Department of Computer Science
University of Cyprus

EPL646 – Advanced Topics in Databases

Lecture 7

Evaluation of Relational Operators (Joins) and Query Optimization

Chapter 14.4: Ramakrishnan & Gehrke

Chapter 15: Ramakrishnan & Gehrke (* exclude 15.5 and 15.7)

Demetris Zeinalipour

http://www.cs.ucy.ac.cy/~dzeina/courses/epl646

Lecture Outline
Evaluation of Relational Operators

- 14.4) Algorithms for Evaluating **Joins**
  - **Simple Nested** Loops Join (SNLJ)
  - **Block-Nested** Loop Join (BNLJ)

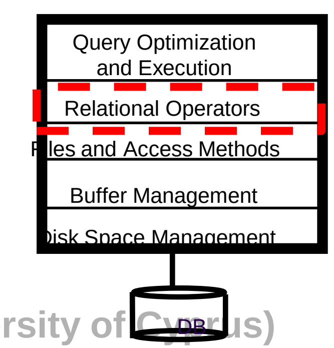
Enumerate Cross Product

  - **Index-Nested** Loops Join (INLJ)

Use Existing Index

  - **Sort-Merge** Join (SNLJ)
- 15) Query Optimization & Blocks:
  - **Enumeration of Alternative Plans** (Απαρίθμηση Εναλλακτικών Πλάνων)
  - **Cost Estimation of Plans** (Υπολογισμός Κόστους με Εκτέλεσης Πλάνων)

Partition the Data to avoid Enumerating the Cross Product



Introduction to Join Evaluation
(Εισαγωγή στην Αποτίμηση του Τελεστή Συνένωσης)

- The **JOIN operator (⊗)** combines records from **two tables** in a database, creating a set that can be **materialized** (saved as an intermediate table) or used **on-the-fly** (we shall only consider the latter case)
- It is among the most **common operators**, thus must be optimized carefully.
- We know that $\mathbf{R \otimes S \Leftrightarrow \sigma_c(R \times S)}$, yet **R** and **S** might be large so $\mathbf{R \times S}$ followed by a selection is inefficient!
- Our objective is to implement the join without enumerating the underlying cross-product.

Schema for Examples
(Σχήμα για Παραδείγματα)

- **Notation:**

  - M tuples in **R (Reserves)**, p_R tuples per page,
    - **M=1000 pages, p_R=100 tuples/page => 100K tuples**
  - N tuples in **S (Sailors)**, p_S tuples per page.
    - **N=500 pages, p_s=80 tuples/page => 40K tuples**

  Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

  Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)

- **Query: SELECT * FROM Reserves R1, Sailors S1**
  WHERE R1.sid=S1.sid
- ***Cost metric*****:** # of I/Os.
- We will **ignore output costs** (as always) as the results are sent to the user **on-the-fly**

Simple Nested Loops Join
(Απλή Συνένωση Εμφωλευμένων Βρόγχων)

foreach tuple **r** in **R** do // **Outer relation**
    foreach tuple **s** in **S** do // **Inner relation**
        if $\mathbf{r_i == s_j}$ then add **<r, s> to result**

- **A) Tuple-at-a-time Nested Loops join (TNLJ):** Scan *outer* relation R, and for each **tuple r ∈ R**, we scan the entire *inner* relation S a **tuple-at-a-time**. **Cost: $M + p_R * M * N$**
  Times scanning R
  Times scanning S
  - Cost: $M + (p_R * M) * N$ = 1000 + 100*1000*500 =**50,001,000 ~50M I/Os**

