



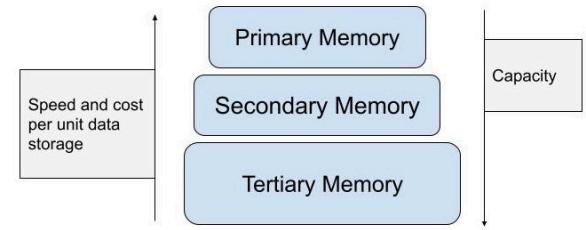
# Introduction to Data Science and Systems

Zaiqiao Meng  
[Zaiqiao.Meng@glasgow.ac.uk](mailto:Zaiqiao.Meng@glasgow.ac.uk)

Lecture 9 - Indexing

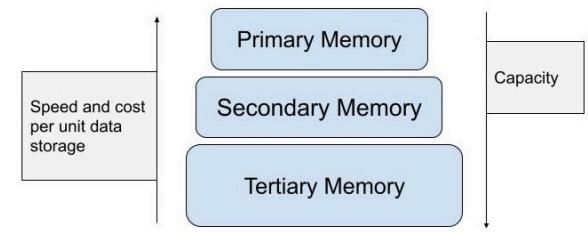
# Recap of Physical Storage

- 3-level hierarchy for data systems
  - Data must **first** be loaded into **primary** from **secondary**
  - Secondary storage access time becomes the bottleneck
- Physical representation of secondary storage
  - Records are stored in blocks
  - **File** is a set of **blocks**, and each block a set of **records**



# Recap of Physical Storage

- 3-level hierarchy for data systems
  - Data must **first** be loaded into **primary** from **secondary**
  - Secondary storage access time becomes the bottleneck
- Physical representation of secondary stor
  - Records are stored in blocks
  - **File** is a set of **blocks**, and each block a set of **records**
- We measure I/O cost for:
  - Retrieving/Inserting/deleting/updating a record x from memory to the disk (update cost)
  - **Cost Function:** # of block accesses (read/write) to search/insert/delete/update record x



# Recap of Physical Storage

- File Organizations
  - Heap (random order) files:
    - Suitable when typical access is a file scan retrieving all records.
  - Sequential Files:
    - Best if records must be retrieved in some order, or only a ‘range’ of records is needed.
  - Hash Files
    - Find records in constant time by hash functions.
    - Cannot efficiently scan sequentially or for range queries

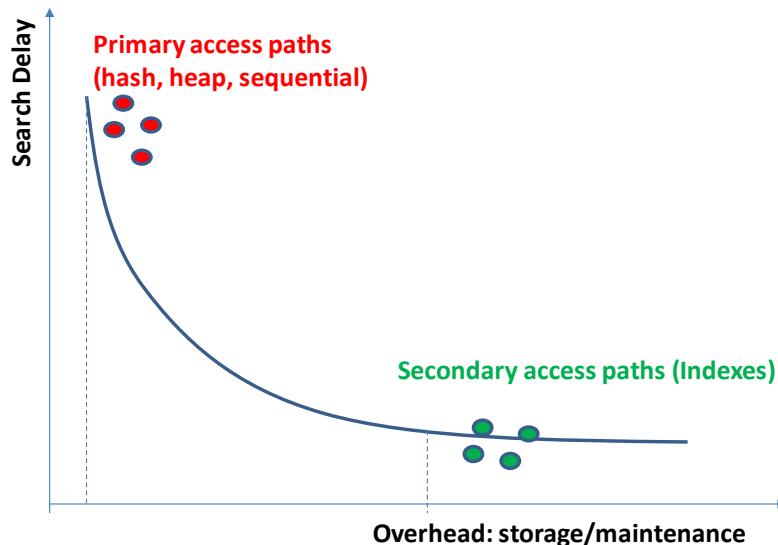


# Recap of Physical Storage

- File Organizations
  - Heap (random order) files:
    - Suitable when typical access is a file scan retrieving all records.
  - Sequential Files:
    - Best if records must be retrieved in some order, or only a ‘range’ of records is needed.
  - Hash Files
    - Find records in constant time by hash functions.
    - Cannot efficiently scan sequentially or for range queries
  - Indexes:
    - Data structures to organize records via trees.
    - Like Sequential files, they speed up searches for a subset of records, based on values in certain (“search key”) fields
    - Updates are much faster than in sequential files.

# Objectives

- **Physical Design**
  - Objective: given a specific file type provide a **primary access path** based on a **specific field**; e.g., search via SSN
- **Index Design**
  - Objective: given any file type provide **secondary access paths** using **more than one fields**; e.g., search via SSN, Salary, Name, etc.
- **Cost:** Additional meta-data file on the disk (index)
- **Benefit:** Significantly accelerate the search process avoiding Linear Scans



# Principles & Indexes: 1-d space

- Key Principles:
  - #1: An index is created over predefined (index) field(s)
  - #2: An index is a file separate to that whose data it indexes
  - #3: All index entries are unique and sorted w.r.t. the index field
- Key Hypotheses:
  - #1: Index file occupies less blocks than the data file
    - Fact: index entries are much smaller than data file records:  
*{index-key, block-pointer}*
  - #2: Searching through the index is faster than through the data file
    - Fact: By definition, index is an ordered file; can use binary- and/or tree-based methods to find a record given its key

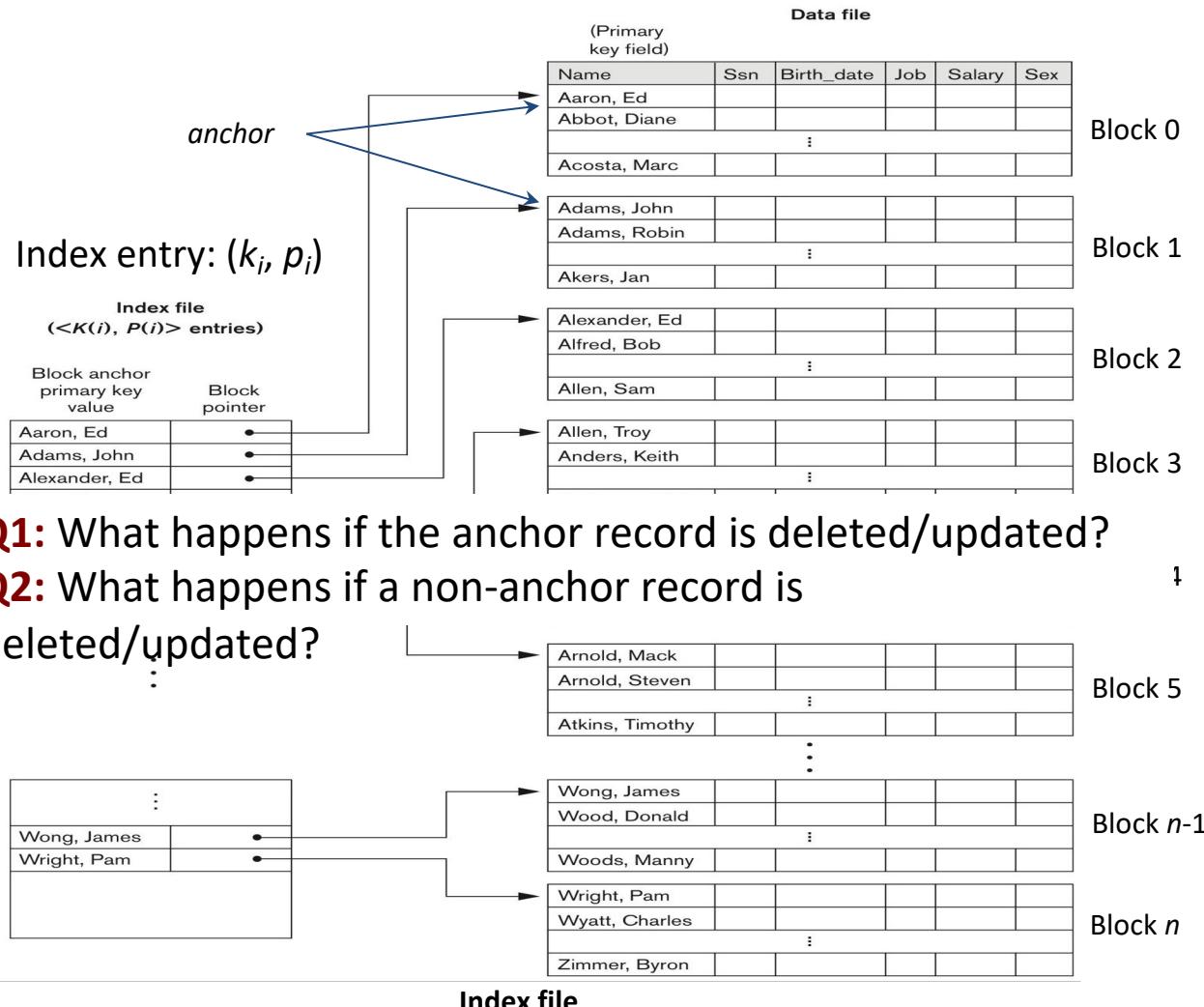
# Principles & Indexes: 1-d space

- Major index types:
  - **Primary Index:** index field is an ordering-key field of a sequential file
    - e.g., SSN; relation is sorted w.r.t. SSN
  - **Clustering Index:** index field is an ordering non-key field of a seq. file
    - e.g., Age; relation is sorted w.r.t. Age
  - **Secondary index:** index field is a non-ordering field over an ordered or non-ordered file...
    - e.g., key-field *unique passport number*
    - e.g., non-key-field *salary*
- Indexes can be:
  - **Dense:** an index entry for every record in the file
  - **Sparse:** index entries only for some of records

# Primary Index

- An ordered file over an **ordering key field** of a sequential data-file:
    - fixed-length index entries := pair  $\{k_i, p_i\}$
    - $k_i$  is a value of the index field
    - $p_i$  is a pointer to the block containing the record with key  $k_i$
- Recall: **unique** value for every record; used to **physically order records** in the file
- **Sparseness:** one index-entry per data block
    - $i^{\text{th}}$  index entry  $\{k_i, p_i\}$  refers to the  $i^{\text{th}}$  data block
    - **Anchor** of block  $i$ : The first data-record in the block -- i.e., the one with the index field value  $k_i$
  - Theorem: Create a primary index iff a block can accommodate at least two index entries
    - Proof: cost over the index < cost over linear search

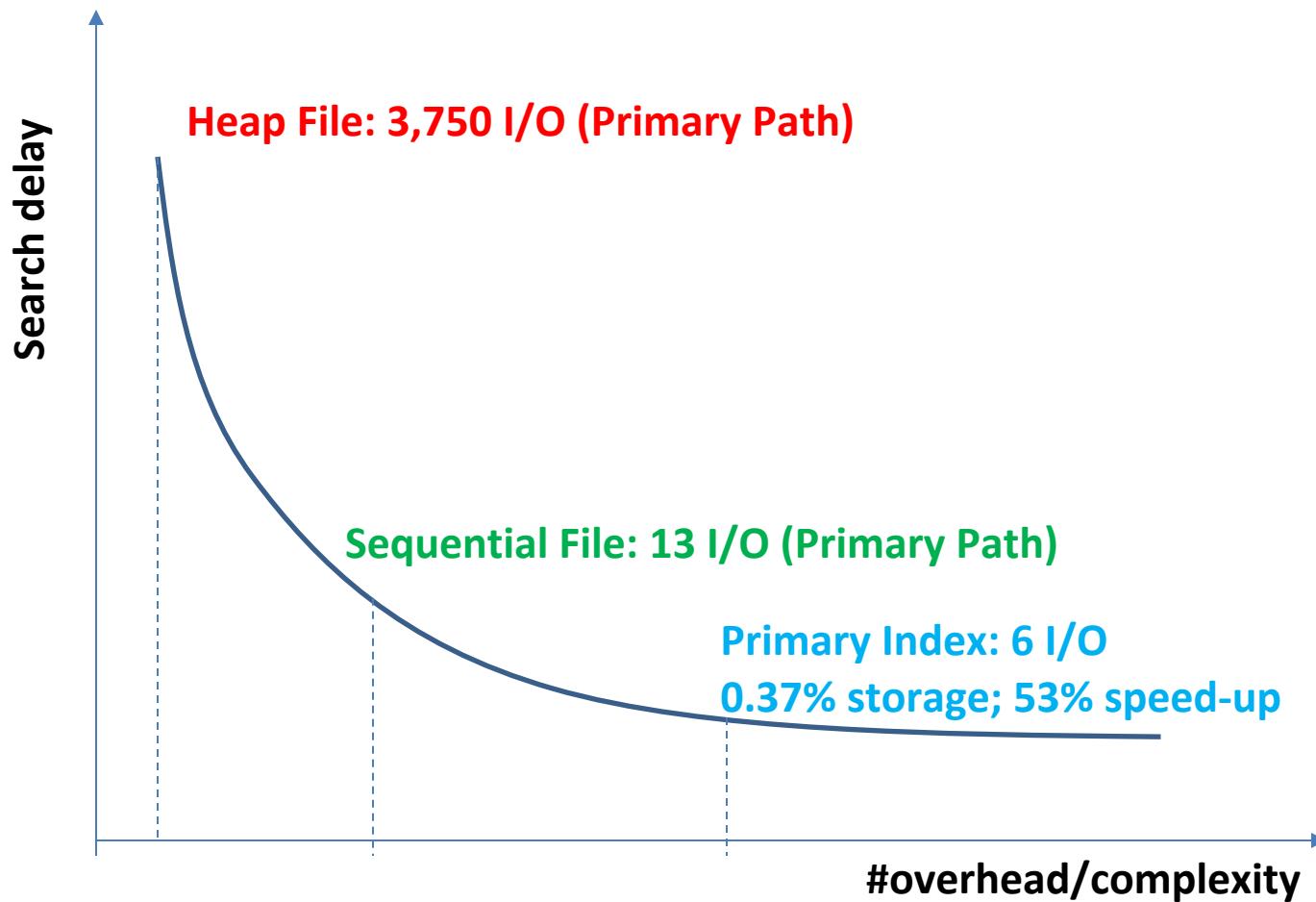
# Primary Index



# In-Class Example

- EMPLOYEE:
  - $r = 300,000$  fixed-length records of size  $R = 100$  bytes each; block size  $B = 4,096$  bytes; SSN size = 9 bytes; pointer size = 6 bytes
- Task: Expected cost of  $\text{SELECT * FROM EMPLOYEE WHERE SSN} = 'k'$ 
  - Blocking factor:  $bfr = \text{floor}(B/R) = 40$  records per block
  - File:  $b = \text{ceil}(r/bfr) = \mathbf{7,500 \text{ blocks}}$
  - Linear Search over the File:  $b/2 = \mathbf{3,750 \text{ block accesses}}$
  - Binary Search/ordered by key SSN:  $\text{ceil}(\log_2(b)) = \mathbf{13 \text{ block accesses}}$
- Create Primary Index on SSN:  $\text{CREATE INDEX ON EMPLOYEE(SSN)}$ 
  - Index Entry: {SSN, Pointer}; size: V = 9 bytes for SSN and P = 6 bytes for Pointer
  - Index Blocking Factor:  $ibfr = \text{floor}(B/(P+V)) = 273$  entries/block
  - Primary Index requires 7,500 entries: one per block ( $b = 7500$ )
  - Index blocks:  $ib = \text{ceil}(7,500/273) = \mathbf{28 \text{ blocks}}$
  - Storage overhead: **28 blocks**; **0.37%** additional storage
  - Performance gain: Binary Search on Index:  $\text{ceil}(\log_2(ib)) = 5$  block accesses + one block access to load the data block pointed by index = **6 block accesses**; **53% (Binary)** and **99.84% (Linear) speed up**

