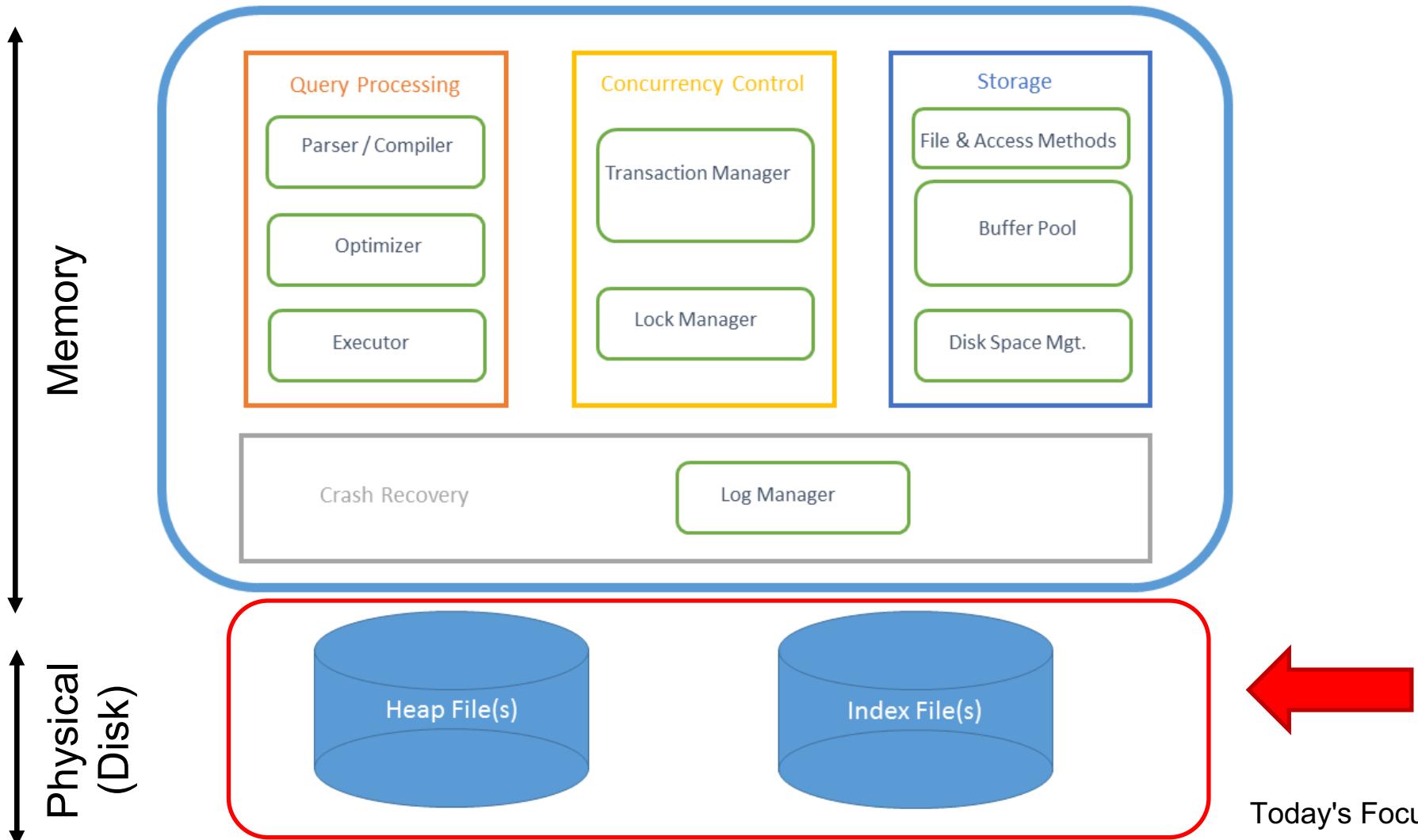


Dr Renata Borovica-Gajic  
David Eccles



# INFO90002 Database Systems & Information Modelling

Lecture 14  
Storage and Indexing



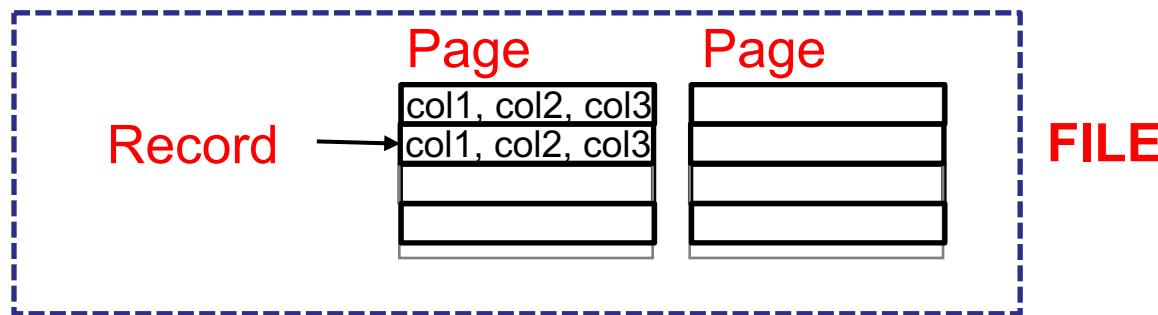


- File organization (Heap & sorted files)
- Index files & indexes
- Index classification

*Readings: Chapter 8, Ramakrishnan & Gehrke, Database Systems*



- **FILE:** A collection of **pages**, each containing a collection of **records**.



