Department of Computer Science
University of Cyprus

EPL646 – Advanced Topics in Databases

Lecture 6

External Sorting & Query Evaluation

Chapter 13-12: Ramakrishnan & Gehrke

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl446

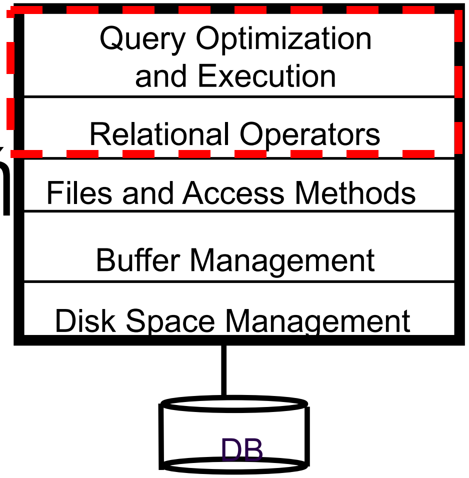
External Sorting Introduction (Εξωτερική Ταξινόμηση: Εισαγωγή)

- **Problem:** We can't sort 1TB of data with 1GB of RAM (i.e., more data than available memory) in main memory
- **Solution:** Utilize an External Sorting Algorithm
  - **External sorting** refers to the sorting of a file that resides on secondary memory (e.g., disk, flash, etc).
  - **Internal sorting** refers to the sorting of an array of data that is in RAM (quick-, merge-, insertion-, selection-, radix-, bucket-, bubble- ,heap-, sort algorithms we saw in the Data Struct. & Alg. Course)
- **Objective:** Minimize number of I/O accesses.
- External Sorting is part of the **Query Evaluation / Optimization subsystem**
  - Efficient Sorting algorithms can speed up query evaluation plans (e.g., during joins)!

Lecture Outline

External Sorting – Εξωτερική Ταξινόμηση

- 13.1) **When** does a DBMS sort Data.
- 13.2) A **Simple Two-Way Merge-Sort** (Απλή Εξωτερική Ταξινόμηση με Συγχώνευση)
- 13.3) **External Merge-Sort** (Εξωτερική Ταξινόμηση με Συγχώνευση)
  - Exclude 13.3.1: Minimizing the Number of Runs.
- 13.4.2) **Double Buffering** (Διπλή Προκαταχώρηση)
- 13.5) **Using B+Trees for Sorting**



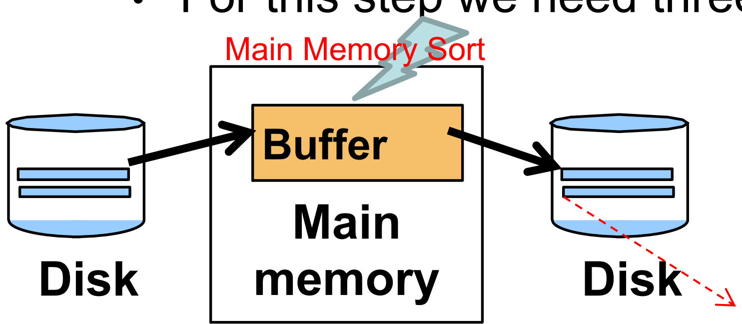
When Does a DBMS Sort Data?
(Πότε μια Β.Δ Ταξινομεί Δεδομένα;)

- **When Does a DBMS Sort Data? (~30% of oper.)**
  - Data requested in **sorted order**
    - e.g., SELECT * FROM Students **ORDER BY** gpa DESC;
  - Sorting is first step in ***bulk loading a*** B+ tree index.
    - i.e., CREATE INDEX StuAge ON Students(age) USING BTREE;
    - Recall how leaf nodes of the B+tree are ordered.
  - Useful for **eliminating *duplicate copies*** in a collection of records.
    - SELECT **DISTINCT** age FROM Students;
    - i.e., to eliminate duplicates in a sorted list requires only the comparison of each element to its previous element so this yields a linear order elimination algorithm.

Two-Way External Merge-Sort

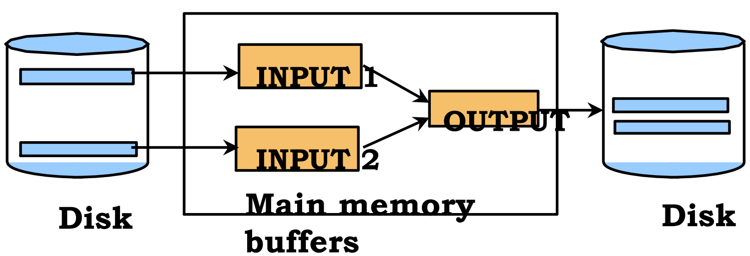
(Απλή Εξωτερική Ταξινόμηση με Συγχώνευση)

- Let us consider **the simplest idea** for external sorting
  - **Assumption:** Only 3 Buffer pages are available
  - **Idea:** Divide and Conquer (similarly to MergeSort, Quicksort)
- **Idea Outline**
  - **Pass 0 (Sort Lists):** For every page, **read** it, **sort** it, **write** it out
    - Only one buffer page is used!
    - Now we need to merge them hierarchically
  - **Pass 1, 2, …, etc. (Merge Lists):** see next page for merging concept
    - For this step we need three buffer pages!



