Database Systems

Session 8 – Additional Topic

Physical Database Design and Query Planning and Execution Summarized

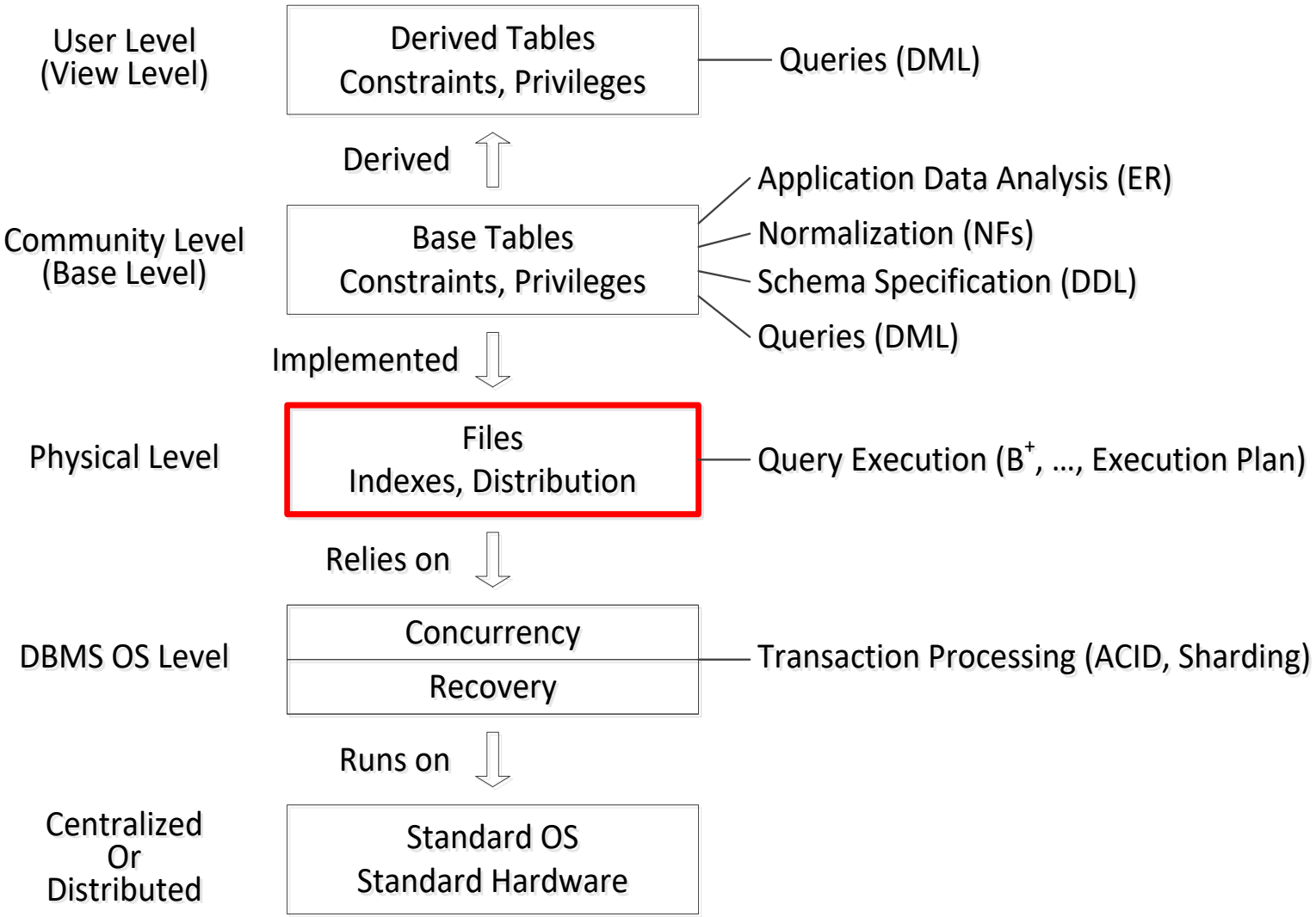
Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

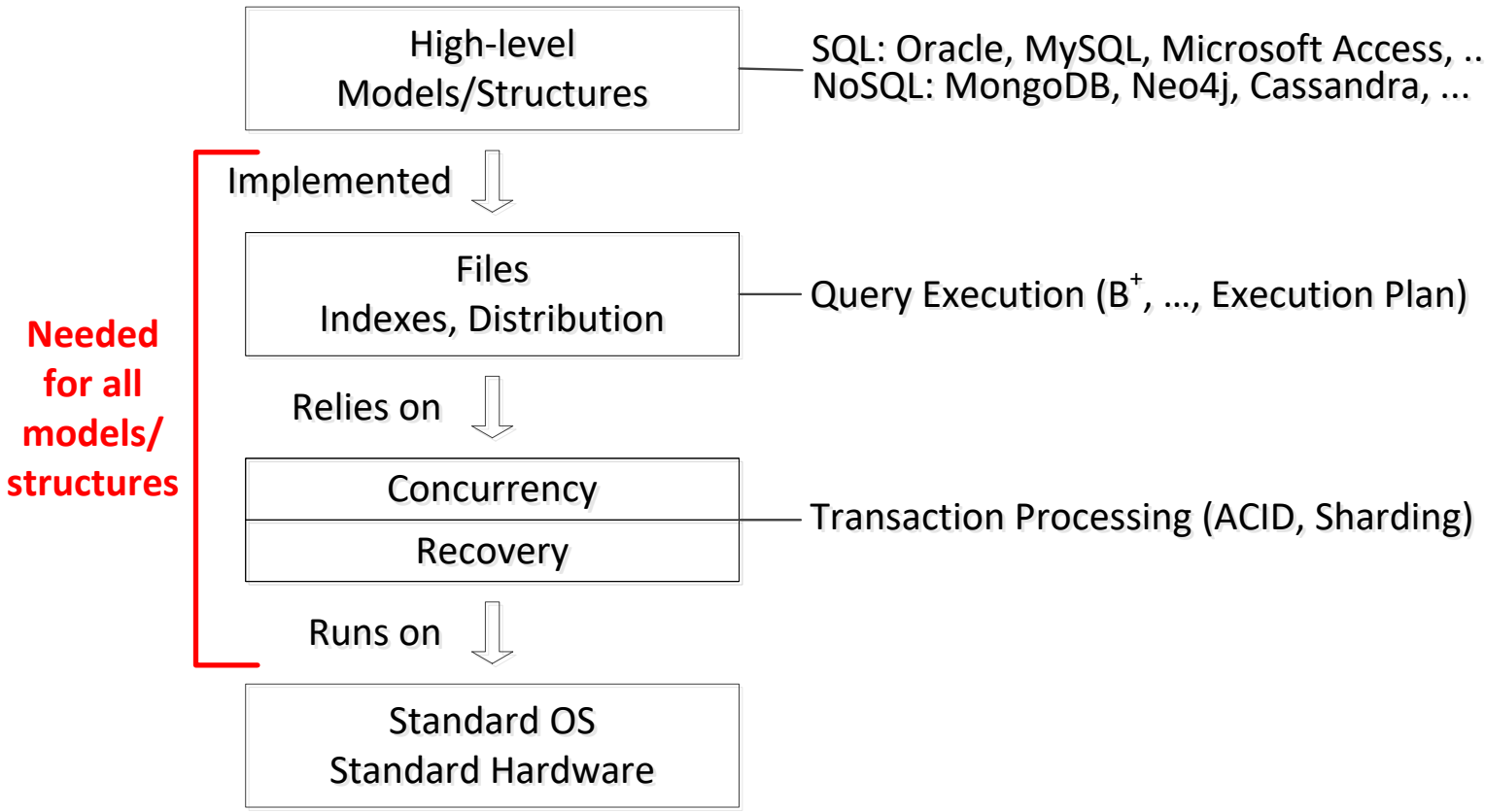
Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

Physical Design and Query Execution in Context



Lower Levels Needed (Not Only For Relational DBMSs)



An Important Note: Wider Applicability

- The material in this unit is applicable to Database Management Systems beyond just Relational Database Management Systems
- The semantics of what relations/tables are is not understood at this level
- The meaning of the fields of the records is not understood at this level
- We do not need to assume that the DBMS is a relational DBMS, but it may be convenient for examples

Database Design Context

- Logical DB Design (what we have done so far):
  1. Create first a model of the enterprise (e.g., using ER)
  2. Create a logical “implementation” (using a relational model and normalization)
  - » Creates the top two layers: “User” and “Community”
  - » Independent of any physical implementation
  - » It is frequently called (regrettably) physical design when specifying the design for a target DBMS
- Physical DB Design
  - » Uses a file system to store the relations
  - » Requires some knowledge of hardware and operating system’s characteristics
  - » Addresses questions of distribution (distributed databases), if applicable
  - » Creates the third layer
- Query execution planning and optimization ties the three layers together; not the focus of this unit, but some issues discussed

Icons / Metaphors

Information

Common Realization

Knowledge/Competency Pattern

Governance

Alignment

Solution Approach

Agenda

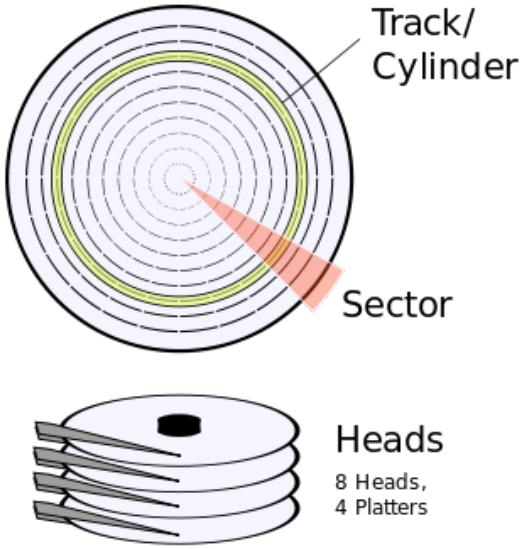
1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

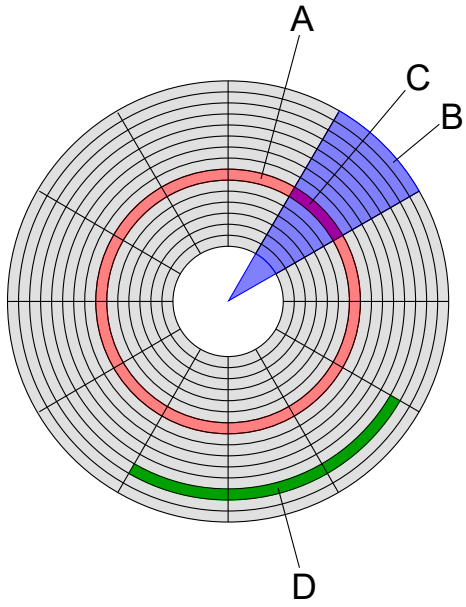
Issues Addressed in Physical Design

- ***The new aspect: the database in general does not fit in RAM, so need to manage carefully access to spinning or (more commonly) solid state disks.***
- Main issues addressed generally in physical design
  - Storage Media
  - File structures
  - Indexes
- We focus on
  - Centralized (not distributed) databases
  - Database stored on a disk using a "standard" file system, not one "tailored" to the database
  - Database using an unmodified general-purpose operating system
  - Indexes
- The only criterion we consider in this unit is ***performance***

What is a Spinning Disk?

- Illustrations from Wikipedia, in public domain
- A disk is a “stack” of platters
- A cylinder is a stack of tracks denoted by letter “A” on our one platter, in which it is only a ring
- Our typical block (next slide) is the red segment denoted by letter “C”





What is a Spinning Disk? (continued)

- Disk consists of a sequence of ***cylinders***
- A cylinder consists of a sequence of ***tracks***
- A track consist of a sequence of ***sectors***
- A sector contains
  - » Data area
  - » Header
  - » Error-correcting information (e.g., how many questions do you need to find a number between 1 and 2000 if person can lie once)
- We only care about the data area and will refer to it as a ***block***
- For us: ***A disk consists of a sequence of blocks***
- All blocks are of same size, say 512 B (bytes) of 4096 B (bytes)
- 512 bytes is rather small, but it is a very long-standing standard, and many legacy programs rely on this size
- We assume: a virtual memory page is the same size a block

What is a Spinning Disk? (continued)

- A physical unit of access is always a block.
- If an application wants to read one or more bits that are in a single block
  - If an up-to-date copy of the block is in RAM already (as a page in virtual memory), read from the page
  - If not, the system reads a whole block and puts it as a whole page in a disk cache in RAM

What is Different about a Solid State Disk?

- Logical access is similar to spinning disk.
- However there is no penalty to accessing blocks that are far away from one another.
- Access is far faster than spinning disks (factor of 10)
- Not good for archival storage.
- In older technologies, writing is much slower than reading.

What is a File?

- File can be thought of as a “logical” or a “physical” entity

- File as a logical entity: a sequence of records
- Records are either fixed size or variable size

- A file as a physical entity: a sequence of fixed size blocks (on the disk), but not necessarily physically contiguous (the blocks could be dispersed)
- Blocks in a file are either physically contiguous or not, but the following is generally simple to do (for the file system):
  - Find the first block
  - Find the last block
  - Find the next block
  - Find the previous block

What is a File? (continued)

- Records are stored in blocks
  - This gives the mapping between a “logical” file and a “physical” file
- Assumptions (some to simplify presentation for now)
  - Fixed size records
  - No record spans more than one block
  - There are, in general, several records in a block
  - There is some “left over” space in a block as needed later (e.g., for chaining the blocks of the file)
- We assume that each relation is stored as a file
- Each tuple is a record in the file

Example: Storing a Relation (Logical File)

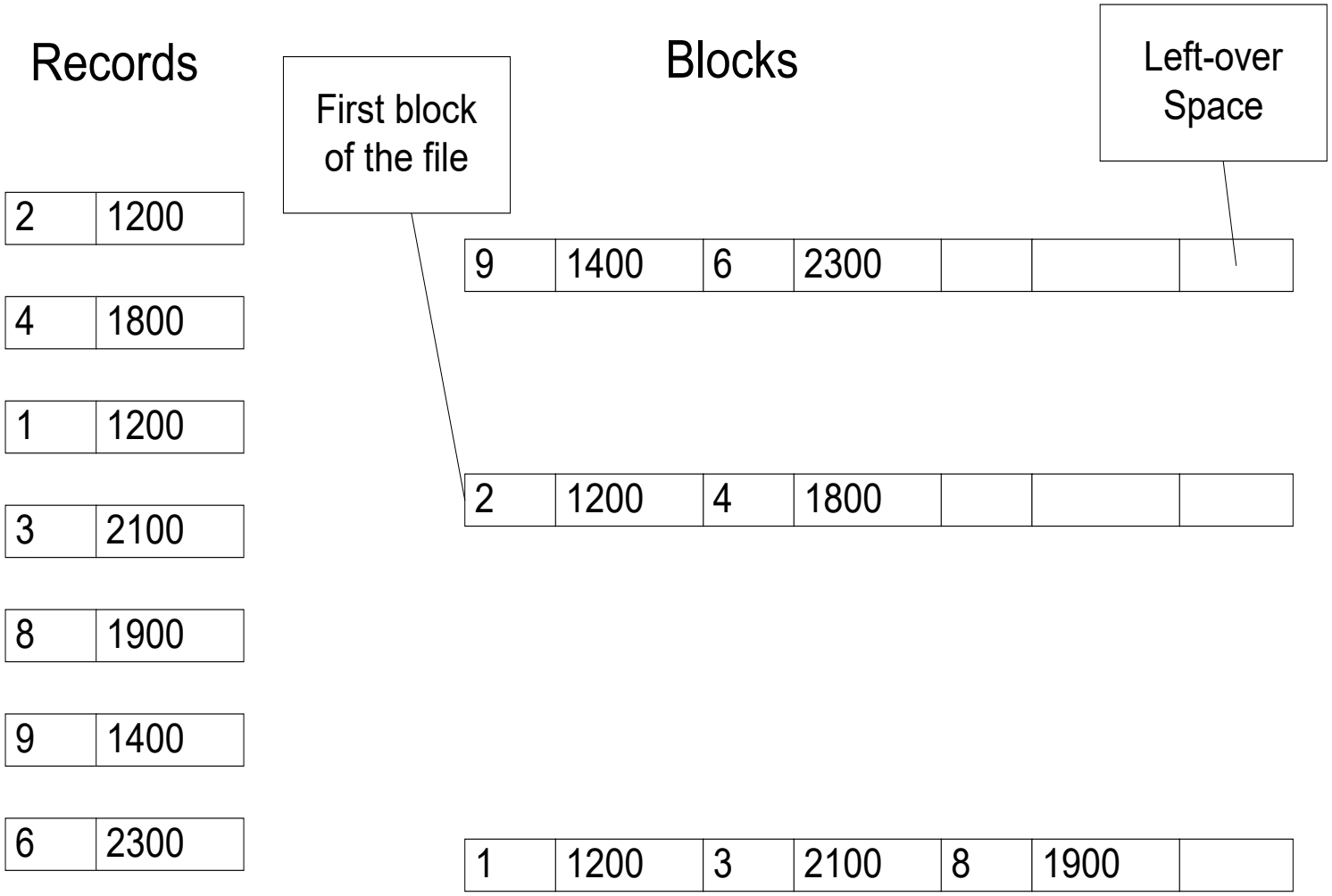
Relation

E#	Salary
1	1200
3	2100
4	1800
2	1200
6	2300
9	1400
8	1900

Records

2	1200
4	1800
1	1200
3	2100
8	1900
9	1400
6	2300

Example: Storing a Relation (Physical File)



Alternative Representations

- Column-wise: Store columns sequentially. Good if most queries need just a few columns of a table having hundreds of columns.
- Store row-wise but treat fixed length differently from variable-length fields.
- Some mixture of these strategies.

Processing a Query (for a Row Layout)

- Simple query

```
SELECT E#
FROM R
WHERE SALARY > 1500;
```

- What needs to be done “under the hood” by the file system:
  - Read into RAM at least all the blocks containing all records satisfying the condition (assuming none are in RAM.
    - It may be necessary/useful to read other blocks too, as we see later
  - Get the relevant information from the blocks
  - Perform additional processing to produce the answer to the query

- What is the cost of this?
- We need a “cost model”

Cost Model

- Two-level storage hierarchy
  - RAM
  - Disk (and more and more Solid State Drive (SSD))
- ***Reading or Writing a block costs one time unit***
  - One time unit assumption makes sense for solid state drives, because they are random access (i.e. proximity of the next block read to the previous block read plays no role in the time).
- ***Processing in RAM is free***
- Ignore caching of blocks (unless done previously by the query itself, as the byproduct of reading)
- Justifying the assumptions
  - Accessing the disk is much more expensive than any reasonable CPU processing of queries (we could be more precise, which we are not here)
  - We do not want to account formally for block contiguity on the disk; we do not know how to do it well in general (and in fact what the OS thinks is contiguous may not be so: the disk controller can override what OS thinks)
  - We do not want to account for a query using cache slots “filled” by another query; we do not know how to do it well, as we do not know in which order queries come

Implications of the Cost Model

- Goal: ***Minimize the number of block accesses***
- A good heuristic: ***Organize the physical database so that you make as much use as possible from any block you read/write***
- Note the difference between the cost models in
  - Data structures (where “analysis of algorithms” type of cost model is used: minimize CPU processing)
  - Data bases (minimize disk accessing)
- There are serious implications to this difference
- Database physical (disk) design is obtained by extending and “fine-tuning” of “classical” data structures

Example

- If you know exactly where E# = 2 and E# = 9 are:
- The data structure cost model gives a cost of 2 (2 RAM accesses)
- The database cost model gives a cost of 2 (2 block accesses)

Array in RAM

2	1200
4	1800
1	1200
3	2100
8	1900
9	1400
6	2300

Blocks on disk

9	1400	6	2300			
2	1200	4	1800			
1	1200	3	2100	8	1900	

Example (continued)

- If you know exactly where E# = 2 and E# = 4 are:
- The data structure cost model gives a cost of 2 (2 RAM accesses)
- The database cost model gives a cost of 1 (1 block access)

Array in RAM

2	1200
4	1800
1	1200
3	2100
8	1900
9	1400
6	2300

Blocks on disk

9	1400	6	2300			

2	1200	4	1800			

1	1200	3	2100	8	1900	

File Organization and Indexes

- If we know what we will generally be reading/writing, we can try to minimize the number of block accesses for “frequent” queries
- Tools:
  - File organization
  - Indexes (structures showing where records are located)

- **Essentially: File organization tries to provide:**
  - **When you read a block you get “many” useful records**

- **Essentially: Indexes try to provide:**
  - **You know where blocks containing useful records are**

- We discuss both in this unit

Tradeoff

- Maintaining file organization and indexes is not free
- Changing (deleting, inserting, updating) the database requires
  - Maintaining the file organization
  - Updating the indexes
- Extreme case: database is used only for SELECT queries
  - The “better” file organization is and the more indexes we have will result in more efficient query processing
- Extreme case: database is used only for INSERT queries
  - The simpler file organization and no indexes will result in more efficient query processing
  - Perhaps just append new records to the end of the file
- In general, there is a tradeoff.

Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

Review of Data Structures to Store N Numbers

- Heap: unsorted sequence (note difference from a common use of the term “binary heap” in data structures)
- Sorted sequence
- Hashing
- 2-3 trees

Heap (Logically an Unordered Array)

- Finding (including detecting of non-membership)
  Takes between 1 and N operations (N/2 on average)
- Deleting
  Takes between 1 and N operations
  (N/2 on average)
  Depends on the variant also
  - If the heap needs to be “compacted” it will take always N (first to reach the record, then to move the “tail” to close the gap)
  - If a “null” value can be put instead of a “real” value, then it will cause the heap to grow unnecessarily
- Inserting
  - Takes 1 (put in front), or N (put in back if you cannot access the back easily, otherwise also 1), or maybe in between by reusing null values
- Linked list: obvious modifications

Sorted Sequence (Rarely Used; B-Trees are Better)

- Finding (including detecting of non-membership)

  log N using binary search. But note that “optimized” deletions and insertions could cause this to grow (next transparency)

- Deleting

  Takes between log N and log N + N operations. Find the integer, remove and compact the sequence.

  Depends on the variant also. For instance, if a “null” value can be put instead of a “real” value, then it will cause the sequence to grow unnecessarily.

Sorted Sequence

- Inserting

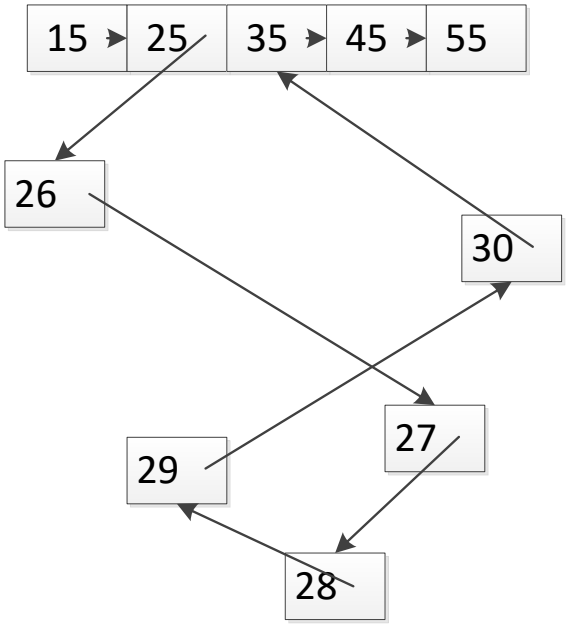
  Takes between $\log N$ and $\log N + N$ operations. Find the place, insert the integer, and push the tail of the sequence.

  Depends on the variant also. For instance, if “overflow” values can be put in the middle by means of a linked list, then this causes the “binary search to be unbalanced, resulting in possibly up to N operations for a Find.