# Trade off: Overhead vs Speed

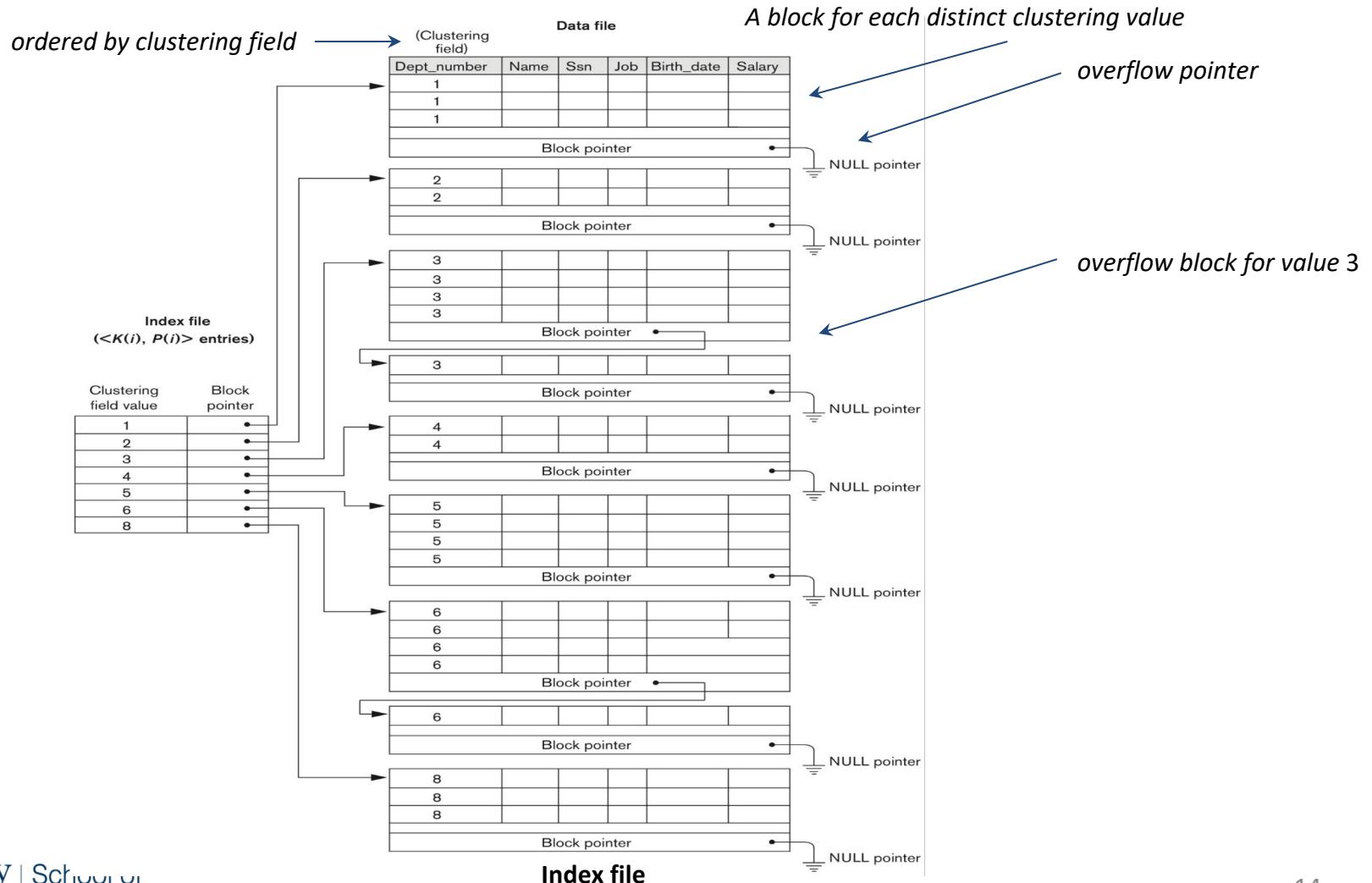


# Clustering Index

- Challenge: Index a sequential file on a **non-key ordering** field
  - e.g., create an index on EMPLOYEE ordered by DNO (dept. no)
- Idea: The file is a *set of clusters*, one per distinct value of the above field (a.k.a. the *clustering* field)
  - **index-entry** := {**clustering-value**, **block-pointer**}
    - One *index-entry* per *distinct clustering value*
  - Block pointer points to the *first* block of a *cluster*
  - Is the clustering index sparse or dense?

Recall: used to **physically order records** in the file;  
**non unique** values

# Clustering Index



# Analysis

$$\sum_{k=1}^n \left(\frac{b}{n}\right) \left(\frac{1}{n}\right) k = \left(\frac{b}{n^2}\right) \left(\frac{n(n+1)}{2}\right) = \frac{b(n+1)}{2n}$$

$n$  = number of clusters  
 $b$  = number of blocks  
lustering index:  $m$  blocks

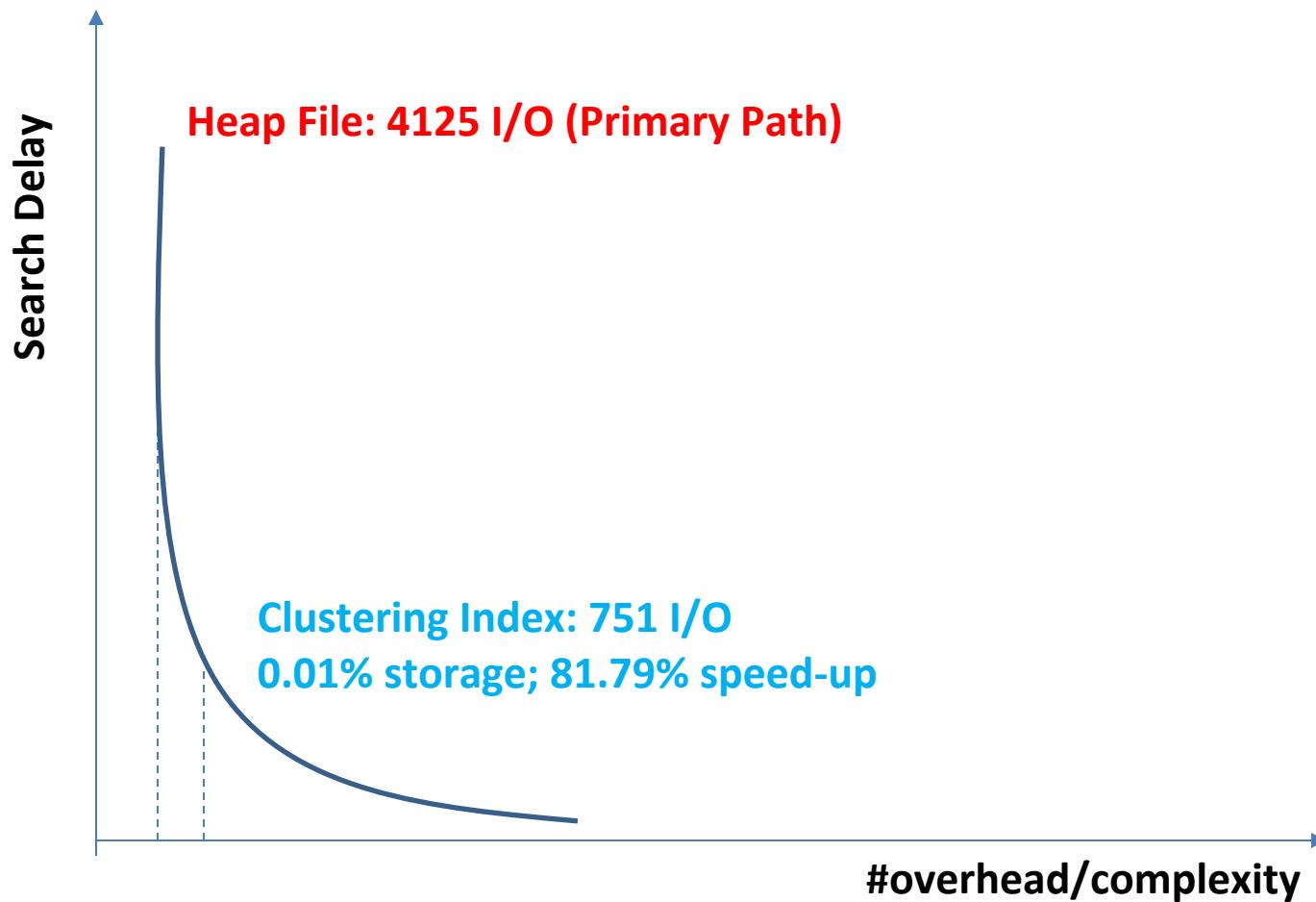
$$\lim_{n \rightarrow \infty} \frac{b(n+1)}{2n} = \left(\frac{b}{2}\right)$$
$$m < 2^{\frac{b(n+1)}{2n}}$$

where m is the number of index blocks

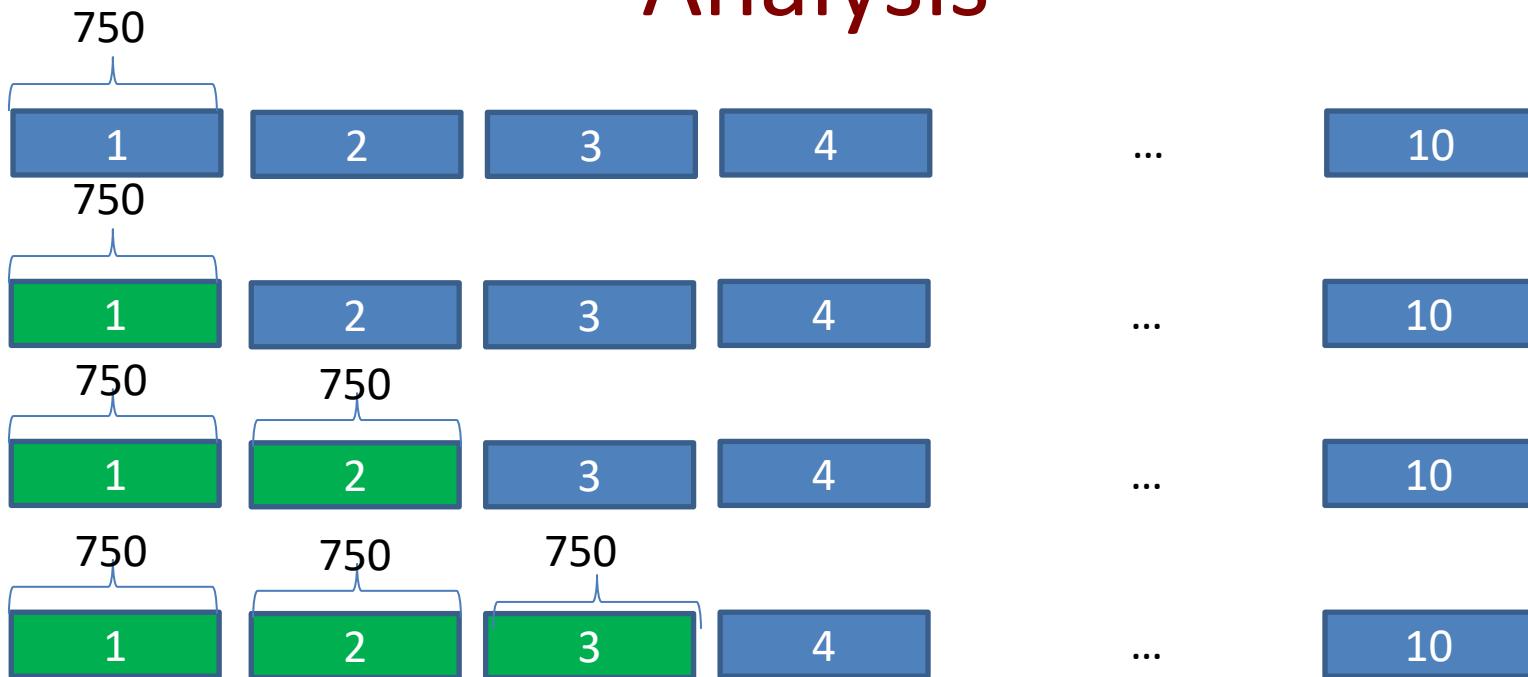
- Proof: cost over the index < cost over linear search

- EMPLOYEE:  $r = 300,000$  fixed-length records of size  $R = 100$  bytes each; block size  $B = 4,096$  bytes; DNO size = 9 bytes;  $P = 6$  bytes (pointer); ordered by DNO
- Assume: 10 departments; DNO values are uniformly distributed over the tuples
- Compute the expected cost of `SELECT * FROM EMPLOYEE WHERE DNO = 'd'`
- Compare with naïve linear search over the data file
  - Data File:  $b = \text{ceil}(r/bfr) = 7,500$  blocks
  - Index Entry: {DNO, Pointer}; size:  $V = 9$  bytes for DNO and  $P = 6$  bytes for a Pointer
  - Index Blocking Factor:  $\text{ibfr} = \text{floor}(B/(P+V)) = 273$  entries/block
  - Clustering Index requires **10 index entries**: one per cluster!
  - Index blocks:  $ib = \text{ceil}(10/273) = 1$  block 😊 (0.01% additional storage)
  - Performance gain:
    - Search on Index: 1 block access
    - $7500/10 = 750$  block accesses to load the data blocks belonging to a cluster
    - Total:  $1 + 750 = 751$  block accesses (**81.79%**)
    - Linear Search on File: **4,125 block accesses! (not simply  $7,500/2 = 3,750$ )**

# Trade off: Overhead vs Speed



# Analysis



$$1/10(750) + 1/10(750+750) + 1/10(3\cdot750) + \dots + 1/10(10\cdot750) = 4125$$

$$\sum_{k=1}^n \left(\frac{b}{n}\right) \left(\frac{1}{n}\right) k = \left(\frac{b}{n^2}\right) \left(\frac{n(n+1)}{2}\right) = \frac{b(n+1)}{2n}$$

$n$  = number of clusters  
 $b$  = number of blocks

$$\log_2(m) + \frac{b}{n}$$

*Clustering index:  $m$  blocks*

# Clustering Index

- **Fact:** If  $n \rightarrow \infty$ , i.e., infinite number of distinct values, then the linear search over an ordering non-key field is  $b/2$ :

$$\lim_{n \rightarrow \infty} \frac{b(n+1)}{2n} = \left(\frac{b}{2}\right)$$

- **Conclusion:** **meaningless** to sort a file w.r.t. a non-key field when  $n$  is large (compared to the number of tuples)
- Create a clustering Index of  $m$  blocks iff:  
$$m < 2^{\frac{b(n+1)}{2n}}$$
where  $m$  is the number of index blocks
- Proof: cost over the index < cost over linear search