Sorted Run or simply Run

Pass 0 **Διατεταγμένες Ακολουθίες** ή απλά **Ακολουθίες**



Passes 1,2,…

Cost of Two-Way External Merge Sort
(Κόστος Απλής Εξωτερικής Ταξινόμηση με Συγχώνευση)

- **Each pass we read + write each of N pages in file.** Pass 0
- **Number of passes:**
$$=\lceil \log_2 N \rceil + 1$$

e.g., for N=7, N=5 and N=4

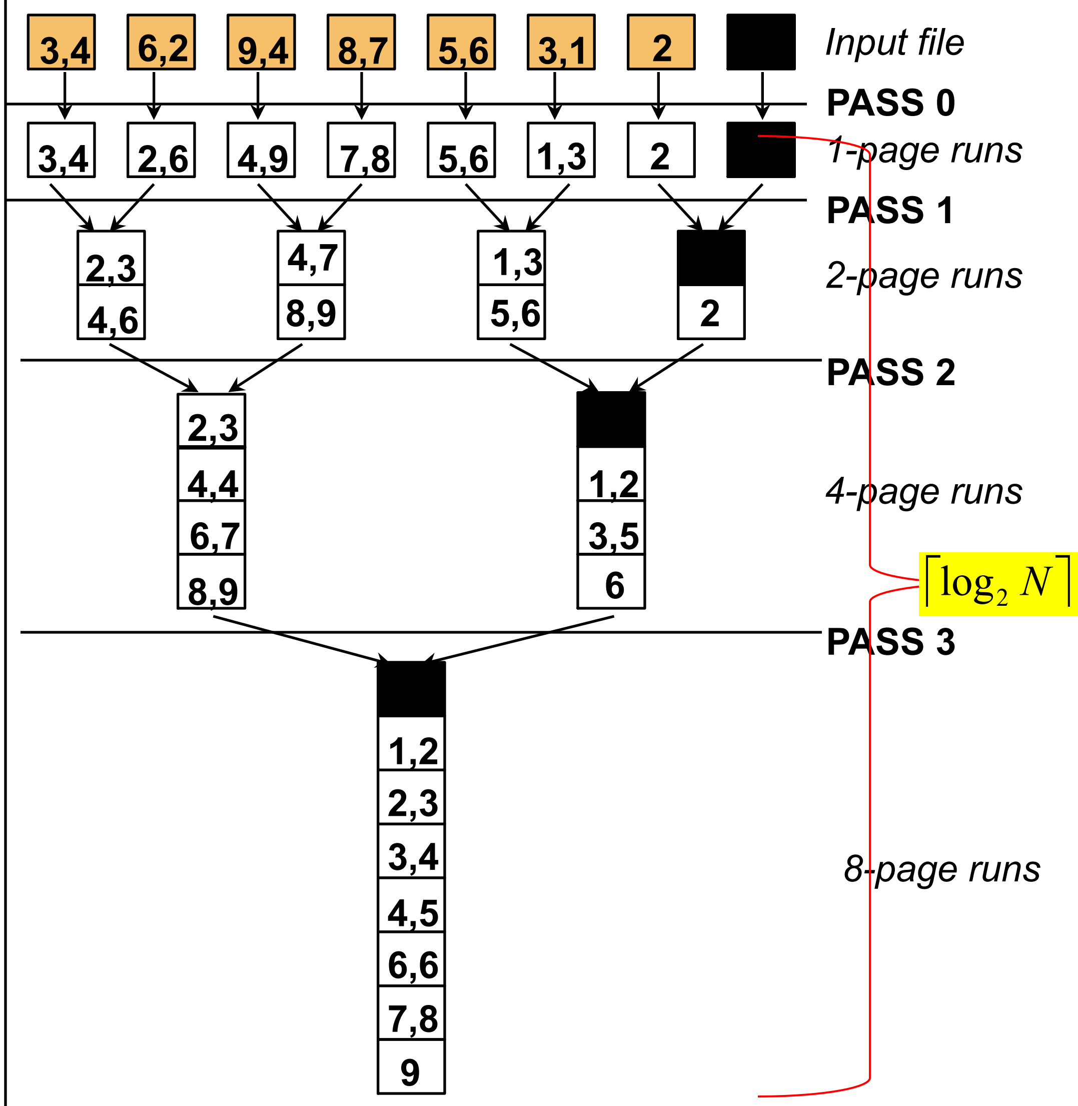
$$\lceil \log_2 7 \rceil + 1 = \left\lceil \frac{\log_{10} 7}{\log_{10} 2} \right\rceil + 1 = \lceil 2.8 \rceil + 1 = 4$$
$$\lceil \log_2 5 \rceil + 1 = \lceil 2.3 \rceil + 1 = 4$$
$$\lceil \log_2 4 \rceil + 1 = \lceil 2 \rceil + 1 = 3$$