Sorted Sequences (continued)

- We have a sorted sequence of 15, 25, 35, 45, 55, and room for a pointer

- 26, 27, 28, 29, 30 arrive and are inserted as “overflow” from the right place



It takes a long time to find 30

Hashing

- Pick a number B "somewhat" bigger than N
- Pick a "good" pseudo-random function h
  h: integers $\rightarrow$ {0,1, ..., B – 1}
- Create a "bucket directory," D, a vector of length B, indexed 0,1, ..., B – 1
- For each integer k, it will be stored in a location pointed at from location D[h(k)], or if there are more than one such integer to a location D[h(k)], create a linked list of locations "hanging" from this D[h(k)]
- Probabilistically, almost always, most of the locations D[h(k)], will be pointing at a linked list of length 1 only

Hashing: Example of Insertion

N = 7

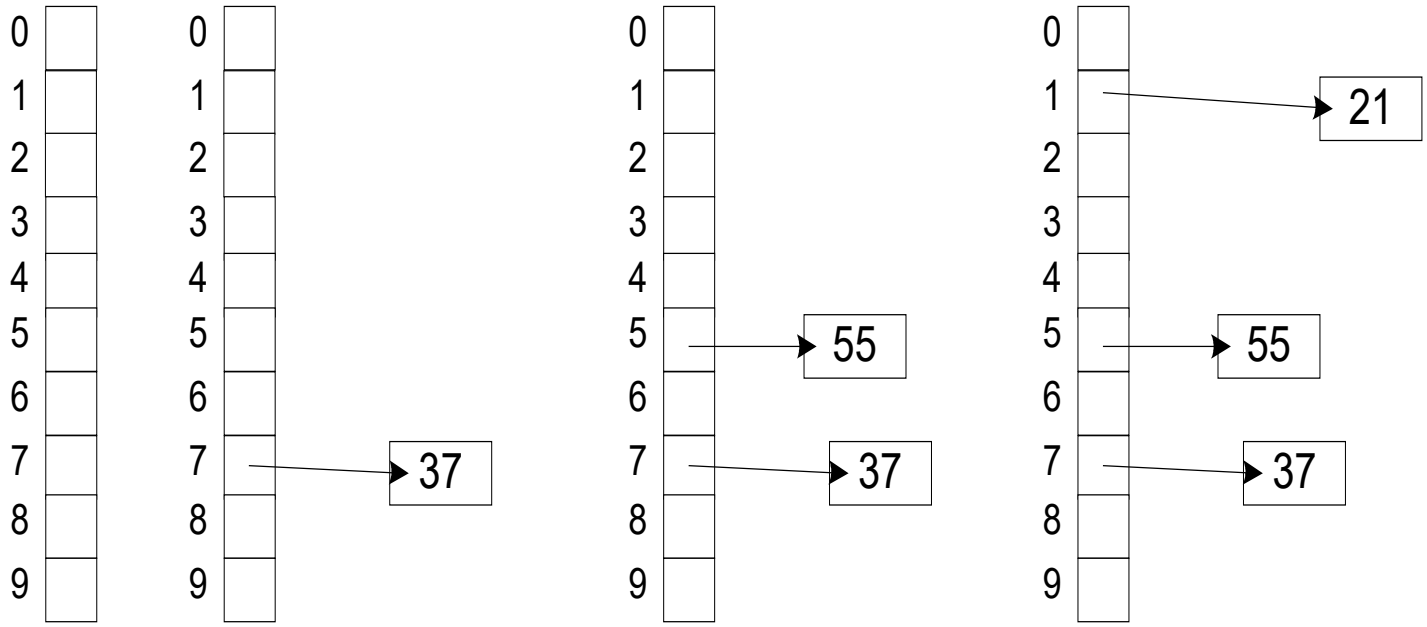
B = 10

h(k) = k mod B (this is an extremely bad h, but good for a simple example)

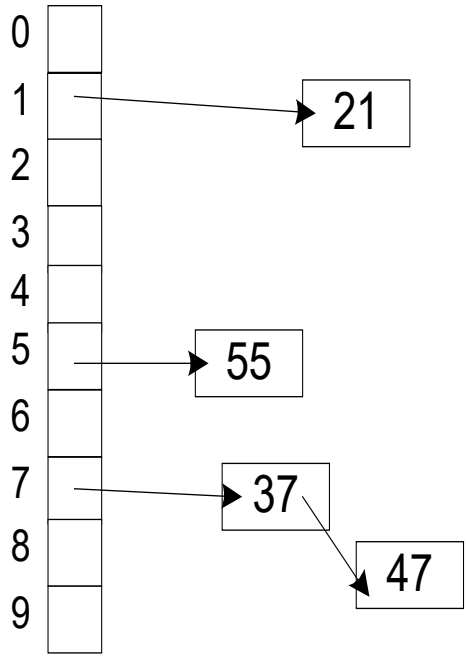
Integers arriving in order:

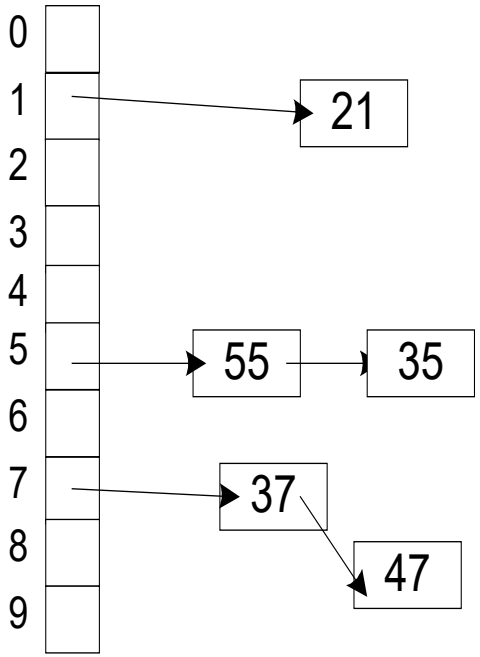
37, 55, 21, 47, 35, 27, 14

Hashing: Example of Insertion (continued)

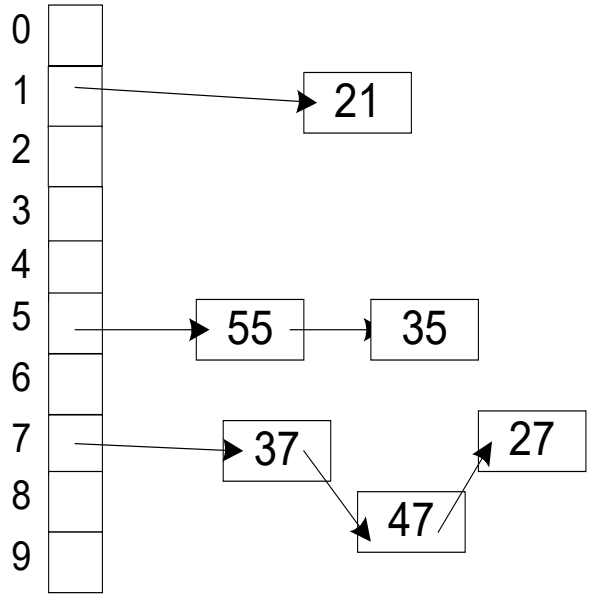


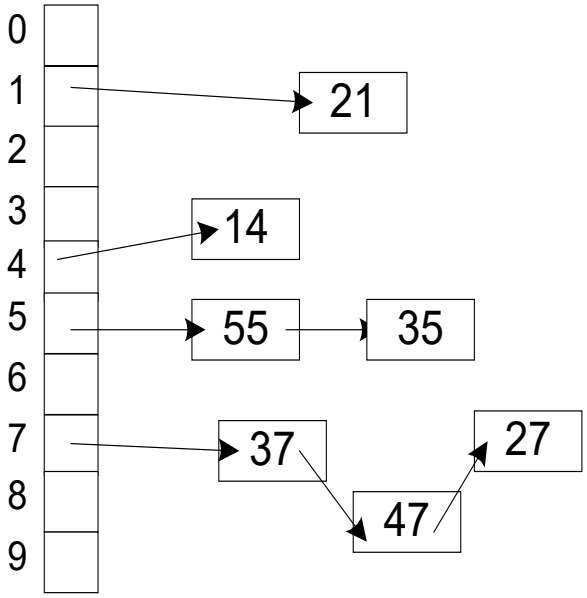
Hashing: Example of Insertion (continued)





Hashing: Example of Insertion (continued)





Hashing

- Assume, computing h is “free”

- Finding (including detecting of non-membership)
  Takes between 1 and N + 1 operations.

  Worst case, there is a single linked list of all the integers from a single bucket.

  Average, between 1 (look at bucket, find nothing) and a little more than 2 (look at bucket, go to the first element on the list, with very low probability, continue beyond the first element)

Hashing (continued)

- Inserting

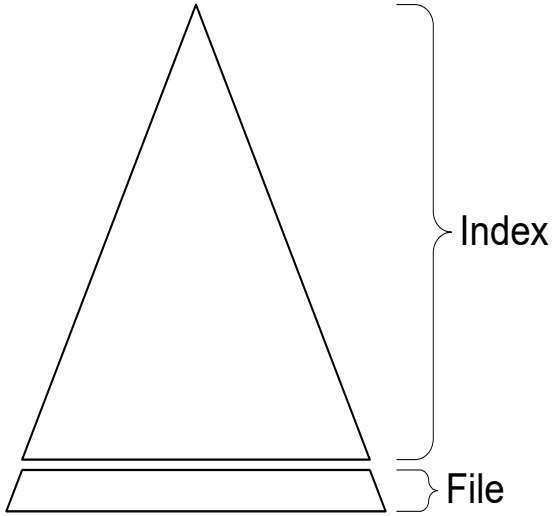
  Obvious modifications of ”Finding”

  Sometimes N becomes is “too close” to B (bucket table becomes too small)
  - Then, increase the size of the bucket table and rehash (can do it incrementally)
  - Number of operations linear in N
  - Can amortize across all accesses (ignore, if you do not know what this means), but know that it is still constant operations per access even when taking into account rehashing

- Deleting

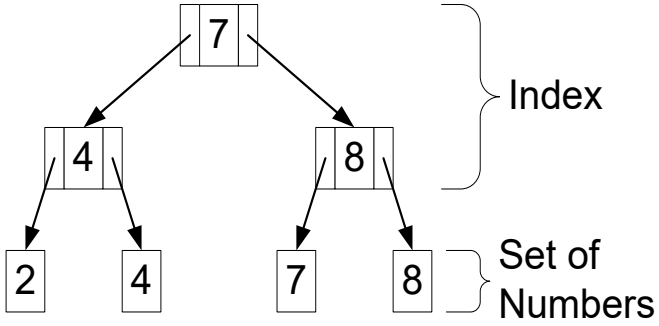
  Obvious modification of “Finding”
  Sometimes N too small for B, act “opposite” to Inserting

Reviewing Binary Search Trees In Data Structures



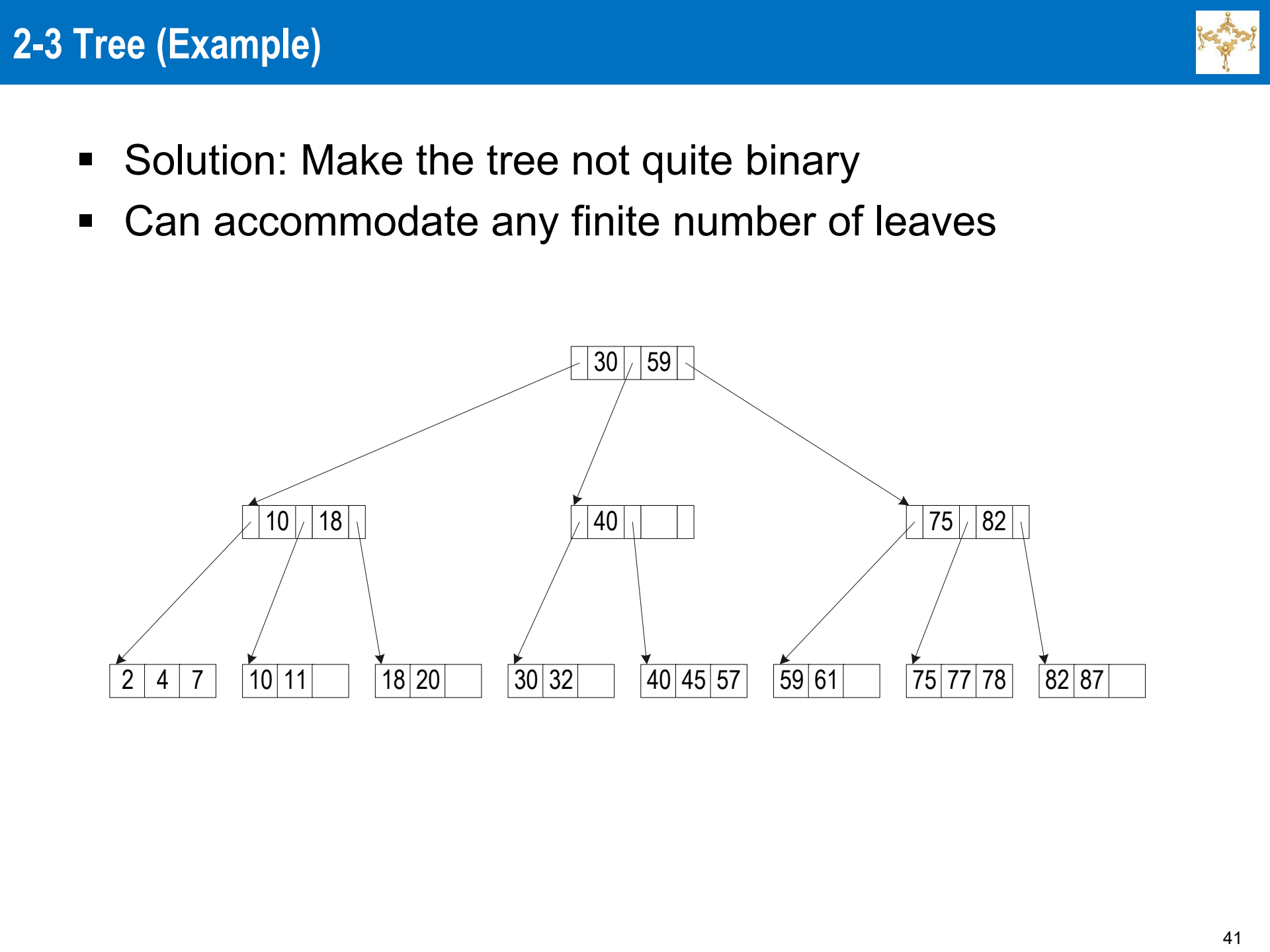
- Example: the index is a binary search tree
- The file is just a set of numbers not organized in any particular way: e.g., the numbers do not have to be contiguous

Reviewing Binary Search Trees In Data Structures (continued)



- Each node of the index consists of
  - A pointer to the root of the left subtree
  - A pointer to the root of the right subtree
  - A separator (number)
- All the leaves of the left subtree are smaller than all the leaves of the right subtree
- The separator is the smallest leaf in the right subtree
- To search for a number: the separator tells you into which subtree to go
- Limitation: A binary tree cannot be “fully balanced”:
  - All paths from root to leaves of the same length

2-3 Tree (Example)



- Solution: Make the tree not quite binary
- Can accommodate any finite number of leaves

2-3 Trees

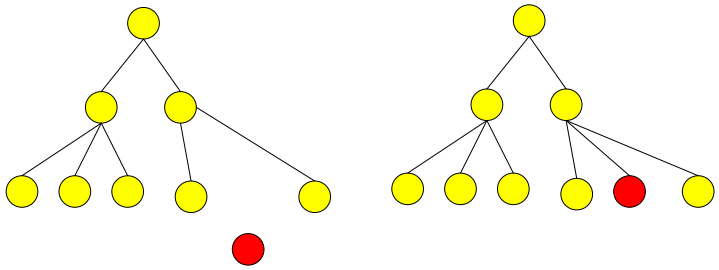
- A 2-3 tree is a rooted (it has a root) directed (order of children matters) tree
- All paths from root to leaves are of same length
- Each node (other than leaves) has between 2 and 3 children
- For each child of a node, other than the last, there is an index value stored in the node
- **For each non-leaf node, the index value indicates the smallest value of the leaf in the subtree rooted to the right of the index value**
- A leaf has between 2 and 3 values from among the integers to be stored

2-3 Trees (continued)

- It is possible to maintain the "structural characteristics above," while inserting and deleting integers
- Sometimes for insertion or deletion of integers there is no need to insert or delete a node
  - » E.g., inserting 19
  - » E.g., deleting 45
- Sometimes for insertion or deletion of integers it is necessary to insert or delete nodes (and thus restructure the tree)
  - » E.g., inserting 88,89,97
  - » E.g., deleting 40, 45
- Each such restructuring operation takes time at most linear in the number of levels of the tree (which, if there are N leaves, is between $\log_3 N$ and $\log_2 N$); so we write O(log N)). Should technically be Θ(log N).
- We show by example of an insertion and of a deletion

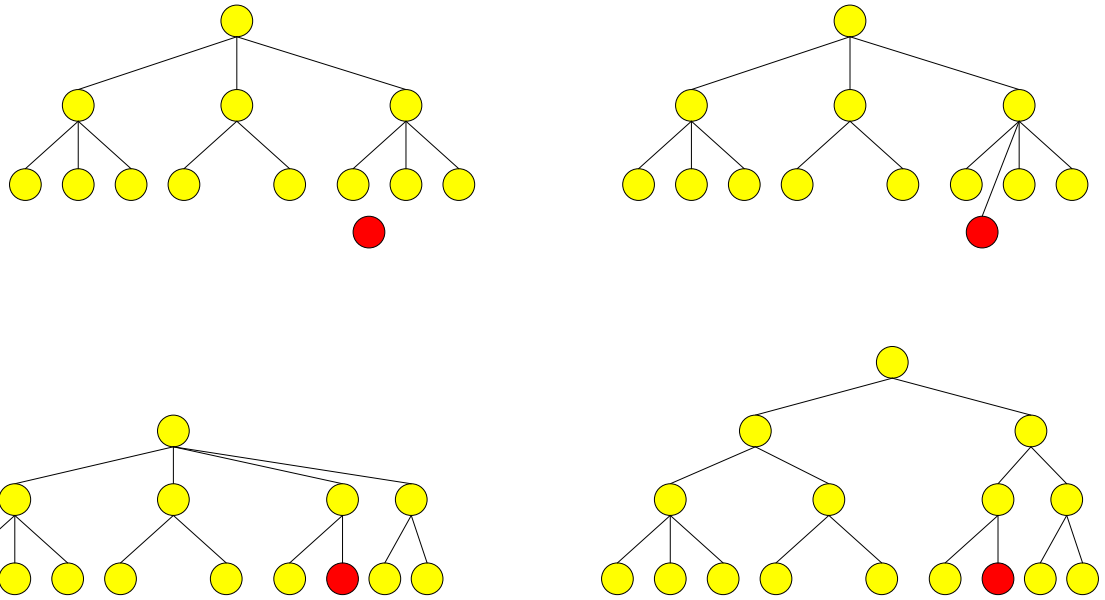
Insertion of a Node in the Right Place (Fasted Case)

- First find the place to insert using $\Theta(\log N)$ operations
- Insertion resolved at the leaf level



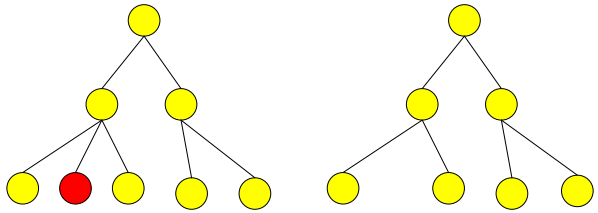
Insertion of a Node in the Right Place (Slowest Case)

- First find the place to insert using $\Theta(\log N)$ operations
- Insertion propagates up to the creation of a new root

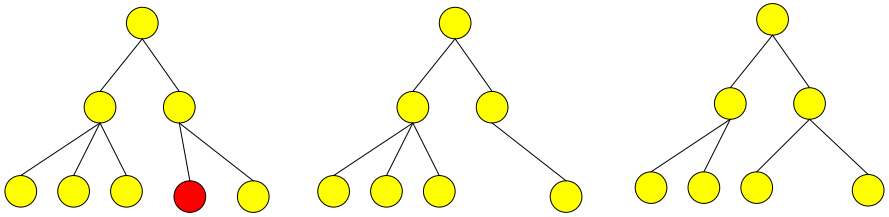


Deletion Of A Node
(The Fastest Case)

- First find the place to insert using $\Theta(\log N)$ operations
- Deletion resolved at the leaf level without sibling stealing

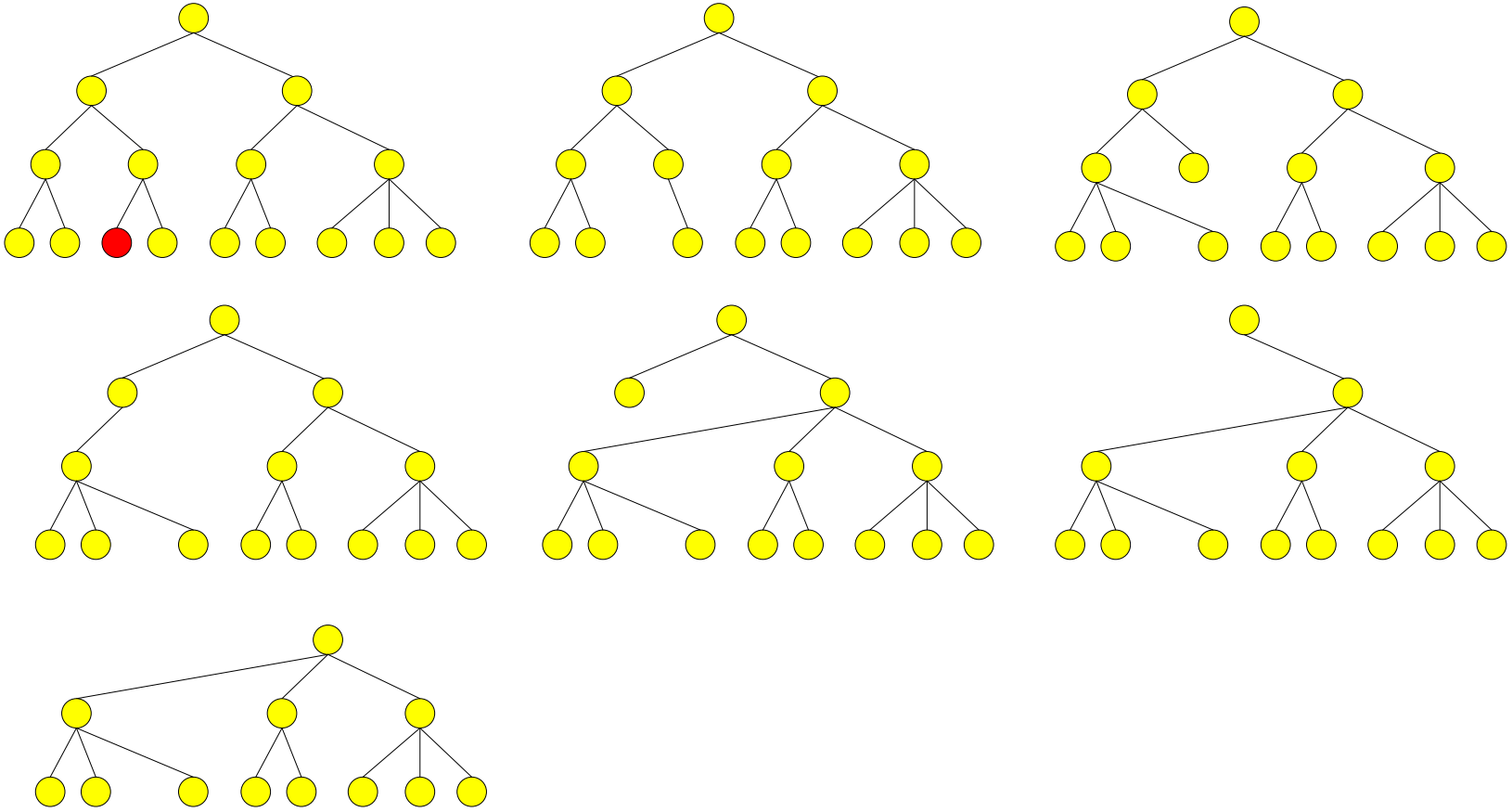


- First find the place to insert using $\Theta(\log N)$ operations
- Deletion resolved at the leaf level with sibling stealing



Deletion Of A Node
(The Slowest Case)

- First find the place to insert using $\Theta(\log N)$ operations
- Deletion propagates up to the deletion of the old root