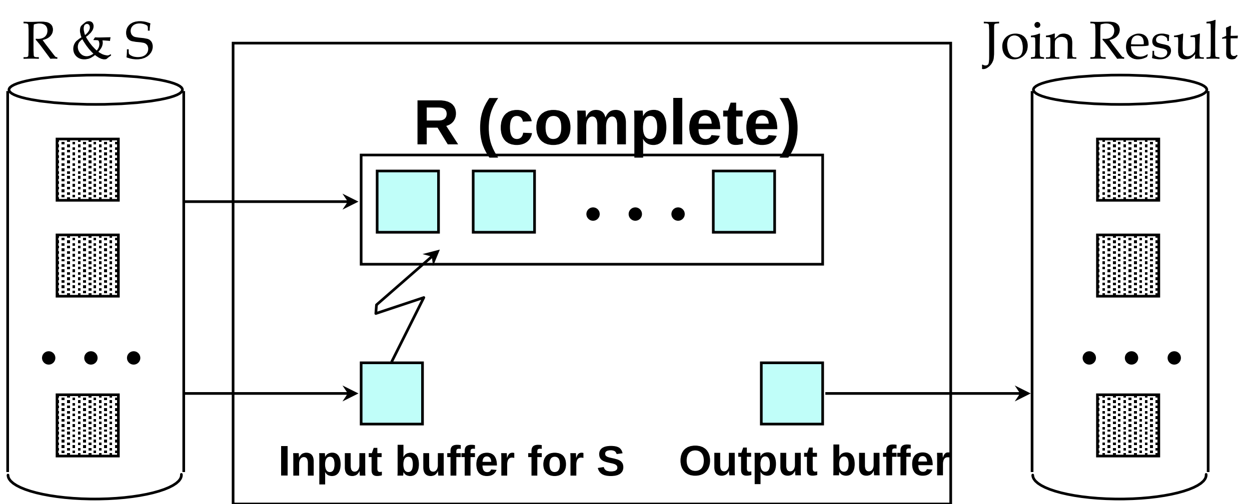
- **B) Page-at-a-time Nested Loops join:** Scan *outer* relation R, and for each **page ∈ R**, scan the entire *inner* relation S a **page-at-a-time (**TNLJ: no caching of retrieved S page)
  - Cost: $M + M * N$ = 1000 + 1000*500 =**501,000 I/Os** **Cost: $M + M * N$**
  - If smaller relation (S) is outer, cost = 500 + 500*1000 = **500,500 I/Os**

Rule: *The **outer relation** should be the **smaller** of the two relations (recall than* **R⊗S ⇔ S⊗R, i.e., Commutative (Αντιμεταθετική))**

SNLJ
BNLJ
INLJ
SMJ

Block Nested Loops Join
(Συνένωση Εμφωλευμένων Βρόγχων με χρήση Μπλόκ)

- **Problem:** SNLJ algorithm does **not** effectively **utilize buffer pages** (i.e., it uses **3** Buffer pages B_R, B_S and B_{out}).
- **Idea:** Load the smaller relation in memory (if it fits, its ideal!)
- **C) Block-Nested Loops Join (Case I)**
  – Load the complete **smaller R** relation to memory (assuming it fits)
  – Use one page as an **output buffer**
  – Use **remaining pages** (even 1 page is adequate) to load the larger S in memory and perform the join.