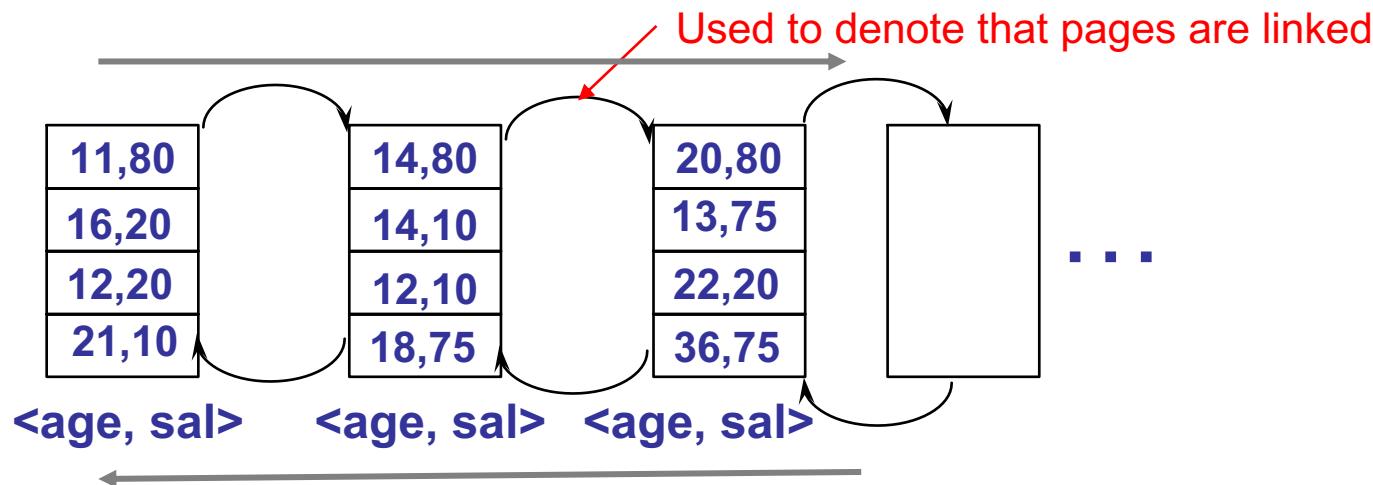
- DBMS must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)



- Many alternatives exist, *each good for some situations, and not so good in others:*
  1. **Heap files:** no particular order among records
    - Suitable when typical access is a file scan retrieving **all** records
  2. **Sorted Files:** pages and records within pages are ordered by some condition
    - Best for retrieval (of a range of records) in some order
  3. **Index File Organizations:**
    - Special data structure that has the fastest retrieval in some order

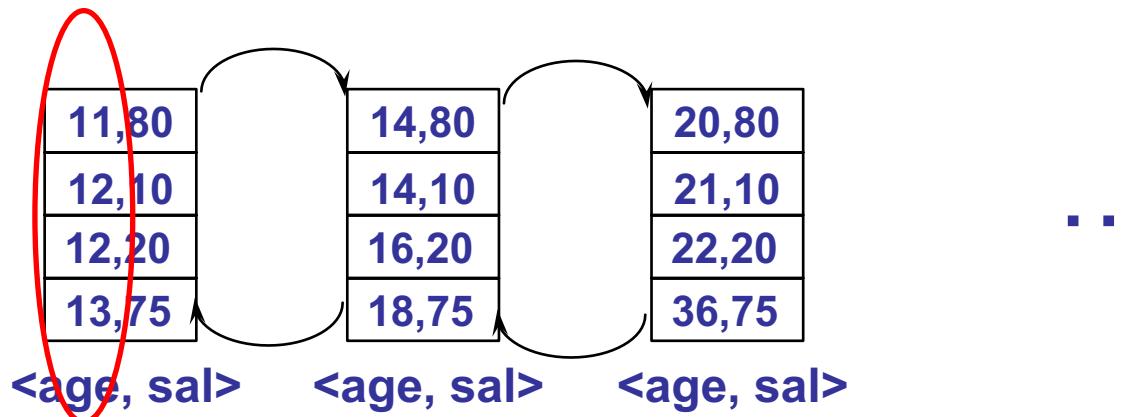


- Simplest file structure, contains records in **no** particular order
- As file grows and shrinks, disk pages are allocated and de-allocated
  - Fastest for inserts compared to other alternatives





- Similar structure like heap files (pages and records), but pages and records are **ordered**
- Fast for range queries, but hard for maintenance (each insert potentially reshuffles records)
- **Example:** A sorted file ordered by age





- DBMS model the **cost** of all operations
- The cost is typically expressed in **the number of page accesses** (or disk I/O operations)
  - 1 page access (on disk) == 1 I/O (used interchangeably)
- **Example:** If we have a table of 100 records, and each page can store 10 records, what would be the cost of accessing the entire file
- **Answer:** For 100 records we have 10 pages in total ( $100/10$ ), thus the cost to access the entire file is 10 I/O (or 10 pages)



- **Example:** Find all records with ages between 20 and 30, for the file that has  $B$  pages. Consider both alternative: having an unsorted and sorted file. What would be the cheapest cost?
- $20 < \text{age} < 30$ , num pages =  $B$ ;

