INFOMDW: Data Wrangling

Hashing and Indexing

Hakim Qahtan

Part of the slides were prepared by

Yannis Velegrakis

Department of Information and Computing Sciences

Utrecht University

Utrecht University

Reading Material for Today

Database System Concepts (7th Edition)

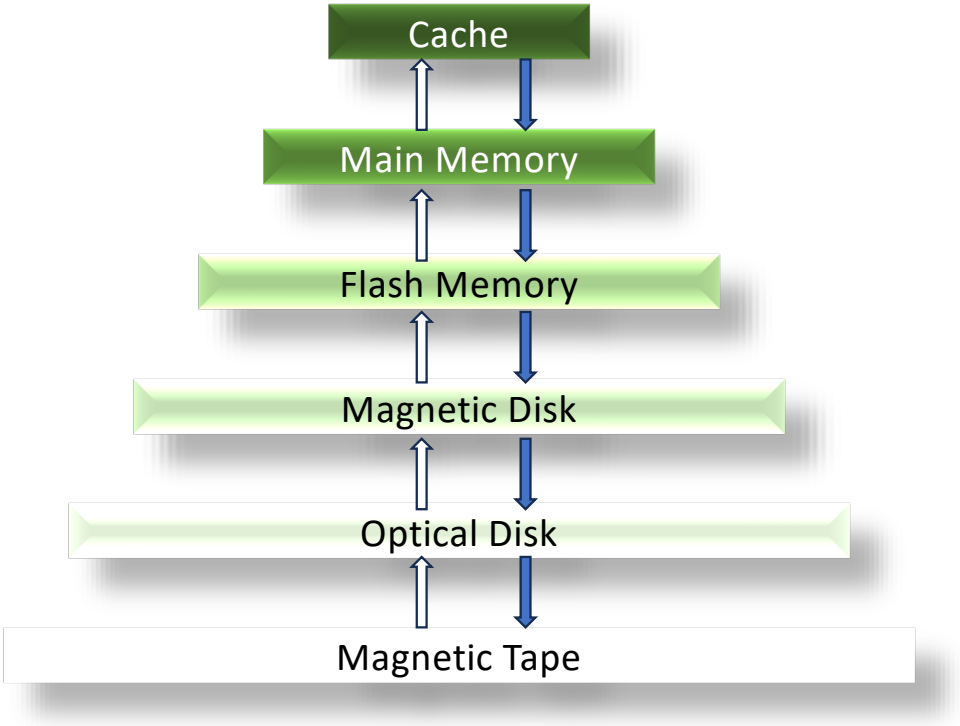
CH 14.1 – 14.7

Office Hours: upon request by email

Retrieving stored records

- Speed of retrieving records depends on
  - The storage media
  - The searching algorithm (supported by data structures)

Storage Hierarchy



Source: Database System Concepts – 7th Edition

Searching Time Complexity

- Sequential search: $O(n)$
  - Linked lists and unsorted arrays
- Binary search: $Olog(n)$
  - Sorted arrays and binary trees
- Direct access: $O(1)$
  - Hash tables

Hashing

Hash Tables: Basic Idea

- Use a key (arbitrary string or number) to index directly into an array – O(1) time to access records
  - A[“brand”] = “Ford”
  - Need a *hash function* to convert the key to an integer

	Key	Data
0	brand	ford
1	color	orange
2	kiwi	Australian fruit

Hashing Techniques

- Division (modulo hashing)
- Multiplication hashing
- Folding Hashing
- Mid-Square Hashing

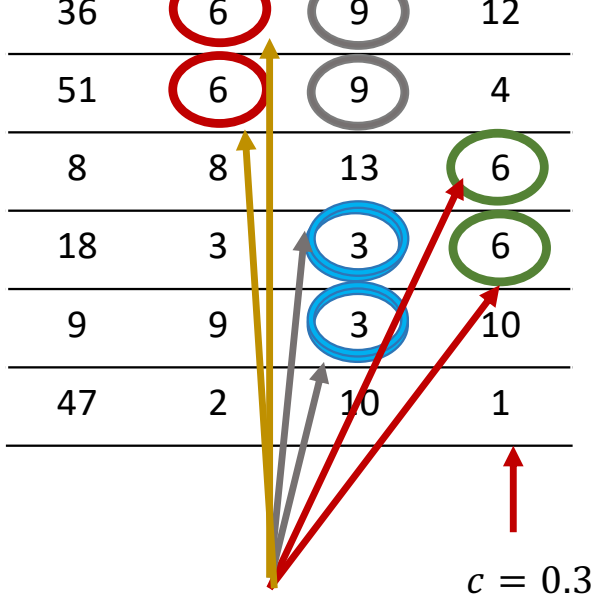
Multiplication and Division Hashing

- For integer keys:
  - x is the key and m is the table size
  - Division (modulo) hashing
    - $h_1(x) = x \,\%\, m$ (% is the modulus function)
    - $h_2(x) = x(x+3) \,\%\, m$
  - Multiplication hashing function
    - Select $0 < c < 1$ and compute $w = xc$
    - Take $u = fraction\ part\ of\ w$
    - $h_3(x) = \lfloor um \rfloor$

- Collisions happen when two different keys are mapped to the same hash value

$m = 15$

x	$h_1(x)$	$h_2(x)$	$h_3(x)$
36	6	9	12
51	6	9	4
8	8	13	6
18	3	3	6
9	9	3	10
47	2	10	1



Collisions

Mid-Square and Folding Hashing

- x is the key and m is the table size
  - Mid-square hashing
    - Take x^2
    - Select a number of digits in the middle of the x^2
    - If the number if larger than the table size, take the modulus
  - Folding hashing function
    - Split the key into chunks with the same number of digits as the digits of the table size
    - Add these chunks together and take the modulus if the summation is greater than the window size

