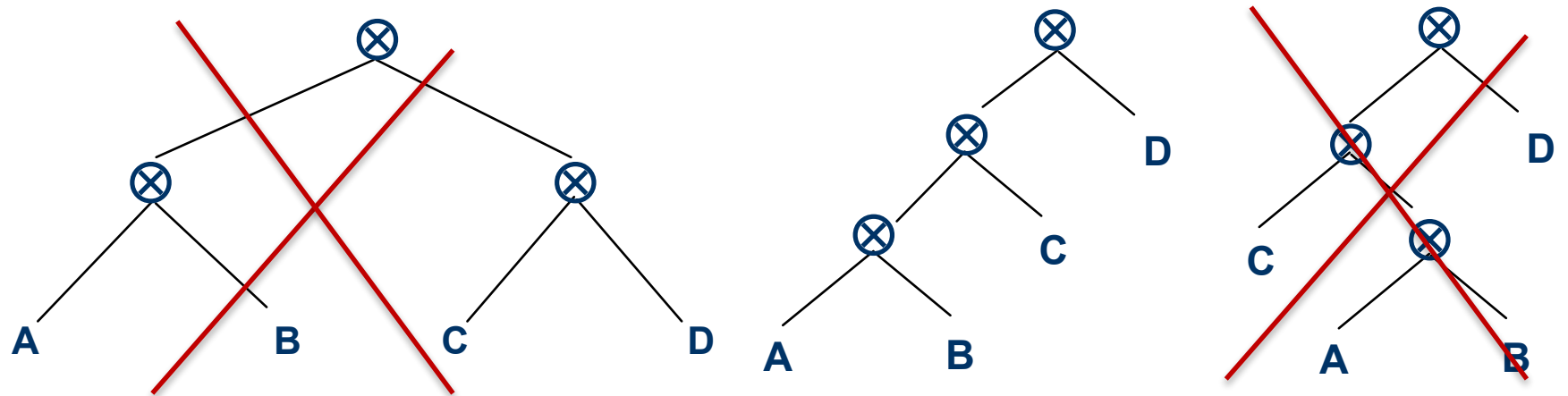
# Exemplu

```
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 tablele și metoda utilizată pentru implementarea fiecărui *join*.
- N pași de dezvoltare a planurilor cu N tablele:
  - **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 tablele*)
  - **Pas N:** Găsirea celei mai bune variante de join al rezultatului unui plan cu N-1 tablele și altă tabelă . (*Toate planurile bazate pe N tablele*)

# 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

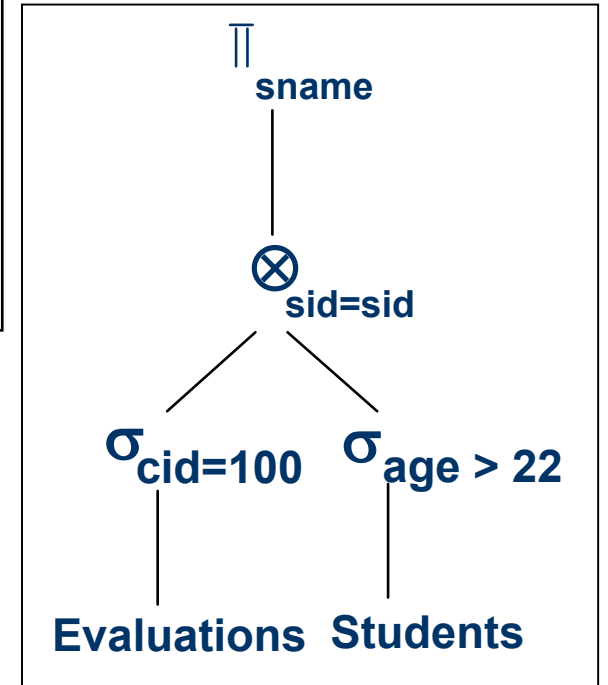
Students:

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.

# Modalitati de verificare a optimizarilor

- Ca proces de optimizare pe oracle - awr reports
- Motiv: pot sa fie nu la nivel de query individual (cat si de cate ori se executa acel query)
- Microsoft - activity monitor care e 'real time'

# Exemplu – același rezultat dar query plan diferit

```
/* SELECT statement folosind un subquery. */  
SELECT [Name] FROM Production.Product  
  
WHERE ListPrice =  
    (SELECT ListPrice FROM Production.Product WHERE [Name] = 'Chair' );  
  
GO /* SELECT statement folosind un join. */  
SELECT Prd1.[Name] FROM Production.Product AS Prd1  
  
JOIN Production.Product AS Prd2 ON (Prd1.ListPrice = Prd2.ListPrice)  
WHERE Prd2.[Name] = 'Chair'; GO
```

# Nume coloane in subqueries / nivele de imbricare

- Regula generală este că numele coloanelor dintr-o declarație sunt implicit calificate de tabelul la care se face referire în clauza FROM la același nivel. Dacă o coloană nu există în tabelul la care se face referire în clauza FROM a unei subinterogări, aceasta este implicit calificată de tabelul la care se face referire în clauza FROM a interogării externe.
- Ce returneaza urmatorul query daca as avea coloanele LastName atat in tabelul Person cat si in tabelul Employee? Cum se comporta BusinessEntityID?