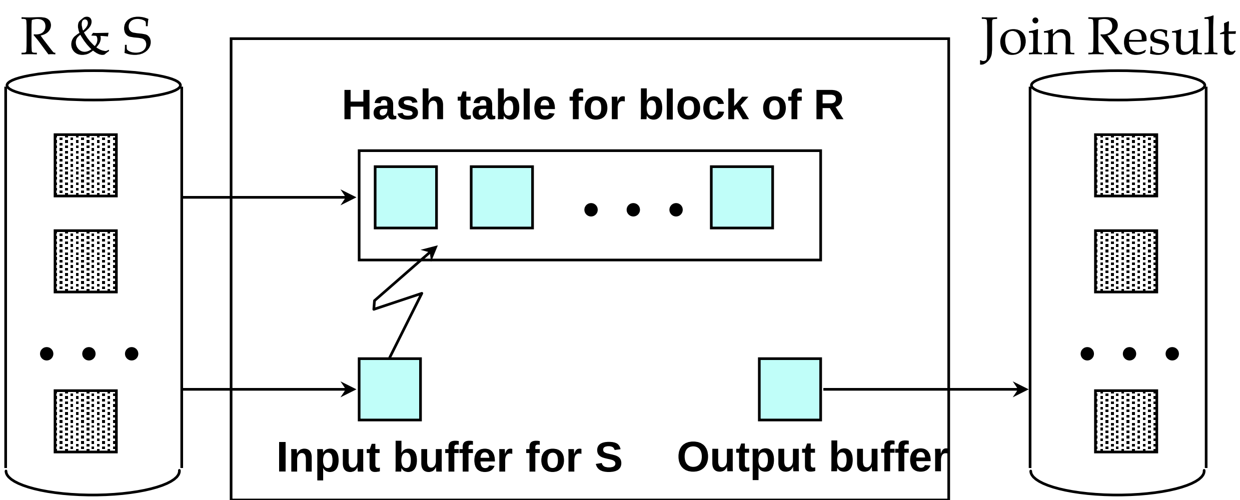


Cost: M+N

SNLJ
BNLJ
INLJ
SMJ

Block Nested Loops Join (Συνένωση Εμφωλευμένων Βρόγχων με χρήση Μπλόκ)

- **Problem: BNLJ spends time to join the results in memory**
- **Idea:** Build an In-Memory Hash Table for R (such that the in-memory matching is conducted in O(1) time)
- **C) Block-Nested Loops Join (Case II)**
  - Load the complete smaller **R** relation to memory and Build a Hashtable
  - Use one page as an **output buffer**
  - Use **remaining pages** (even 1 page is adequate) to load the larger S in memory and perform the join (by using the in-memory Hashtable).



Like previously,

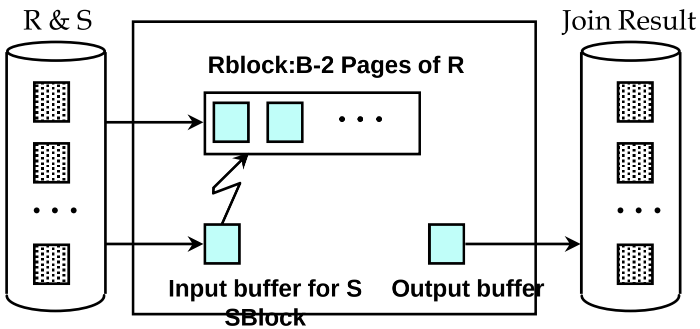
Cost: M+N

(But CPU cost is lower)

Block Nested Loops Join

(Συνένωση Εμφωλευμένων Βρόγχων με χρήση Μπλόκ)

- **Problem: What if smaller relation can't fit in buffer?**
- **Idea:** Use the previous idea but **break the relation R** into **blocks** (of size B-2) that can fit into the buffer.
- **C) Block-Nested Loops Join (Case III)**
  - Scan **B-2** pages of smaller **R** to memory (named **R-block**) (additionally, could build a hash table for this in-memory table)
  - Use 1 page as an **output buffer** and **1 page to scan S** relation to memory a page-at-a-time (named **S-page**) and perform the join.
  - Need to repeat the above ⌈M/(B-2)⌉ times (i.e., Number of Rblocks)




```
foreach block of B − 2 pages of R do
    foreach page of S do {
        for all matching in-memory tuples r ∈ R-block and s ∈ S-page,
        add ⟨r, s⟩ to result
    }
```

Cost: M + N * ⌈M/(B-2)⌉

Examples of Block Nested Loops
(Παράδειγμα Εμφωλευμένων Βρόγχων με χρήση Μπλόκ)

- **Let us consider an Example with BNLJ (case III), which has a cost of: M + N * ⌈M/(B-2)⌉**
- Let us consider various scenarios:
  - **Reserves (R) bigger as outer** and **B=102**
    - **Cost =** 1000 + 500 * ⌈1000/100 ⌉ = 1000 + 500*10 = **6000 I/Os**
  - **Reserves (R) bigger as outer** and **B=92**
    - **Cost =**1000 + 500 * ⌈1000/90 ⌉ = 1000 + 500*12 = **7000 I/Os**
  - **Sailors (S) smaller as outer** and **B=102**
    - **Cost =**500 + 1000 * ⌈500/100 ⌉ = 500 + 1000*5 = **5500 I/Os**
  - **Sailors (S) smaller as outer** and **B=92**
    - **Cost =**500 + 1000 * ⌈500/90 ⌉ = 500 + 1000*6 = **6500 I/Os**

Less Buffers
=> More I/O

Bigger Outer
=> More IO

- It might be best to **divide buffers evenly** between R and S (instead of allocating B-2 to one of the two relations)
  - **Seek time** can be **reduced** (data can be transferred sequentially to memory instead of **1 page-at-a-time for the S-page**)

