NYU | TANDON SCHOOL OF ENGINEERING

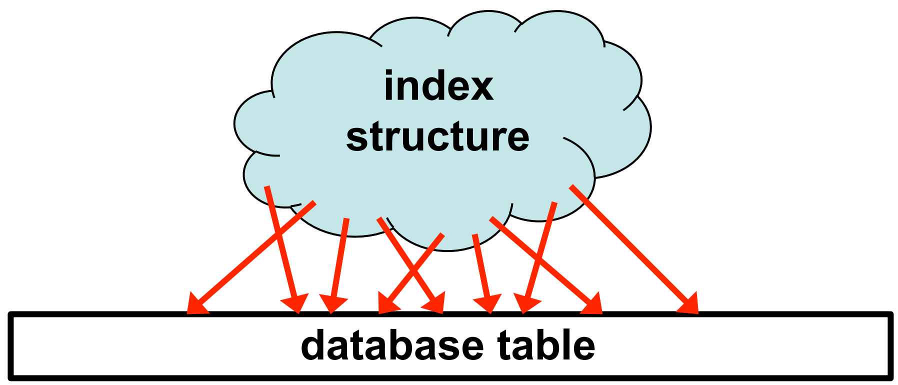
Introduction to Index Structures

CS6086: Principles of Database Systems
NYU Tandon

NYU TANDON SCHOOL OF ENGINEERING

Indexing:

- An index is a data structure that supports efficient retrieval of "certain types of tuples" from a table
- **Setup:**



- **Without index:**
  - Need to scan entire table
  - Or binary search if sorted

Problem:

“Find records of a certain type quickly”

- **Equality Queries**

  $\sigma_{\text{ssn} = 619441789}$ **(Employees)**

- **Range Queries**

  $\sigma_{10 <= \text{age} <= 20}$ **(Employees)**

These are the most interesting types of conditions

Plus AND and OR of several such conditions

Goal: faster processing of SQL queries (selects and joins)

NYU TANDON SCHOOL OF ENGINEERING

Various types of indexes

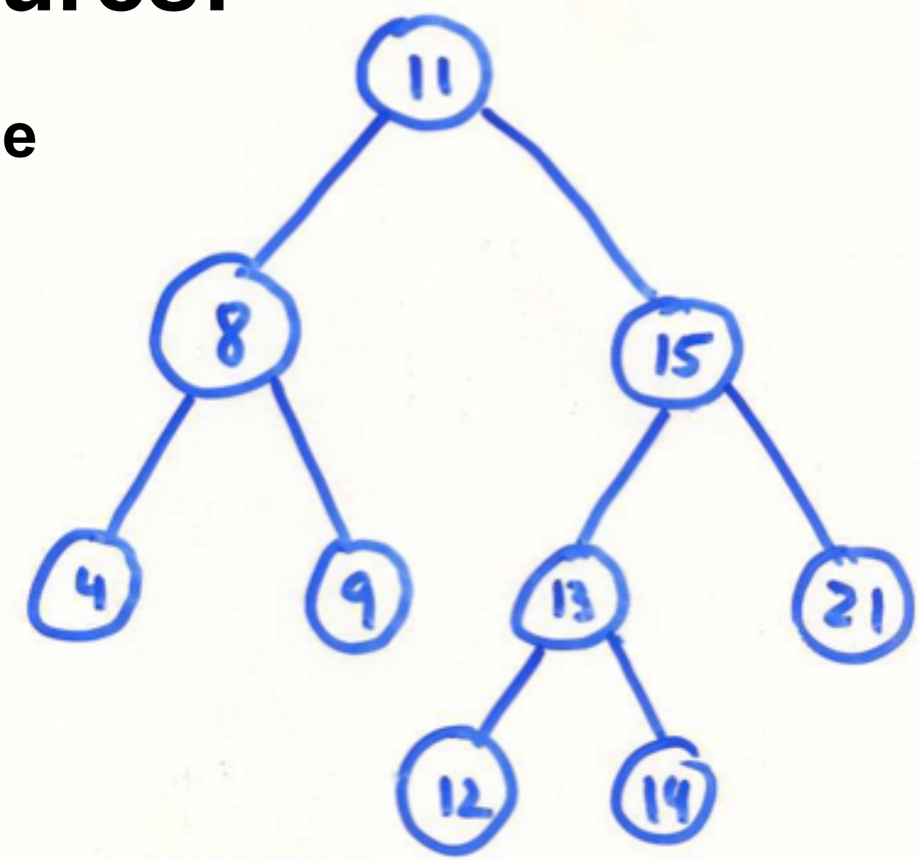
1. Clustered (Primary) or Nonclustered (Secondary)
2. Sparse or Dense
3. Single Key or Composite or Multi-Dimensional

Different implementation (algorithms)

- Tree Structures (ordered)
- Hashing (non-ordered)
- Others

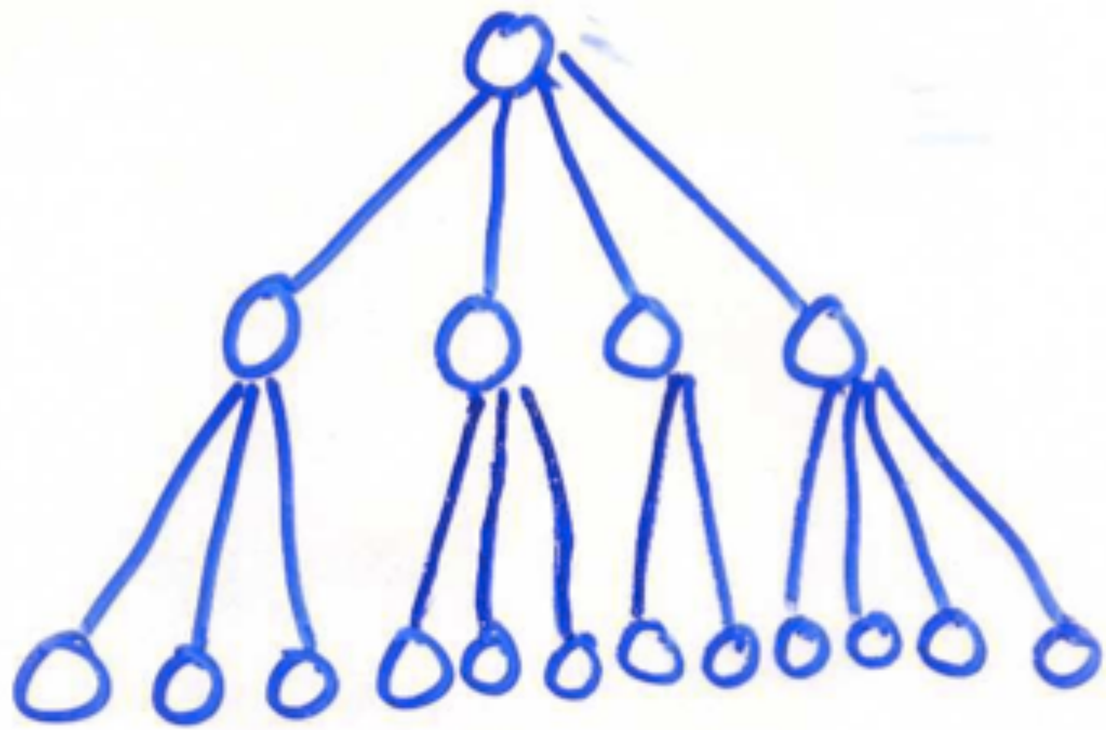
Indexing Structures:

E.g., Binary Search Tree



NYU TANDON SCHOOL OF ENGINEERING

B$^+$-Tree:



- Degree >> 2 (why?)
- Data in leaves only (why?)
- Same depth everywhere (how?)

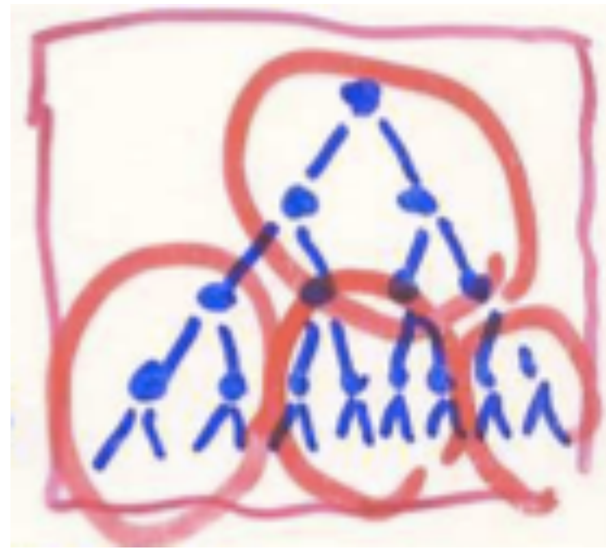
Compare: Trees with N Elems in Leaves

- Binary Tree with N keys:
  - $\log_2$ N levels
  - Each internal node:
    - 20 bytes? (2 pointers + 1 key)
  - N = 10^6 => 20 levels

- Degree-100 tree:
  - $\log_{100}$ N levels
  - Each internal node:
    - 1200 (1196) bytes (100 pointers + 99 keys)
  - N = 10^6 => 3 levels

- => This is much better in the disk case!

Compare: Trees with N Elems in Leaves

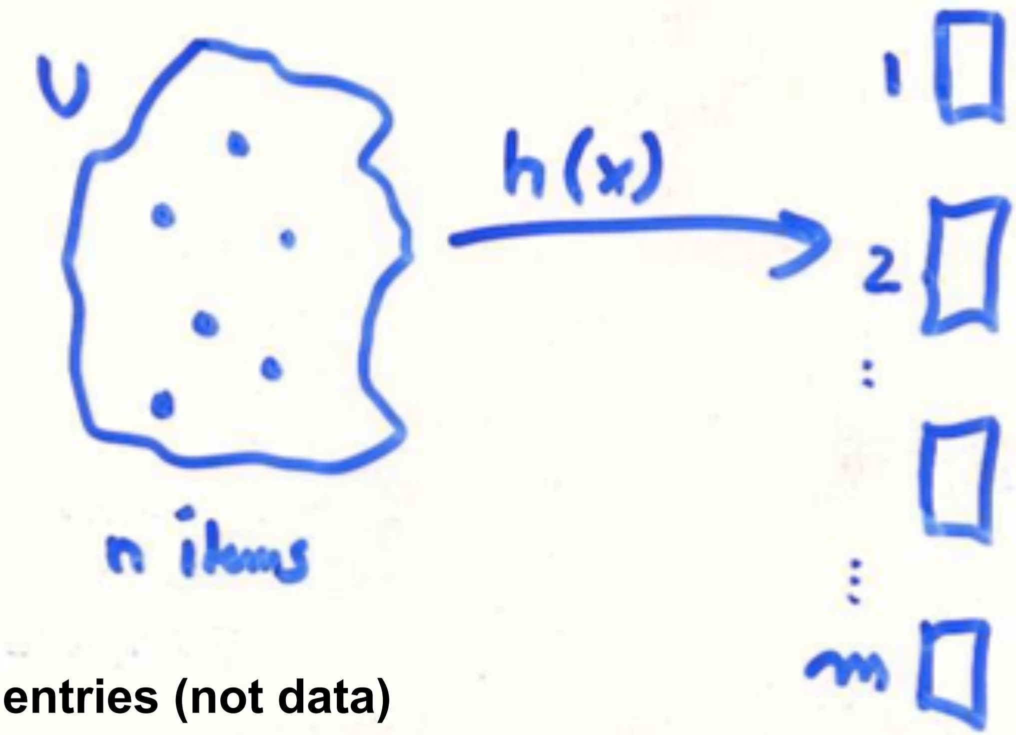


- Binary Tree with N keys:
  - $\log_2$ N levels
  - Each internal node:
    - 20 bytes?   (2 pointers + 1 key)
  - N = 10^6 => 20 levels
- Degree-100 tree:
  - $\log_{100}$ N levels
  - Each internal node:
    - 1200 (1196) bytes (100 pointers + 99 keys)
  - N = 10^6 => 3 levels

