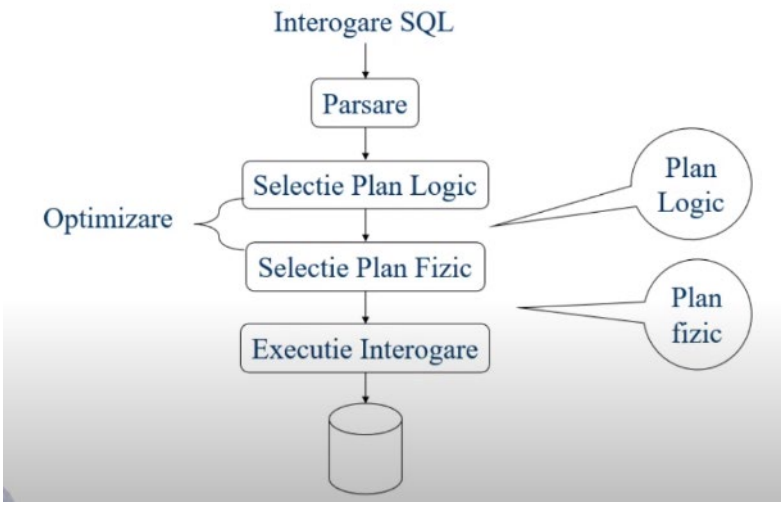
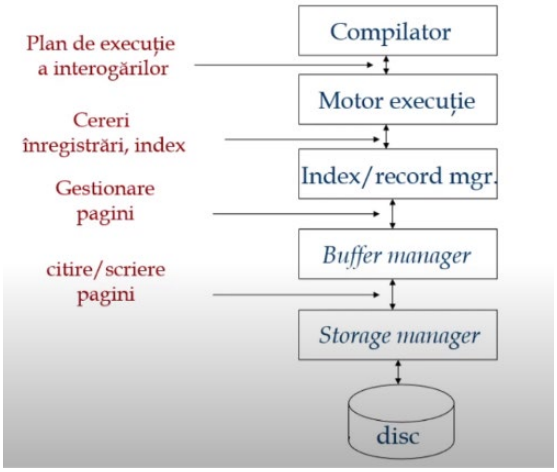
Optimizarea interogărilor

Tot ceea ce încearcă un SGBD este să găsească modalități de a executa cât mai rapid orice interogare pe care o lansăm pe acea bază de date. Există un modul de execuție a acestor interogări, care traversează niște pași:



- Parsarea – se ia interogarea, se sparge într-o secvență de operatori algebrici relaționali care sunt puși împreună într-un plan logic
- Plan logic – un arbore care are ca frunze tabele, și ca noduri interioare operatori algebrici relaționali
- Plan fizic – informații legate de modul în care se vor executa efectiv operatorii algebrici relaționali

Executarea interogărilor:



Structura folosită în exemple:

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

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

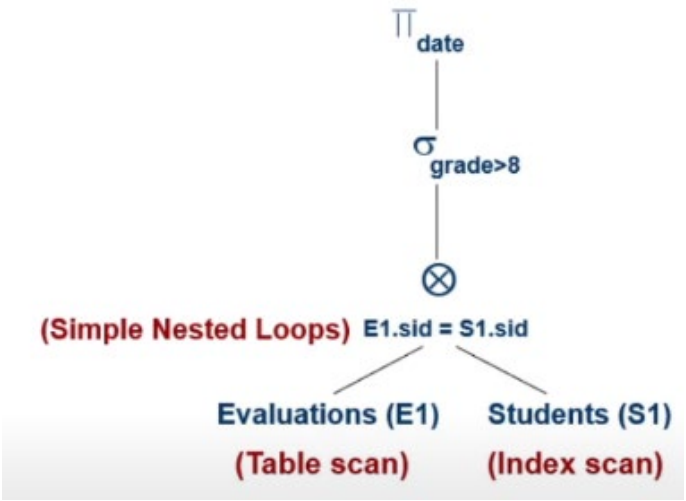
Evaluations (*sid*: integer, *cid*: integer, *day*: date, *grade*: integer)