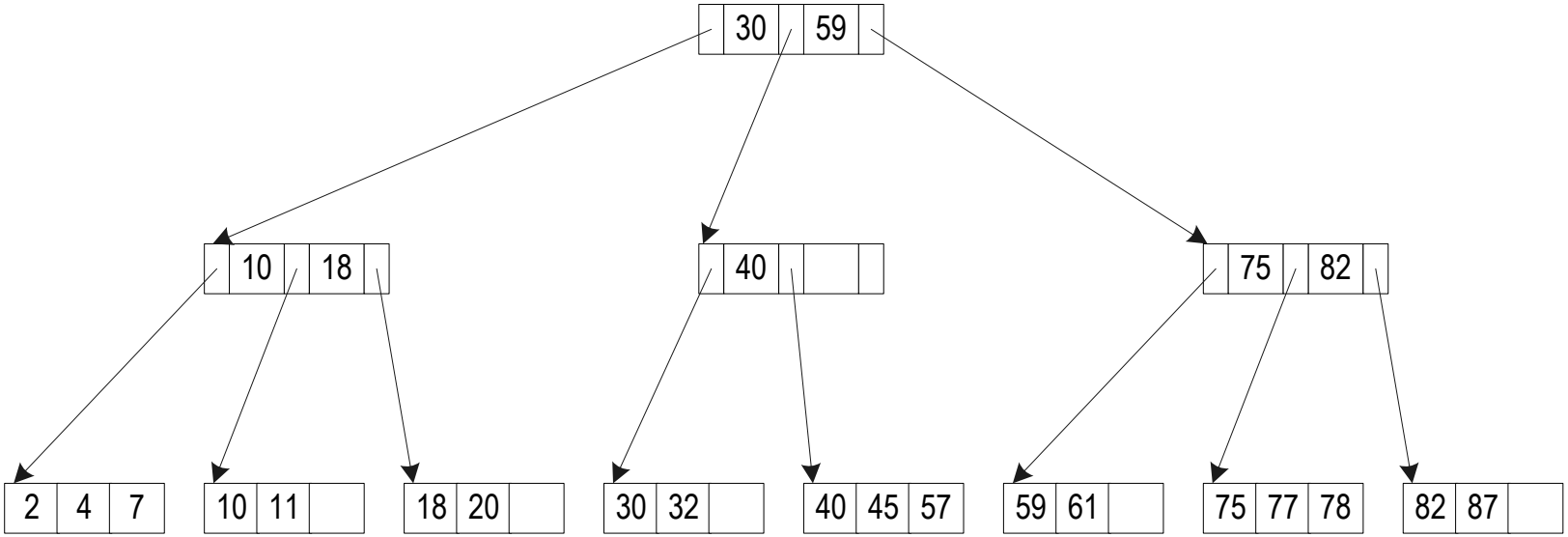


A Detailed Example of Inserting an Item Into a 2-3 Tree

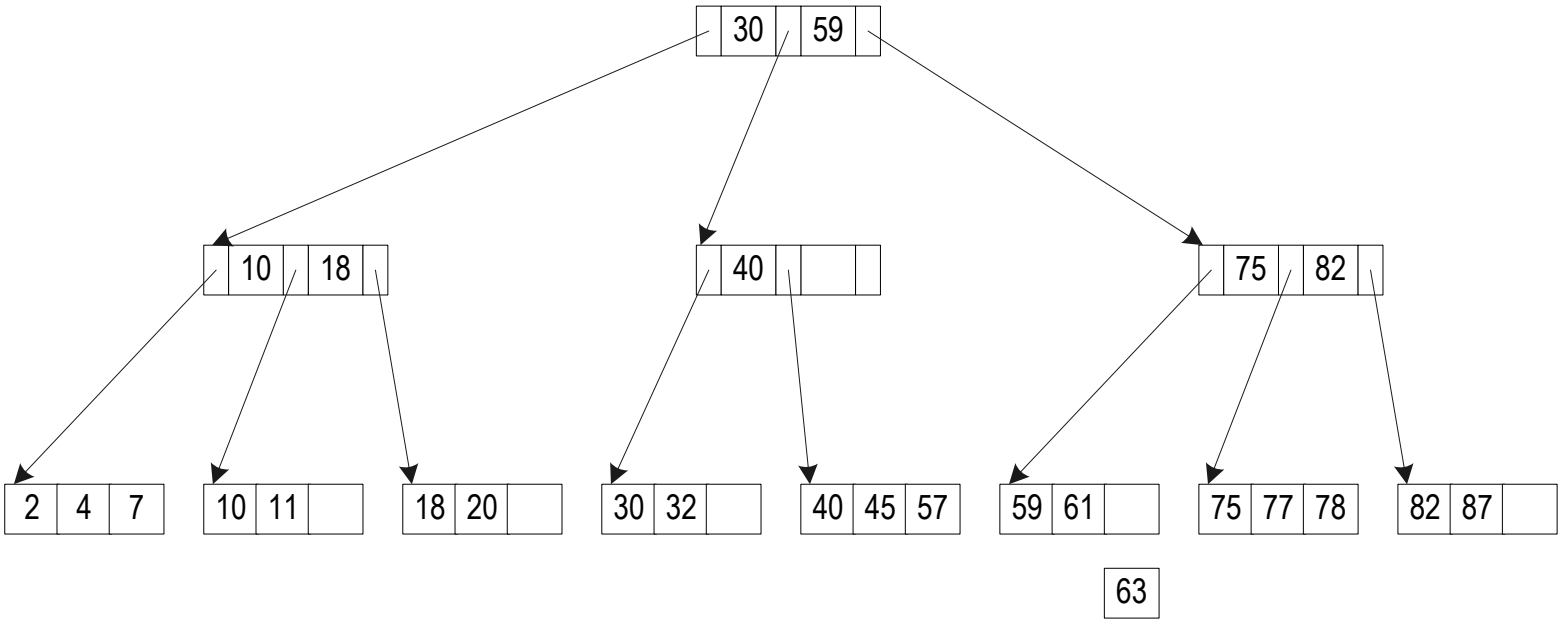
- We have a 2-3 tree
- We will insert 63 into it
- Then starting again with the original tree, we will insert 76 into it

- When we want to insert, we
  - Start with the root we traverse the path to the leaves level and find where to insert the new item
  - Then if we can, we insert it
  - If we cannot, we add a new vertex to the tree and “patch up” the parent
  - If necessary, we go all the way up, add a sibling to the root, and create a new root