- **Total (I/O) cost is:**
$$2N * (\# \, passes)$$

e.g., for N=7

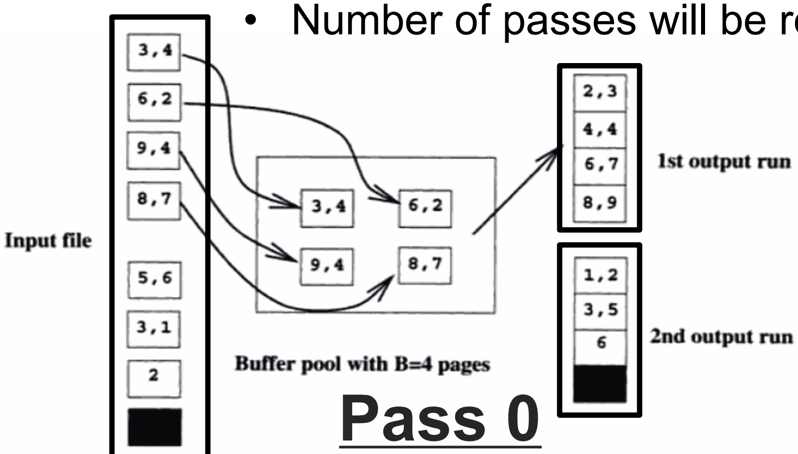
$$2 * 7 * (\lceil \log_2 7 \rceil + 1) = \boxed{2 * 7 * 4} = 56$$

- *i.e., (read+write) * 7 pages * 4 passes*
- *That can be validated on the right figure*
  - *Pass#0=2*7* *Pass#1=2*7*
  - *Pass#2=2*7* *Pass$3=2*7*

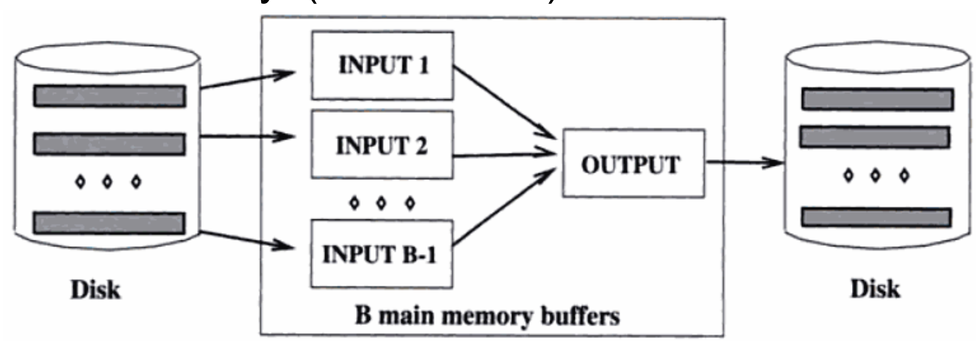


General External Merge Sort
(Γενικευμένη Εξωτερική Ταξινόμηση με Συγχώνευση)

- Let’s turn the 2-way Merge-sort Algorithm into a **Practical Alg.**
  - **Assumption: B** Buffer pages are available
  - **Idea:** Merge (B-1) pages in each step rather than only 2 (faster!)
- **Idea Outline**
  - **Pass 0 (Sort): Sort** the **N** pages using **B** buffer pages
    - Use **B buffer pages** for input
    - That generates $N1 = \lceil N/B \rceil$ sorted runs e.g., N=8 and B=4 => N1=$\lceil 8/4 \rceil = 2$
  - **Pass 1, 2, …, etc. (Merge):** Perform a **(B-1)-way** merge of runs
    - Use **(B-1) buffer pages** for input + **1 page for output**
    - Number of passes will be reduced dramatically! (see slide 13)



Pass 0



Passes 1,2,…

Number of Passes of External Sort

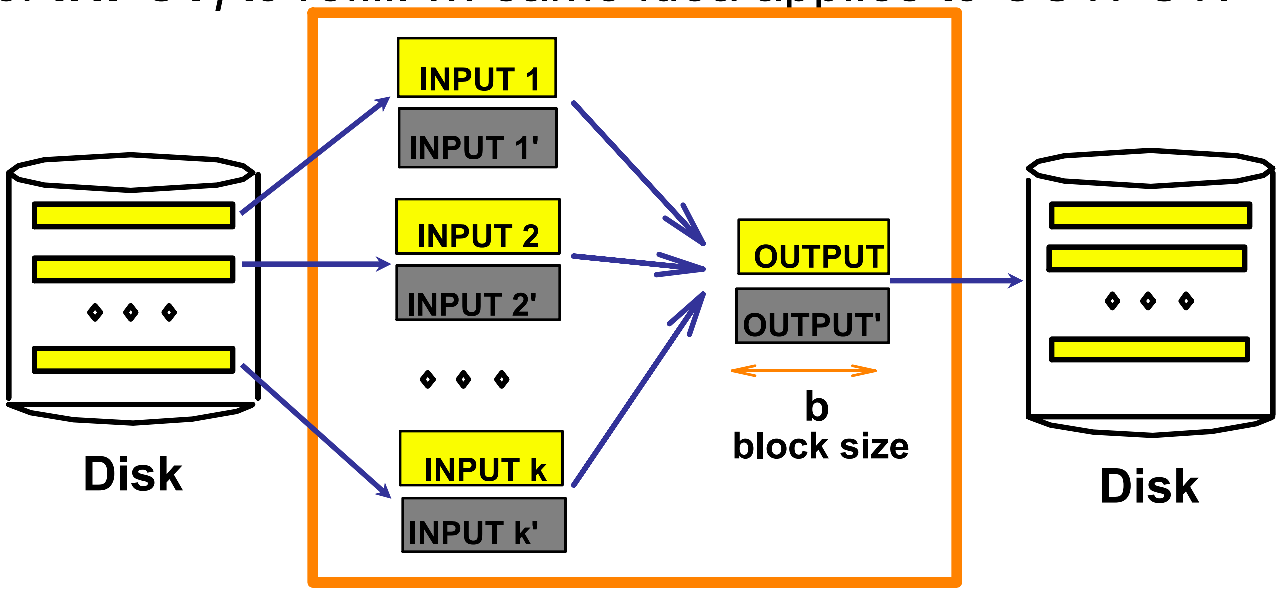
- External Merge Sort is quite efficient!
- With only **B=257 (~1MB)** Buffer Pages we can sort **N=1 Billion records** with four **(4) passes** … in practice B will be larger
- Two-Way Mergesort would require $= \lceil \log_2 10^9 \rceil + 1 = 30 + 1$ passes!