- => This is much better in the disk case!
- .. like collapsing several binary levels into one

Hash Tables

buckets



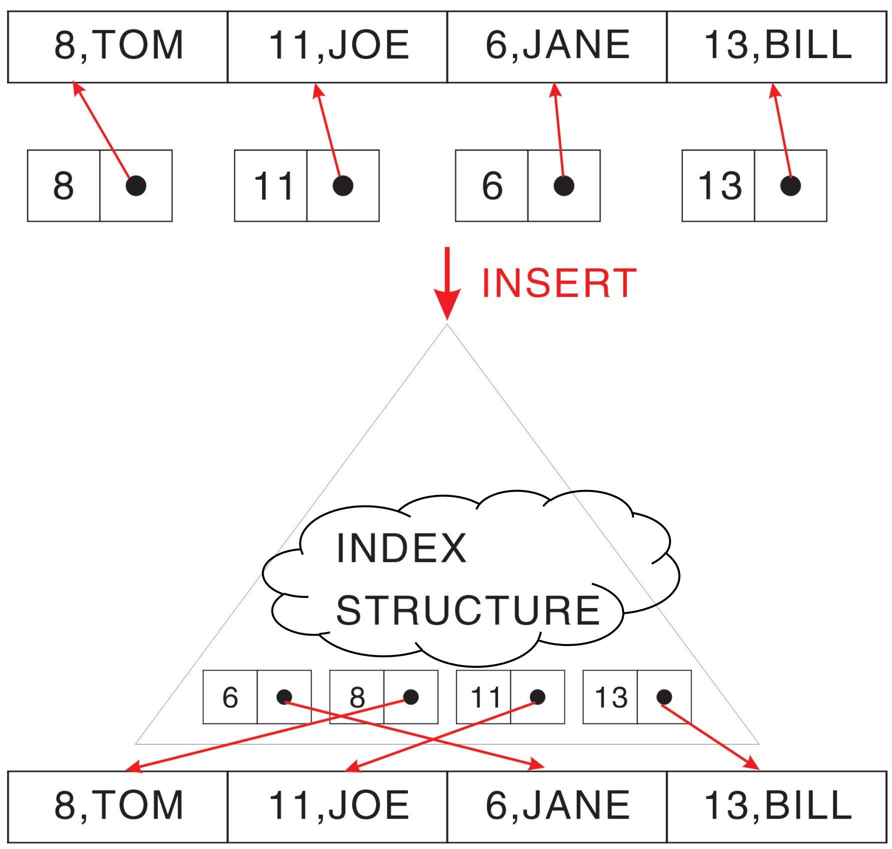
- x = key value
- h: U ➔ [1,2,3,…,m]
- buckets contain index entries (not data)
- No range queries supported
- Exception: small enumerable ranges

Index entries

KEY	POINTER

- An index entry is a pair where:
  - Key is a key value
  - Pointer is a record ID (usually not a physical pointer)
  - Every object in the DB has an RID.
- Index Construction:
  - Make an index entry for each tuple in the table
  - Insert index entries into index structure (B^+-tree, hash, etc)

Example: Schema is (age, name) with index on age



Indexing: Basic Setup

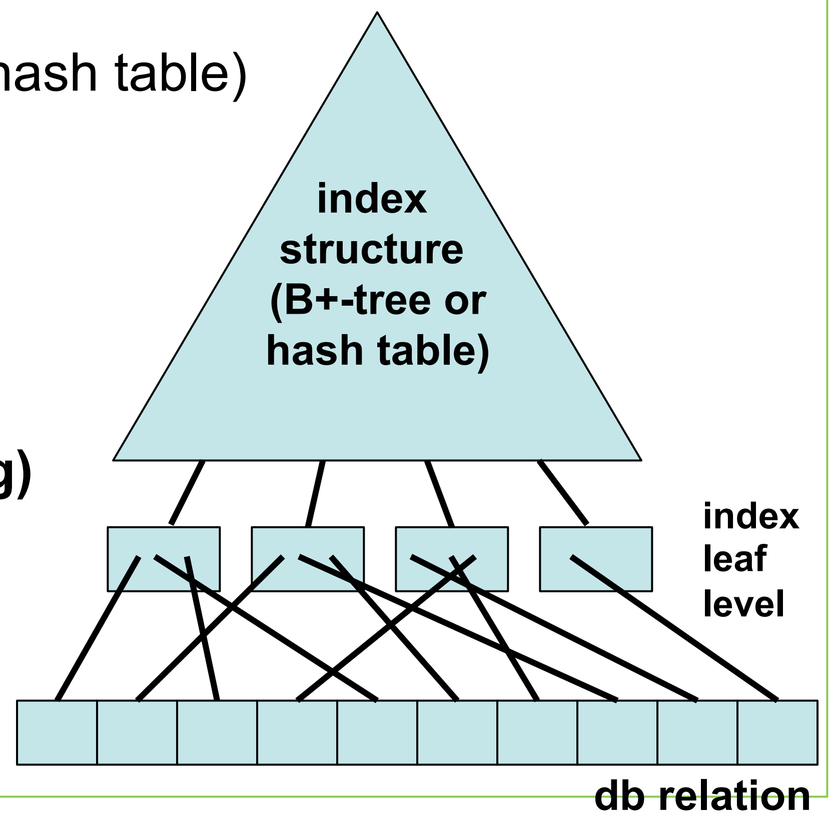
Hides:

- Type of structure used
  (tree or hash)
- File/record organization used
  (data is not moved during indexing)

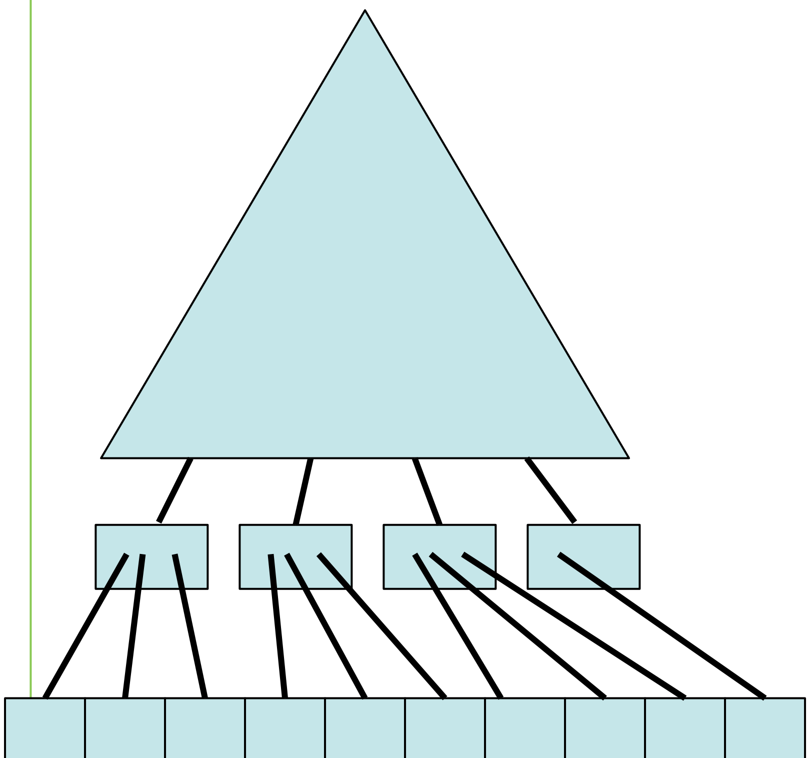
Indexing: Basic Setup

Hides: index structure (B+-tree or hash table)

- **Type of structure used (tree or hash)**
- **File/record organization (data not moved by indexing)**

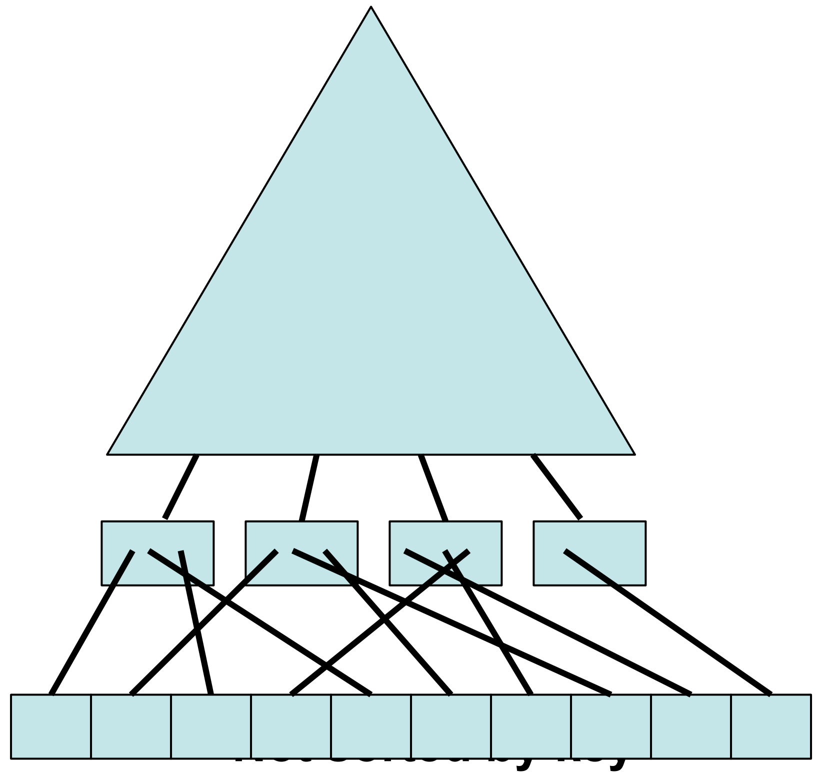


Primary Index



relation sorted by key

Secondary Index



not sorted by key

Primary Indexes

- Can have only one primary index per table
- Not (always) same as primary key! (e.g., SSN)
- Advantages
  - Efficient range search
  - Smaller index size possible (sparse index)
- Sparse Index:
  - Idea: don't create index entry for each row in table
  - E.g., only pointers to row at start of a data blocks

Why Primary Indexes?

- Table is already sorted by attribute
- Why not just do binary search?

Why Primary Indexes?