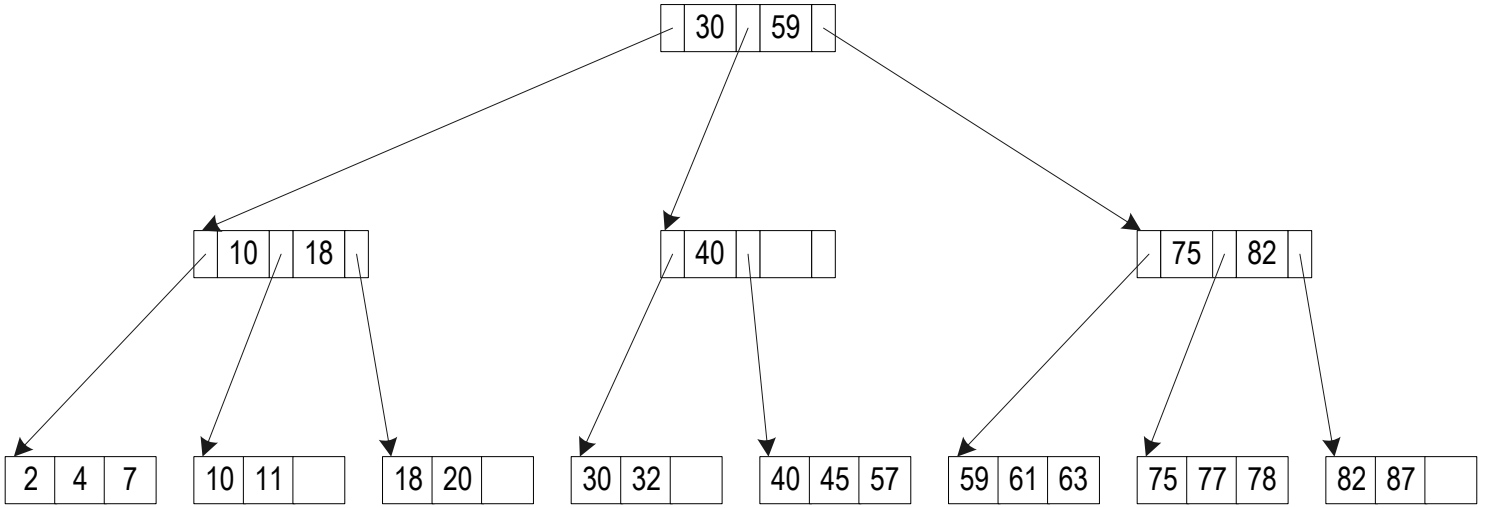
The Original Tree



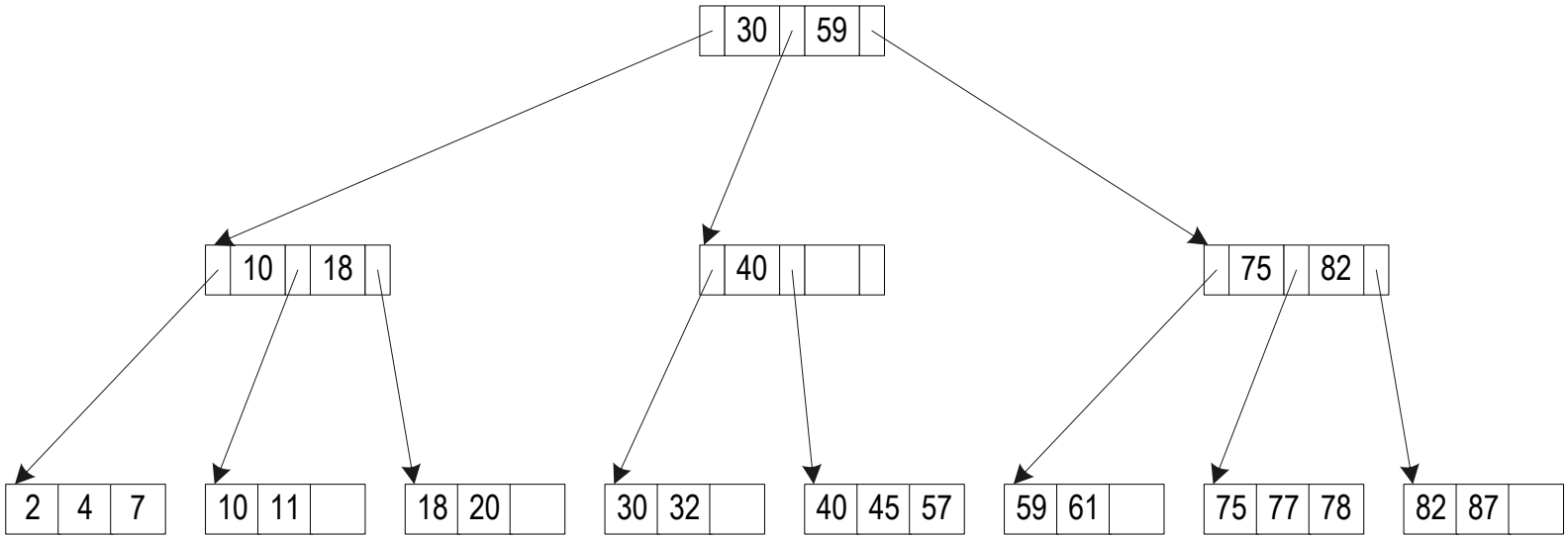
Inserting 63



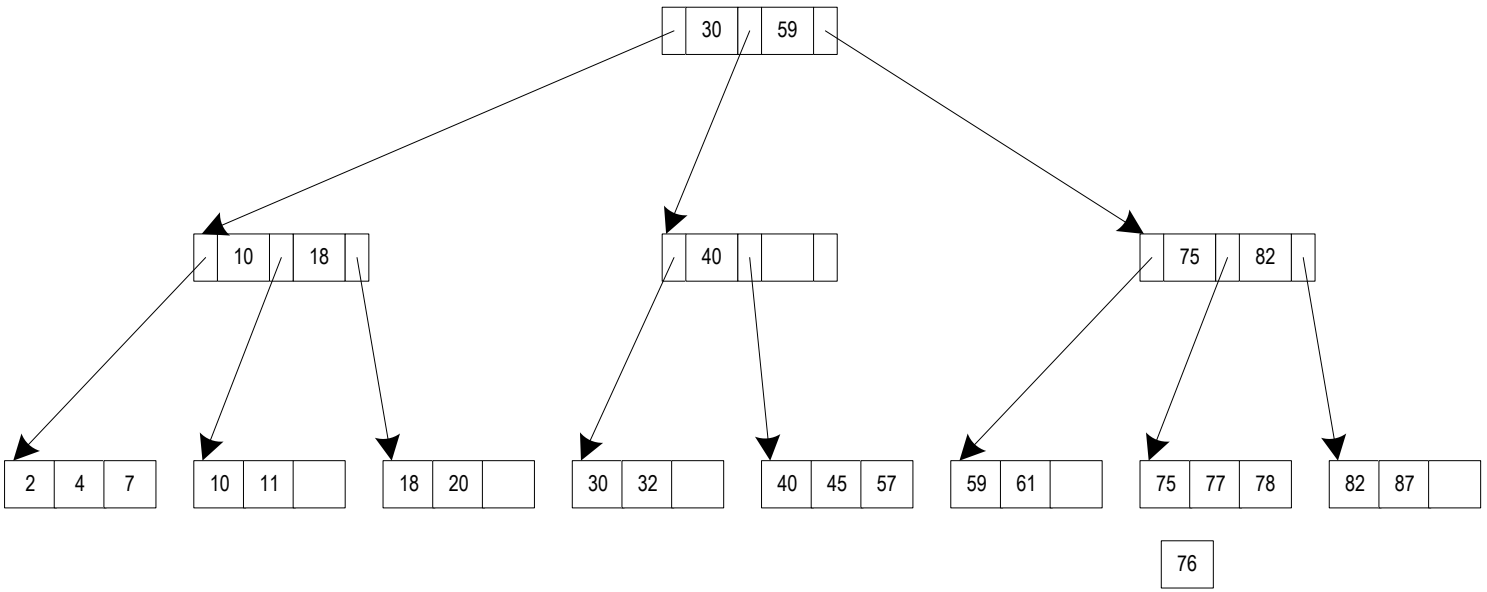
Inserting 63



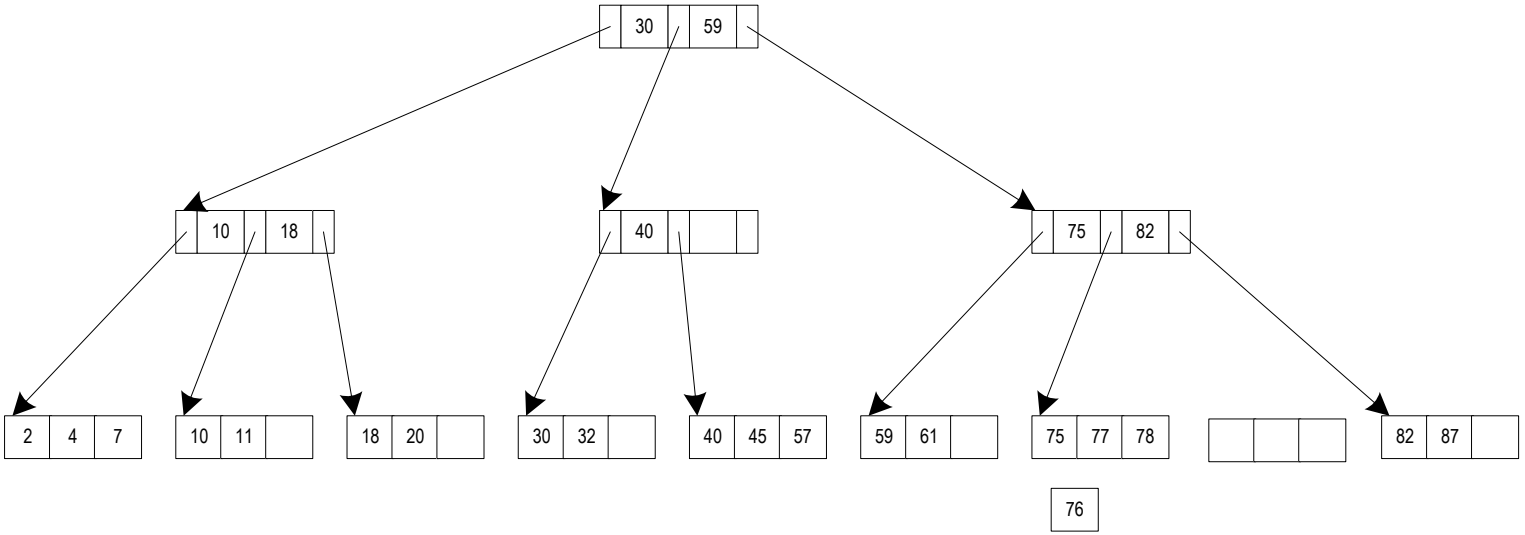
The Original Tree



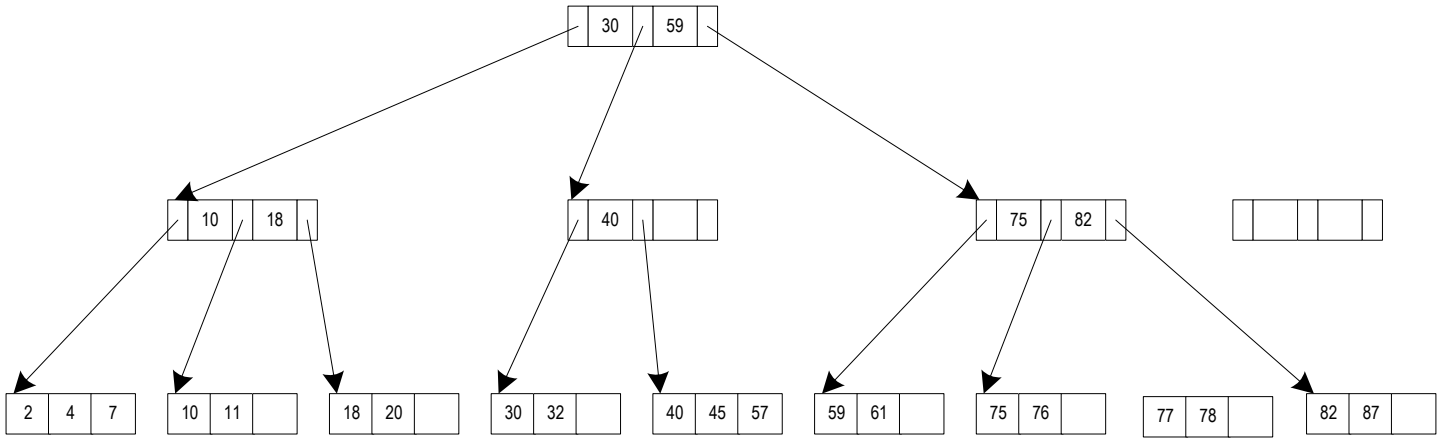
Inserting 76



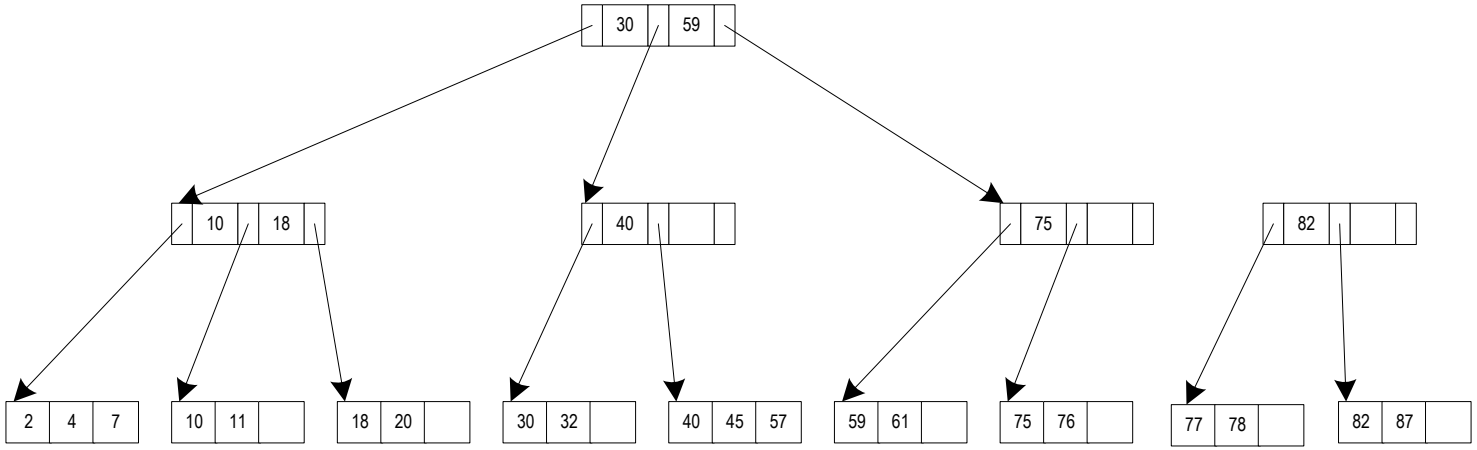
Inserting 76



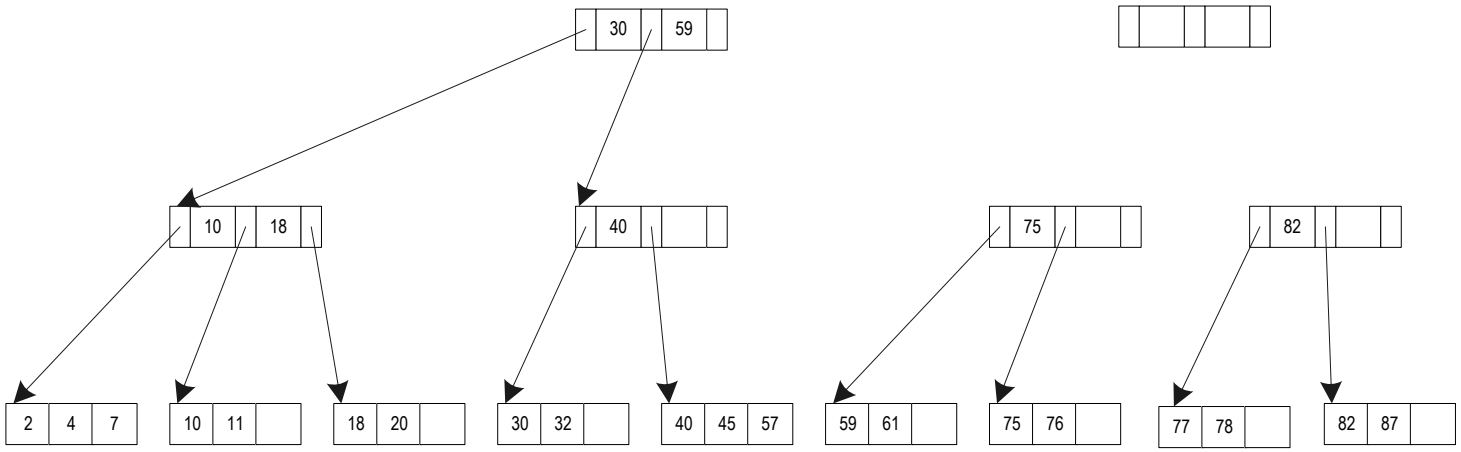
Inserting 76



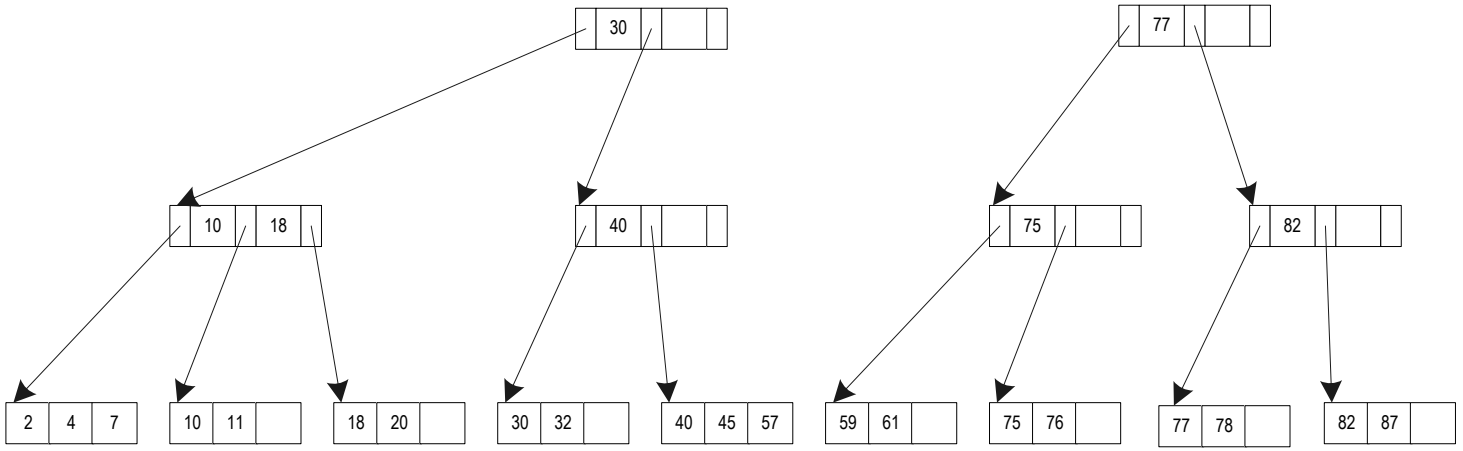
Inserting 76



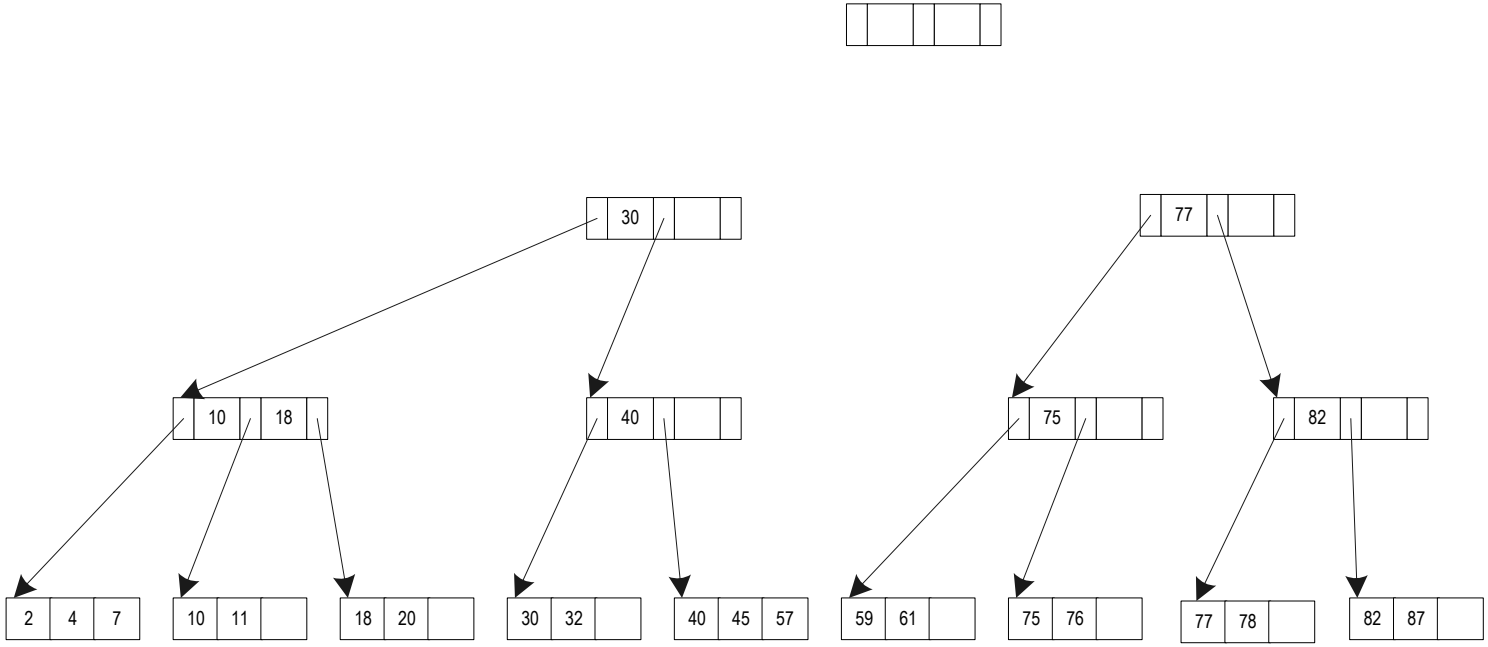
Inserting 76



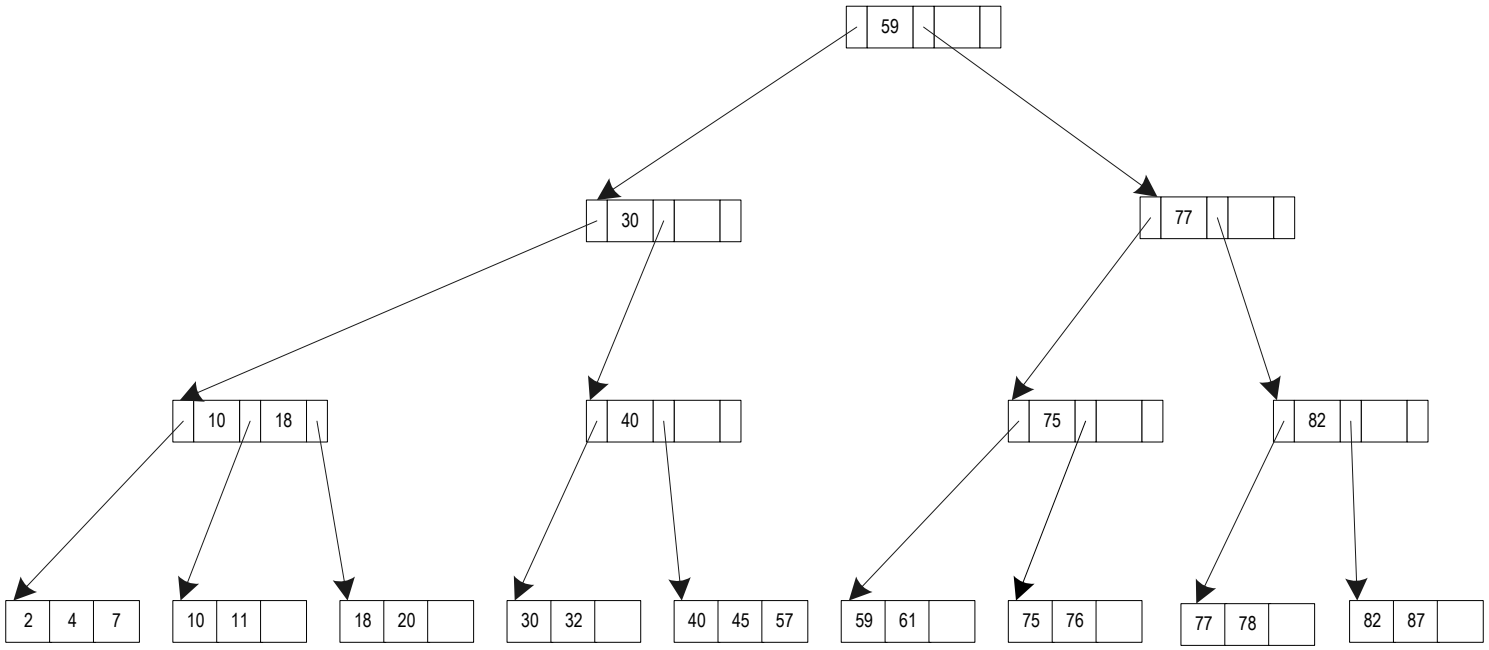
Inserting 76



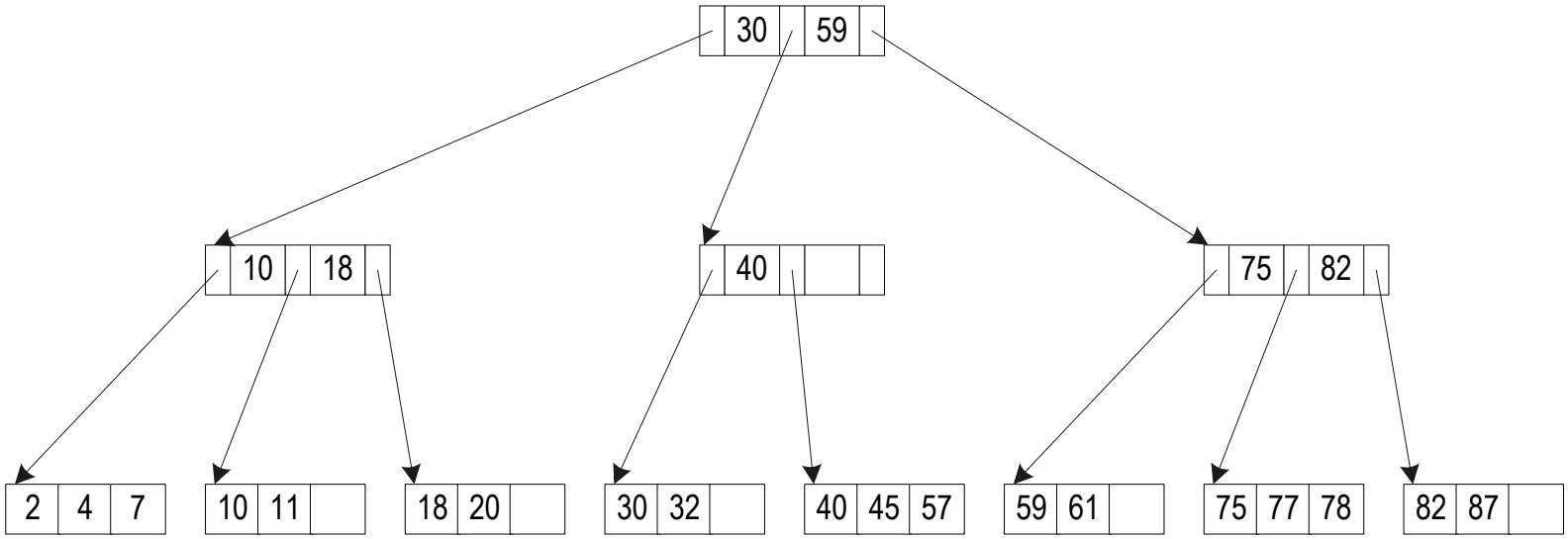
Inserting 76



Inserting 76



The Original Tree



2-3 Trees

- Finding (including detecting of non-membership)

  Takes $\Theta(\log N)$ operations

- Inserting while keeping trees balanced

  Takes $\Theta(\log N)$ operations

- Deleting while keeping trees balanced

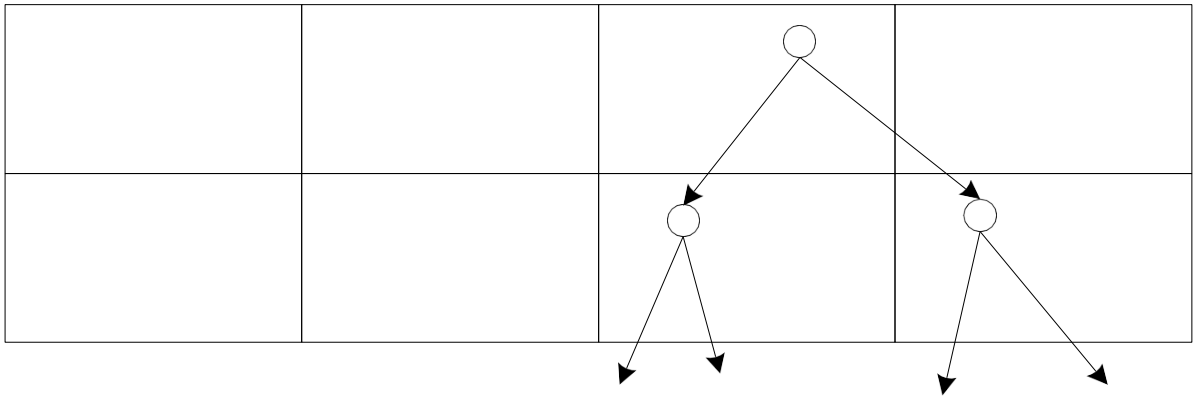
  Takes $\Theta(\log N)$ operations

2-Trees vs. 4-Trees Efficiency?

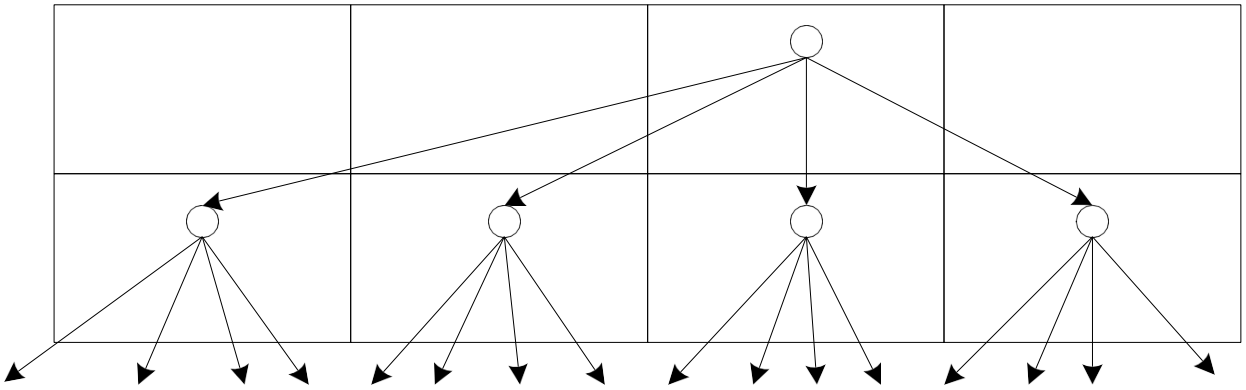
- We consider 2-trees and 4-trees
- The vertices of the tress are distributed “randomly” in the blocks of the disk
- Let’s assume, for the purpose of a sharp example that no two vertices are in the same block
- A block is large compared to the size required to store a vertex of a tree
- A vertex contains the appropriate number of pointers and additional information (separators) telling us which pointer to follow based on what we want to find

- A block can store a vertex of a 4-tree, and therefore, of course a vertex of a 2-tree
- Let’s examine what we can do with 2 block access

2-Trees vs. 4-Trees Efficiency?



There are 8 blocks. We store 2-trees.
By reading two blocks (the root and one of its children), we know the location of any one of 4 leaves



There are 8 blocks. We store 4-trees.
By reading two blocks (the root and one of its children), we know the location of any one of 16 leaves

2-Trees vs. 4-Trees Efficiency?

- We conclude that 4-trees are more efficient than 2-trees
- And if we could store *m*-trees, for $m > 4$, that would be even better

- Conclusion: Choose *m* as big as possible so that a vertex can still fit in a single block

B+ Trees: Generalize 2-3 Tree to k-(2k-1) Tree

- Note that a 2-3 tree is a B^+-tree with m = 3 (k= 2)
- Important properties:
- Much greater fanout than a 2-3 trees, so fewer levels.
- When inserting a pointer at the leaf level, restructuring of the tree may propagate upwards to the root just as in 2-3 tree.
- Unlike 2-3 tree, normally store all data at the leaves.

B+ Trees: Why Good for Disk?

- Depth of the tree is “logarithmic” in the number of items in the leaves
- In fact, this is logarithm to the base at least ceiling of m/2 (ignore the children of the root; if there are few, the height may increase by 1)

- Imagine a trillion data items.
- Room only for one billion.
- If fanout is 1000, then one more level on disk.
- If fanout is 2 then 10 more levels on disk.

An Example

- Our application (parameters not necessarily realistic):
  - Disk of 16 GiB (GiB or gibibyte means 2^{30} bytes, GB or gigabyte means 10^9 bytes); very small, but a good example
  - Each block of 512 bytes; rather small, but a good example
  - File of 20 000 000 records
  - Each record of 25 bytes
  - Each record of 3 fields:
    SSN: 5 bytes (packed decimal), name: 14 bytes, salary: 6 bytes
- We want to access the records using the value of SSN
- We want to use a B^+-tree index for this
- Our goal, to minimize the number of block accesses, so we want to derive as much benefit from each block, but there is a tradeoff with the work to do a split and possibly with lock contention.

An Example: If We Can Do Our Own Design of B+ Trees

- There are 16 GiB = 2^{34} bytes on the disk, and each block holds 2^9 = 512 bytes.
- Therefore, there are 2^{34} / 2^9 = 2^{25} blocks, say numbered 0,1,…,2^{25} – 1; each such number can be stored in 25 bits
- Therefore, a block address can be specified in 25 bits; in practice likely determined by the operating system, e.g., 32 or 64 bits,
- We will not not rely on what the file system may do and we will optimize ourselves
  - The system may allocate 32 bits or 64 bits to a block address even if it is more than necessary
- That is what you should also do during the course, that is compute how much space you need to store a pointer, following what we will do next

An Example: If We Can Do Our Own Design of B+ Trees (continued)

- We need 25 bits to store a block address
- We will allocate 4 bytes to a block address by rounding up 25/8 = 3.125, getting 4
  - » We may be "wasting" space, by working on a byte as opposed to a bit level, but simplifying the calculations
  - » It is just a coincidence that this is what a 32-bit operating system may do
- A node in the B-tree will contain some m pointers (each storing a block address) and m – 1 keys, so what is the largest possible value for m, so a node fits in a block?
- (m) × (size of pointer) + (m – 1) × (size of key) ≤ size of the block
  (m) × (4) + (m – 1) × (5) ≤ 512
  9m ≤ 517; m ≤ 57.4...
  m has to be an integer and therefore m = 57

An Example

- Therefore, the root will have between 2 and 57 children
- Therefore, an internal node will have between 29 and 57 children
  - 29 is the ceiling of 57/2
- We will examine how a tree can develop in two extreme cases:
  - "narrowest" possible
  - "widest possible
- Level Nodes in the narrow tree Nodes in the wide tree

1	1	1
2	2	57
3	58	3 249
4	1 682	185 193
5	48 778	10 556 001
6	1 414 562	601 692 057
7	41 022 298	34 296 447 249

What To Use?

- If the set of integers is large, use hashing or 2-3 trees
- Use 2-3 trees if “many” of your queries are range queries
- Find all the SSNs (integers from the range 0…999999999) in your set that lie in the range 070430000 to 070439999
- Use hashing if “very few” of your queries are range queries
- If you have a total of 10,000 employees (integers) randomly chosen from the set 0 ,..., 999999999, how many, on the average, will fall in the range above?
  - The probability of an employee to have a “desired” social security number: number of employees / number of Social Security Numbers = 10,000 / 1,000,000,000 = 0.00001
  - Almost correct: Expected Number of “desired” employees = 10,000 × 0,00001 = 0.1
- Likely: there is no “desired” employee, but we must check if we want to find all the “desired” employees: hashing useless

What To Use? (continued)

- How will you find the answer using hashing, and how will you find the answer using 2-3 trees?
  - Hashing: make a probe for each bucket: 10,00 operations. May as well examine the list sequentially
  - 2-3 trees: Search for 070439999 and go right: $\log_2 10{,}000 = 14$

Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

Back To Databases

- We have a file of blocks
  - » Each block is a sequence of records
- Whenever we read or write, we read or write a block: no smaller unit is physically accessible
- We had two “principles”
- Organize the file so that when you read a block you get “many” useful records
- Organize an index to a file so that you easily find the useful blocks

Index And File

- In general, we have a data file of blocks, with blocks containing records
- Each record has a field, which contains a key, uniquely labeling/identifying the record and the "rest" of the record
- There is an index, which is another file which essentially (we will be more precise later) consists of records which are pairs of the form (key, block address)
- ***For each (key, block address) pair (K, B) in the index file, the block address contains a pointer to a block of the file that contains the record with that key***
- ***But we are not saying that necessarily*** for every record in the file with key = K in block address B there is an there is a pair (K, B) in the index file

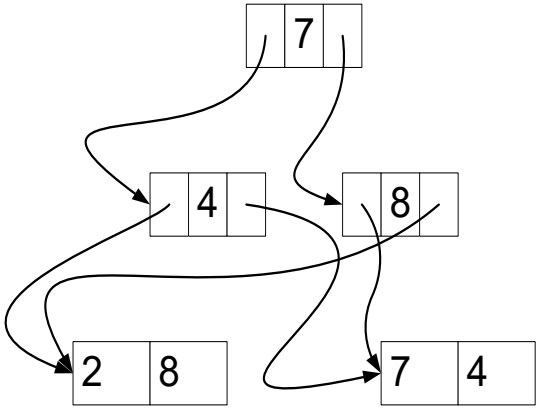
File Types And Index Types - Examples Setup

- We will store records in blocks
- Each block can store at most 3 records
  - » We will actually store between 2 and 3 and will see later why
- We will only show the keys of the records in our examples
  - » But, in general, the records contain additional information

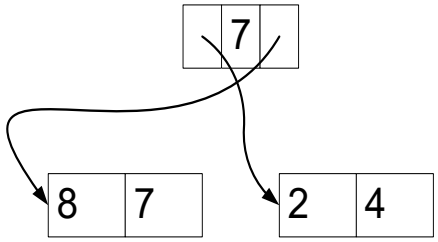
- We will not think in terms of trees to go over two concepts
  - » Is the file clustered or unclustered?
  - » Is the index dense of sparse?
- How are these two concepts related?

Index: Dense or Sparse

- An index is ***dense*** if for every record (that is for every key value) that we have in the database there is a dedicated pointer from the index to the block containing the record (in our example a block contains two records and we only show the key values in the records)
  - » Pointers always point at blocks and not at records
  - » Once a block is in RAM, useful records are easily found
- Otherwise, an index is ***sparse***
- Next we talk about ***clustering***



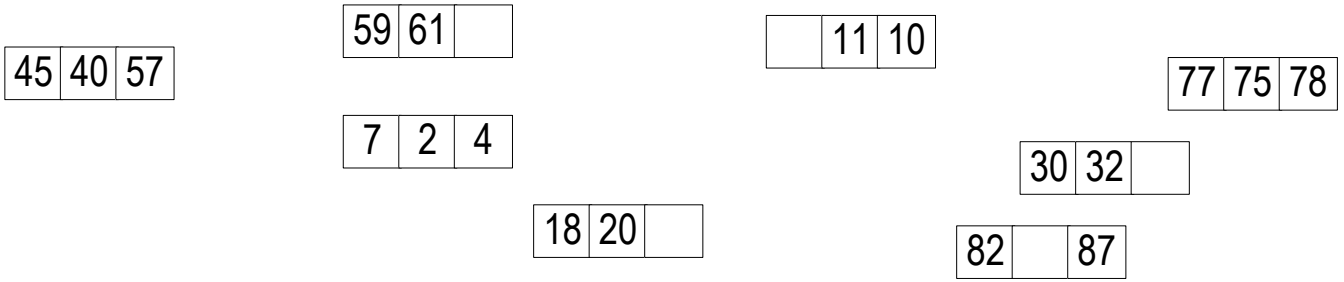
dense index and unclustered file



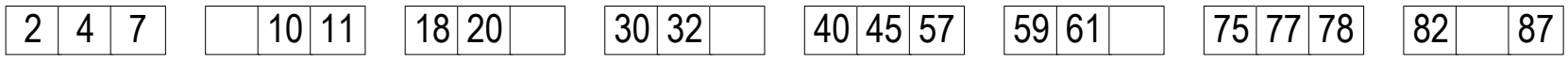
sparse index and clustered file

File: Clustered Or Unclustered?

- Below are blocks of a file laid out on a disk



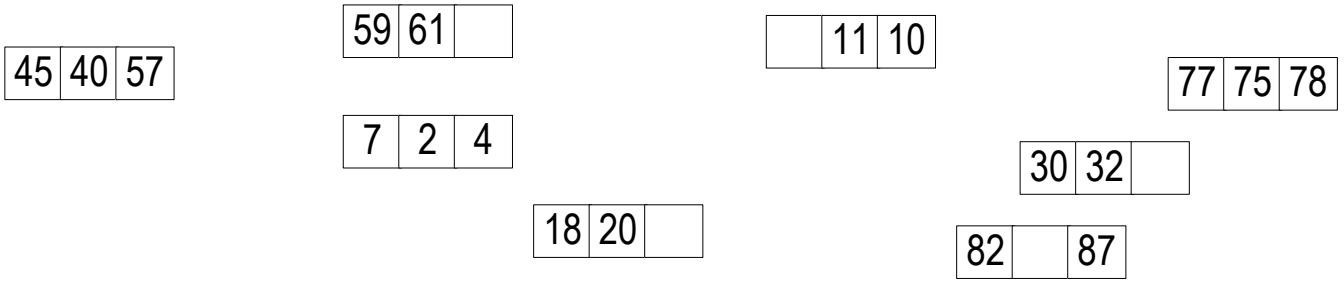
- By performing two types of operations, ***we can obtain a sorted file, sorted by keys*** (do not worry about the "holes")
  - » Moving blocks around
  - » Moving records in blocks (but not records between blocks)



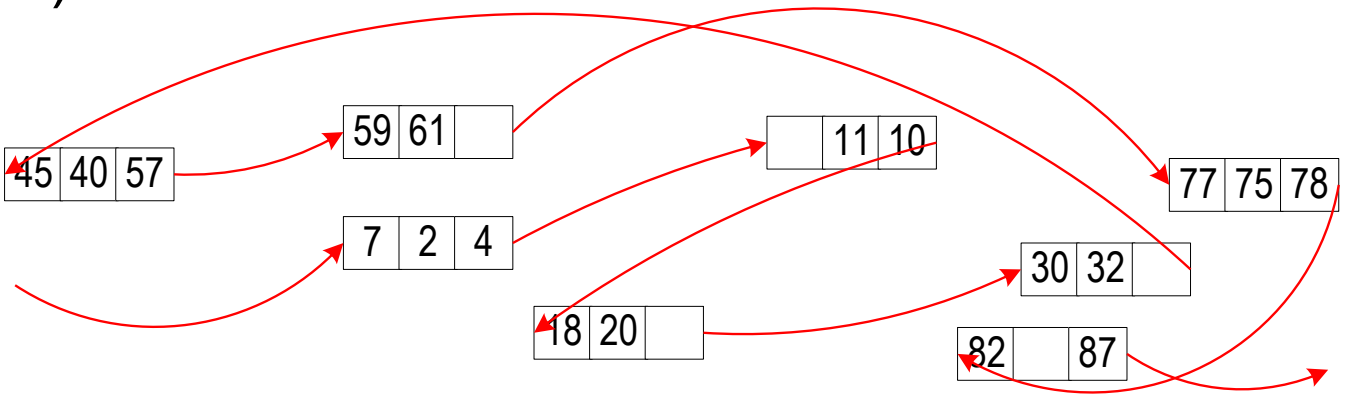
- Therefore the (original file) is ***clustered***

Another View Of A Clustered File

- Below are blocks of a file laid out on a disk

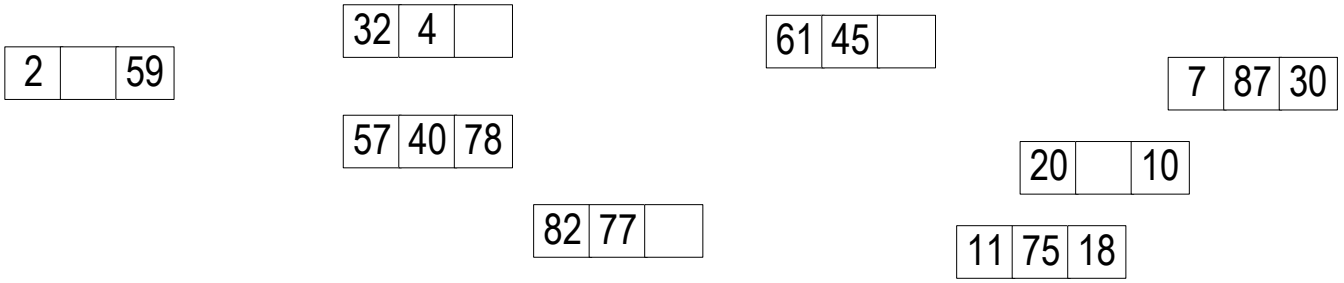


- Can consider a linked list of blocks (by adding pointer fields)



File: Clustered Or Unclustered?

