

# Advanced Topics in Distributed Systems

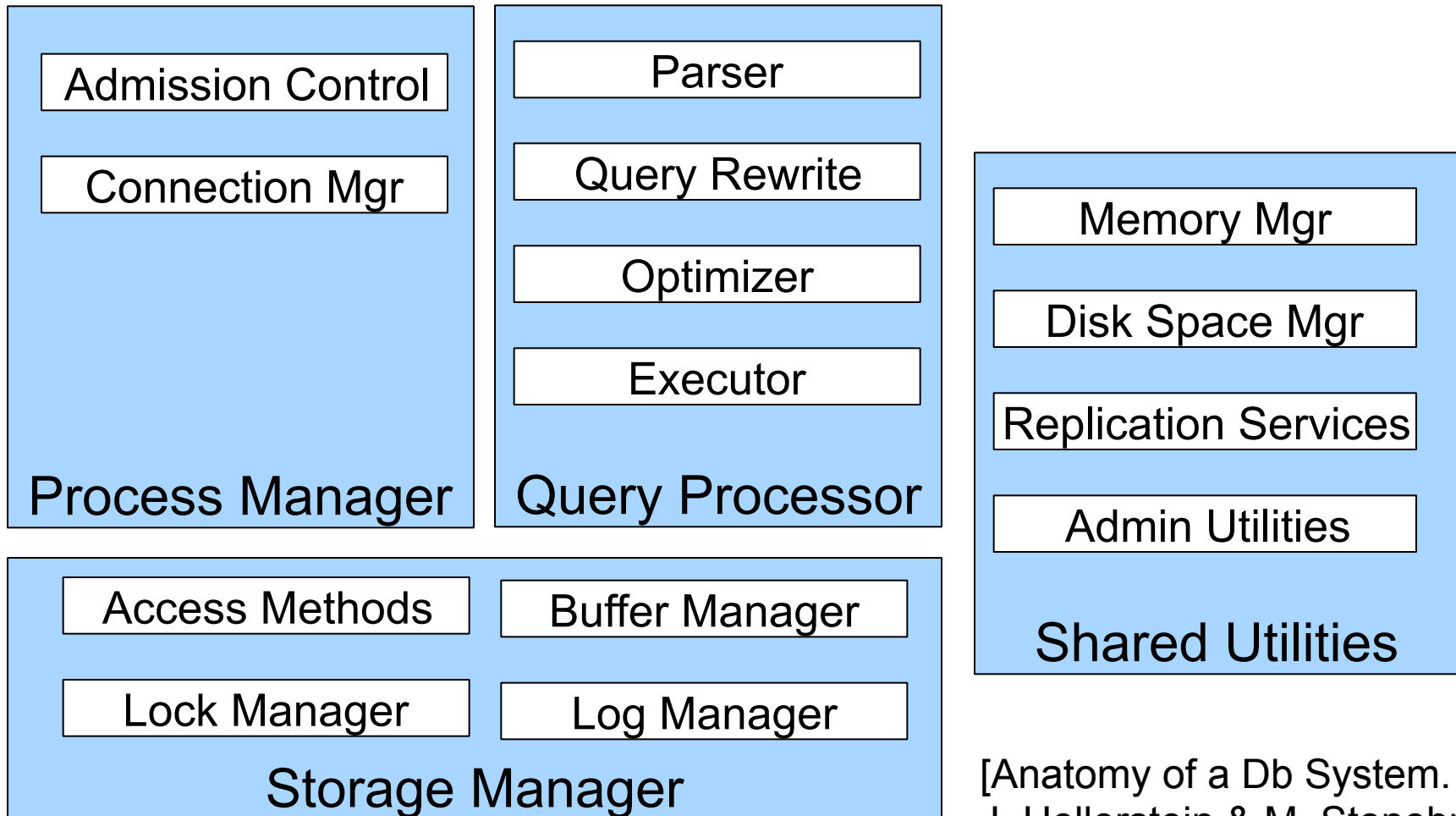
Philippe Cudré-Mauroux

September 2013

Erasmus Mundus Program, KTH--Sweden

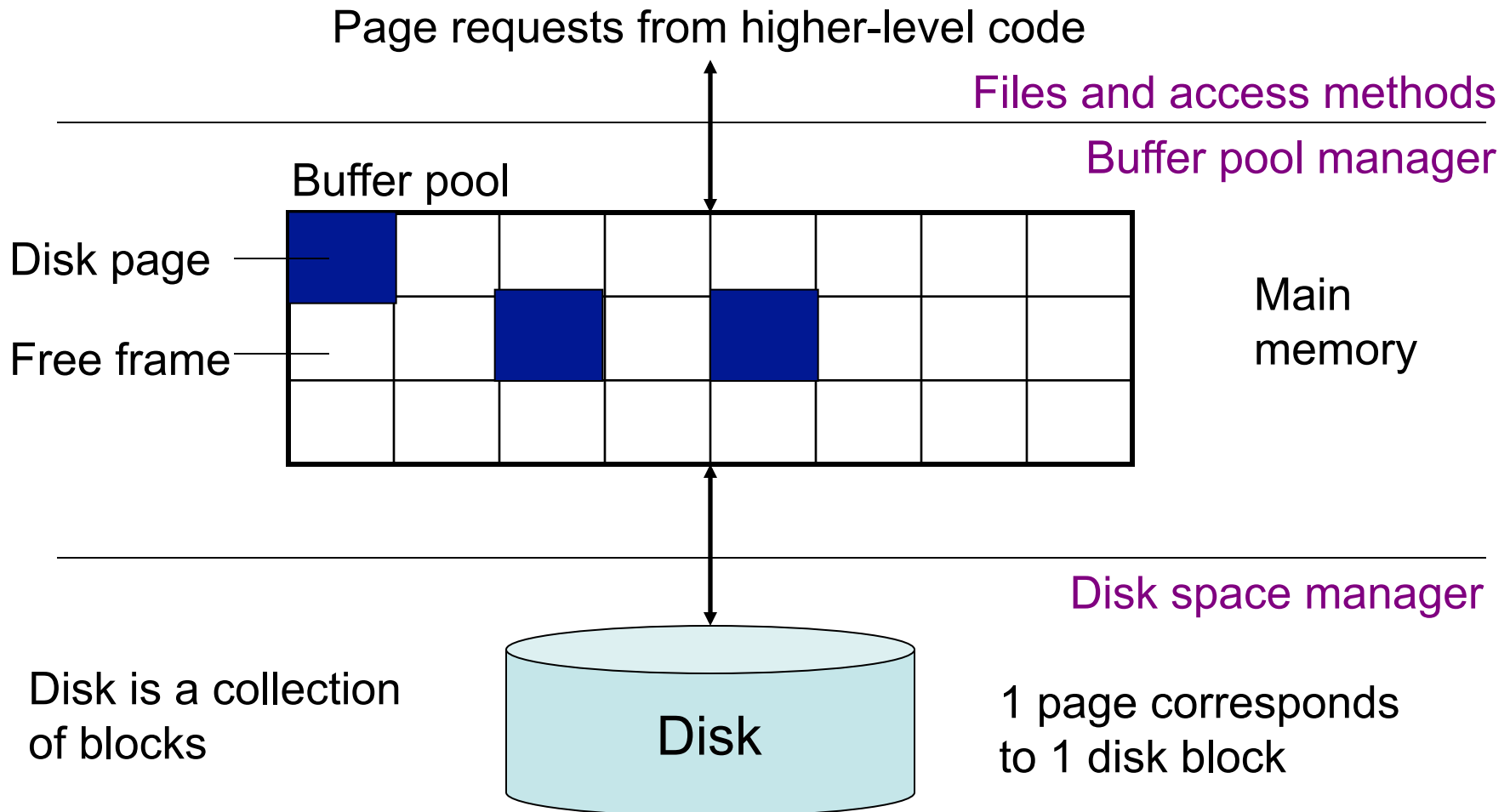
Lecture 2 – DMBS Internals

# DBMS Architecture



[Anatomy of a Db System.  
J. Hellerstein & M. Stonebraker.  
Red Book. 4ed.]