Index Nested Loops Join

(Συνένωση Εμφωλευμένων Βρόγχων μέσω Ευρετηρίου)

- **Problem:** Previous approaches essentially enumerate the R×S set and do not exploit any existing indexes.
- **Idea:** If there is **an index** on the join column of one relation (say S), why not make it the **inner** and exploit the index.
- **d) Index-Nested Loops Join**
  - Scan *outer* relation R (page-at-a-time), for each **tuple r ∈ R**, we use the available index to retrieve the matching tuples of **S.**
  - **Cost: M + (p_R* M* Index_Cost)**
- **Index_Cost = Probing_Cost + Retrieval_Cost**
  - **Probing_Cost:** Depends on Index Type
    - **Hash Index:** ~1.2 I/Os **B+Tree Index:** 2-3 I/Os
  - **Retrieval_Cost:** Depends on Clustering
    - **Clustered (Altern. 2)**: 1 I/O (typical) **Clustered (Altern. 1)**: 0 I/Os
    - **Unclustered (Altern. 2):** upto 1I/O per matching S tuple.

Examples of Index Nested Loops

(Παράδειγμα Εμφωλευμένων Βρόγχων με χρήση Ευρετηρίου)

```
foreach tuple r in R do
        foreach tuple s in S where ri == sj do
                add <r, s> to result
```

Use Index on S

- Let us consider an Example with INLJ which has a cost:

  M + (p_R* M* Index_Cost)

- **Hash-index (Alt. 2)** on *sid* of **Sailors** (as **inner**):
  - **Cost =** 1000 + 100 * 1000 * (**1.2 + 1.0**) = **220,000 I/Os**
  - **Retrieval_Cost: 1.2 I/Os** to get data entry in index, plus **1.0 I/O** to get (**the exactly one, as sid is sailor's key**) matching Sailors tuple.
  - **Note:** Better than Simple (Page-at-a-time) Nested Loops join: M + M* N, which was **500,500 I/Os!**
  - Not comparing with **BNLJ** as the performance of the latter depends on the buffer size (shall compare BNLJ with SMJ later).
- **Hash-index (Alt. 1)** on *sid* of **Sailors** (as **inner**):
  - **Cost =** 1000 + 100 * 1000 * (**1.2 + 0.0**) = **120,000 I/Os**

Sort-Merge Join
(Σύζευξη με Ταξινόμηση και Συγχώνευση)

- Another method, like Index-Nested Loop Join, which avoids enumerating the **R×S** set.
- **Sort-Merge Join** utilizes a **partition-based approach** to join two relations (works only for equality joins)

e) Sort Merge Join Algorithm:

- **Sort Phase: Sort** both relations **R** and **S** on the **join attribute** using an **external sort** algorithm.
- **Merge Phase:** Look for **qualifying tuples** r ∈ R and s ∈ S by **merging** the two relations.

- Sounds similar to **external sorting**. In fact the Sorting phase of the sort alg. can be combined with the sorting phase of SMJ (we will see this next)

Sort-Merge Join
(Σύζευξη με Ταξινόμηση και Συγχώνευση)

- **Sort-Merge Join I/O Cost**

= **ExternalSort(R) + ExternalSort(S) + $\underbrace{\text{M + N}}_{\text{merge}}$**

=2M*#passes + 2N*#passes + M + N

$$=2M(1+\lceil \log_{B-1}\lceil M/B \rceil\rceil)+2N(1+\lceil \log_{B-1}\lceil N/B \rceil\rceil)+M+N$$

- Asymptotically, the I/O cost for SMJ is :

$$=O(M\log M)+O(N\log N)+O(M+N)\in O(M\log M+N\log N)$$

(however we will utilize the real cost in our equations)

- See next slide for examples…

Sort-Merge Join
(Σύζευξη με Ταξινόμηση και Συγχώνευση)