$m = 15$

x	$h_{ms}(x)$	x	$h_f(x)$
36	14	2374	7
51	0	1522	7
8	4	356	11
18	2	1742	14
9	6	428	5
47	5	1877	5

h_{ms} = hashing using mid-square
h_f = hashing using folding

Hashing Noninteger Keys

- For noninteger keys: transform the keys into integers
- Floating point:
  - take the integer part
  - Take the ceiling, floor or round the number to the closest integer
- Character strings
  - Define a function $code(x[i])$, for every character in the string key x
  - For example, use $\text{ascii}(x[i])$, which returns the ascii code associated with each character

Hashing String Keys

- For string keys: x is the key and m is the table size
  - $h_1(x) = sum\big(ascii(x[i])\big)\%\, m,\ \ 0 \le i < length(x)$
  - $h_2(x) = \prod_{0 \le i < length(x)} \big(ascii(x[i])\big)\%\, m$
    - Problem: string with the same set of characters hash to the same number (`abc', `bca', `acb', …)
  - Solution: consider the string to be integer with base 128
    - $h_3(x) = sum\big(ascii(x[i]) * 128^i\big)\ \%\, m,\ \ 0 \le i < length(x)$
    - Problem: values become very big for long strings

Examples

- use h_1, h_2, and h_3 to hash the strings ``ace", ``aec" (table size $m = 15$)
  - $h_1(abc) = (97 + 99 + 101)\%15 = 297\ \%15 = 12$
  - $h_1(acb) = 297\ \%15 = 12$
  - $h_2(abc) = (97\%15) * (99\%15) * (101\%15) = 693\ \%15 = 3$
  - $h_2(acb) = 693\ \%15 = 3$
  - $h_3(abc) = \big((97 * 128^2) + (99 * 128) + (101 * 1)\big)\%15 = 6$
  - $h_3(acb) = \big((97 * 128^2) + (101 * 128) + (99 * 1)\big)\%15 = 5$

Characteristics of a Good Hashing Function

- Returns an integer between 0 and the table size
- Efficiently computable
- Does not waste extra space
- At least one key is hashed to every integer between 0 and the table size
- Minimizes the collisions

Indexing

Link to Video

Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is NOT the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.
  - Given data entry k*, we can find record with key k in at most one disk I/O. (Details soon …)

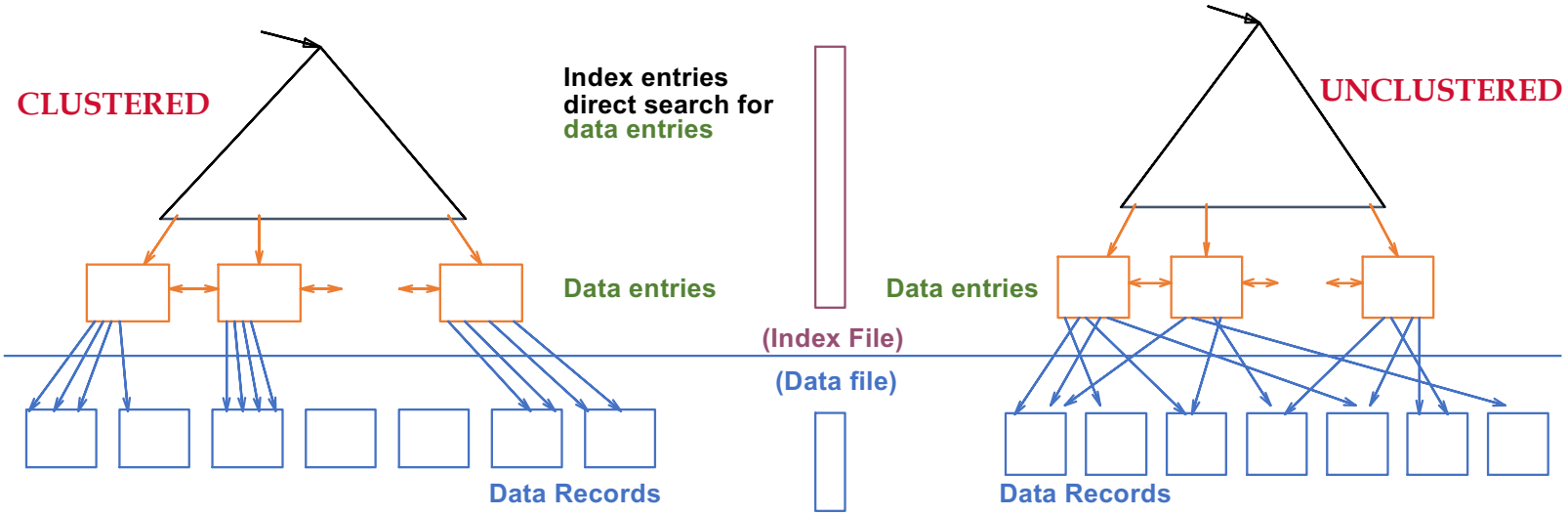
Basics

- An *index* file contains records of the form (search-key, pointer)
  - Should be much smaller than the data file
  - Two kinds of indices:
    - Ordered indices: index entries are sorted
    - Hash indices: index entries are determined using hash function