11,80
12,10
52,20
13,75

14,80
14,10
36,75
18,75

20,80
21,10
22,20
16,20

36,80
41,10
12,20
80,75

Heap file

- Sorted file (exploit order) =  $\log_2 B$

11,80
12,10
12,20
13,75

14,80
14,10
16,20
18,75

20,80
21,10
22,20
36,75

36,80
41,10
52,20
80,75

Sorted file



**B:** Number of data pages (NOT records)

	Heap File	Sorted File	Notes
Scan all records	B <i>Average case</i>	B	
Equality Search	0.5B	$\log_2 B$	<i>Assumption (exactly one match)</i>
Range Search	B	$(\log_2 B) + (\#\text{match pages})$	
Insert	2	$(\log_2 B) + 2*(B/2)$	<i>must Read &amp; Write</i>
Delete	$0.5B + 1$	$(\log_2 B) + 2*(B/2)$	<i>must Read &amp; Write</i>

**Heap file faster to insert, Sorted file faster to search**



- Assume 10,000 pages approximately 32,000 records
- Scan all records

```
SELECT *  
FROM employee
```

**Heap:** 10000 I/O **Sorted:** 10000 I/O

- Equality Condition

```
SELECT *  
FROM employee  
WHERE departmentID = 5
```

**Heap:** 5000 **Sorted:**  $\log_2 10000$  (13.2) 14 I/O #

- Range Search

```
SELECT *  
FROM employee  
WHERE salary > 85000  
AND salary < 100000
```

**Heap:** 10000 **Sorted:**  $\log_2 10000$  (13.28) 14 + match pages (e.g 588\*\*) = 602 I/O

# We must take the ceiling value i.e.next highest integer \*\*588 assumes approximately 5.9% of all records fall into this range)

#### Examples of Range Conditions

```
1  SELECT * FROM t1  
2  WHERE key_col > 1  
3  AND key_col < 10;  
4  
5  SELECT * FROM t1  
6  WHERE key_col = 1  
7  OR key_col IN (15, 18, 20);  
8  
9  SELECT * FROM t1  
10 WHERE key_col LIKE 'ab%'  
11 OR key_col BETWEEN 'bar' AND 'foo';
```



- Assume 10,000 pages approximately 32,000 records
- **INSERT**

```
INSERT INTO employee
```

```
VALUES (32983, 'Andrew', 'Dale', 43000, 11, 1, '1975-11-10');
```

**Heap:** 2 **Sorted:**  $\log_2 10000$  (13.28) 14 + 2\*(10000/2) = 10014

- **DELETE**

```
DELETE FROM employee
```

```
WHERE employeeID = 15;
```

**Heap:** 5001 **Sorted:**  $\log_2 10000$  (13.28) 14 + 2\*(10000/2) = 10014

**Heap file faster to INSERT, Sorted file faster to SEARCH**

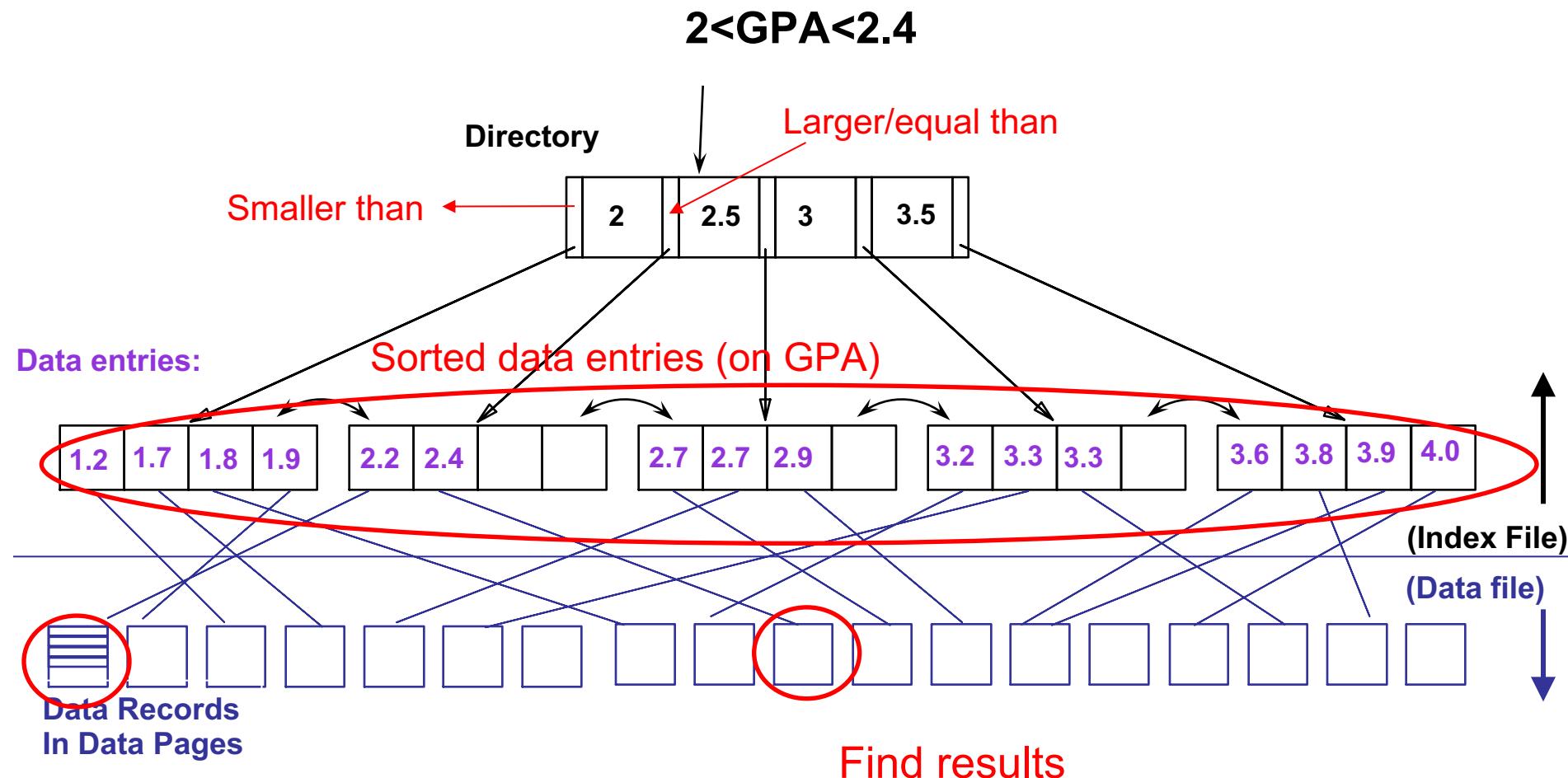
- File organization (Heap & sorted files)
- Index files & indexes
- Index classification