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	$\boxed{4}$

* Results generated with formula: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$

Double Buffering
(Διπλή Προκαταχώρηση)

- **An I/O request takes time to complete**. Only think about all the involved layers and delays (Disk Delays, Buffer Manager, etc)
- To **reduce wait time** of CPU for **I/O** request to complete, can ***prefetch*** (προανάκτηση) into **`shadow block' (μπλοκ αντίγραφο)**
- **Main Idea:** When all tuples of ***INPUT$_i$*** have been consumed, the CPU can process ***INPUT$_i$'*** which is prefetched into main memory instead of waiting for ***INPUT$_i$*** to refill. … same idea applies to OUTPUT.



(B main memory buffers, k-way merge)

Department of Computer Science
University of Cyprus

EPL646 – Advanced Database Systems

Overview of Query Evaluation

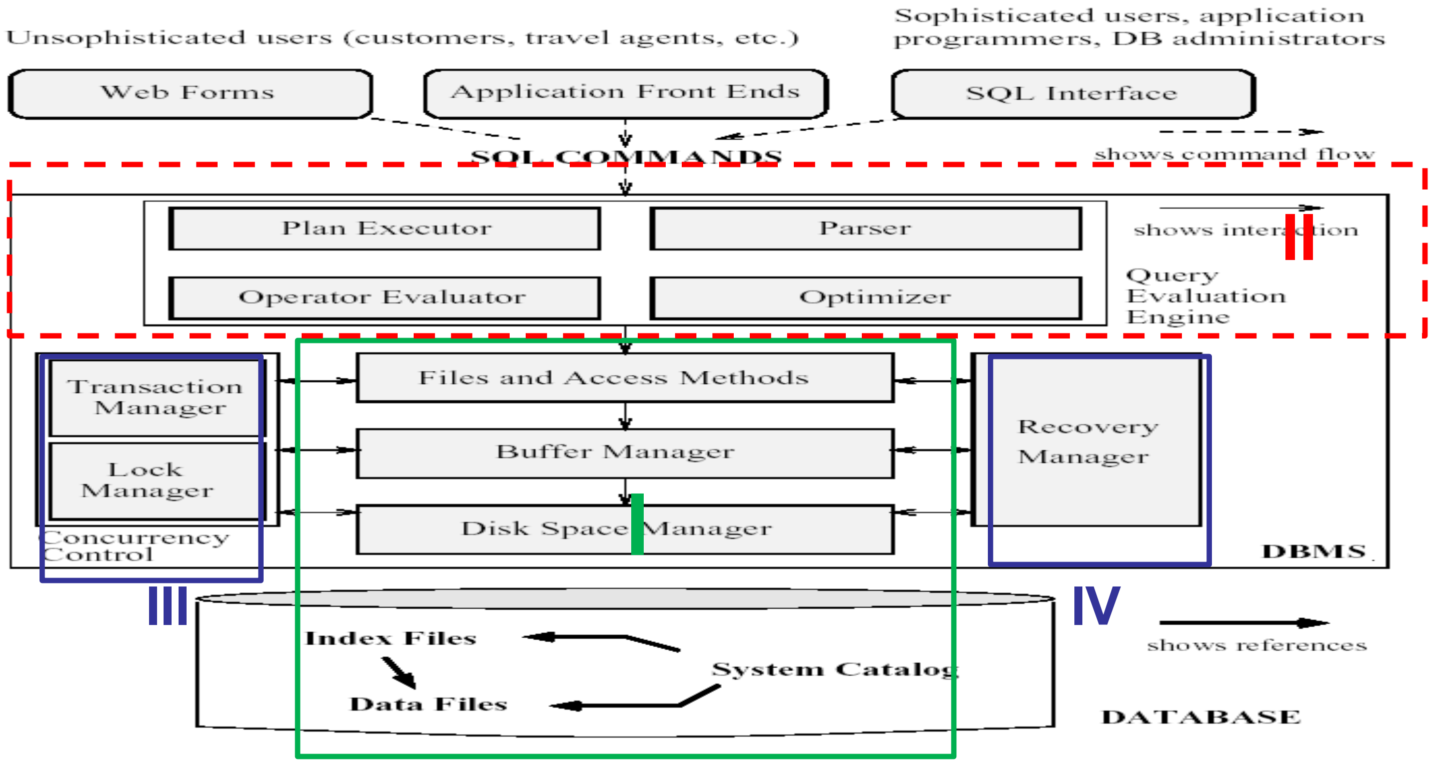
Chapter 12: Ramakrishnan & Gehrke

Demetris Zeinalipour

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

Overview of Query Evaluation (Επισκόπηση Αποτίμησης Επερωτήσεων)

- We will now focus on **Query Evaluation (Αποτίμηση Επερωτήσεων),** specifically **Query Optimization (Βελτιστοποίηση Επερωτήσεων)**



Query Evaluation (Αποτίμηση Επερωτήσεων)

- We shall next discuss techniques used by a DBMS to **process (επεξεργάζεται)**, **optimize (βελτιστοποιεί)**, and **execute (εκτελεί)** high-level queries.