# Buffer Manager



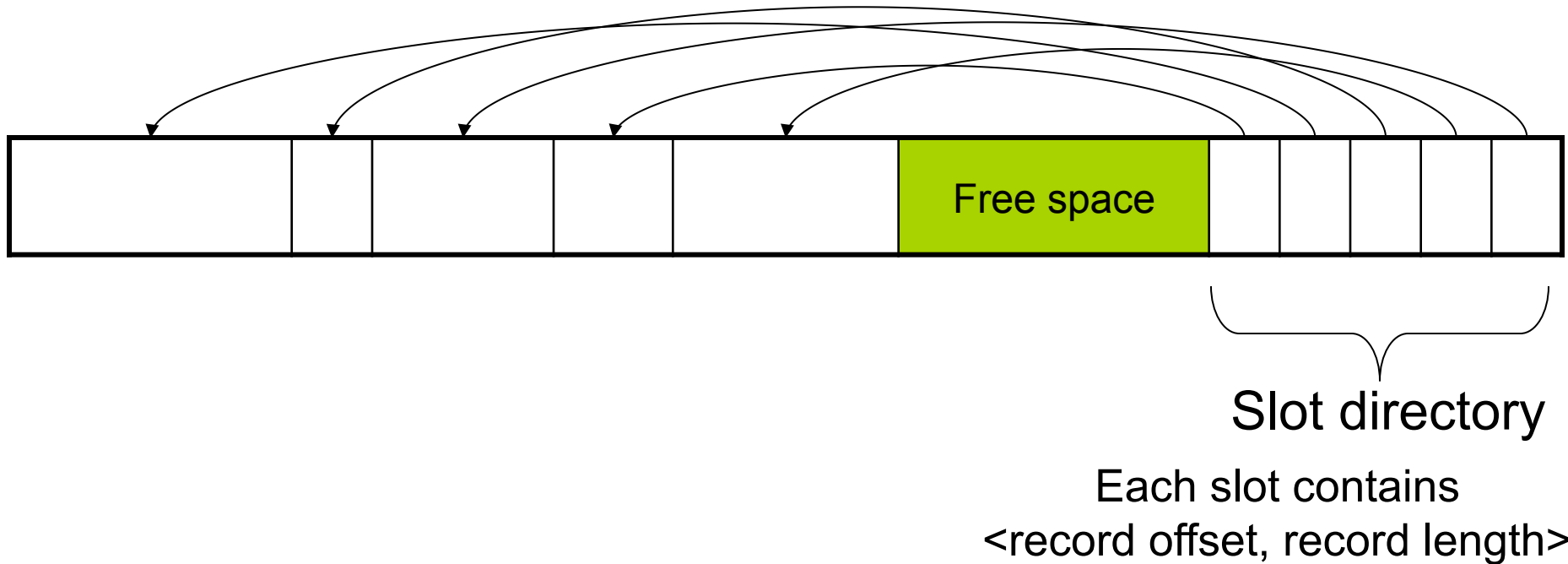
# Data Storage

---

- Basic **abstraction**
  - *Collection of records or file*
  - Typically, 1 relation = 1 file
  - A file consists of *one or more pages*
- How to organize pages into files?
- How to organize records inside a file?
- Classical approach: **n-ary storage**

# Page Format Approach

---

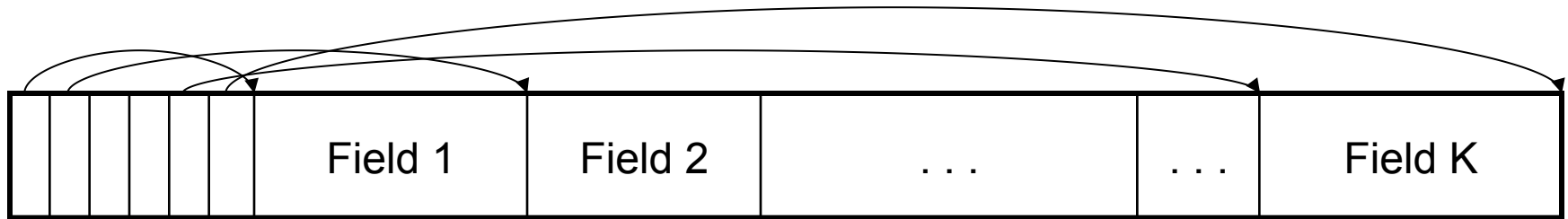


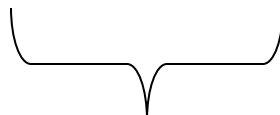
Can handle variable-length records

# Record Formats

---

Variable length records



  
Record header

Remark: NULLS require no space at all (why ?)

# Indexes

---

- **Index**: data structure that organizes data records on disk to optimize selections on the **search key fields** for the index
- An index contains a collection of **data entries**, and supports efficient retrieval of all data entries with a given search key value **k**

# Index Classification Overview

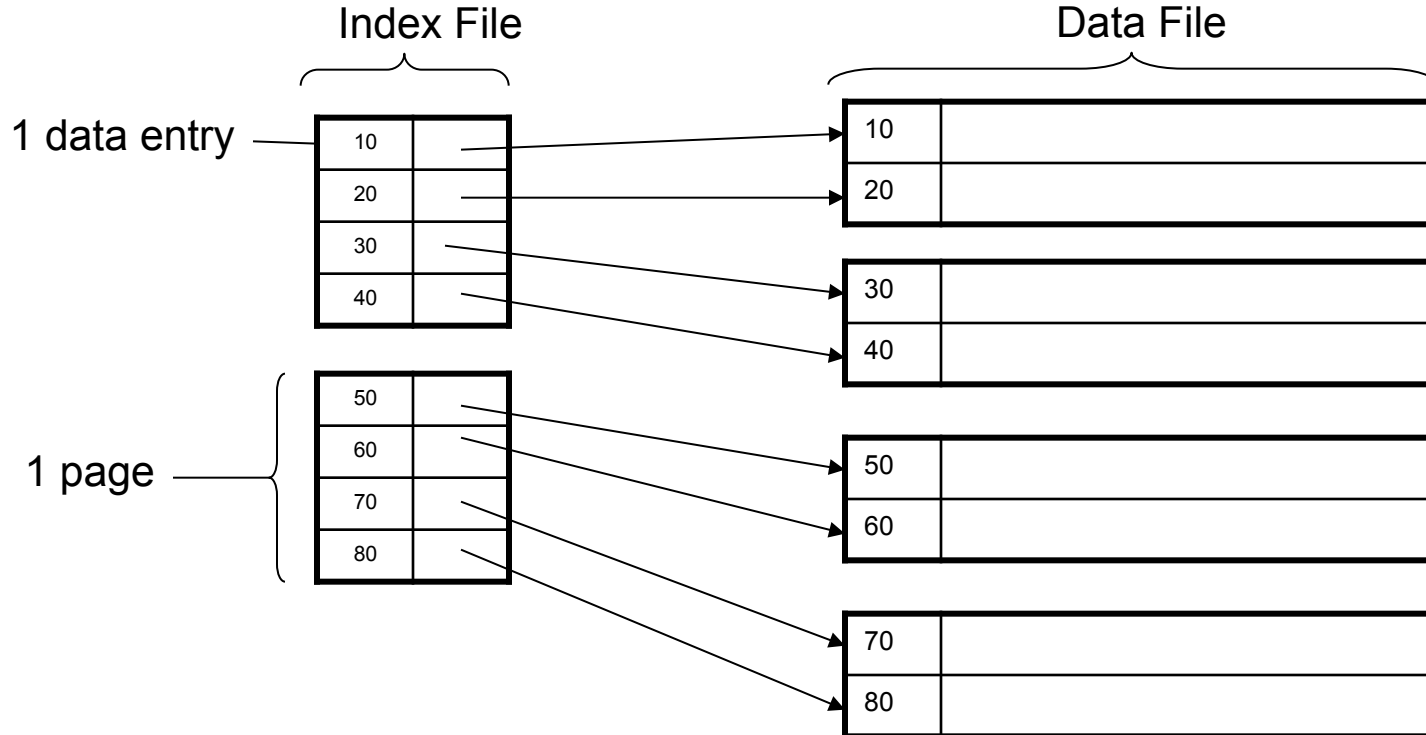
---

- Primary/secondary
  - Primary = determines the location of indexed records
  - Secondary = cannot reorder data, does not determine data location
- Dense/sparse
  - Dense = every key in the data appears in the index
  - Sparse = the index contains only some keys
- Clustered/unclustered
  - Clustered = records close in index are close in data
  - Unclustered = records close in index may be far in data
- B+ tree / Hash table / ...