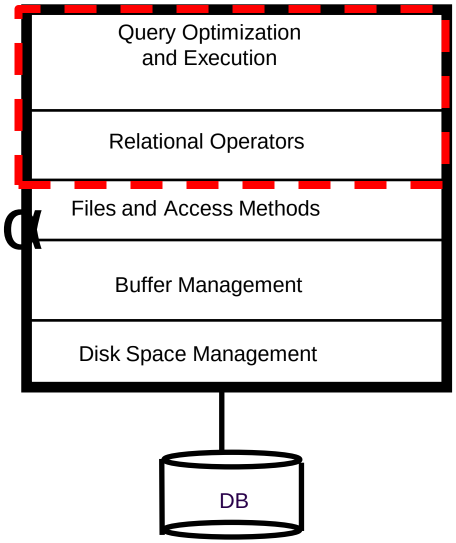
- **Let us consider an Example with SMJ, which has a cost of:** $2M(1+\lceil \log_{B-1}\lceil M/B \rceil\rceil)+2N(1+\lceil \log_{B-1}\lceil N/B \rceil\rceil)+M+N$
- Let us consider various scenarios:
  - **Buffer B=35, M=1000, N=500**
    - **Cost =** 2*1000***2** + **2***500***2** + 1000 + 500 = **7500 I/Os**
      - **Note:** $1+\lceil \log_{B-1}\lceil M/B \rceil\rceil = 1+\lceil \log_{34}\lceil 1000/35 \rceil\rceil = 1+\lceil 0.73 \rceil = 2$
    - Block-Nested Loops Join: **N + M*[N/(B-2)] =500+1000*[500/33] =16,500 I/Os**
  - **Buffer B=100, M=1000, N=500**
    - **Cost =** 2*1000***2** + **2***500***2** + 1000 + 500 = **7500 I/Os**
    - Similar to the Block-Nested Loops Join: **N + M*[N/(B-2)]= 6500 I/Os**
  - **Buffer B=300, M=1000, N=500**
    - **Cost =** 2*1000***2** + **2***500***2** + 1000 + 500 = **7500 I/Os**
    - Block-Nested Loops Join: **M + N*[M/(B-2)]=500+1000*[500/300] = 2,500 I/Os**

SMJ not better with larger buffer (i.e., Number of passes won't drop below 2)

* The number of passes during sorting remains at 2 in the above examples

Lecture Outline
Relational Query Optimizer

- **Introduction** to Relational Query Optimization (Σχεσιακή Βελτιστοποίηση Επερωτήσεων)
- **Query Blocks:** Units of Optimization (Μπλοκ Επερώτησης: Η Βασική μονάδα βελτιστοποίησης)
- **Enumeration of Alternative Plans** (Απαρίθμηση Εναλλακτικών Πλάνων)
- **Cost Estimation of Plans** (Υπολογισμός Κόστους με Εκτέλεσης Πλάνων)



Relational Query Optimization
(Σχεσιακή Βελτιστοποίηση Επερωτήσεων)

- A user of a DBMS formulates SQL queries.
- The query optimizer translates this query into an ***equivalent*** **Relational Algebra (RA)** query, i.e. a RA query with the same result.
- To optimize the efficiency of query processing, the query optimizer **reorders** the **individual operations (τελεστές)** within the RA query.
- **Re-ordering** has to preserve the query semantics (σημασιολογία) and is based on **Rel. Algebra equivalences,** e.g., some random examples:
  - $(R \otimes S) \equiv (S \otimes R)$ **(Commutative, Αντιμετάθεση)**
  - $\sigma_{A1 \wedge \ldots\ An}(R) \equiv \sigma_{A1}(\ldots \sigma_{An}(R))$ (*Cascade Conditions*, *Διάδοση)*
  - $\sigma_{A1}(\sigma_{A2}(R)) \equiv \sigma_{A2}(\sigma_{A1}(R))$ **(Commutative, Αντιμετάθεση)**

Query Blocks: Units of Optimization

(Μπλοκ Επερώτησης: Η Βασική μονάδα βελτιστοποίησης)

- An **SQL query** is parsed into a collection of ***query blocks (μπλοκ επερωτήσεων)***, and these are optimized one **block-at-a-time**.
- **Nested blocks** are usually treated as calls to a **subroutine**, made once per outer tuple.