- ***A Query Evaluator (Απομιμητής Επερωτήσεων)***
  - ***A) Parses (Αναλύει):*** *Takes a **declarative** description (**δηλωτική** περιγραφή) of a query (i.e., expression what we want without describing how to do it (e.g., SQL).*
    - *That is different from **imperative** descriptions (**προστακτική** περιγραφή): expression of how to achieve the expected result (e.g., C, C++, JAVA, etc):*
  - **B) Optimizes (Βελτιστοποιεί):** Determines plan for answering query (expressed as DBMS operations).
  - **C) Executes (Εκτελεί):** Executes method via DBMS engine (to obtain a set of answers that answer a given query)

=> Next slides contain further details….

Query Evaluation (Αποτίμηση Επερωτήσεων)

Parse
Optimize
Execute

- **A) Parser (Αναλυτής):**
  - **Scanner (Λεκτική Ανάλυση):** Identifies the language tokens – such as SQL keywords, **attribute names (γνωρίσματα**), **relation names (ονόματα σχέσεων)**, etc.
  - **Parser (Συντακτική Ανάλυση):** Checks the query syntax to determine whether it is formulated according to the **syntax rules** (κανόνες γραμματικής) of the query language.
    - E.g., FROM TABLE SELECT *; // not syntactically correct.
  - **Validator (Επικύρωση Εγκυρότητας):** Checks that all attributes and relation names are **valid (ορθά)** and **semantically (σημασιολογικά)** meaningful names
    - E.g., WHERE JOB = 'SECRETARY' AND JOB = 'MANAGER' is **syntactically** correct but **semantically** incorrect. (will always yield an empty set as an employee can't be both)
- Parsing yields an Internal tree representation of the query, called a **Relational Algebra Tree** (Δένδρο Σχεσιακών Τελεστών) e.g.,