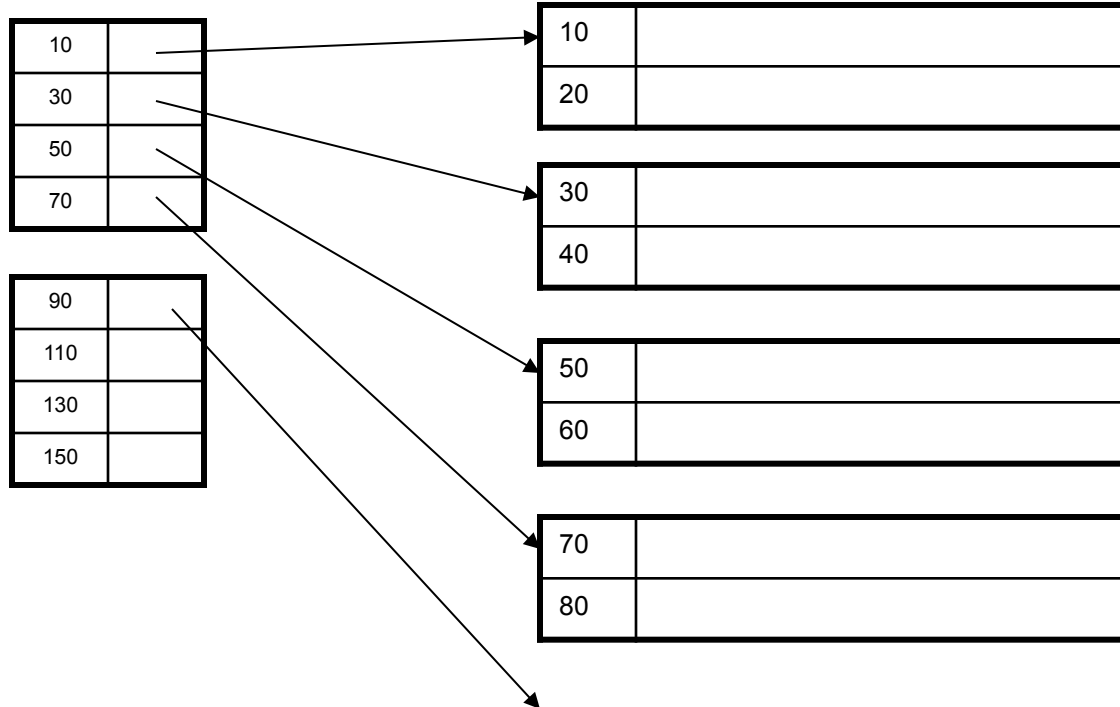
# Primary Index

- Index determines the location of indexed records
- Dense index: sequence of (key,pointer) pairs



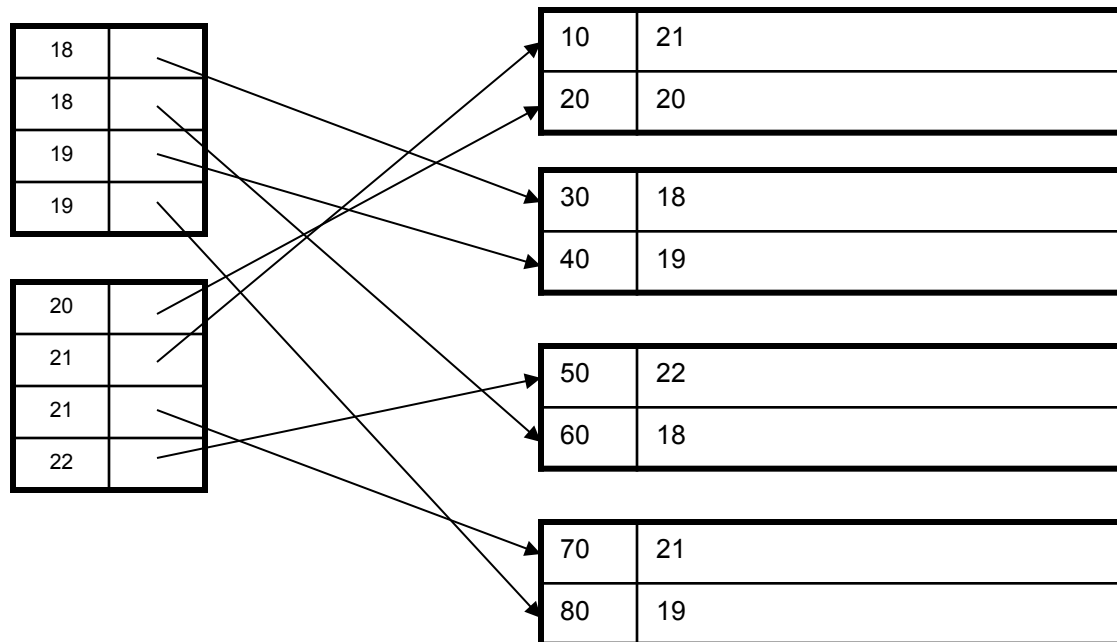
# Primary Index

- Sparse index

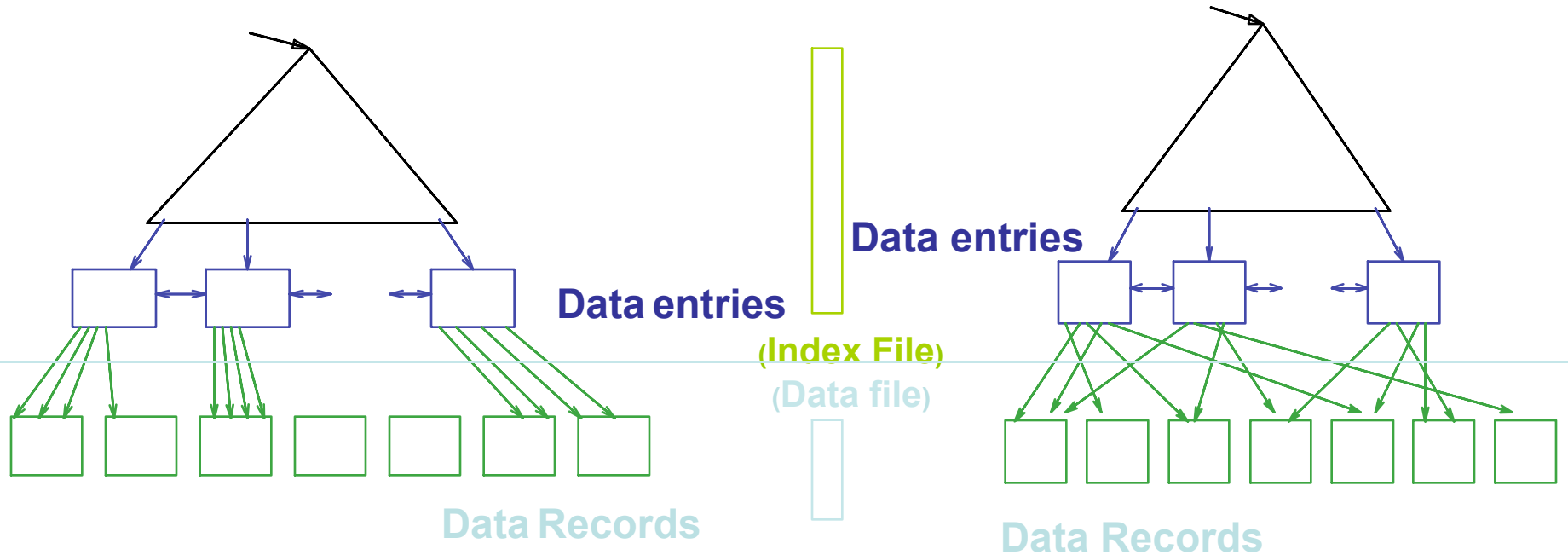


# Secondary Indexes

- To index **other attributes than primary key**
- Always dense (why ?)