# Range queries over a clustering index

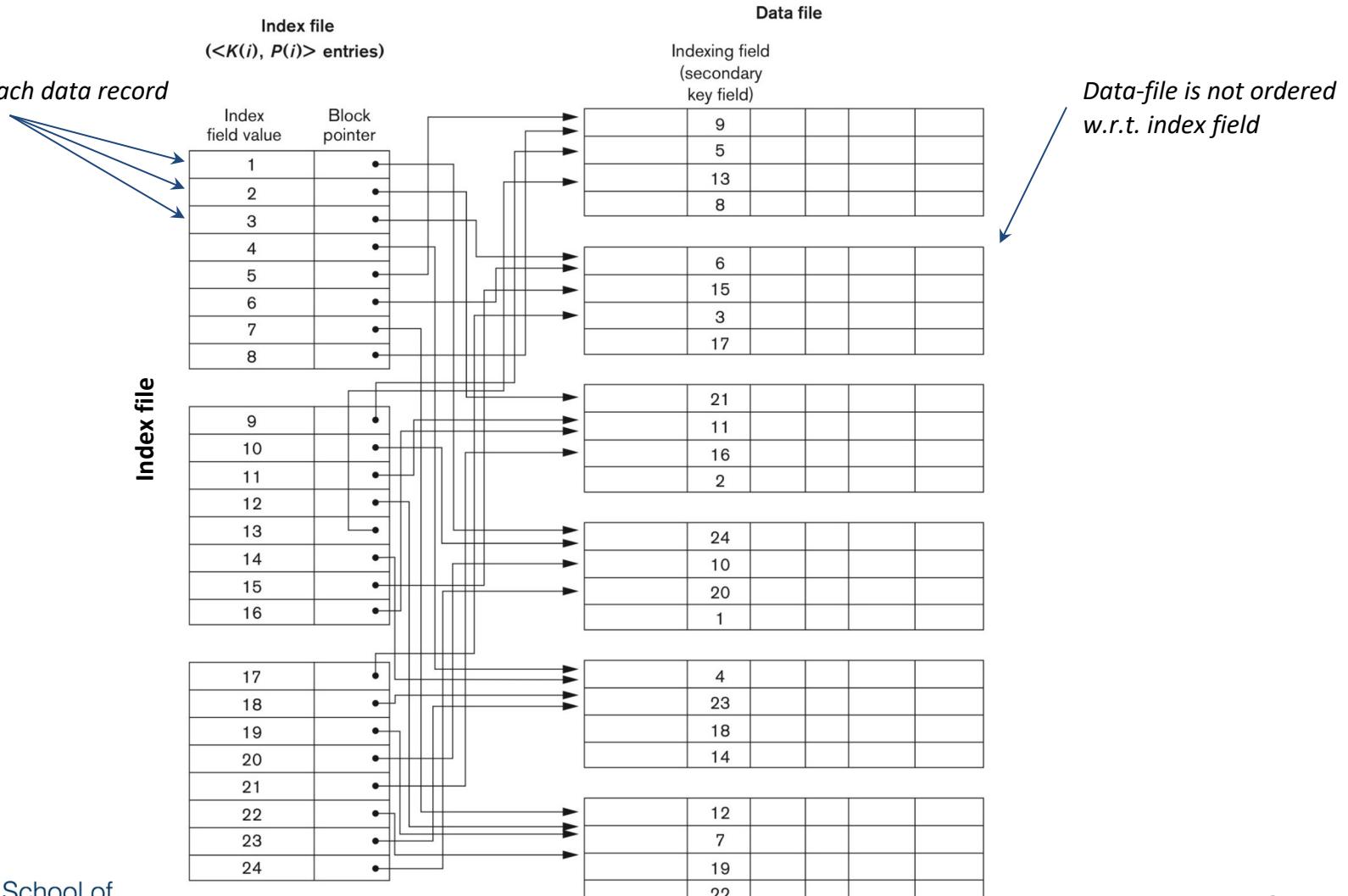
- Query index for block of lower bound value:
  - Expected cost  $\log_2(m)$  block accesses
- Fetch data for lower bound cluster value:
  - Expected cost  $(b/n)$  block accesses
- Assuming  $L$  additional clusters will be fetched in total:
  - Expected cost  $L * (b/n)$  block accesses
- Total:  $\log_2(m) + (L + 1)*(b/n)$  block accesses
- Linear search:  $b$  block accesses
- Benefit iff:
$$\log_2(m) + (L + 1) * (b/n) < b \Leftrightarrow m < 2^{b(n-(L+1))/n}$$

# Secondary Index

- Challenge: Index a file on a non-ordering field
    - The file might be unordered, hashed, or ordered but not according to the indexing field
  - Cases:
    - [S1] Secondary Index on a non-ordering, **key** field; e.g., SSN
    - [S2] Secondary Index on a non-ordering, **non-key** field; e.g., DNO
  - Type [S1]: One index entry per data record --> a **dense** index
  - Why?
    - The file is **not** ordered on the indexing field --> we **cannot** use anchor records
- i.e.; **unique**
- i.e.; **not necessarily unique**

# [S1] Secondary Index (non-ordering; key)

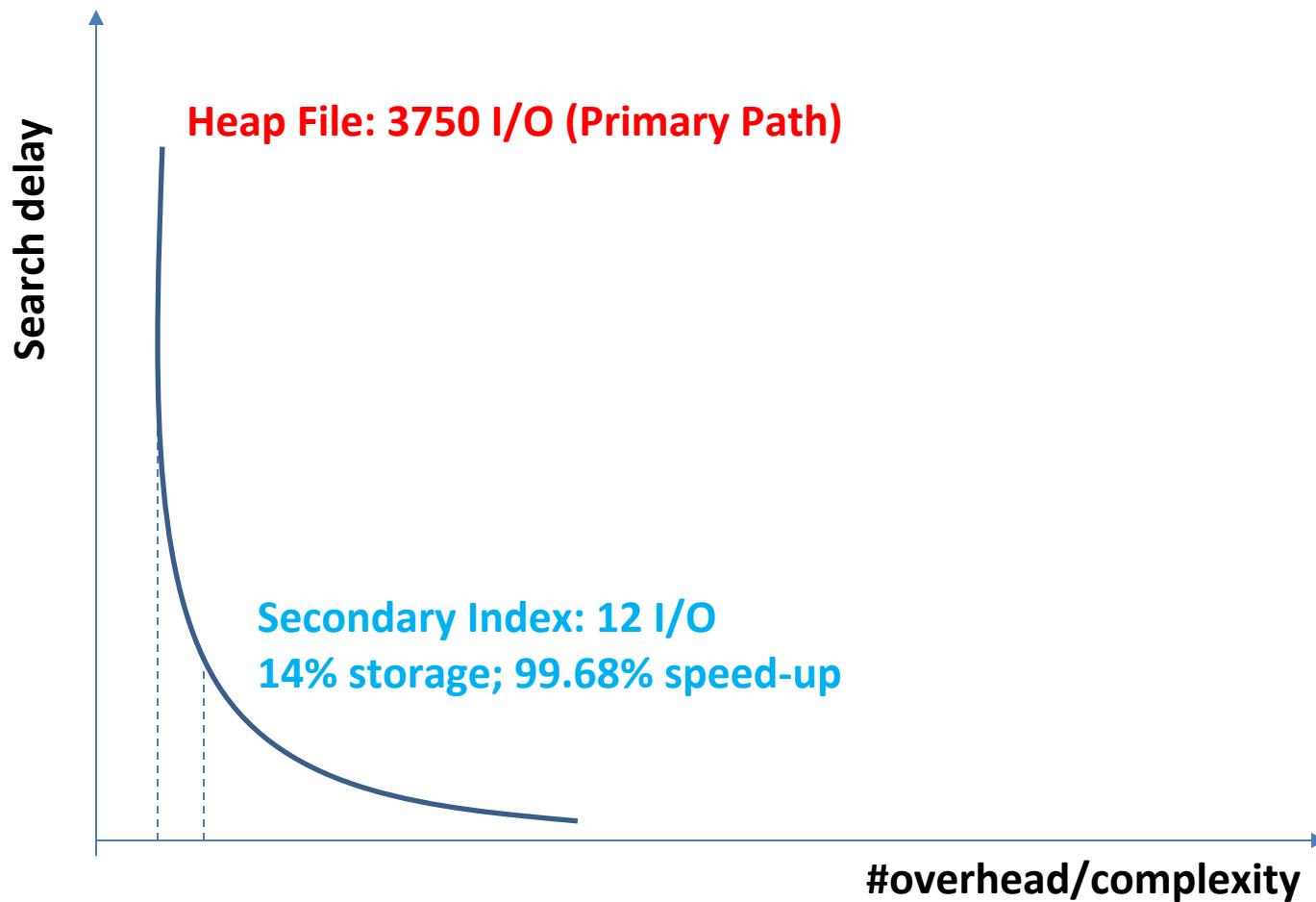
An index entry for each data record



# In-Class Activity

- Task: Secondary Index on a non-ordering, key attribute: ID
- File:  $r = 300,000$  fixed-length records of size  $R = 100$  bytes each; block size  $B = 4096$  bytes; ID is  $V = 9$  bytes, pointer  $P = 6$  bytes
- Compute cost of `SELECT * FROM EMPLOYEE WHERE ID = 'x'`
  - Blocking factor  $bfr = \text{floor}(B/R) = 40$  records per block
  - File blocks:  $b = \text{ceil}(r/bfr) = 7,500$  data blocks
  - Linear Search on File:  $b/2 = 3,750$  block accesses
  - **We cannot do binary search since it is non-ordering**
  - Blocking factor of index:  $ibfr = \text{floor}(B/(V+P)) = 273$  entries/block
  - Secondary Index is dense:  $n = 300,000$  index entries
  - Index File blocks  $ib = \text{ceil}(n/ibfr) = 1,099$  blocks (14% overhead)
  - Cost: Binary Search on Index:  $\text{ceil}(\log_2(ib)) = 11$  block accesses
  - ... plus one more block access to load the block pointed by the index
  - Total:  $11 + 1 = 12$  block accesses (99.68% speed-up)
- We can do better by splitting the data space into more than two subspaces

# Trade off: Overhead vs Speed



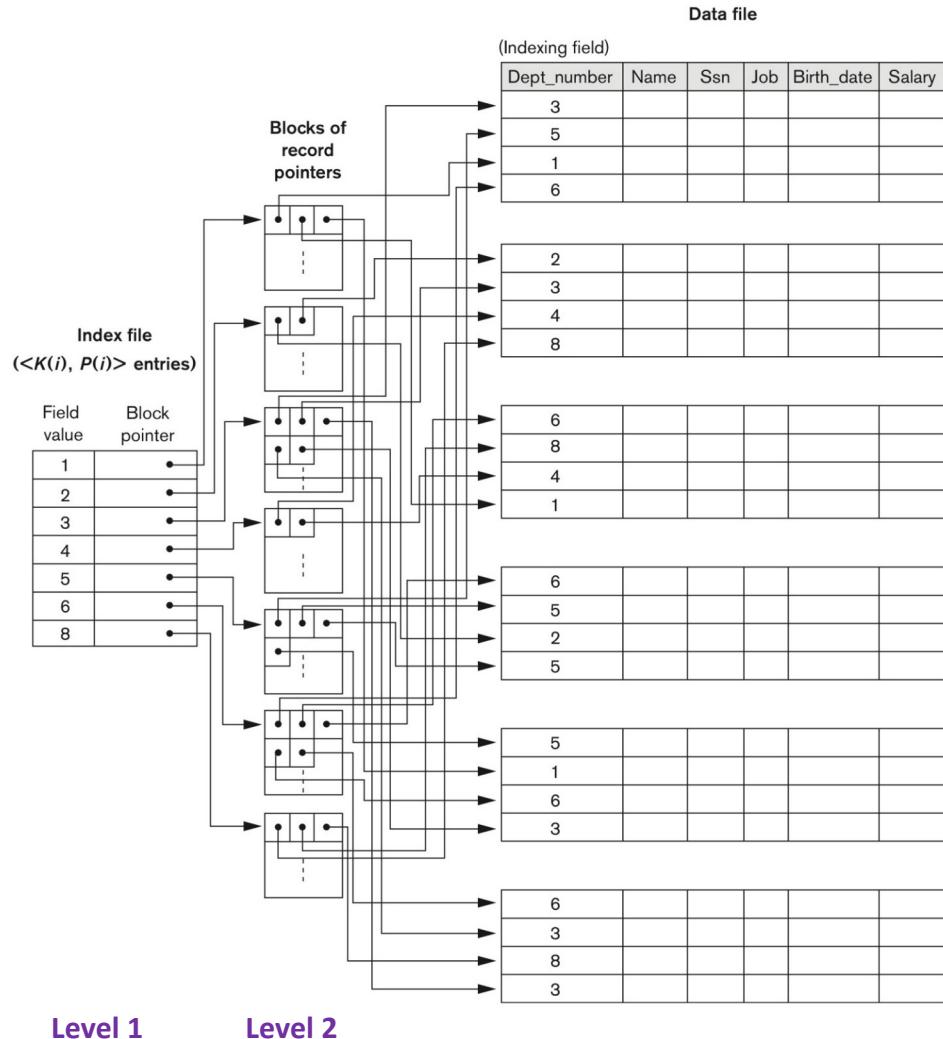
# Secondary Index

- Type [S2]: Indexing field is not a key & the file is not ordered w.r.t. indexing field
- Idea 1: cluster the block addresses of the records having the same index value
- Idea 2: assign an index entry per cluster (of block addresses)
- **index-entry := {cluster-value, cluster-pointer}**

Remember this trick;  
applicable in many cases

- A cluster-pointer points to:
  - (Level 1) a block of {block-pointers} of a cluster
  - (Level 2) a block-pointer points to the data-block that has records with this distinct index value

# [S2] Secondary Index (non-ordering; non-key)



## Note:

- Index is *sparse*
- One entry per cluster
- Level 1 is a set of blocks
- Each Level-1 block contains records of block pointers

## Search for ID = 3:

1. Binary search in Level-1 (1)
2. Direct access to Level-2 (1)
3. Load *all* the corresponding data-blocks (4)

## Total:

1 + 1 + 4 = 6 block accesses

# Multilevel Index

- **Observation:** in all index files it holds true that:
  - They are **ordered** on the indexing field
  - The indexing field has **unique (distinct)** values
  - Each index entry is of **fixed length**

Field value	Block pointer
1	•
2	•
3	•
4	•
5	•
6	•
7	•
8	•

Block anchor primary key value	Block pointer
Aaron, Ed	•
Adams, John	•
Alexander, Ed	•
Allen, Troy	•
Anderson, Zach	•
Arnold, Mack	•
:	