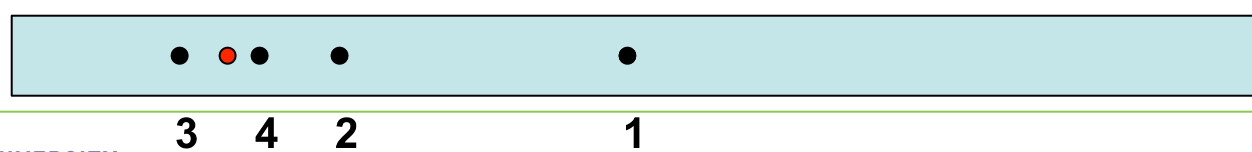
- Table is already sorted by attribute
- Why not just do binary search?

- Answer: binary search is not I/O-efficient
- Table with 10^6 records: probe $lg_2(10^6)$ ~ 20 records
- B+-tree: fetch only 4-5 index nodes
- Also, top levels of tree are cached
- Binary search:

Why Primary Indexes?

- Table is already sorted by attribute
- Why not just do binary search?

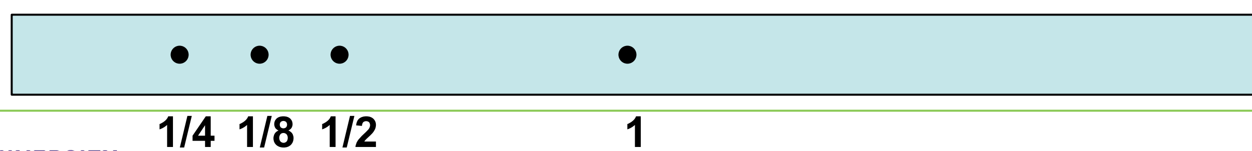
- Answer: binary search is not I/O-efficient
- Table with 10^6 records: probe $\lg_2(10^6) \sim 20$ records
- B+-tree: fetch only 4-5 index nodes
- Also, top levels of tree are cached
- Binary search:



Why Primary Indexes?

- Table is already sorted by attribute
- Why not just do binary search?

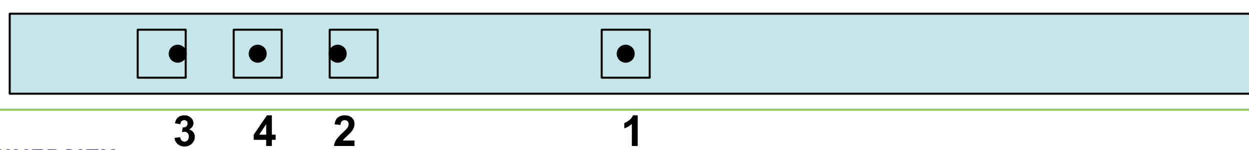
- Answer: binary search is not I/O-efficient
- Table with 10^6 records: probe $\lg_2(10^6) \sim 20$ records
- B+-tree: fetch only 4-5 index nodes
- Also, top levels of tree are cached
- Binary search: cache commonly probed items?



Why Primary Indexes?

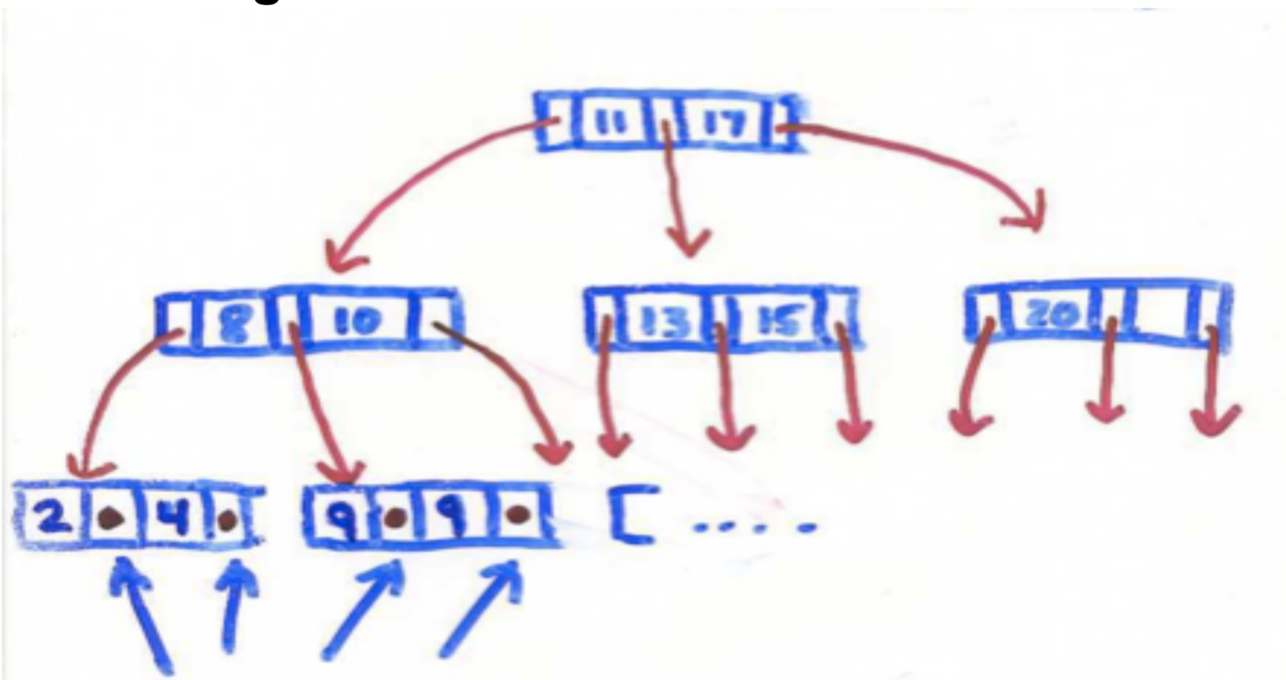
- Table is already sorted by attribute
- Why not just do binary search?

- Answer: binary search is not I/O-efficient
- Table with 10^6 records: probe $\lg_2(10^6) \sim 20$ records
- B+-tree: fetch only 4-5 index nodes
- Also, top levels of tree are cached
- Binary search: caching is inefficient



Example: B$^+$ - Tree

- **"Switch to algorithms mode"**

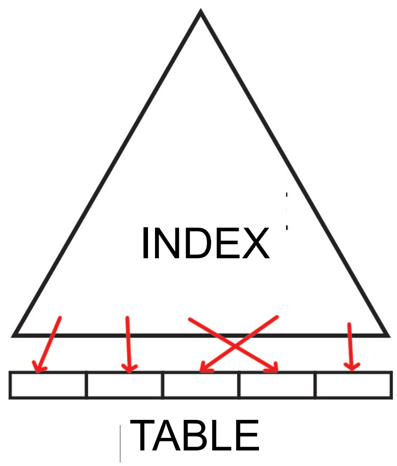


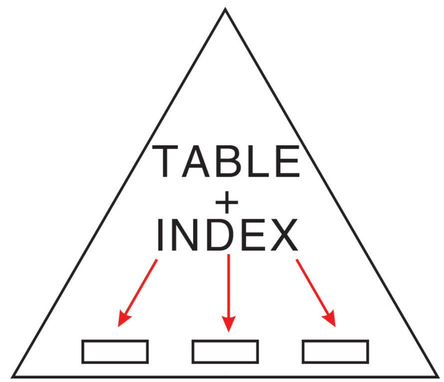
- **"PAYLOAD" (Pointer to record) in index entries**

Types of Data Structures Used

- **Tree – Based**
- **Hash – Based**
- **Other (often multi-dim)**

<u>Index vs. File Organization:</u>



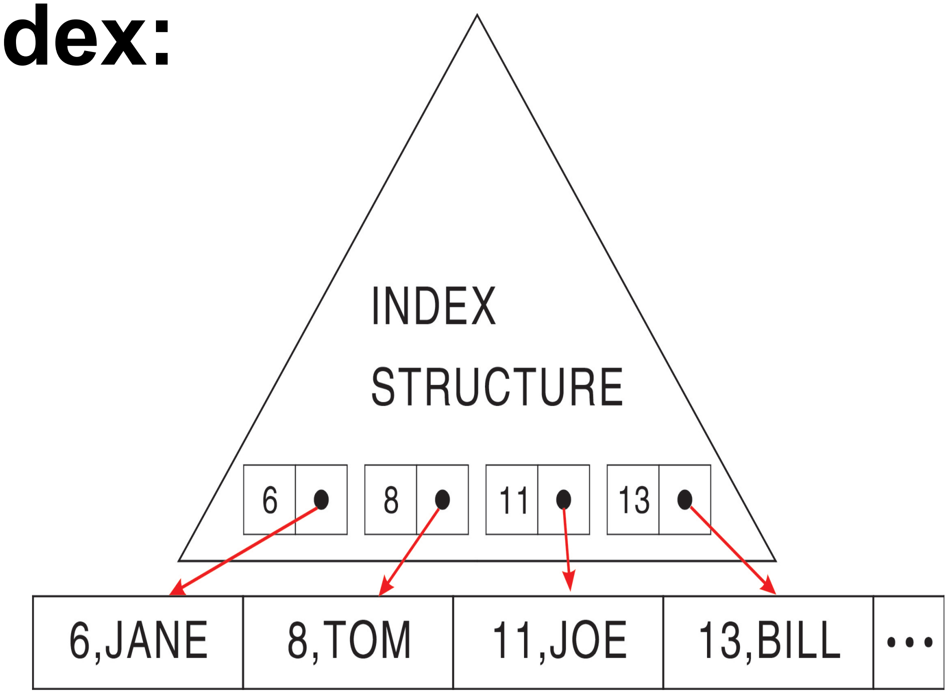


➔ **We assume index organization**

Types of indexes

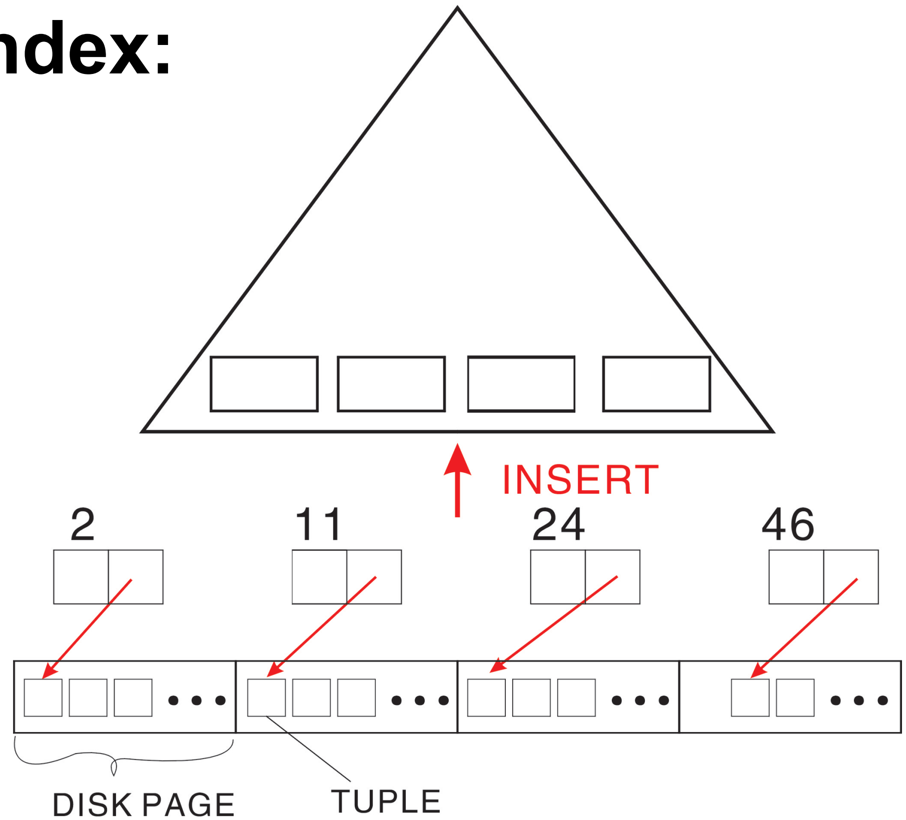
- Primary / Secondary & Clustered / Unclustered
  - primary (clustered) means table sorted on index key
  - only one primary index per table
- Dense / Sparse
  - sparse only possible if clustered index
- Single Attribute / Composite
  - single attribute
  - attribute 1, attribute 2, …
  - True multi-attribute (multi–dimensional) index

Clustered Index:



- No crossing pointers
- Do we really need index ?
- Do we need all index entries?