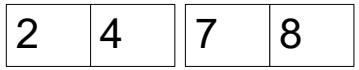
- Below are blocks of a file laid out on a disk



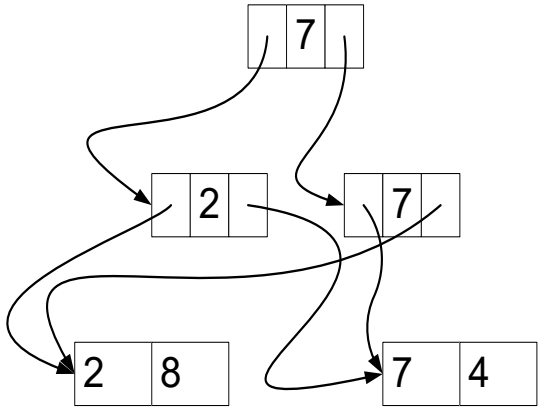
- By performing two types of operations, ***we cannot obtain a sorted file, sorted by keys***
  - » Moving blocks around
  - » Moving records in blocks (but not records between blocks)

- Therefore, the file is ***unclustered***

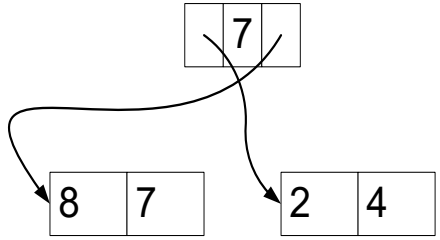
File: Clustered or Unclustered

- A file is ***clustered on attributes X*** if the records are organized based on their X values.
- Otherwise, a file is ***unclustered***
- In our example, the first file is unclustered because it ***cannot be sorted*** without moving records between blocks
- In our example, the second file is clustered because it ***can be sorted*** like this 

without moving records between blocks

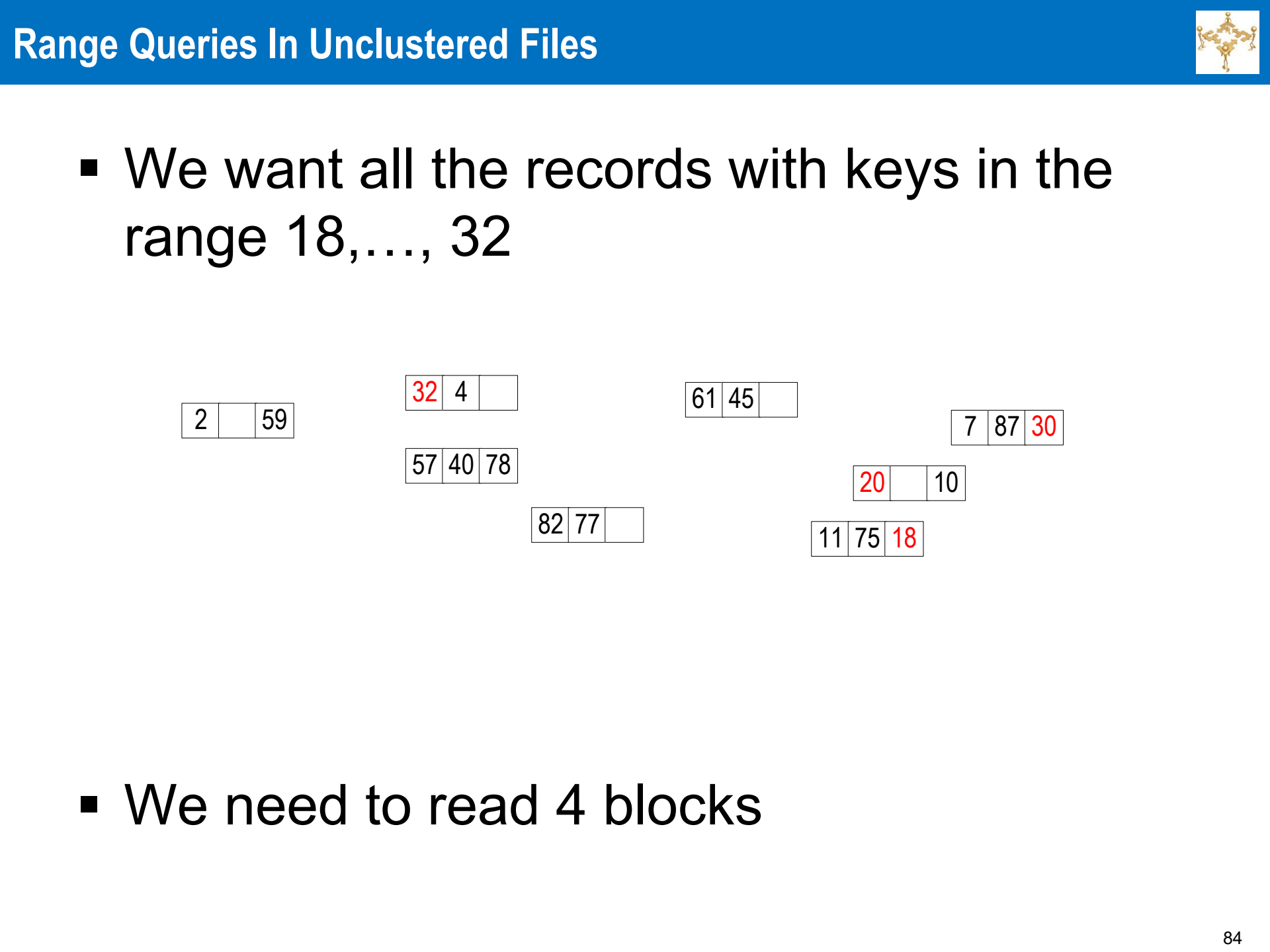


dense index and unclustered file



sparse index and clustered file

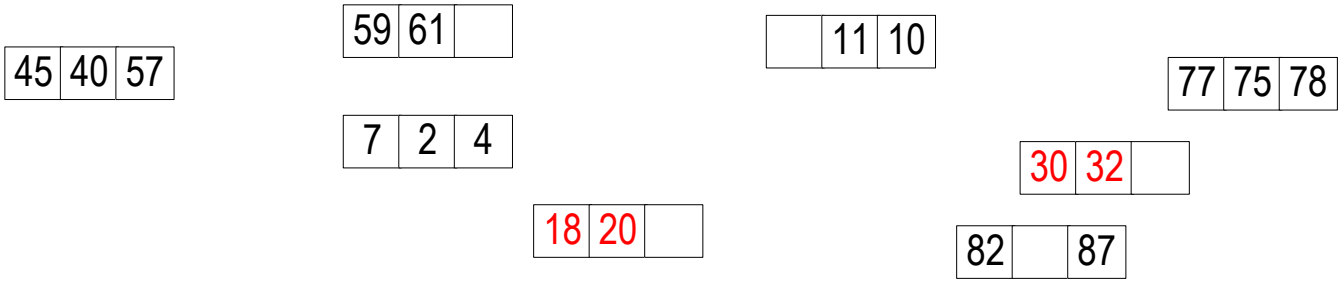
Range Queries In Unclustered Files



- We want all the records with keys in the range 18,…, 32

- We need to read 4 blocks

Range Queries In Clustered Files

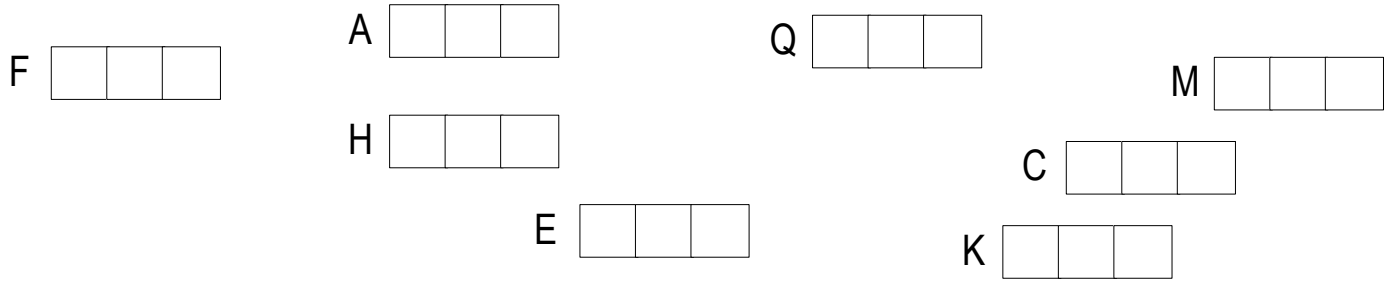


- We want all the records with keys in the range 18,…, 32

- We need to read 2 blocks

Accessing Files With An Index

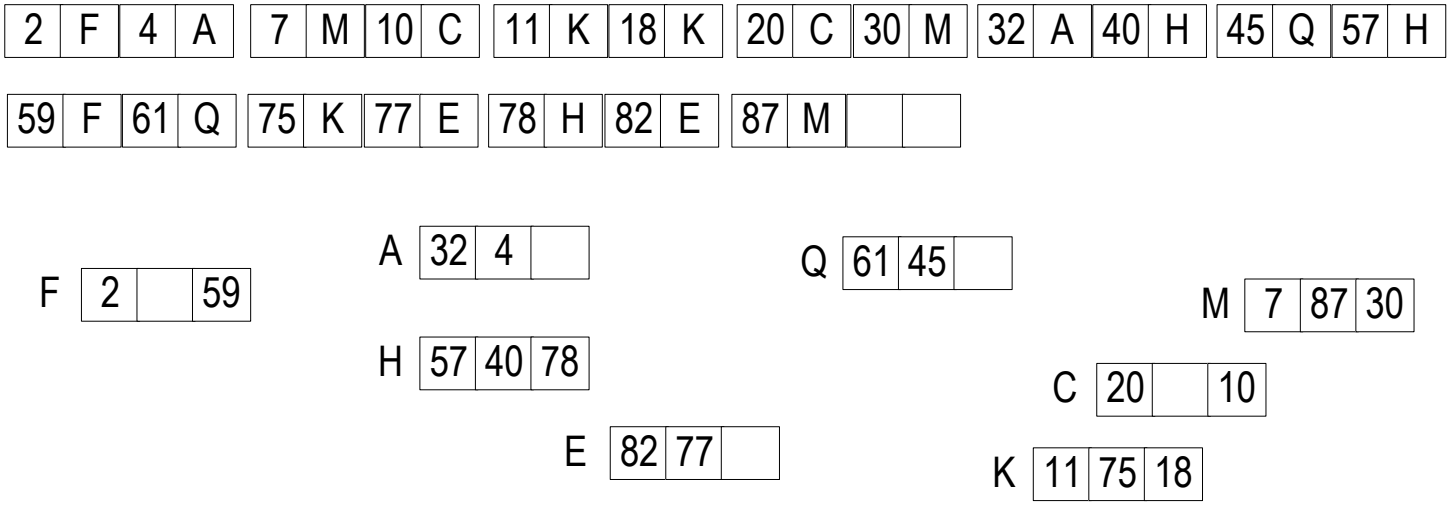
- We will represent the address of a block by a capital letter



- We will start with the toy implementation, when the index will be just a sorted file of keys and blocks in which their associated records are stored in

Dense Index And An Unclustered File

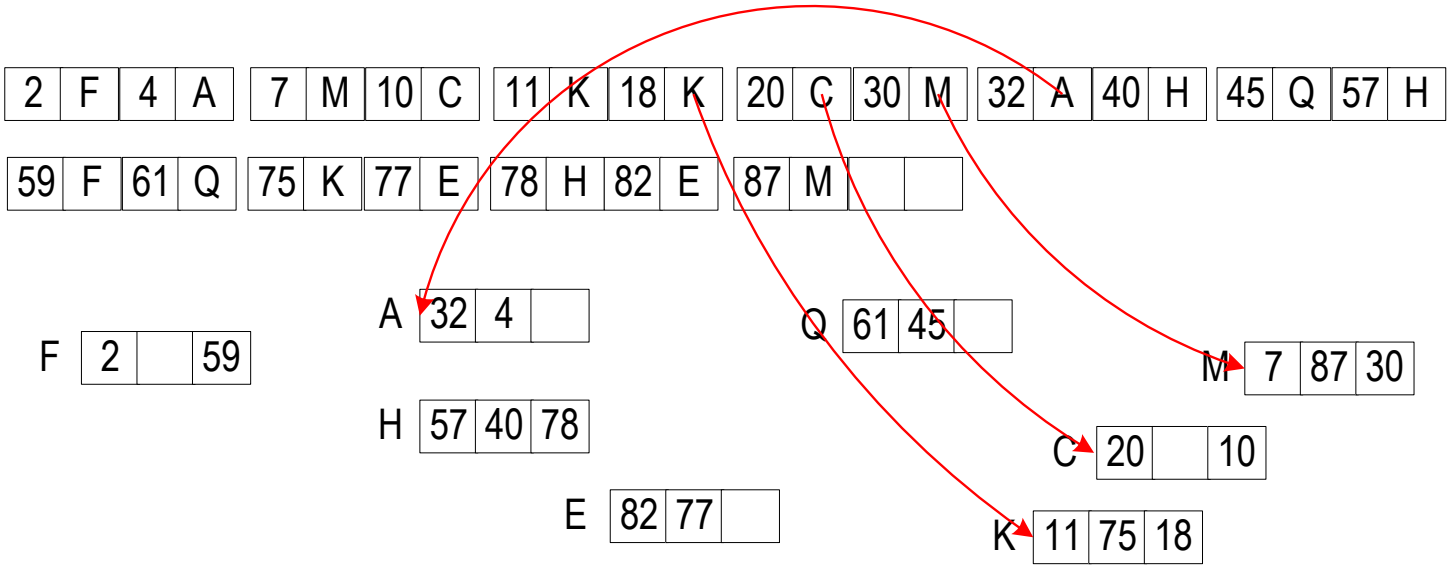
- The index on top and the file on the bottom
- Index: records of pairs (key, block address = pointers)
- ***For every key value in the file there is a record with that key in the index***
- The index is ***dense,*** and it is large
- Several pointers point at a block



- By a binary search through the index, we determine that 61 is in block with address Q

Dense Index And An Unclustered File - Range Query

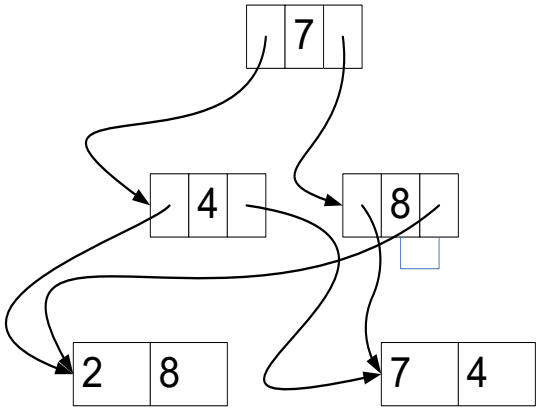
- We want all the records with keys in the range 18,…, 32
- Binary search on the index for key value 18 (or if there is no 18, then . . .)
- Sequentially go through the index until we find (or if there is no 32, then . . .)



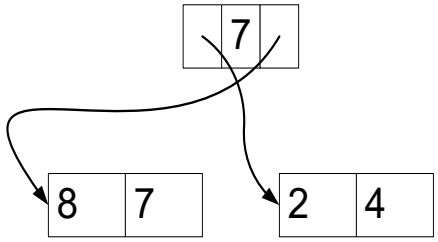
Fewest levels if accessing entire records:
Clustered File + Sparse Index

- Clustered file implies that a block has “related” records
- Sparse index implies that it is efficient to find a block containing “interesting” records, as the index is smaller

- Below, on the right, all records with key greater than 4 are in one block
- Below, on the right, the index has only 1 level so only 1 block of the index is read to know the block of interest



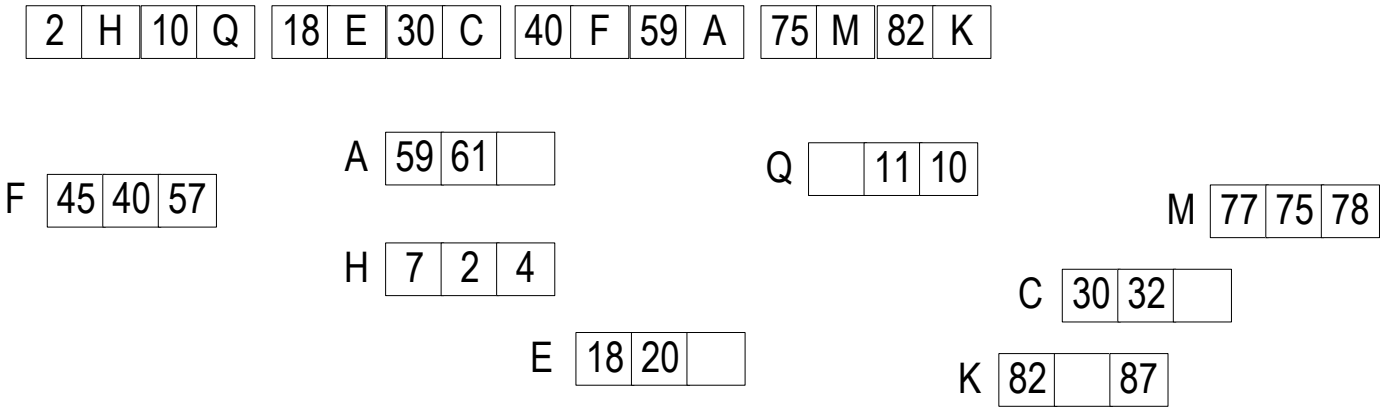
dense index and unclustered file



sparse index and clustered file

Sparse Index And A Clustered File

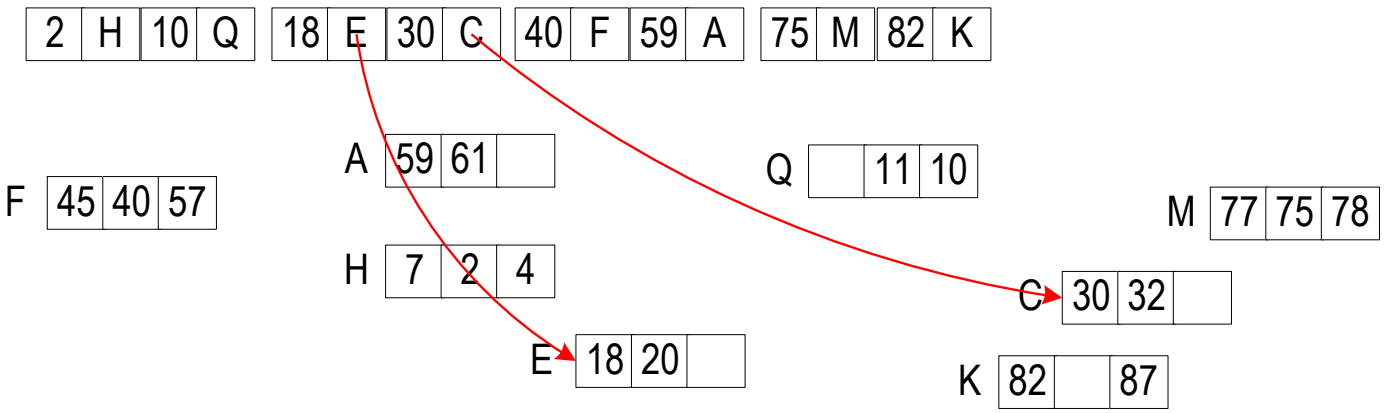
- The index on top and the file on the bottom
- Index: records of pairs (key, block address = pointer)
- ***For every key value in the file that corresponds to the smallest key value in a block there is a record with that key in the index***
- The index is ***sparse,*** and it is small
- A single pointer points at a block



- By a binary search through the index, we determine that 61 is in block with address A (or is not in the file) because 61 is greater or equal to 59 but smaller than 75

Sparse Index And A Clustered File - Range Query

- We want all the records with keys in the range 18,…, 32
- Binary search on the index for key value 18 (or if there is no 18, then . . .)
- Sequentially go through the index until we find (or if there is no 32, then . . .)



What To Choose?