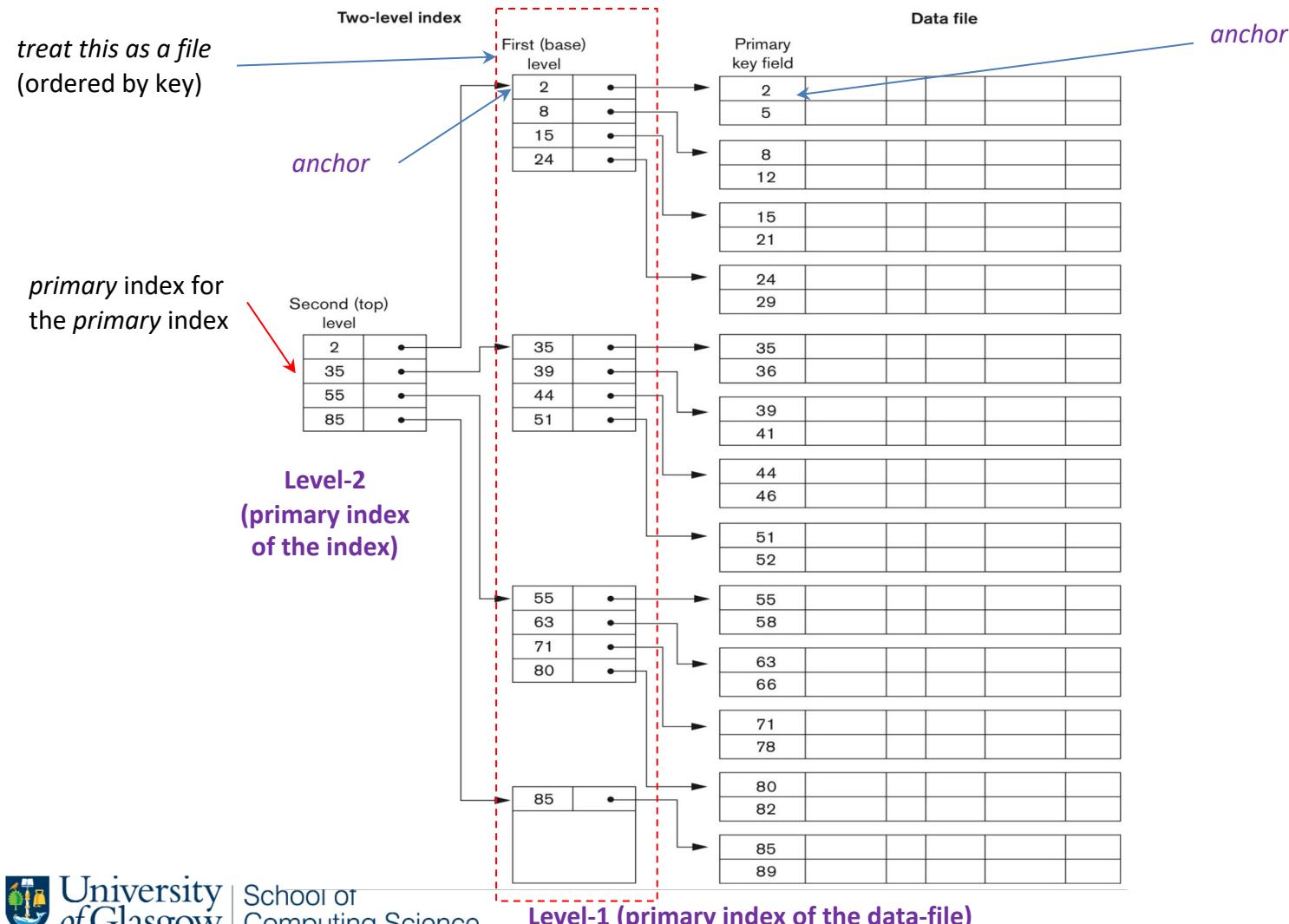
Index field value	Block pointer
1	•
2	•
3	•
4	•
5	•
6	•
7	•
8	•
9	•
10	•
11	•
12	•
13	•
14	•
15	•
16	•

- Conclusion: we can build a **primary index** over **any index file**, since it is an ordered file w.r.t. a key field (*index of an index*)

# Multilevel Index

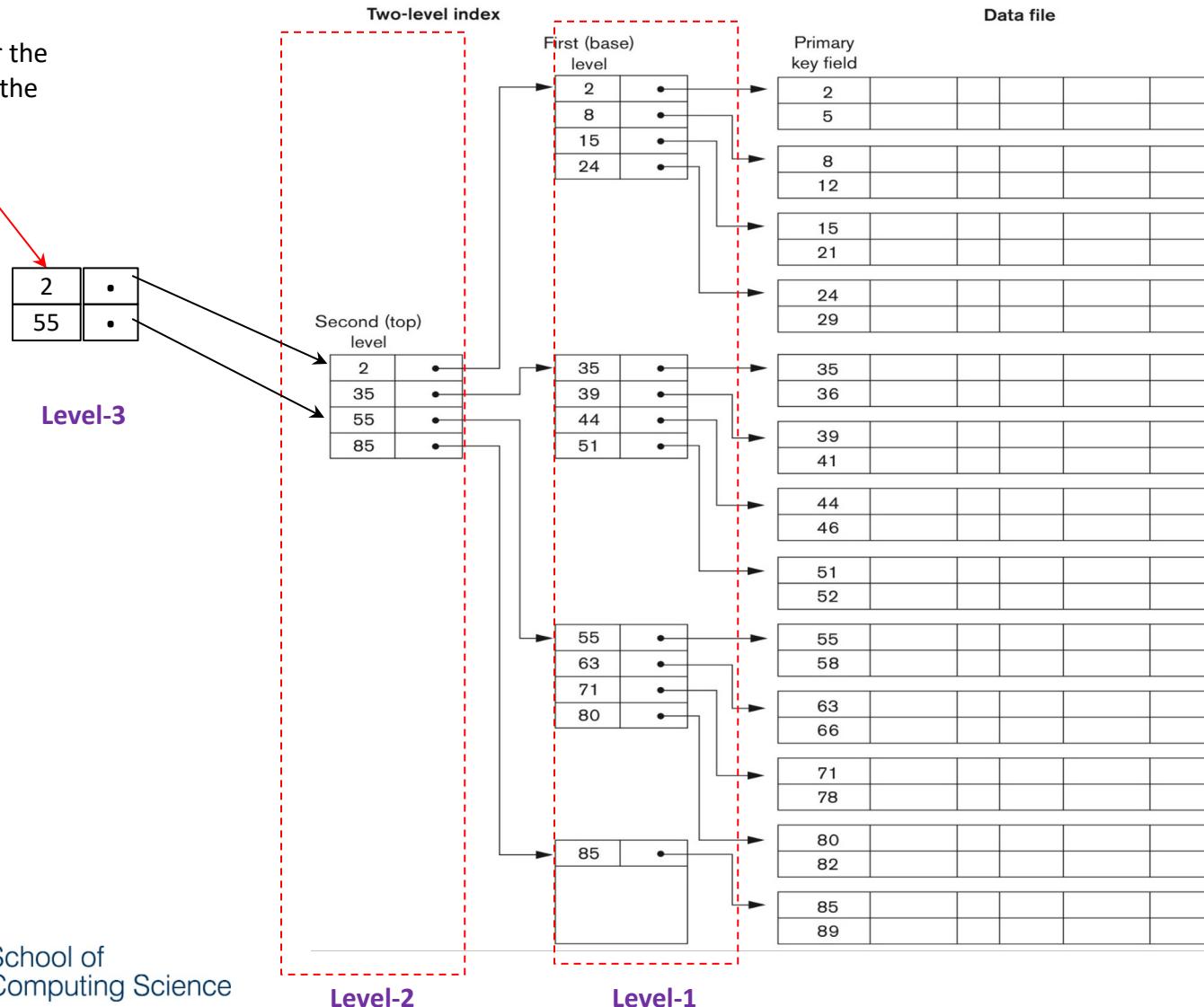
- To index the index...
  - The original index file is referred to as the base or Level-1 index
  - The additional index is referred to as Level-2 index (index of an index)
  - ...
  - if we repeat this to level  $> 2$  we obtain... Level-t index, i.e., index of an index of an index ...
- **Challenge:** Identify the best level  $t$  of a multi-level index to expedite the search process trading off speed-up with overhead

# 2-level Index



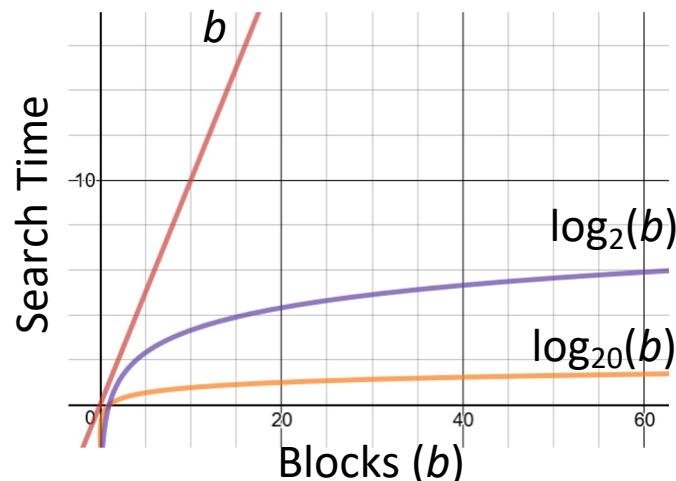
# 3-level Index

*primary index for the  
primary index of the  
primary index*



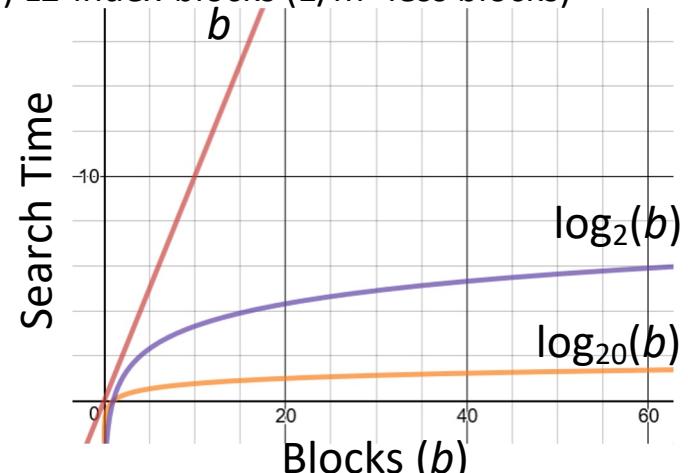
# Multilevel Index: Reasoning

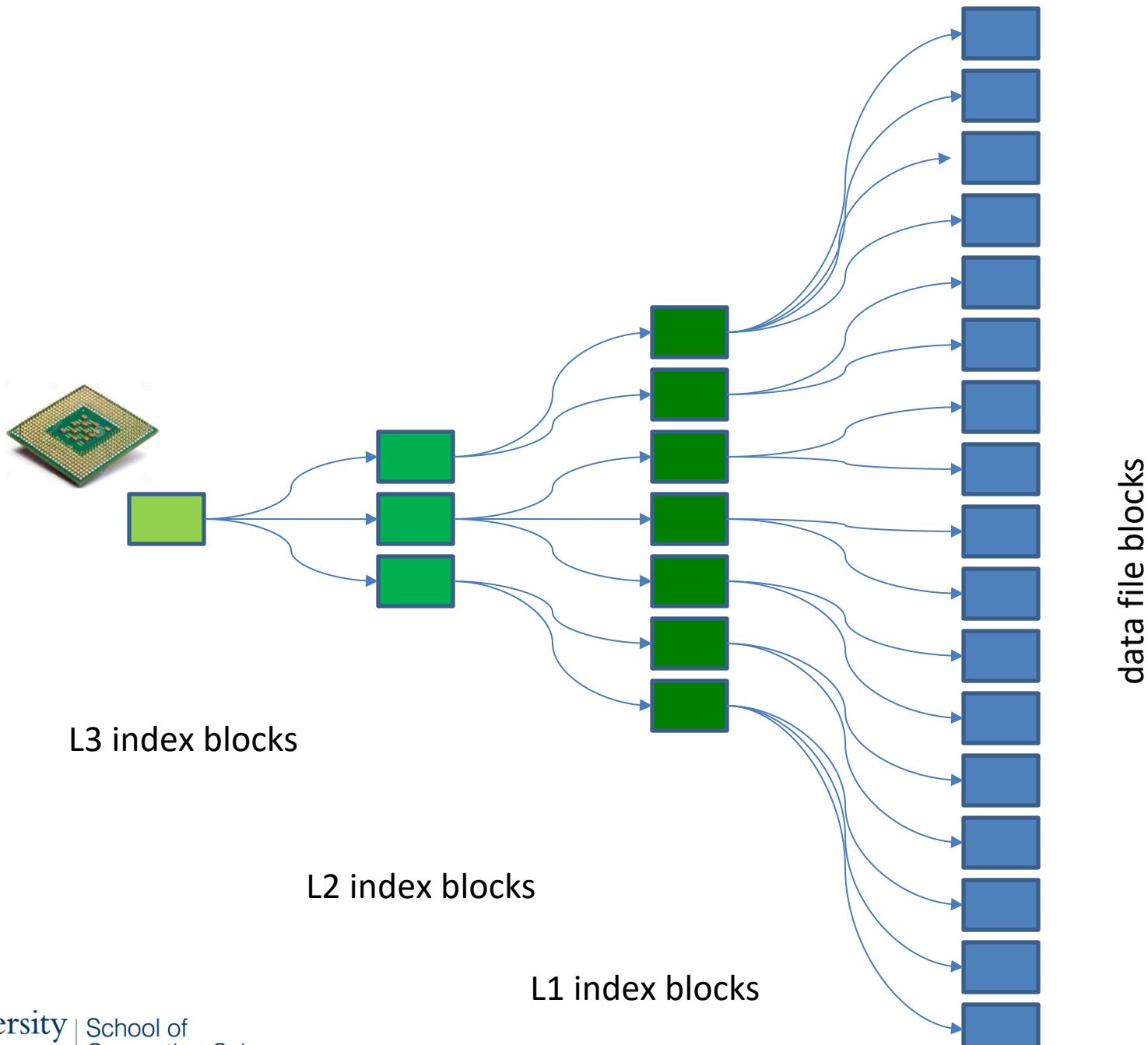
- **History:** Logarithms by John Napier (1550-1617) here in Scotland (Uni St Andrews)
- **Idea:** A logarithm with base  $m > 2$  splits the search space into  $m$  sub-spaces
- **Target:** searching steps:  $t = \log_m(b) < \dots < \log_2(b)$  iff  $m > 2$  (logarithm with base  $m$ )
- Theorem: Given a Level-1 Index with  $m$  being its blocking factor, then we need a multi-level index of Level  $t = \log_m(b)$ 
  - Proof: soon...
- Note:  $m$  is also known as the fan-out



# ... and now the proof

- Block size  $B$  bytes; File with  $r$  records; data-record has size  $s$ .
  - Blocking factor for the data-file is  $f = \text{floor}(B/s)$  records/block
  - Data file is  $b = \text{ceil}(r / f)$  data-blocks.
- Level-1 Primary index: each index entry points to a file-block (anchor)
  - Let  $I$  be the size of the index entry
  - The Level-1 index has  $b$  entries, with blocking factor  $m = \text{floor}(B/I)$
  - The Level-1 index has  $b_1 = \text{ceil}(b / m)$  L1-index-blocks
- Level-2 Primary index: each index entry points to each index-block of Level-1
  - The Level-2 index has  $b_1$  entries, with blocking factor  $m_2 = \text{floor}(B/I)$
  - The Level-2 index has  $b_2 = \text{ceil}(b_1 / m) = \text{ceil}(b / m^2)$  L2-index-blocks ( $1/m^2$  less blocks)
- ...
- Level- $t$  primary index:
  - The  $t^{\text{th}}$  top level will have only 1 block thus  $1 \leq (b / m^t)$  or  $t = \log_m(b)$
  - Split the search space into  $m$  sub-spaces  $\approx t$  steps to find the desired block