- 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
```

SGBD încearcă să evite pe cât posibil produsul cartezian → se uită în clauza where și încearcă să determine dacă există niște condiții ce conțin un câmp dintr-una dintre tabele, și un câmp din cealaltă, și încearcă să execute un JOIN.



Plan logic de execuție – o ordonare logică a operatorilor algebrici relaționali care trebuie să se execute astfel încât să obținem rezultatul pe care ni l-ar da o astfel de interogare scrisă în SQL.

Plan fizic de execuție – adăugăm detalii legate de modul de implementare.

Index scan – deoarece se dorește doar câmpul *sid* și există un index pe el, e mai eficient să se facă un index scan decât să scaneze toată tabela

Simple Nested Loop Join – cea mai simplă metodă, dar care dă un cost ridicat

Mai departe nu mai este nimic specificat – se face *on the fly* – pe măsură ce mai obținem o pereche de înregistrare de Evaluations și una din Students pentru care există condiția E.*sid* = S.*sid*, îl dă mai departe. Intră în operatorul de selecție, se testează dacă *grade* > 8 și, dacă e adevărat, merge mai departe la operatorul de proiecție, elimină câmpurile de care nu avem nevoie, iar valoarea pentru coloana *date* merge mai departe. Deci, toată procesarea de la acel *join* se face în *pipeline*. Evaluările operatorilor nu trebuie făcute separat, se pot face înregistrare cu înregistrare.

Planul interogării:

- arbore logic
- se specifică decizia de implementare (la planul fizic)
- planificarea operațiilor

Frunzele planului de execuție: scanări

- Table scan: iterează prin înregistrările tabelei
- Index scan: accesează înregistrările index-ului tabelei
- Sorted scan: accesează înregistrările tabelei după ce aceasta a fost sortată în prealabil (printr-un algoritm de sortare extern)

Cum se combină operațiile?

Pentru a suporta acel flux în pipeline, se utilizează modelul iterator – pentru fiecare operator implementăm 3 funcții:

- *Open*: inițializări / pregătește structurile de date
- *GetNext*: returnează următoarea înregistrare din rezultat
- *Close*: finalizează operația / eliberează memoria

Când folosim sorted scan – nu putem folosi operatorul, deoarece nu știm cine e *GetNext*. De obicei, când avem nevoie de sortare, prima dată apelăm un algoritm de sortare externă, tot conținutul de după sortare îl memorăm într-o tabelă temporară, și abia după putem aplica operatorul → nu mai merge modelul iterator. Pentru asta este modelul materializat (*data-driven*), în care folosim niște tabele temporare pentru a executa niște operații.

Procesul de optimizare a interogărilor – presupune ca, odată ce am făcut acel plan logic de execuție, să aplicăm niște transformări algebrice (ne jucăm cu operatorii, vedem ce s-ar întâmpla dacă i-am muta pe arbore) pentru a obține un plan mai puțin costisitor.