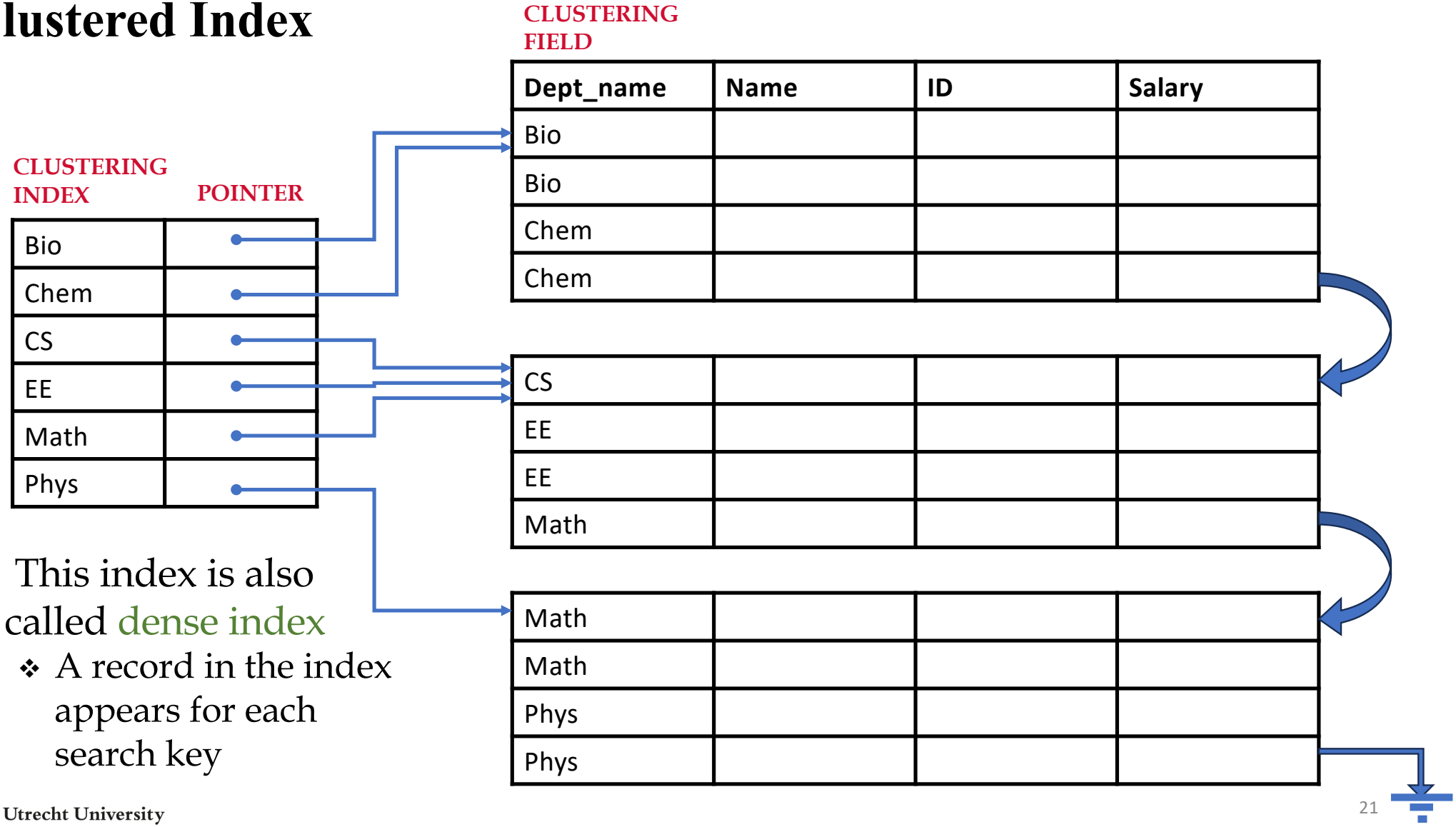
Index classification

- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*:
  - Some database systems allow declaring one of the indices to be clustered.
  - The system stores the relation sorted by the search-key of the clustered index