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

Outer block (Εξωτερικό Μπλοκ)

Nested block (εμφωλευμένο μπλοκ)

- For each **block**, the plans considered are:
  - **All available access methods**, for each relation in the FROM clause.
  - **All possible join trees** for the relations in the FROM clause.
- We shall the above in further details in the following slides…

Query Blocks: Units of Optimization

(Μπλοκ Επερώτησης: Η Βασική μονάδα βελτιστοποίησης)

- A query is treated as a **σ-π-⊗ algebra expression** with the remaining operations (if any) carried out on the result.
- For our example, the optimizer only considers:

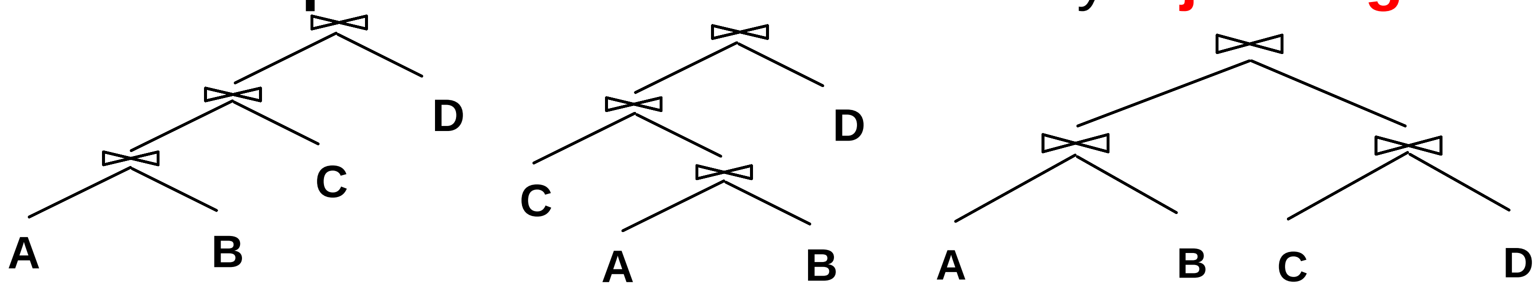
Relational Algebra Block (will be considered for evaluation):

$$\pi_{S.sid,R.day}($$
$$\sigma_{S.sid=R.sid \wedge R.bid=B.bid \wedge B.color='red' \wedge S.rating=value\_from\_nested\_block}($$
$$Sailors \times Reserves \times Boats))$$

- Aggregates, Having, Group-By are calculated after computing the **σ-π-⊗** of a query.
- Now the Optimizer needs to **i) enumerate** the alternative plans and **ii)** estimate **cost** of each plan.

Enumeration of Alternative Plans (Απαρίθμηση Εναλλακτικών Πλάνων)

- Problem: The **space** of **alternative plans** for a given query is **very large**!
- To motivate the discussion consider the **binary query evaluation plans** and **assume** that only **1 join alg. exists.**



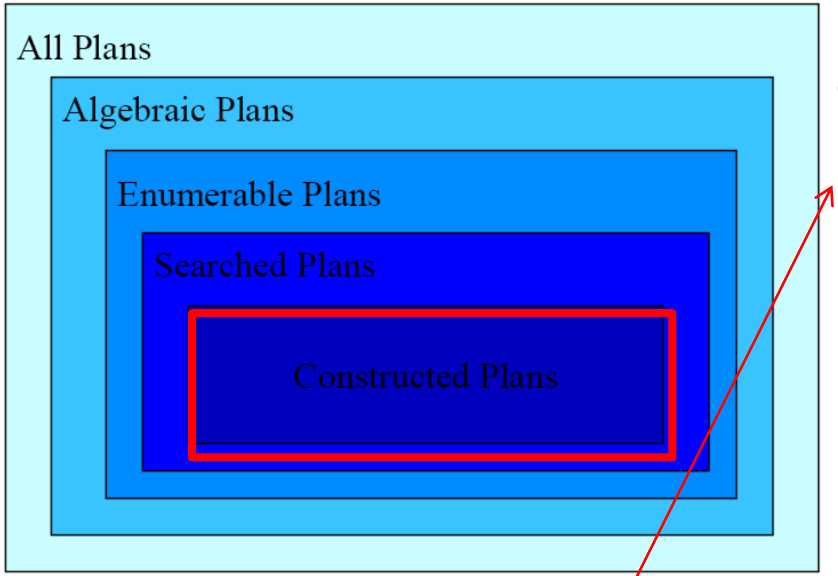
- **Question:** How many such plans can we have?
- **Answer:** Number of Binary Trees with **n** nodes:
  - N=4 we have 336 possible trees
  - N=5 we have 1008 possible trees
  - ….
  - N=10 we have 6 x 10^{10} possible trees

Number of Binary Plans: $C_n = \frac{(2n)!}{(n+1)!}$

We certainly need to prune (κλαδέψουμε) the search space!