- If possible, have a clustered file
  - But you cannot cluster on every field in the record
- We need a better implementation for the index than a sorted file of blocks.
  - Sorted files are very difficult to maintain
- Our (and the industry's) main implementation for an index is some variant of B^+ tree
  - We will cover it in detail

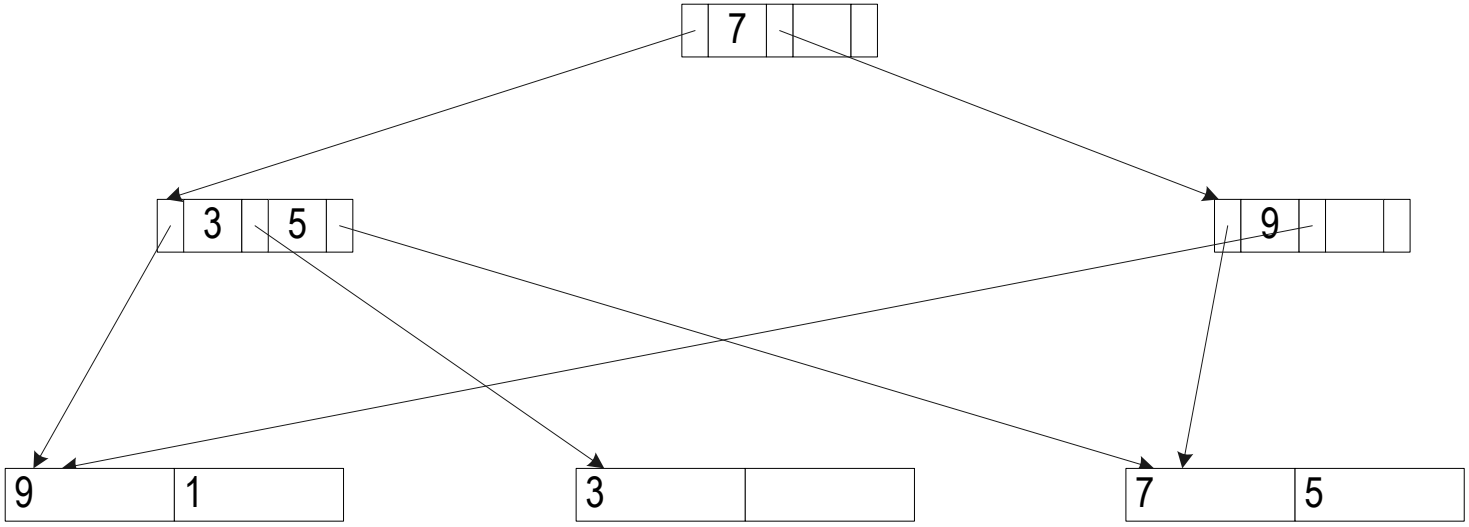
2-3 Trees Revisited

- We again consider a file of records of interest
- Each record has one field, which serves as primary key, which will be an integer
- 2 records can fit in a block
- Sometimes a block will contain only 1 record
- We will consider 2-3 trees as indexes pointing at the records of this file

- The file contains records with indexes: 1, 3, 5, 7, 9

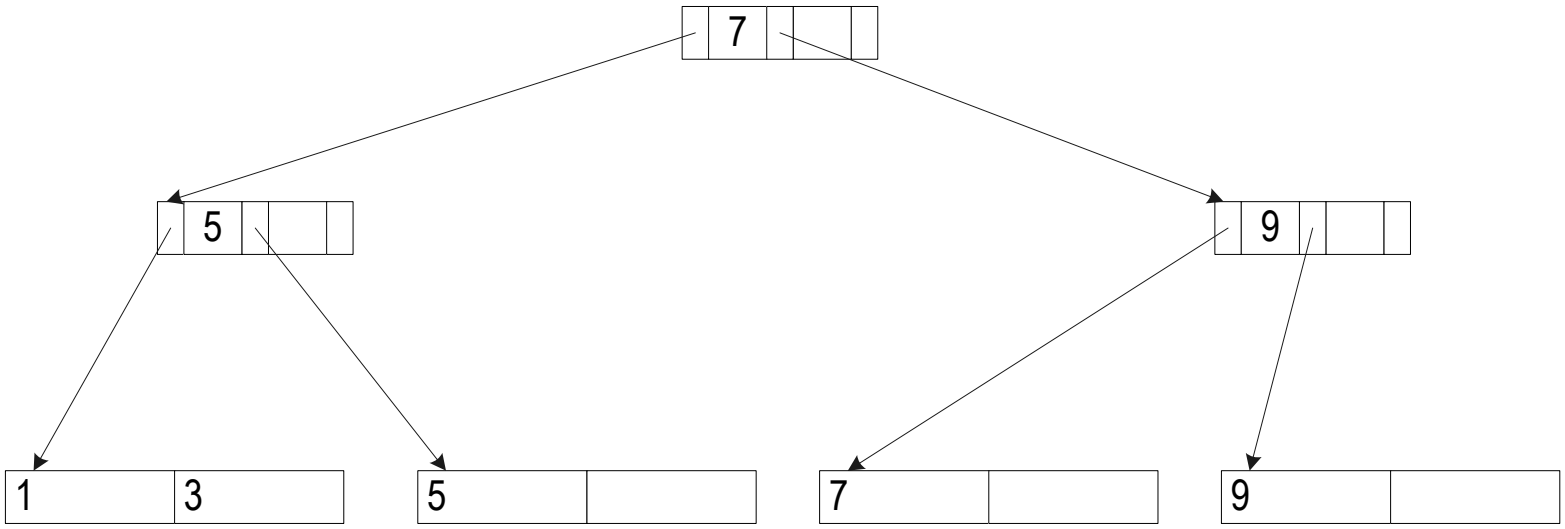
Dense Index and Unclustered File

- Pointers point from the leaf level of the tree to blocks (not records) of the file
  - » Multiple pointers point at a block: 1 for each record in the block
- For each value of the key there is a pointer for such value
  - » Sometimes the value of the key in index is ***explicit***, such as 3
  - » Sometimes the value of the key in index is ***implicit***, such as 1 (there is one value less than 3 so the pointer to the left of 3 points at a block containing a number smaller than 3 (and it is here 1)



Sparse Index and Clustered File

- Pointers point from the leaf level of the tree to blocks (not records) of the file
  - A single pointer points at a block
- For each value of the key that is the smallest in a block there is a pointer for such value
  - Sometimes the key value is explicit, such as 5
  - Sometimes the key value is implicit, such as 1
- Because of clustering we know where 3 is, it is in a block “left” to the block of 5



To Summarize The “Quality” Of Choices In General

- Sparse index and unclustered file: generally bad, cannot easily find records for some keys. So never used
- Dense index and clustered file: generally unnecessarily large index as it is enough to be concerned with the largest key in a block. But still useful (“covering index”).
- Dense index and unclustered file: good, can easily find the record for every key. But can answer certain queries within the index and can have many of them.
- Sparse index and clustered file: best if you need the full record, index is small. But can have only one of them, because can cluster on only one field.

Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

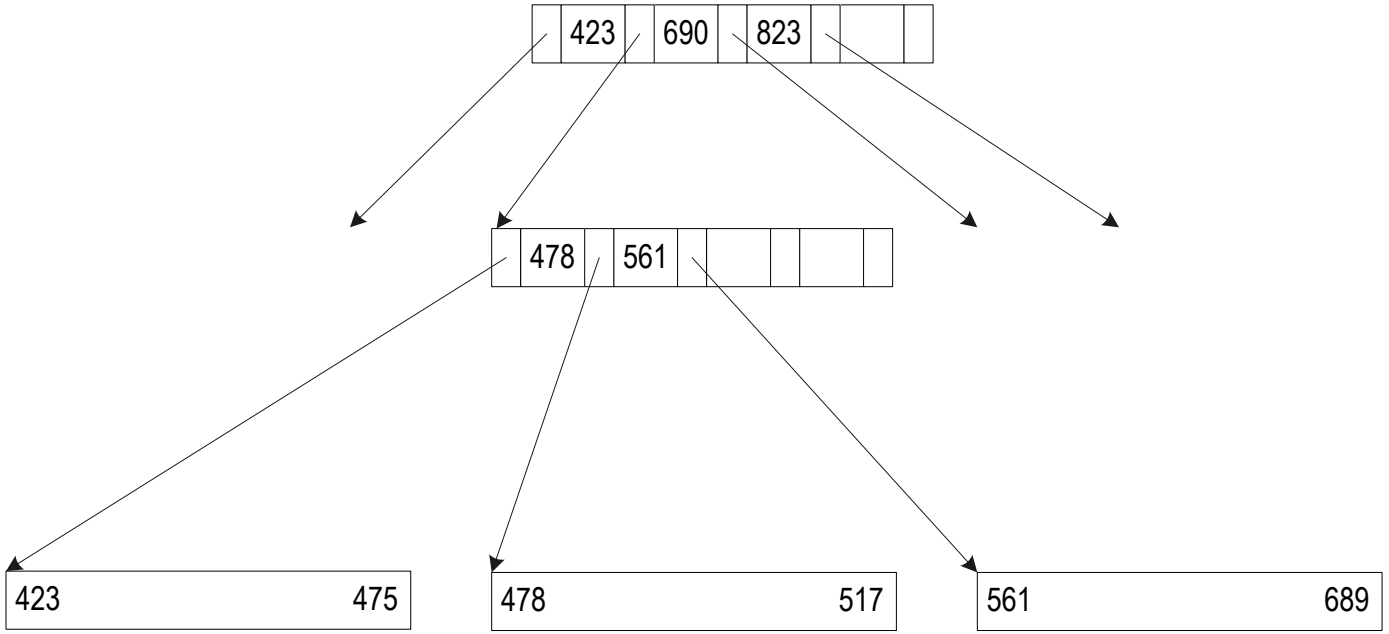
Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

B+ Tree: Generalization of 2-3 Trees

- A B^+-tree is a tree satisfying the conditions
  - » It is rooted (it has a root)
  - » It is directed (order of children matters)
  - » All paths from root to leaves are of same length
  - » For some parameter m:
    - All internal (not root and not leaves) nodes have between ceiling of m/2 and m children
    - The root between 2 and m children
- We ***cannot***, in general, avoid the case of the root having only 2 children
  - » If, for example, we need to split the root as it otherwise would have had m + 1 children, we create a new root with 2 children only

B^+-Trees: Generalization Of 2-3 Trees
"Bottom" Fragment of a 3-5 Tree: m = 5

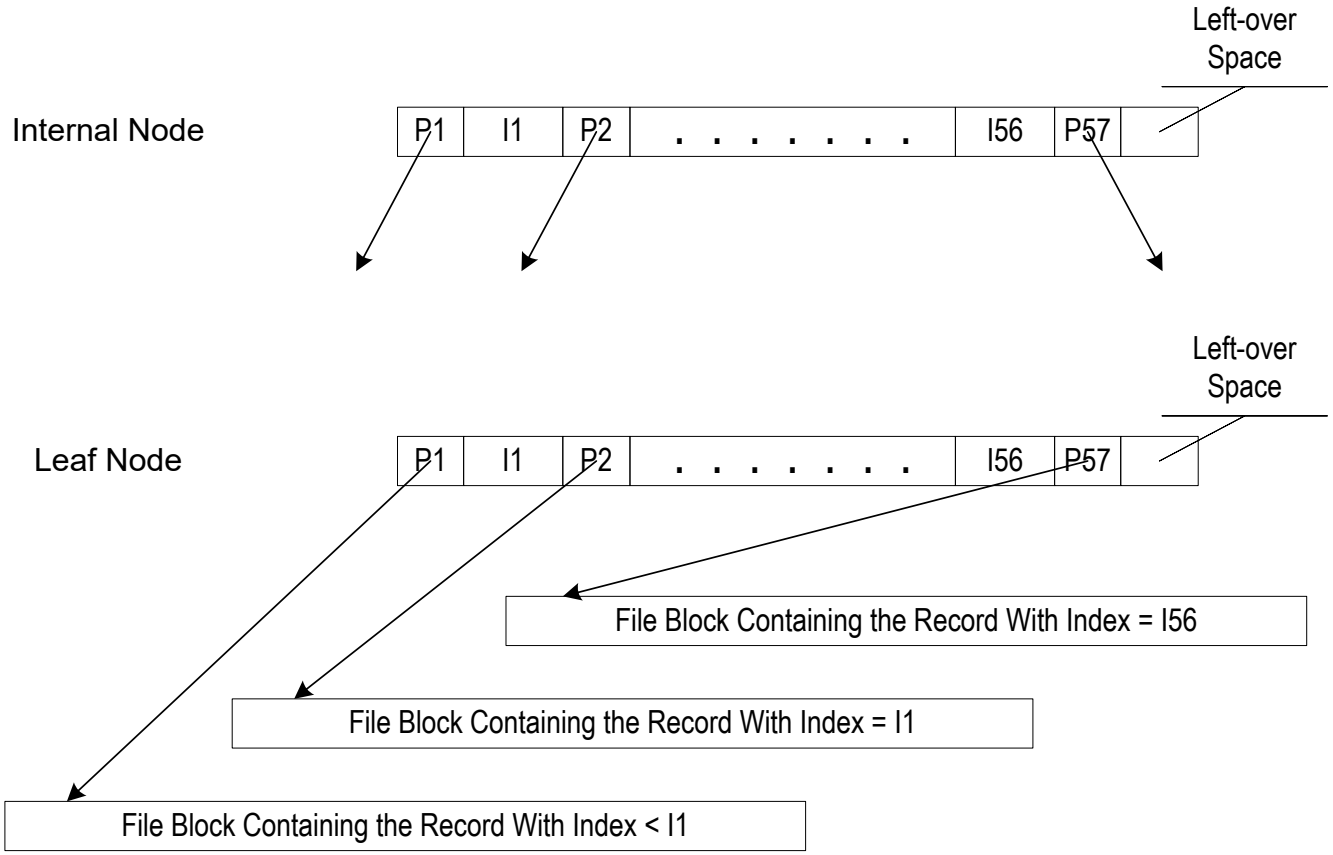


B+ Tree: Generalization of 2-3 Trees

- Each node consists of a sequence (P is ***address*** or ***pointer***, I is ***index*** or ***key***): $P_1,I_1,P_2,I_2,...,P_{m-1},I_{m-1},P_m$
- I_j's form an increasing sequence
- I_j is the ~~largest~~ smallest key value in the leaves in the subtree pointed by ~~P_j~~ P_{j+1}
- Some "implementations" may have slightly different conventions, such as I_j is the smallest key value in the leaves in the subtree pointed by P_{j+1}
- Compression is possible on keys (e.g. perhaps they all share the same prefix). Look at the similarity with tries.

Example: Dense Index & Unclustered File

- m = 57
- Although 57 pointers are drawn, in fact between 29 and 57 pointers come out of each non-root node



B^+-trees: Properties

- Note that a 2-3 tree is a B^+-tree with m = 3
- Important properties
  - For any value of N, and $m \geq 3$, there is always a B^+-tree storing N pointers (and associated key values as needed) in the leaves
  - It is possible to maintain the above property for the given m, while inserting and deleting items in the leaves (thus increasing and decreasing N)
  - Each such operation only O(depth of the tree) nodes need to be manipulated.
- When inserting a pointer at the leaf level, restructuring of the tree may propagate upwards
- At the worst, the root needs to be split into 2 nodes
- The new root above will only have 2 children

B+-trees: Properties

- Depth of the tree is “logarithmic” in the number of items in the leaves
- In fact, this is logarithm to the base at least $\lceil m/2 \rceil$ (ignore the children of the root; if there are few, the height may increase by 1)

- What value of m is best in RAM (assuming RAM cost model)?
- m = 3
- Why? Think of the extreme case where N is large and m = N
  - » You get a sorted sequence, which is not good (insertion is extremely expensive
- “Intermediate” cases between 3 and N are better than N but not as good as 3
- But on disk the situation is very different, as we will see

An Example

- Our application (parameters not necessarily realistic):
  - Disk of 16 GiB (GiB or gibibyte means 2^{30} bytes, GB or gigabyte means 10^9 bytes); very small, but a good example
  - Each block of 512 bytes; rather small, but a good example
  - File of 20 000 000 records
  - Each record of 25 bytes
  - Each record of 3 fields:
    SSN: 5 bytes (packed decimal), name: 14 bytes, salary: 6 bytes
- We want to access the records using the value of SSN
- We want to use a B^+-tree index for this
- Our goal, to minimize the number of block accesses, so we want to derive as much benefit from each block
- ***So, each node should be as big as possible (have many pointers), while still fitting in a single block***
  - Traversing a block once it is in the RAM is free
- Let's compute the optimal value of m

An Example: If We Can Do Our Own Design Of B+ Trees

- There are 16 GiB = 2^{34} bytes on the disk, and each block holds 2^9 = 512 bytes.
- Therefore, there are 2^{34} / 2^9 = 2^{25} blocks, say numbered 0,1,…,2^{25} – 1; each such number can be stored in 25 bits
- Therefore, a block address can be specified in 25 bits; in practice likely determined by the operating system, e.g., 32 or 64 bits,
- We will not not rely on what the DBMS system could do, and we will optimize ourselves
  - The system may allocate 32 bits or 64 bits to a block address even if it is more than necessary
  - That might be better for portability
- That is what you should also do during the course, that is compute how much space you need to store a pointer, following what we will do next
  - Good for practicing the issues

An Example: If We Can Design Our Own B+ Trees (continued)

- We need 25 bits to store a block address
- To compute how many bytes we need to store a block address, we compute $\lceil 25/8 \rceil = \lceil 3.125 \rceil = 4$
  - » We may be "wasting" space, by working on a byte as opposed to a bit level, but simplifying the calculations
  - » It is just a coincidence that this is what a 32-bit operating system may do
- A node in the B+-tree will contain some m pointers (each storing a block address) and m – 1 keys, so what is the largest possible value for m, so a node fits in a block?
- $(m) \times (\text{size of pointer}) + (m - 1) \times (\text{size of key}) \leq \text{size of the block}$
  $(m) \times (4) + (m - 1) \times (5) \leq 512$
  $9m \leq 517;\ m \leq 57.4...$
  m has to be an integer and therefore m = 57

An Example

- Therefore, the root will have between 2 and 57 children
- Therefore, an internal node will have between 29 and 57 children
  - 29 is $\lceil 57/2 \rceil$
- We will examine how a tree can develop in two extreme cases:
  - "narrowest" possible
  - "widest possible

Level	Nodes in the narrow tree	Nodes in the wide tree
1	1	1
2	2	57
3	58	3 249
4	1 682	185 193
5	48 778	10 556 001
6	1 414 562	601 692 057
7	41 022 298	34 296 447 249

B+-tree vs. Binary Search (Simplified Discussion)

- Imagine that we have a clustered file of integers stored in 40,000,000 blocks
  - » Reminder: cluster is like sorted but blocks are not contiguous
- We want to see if an integer of interest is in the file
- Using our B+-tree we can find the block containing the integer (if it exists) in between 6 and 7 block access

- Imagine that you have a sorted file of integers stored in 1,000 blocks
- Using a binary search, you cannot find the block containing the integer (if it exists) in 7 block accesses

- Our B+ tree can be easily maintained and is very efficient
- Our sorted file cannot be easily maintained and is not that efficient

When To Use An Index (Simplified Discussion)

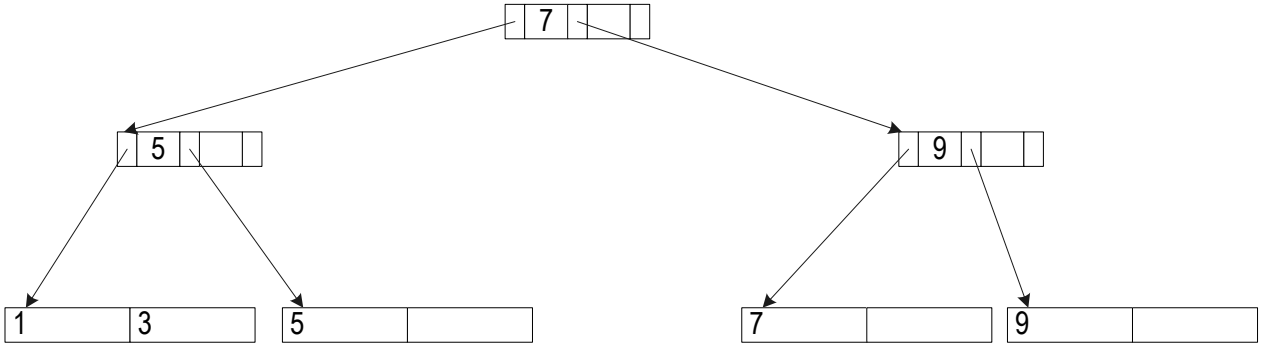
- Use an index if the result would require accessing a small number of blocks
- Otherwise, may as well examine the whole file
- Imagine a file about large number of employees with fields SSN, Date of Birth, Sex
- The file is clustered on SSN, but we have indexes (we will see later how) on SSN, Date of Birth, Sex
- To find all men, index likely useless
- To find all employees with Date of Birth of 1973-02-23, index could be useful
- To find all employees with SSN between 123456788 and 12345679 likely useful

An Example - Maintaining A Clustered File

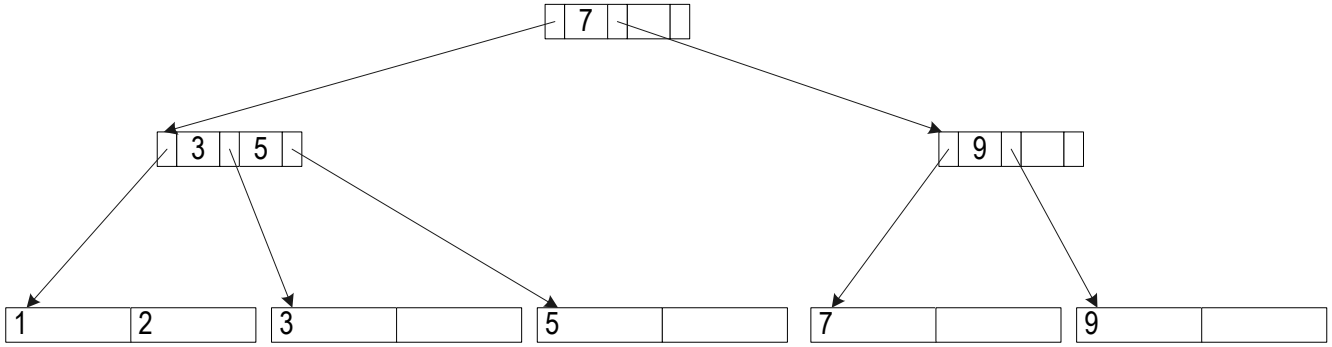
- Assume that we ***are*** permitted to reorganize the data file and therefore we decide to cluster it
- We will in fact treat the file as the lowest level of the tree
- The tree will have two types of nodes:
  - » nodes containing indexes and pointers as before
  - » nodes containing the data file
- ***The leaf level of the tree will be the file***
- Parameters
  - » The B^+ tree is a 2-3 tree
  - » A block of the file contains between 1 and 2 records

An Example - Insertion Of Record And Splitting A Block

- Old structure



- New structure after record with key 2 is inserted into the clustered file



- In a worse case, “splitting” may propagate up to the root

Our Design Was Not Portable

- In designing our tree, we used the size of the disk to compute the parameter *m*
- It was based on both the size of the key and the size of the pointer
- This allowed us to build very space-efficient trees
- But the size of the pointer was based on the number of blocks in the disk
- So the tree may fail to work if we move it to a larger disk, where the size of the pointer may need to be bigger
- So to be safe, the space allocated to the pointer could be large enough to accommodate any disk supported by the operating system
- So in practice, 32 bits (4 bytes) for a 32-bit operating system and 64 bits (8 bytes) for a 64-bit operating system
- Then the only parameter that is application dependent is the size of the key

B^+-tree vs. Binary Search (expanded discussion)

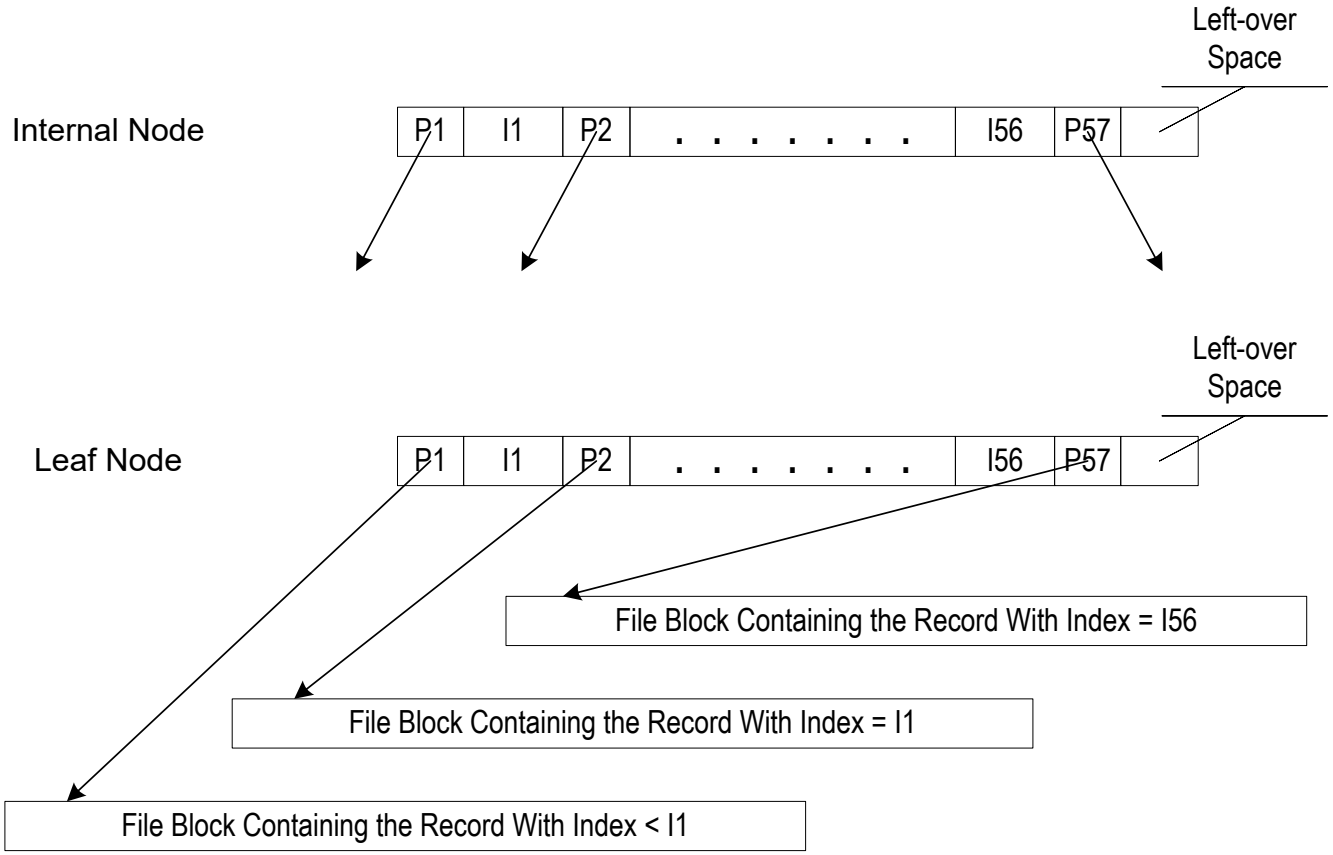
- For simplicity, assume that the leaves are just integers
- If can fit 1 gig in memory but there are 1 trillion integers, then binary search tree will have 10 levels on disk.
- B tree with fanout of 1000, just one or two.