Clustered vs. Unclustered Index

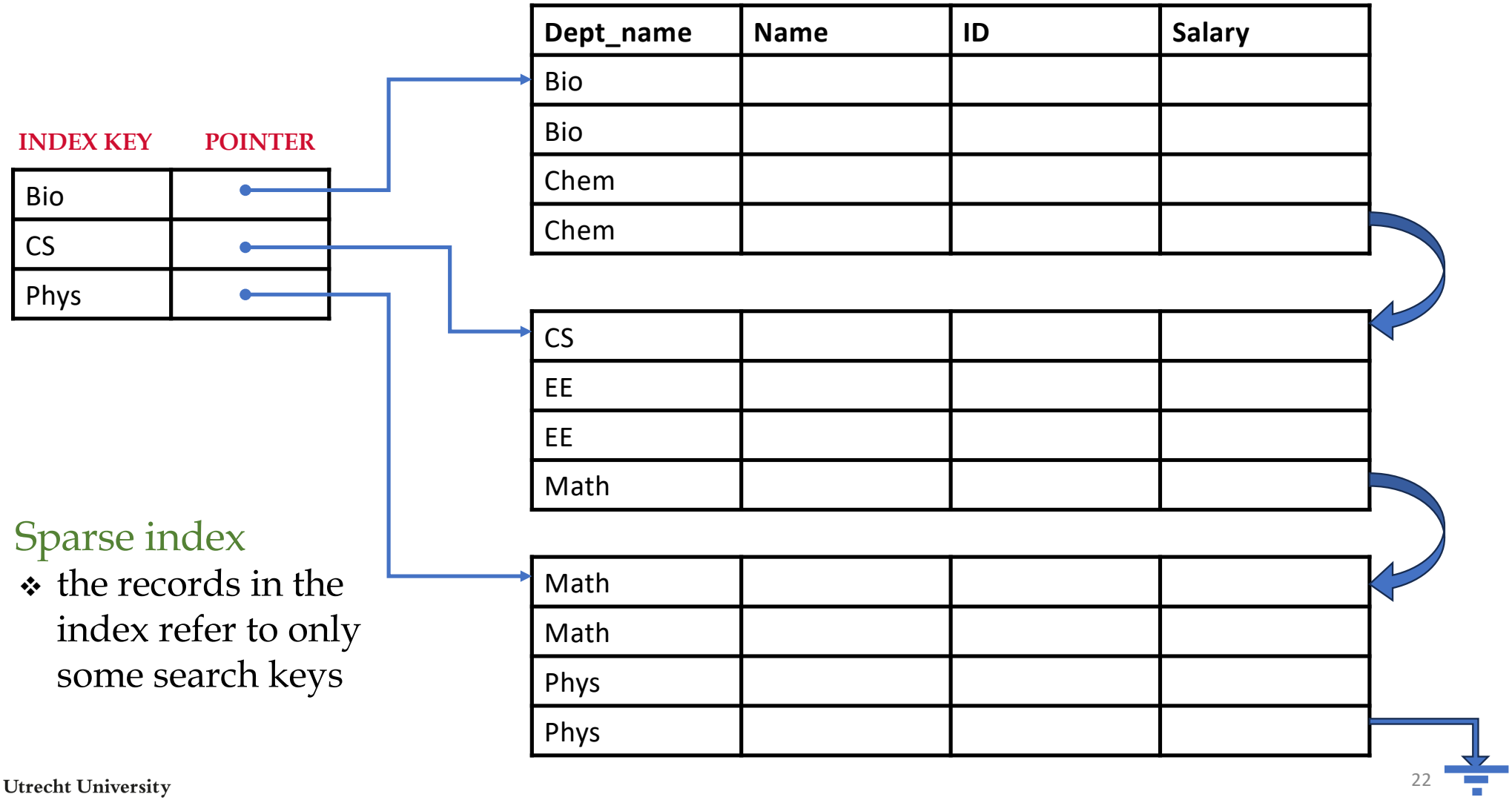


Clustered Index



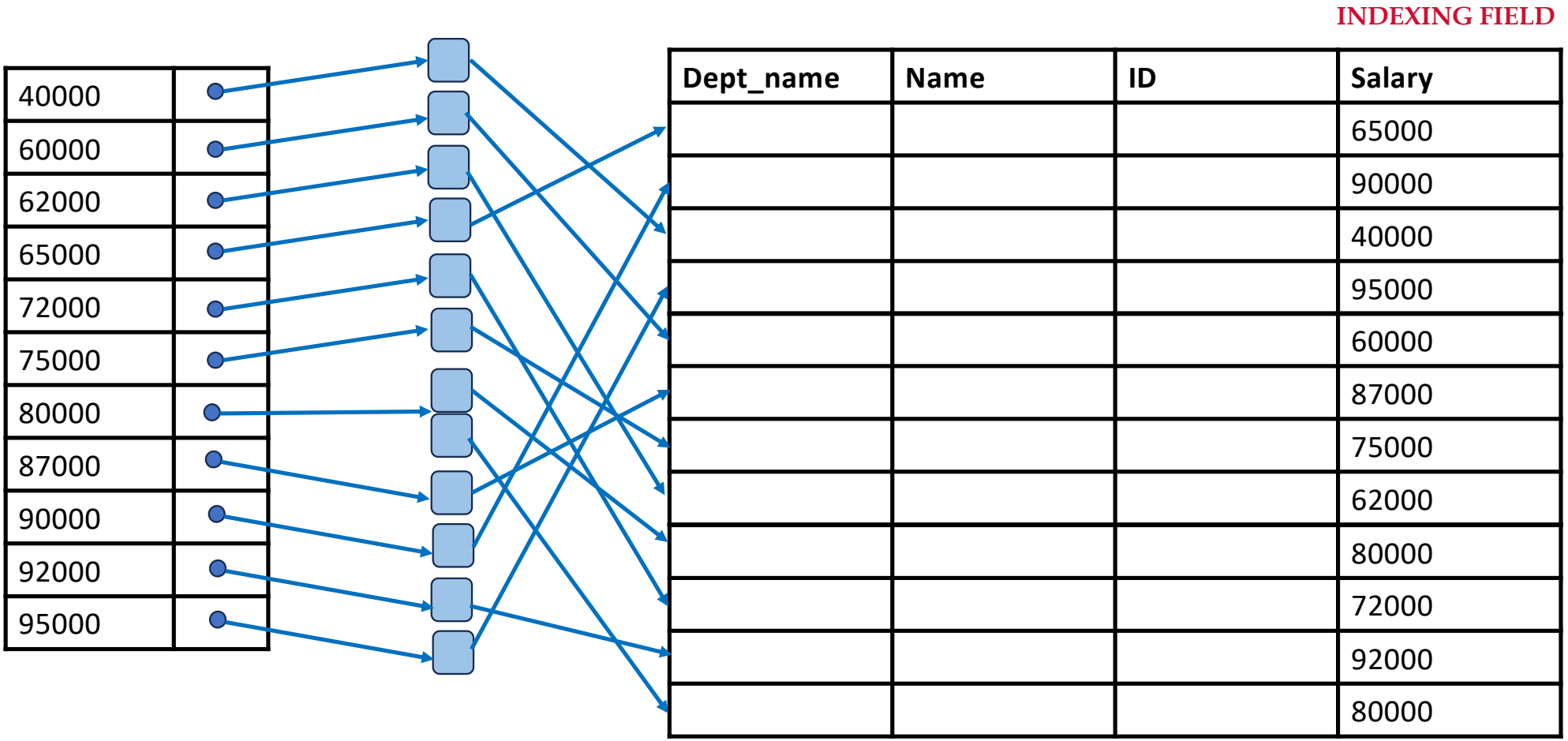
- This index is also called dense index
  - A record in the index appears for each search key