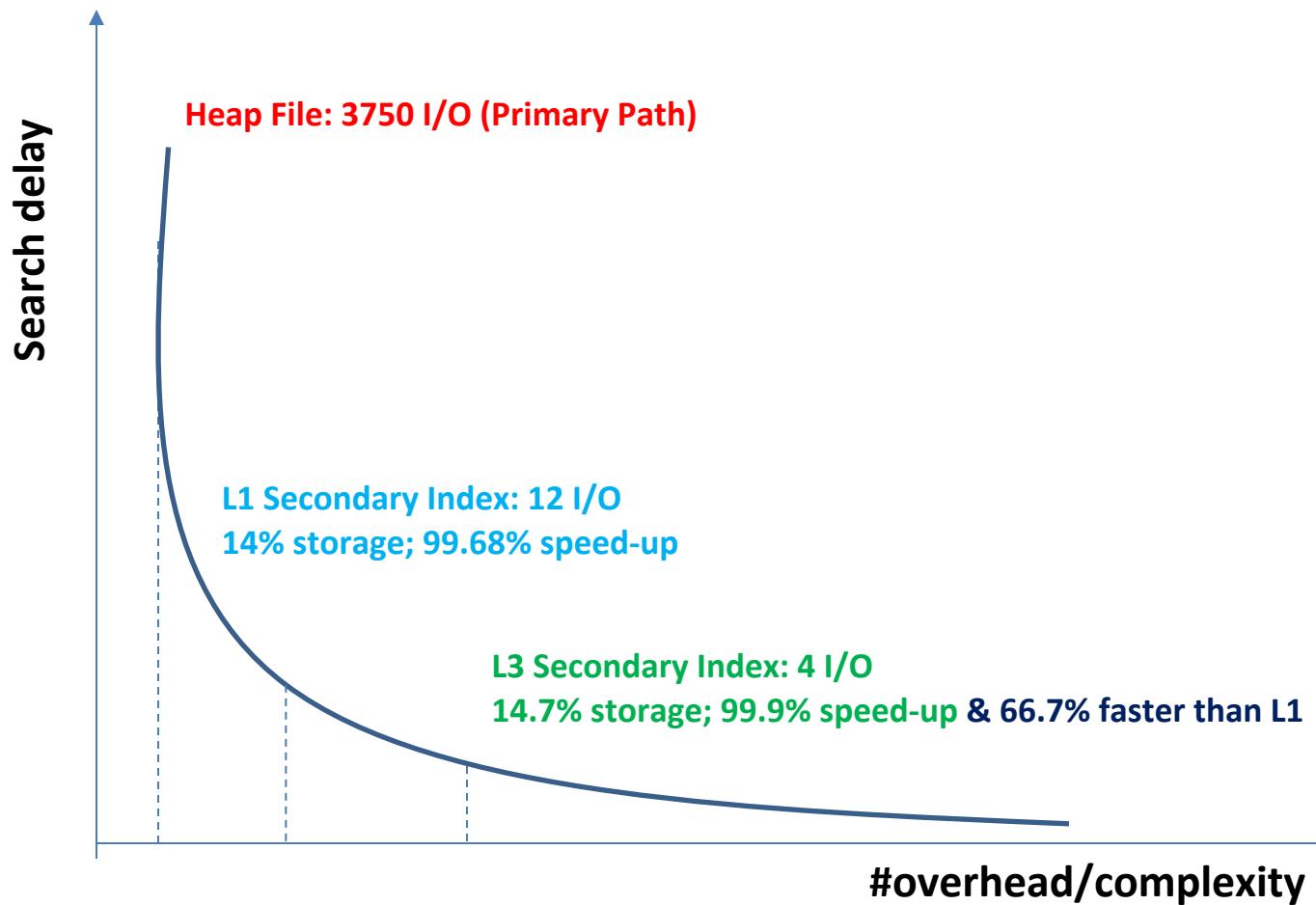




# In-Class Example

- Amazing Gain: one block per level plus one data-block :=  $t+1$  block accesses
- Build a *non-ordering/key secondary multi-level index* over a file:  $b = 7,500$  blocks and  $r = 300,000$  records of size  $R = 100$  bytes each; block size  $B = 4,096$  bytes; ID is  $V = 9$  bytes, pointer  $P = 6$  bytes
- Compute cost of `SELECT * FROM EMPLOYEE WHERE ID = 'x'`
  - Fan-out (index-blocking factor)  $m = 273$  index entries/block
  - Level-1: index with  $b_1 = 1,099$  index-blocks
  - Level-2: index entries: 1,099 thus, number of blocks  $b_2 = \text{ceil}(b_1/m) = 5$  blocks
  - Level-3: index entries: 5 thus, number of blocks  $b_3 = \text{ceil}(b_2/m) = 1$  block
- Structure := L1: Secondary Index (dense); L2 & L3: Primary Indexes (sparse)
- 3-level index search cost:  $3 + 1 = 4$  block accesses only!
- Level-1 Secondary Index  $\text{ceil}(\log_2(1099)) + 1 = 12$  blocks accesses
- No Index: 3750 blocks accesses

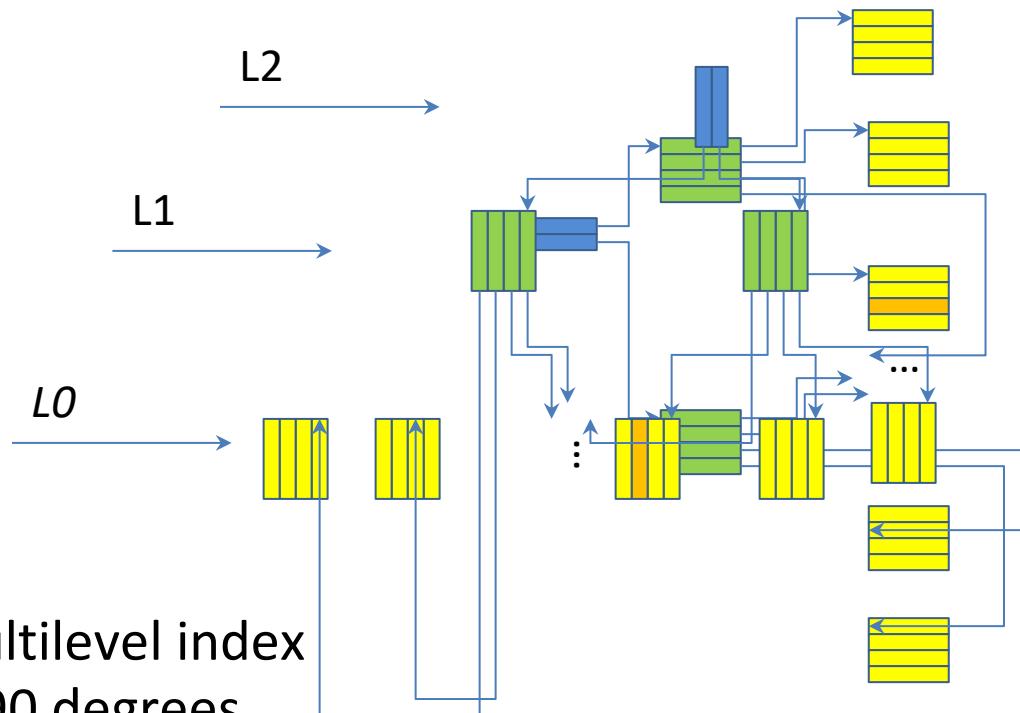
# Trade off: Overhead vs Speed



# Dynamic Multi-Level Indexes

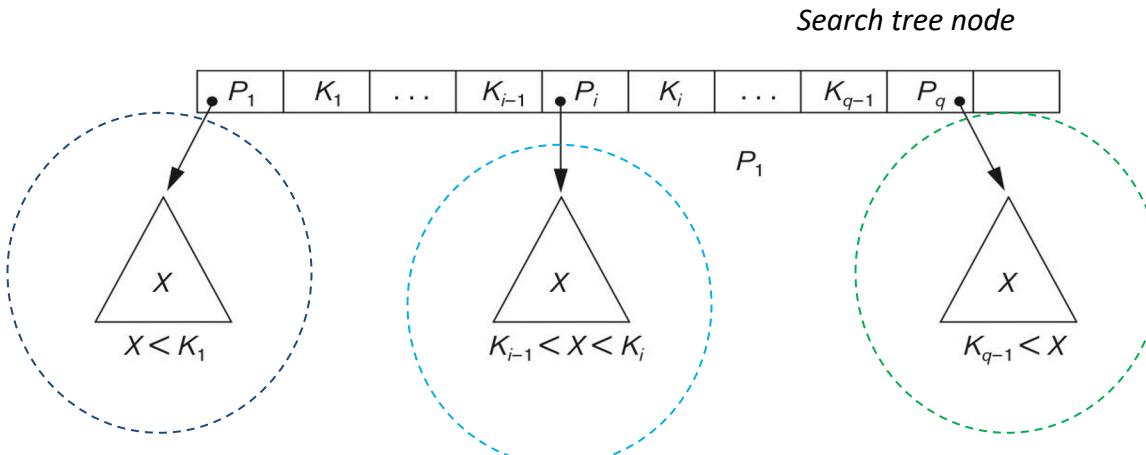
- **Summary so far:** Search for a tuple over a  $t$ -level index:  **$t+1$  block accesses**
- **Challenge:** Insertions, deletions, and updates are costly!
  - **Why?**
  - **Because:** *all* encapsulated indexes are *physically* ordered files → *all* updates should be reflected to *all* levels
- Demand for ***dynamic*** multi-level indexes
  - I.e., indexes that expand and shrink with insertions/deletions of records

# Search Tree



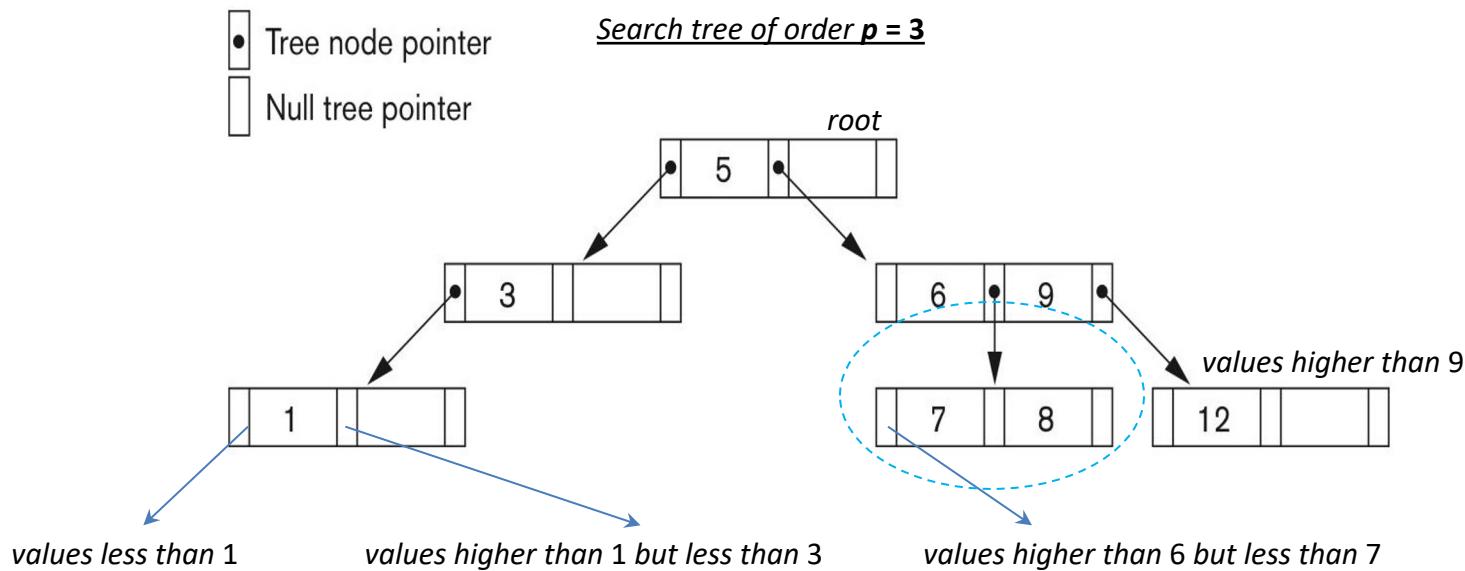
- Observe: 2-level multilevel index
- Turn: clockwise by 90 degrees
- Result: A Tree structure
  - Root is the L2-Index
  - Root's children are blocks of the L1-Index
  - Leaves are actual file data blocks (L0)

# Search Tree over Non-ordering Key



- **Search Tree of order  $p$  (splitting factor) over a key field  $k$  (e.g., SSN)**
  - Each *node* has a set of  $q$  pairs:  $\{\text{pointer}, \text{key}\}$  where  $q \leq p$ :  
$$\text{Tree-Node} := \{(P_1, K_1), (P_2, K_2), \dots, (P_{q-1}, K_{q-1}), P_q\}$$
    - Pointer  $P_i$  either points to another tree node or is NULL
    - Key  $K_i$  is a *search key* value:  $K_1 < K_2 < K_i < \dots < K_{q-1}$
    - For all key values  $X$  in a *subtree* pointed to by  $P_i$ :
      - $K_{i-1} < X < K_i$ , for  $1 < i < q$
      - If  $i = 1$ ,  $X < K_1$  and if  $i = q$ ,  $K_{q-1} < X$

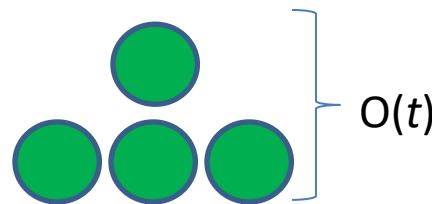
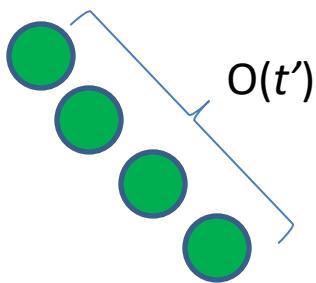
# Search Tree over Non-ordering Key



- **Search for key:** Follow the tree pointers to and within a tree node
- **Search:** 13; 8; 10; 0
  - There is no 13
  - There exists 8
  - There is no 10
  - There is no 0

# Search Tree: Limitations

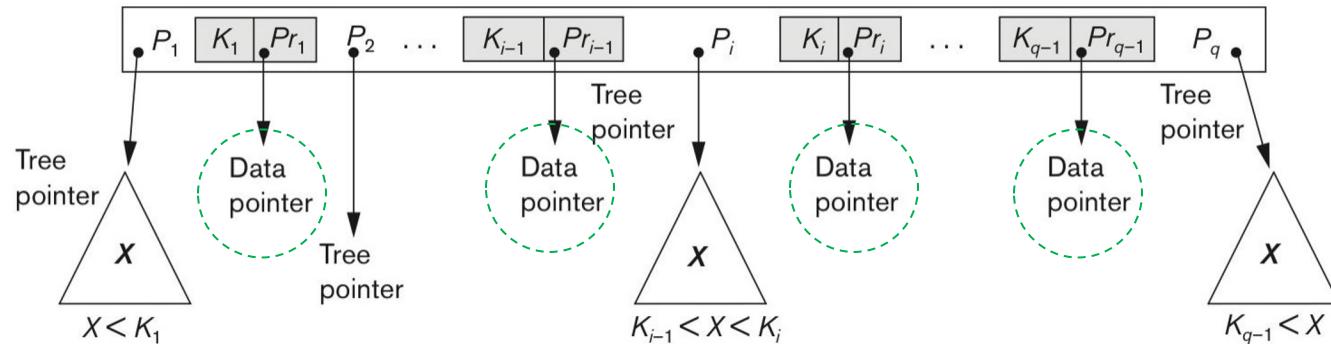
- It can become **unbalanced**
- It **does not adjust to the distribution of the keys**; i.e., leaf-nodes are at different levels...
  - **Worst case:** a **linked-list of nodes** instead of a tree structure
  - Larger tree depth  $t$  results to higher expected search time  $O(t)$