An Example

- Therefore, the root will have between 2 and 57 children
- Therefore, an internal node will have between 29 and 57 children
  - 29 is the ceiling of 57/2
- We will examine how a tree can develop in two extreme cases:
  - “narrowest” possible
  - “widest possible
- To do this, we need to know how the underlying data file is organized, and whether we can reorganize it by moving records in the blocks as convenient for us
- We will assume for now that
  - the data file is already given and,
  - it is not clustered on the key (SSN) and,
  - that we cannot reorganize it but have to build the index on “top of it”

An Example (continued)

- In fact, between 2 and 57 pointers come out of the root
- In fact, between 29 and 57 pointers come out of a non-root node (internal or leaf)



An Example

- We obtained a ***dense*** index, where there is a pointer "coming out" of the ***index file*** for every existing key in the data file
- Therefore we needed a pointer "coming out" of the ***leaf level*** for every existing key in the data file, that is for every record
- We must get a total of 20 000 000 pointers "coming out" in the lowest level
- In the narrow tree, other than at the root, every node has 29 pointers "coming out of it" (29 children)
- In the narrow tree, the root has 2 pointers "coming out of it" (2 children)
- In the wide tree, every node has 57 pointers coming out of it (57 children)

An Example (continued)

Level	Nodes in a narrow tree	Nodes in a wide tree
1	1	1
2	2	57
3	58	3 249
4	1 682	185 193
5	48 778	10 556 001
6	1 414 562	
7	41 022 298	

- We must get a total of 20 000 000 pointers “coming out” in the lowest level.
- For the narrow tree, 6 levels is too much
  - » If we had 6 levels there would be at least 1 414 562 × 29 = 41 022 298 pointers, but there are only 20 000 000 records
  - » So it must be 5 levels, and some nodes must have more than 29 children
- In search, there is one more level, but “outside” the tree, the file itself

An Example (continued)

Level	Nodes in a narrow tree	Nodes in a wide tree
1	1	1
2	2	57
3	58	3 249
4	1 682	185 193
5	48 778	10 556 001
6	1 414 562	
7	41 022 298	

- For the wide tree, 4 levels is too little
  - If we had 4 levels there would be at most 185 193 × 57 = 10 556 001 pointers, but there are 20 000 000 records
  - So it must be 5 levels
- Not for the wide tree we round up the number of levels
- Conclusion: it will be 5 levels exactly in the tree (accident of the example; in general could be some range)
- In search, there is one more level, but “outside” the tree, the file itself

An Example (continued)

- How does B^+ compare with a sorted file?
- If a file is sorted, it fits in at least 20 000 000 / 20 = 1 000 000 blocks, therefore:

  Binary search will take ceiling of $\log_2(1\ 000\ 000) = 20$ block accesses

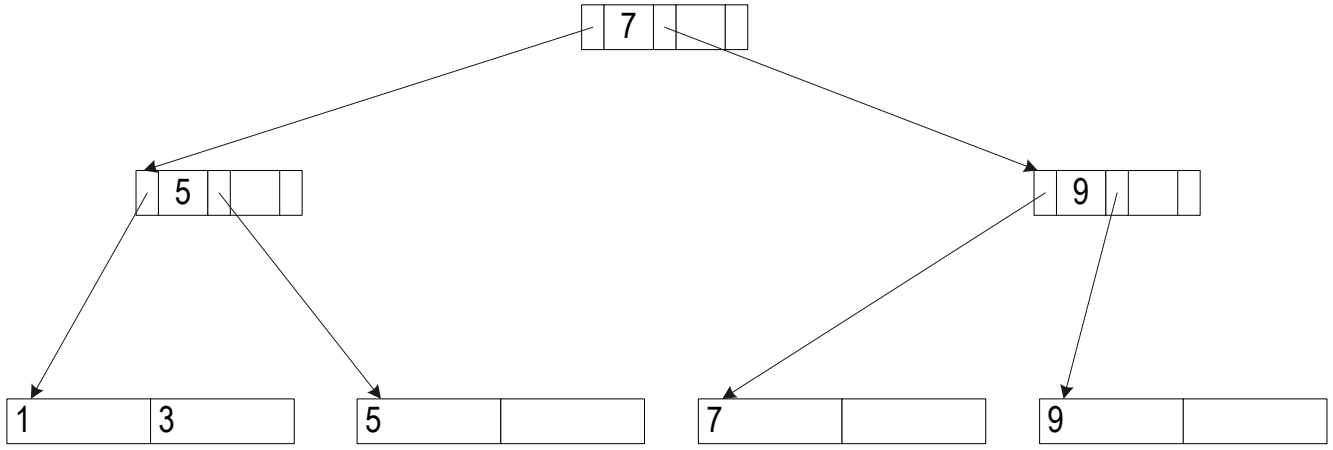
- As the narrow tree had 5 levels, we needed at most 6 block accesses (5 for the tree, 1 for the file)
  - » We say “at most” as in general there is a difference in the number of levels between a narrow and a wide tree, so it is not the case of our example in which we could say “we needed exactly 6 block accesses”
  - » But if the page/block size is larger, this will be even better
- A 2-3 tree, better than sorting but not by much

Finding 10 Records

- So how many block accesses are needed for reading, say 10 records?
- If the 10 records are “random,” then we need possibly close to 60 block accesses to get them, 6 accesses per record
  - » In fact, somewhat less, as likely the top levels of the B-tree are cashed and therefore no need to read them again for each search of one of the 10 records of interest
- Even if the 10 records are consecutive, then as the file is not clustered, they will still (likely be) in 10 different blocks of the file, ***but “pointed from” 1 or 2 leaf nodes of the tree***
  - » We do not know exactly how much we need to spend traversing the index, worst case to access 2 leaves we may have 2 completely disjoint paths starting from the root, but this is unlikely
  - » In addition, maybe the index leaf blocks are chained so it is easy to go from a leaf to the following leaf
  - » So in this case, seems like 16 or 17 block accesses in most cases

An Example

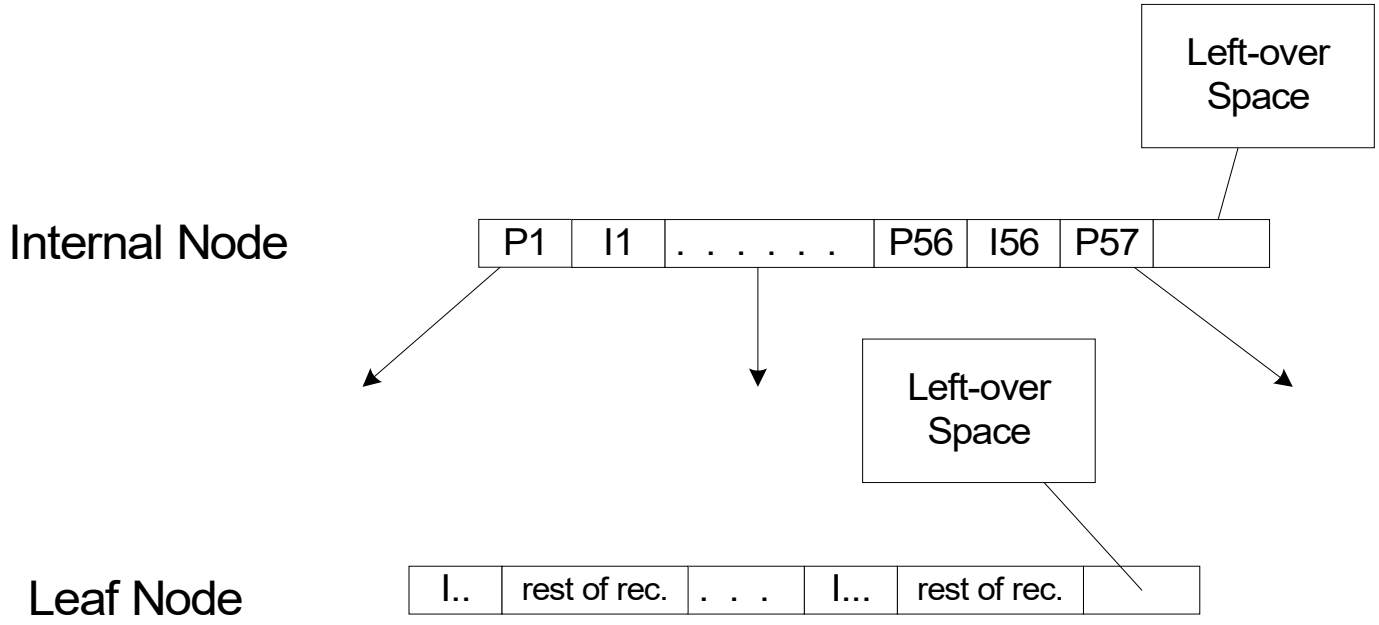
- We will now assume that we ***are*** permitted to reorganize the data file and therefore we decide to cluster it
- We will in fact treat the file as the lowest level of the tree
- The tree will have two types of nodes:
  - » nodes containing indexes and pointers as before
  - » nodes containing the data file
- ***The leaf level of the tree will in fact be the file***
- We have seen this before



An Example (continued)

- For our example, at most 20 records fit in a block
- Each block of the file will contain between 10 and 20 records
- So the bottom level is just like any node in a B-tree, but because the "items" are bigger, the value of *m* is different, it is *m* = 20
- So when we insert a record in the right block, there may be now 21 records in a block and we need to split a block into 2 blocks
  - This may propagate up
- Similarly for deletion
- We will examine how a tree can develop in two extreme cases:
  - "narrowest" possible
  - "widest possible

An Example (continued)



- For each leaf node (that is a block of the file), there is a pointer associated with the smallest key value from the key values appearing in the block; the pointer points from the level just above the leaves of the structure

An Example (continued)

Level	Nodes in a narrow tree	Nodes in a wide tree
1	1	1
2	2	57
3	58	3 249
4	1 682	185 193
5	48 778	10 556 001
6	1 414 562	
7	41 022 298	

- Let us consider the trees, without the file level for now
- For the narrow tree and worst case of clustering (file blocks only half full: 10 records per block), we must get a total of 20 000 000 / 10 = 2 000 000 pointers in the lowest level
- So we need 2 000 000 / 29 = 68 965.5... nodes
- So it is between level 5 and 6, so must be 5
  - » Rounding down as before

An Example (continued)

Level	Nodes in a narrow tree	Nodes in a wide tree
1	1	1
2	2	57
3	58	3 249
4	1 682	185 193
5	48 778	10 556 001
6	1 414 562	
7	41 022 298	

- For the wide tree and best case of clustering (file blocks completely full: 20 records per block), we must get a total of 20 000 000 / 20 = 1 000 000 pointers in the lowest level
- so we need 1 000 000 / 57 = 17 543.8… nodes
- So it is between level 3 and 4, so must be 4
  - » Rounding up as before
- Conclusion: it will be between 5 + 1 = 6 and 4 +1 = 5 levels (including the file level), with this number of block accesses required to access a record.

Finding 10 Records

- So how many block accesses are needed for reading, say 10 records?
- If the 10 records are “random,” then we need possibly close to 60 block accesses to get them, 6 accesses per record.
- ***In practice (which we do not discuss here), somewhat fewer, as it is likely that the top levels of the B-tree are cached and therefore no need to read them again for each search of one of the 10 records of interest***

Finding 10 Records (continued)

- If the 10 records are consecutive, and if there are not too many null records in the file blocks, then all the records are in 2 blocks of the file
- So we only need to find these 2 blocks, and the total number of block accesses is between 7 and a “little more”

- In general, we do not know exactly how much we need to spend traversing the index
  - » In the worst case in order to access two blocks we may have two completely different paths starting from the root
- But maybe the index leaf blocks are chained so it is easy to go from leaf to leaf

An Example

- To reiterate:

- The first layout resulted in an unclustered file
- The second layout resulted in a clustered file

- Clustering means: “logically closer” records are “physically closer”
- More precisely: ***as if the file has been sorted with blocks not necessarily full and then the blocks were dispersed***

- So, for a range query the second layout is much better, as it decreases the number of accesses to the file blocks by a factor of between 10 and 20

Our Design Was Not Portable

- In designing our tree we used the size of the disk to compute the parameter *m*
- It was based on both the size of the key and the size of the pointer
- This allowed us to build very space-efficient trees
- But the size of the pointer was based on the number of blocks in the disk
- So the tree may fail to work if we move it to a larger disk, where the size of the pointer may need to be bigger
- So to be safe, the space allocated to the pointer should be large enough to accommodate any disk supported by the operating system
- So in practice, 32 bits (4 bytes) for a 32-bit operating system and 64 bits (8 bytes) for a 64-bit operating system
- Then the only parameter that is application dependent is the size of the key

Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

Hashing on a Disk

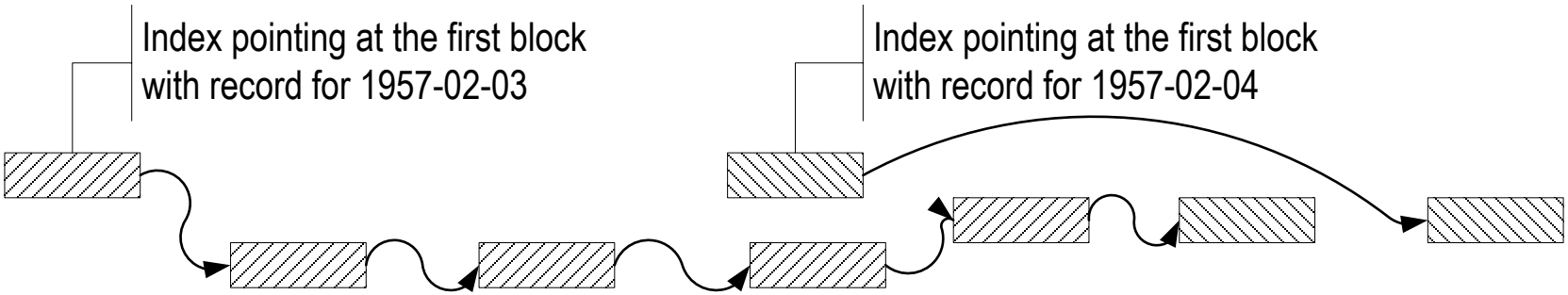
- Same idea
- Hashing as in RAM, but now choose B, so that “somewhat fewer” than 20 key values from the file will hash on a single bucket value
- Then, from each bucket, “very likely” we have a linked list consisting of only one block
- But how do we store the bucket array? Very likely in memory, or in a small number of blocks, which we know exactly where they are
- So, we need about 2 block accesses to reach the right record “most of the time”

Agenda

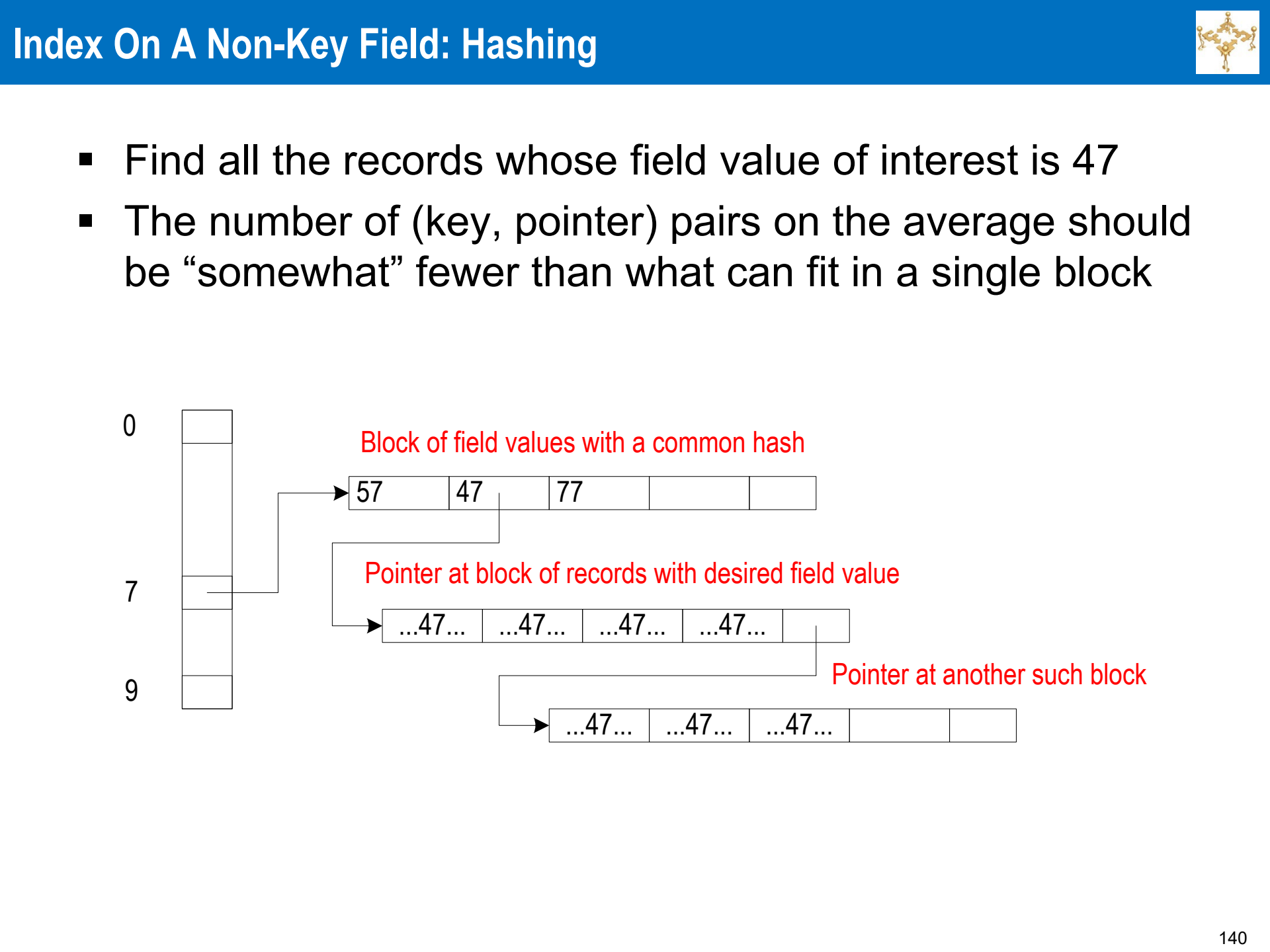
1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

Index on a Non-Key Field

- So far we consider indexes on “real” keys, i.e., for each search key value there was at most one record with that value
- One may want to have an index on non-key value, such as DateOfBirth
  - » Choosing the right parameters is not as easy, as there is an unpredictable, in general, number of records with the same key value.
- One solution, using index, an index find a “header” pointing at the structure (e.g., a sequence of blocks), containing records of interest
- Very efficient access to all the records of interest



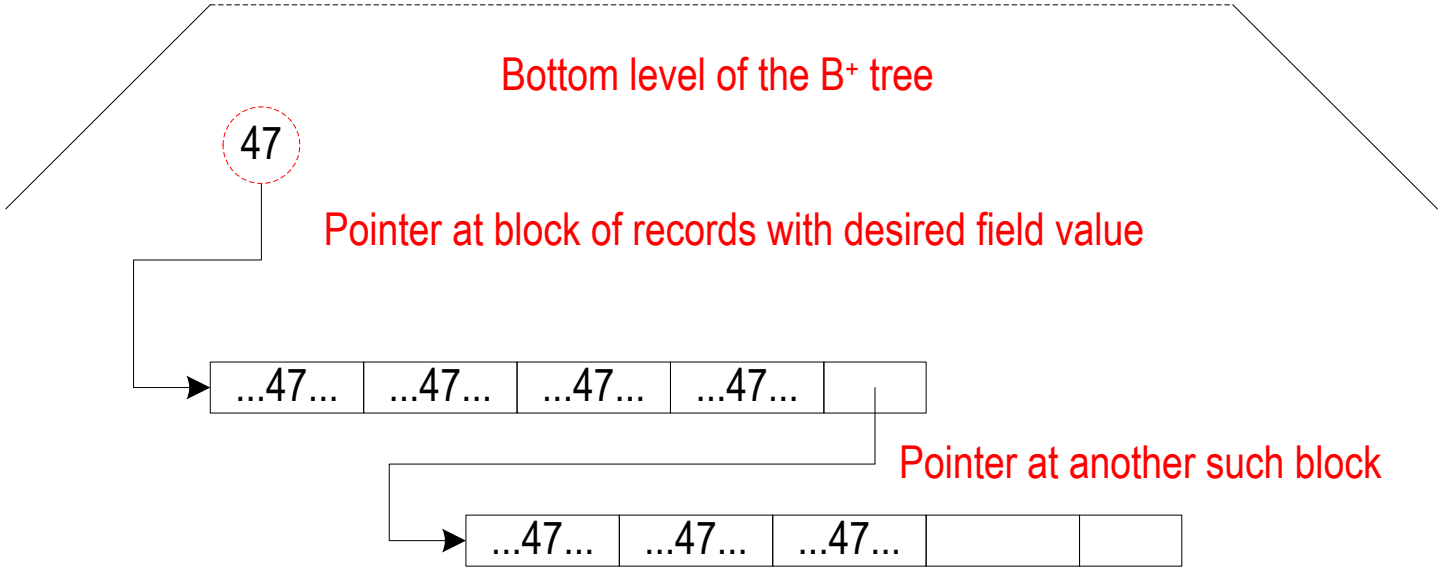
Index On A Non-Key Field: Hashing



- Find all the records whose field value of interest is 47
- The number of (key, pointer) pairs on the average should be “somewhat” fewer than what can fit in a single block

Index On A Non-Key Field: B+-tree

- Find all the records whose field value of interest is 47



Agenda

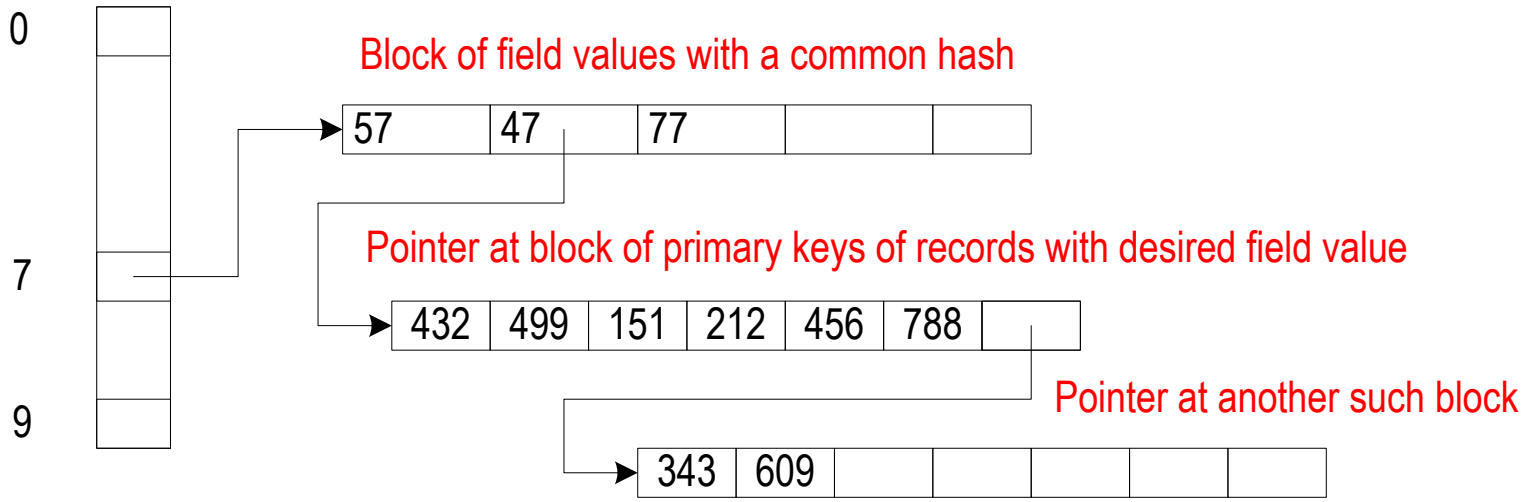
1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

Primary vs. Secondary Indexes

- In the context of clustered files, we discussed a primary index, that is the one according to which a file is physically organized, say SSN
- But if we want to efficiently answer queries such as:
  - Get records of all employees with salary between 35,000 and 42,000
  - Get records of all employees with the name: “Ali”
- For this we need more indexes, if we want to do it efficiently
- They will have to be put “on top” of our existing file organization.
  - The primary file organization was covered above, it gave the primary index
- We seem to need to handle range queries on salaries and non-range queries on names
- We will generate secondary indexes