- Sometimes, we want to retrieve records by specifying the *values in one or more fields*, e.g.,
  - Find all students in the “**CIS**” department
  - Find all students with a **gpa > 3**
- An **index** is a data structure built on top of data pages used for efficient search. The index is built over specific fields called **search key fields**. E.g. we can build an index on **GPA**, or department name.
  - The index speeds up selections on the *search key fields*
  - Any** subset of the fields of a relation can be the *search key* for an index on the relation
  - Note:** *Search key* is not the same as **key** (e.g., doesn't have to be unique)



An index contains a collection of **data entries**, and supports efficient retrieval of **data records** matching a given **search condition**



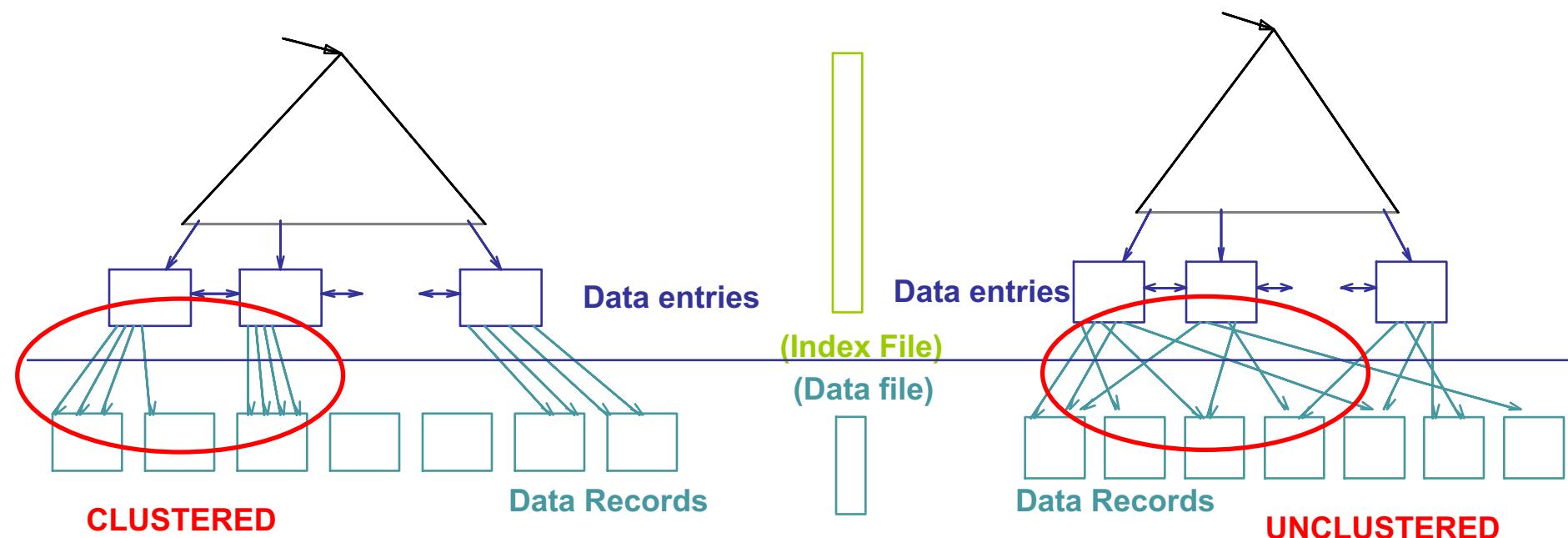
- File organization (Heap & sorted files)
- Index files & indexes
- Index classification