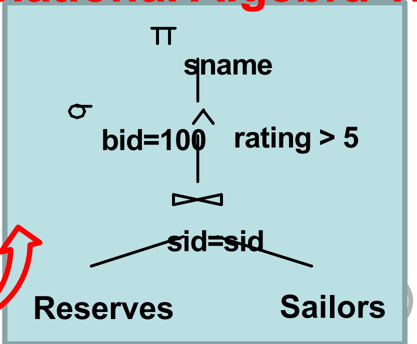
SQL

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid = S.sid
        AND R.bid = 100 AND S.rating > 5
```

Relational Algebra

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

Relational Algebra Tree



Query Evaluation (Αποτίμηση Επερωτήσεων)

Parse
Optimize
Execute

B) Optimize (Βελτιστοποίηση)

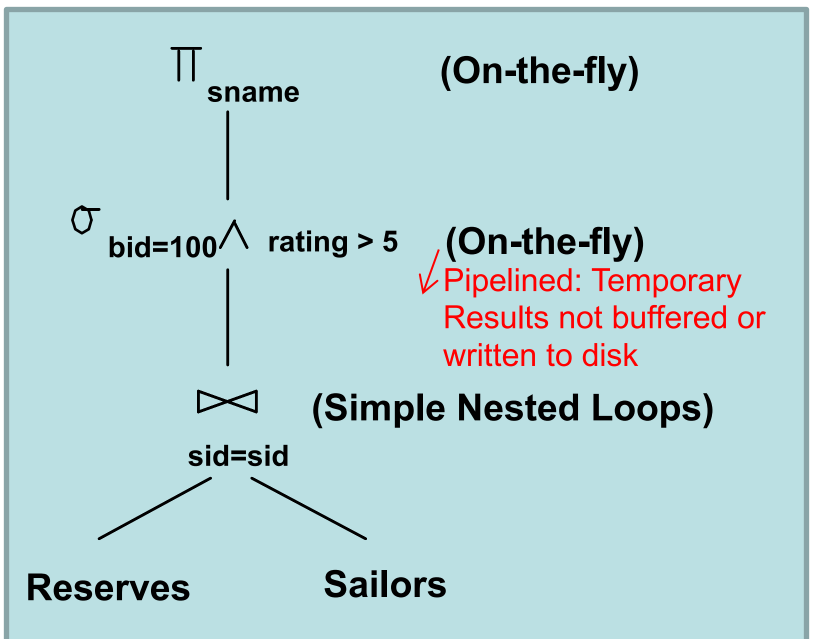
- The DBMS must then devise an execution strategy (**query evaluation plan**).
- That is difficult though, as a query might have many alternative options!
- **Best choice depends** on many factors: **size of tables**, **existing indexes**, **sort orders (asc/desc), size of available buffer pool**, **BM replacement policy**, implemented algorithm for operator evaluation (e.g., Sort-Merge Join, Hash-Join, etc).
- The process of choosing a **suitable**, «reasonably efficient but most of the time NOT optimal» one is known as **Query Optimization (Βελτιστοποίηση Επερωτήσεων)**.

Query Evaluation Plans

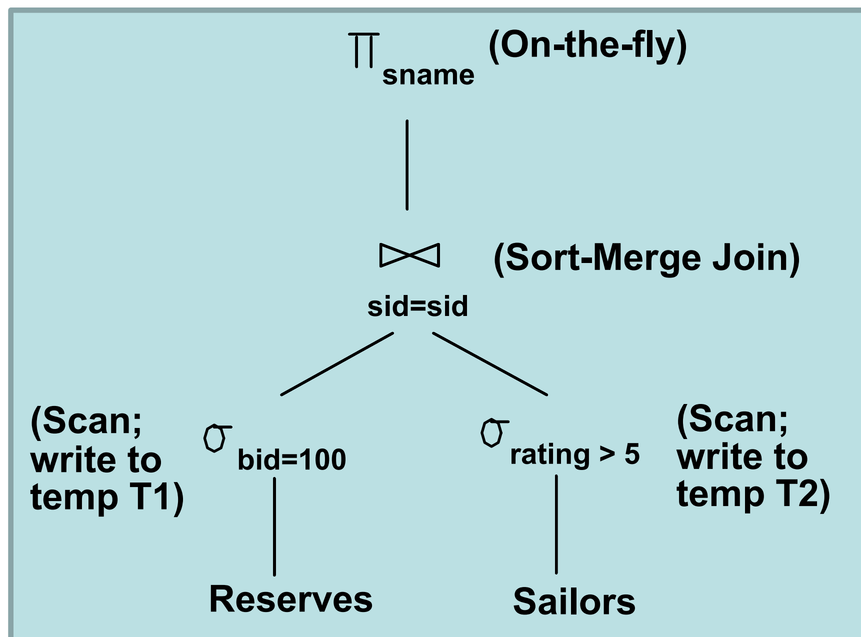
(Πλάνο Αποτίμησης Επερώτησης)

- *Query Evaluation Plan (or simply Plan):* A *Tree of Relational Algebra* ***operators*** *(essentially σ-π-join [basic block], while rest operators are carried out on the result)* ***with choice of algorithm for each operator.***

Query Evaluation Plan A



Query Evaluation Plan B



Relational Operations (Σχεσιακοί Τελεστές)

- We will consider how a DBMS implements:
  - *14.1-14.2) Selection - Επιλογή* **(σ):** Selects a **subset of rows** from a Relation.
  - *14.3) Projection – Προβολή* **(π): Deletes unwanted columns** from a Relation.

Subsequent slides

  - *14.4) Join - Συνένωση* **(⊗)** Allows us to combine two relations
  - *14.5) Set-difference - Διαφορά* **(-):** Tuples in Relation 1, but not in Relation 2.
  - *14.5) Union - Ένωση* **(∪):** Tuples in Relation 1 or in Relation 2.
  - *14.6) Aggregation - Συνάθροιση* (SUM, MIN, etc.) and GROUP BY
- Since each **op** returns a relation, operators can be *composed*!
- After we cover the operations, we will discuss how to *optimize* queries formed by composing them.
  - **Relational Algebra operators** are **closed**: a **set** is said to be **closed under some operation** if the operation on members of the set **produces a member** of the set.

Schema for Examples (Σχήμα για Παραδείγματα)

- Assume the following Schema:

> **Sailors** (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
> **Reserves** (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Also assume the following values:
  - **Sailors:** Each tuple is **50 bytes** long, **80 tuples** per page, **N=500 pages** of such records stored in the database.
  - **Reserves:** Each tuple is **40 bytes** long, **100 tuples** per page, **M=1000 pages** of such records stored in the database.

The Selection Operation I
(Ο Τελεστής Επιλογής I)

- Consider the selection query listed below.