Sparse Clustered Index:



- One entry per page
- Smaller Index
- Fewer levels
- But need to search inside pages

B^+-Trees in a Nutshell:

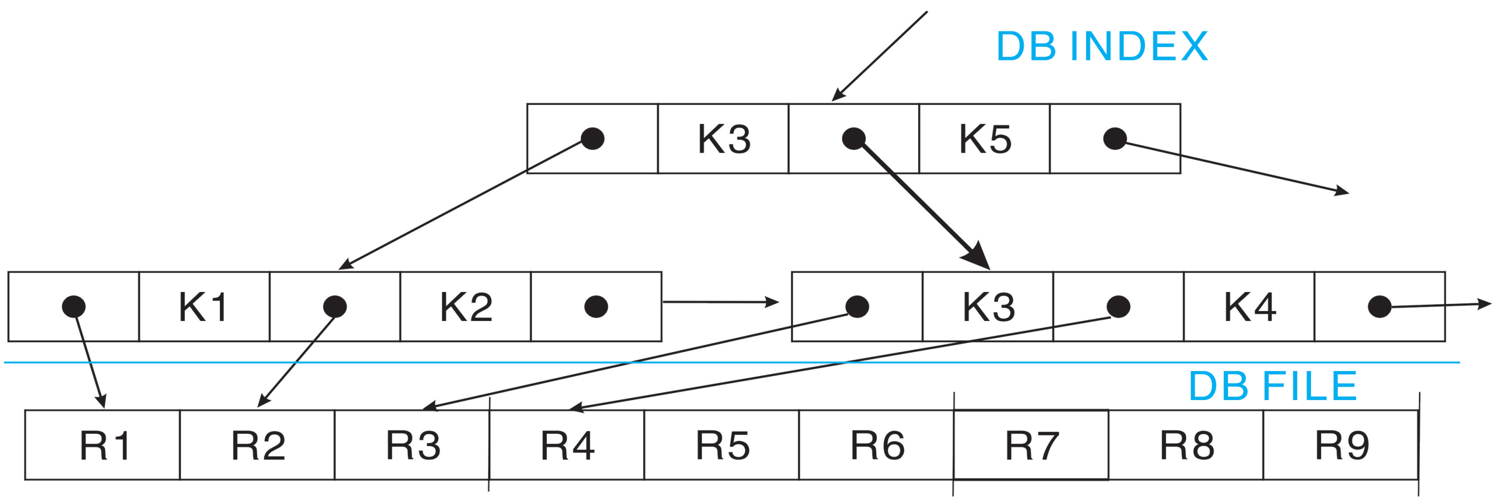
- Each node ≈ disk block
- Typically 4 KB
- Internal nodes
  - $\lceil n/2 \rceil <= c <= n$ # of child pointers
  - $\lceil n/2 \rceil -1 <= k <= n-1$ # of key values
- Root note: like internal, but at least 2 kids & 1 key
- Leaf nodes
  - $\lceil (n-1)/2 \rceil <= k <= n-1$

    # of key values and also # of record pointers

Example:

- 4 KB node size
- 8 – byte integer keys
- 8 – byte pointers

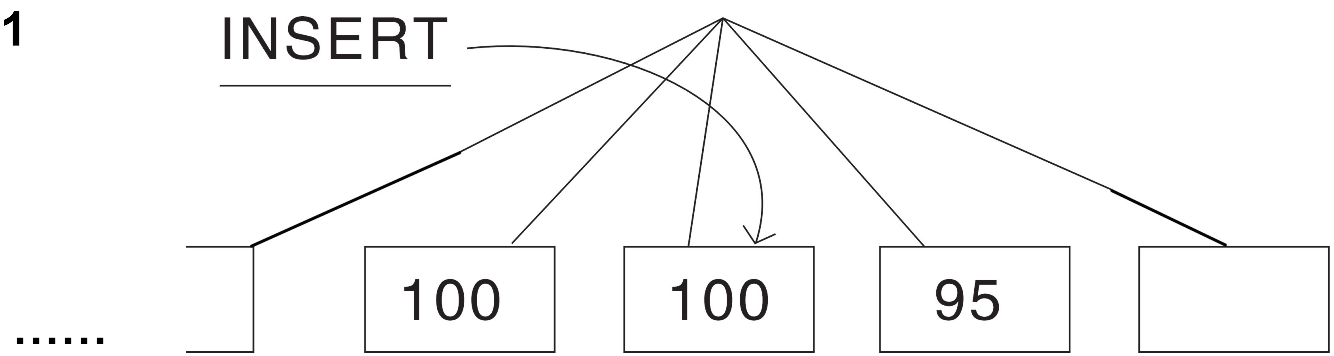
- Choose n = 250 so that 250*8 + 249*8 = 3992 bytes ≈ 4 KB (including some overhead)



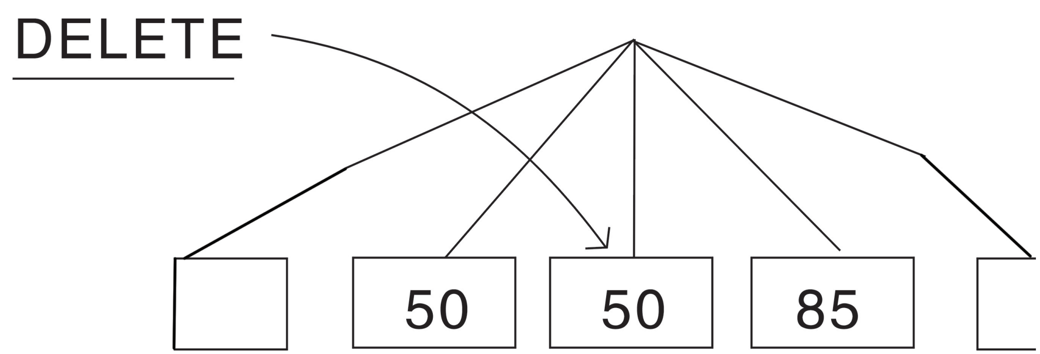
Insertion & Deletion:

- **Insert:**
  - If space left in leaf, just insert and done
  - Else, try to rebalance with sibling neighbor (optional)
  - If impossible, split node, insert new node as child in parent
  - If parent has no space, rebalance/split at parent level, etc.
- **Delete**
  - If leaf still >= $\lceil n/2 \rceil$ children, just delete and done
  - Else, try to rebalance with sibling neighbor
  - If impossible, merge with sibling, delete node from parent
- **Rebalance:** balance data with sibling so each has same num

- Note: numbers are # of index entries in leaves
  - n = 101



- Could balance with right neighbor (98 each afterwards) or not
- Delete: try to balance

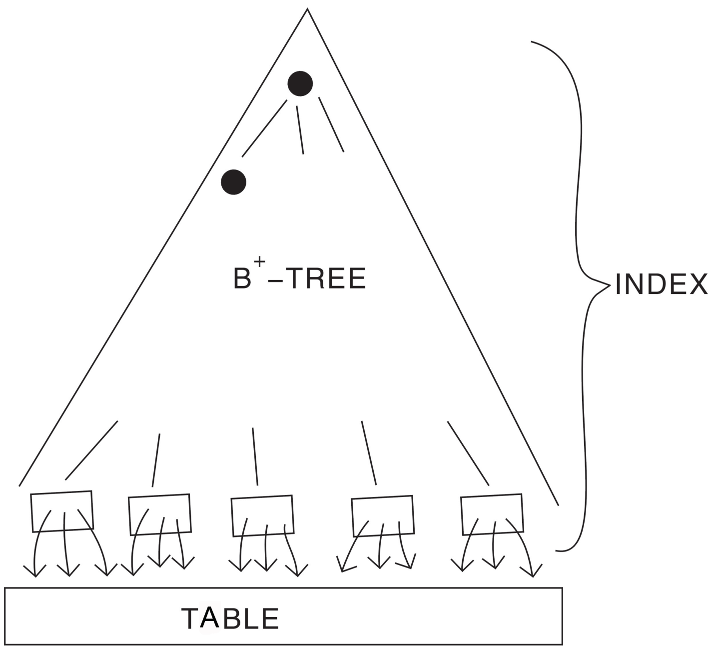


- Never go beyond immediate neighbor!

File vs. Index Organization

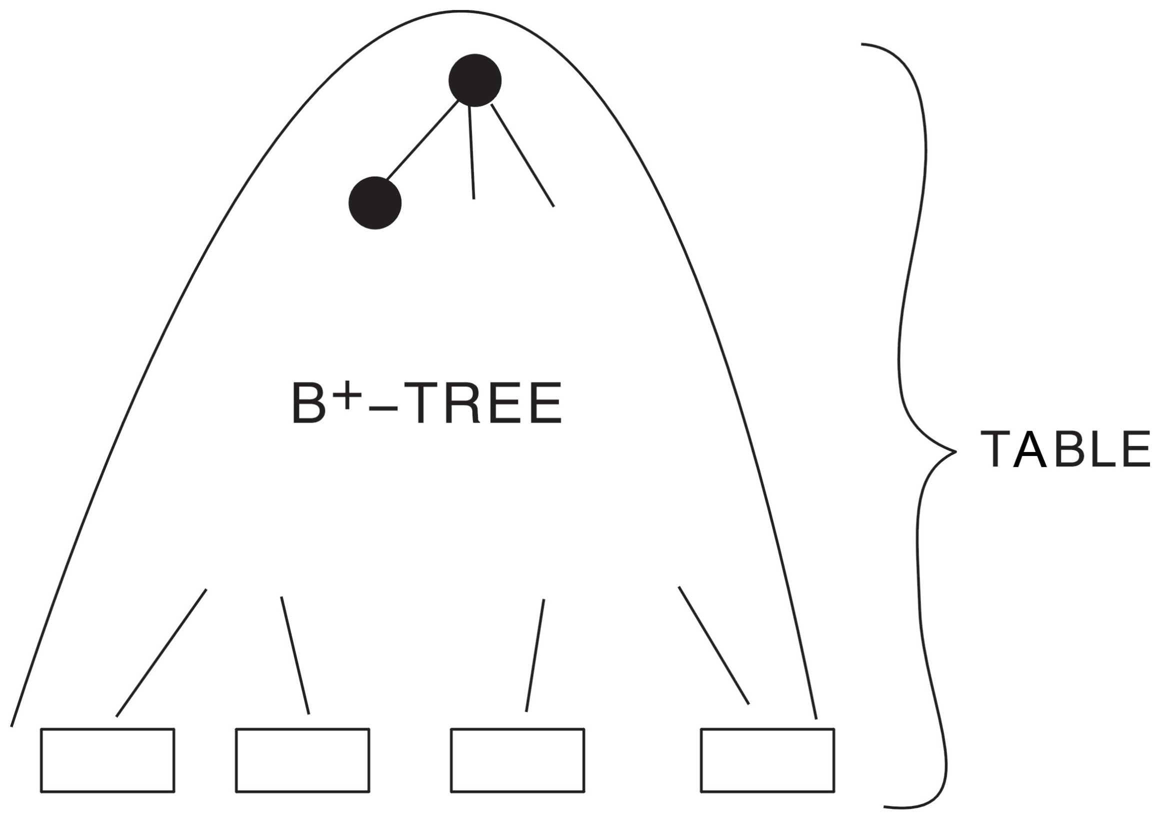
- Example of B^+ - Tree (Clustered) Index Organization

leaves



- Index and relation are separate!