# Clustered vs. Unclustered Index



**CLUSTERED**

**UNCLUSTERED**

Clustered = records close in index are close in data

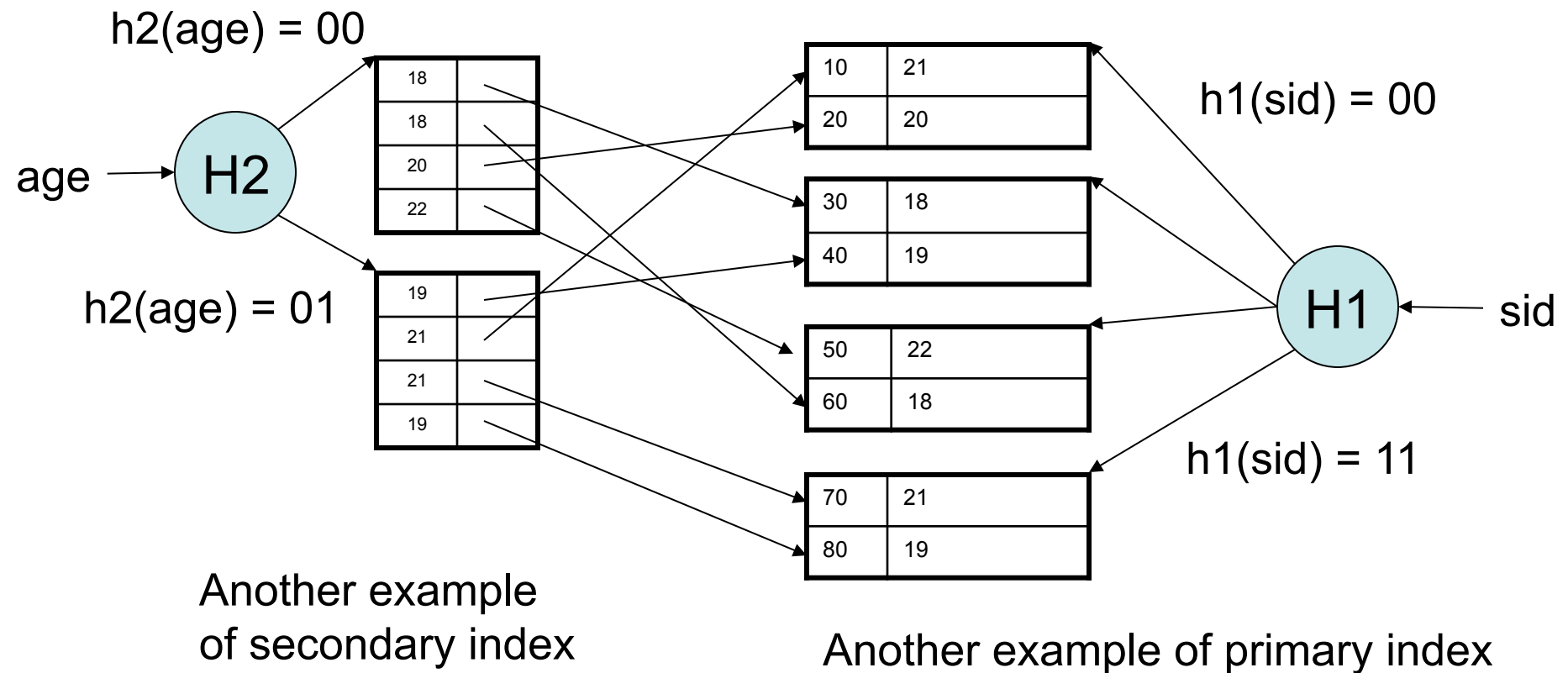
# Clustered/Unclustered

---

- Primary index = clustered by definition
- Secondary indexes = usually unclustered

# Hash-Based Index

Good for point queries but not range queries



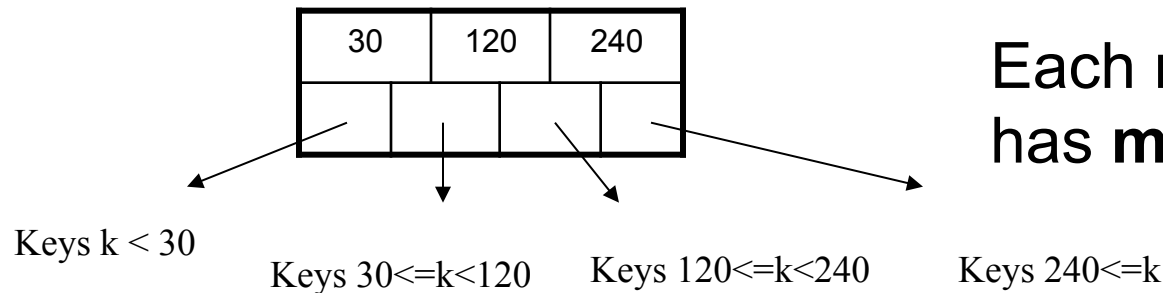
# B+ Trees

---

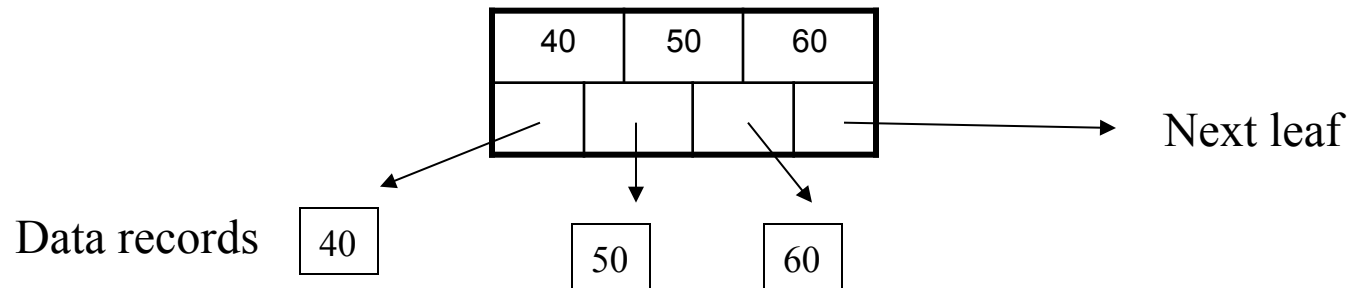
- Search trees
- Idea in B Trees
  - Make 1 node = 1 page (= 1 block)
  - Keep tree balanced in height
- Idea in B+ Trees
  - Make leaves into a linked list : facilitates range queries

# B+ Trees Basics

- Parameter  $d$  = the degree
- Each node has  $d \leq m \leq 2d$  keys (except root)



- Each leaf has  $d \leq m \leq 2d$  keys:

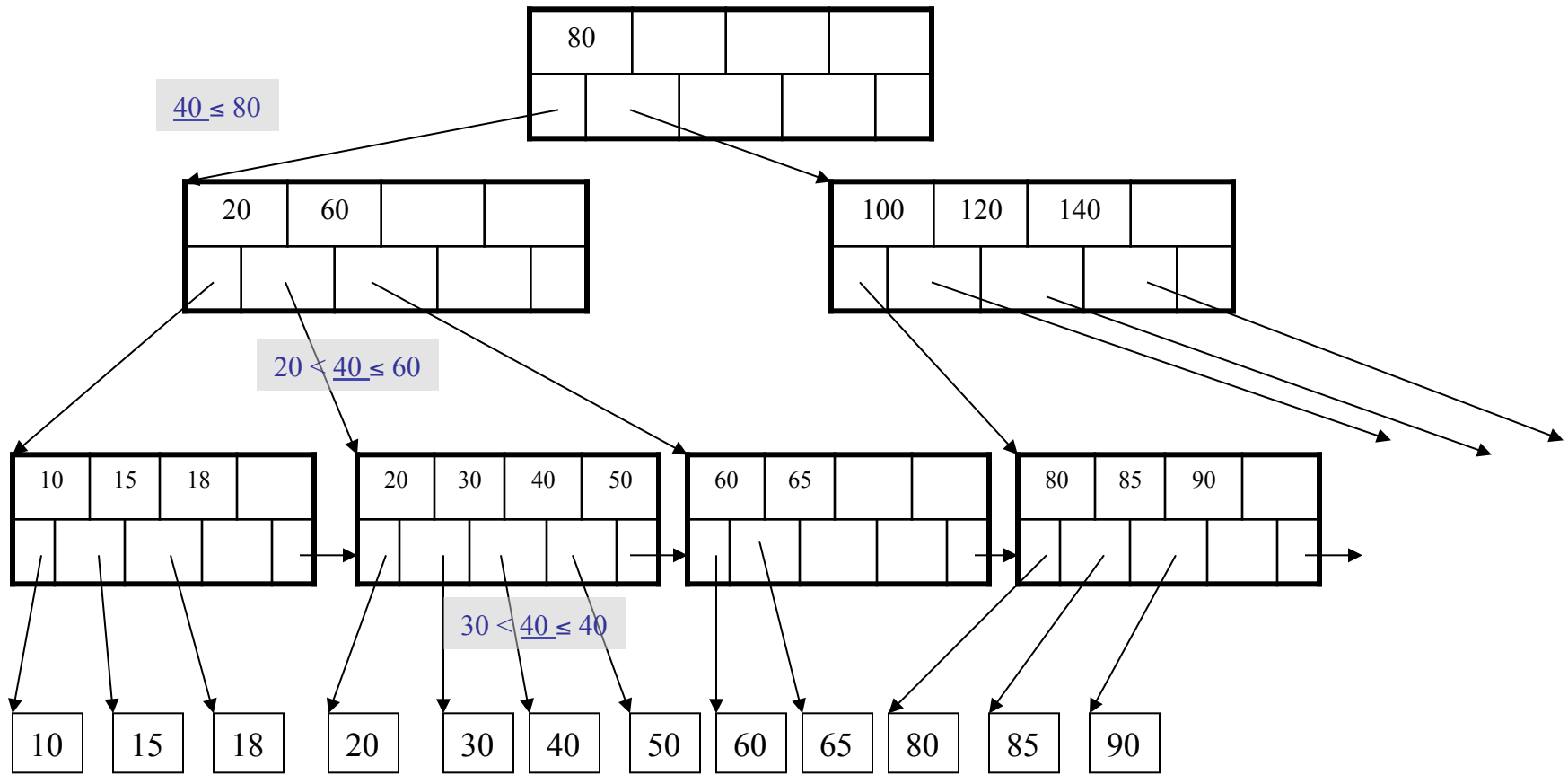




# B+ Tree Example

$d = 2$

Find the key 40



# Searching a B+ Tree

---

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf
- Range queries:
  - Find lowest bound as above
  - Then sequential traversal

```
Select name  
From Student  
Where age = 25
```

```
Select name  
From Student  
Where 20 <= age  
and age <= 30
```

# B+ Tree Design

---

- How large  $d$  ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
- $2d \times 4 + (2d+1) \times 8 \leq 4096$
- $d = 170$

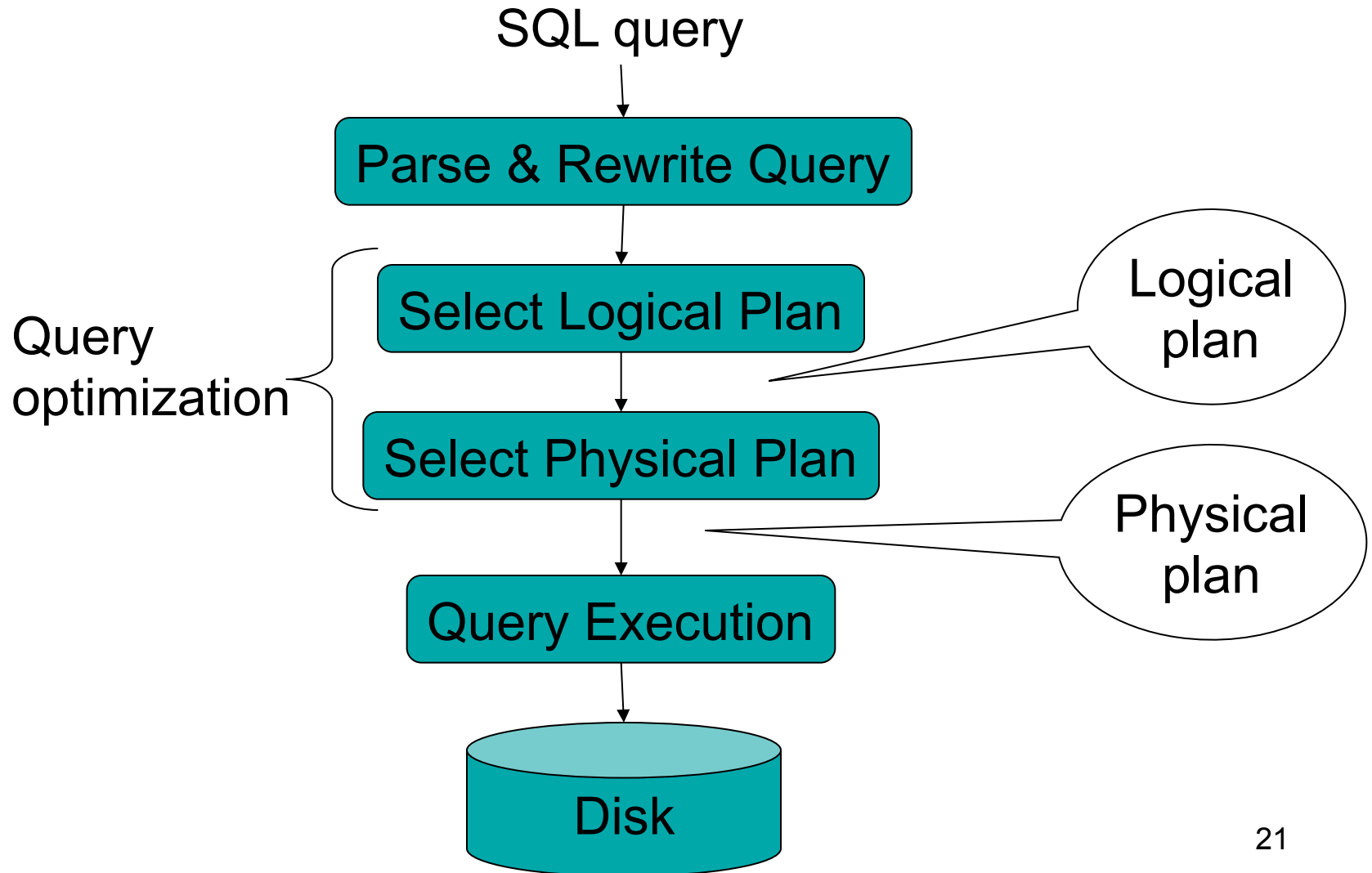
# B+ Trees in Practice

---

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities
  - Height 4:  $133^4 = 312,900,700$  records
  - Height 3:  $133^3 = 2,352,637$  records
- Can often hold top levels in buffer pool
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 Mbytes

# Query Evaluation Steps

---



# Example Database Schema

---

Supplier(sno, sname, scity)

Part(pno, pname, psize, pcolor)

Supply(sno, pno, price)

## View: Suppliers in Stockholm

```
CREATE VIEW NearbySupp AS
```

```
SELECT sno, sname
```

```
FROM Supplier
```

```
WHERE scity='Stockholm'
```

# Example Query

---

- Find the names of all suppliers in Stockholm who supply part number 2

```
SELECT sname FROM NearbySupp
WHERE sno IN ( SELECT sno
                FROM Supplies
                WHERE pno = 2 )
```

# Steps in Query Evaluation

---

- **Step 0: admission control**
  - User connects to the db with username, password
  - User sends query in text format
- **Step 1: Query parsing**
  - Parses query into an internal format
  - Performs various checks using catalog
    - Correctness, authorization, integrity constraints
- **Step 2: Query rewrite**
  - View rewriting, flattening, etc.