Sparse Index



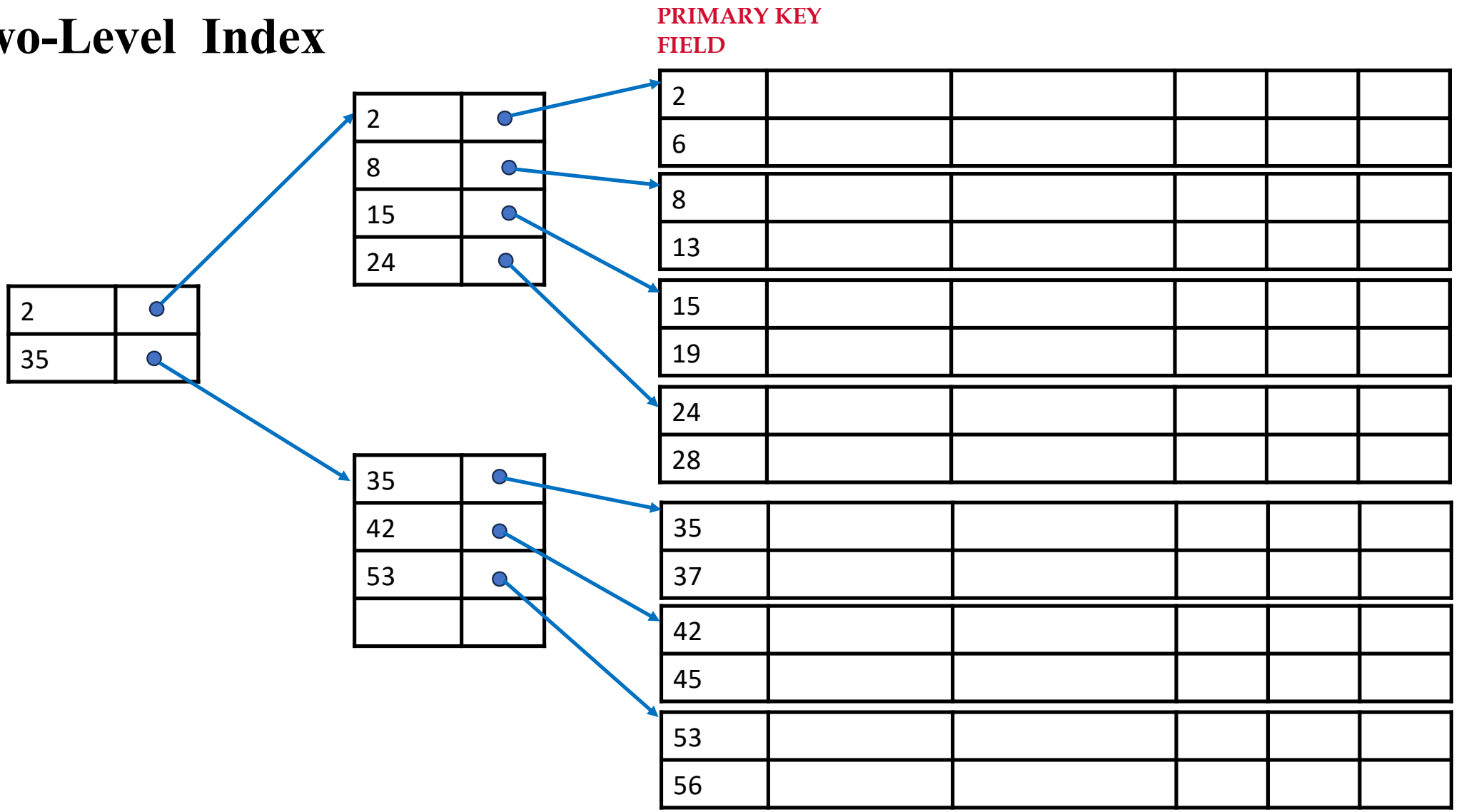
- Sparse index
  - the records in the index refer to only some search keys