B^+-Tree File Organization



B^+-Tree File Organization

- Tuples are contained in leaves (instead of pointers to tuples)
- Maybe one less indirection (1 fewer disk access to find tuple)
- But more leaf nodes (height may change)
- Can still add secondary indexes
- Similar for hash tables
- "The file is the Index"

We mosly assume the other case: tuples not at leaf level

Dense Non-Clustered B+-Tree Index:

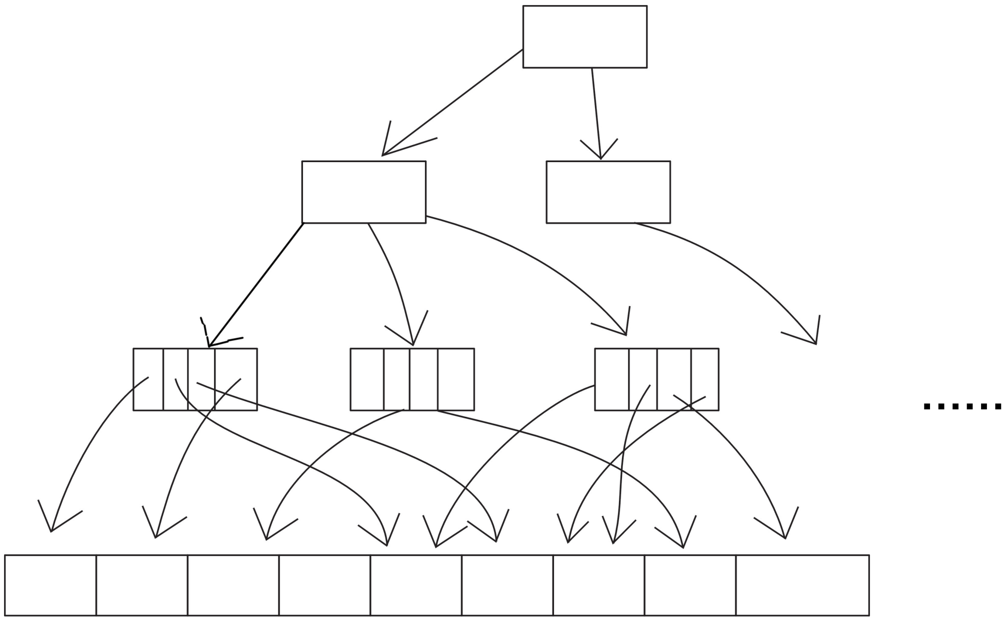
n = 250

1 root

16 ~ 64 nodes

4000 ~ 8000 nodes

1000000 records



- Leaf level: 1000000 (key, pointer) pairs
- Next level: 4000 ~ 8000 pairs

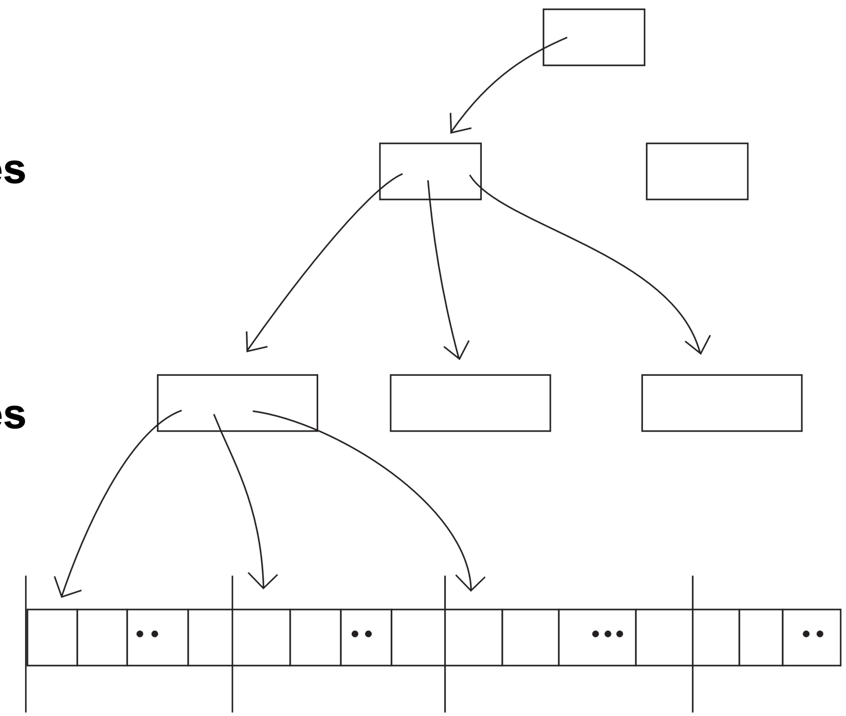
Sparse, Clustered B$^+$-Tree Index:

- **n = 250, 20 records/block**

1 ~ 4 nodes

200 ~ 400 nodes

1000000 records in 50000 blocks



Analysis of B$^+$ - Tree Costs

- Assume occupancy factor
- E.G. 80% → degree 200 for n = 250
- Or upper / lower bounds as shown
- Also occupancy factor for DB file if sparse index
- Then count cost in terms of disk accesses and/or transfer costs

ISAM vs. B+-Trees

- Index sequential access method
- Not as good for updates
- Slightly simpler
- But B+–trees overall better
- Conceptually: ISAM is “Indexing an index” but recursively
- We focus on B^+-trees

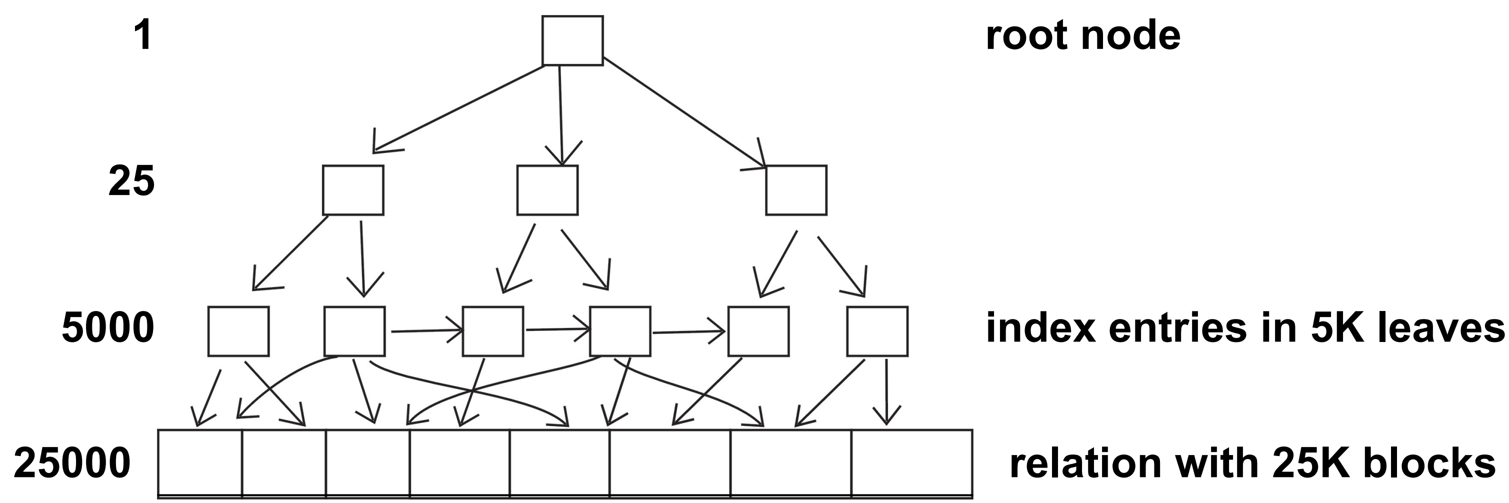
Example

- Disk with $T_s + T_r$ = 10ms and tr = 10 MB/S
- Database with 1000000 records of size 100 bytes
- Assume block size 4 KB

  ➔ 25000 blocks with 40 records each (100 % occupancy) and T_B = 10ms + 0.4ms = 10.4ms (time per table)

Case 1: Unclustered B$^+$-tree, 4KB blocks

- Key of size = 8 bytes and Pointer = 8 bytes
- ➔ n = 250 (approximation)
- Assume average degree 200 (80%)

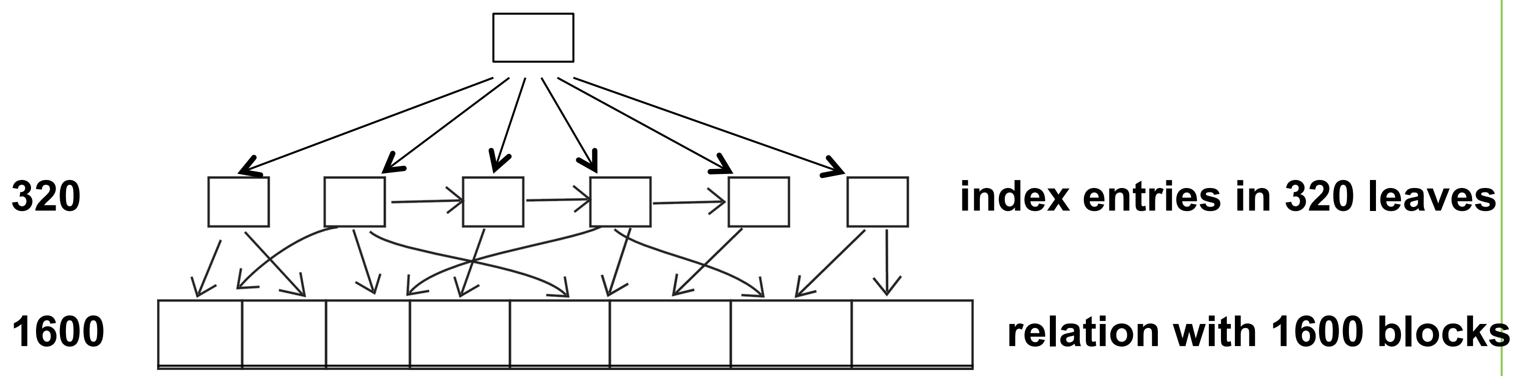


- Find one item: 4 * 10.4ms = 41.6ms
- Range Query for 1000 records: 1000 * 10.4ms + 5 * 10.4ms + 2 * 10.4ms ≈ 10.5s