# Rewritten Version of Our Query

---

Original query:

```
SELECT sname
FROM NearbySupp
WHERE sno IN ( SELECT sno
                FROM Supplies
                WHERE pno = 2 )
```

Rewritten query:

```
SELECT S.sname
FROM Supplier S, Supplies U
WHERE S.scity='Stockholm'
AND S.sno = U.sno
AND U.pno = 2;
```

# Continue with Query Evaluation

---

- **Step 3: Query optimization**
  - Find an efficient query plan for executing the query
  - We will spend another lecture on this topic
- **A query plan is**
  - **Logical query plan**: an extended relational algebra tree
  - **Physical query plan**: with additional annotations at each node
    - Access method to use for each relation
    - Implementation to use for each relational operator

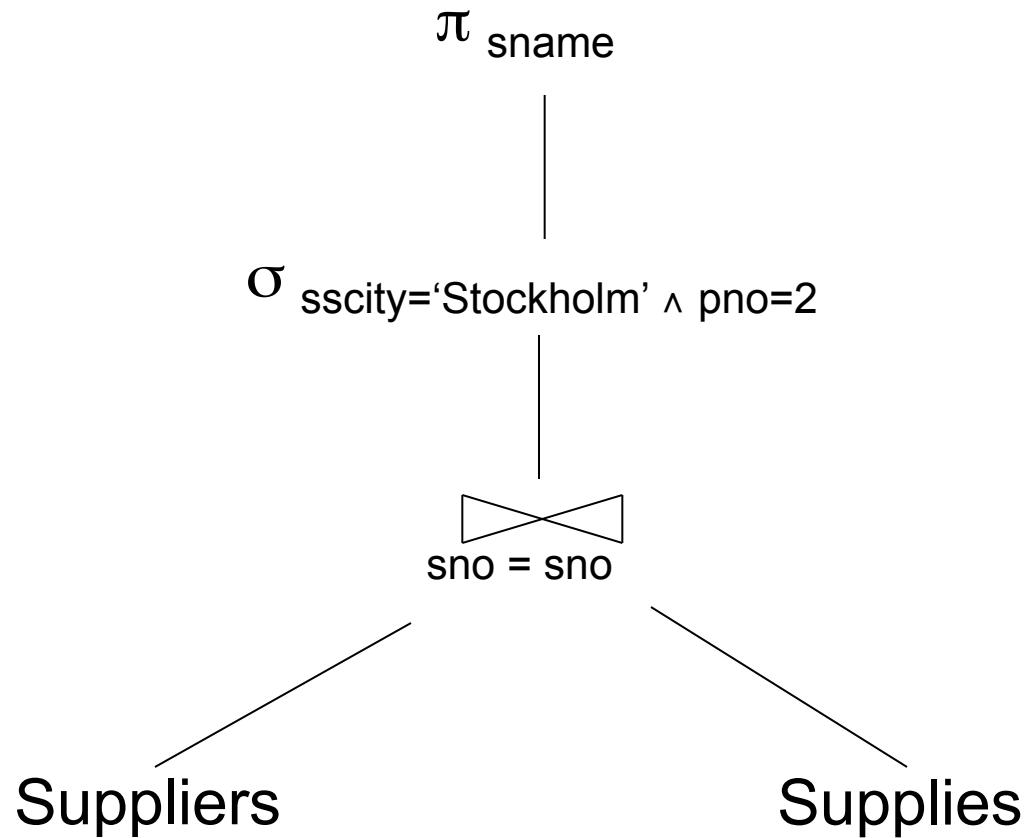
# Extended Algebra Operators

---

- Union  $\cup$ , intersection  $\cap$ , difference  $-$
- Selection  $\sigma$
- Projection  $\pi$
- Join  $\bowtie$
- Duplicate elimination  $\delta$
- Grouping and aggregation  $\gamma$
- Sorting  $\tau$
- Rename  $\rho$

# Logical Query Plan

---



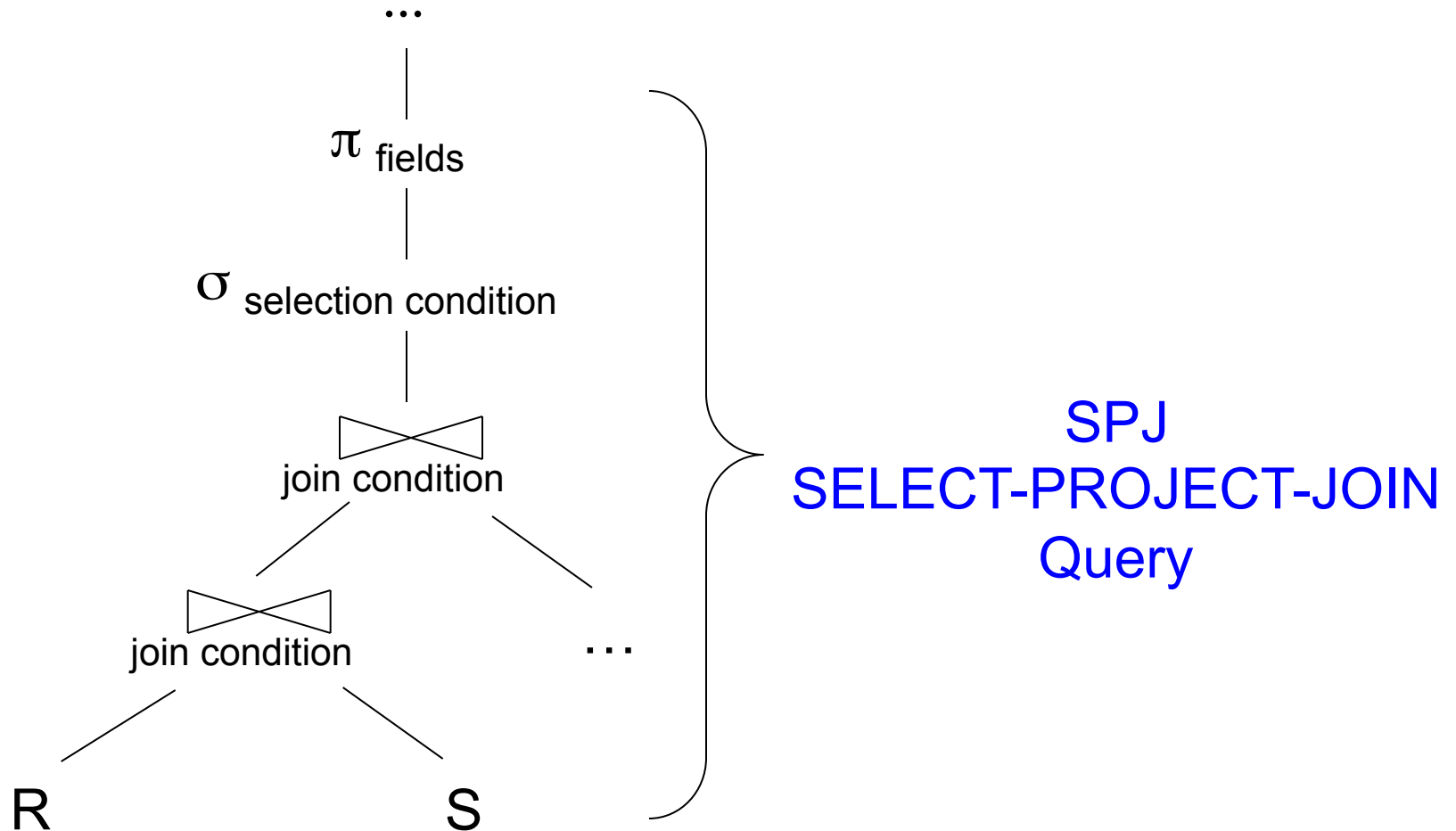
# Query Block

---

- Most optimizers operate on individual query blocks
- A query block is a SQL query with **no nesting**
  - **Exactly one**
    - SELECT clause
    - FROM clause
  - **At most one**
    - WHERE clause
    - GROUP BY clause
    - HAVING clause