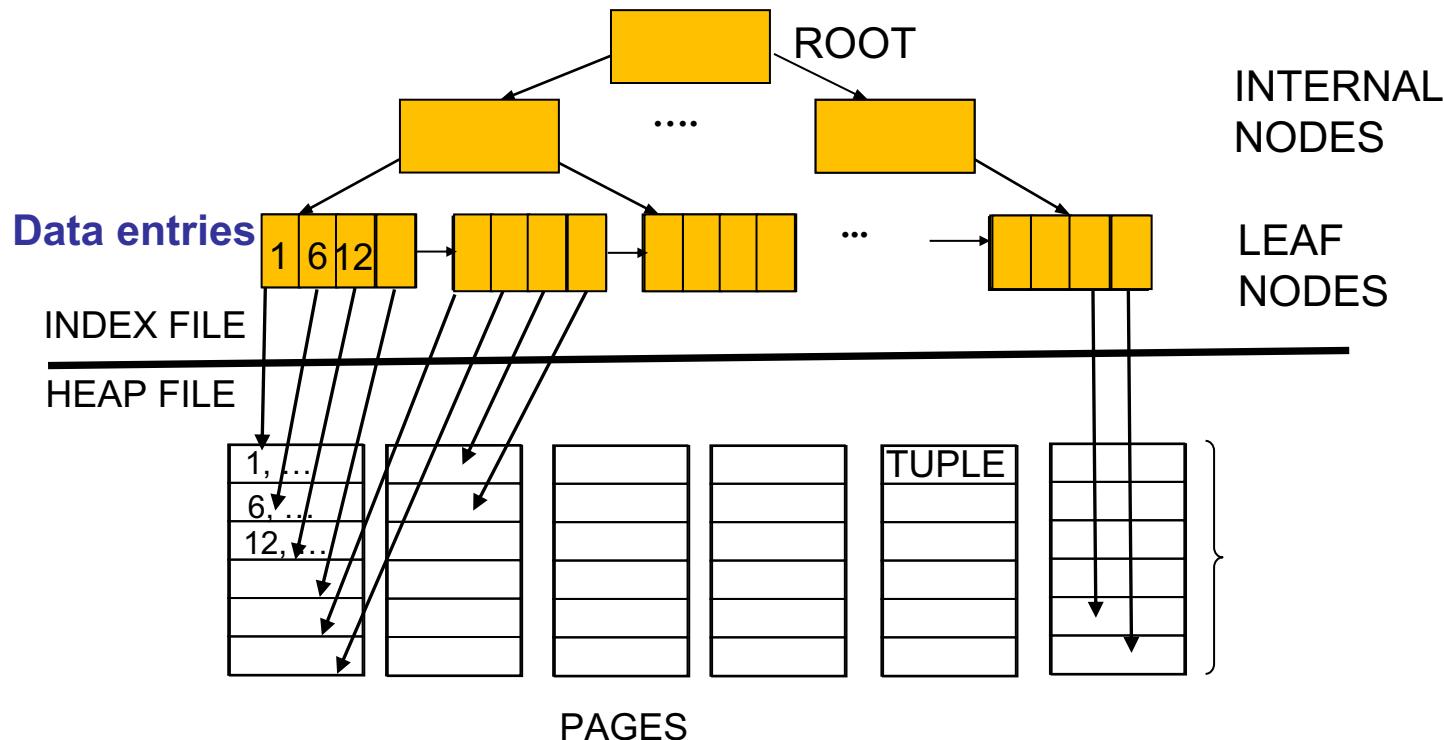


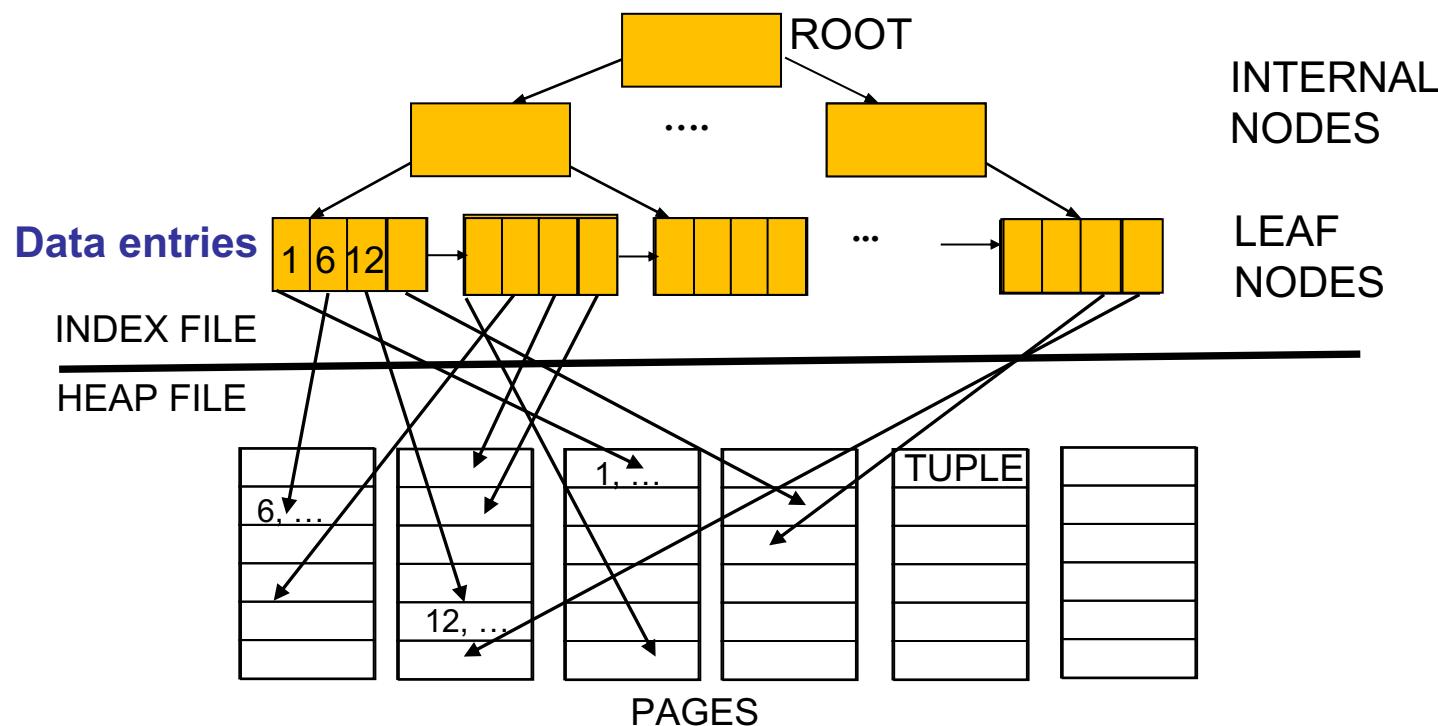
- Classification based on various factors:
  - Clustered vs. Unclustered
  - Primary vs. Secondary
  - Single Key vs. Composite
  - Indexing technique:
    - Tree-based, hash-based, other