```
SELECT   *
FROM Reserves R
WHERE R.rname = 'Joe'
```

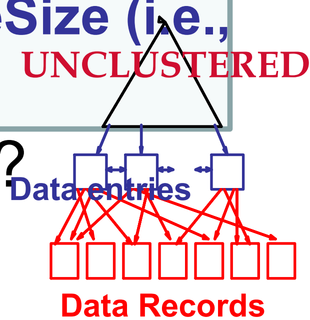
- **Selection with No Index, Unsorted Data:**
  - **Idea: Scan** R, **checking condition** on each tuple **on-the-fly** and returning qualifying objects.
  - **Cost:** **M**, where M is the # of pages for Reserves
- **Can we improve the above approach?**
  - **e.g.,** if data is **Sorted** or if **Hash index** on **R.rname** is available then this query could be answered more quickly!
- **We shall now only focus on simple** $\sigma_{R.attr\ \mathbf{oper}\ value}(R)$ queries and then extend the discussion to more complex boolean queries (e.g., $\sigma_{R1.attr\ \mathbf{oper}\ value\ AND\ R2.attr\ \mathbf{oper}\ value}(R)$)

The Selection Operation II
(Ο Τελεστής Επιλογής II)

- **Selection using No Index, Sorted Data:**
  - **Idea:** Perform binary search over target relation; Identify First Key; and finally scan remaining tuples starting at first key.
  - **Search Cost:** $\log_2 M$
  - **Retrieval Cost: #matching_records / PageSize (i.e., #matching_pages)**
    - **For the R relation the search cost is:** $\log_2 1000 \sim 10$ I/Os
- In practice it is difficult to maintain a file sorted.
- It is more realistic to use a B+Tree using Alternative 1 (see next slide)

The Selection Operation III
(Ο Τελεστής Επιλογής ΙΙΙ)

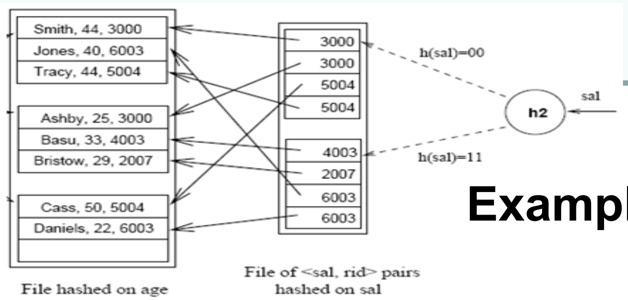
- **Selection using B+Tree Index:**
  - **Idea:** Use tree to **find the first index entry** that points to a qualifying tuple of R; Then **scan the leaf pages** to retrieve all entries in the key value that satisfy the selection condition.
  - **Search Cost: $\log_F M$, (typically 2-3 I/Os) F:branching factor**
  - **Retrieval Cost: i) Unclustered: #matching_records** (each record on separate page); **Clustered: #matching_records/PageSize (i.e., #matching_pages)**



- Why is a B+Tree **NOT** always superior to Scanning?
  - **Query:** SELECT * FROM Reserves R WHERE R.rname = ‘Joe’
  - **R** relation features 1000 pages.
  - **Assumption:** Selectivity (επιλεξιμότητα) of Query is **10%** (i.e., 10%*1000 pages * 100 tuples/page = 10,000 tuples)
  - **Clustered Index Cost: 3 I/Os + 100 I/Os** (tuples on 100 consec. pages)
  - **Unclustered Index Cost: 3 I/Os + 10,000 I/Os** (each tuple on differ. Page)
    - **It is cheaper to perform a linear scan that only costs 1000 I/Os!**

The Selection Operation IV (Ο Τελεστής Επιλογής IV)

- **Selection using Hash Index:**
  - **Idea:** Use hash index to **find the index entry** that points to a qualifying tuple of **R**; Retrieve all entries in which the key value satisfies the selection condition.
  - **Search Cost: Const (typically 1.2 I/Os, recall lin./extd. hashing)**
  - **Retrieval Cost: i) Unclustered: #matching records** (each record on separate page); **Clustered: #matching_records/PageSize (i.e., #matching_pages)**



Example of Unclustered Index

- Example
  - **Query:** SELECT * FROM Reserves R WHERE R.rname = 'Joe'
  - **Assumption:** Selectivity (επιλεξιμότητα) of Query is **10%** (i.e., 10%*1000 pages * 100 tuples/page = 10,000 tuples)
  - **Clustered Index Cost: 1.2 I/Os + 100 I/Os** (tuples on 100 consec. pages)
  - **Unclustered Index Cost: 1.2 I/Os + 10,000 I/Os** (each tuple on differ. Page)
    - Again, it is cheaper to perform a linear scan that only costs 1000 I/Os.

Complex Selections
(Σύνθετες Επιλογές)

Selection WITHOUT Disjunctions

Template: $\sigma_{A \wedge B \wedge .... \wedge Z}(R)$

That is in CNF
(conjunction of clauses, where a clause is a disjunction of literals)

First Approach

- Compute the **most selective access path** **$R' = \sigma_A$** (i.e., the one that returns the **fewest irrelevant** results compared to **$\sigma_B(R)$** … **$\sigma_Z(R)$**)
  - This could be a composite selection e.g., **$(\sigma_{A \wedge B \wedge C}(R))$** … depends on what access methods (indexes) are available.
- Then apply **on-the-fly** the rest conditions on **R' (i.e., $\sigma_{B \wedge .... \wedge Z}(R')$)**

Example

- Consider *day<8/9/23 AND bid=5 AND sid=3.*
- A **B+ tree index** on ***day*** can be used; then, ***bid=5*** and ***sid=3*** must be checked for each retrieved tuple on-the-fly.
- Similarly, a **hash index** on **<*bid, sid*>** could be used; *day<8/9/23* must then be checked on-the-fly.

Complex Selections
(Σύνθετες Επιλογές)

Selection WITHOUT Disjunctions

$\sigma_{A \wedge B \wedge \ldots \wedge Z}(R)$

That is in CNF

Second Approach

- Compute $R_A = \sigma_A(R)$ and $R_B = \sigma_B(R)$, and … and , $R_z = \sigma_z(R)$ using independent access methods (if indexes are available)
- **Intersect RID Sets:** $sort(R_A) \cap sort(R_B) \cap \ldots \cap sort(R_z)$
  - Note: Intersecting sorted runs is cheaper than intersecting arbitrary runs.
- Each DBMSs uses different ways to achieve RID intersection.

Example

- Consider *day<8/9/23 AND bid=5 AND sid=3.*
- If we have a B+ tree index on ***day*** and a hash index on ***sid***, both using Alternative (2)
- Use both indexes (i.e., *day<8/9/23 [B+tree] and sid=3 [Hash Index])*
- Intersect results, retrieve records and then check *bid=5.*

The Projection Operation (Τελεστής Προβολής)

- Consider the projection query $\pi_{sid,bid}$**(Reserves)** listed below.