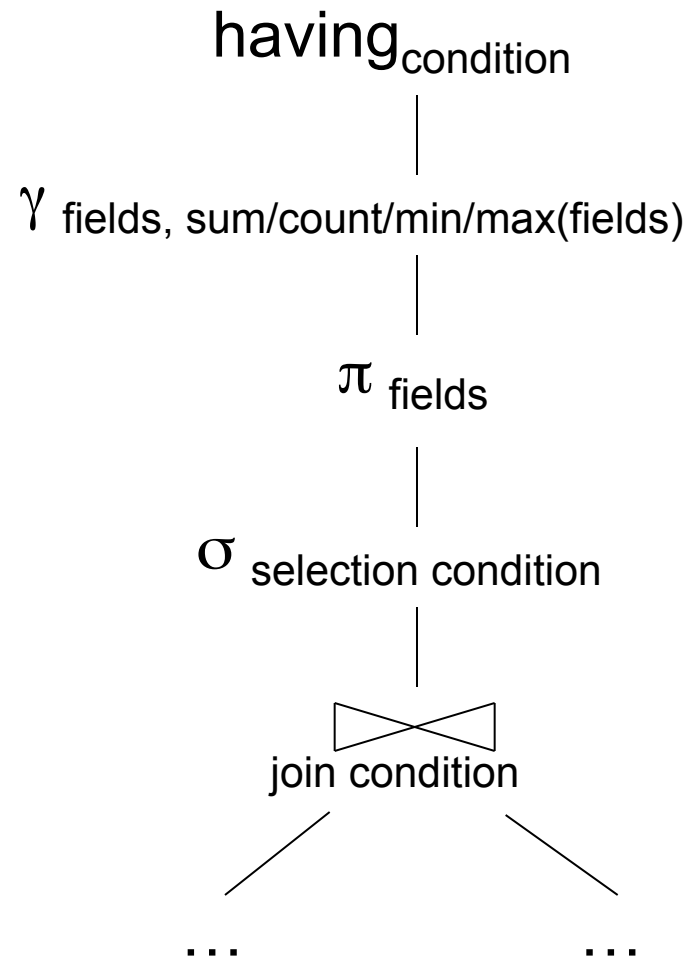
# Typical Plan for Block (1/2)

---



# Typical Plan For Block (2/2)

---



# How about Subqueries?

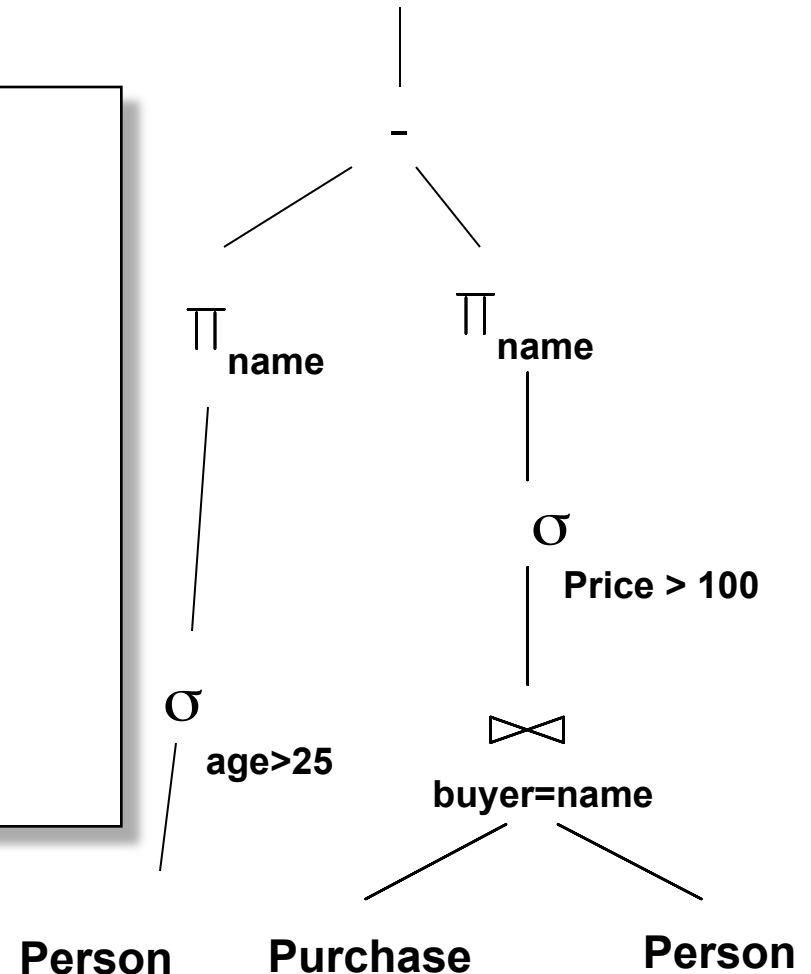
---

```
SELECT Q.name
FROM Person Q
WHERE Q.age > 25
    and not exists
        SELECT *
        FROM Purchase P
        WHERE P.buyer = Q.name
            and P.price > 100
```



# How about Subqueries?

```
SELECT Q.name
FROM Person Q
WHERE Q.age > 25
and not exists
  SELECT *
  FROM Purchase P
  WHERE P.buyer = Q.name
        and P.price > 100
```



# Physical Query Plan

---

- Logical query plan with extra annotations
- **Access path selection** for each relation
  - Use a file scan or use an index
- **Implementation choice** for each operator
- **Scheduling decisions** for operators

# Physical Query Plan

---

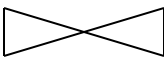
(On the fly)

$\pi_{\text{sname}}$

(On the fly)

$\sigma_{\text{sscity}='Stockholm' \wedge \text{pno}=2}$

(Nested loop)

  
 $\text{sno} = \text{sno}$

Suppliers  
(File scan)

Supplies  
(File scan)

# Final Step in Query Processing

---

- **Step 4: Query execution**
  - How to **synchronize operators?**
  - How to **pass data between operators?**
- Standard approach:
  - **Iterator interface and**
  - **Pipelined execution or**
  - **Intermediate result materialization**

# Iterator Interface

---

- Each **operator implements this interface**
- Interface has only three methods
- **open()**
  - Initializes operator state
  - Sets parameters such as selection condition
- **get\_next()**
  - Operator invokes get\_next() recursively on its inputs
  - Performs processing and produces an output tuple
- **close()**: clean-up state

# Pipelined Execution

---

- Applies parent operator to tuples directly as they are produced by child operators
- Benefits
  - No operator synchronization issues
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk
  - Good resource utilizations on single processor
- This approach is used whenever possible

# Pipelined Execution

---

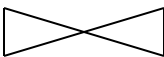
(On the fly)

$\pi_{\text{sname}}$

(On the fly)

$\sigma_{\text{sscity}='Stockholm' \wedge \text{pno}=2}$

(Nested loop)

  
 $\text{sno} = \text{sno}$

Suppliers  
(File scan)

Supplies  
(File scan)