Enumeration of Alternative Plans (Απαρίθμηση Εναλλακτικών Πλάνων)

- The Query Optimizer therefore focuses on a **subset of plans**.

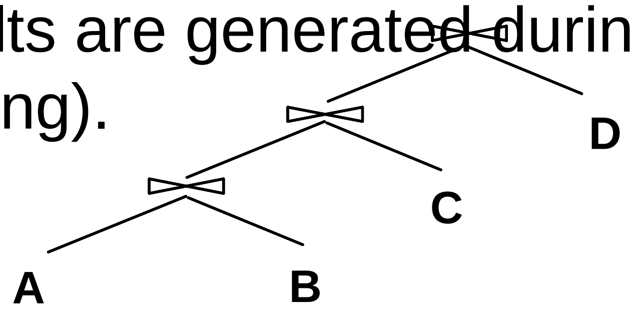


- **Algebraic plans:** those that can be expressed with Relational Algebra operators **σ-π-⊗**
- **Enumerable plans:** e.g., only binary plans.
- **Searched plans:** Among binary plans only consider the left-deep plans, i.e., where **right child** of each **join** is a leaf (base relation)
- **Constructed plans:** Those that are actually constructed.

Focus of the Query Optimizer

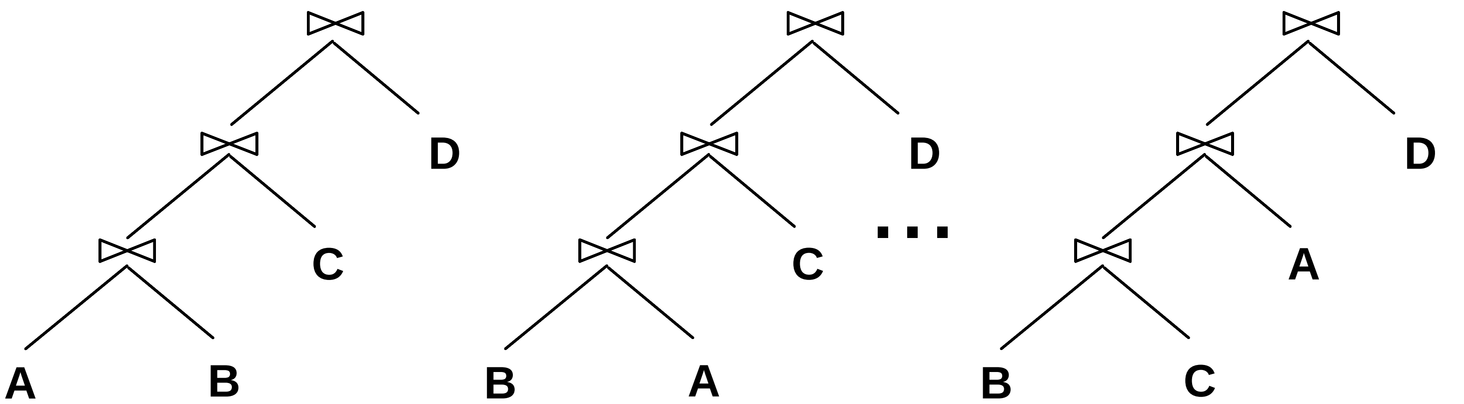
Enumeration of Alternative Plans (Απαρίθμηση Εναλλακτικών Πλάνων)

- *Left-deep (αριστεροβαθή) join trees:*
  - A left-deep tree is a tree in which the **right child** of each **join** is a leaf **(i.e., a base table or index).**
  - Left-deep trees allow us to generate all ***fully pipelined* plans (πλήρως σωληνωμένα πλάνα εκτέλεσης)** .
    - As **results are generated** these are forwarded to the operator higher in the tree hierarchy.
    - Intermediate results **not** written to **temporary files**.
    - **NOT** all left-deep trees are **fully pipelined** (e.g., SM join, no results are generated during sorting but only during merging).