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? Chiar să luăm toate permutările posibile între acei operatori?
- Cum se estimează costul unui plan?

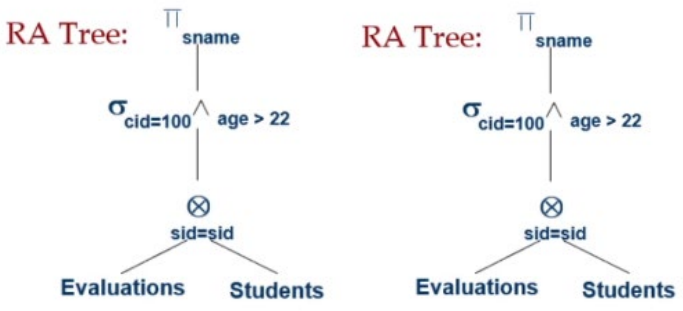
În mod ideal, SGBD ar reuși, după estimări, să obțină planul optic. În practică, se ajunge la eliminarea planurilor cele mai proaste din punct de vedere al costului și al timpului, și, din ceea ce a mai rămas, selectează una. Practic, ne garantează că nu va alege un plan foarte costisitor, dar nu va alege neapărat cel mai bun plan posibil.

System R Optimizer

- algoritm care e destul de popular, e implementat în marea majoritate a SGBD
- funcționează foarte bine dacă avem mai puțin de 10 *join*-uri
- estimarea costului se face prin aproximări
- ia în considerare costul CPU
- nu se estimează toate planurile:
    - sunt considerate doar planurile left-deep join – presupunem că în clauza FROM avem 10 tabele; un astfel de plan de execuție care urmează metoda left-deep join ia două tabele, face join între ele, apoi ia un al treilea tabel și face join cu rezultatul primelor două, și tot așa
    - se exclude produsul cartezian – încearcă să găsească orice modalitate de filtrare, de combinare a înregistrărilor unor tabele

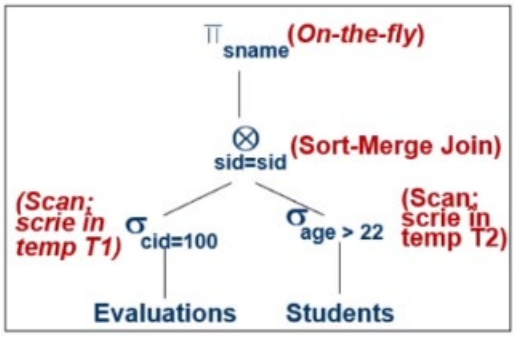
Exemplu:

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



Cost: 500 + 500 * 1 000 = 500 500 I / Os → plan inadecvat, cost mare

Plan alternativ 1:



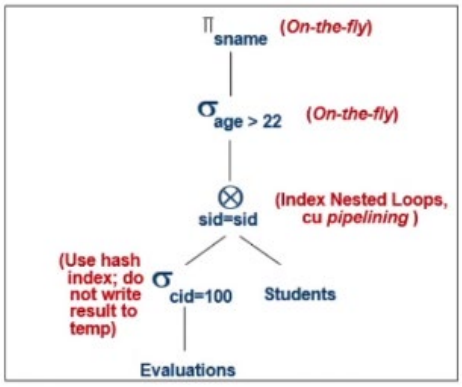
Sort-Merge Join – sortăm T_1, sortăm T_2 după *sid* și, odată sortate, le scanăm pentru a găsi perechile

Diferența esențială: *poziția operatorilor de selecție*

Costul planului (presupunem că sunt 5 pagini în buffer):

- Scan *Evaluations* (1000) + memorează temp T_1 (10 pagini, dacă avem 100 cursuri și distribuție uniformă) – *total 1 010 I / Os*
- Scan *Students* (500) + memorează temp T_2 (250 pagini, dacă avem 10 vârste) – *total 750 I / Os*
- Sortare T_1 (2 * 2 * 10), sortare T_2 (2 * 4 * 250), interclasare (10 + 250) – *total 2300 I / Os*
- Total: 4 060 pagini I / Os
- Dacă se folosește Block-Nested Loop Join (nu mai sortăm T_1 și T_2):
  - cost join = 10 + 4 * 250
  - cost total = 2 770
- Dacă "împingem proiecțiile":
  - T_1 rămâne cu *sid*, T_2 rămâne cu *sid* și *sname*
  - T_1 încape în 3 pagini, costul BNL este sun 250 pagini, total < 2 000

Plan alternativ 2:



- Cu index grupat pe *cid* din *Evaluations*, avem 100 000 / 100 = 1 000 tupluri în 1 000 / 100 = 10 pagini.
- INL cu *pipelining* (rezultatul nu e materializat) → se elimină câmpurile inutile din output.
- Coloana *sid* e cheie pentru *Students* → 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 * 1.2) → total = *1 210 I / Os*

Estimarea costului:

- Se estimează costul fiecărui plan considerat