- **Clustered vs. unclustered:** If order of **data records** is the same as the order of **index data entries**, then the index is called **clustered index**. Otherwise is **unclustered**.





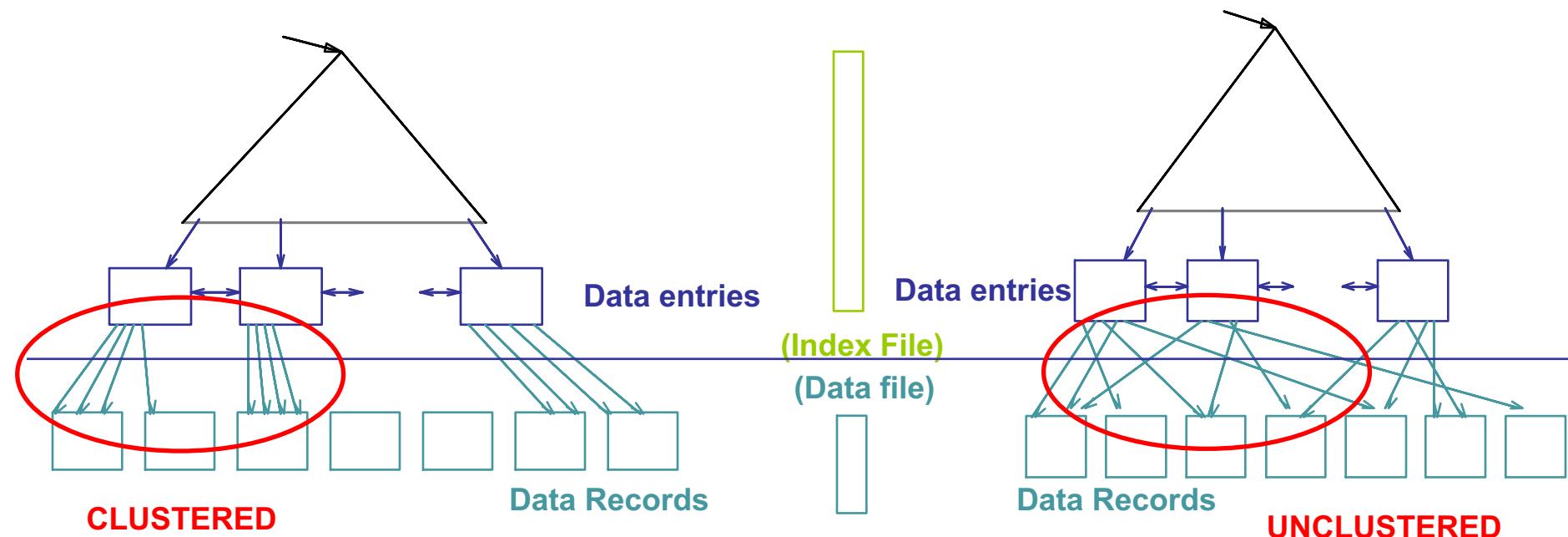




- A data file can have a clustered index on at most **one search key combination** (i.e. we cannot have multiple clustered indexes over a single table)
- Cost of retrieving data records through index varies *greatly* based on whether index is clustered (cheaper for clustered)
- Clustered indexes are more **expensive to maintain** (require file reorganization), but are really efficient for **range search**

www.csse.unimelb.edu.au

- (Approximated) cost of retrieving records found in range scan:
  1. Clustered: cost  $\approx$  # **pages** in data file with matching records
  2. Unclustered: cost  $\approx$  # of matching index **data entries (data records)**