```
SELECT LastName, FirstName FROM Person p
WHERE BusinessEntityID IN
    (SELECT BusinessEntityID FROM Employee e
    WHERE BusinessEntityID IN
        (SELECT BusinessEntityID FROM SalesPerson sp) );
```



# Optimizari JOIN

Modificari la structura:

- **Filtre:** Aplicarea clauzei WHERE clauses *inaintea* join-ul poate reduce nr de inregistrari aduse
- **Subqueries vs. Joins:** rescrierea queries, preferabil nu se folosesc subqueries (e.g., exceptie using EXISTS instead of JOIN).
- **ON vs WHERE**
- **Join Order**
  - Optimizatorii rearanjează adesea ordinea JOINURILOR pentru performanță.
  - In unele cazuri pot alege si o oordine mai putin optima
  - Pot fi “ghidati” folosind **JOIN hints**.

# Example

- orders(id, custId, productId, quantity)
- customers(id, name, region)
- products(id, name, price)

SELECT \*

FROM orders, customers, products

WHERE orders.custId = customers.id  
AND orders.productId = products.id  
AND customers.region = "ÉU";

SELECT

o.id, c.name AS customer\_name,  
p.name AS product\_name,  
o.quantity, p.price

FROM orders o

JOIN customers c ON o.custId = c.id  
JOIN products p ON o.productId = p.id

WHERE c.region = "EU";

# Explicatii:

- Utilizează JOIN-uri explicite, care sunt mai lizibile și mai ușor de întreținut.
- Selectează doar coloanele necesare.
- Filtrează devreme folosind clauza WHERE, reducând rândurile la începutul planului de execuție.
- Permite optimizerului să utilizeze potențial indici pentru custId, productId și regiune.

- Obțineți venitul total per client în UE, arătând doar clienții al căror venit total este peste 1.000 USD.

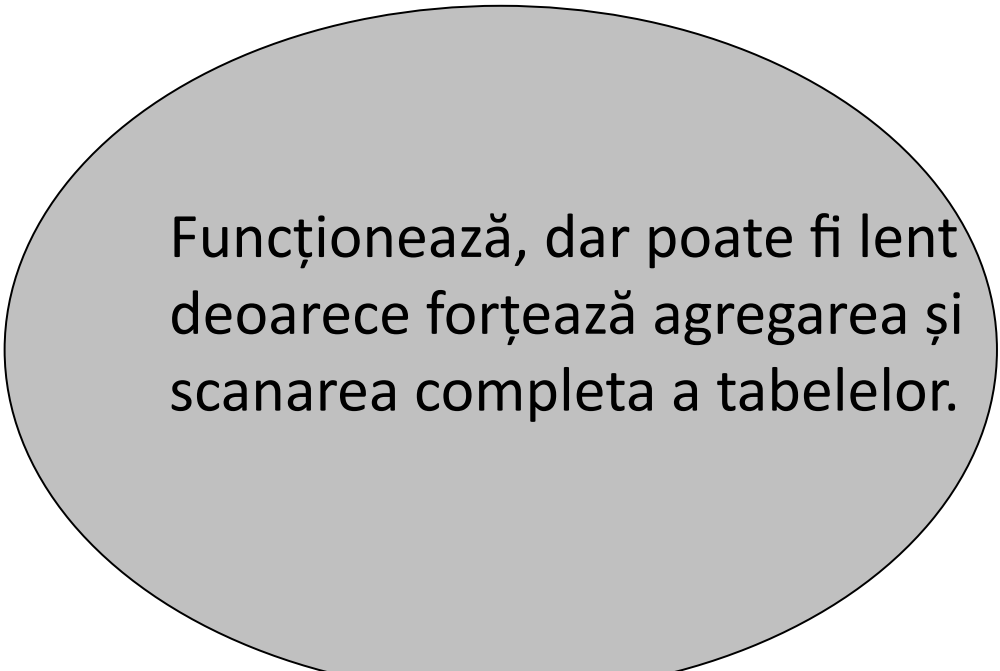
```
SELECT c.name AS custName, customer_revenue.total_revenue
FROM (
    SELECT o.custId, SUM(o.quantity * p.price) AS total_revenue
    FROM orders o JOIN products p ON o.productId = p.id
    GROUP BY o.custId
) AS customer_revenue

JOIN customers c ON customer_revenue.custId = c.id
WHERE c.region = "ÉU"
    AND customer_revenue.total_revenue > 1000;
```

# Inner join vs Right/Left Join

- Use Case: Gasiti lista clientilor care nu au comenzi de cumparare
  - orders(id, cust\_id, product\_id, quantity)
  - customers(id, name, region)

```
SELECT c.custId, c.name  
FROM customers c  
LEFT JOIN orders o ON c.custId = o.custId  
GROUP BY c.custId, c.name  
HAVING COUNT(o.orderId) = 0;
```



Funcționează, dar poate fi lent deoarece forțează agregarea și scanarea completa a tabelelor.

# Inner join vs Right/Left Join