Enumeration of Alternative Plans (Απαρίθμηση Εναλλακτικών Πλάνων)

- **Even by only considering left-deep plans, the number of plans still grows rapidly when number of join increases!**



Optimizers rely on System-R's dynamic programming approach to reduce the search space

- In particular, we have **N!** possible plans, where N the number of base relations participating in a join.
  - With N=4, we have 24 possible plans
  - With N=5, we have 120 possible plans
  - With N=6, we have 720 possible plans
  - ….
  - With N=10, we have 3628800 possible plans

Number of Left-Deep Plans*: N!

* Again assuming that only 1 join algorithm exists

Cost Estimation of Plans

(Υπολογισμός Κόστους με Εκτέλεσης Πλάνων)

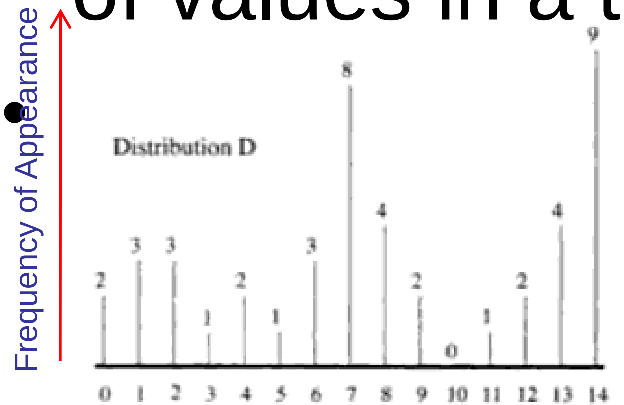
- Consider a Query Block:

```
SELECT  attribute list
FROM  A, B, …, Z
WHERE  term1 AND ... AND termz
```

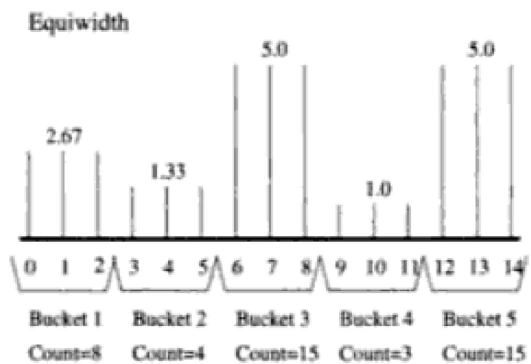
- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
  - i.e., |A|*|B|* … * |Z|
- ***Reduction factor (RF) (Συντελεστής Μείωσης): defines*** the ratio of the expected result size / input size
  - e.g., term1 yields 200 expected answers out of 1000 => $RF_{term1=}0.2$
- How can a DBMS know these RFs for a table without spending too much time? (next slide)

Reduction Factors Using Histograms
(Συντελεστές Μείωσης με Ιστογράμματα)

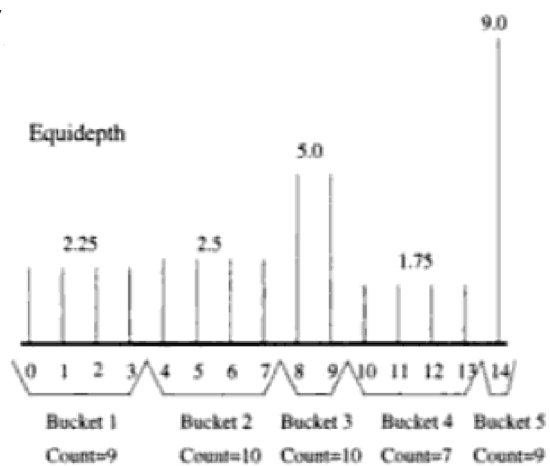
- **Wrong Answer:** Scan the table => Too Expensive
- **Correct Answer:** Utilize Histograms (tiny data structures that approximate the real distribution of values in a table (stored in sy



Initial Distribution of “age”



Equiwidth Histogram



Equidepth Histogram