Case 2: Unclustered with block size 64 KB

→ ~ 1600 blocks with 640 records each

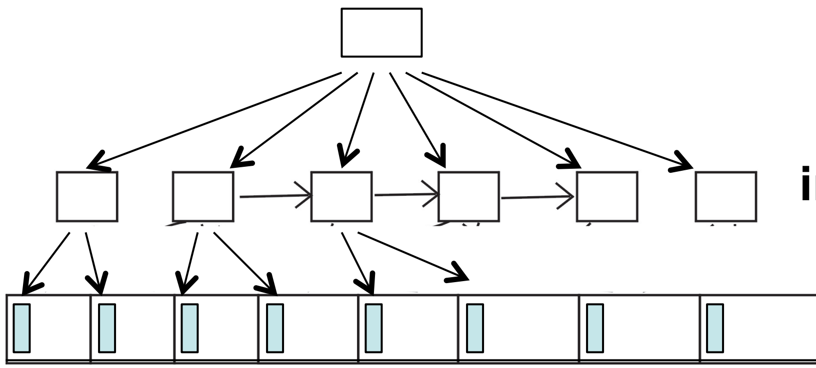
- n = 4000 ! and T_B = 16.4ms
- Assume average degree 3200



- Retrieve one item: 3 * 16.4ms = 49.2ms
- Retrieve 1000 items: 1000 * 16.4ms + 2 * 16.4ms = 16.5s
- Overall, large blocks not a good idea

Case 3: Sparse clustered B+-Tree, 4KB blocks

- 125 blocks
- 25000 blocks



index entries in leaves

relation with blocks

- Retrieve one item ➔ 3 * 10.4ms = 31.2ms
- 1000 consecutive elements (range search) ➔ (25+2) * 10.4 = 280ms

Case 4: Scanning the entire table

- 4KB page, 25000 pages of data
- 25000 * 10.4ms ≈ 280s (4KB block model)
- ---
- 64KB page, 1600 page of data
- 1600 * 16.4ms ≈ 26s (64KB block model)
- ---
- Scan model
- 10ms + 10 sec = 10.01 sec (To scan 100MB file)

➡ Disk model predicts that sometimes, scanning is better than using an unclustered index. Also, 64KB blocks worse

NYU TANDON SCHOOL OF ENGINEERING

Composite Keys

- $\sigma_{\text{salary = 10000 and age = 30}}$ (employees)
- $\sigma_{\text{salary > 80000 and 20 < age < 30}}$ (employees)

- How to access these tuples:
  - Use index on salary
  - Use index on age
  - Use composite index
    - - (age, salary) or (salary, age)
    - - hash age × salary
    - - 2 – D data structures (grid file)

Multi–Attribute B+-Trees

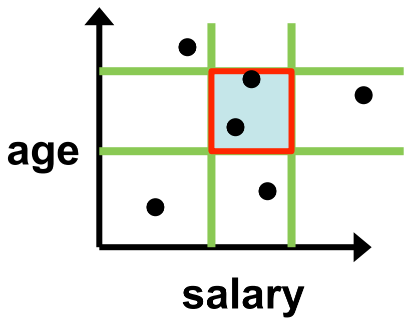
- (age, salary) ➔ 6 bytes per key
  - age size is 2 bytes and salary size is 4 bytes
- “Sort by age, and by salary if same age”
  - Age = 20 $\wedge$ salary = 10000 ✔
  - Age = 20 $\wedge$ salary > 10000 ✔
  - Age > 20 $\wedge$ salary = 10000 ?
  - Age > 20 $\wedge$ salary > 10000 ?

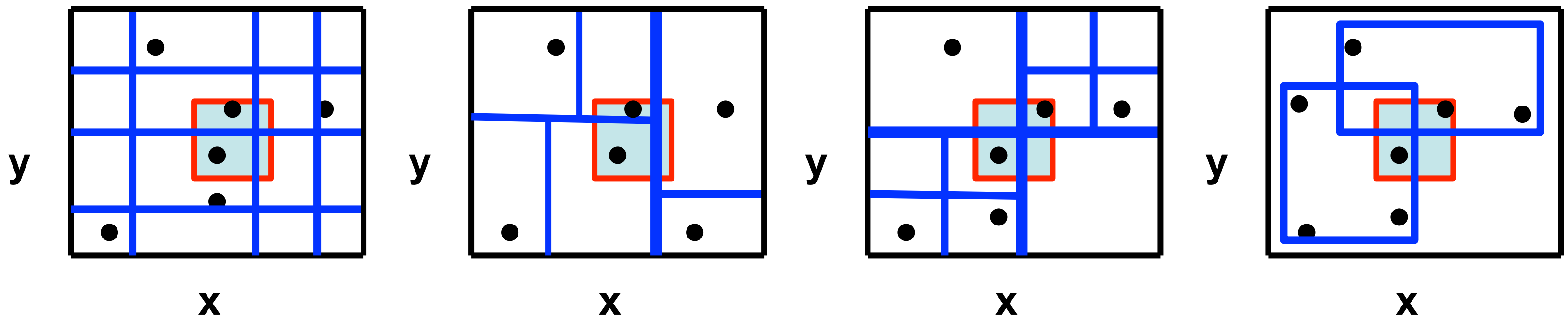
age	salary
20	18000
22	8000
22	11000
22	25000
22	40000
22	80000
24	11000
24	25000
25	25000
26	11000

salary	age
8000	22
11000	22
11000	24
11000	26
18000	20
25000	22
25000	24
25000	25
40000	22
80000	22

Multi-Dimensional Indexes

- Not same as multi-attribute composite
- Assume 2 attributes (dimensions)
- Often used in spatial DB and GIS
- Some examples:

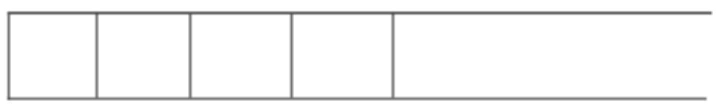




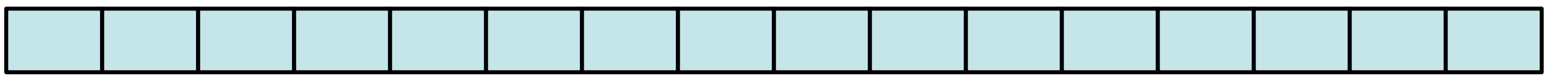
(a) grid file (b) k-d tree (d=2) (c) quad-tree (d) R*-tree

Bulk-Building B$^+$-Trees

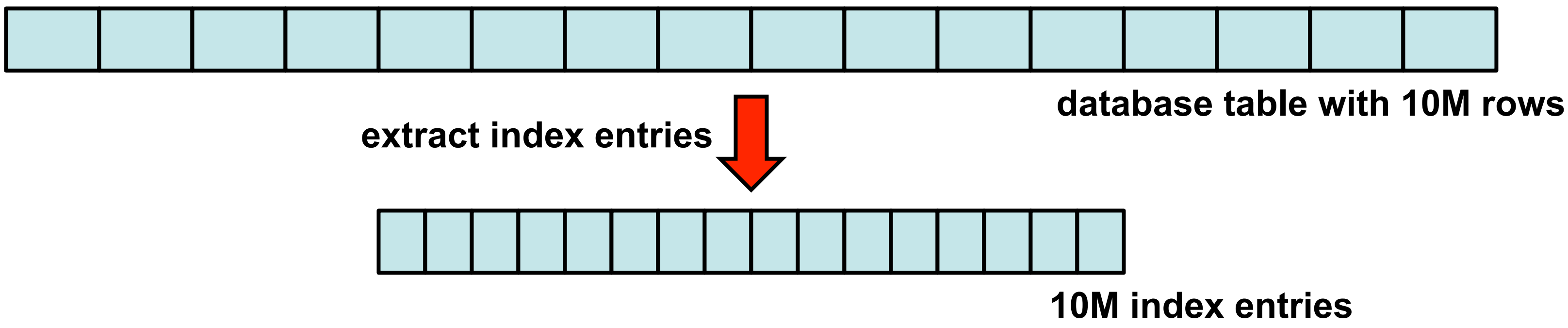
- Determine desired occupancy (90%)
- Extract (key, pointer) pairs from relation (scan)
- Sort pairs