```
SELECT c.id, c.name  
FROM customers c  
      LEFT JOIN orders o ON c.id = o.custId  
WHERE o.orderId IS NULL;
```

- LEFT JOIN asigură includerea tuturor clienților.
- WHERE o.orderId IS NULL filtrează numai pe cele nepotrivite (adică, fără comenzi).
- Nu este nevoie de agregare sau grupare - plan de execuție mai simplu.

# AWR reports - example

## SQL ordered by Executions

- %CPU - CPU Time as a percentage of Elapsed Time
- %IO - User I/O Time as a percentage of Elapsed Time
- Total Executions: 1,387,330
- Captured SQL account for 68.8% of Total

Executions	Rows Processed	Rows per Exec	Elapsed Time (s)	%CPU	%IO	SQL Id	SQL Module	SQL Text
150,959	150,992	1.00	2.72	14.9	0	<a href="#">65kfww9x61q88</a>	JDBC Thin Client	select max(businessda0_.busine...
122,824	73,474	0.60	5.20	18.2	0	<a href="#">dgy4un1mj3yy1</a>	JDBC Thin Client	select csdpartici0_.id as id2_...
106,049	106,059	1.00	2.46	14.2	0	<a href="#">gw58cbv5j4uur</a>	JDBC Thin Client	select synthetica0_.id as id1_...
49,361	17,715	0.36	2.30	17.9	0	<a href="#">2r760xt4p3hak</a>	JDBC Thin Client	select csdpartici0_.id as id2_...
43,098	43,096	1.00	2.24	20.5	0	<a href="#">5fsah9s5c79jc</a>	JDBC Thin Client	select issue0_.id as id1_183_...
39,924	39,927	1.00	4.92	28.1	0	<a href="#">bwfu41dyapp0v</a>	JDBC Thin Client	select instructio0_.id as id2_...

# AWR reports – example 2

## SQL ordered by Elapsed Time

- Resources reported for PL/SQL code includes the resources used by all SQL statements called by the code.
- % Total DB Time is the Elapsed Time of the SQL statement divided into the Total Database Time multiplied by 100
- %Total - Elapsed Time as a percentage of Total DB time
- %CPU - CPU Time as a percentage of Elapsed Time
- %IO - User I/O Time as a percentage of Elapsed Time
- Captured SQL account for 96.5% of Total DB Time (s): 18,726
- Captured PL/SQL account for 0.3% of Total DB Time (s): 18,726

Elapsed Time (s)	Executions	Elapsed Time per Exec (s)	%Total	%CPU	%IO	SQL Id	SQL Module	SQL Text
12,797.19	2,234	5.73	68.34	56.50	0.00	<a href="#">ab8xwyfahcj1</a>	JDBC Thin Client	select interfacem0_.tran_id as...
1,584.44	2,551	0.62	8.46	46.67	0.00	<a href="#">9uudc4bbawd28</a>	JDBC Thin Client	select distinct csdtran0_.id a...
1,382.64	12,488	0.11	7.38	51.34	0.00	<a href="#">38cu033tpnvcg</a>	JDBC Thin Client	select swifttrani0_.id as id2_...
384.71	1,399	0.27	2.05	46.44	0.00	<a href="#">400b3nfgjpk48</a>	JDBC Thin Client	select csdtran0_.id as id1_123...
217.09	1,024	0.21	1.16	47.48	0.00	<a href="#">2uk5zw6zufbjg</a>	JDBC Thin Client	select nsecorresp0_.id as id1_...
200.55	484	0.41	1.07	55.66	0.00	<a href="#">cfsf9vk9v3a3j</a>	JDBC Thin Client	select inputs0_.tran_id as tra...
154.03	233	0.66	0.82	44.97	0.00	<a href="#">5rb09uqgbatrs</a>	JDBC Thin Client	SELECT current_timestamp as c1...
145.50	13,751	0.01	0.78	47.82	0.00	<a href="#">3v17bkq4at7zq</a>	JDBC Thin Client	select count(csdtran0_.id) as ...
130.03	12,488	0.01	0.69	51.82	0.00	<a href="#">cys45rkvs7bu1</a>	JDBC Thin Client	select count(csdtran0_.id) as ...
126.07	256	0.49	0.67	48.00	0.00	<a href="#">3s9uu3vuaj5qx</a>	JDBC Thin Client	select activitylo0_.id as id1_...

mergand mai departe de la sql id se poate afla execution

plan > am vazut ca e full table scan

[Back to SQL Statistics](#)



# Reguli:

- 1) Performanta depinde si de server, de exemplu se recomanda evitarea OR in microsoft sql server. Oracle nu are probleme cu el, mssql e f lent
- 2) indecsi specializati. Daca pt un query se cauta dupa 2-3-x coloane, e preferabil folosirea unui index compus
  - Ordinea coloanelor in index conteaza.
- Bazele de date folosesc 1 index pt access la o tabela,
  - daca e un index pe sender si un index pe businessdate si