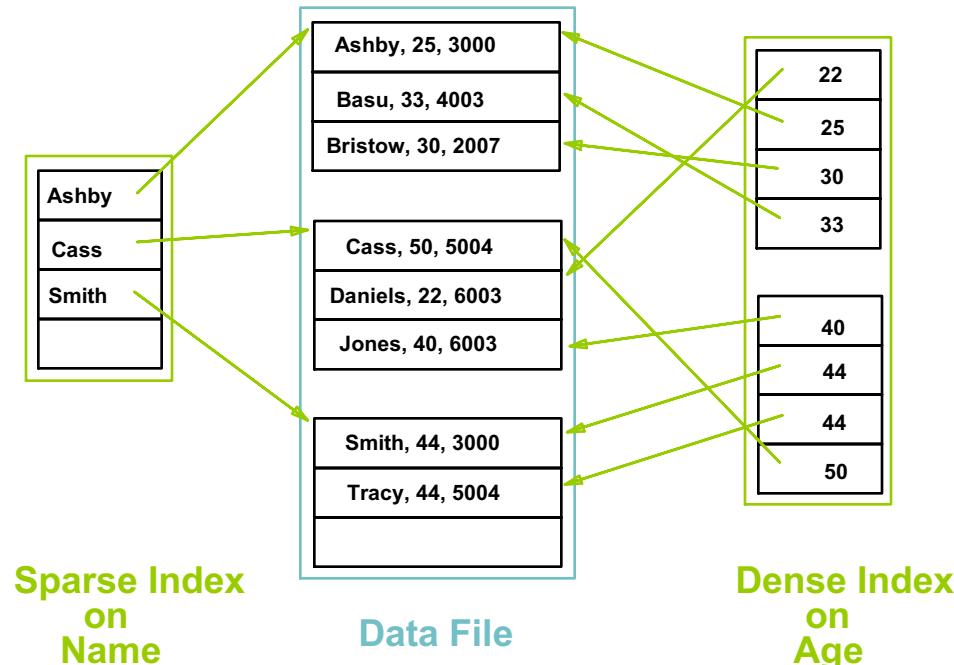




- **Primary** index includes the table's **primary key**
- **Secondary** is any other index
- Properties:
  - Primary index **never** contains duplicates
  - Secondary index **may** contain duplicates

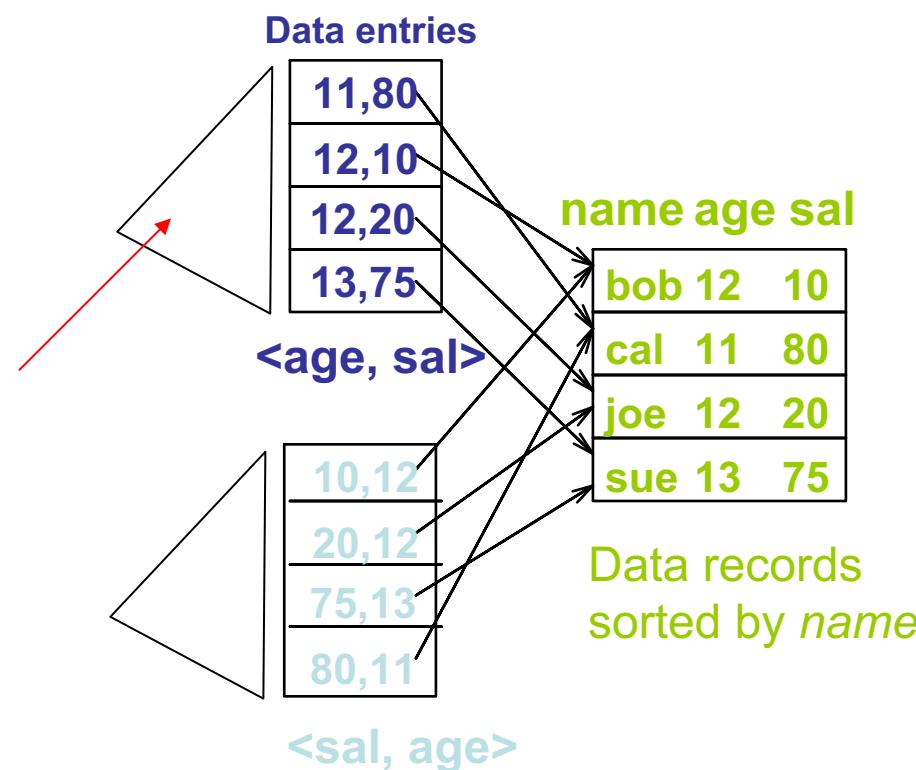


- **Dense:** at least one data entry per key value
- **Sparse:** an entry per data page in file
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.
  - Alternative 1 always leads to dense index.