- Challenge 1: remove **redundant** levels, i.e., estimate the **best** level  $t$
- Challenge 2: ensure **balanced tree** by **minimizing** the tree depth  $t$ 
  - Challenge 2.1: what happens if key values are inserted in a **full** node? (split)
  - Challenge 2.2: what happens if the **last** key value in a node is **deleted**? (merge)
- The maintenance algorithms become very complex → Overhead
- **Merging & Splitting complexity:**  $O(p^t)$

# B-Tree over Non-ordering Key

- **B Tree of Order p:**
  - **B-Tree-Node** :=  $\{P_1, (K_1, Pr_1), P_2, (K_2, Pr_2) \dots, P_{q-1}, (K_{q-1}, Pr_{q-1}), P_q\}$ , where  $q \leq p$

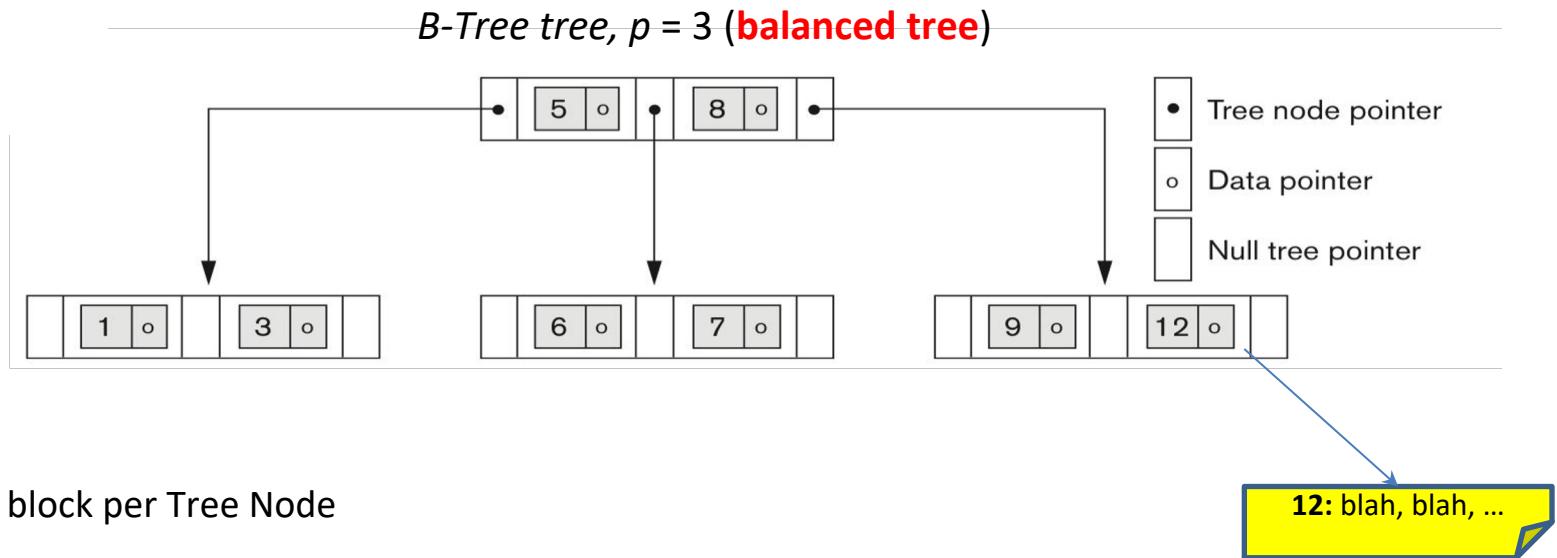


- $Pr_i$  is a data-pointer to the data-block holding the value  $K_i$
- Key order:  $K_1 < K_2 < K_i < \dots < K_{q-1}$
- For all key values  $X$  in a subtree pointed to by tree-pointer  $P_i$ :
  - $K_{i-1} < X < K_i$ , for  $1 < i < q$
  - If  $i = 1$ ,  $X < K_1$  and if  $i = q$ ,  $K_{q-1} < X$
- Each node with **q tree-pointers** has at most **q-1 values & q-1 data-pointers**

# B-Tree

**Search:** Traversing the tree nodes until finding the record

**Rationale:** Immediate access to the block of the searching key!



**Search 8:** 1 index block access, then access the data block *immediately* → 2 block accesses

**Search 5:** 1 index block access, plus data block access → 2 block accesses

**Search 7:** 2 index block accesses, plus data block access → 3 block accesses

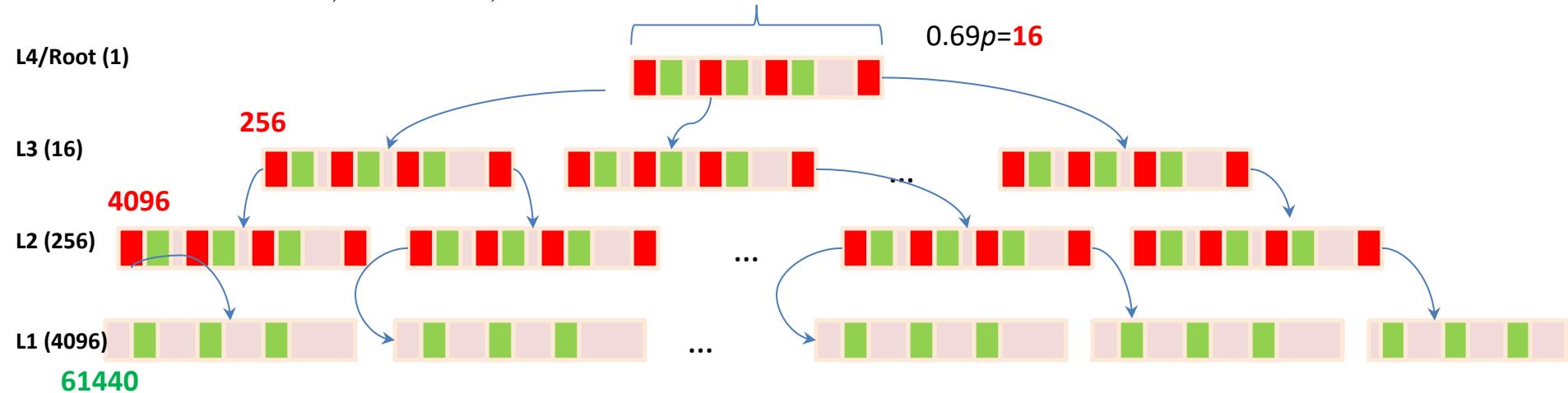
**Search 12:** 2 index block accesses, plus data block access → 3 block accesses

**Search 31:** 2 index block accesses; no data block access needed → 2 block accesses

- Task 1: Create a **3-level B-Tree index** of order  $p = 23$  over a non-ordering / key field
- Assume each B-Tree node is 69% full of information (pointers/keys)
  - On average, each B-Tree node accommodates  $0.69p = 16$  tree pointers/ 15 key values
  - Average fan-out = 16 per tree node, i.e., split the tree space into 16 sub-trees

$p=23$ : 这表示B树的阶数是23。在B树的上下文中，阶数定义了每个节点最多可以有多少个子节点。具体来说，一个阶数为p的B树意味着每个节点最多有p个子节点。

$\$rho$ : 这里的 $\$rho$ 表示实际填充因子，也就是说，节点中实际存储的指针和键值占总容量的百分比。在这个例子中，每个节点被填充到其容量的69%。因此，如果一个节点可以有最多23个子节点（由阶数p决定），那么实际上每个节点平均有16个子节点（0.69乘以23等于大约16）。这也意味着每个节点有15个键值，因为在B树中，一个节点的键值总数总是比子节点的数量少一个。



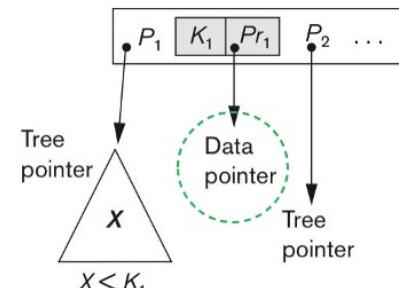
- Level-4/Root: 1 node with 15 keys/data-pointers & 16 pointers to tree nodes
- Level-3: 16 nodes with  $16 \times 15 = 240$  keys/data-pointers;  $16 \times 16 = 256$  pointers to nodes
- Level-2: 256 nodes with  $256 \times 15 = 3840$  keys/data-pointers;  $256 \times 16 = 4096$  pointers to nodes
- Level-1: 4096 nodes with  $4096 \times 15 = 61440$  keys/data-pointers; and null node pointers (leaves)

# In-class Example

- How many keys are stored in this tree?
  - Structure:
    - Level-4/Root: 1 node with 15 keys/data-pointers; 16 pointers to tree nodes;
    - Level-3: 16 nodes with  $16 \times 15 = 240$  keys/data-pointers;  $16 \times 16 = 256$  pointers to nodes;
    - Level-2: 256 nodes with  $256 \times 15 = 3,840$  keys/data-pointers;  $256 \times 16 = 4096$  pointers to nodes;
    - Level-1: 4096 nodes with  $4096 \times 15 = 61,440$  keys/data-pointers; null pointers (leaves);
  - There are:  $61,440 + 3,840 + 240 + 15 = 65,535$  key entries pointing to data blocks
- What if our file has 65,536 (65,535+1) keys to be indexed?
- What if our file has more than 279,840 keys to be indexed?
  - $= 22 + 23 \times 22 + 23 \times 23 \times 22 + 23 \times 23 \times 23 \times 22$

# In-class Example

- Assume block B = 512 bytes, data-pointer Pr = 7 bytes, tree-pointer P = 6 bytes, key V = 9 bytes
- How many bytes is the index?
- How many blocks is the index?
  - Storage (used for metadata):
    - Storage for data-pointers =  $65,535 * 7 = 458,745$  bytes (460KB)
    - Storage for key entries =  $65,535 * 9 = 589,815$  bytes (590KB)
    - Storage for tree-pointers =  $(4096 + 256 + 16) * 6 = 4368 * 6 = 26,208$  bytes (27KB)
    - **Total (metadata):** 1,074,768 bytes = **1.07MB index**
  - Storage (complete tree) :
    - Number of nodes:  $1 + 16 + 256 + 4096 = 4369$  nodes
    - Complete tree node size:  $23 * 6 + 22 * 7 + 22 * 9 = 490$  bytes
    - **Total:**  $4369 * 490 = 2,140,810$  bytes = **2.04MB**
  - Blocks:
    - Blocking factor:  $\text{floor}(512/490) = 1$  tree-node per block (normally)!
    - 1 node per block, thus **4,369 blocks!**
    - **Actual space on disk:**  $4,369 * 512 = 2236928$  bytes = **2.13MB**



# Maximize fan-out & minimize storage

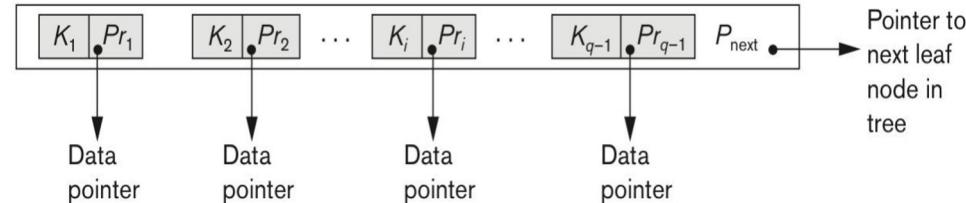
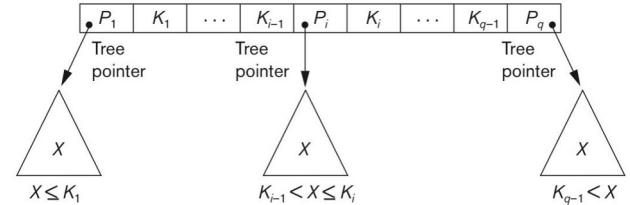
- Hmm, a B-Tree stores too much meta-data!
  - Data-pointers to data blocks (addresses, e.g., URI/L average size 647 bytes!)
  - Tree-pointers to tree nodes (structural meta-data)
  - Search key values (data values)
- Objective 1: Be more **storage efficient**... free up some space from the nodes. **Why?**
- Objective 2: **Maximize the fan-out** of a node. **Why?**
- Recall: The fan-out indicates the number of search-space splits (at every step)
  - Thought: The fan-out is represented through the number of tree-pointers per node
- Note:
  - Maximizing the number of tree-pointers per node maximises fan-out!
  - Data-pointers can be removed from the tree nodes!

# B+-Tree over Non-ordering Key

- Ta-dah: **B+ Tree** (give semantics to the nodes!)
  - Internal Nodes
  - Leaf Nodes
  - Principle: internal nodes have no data pointers to maximize fan-out
- Key ideas:
  1. Only Leaf Nodes hold the actual **data-pointers** thus removing them from the Internal Nodes
  2. Leaf Nodes hold all the **key values** and their corresponding **data-pointers**
  3. Some **key values** are replicated in the **Internal Nodes** to guide the search process
  4. Fusion/combination of **ISAM Search Tree** with **B Tree**

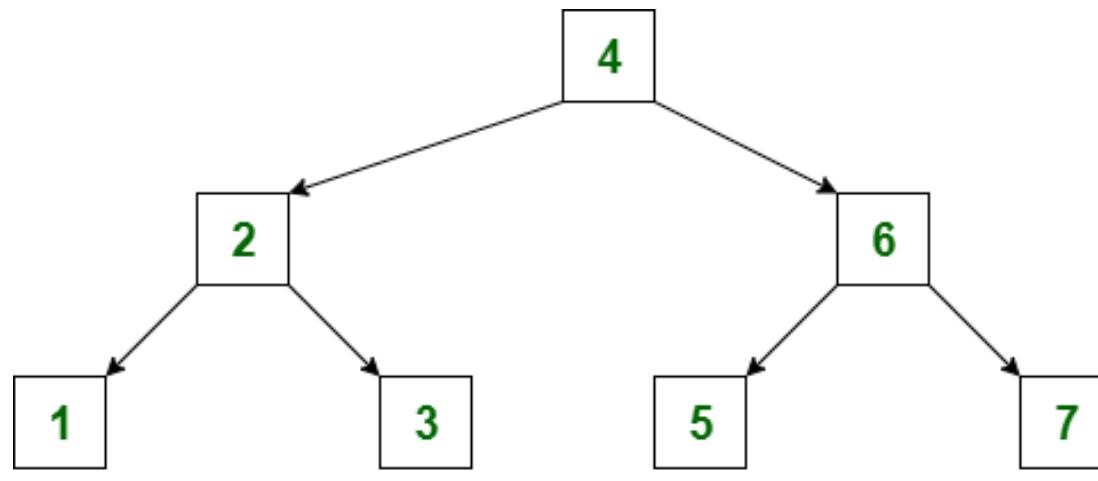
# B+-Tree Nodes

- **Assume B+ Tree of order p**
- **B+-Internal-Node** :=  $\{P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q\}$ ,  $q \leq p$ 
  - Order:  $K_1 < K_2 < \dots < K_{q-1}$
  - For all key values X in a subtree pointed to by tree pointer  $P_i$ :
    - $K_{i-1} < X \leq K_i$ , for  $1 < i < q$
    - If  $i = 1$ ,  $X \leq K_1$  and if  $i = q$ ,  $K_{q-1} < X$
  - Note: Each internal node with q tree-pointers has at most  $q-1$  key values
- **B+-Leaf-Node** :=  $\{(K_1, Pr_1), (K_2, Pr_2) \dots, (K_{q-1}, Pr_{q-1}), P_{next}\}$ ,  $q \leq p$ 
  - $Pr_i$  is a data-pointer to the actual block holding value  $K_i$
  - $P_{next}$  is a tree-pointer to the next leaf node (sibling) → Linked-list of leaf-tree nodes!
  - All leaf nodes are at the same level → tree is balanced