  - Trebuie estimat costul fiecărui operator din plan (depinde de cardinalitatea tabelelor de intrare)
  - Trebuie estimată dimensiunea rezultatului pentru fiecare operație a arborelui (cu câte pagini rămânem)
- Algoritmul System R
  - Inexact, dar cu rezultate bune în practică
  - Viteza de execuție e una potrivită, satisfăcătoare
  - Există metode mai sofisticate, însă și acesta e destul de complex

Echivalențe în algebra relațională

Permit alegerea unei ordini diferite a *join*-urilor și "împingerea" selecțiilor și proiecțiilor în fața *join*-urilor.

- *Selecții*: $\sigma_{c1\wedge\ldots\wedge cn}(R) \equiv \sigma_{c1}(\ldots(\sigma_{cn}(R)))$ *(Cascadă)*
  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ *(Comutativitate)*
- *Proiecții*: $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$ *(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

- O 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 atributele lui R comută cu R *join* S → dacă condiția dintr-o selecție se referă doar la atribute dintr-o tabelă, putem muta acea selecție doar pe tabela respectivă și rezultatul să intre într-un join pe cealaltă tabelă

$$\sigma (R \otimes S) \equiv \sigma (R) \otimes S$$

- Dacă o proiecție urmează unui *join* între R și 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* – încearcă să vadă care sunt modalitățile cele mai bune de parcurgere a înregistrării dintr-o tabelă, după care combină tabelele între ele
- *Planuri cu tabele multiple*

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 tabelei:
  - *NPages(R).*

Exemplu:

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

Dacă există index pentru age:

- (1 / NKeys(I)) * NTuples(R) = (1 / 10) * 40 000 înregistrări 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 + 40 000) pagini returnate

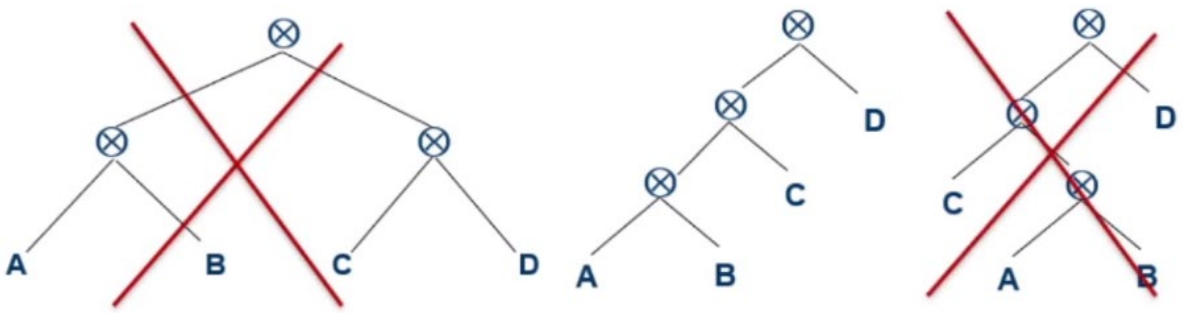
Dacă se scanează tabela:

- Sunt citite toate paginile (500)

Interogări pe tabele multiple

System R consideră doar arborii *left-deep join*

- Pe măsură de numărul de *join*-uri crește, numărul de planuri alternative este tot mai semnificativ → e necesară *restricționarea spațiului de căutare*
- Arborii *left-deep* ne permit generarea tuturor planurilor ce suportă *pipeline* complet
- Nu toți arborii *left-deep* suportă *pipeline* complet



Enumerarea planurilor left-deep

- Diferă prin ordinea tabelelor
- N pași de dezvoltare a planurilor cu N tabele:
  - 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 tabele)*
  - Pas N: Găsirea celei mai bune variante de *join* al rezultatului unui plan cu N – 1 tabele și altă tabelă *(Toate planurile bazate pe N tabele)*
- Pentru un număr mare de tabele crește exponențial numărul de combinații, de aceea acest algoritm este util și eficient până la maxim 10 *join*-uri