# Intermediate Tuple Materialization

---

- Writes the results of an operator to an intermediate table on disk
- No direct benefit but
- Necessary for some operator implementations
- When operator needs to examine the same tuples multiple times



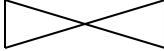
# Intermediate Tuple Materialization

---

(On the fly)

$\pi_{\text{sname}}$

(Sort-merge join)

  
 $\text{sno} = \text{sno}$

(Scan: write to T1)

$\sigma_{\text{sscity}='Stockholm'}$

Suppliers  
(File scan)

(Scan: write to T2)

$\sigma_{\text{pno}=2}$

Supplies  
(File scan)

# Outline

---

- **Steps involved in processing a query**
  - Logical query plan
  - Physical query plan
  - Query execution overview
- **Operator implementations**
  - One pass algorithms
  - Two-pass algorithms

# Why Learn About Op Algos?

---

- Implemented in commercial DBMSs
- Different DBMSs implement different subsets of these algorithms
- Good algorithms can greatly improve performance
- Need to know about physical operators to understand query optimization

# Numbers Everyone Should Know [Google]

---

## Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

# Cost Parameters

---

- In database systems the data is on disk
- **Cost = total number of I/Os**
  - More complex models possible (which ones?)
  - Different models for main-memory / distributed DBMSs (which ones?)
- Parameters:
  - $B(R)$  = # of blocks (i.e., pages) for relation  $R$
  - $T(R)$  = # of tuples in relation  $R$
  - $V(R, a)$  = # of distinct values of attribute  $a$

# Cost

---

- Cost of an operation = number of disk I/Os to
  - read the operands
  - compute the result
- Cost of writing the result to disk is *not included*
  - Need to count it separately when applicable

# Cost of Scanning a Table

---

- Clustered relation:
  - Result may be unsorted:  $B(R)$
  - Result needs to be sorted:  $3B(R)$
- Unclustered relation
  - Unsorted:  $T(R)$
  - Sorted:  $T(R) + 2B(R)$

# One-pass Algorithms

---

Selection  $\sigma(R)$ , projection  $\Pi(R)$

- Both are ***tuple-at-a-time*** algorithms
- Cost:  $B(R)$ , the cost of scanning the relation





# Hash Join

---

Hash join:  $R \bowtie S$

- Scan R, build buckets in main memory
- Then scan S and join
- Cost:  $B(R) + B(S)$
- One pass algorithm when  $B(R) \leq M$

# Nested Loop Joins

---

- Tuple-based nested loop  $R \bowtie S$
- R is the outer relation, S is the inner relation

```
for each tuple r in R do  
    for each tuple s in S do  
        if r and s join then output (r,s)
```

- Cost:  $B(R) + T(R) B(S)$  when S is clustered
- Cost:  $B(R) + T(R) T(S)$  when S is unclustered

# Page-at-a-time Refinement

---

for each page of tuples  $r$  in  $R$  do  
    for each page of tuples  $s$  in  $S$  do  
        for all pairs of tuples  
            if  $r$  and  $s$  join then output  $(r,s)$

- Cost:  $B(R) + B(R)B(S)$  if  $S$  is clustered
- Cost:  $B(R) + B(R)T(S)$  if  $S$  is unclustered

# Two-Pass Algorithms

---

- What if data does not fit in memory?
- Need to process it in multiple passes
  - Two-pass algorithms
  - ... N-pass algorithms

# Summary of Query Execution

---

- For each logical query plan
  - There exist many physical query plans
  - Each plan has a different cost
  - Cost depends on the data
- Additionally, for each query
  - There exist several logical plans
- Query optimization
  - How to compute the cost of a complete plan?
  - How to pick a good query plan for a query?

# Query Optimization Algorithm

---

- For a query
  - There exists many physical query plans
  - Query optimizer needs to pick a good one
- Basic query optimization algorithm
  - Enumerate alternative plans
  - Compute estimated cost of each plan
    - Compute number of I/Os
    - Optionally take into account other resources
  - Choose plan with lowest cost
  - This is called cost-based optimization

# Computing the Cost of a Plan

---

- Collect statistical summaries of stored data
- Compute cost in a bottom-up fashion
- For each operator compute
  - Estimate cost of executing the operation
  - Estimate statistical summary of the output data

# Relational Algebra Equivalences

---

- Selections

- Commutative:  $\sigma_{c_1}(\sigma_{c_2}(R))$  same as  $\sigma_{c_2}(\sigma_{c_1}(R))$
- Cascading:  $\sigma_{c_1 \wedge c_2}(R)$  same as  $\sigma_{c_2}(\sigma_{c_1}(R))$

- Projections

- Cascading

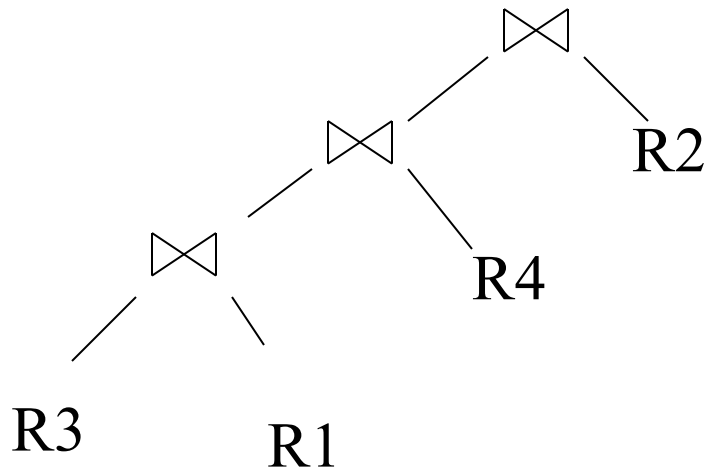
- Joins

- Commutative :  $R \bowtie S$  same as  $S \bowtie R$
- Associative:  $R \bowtie (S \bowtie T)$  same as  $(R \bowtie S) \bowtie T$

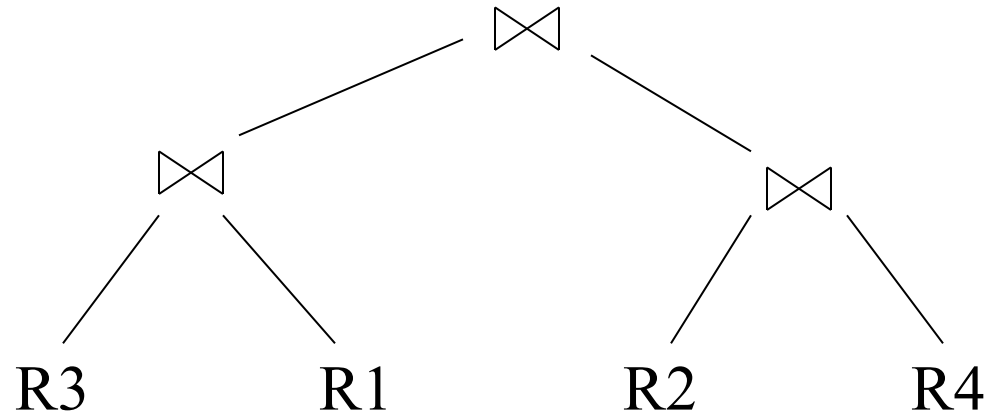


# Left-Deep Plans and Bushy Plans

---



Left-deep plan



Bushy plan

# Search Space Challenges

---

- Search space is huge!
  - Many possible equivalent trees
  - Many implementations for each operator
  - Many access paths for each relation
- Cannot consider ALL plans
- Want a search space that includes low-cost plans

# Readings

---

- **Generalized Search Trees for Database Systems.** J. M. Hellerstein, J. F. Naughton and A. Pfeffer. VLDB 1995.
- **An Overview of Query Optimization in Relational Systems.** S. Chaudhuri. PODS 1998: 34-43
- **The Case for RodentStore: An Adaptive, Declarative Storage System.** P. Cudré-Mauroux, E. Wu, S. Madden. CIDR 2009