# B tree vs B+- tree

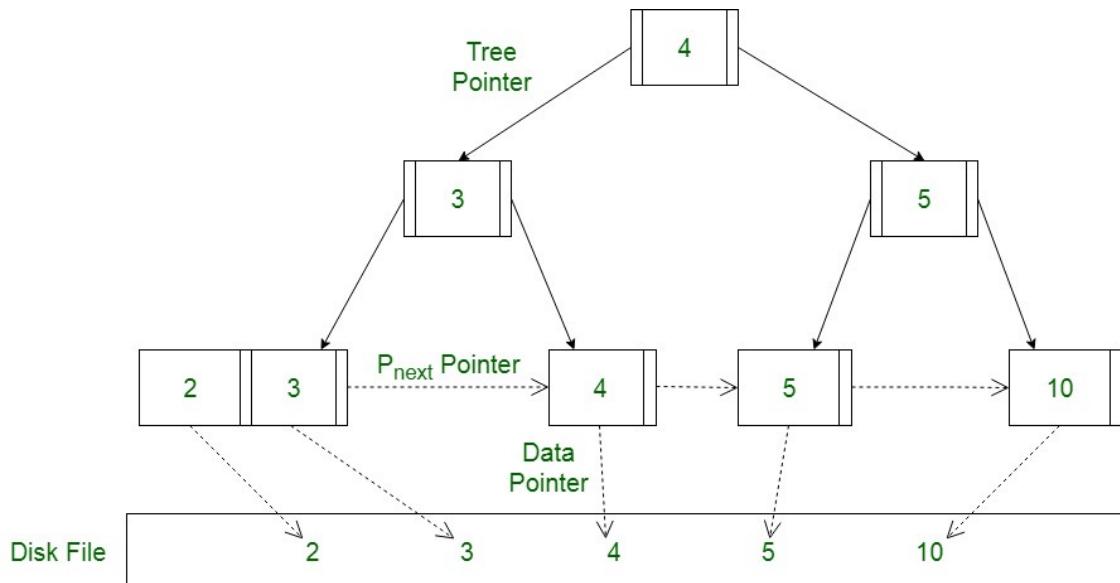
- B Tree:
  - All the leaf nodes of the B-tree must be at the same level.
  - Above the leaf nodes of the B-tree, there should be no empty subtrees.
  - B- tree's height should lie as low as possible.



**B-Tree**

# B tree vs B+- tree

- B+ Tree:
  - Only leaf nodes have data pointers
  - The leaf nodes are linked to provide ordered access to the records
  - Some of the key values of the leaf nodes also appear in the internal nodes

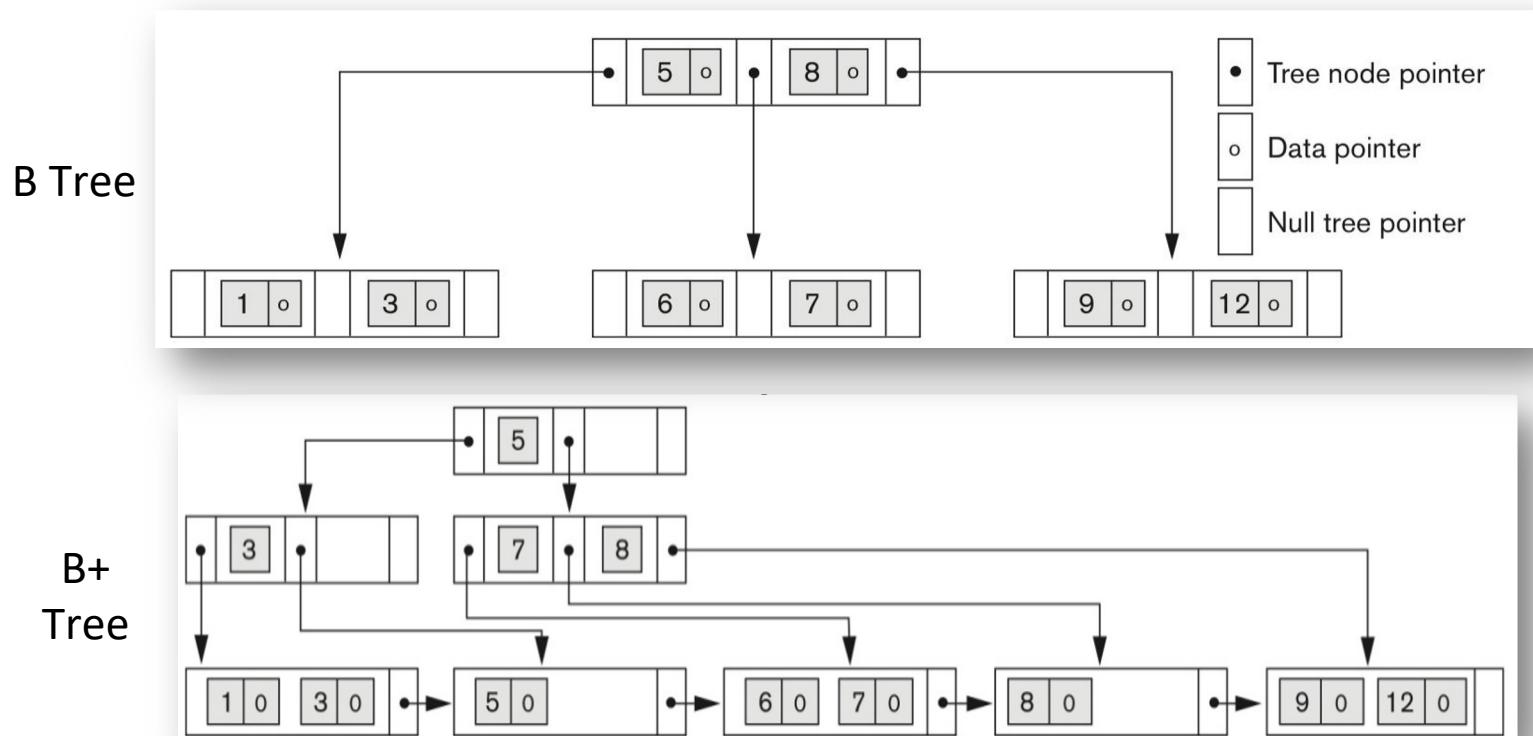


# B Tree vs B+-Tree

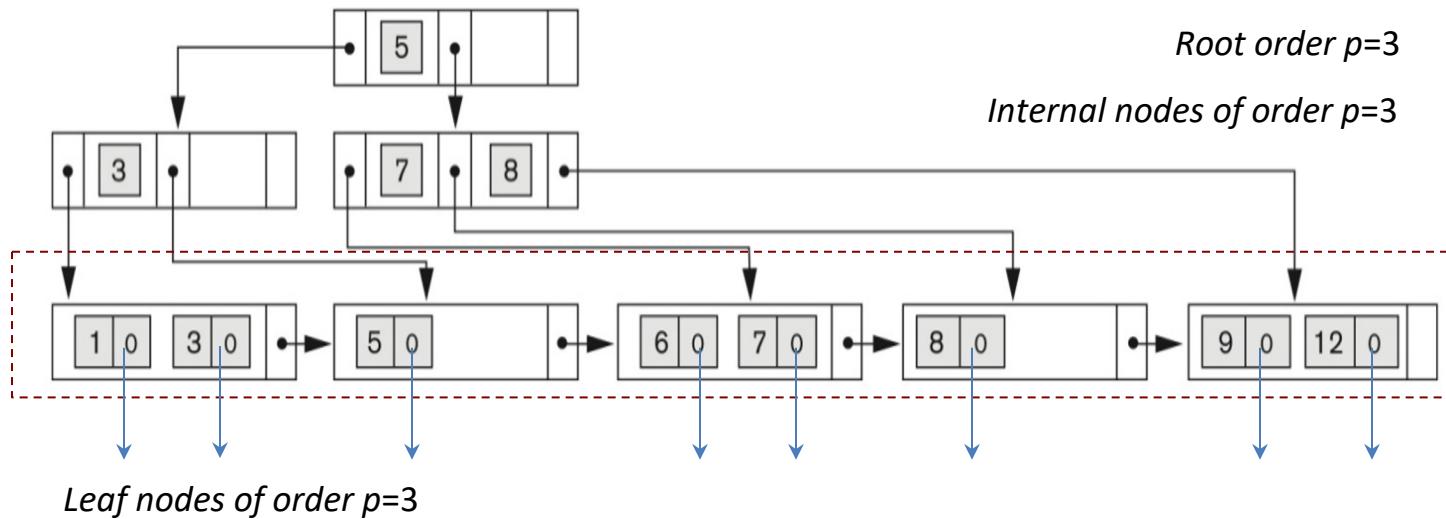
**Observation:** No data-pointers in the internal nodes of B+ Tree

**Observation:** values are distributed *all* over the B Tree while stored *only* in the B+ Tree *leaves*

**Target:** More space in the nodes, *thus*, more tree-pointers, *thus*, higher fan-out



# B+-Tree: Example

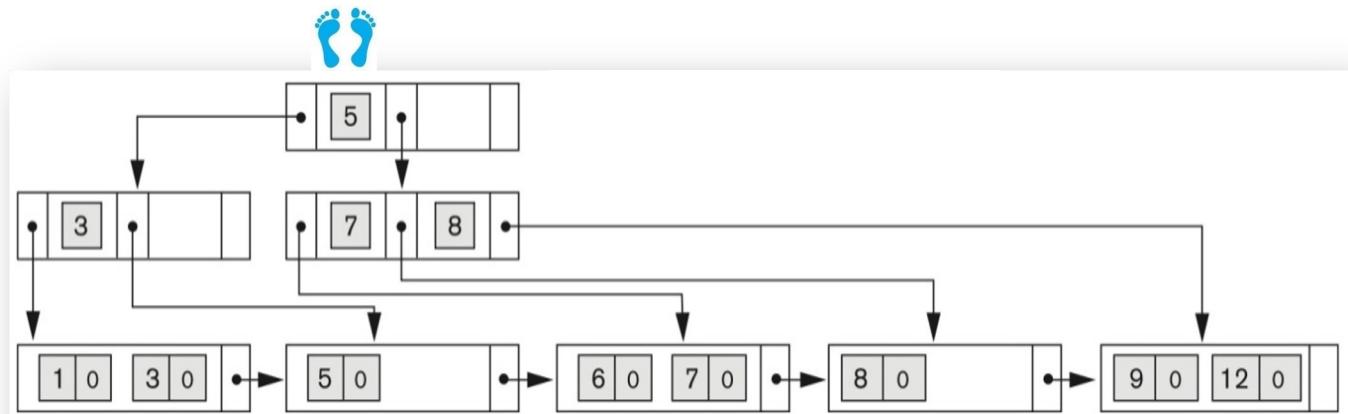
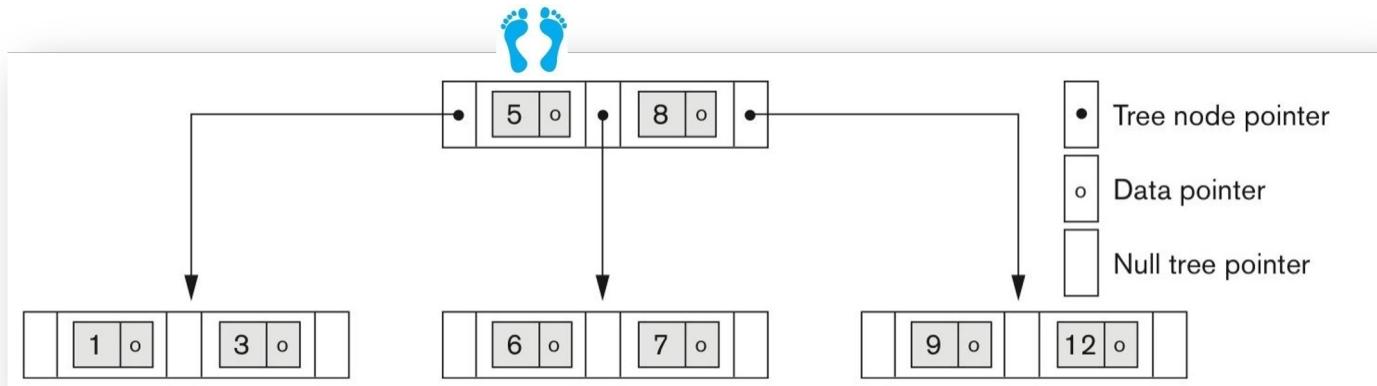


1. Leaf nodes are *linked & ordered* w.r.t. key (*it is good; why?*)
2. All key values appear at the Leaf nodes! (*it is good; why?*)
3. Leaf nodes contain data-pointers *only* (*it is good; why?*)
4. Leaf nodes are equi-balanced (*it is good; why?*)
5. *Some* selected keys are replicated in the internal nodes (*sparse & dense*)

# B & B+-Tree

Range Query:

```
SELECT * FROM R WHERE key > 3 AND key < 10
```

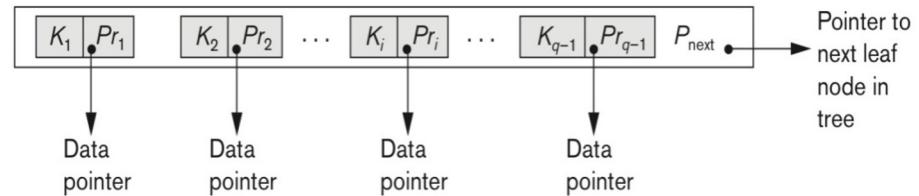
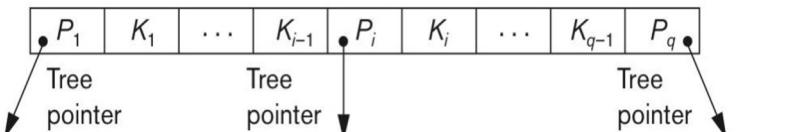


# B+-Tree & B-Tree Example

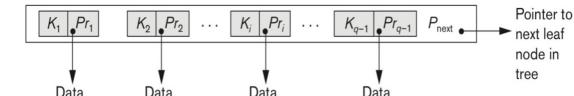
- Hypothesis: Removing the **data-pointers** from **internal nodes** results in higher fan-out thus **more** index-entries thus **quicker** search process
- Sounds plausible?
- Assume block size  $B = 512$  bytes, key size  $V = 9$  bytes, data-pointer size  $Pr = 7$  bytes, tree-pointer size  $P = 6$  bytes
- Calculate the order  $p$  of a B-Tree and a B+ Tree so that each node will fit in one block!
  - Recall: B-Tree nodes and internal B+ Tree nodes can have up to  $p$  tree pointers ( $p-1$  key values)
  - B+ Tree internal node:
    - Size:  $p \cdot P + (p-1) \cdot V$
    - To fit it into a block we have:  $p \cdot P + (p-1) \cdot V \leq B \rightarrow p \leq (B + V) / (P + V)$
    - The maximum order is  **$p = 34$**  (i.e., 34 tree pointers; 33 key values)
  - B+ Tree leaf node:
    - Size:  $pL$  data-pointers,  $pL$  key values, one next-tree pointer
    - To fit into a block we have:  $pL \cdot (Pr + V) + P \leq B \rightarrow pL \leq (B - P) / (Pr + V)$
    - Each leaf node can store up to  **$pL = 31$  pairs** of key-value/data-pointers
  - B Tree nodes:
    - Size:  $p-1$  data-pointers,  $p-1$  key values,  $p$  tree-pointers
    - To fit into a block we have:  $p \cdot P + (p-1) \cdot (V + Pr) \leq B \rightarrow p \leq (B + V + Pr) / (P + V + Pr)$
    - The maximum order for B Tree is  **$p = 24 < 34$**