```
SELECT DISTINCT (R.sid, R.bid)
FROM Reserves R
```

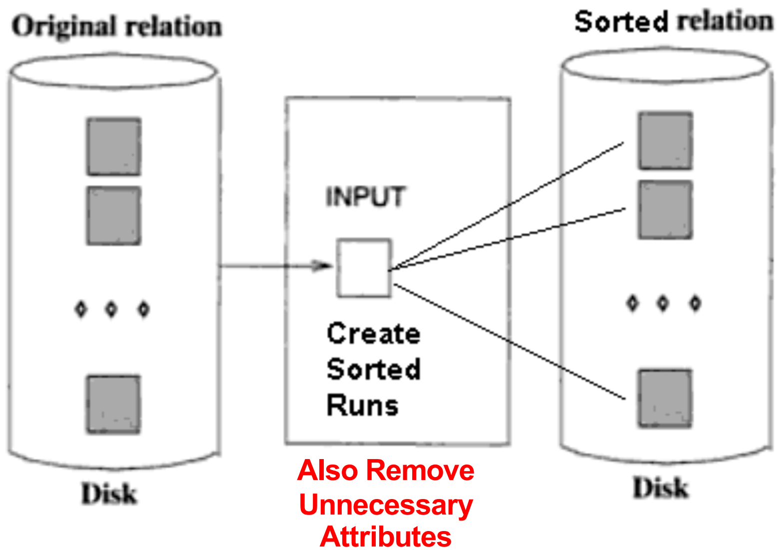
  – recall that in **relational algebra all rows have** to be **distinct** as the **query answer is a set**.
- The projection operator is of the form

$$\pi_{attr1,attr2,\ldots,attrm}(R)$$

- **The implementation requires the following**
  – Remove unwanted columns (on-the-fly)
  – Eliminate any duplicate tuples produced.
    - This step is the difficult one!
- We will describe a technique to cope with **duplicate** elimination based on **Sorting**

Projection Based on Sorting (Προβολή μέσω Ταξινόμησης)

- **First *approach: use External Merge Sort Algorithm***
- **Phase 1: Create Internally Sorted Runs (selected attrib.)**
  - **Scan R** and **produce** a set of tuples that contain only the **desired attributes i.e.,** only <R.sid, R.bid> (First Step of ExternalMergeSort)
    - **Read_Cost: M I/Os** & **Write_Cost: T I/Os,** where T is some fraction of **M** (i.e., depending on fields projected out) **Total: M + T I/Os**
    - **Example:** Assume that **T=250** then **Total Cost:** 1000 + 250 **= 1250 I/Os**



SELECT DISTINCT (R.sid, R.bid)
FROM Reserves R

Projection Based on Sorting (Προβολή μέσω Ταξινόμησης)

- **Phase 2: Merge** tuples using the External Sort Phase 2 based on the projected keys (i.e., <R.sid, R.bid>)
  - **Cost: 2T*(#passes) I/Os** where #passes: $\lceil \log_{B-1} \lceil T/B \rceil \rceil$ (i.e., cost of the External Merge Sort without first step)
  - **Example:** Using **B=20** Buffer pages and **T=250 I/Os**

    Passes: $\lceil \log_{19} \lceil 250/20 \rceil \rceil = 2$ **Total Cost:** 2*250*2 = **1000 I/Os**
- **Step 3: Scan the sorted result** (on-the-fly, consequently costs nothing), comparing adjacent tuples and discard duplicates (could have been carried out during Phase 1-2)
    - e.g., <1,2>, ~~<1,2>~~, <2,2>, <2,3>,
  - **Total Cost: M + T+ (2T * $\lceil \log_{B-1} \lceil T/B \rceil \rceil$)**
  - Using Example: 1000 + 250 + 2*250*2 = 2250 I/Os
  - This step could also have been carried out during steps 1-2.

Use of Indexes for Projections
(Χρήση Ευρετηρίων για Προβολή)

- **So far we have NOT considered using Indexes for Projections**
- If an **existing index** contains all wanted attributes as its search key then we can apply an *index-only* scan.
  – e.g., Q="**SELECT DISTINCT R.rname FROM R**" and **Hash Index** <R.rname> is available.
  – We can use the index to identify the **R.rname** set (i.e., index scan). We must then use **sorting** or **hashing** to eliminate duplicates.
- If an **ordered (i.e., tree) index** contains all wanted attributes as ***prefix*** of search key, can do even better:
  – e.g., Q="**SELECT DISTINCT R.rname FROM R**" and **B+Tree Index** <R.rname> is available.
  – We can use the index to identify R.rname set (index scan) discard unwanted fields, compare adjacent tuples to check for duplicates.
  – We do not even need to apply **sorting** or **hashing for the duplicate elimination part!**