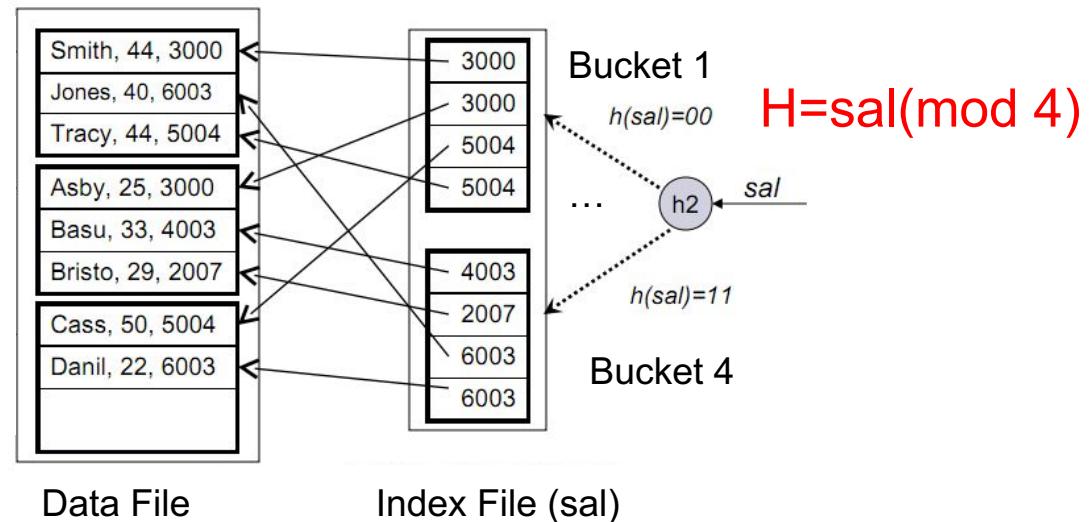
- An index can be built over a combination of search keys
- Data entries in index sorted by search keys
- Examples:
  1. Index on  $\langle \text{age}, \text{sal} \rangle$
  2. Index on  $\langle \text{sal}, \text{age} \rangle$
- 3. Efficient to answer:  
 $\text{age}=12 \text{ and } \text{sal} = 10$   
 $\text{age}=12 \text{ and } \text{sal} > 15$





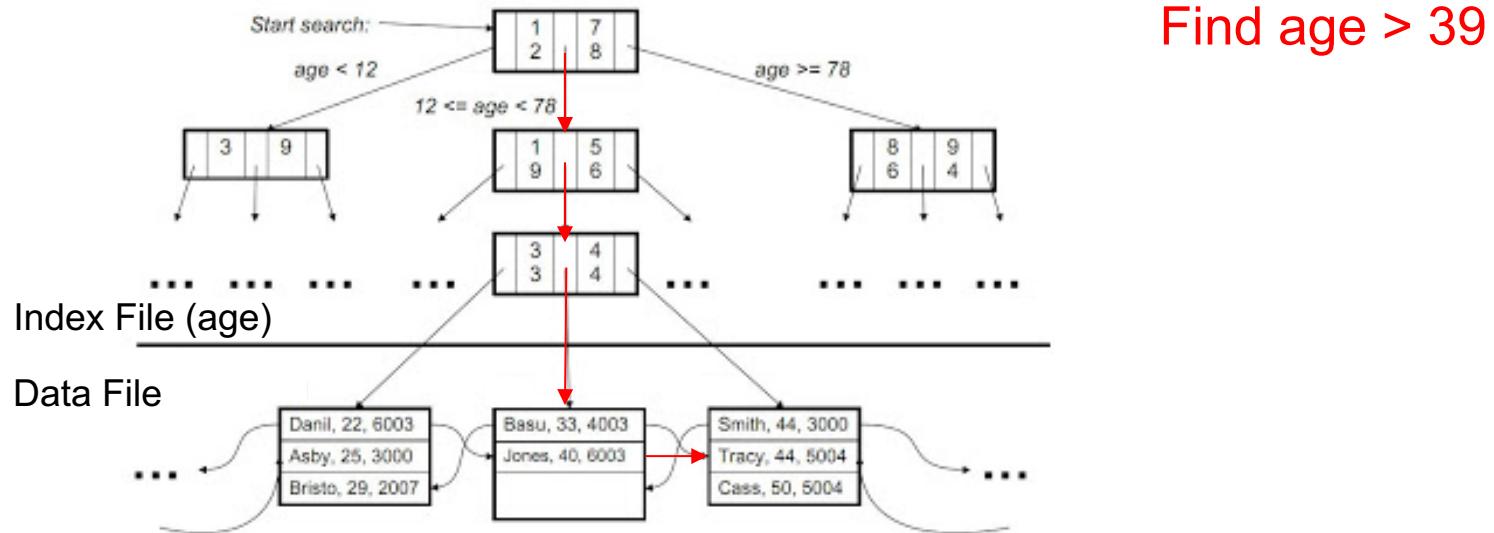
- Hash-based index:
  - Represents index as a collection of *buckets*. Hash function maps the search key to the corresponding bucket.
    - $h(r.\text{search\_key}) = \text{bucket}$  in which record  $r$  belongs
  - Good for **equality** selections
- **Example:** Hash-based index on (sal)

Find Sal = 2007  
 $2007 \bmod 4 = 3$  go to Buck.4





- Tree-based index:
  - Underlying data structure is a binary (B+) tree. Nodes contain pointers to lower levels (search left for lower, right for higher). Leaves contain data entries sorted by search key values.
  - Good for **range** selections
  - So far we have shown those
- **Example:** Tree-based index on (age)





- Many alternative file organizations exist, each appropriate in some situation
- If selection queries are frequent, sorting the file or building an *index* is important
- Index is an additional data structure (i.e. file) introduced to quickly find entries with given key values
  - Hash-based indexes only good for equality search
  - Sorted files and tree-based indexes best for range search; also good for equality search
  - Files rarely kept sorted in practice (because of the cost of maintaining them); B+ tree index is better



INFO90002 Database Systems & Information Modelling

- Describe alternative file organizations
- What is an index, when do we use them
- Index classification

\* All material is examinable – these are the suggested key skills you would need to demonstrate in an exam scenario



- DBMS Architecture & Distributed Databases