# In-Class Activity

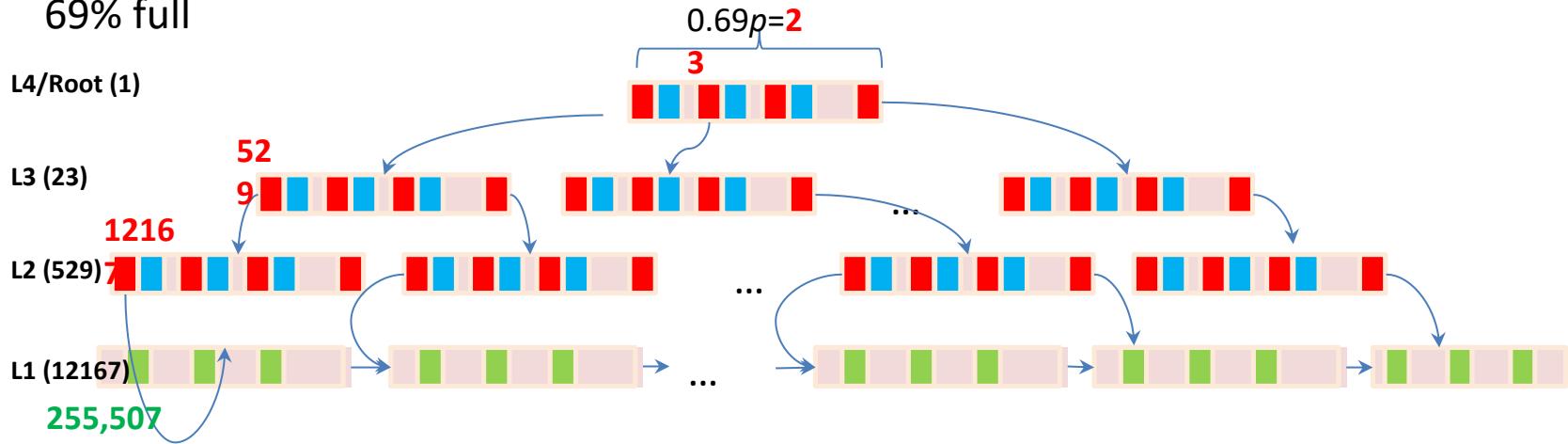
- Consider a **3-level B+ Tree** index (i.e., 3 internal levels and one leaf level) of order  $p = 34$  and  $pL = 31$
- Assume that each internal & leaf tree node is **69% full**
- Task 1:** Compare its storage capacity to that of a 3-level B Tree in terms of #keys (earlier example -- 65,535 keys)
- Task 2:** How many blocks can we store?



- Consider a 3-level B+ Tree index (i.e., 3 internal levels and one leaf level) of order  $p = 34$  and  $pL = 31$
- Assume that each internal & leaf tree node is 69% full
- Task 1:** Compare its storage capacity to that of a 3-level B Tree in terms of #keys (earlier example -- 65,535 keys)
- Task 2:** How many blocks can we store?



- 3-level B+ Tree index: 3 internal levels and one leaf level, order  $p = 34$  and  $pL = 31$ , 69% full



- Task 1:** Compare its storage capacity in terms of #keys to that of a 3-level B Tree (65,535 keys)
  - Internal tree-nodes have  $0.69p = 23$  tree-pointers and 22 keys (on average)
  - Leaf nodes have  $0.69pL = 21$  data-pointers (on average)
  - Level-4/Root:** 1 node with 22 keys and 23 pointers to tree nodes
  - Level-3:** 23 nodes with  $23 \times 22 = 506$  keys and  $23 \times 23 = 529$  pointers to nodes
  - Level-2:** 529 nodes with  $529 \times 22 = 11638$  keys and  $529 \times 23 = 12167$  pointers to leaves
  - Level-1/Leaf Level:** 12167 nodes with  $12167 \times 21 = 255,507$  keys/data-pointers

# In-Class Activity

- **Task 2:** How many blocks do we store?
  - Level-4/Root: 1 node with 22 keys and 23 pointers to tree nodes
  - Level-3: 23 nodes with  $23 \times 22 = 506$  keys and  $23 \times 23 = 529$  pointers to nodes
  - Level-2: 529 nodes with  $529 \times 22 = 11,638$  keys and  $529 \times 23 = 12,167$  pointers to leaves
  - Level-1/Leaf: 12,167 nodes with  $12167 \times 21 = 255,507$  keys/data-pointers
  - Number of nodes:  $1 + 23 + 529 + 12167 = 12720$  nodes > **12,720 blocks!**
  - Compare:
    - B Tree (3-Level) stores 65,535 keys/data-pointers (at a fill factor of 69%)
    - B+ Tree (3-Level) achieves 290% more capacity over the same file and same fill factor!
    - And, expected search cost is the same! **4 + 1 = 5 block accesses**
    - And, if the file needs to store more than 65,535 records (e.g., 65,536 records) then we may need to define one more level for the B Tree; B+ Tree level does **not** change!

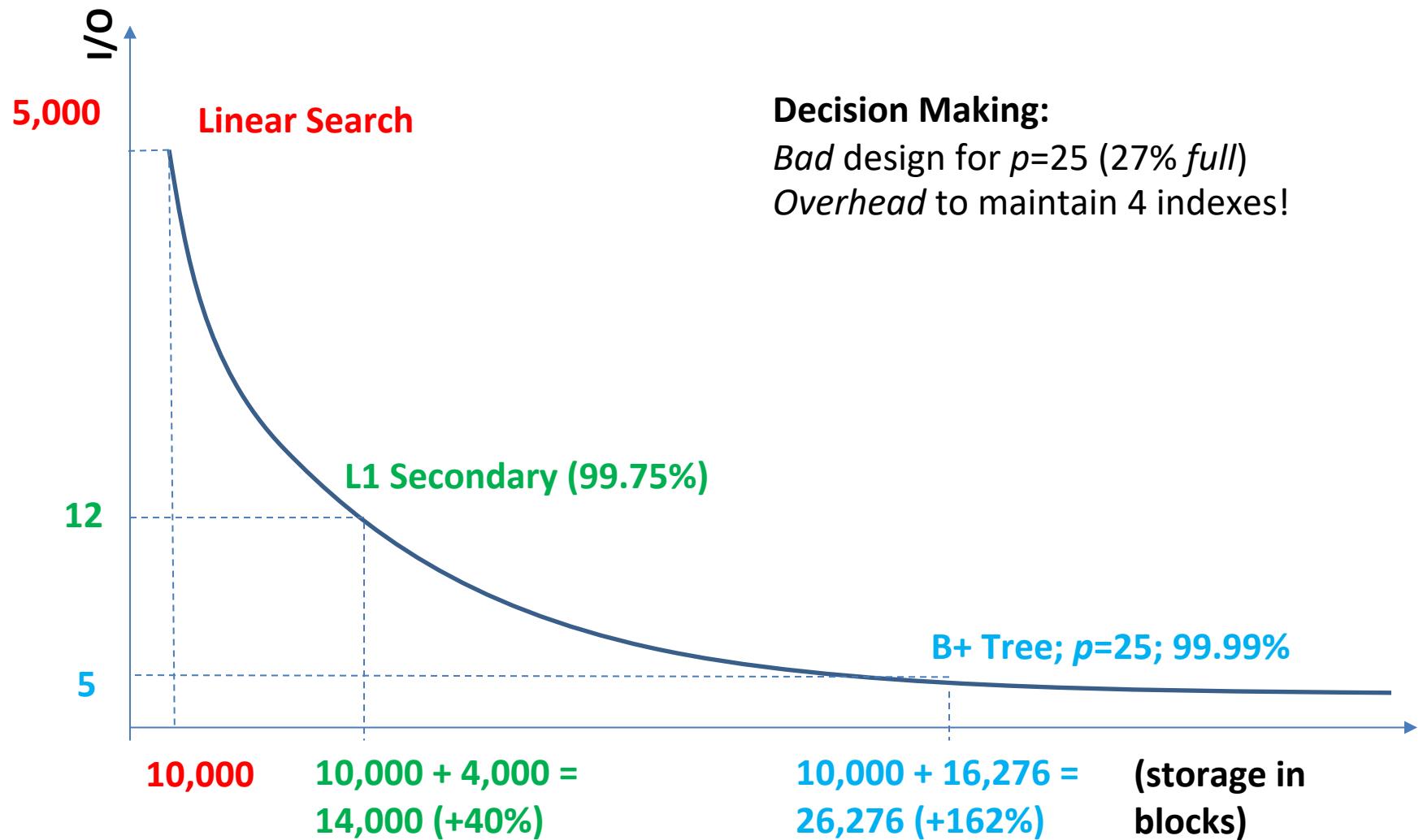
# Putting it all together

- Assume:
  - Block size  $B = 512$  bytes
  - File  $r = 100,000$  records
  - File blocking factor:  $bfr = 10$  records/block
  - Any pointer = 10 bytes
  - Indexing key = 10 bytes
- Consider:
  - L1 Secondary Index on non-ordering key; index blocking factor =  $m$
  - B+ Tree Index of order  $p = m$ ; 100% full all nodes
- Project onto the Speed-vs-Overhead trade-off plane the performance of:
  - Expected cost of Linear Search on File vs File Storage
  - Expected cost of search on L1 Secondary Index vs Index Storage
  - Expected cost of search on B+ Tree Index vs B+ Tree Storage

# Putting it all together

- Block size B = 512 bytes; File r = 100,000 records; blocking factor: bfr = 10 records/block; pointers = 10 bytes; Indexing key = 10 bytes
- File: b = 10,000 blocks → Linear Search on File b/2 = **5,000 block accesses**
- L1 Secondary Index:
  - L1 index-entry =  $10 + 10 = 20$  bytes
  - L1 blocking factor (fan-out) m =  $\text{floor}(512/20) = 25$  entries/block
  - L1 blocks =  $\text{ceil}(r/m) = 4,000$  blocks
  - Binary Search on L1 Index:  $\log_2(4000) = \mathbf{12}$  block accesses
- B+ Index of order p = m = 25 (25 pointers; thus 24 keys)
  - Internal Node size:  $p*10 + (p-1)*10 = 490$  bytes; 1 block per Internal Node
  - Leaf Node size:  $(p-1)*(10 + 10) + 1*10 = 490$  bytes; 1 block per Leaf Node
  - B+ Root: 1 node with 24 keys and 25 pointers to tree nodes;
  - B+ L1: 25 nodes with  $25*24 = 600$  keys and  $25*25 = 625$  pointers to nodes;
  - B+ L2: 625 nodes with  $625*24 = 15000$  keys;  $625*25 = 15,625$  pointers to leaves;
  - B+ Leaf: 15625 nodes with  $15625*24 = 375,000$  keys/data-pointers
  - Reasoning: Only 27% ( $100,000/375,000$ ) of the leaf nodes are occupied! → Redundancy!
  - Total:  $15625 + 625 + 25 + 1 = \mathbf{16,276 nodes/blocks!}$  (file is only 10,000 blocks)
  - B+ Tree search:  $1 + 1 + 1 + 1 + 1 = \mathbf{5}$  block accesses

# Trade off: Overhead vs Speed

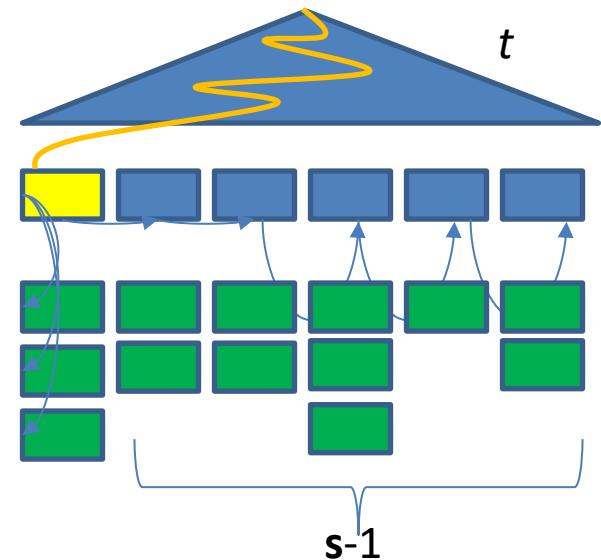


# Putting it all together

- Consider B+ Tree as Secondary Index of Level  $t$ ,  $t > 1$ , over non-ordering key SSN (**Range query**)
  - `SELECT COUNT(SSN) FROM EMPLOYEE WHERE Salary > £1000`
- Context:
  - Salary is a non-unique attribute in the EMPLOYEE
  - B+ Tree occupies  $s$  blocks for all the leaf-nodes of order  $q$ ,  $q-1$  data-pointers/keys + 1 tree-pointer
  - All leaf nodes are 100% full
  - File has  $b$  blocks
- Tasks:
  1. Compute the expected cost in block accesses
  2. Compute the maximum level  $t$  as a function of  $b$ ,  $s$ ,  $q$ ?
  3. Compute the maximum leaf-node order  $q$ ?

# Putting it all together

- Task 1: Methodology
  - 1. Follow the leftmost pointers to get to the leftmost leaf node
  - 2. There, load the data block for each SSN (visit the data-pointer)
  - 3. Count only the SSN values after checking in-memory: ‘Salary > £1000’
  - 4. Follow the sibling leaf node (tree-next pointer)
  - 5. Repeat the same, i.e., GO TO 2
  - 6. Stop at the end, i.e., when tree-next pointer is NULL
  - Sum it up:
    - **t block accesses** to reach the leftmost leaf-node
    - **q-1 block accesses** for each leaf node to load data-blocks ('Salary > £ 1000')
    - Access the next **s-1 blocks**, i.e., loading **s-1 sibling leaf nodes**
  - **Result:  $t + (q-1) + (s-1) + (s-1)(q-1) = t + sq$  block accesses**
- Task 2: Which is the maximum level  $t$  as a function of  $b$ ,  $s$ ,  $q$ 
  - **B+ Tree Total:**  $t + (q-1) + (s-1) + (s-1)(q-1) = t + sq$  block accesses
  - **Naïve solution:** just scan the whole file  $b$  block accesses
  - **Benefit:**  $t + sq < b \rightarrow t < b - sq \rightarrow t_{MAX} = b - sq$
  - **Result:** IF B+ Tree Level >  $t_{MAX}$  THEN Linear Scan ELSE Use B+ Tree
- Task 3: Which is the maximum leaf-node order  $q$ 
  - Since  $t_{MAX} > 1$  then  $b - sq > 1$  or  $q_{MAX} = (b-1)/s$
  - **Result:** The blocking factor at the Leaf Level is bounded!



# Special Thanks

Special Thanks to Dr Nikos Ntarmos who is the original author of the slides.