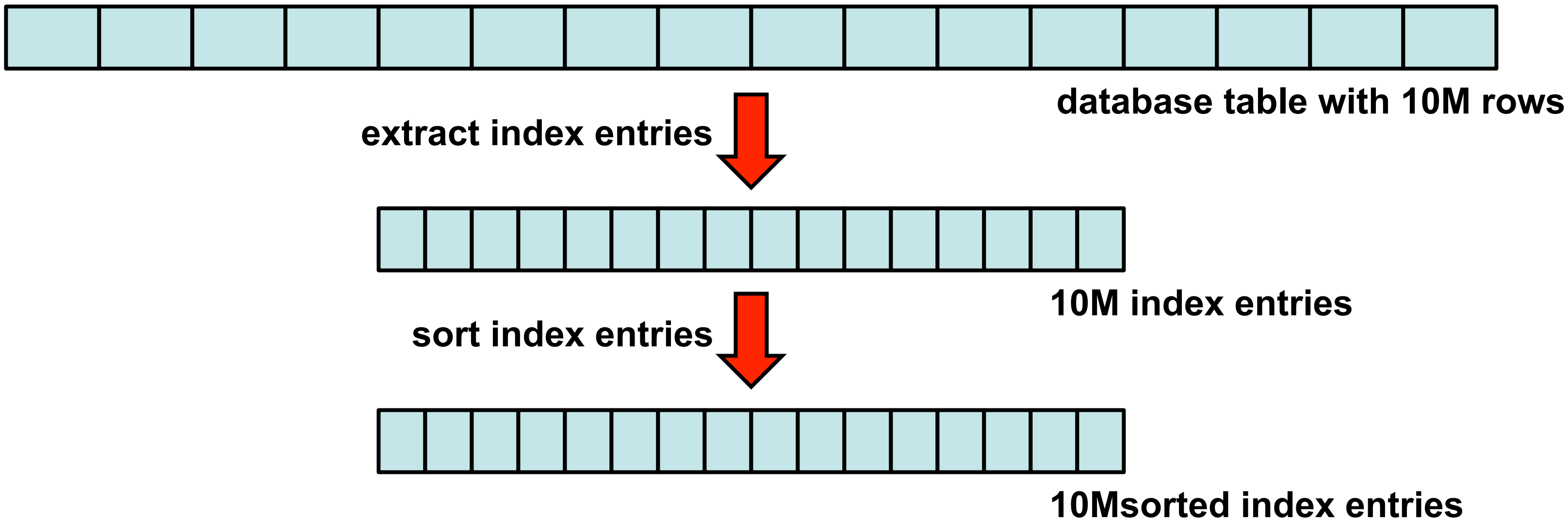


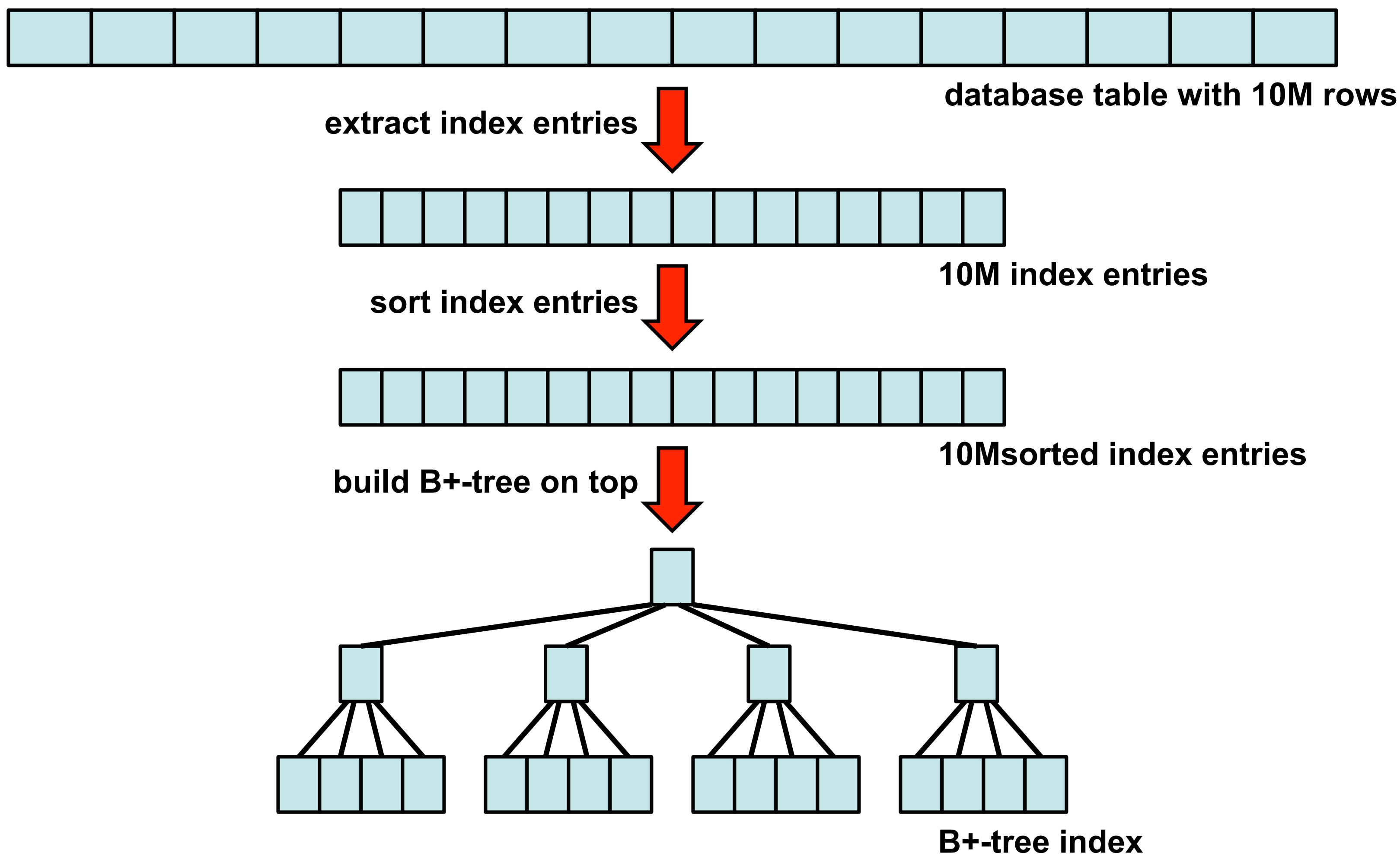
- Build leaf nodes by scanning pairs in sorted order until node 90% full
- Repeat on next level
- Much much faster than inserting one item at a time



database table with 10M rows

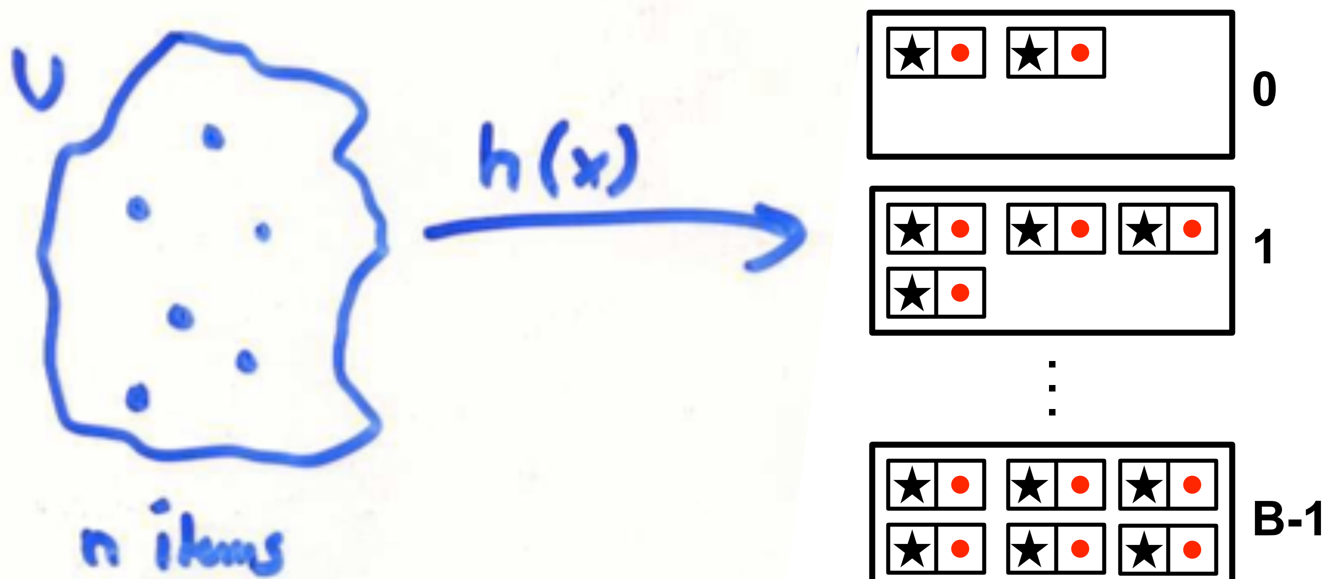

database table with 10M rows
extract index entries
10M index entries


database table with 10M rows
extract index entries
10M index entries
sort index entries
10Msorted index entries


database table with 10M rows
extract index entries
10M index entries
sort index entries
10Msorted index entries
build B+-tree on top
B+-tree index

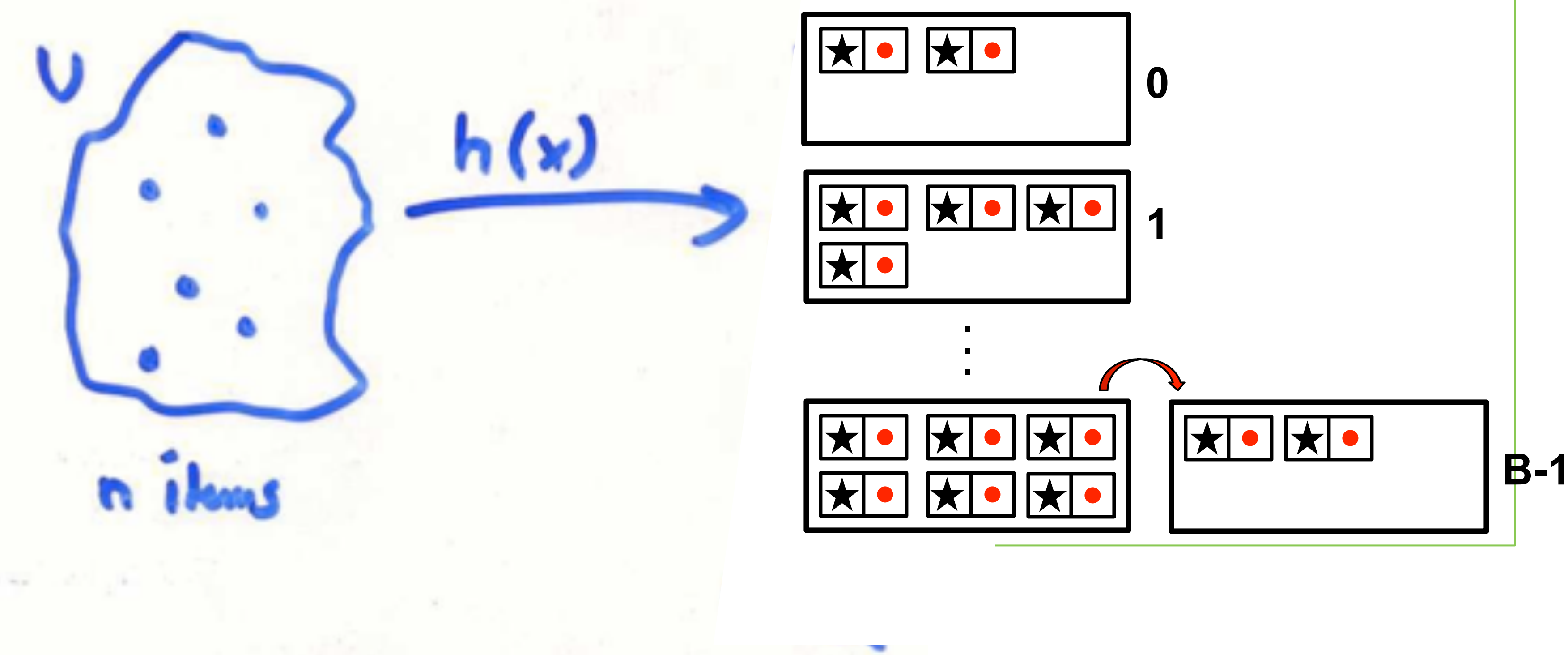
Hashing Basics

- Use hash function to assign index entries in to buckets
- If bucket ≈ block then great !
- K = key domain, B = # buckets, h: K ➔ [0, 1,2,…,B-1]



Problem: Some buckets may overflow

- Why does this happen?
- Static hashing: must add overflow blocks



Extendible Hashing

- Hash h: K ➔ [0,1,2,…, 2^{32}-1]   (or even more bits)
- But at first only look at a few bits

```
10100101101110011001 ...
01101010010110110101 ...
01011010010110100101 ...
10010100101101111011 ...
10110101101110101001 ...
```

Extendible Hashing

- Hash h: K → [0,1,2,…, 2^{32}-1]   (or even more bits)
- But at first only look at a few bits

```
1010010110111001100 1 ...
```