Secondary Index



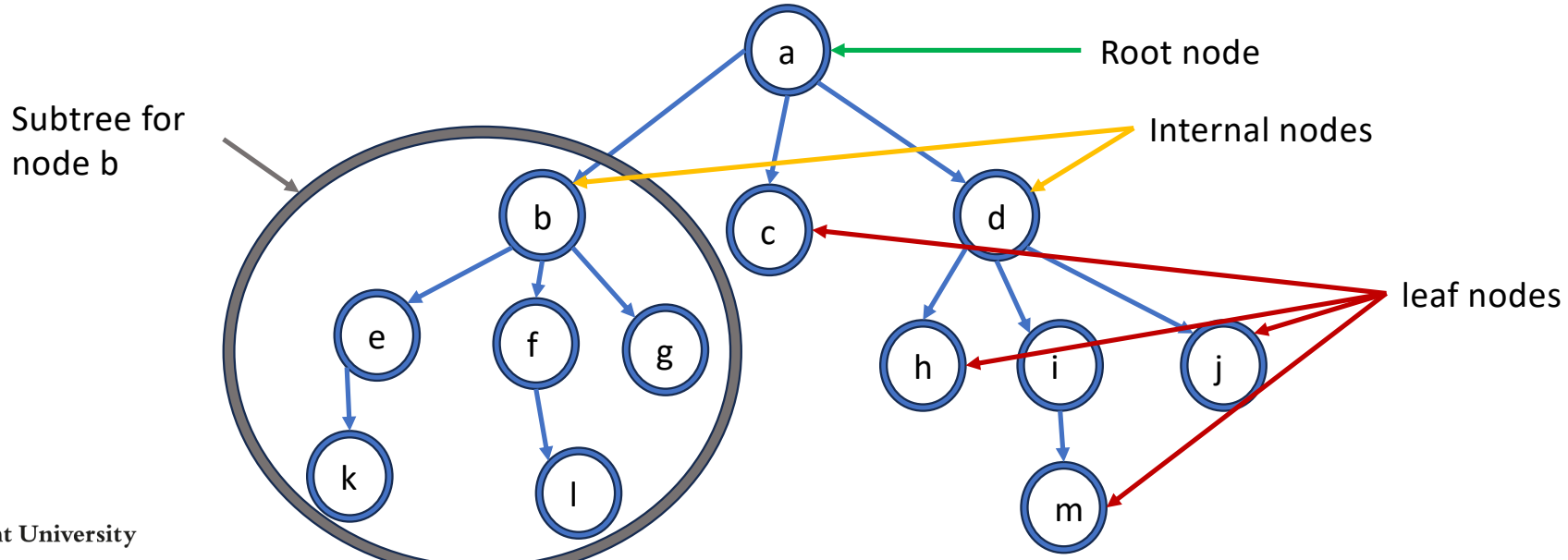
- Secondary indices have to be dense

Two-Level Index



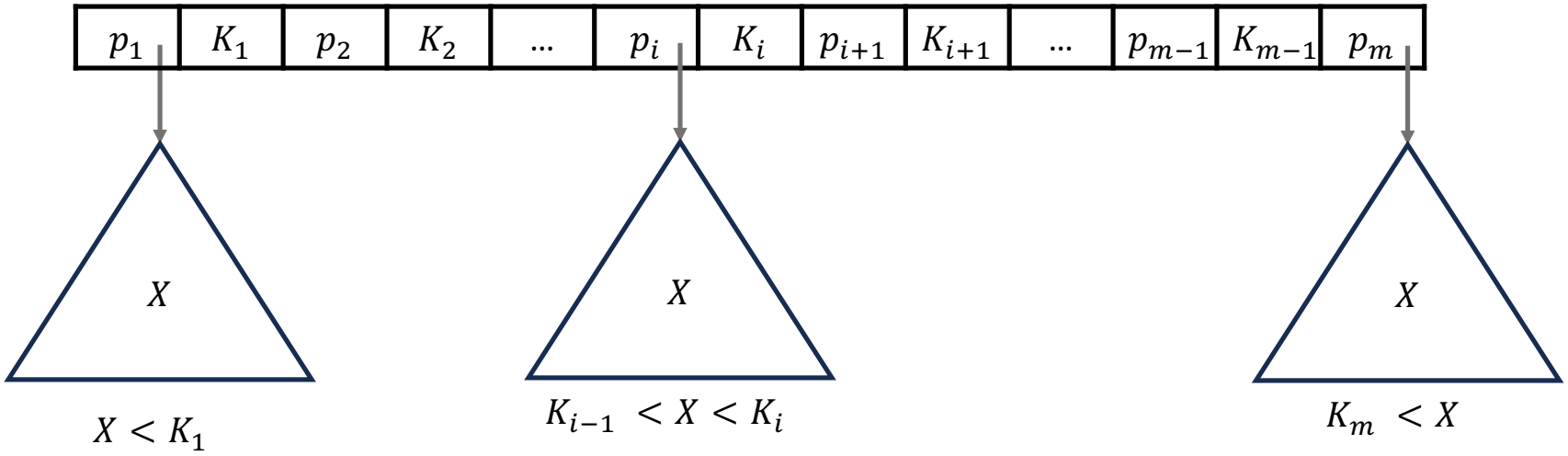
Dynamic Indexes using Trees

- Tree data structure
  - Formed of nodes
  - Nodes (except the root) have one parent and zero or more child nodes
  - Leaf nodes have no child nodes and root node has no parent node
  - Subtree consists of a node and all its children

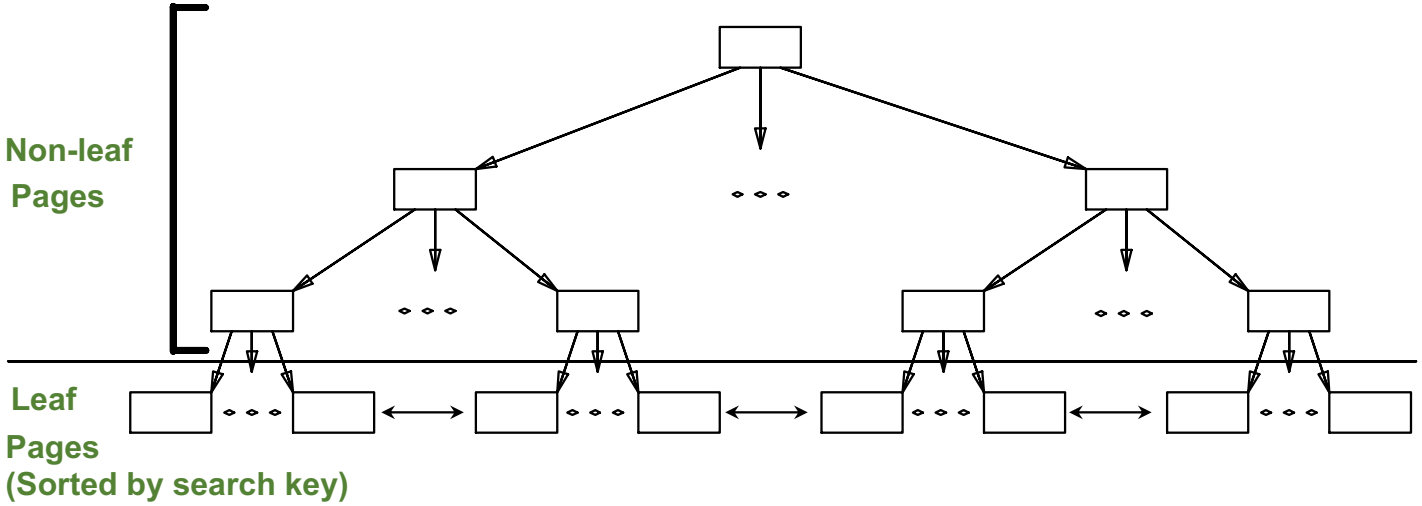


B-Trees

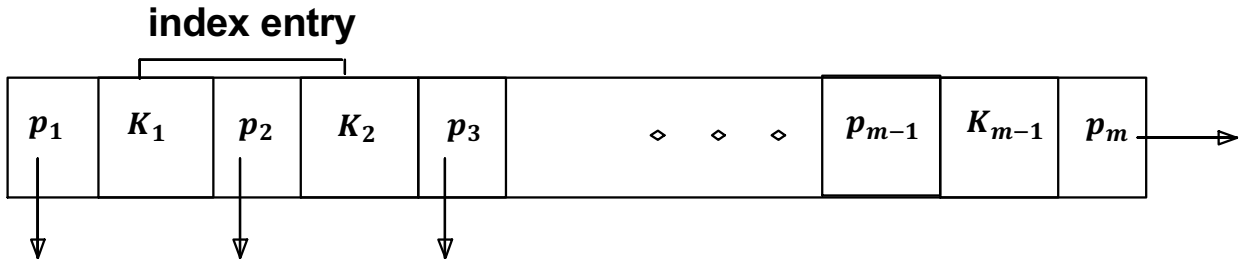
- Nodes contain values and pointers to subtrees