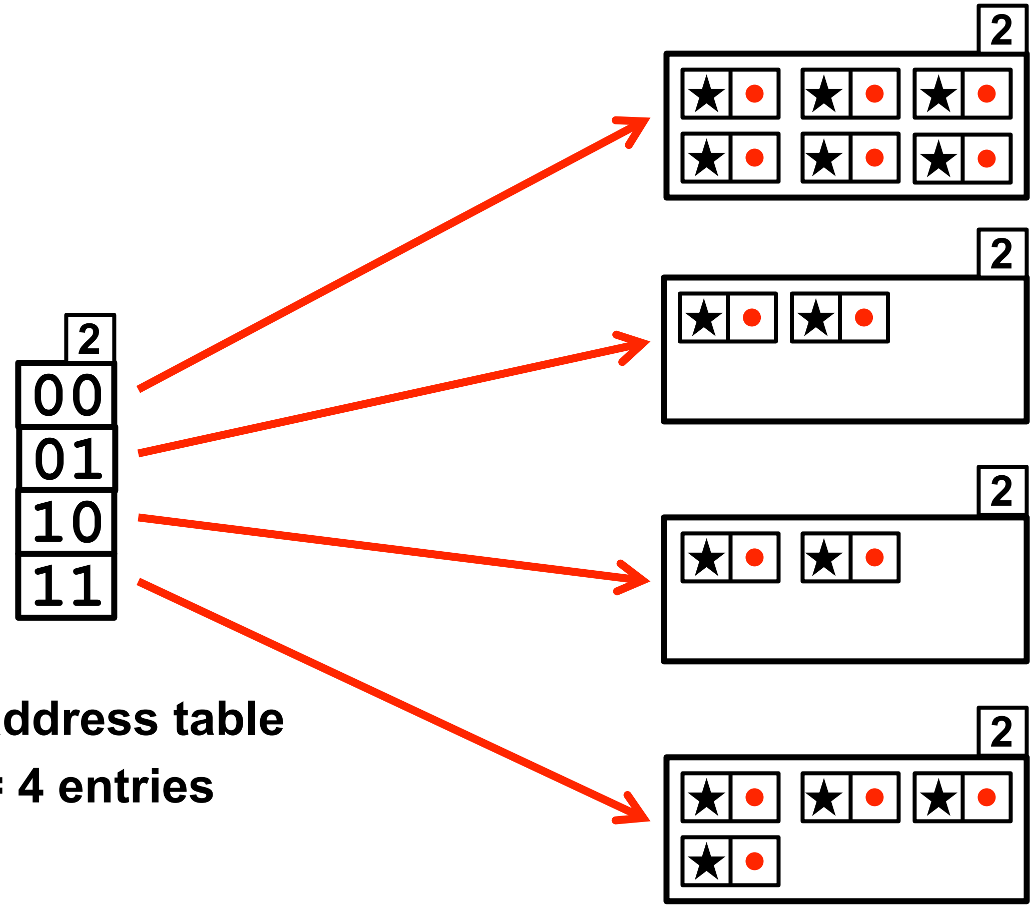
- Maybe only 16 buckets first → 4 bits

Extendible Hashing

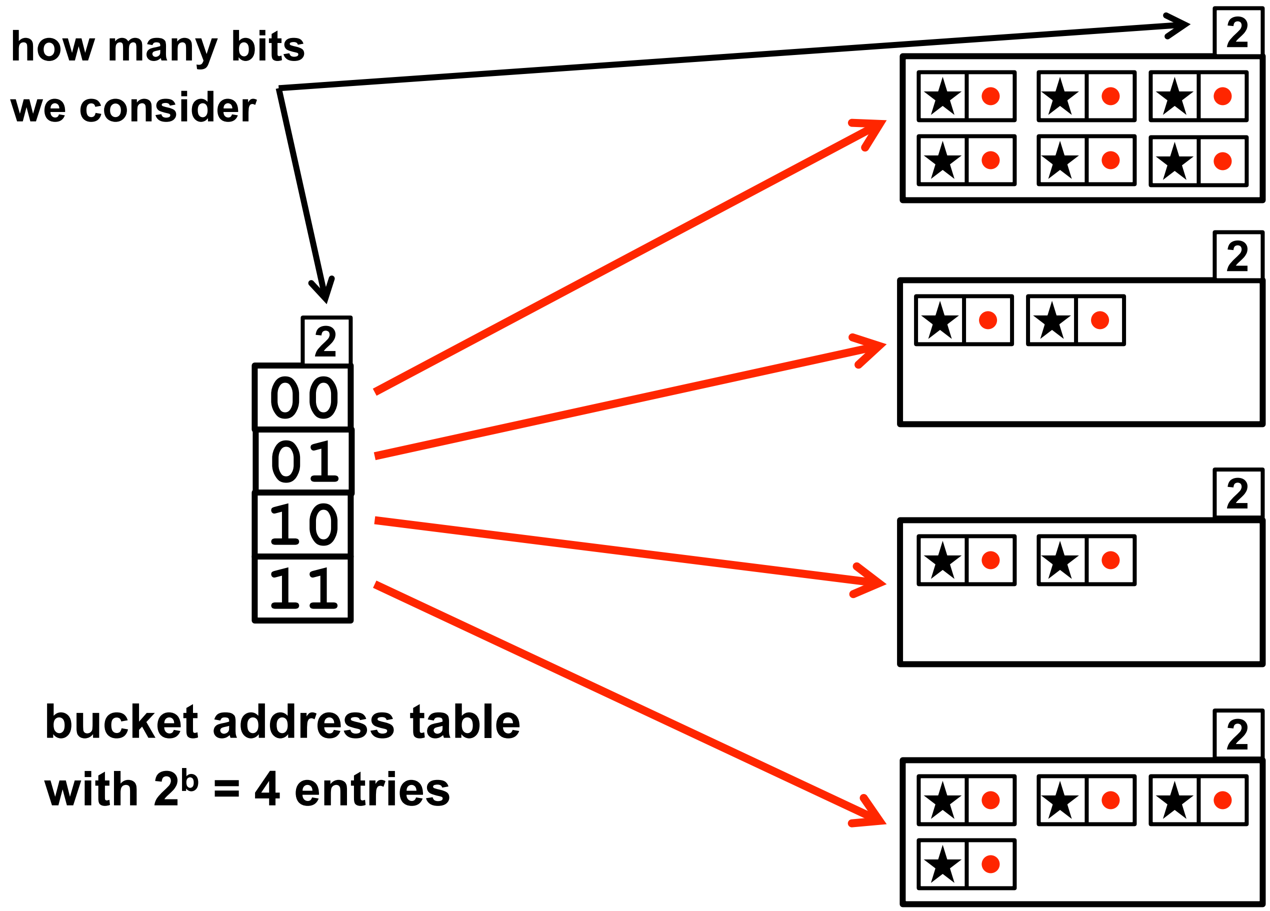
- Hash h: K → [0,1,2,…, 2^{32}-1]   (or even more bits)
- But at first only look at a few bits

```
10100101101110011001 ...
01101010010110110101 ...
01011010010110100101 ...
10010100101101111011 ...
10110101101110101001 ...
```

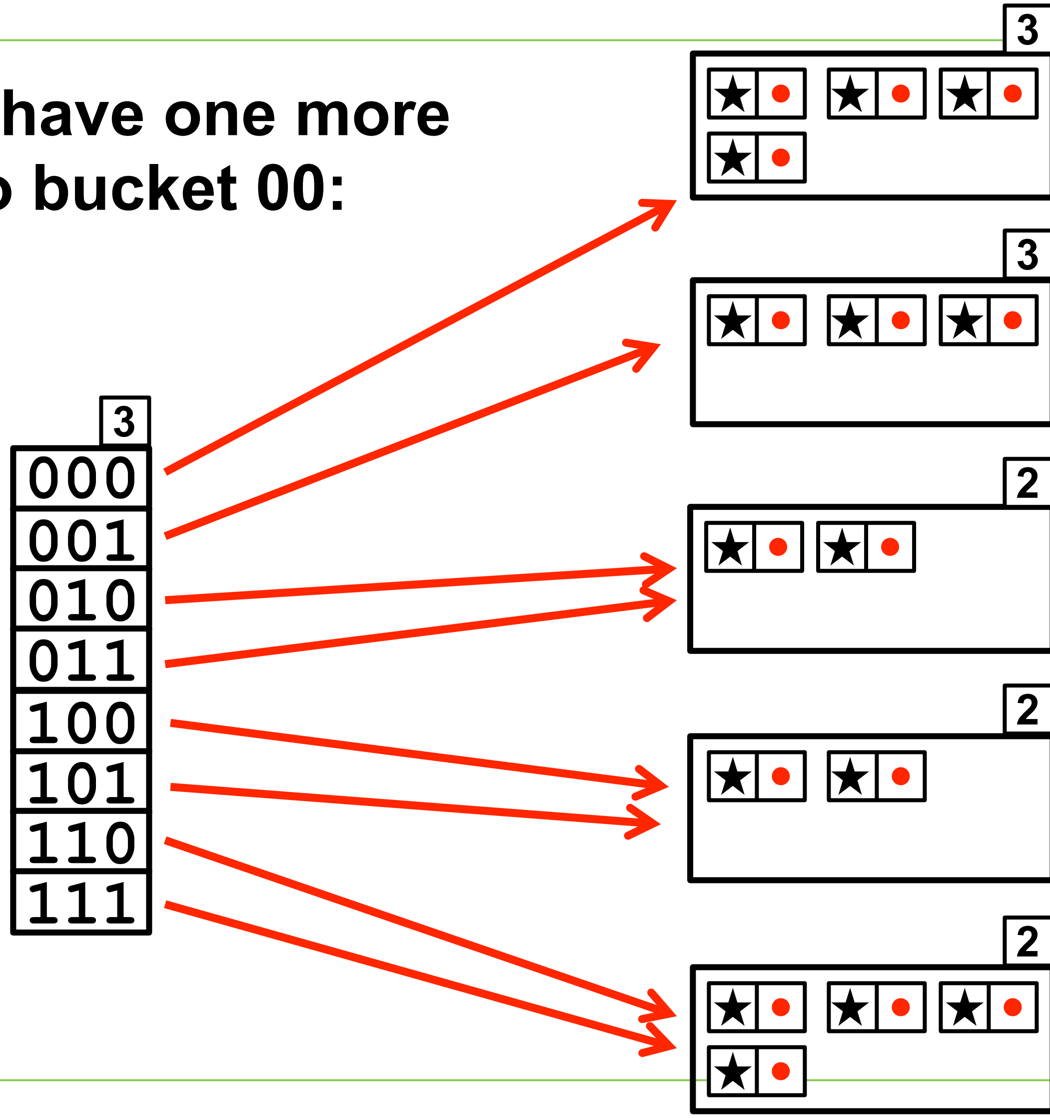
- Maybe only 16 buckets first → 4 bits
- Later: look at more bits



bucket address table
with $2^b = 4$ entries


how many bits
we consider
2
00
01
10
11
2
2
2
2
bucket address table
with 2^b = 4 entries

- Suppose we have one more insertion into bucket 00:



Extendible Hashing

- Uses bucket address table (BAT)
- No need to split non-full buckets
- BAT should fit in main memory
- Note: BAT about size of leaf level of dense B+-tree

- Cost to find one record: 2 seeks (bucket, record)
- Still need overflow buckets