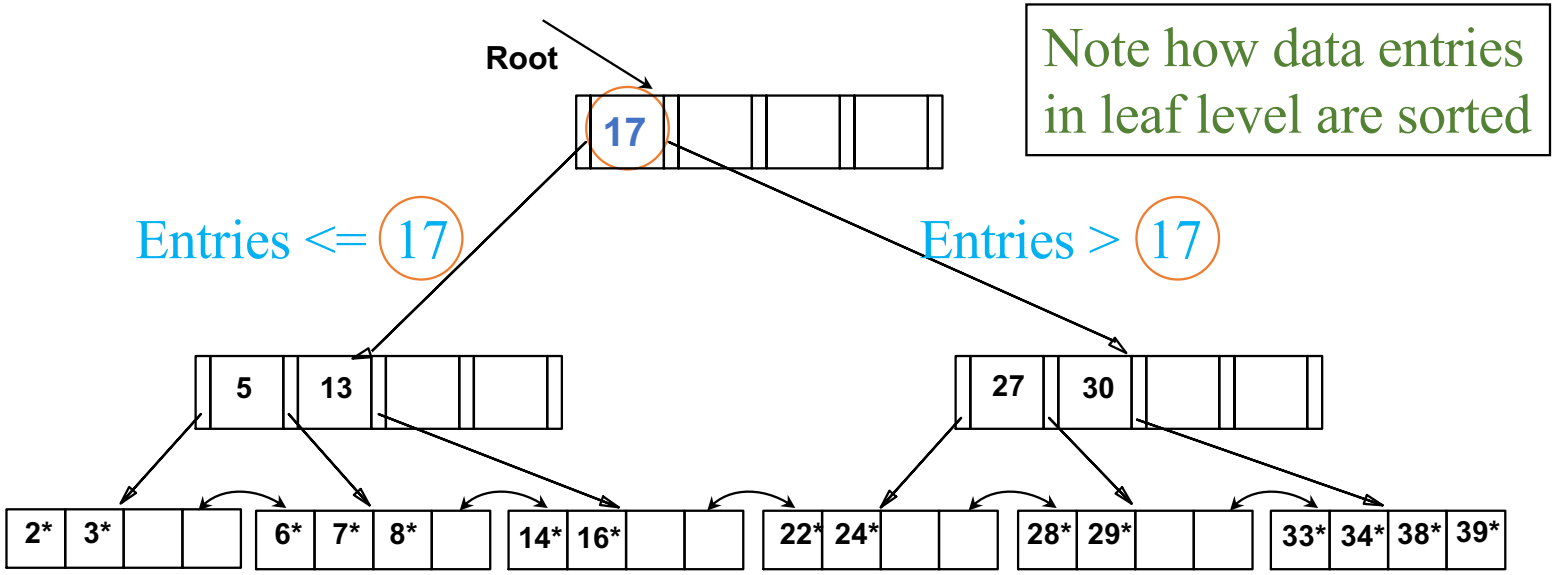
B+ Tree Indexes



- Leaf pages contain *data entries (keys)*, and are chained (prev & next)
- Non-leaf pages have *index entries*; only used to direct searches:



Example B+ Tree

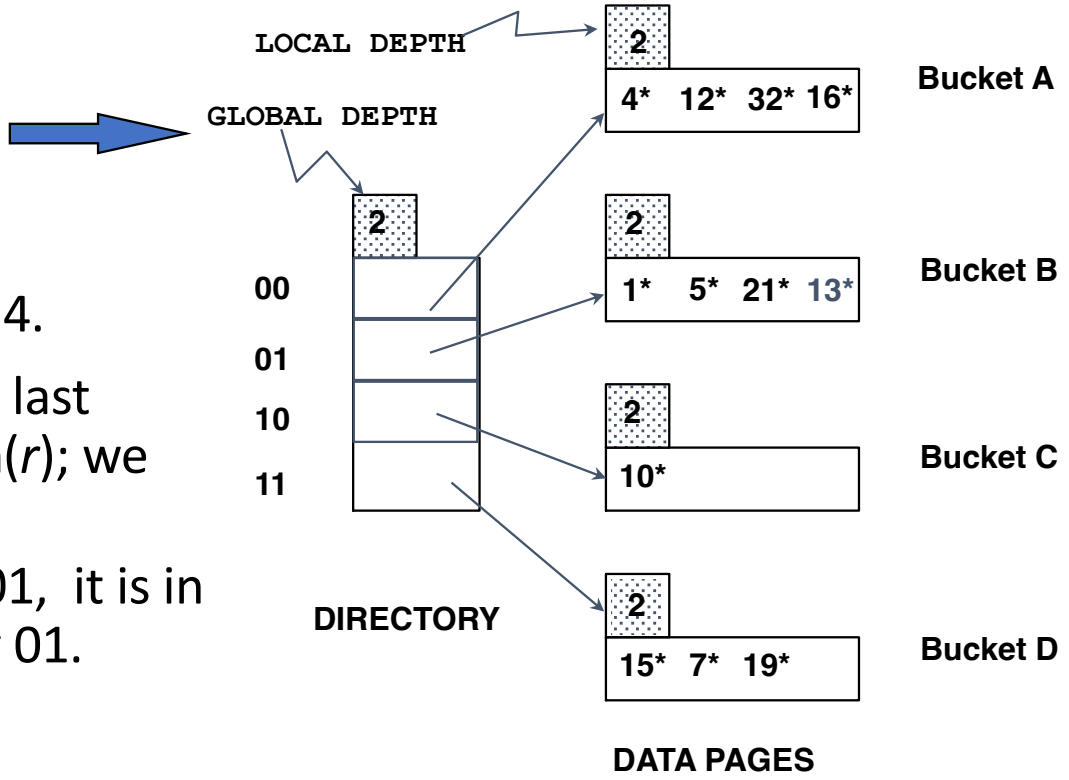


- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - Change sometimes bubbles up the tree

Hash-based Indexes

- Good for equality selections.
- Index is a collection of *buckets*.
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - Buckets contain data entries.
- *Hashing function* $\boldsymbol{h}$: $\boldsymbol{h}(r)$ = bucket in which (data entry for) record *r* belongs. $\boldsymbol{h}$ looks at the *search key* fields of *r.*
  - *No need for "index entries" in this scheme.*

Example



- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: if bucket is full, *split* it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory. (Splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Index Creation/Deletion

- Single column index

```
CREATE INDEX <index_name> ON table_name(column_name)
```

  - Example

```
CREATE INDEX idxInstSalary ON instructor(salary)
```

- Use CREATE UNIQUE INDEX to indirectly enforce the condition that the search key is a candidate key
- To drop an index

```
DROP INDEX <index_name>
```

Which Index to Use?

- B+ tree index on E.age can be used to get qualifying tuples.
  - Is better to use clustered index?
- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- Equality queries and duplicates:
  - Clustering on *E.hobby* helps!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

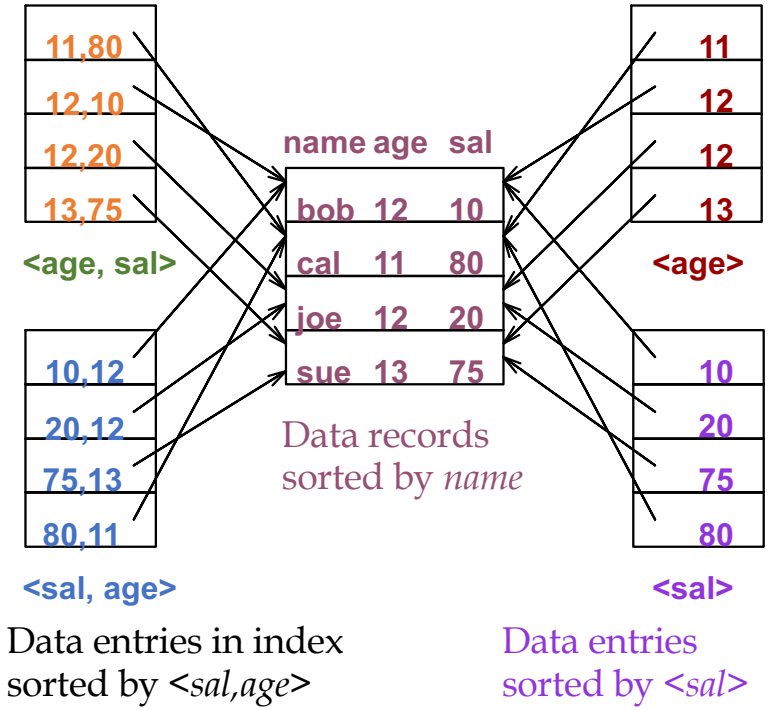
```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>10
GROUP BY E.dno
```

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Indexes with Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
- Data entries in index sorted by search key to support range queries.
  - E.g. Lexicographic order

Examples of composite key indexes using lexicographic order.



Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
- If condition is 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often
- Can be created using:

```
CREATE INDEX <index_name> ON table_name(column1, column2)
```

Composite Indexes for Single Attribute Search

- An index on <*age,sal*> can be used to retrieve records with *age*=30 (or age>10) but is NOT of any help for retrieving records with sal=20 (or sal>30)

Index-Only Plans

- A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

<*E.dno*>

```
SELECT  E.dno, COUNT(*)
FROM  Emp E
GROUP BY  E.dno
```

<*E.dno,E.sal*>

Tree index!

```
SELECT  E.dno, MIN(E.sal)
FROM  Emp E
GROUP BY  E.dno
```

<*E. age,E.sal*>
or
<*E.sal, E.age*>
Tree index!

```
SELECT AVG(E.sal)
FROM  Emp E
WHERE  E.age=25 AND
  E.sal BETWEEN 3000 AND 5000
```

Index-Only Plans (contd.)

- First query: index-only plans are possible if the key is <dno,age> or we have a tree index with key <age,dno>
  - Which one is better?
  - What if we consider the second query?

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age=30
GROUP BY E.dno
```

```
SELECT  E.dno,  COUNT (*)
FROM  Emp E
WHERE  E.age>30
GROUP BY E.dno
```

Choice of Indexes

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree?

Choice of Indexes (contd.)

- One approach:
  - Consider the most important queries in turn.
  - Consider the best plan using the current indexes
  - Look for a better plan is possible with an additional index.
    - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!
- Before creating an index, must also consider the impact on updates in the workload!
  - Indexes can make queries go faster, updates slower.
  - Require disk space, too.

Operations to Consider & Evaluation Metrics

- **Operations**
  - Scan: fetch all records from disk
  - Equality search
  - Range selection
  - Insert a record
  - Delete a record

- **Metrics:**
  - Access types supported efficiently
    - Equality search
    - Range selection
  - Insertion time
  - Deletion time
  - Space overhead

Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
  - Order of attributes is important for range queries.
  - Such indexes can sometimes enable index-only strategies for important queries.
    - For index-only strategies, clustering is not important!
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

- Summarize what you learned today in 2-minutes

The information in this presentation has been compiled with the utmost care,
but no rights can be derived from its contents.