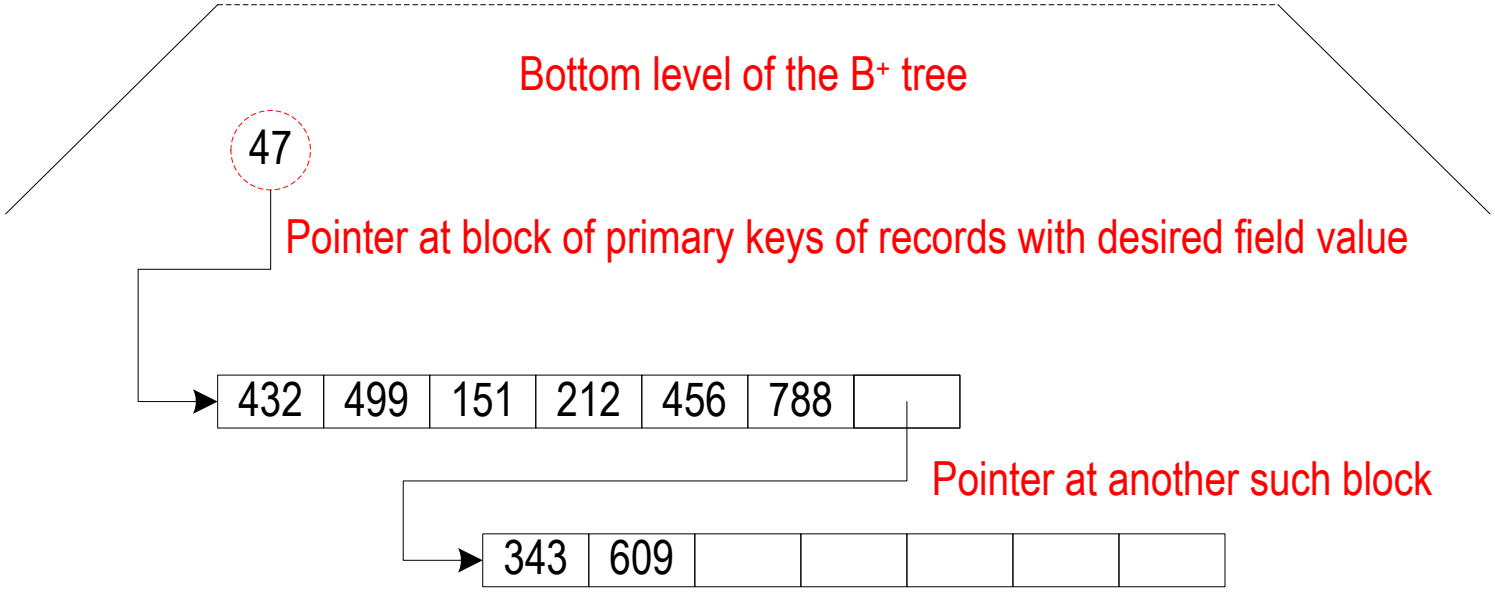
Secondary Index: Hashing

- Find the primary keys of all the records whose field value of interest is 47



Secondary Index: B+ Trees

- Find the primary keys of all the records whose field value of interest is 47



Secondary Index on Salary

- The file's primary organization is on SSN (social security number)
- So, it is "very likely" clustered on SSN, and therefore it cannot be clustered on SALARY
- Create a new file of variable-size records of the form:

  (SALARY)(SSN)*

  For each existing value of SALARY, we have a list of all SSN of employees with this SALARY.
- This is clustered file on SALARY
- It is good if the primary index field is short because that makes the secondary index shorter.
  - » Here if SSN is short then collection of all (SALARY)(SSN)* is small
- Surrogate keys are likely to be short

Secondary Index on Salary (continued)

- Create a B^+-tree index for this file
- Variable records, which could span many blocks are handled similarly to the way we handled non-key indexes
- This tree together with the new file form a secondary index on the original file
- Given a range of salaries, using the secondary index we find all SSN of the relevant employees
- Then using the primary index, we find the records of these employees
  - But they are unlikely to be contiguous, so may need "many" block accesses.
- In fact using the index may sometimes be less efficient than scanning the table (e.g. all bonds having a certain interest rate).

Secondary Index on Name

- The file's primary organization is on SSN
- So, it is "very likely" clustered on SSN, and therefore it cannot be clustered on NAME
- Create a file of variable-size records of the form:

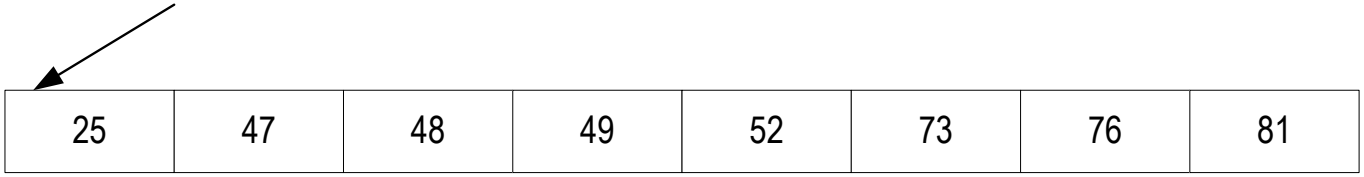
  (NAME)(SSN)$^+$

  For each existing value of NAME, we have a list of all SSN of employees with this NAME.
- Create a hash table for this file
- This table together with the new file form a secondary index on the original file
- Given a value of name, using the secondary index we find all SSN of the relevant employees
- Then using the primary index, we find the records of these employees

A Fleshed-Out Example with B+ Trees

A Fleshed Out Example with B+ Trees: Implementing a Dictionary

- We start by reviewing a “standard” search tree for accessing records through their primary keys

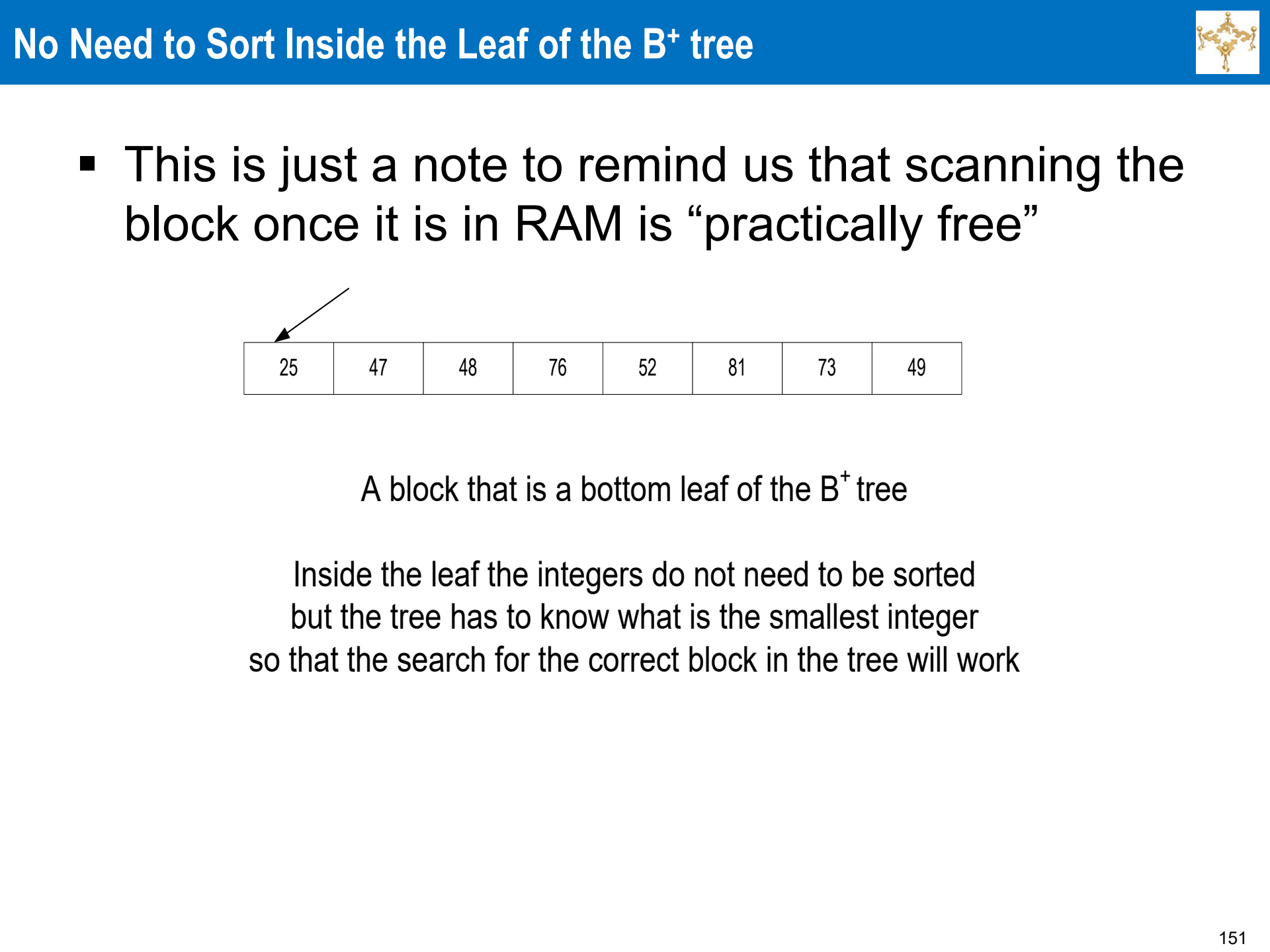


A block that is a bottom leaf of the B^+ tree

We put as many integers as can fit in the block

Insertion and deletion of integers just like in a 2-3 tree but the parameters are need to be adjusted to the block size

No Need to Sort Inside the Leaf of the B+ tree



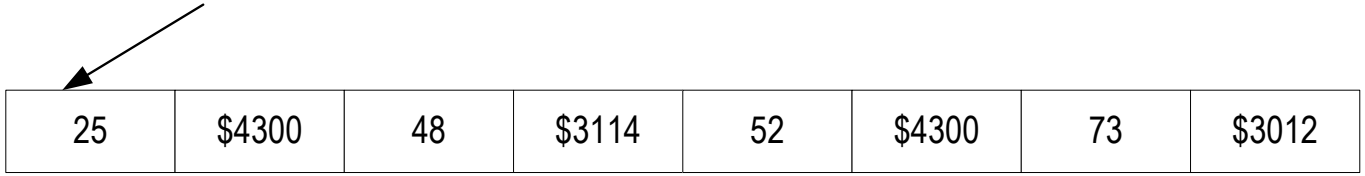
- This is just a note to remind us that scanning the block once it is in RAM is “practically free”

A block that is a bottom leaf of the B+ tree

Inside the leaf the integers do not need to be sorted
but the tree has to know what is the smallest integer
so that the search for the correct block in the tree will work

General Records - Example: ID and Salary

- To review how to store records, and not just individual items



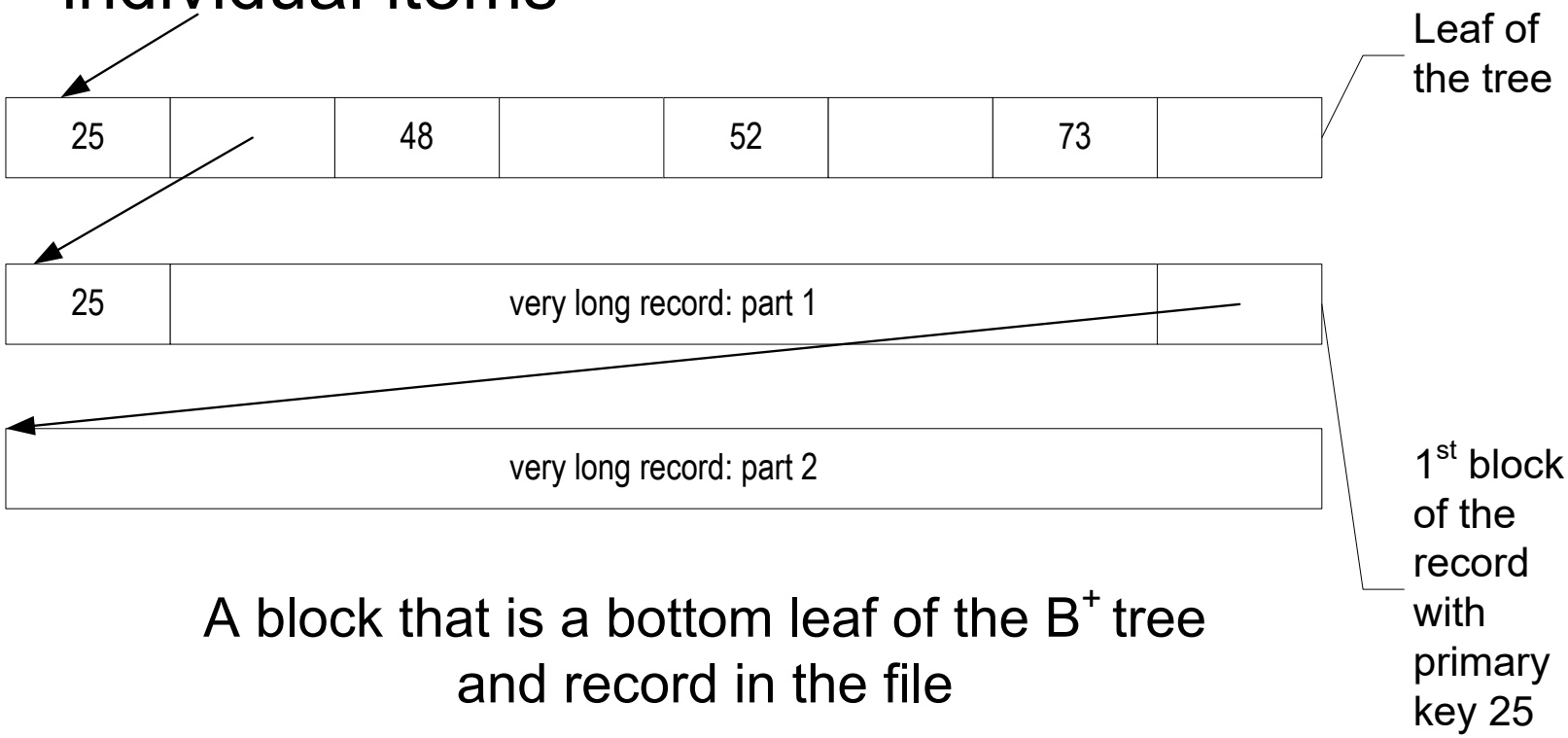
A block that is a bottom leaf of the B^+ tree

Records are short: fixed or variable but a record does not need more than 1 block

In our example, the records have 2 fields
ID (primary key) and Salary

Long Records - Primary Key and Other Fields

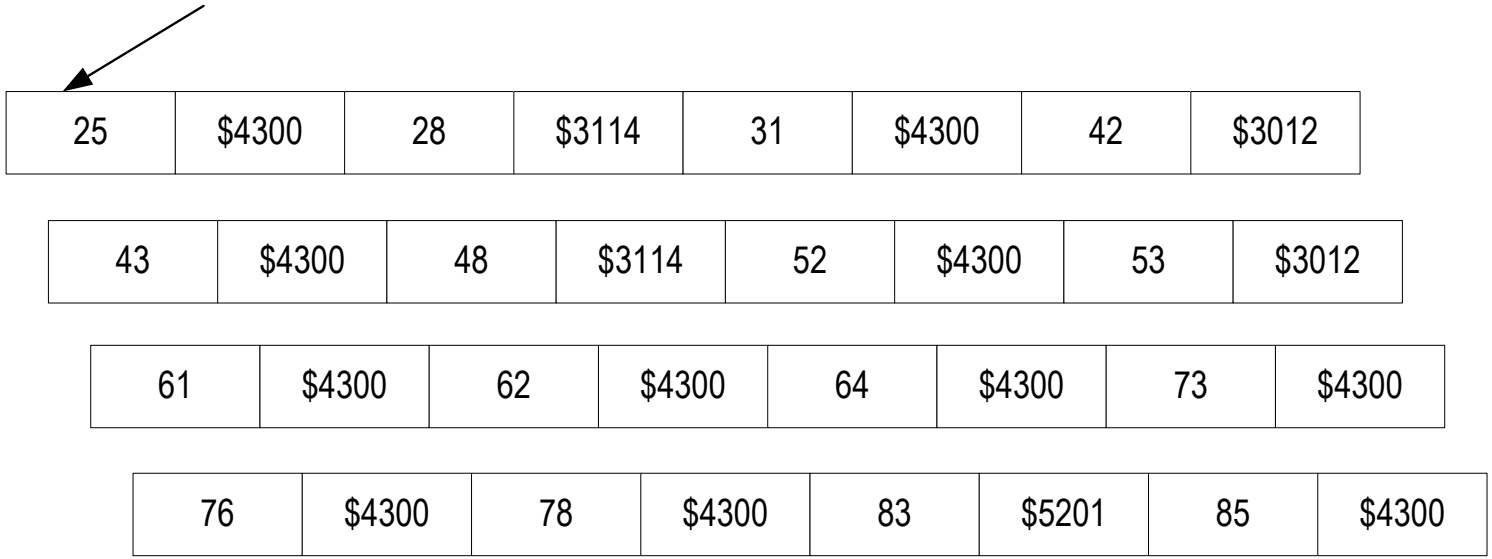
- To review how to store records, and not just individual items



A block that is a bottom leaf of the B^+ tree and record in the file

Records are long: fixed or variable
but a record may need more than 1 block
they are not stored in the B^+ tree

Secondary Index / Inverted Index (The Primary Tree Is Shown)

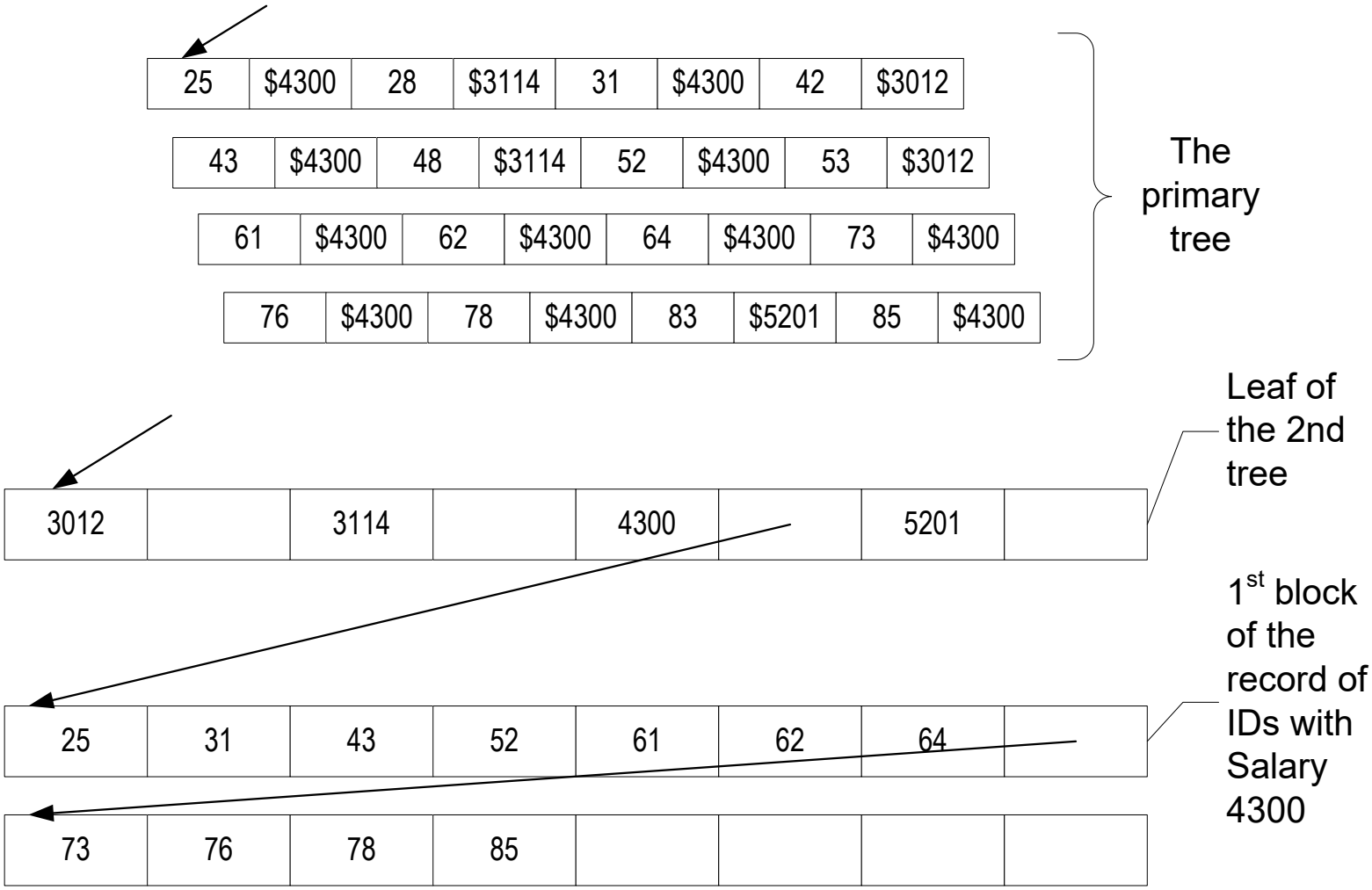


4 consecutive blocks (leaves) at the bottom of the B^+ tree

We will want to quickly find all the people earning a specific Salary (or range of Salaries)

We will create a 2^{nd} tree:
for every salary value we will have a variable record with all the IDs of the people with that salary

Secondary Index / Inverted Index



Once the IDs of people with Salary = $4300 are found, a search for the record with this primary key can be done using the primary tree

A Little More …

Index on Several Fields (Compound Indexes)

- In general, a single index can be created for a set of columns
- So if there is a relation R(A,B,C,D), an index can be created for, say (B,C)
- This means that given a specific value or range of values for (B,C), appropriate records can be easily found
- This is applicable for all type of indexes

Symbolic vs. Physical Pointers

- Our secondary indexes were symbolic
  Given value of SALARY or NAME, the “pointer” was primary key value
- Instead we could have physical pointers
  (SALARY)(block address)* and/or (NAME)(block address)*
- Here the block addresses point to the actual physical locations of the blocks containing the relevant records
- Access more efficient as we skip the primary index
- Maintaining more difficult
  - » If primary index needs to be modified (new records inserted, causing splits, for instance) need to make sure that physical pointers properly updated
  - » And moving to another disk means rebuilding everything as physical pointers will change

Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

How About SQL?

- Most commercial database systems implement indexes
- Assume relation R(A,B,C,D) with primary key A
- Some typical statements in commercial SQL-based database systems
  - CREATE UNIQUE INDEX index1 ON R(A); unique implies that this will be a “real” key, just like UNIQUE in SQL DD

    “index1” is the name we chose for the index
  - CREATE INDEX index2 ON R(B ASC,C)
  - CREATE CLUSTERED INDEX index3 on R(A)
  - DROP INDEX index4
- Generally, some variant of B^+ tree is used (not hashing)
  - In fact, generally you cannot specify whether to use B^+ -trees or hashing
  - Most systems allow you to specify B-tree or Hash or even more exotic indexes like R-trees or bitmaps

- Generally:
  - When a PRIMARY KEY is declared the system generates a primary index using a B^+-tree

    Useful for retrieval and making sure that this indeed is a primary key (two different rows cannot have the same key); in fact two identical rows are not permitted by Oracle also
  - When UNIQUE is declared, the system generates a secondary index using a B^+-tree

    Useful as above
- It is possible to specify hash indexes using, so called, HASH CLUSTERs
  - Useful for fast retrieval, particularly on equality (or on a very small number of values)

Bitmap Index

- It is possible to specify ***bit-map indexes***, particularly useful for querying databases (not modifying them—that is, useful in an OLAP "warehouse" environment)
- Assume we have

ID	Sex	Region	Salary
10	M	W	450
31	F	E	321
99	F	W	450
81	M	S	356
62	M	N	412
52	M	S	216
47	F	N	658
44	F	S	987
51	F	N	543
83	M	S	675
96	M	E	412
93	F	E	587
30	F	W	601

Bitmap Index (continued)

- Bitmap index table listing whether a row has a value (F – female, M – mail, E– east, N– north, S – south, W – west)

ID	F	M	E	N	S	W
10	0	1	0	0	0	1
31	1	0	1	0	0	0
99	1	0	0	0	0	1
81	0	1	0	0	1	0
62	0	1	0	1	0	0
52	0	1	0	0	1	0
47	1	0	0	1	0	0
44	1	0	0	0	1	0
51	1	0	0	1	0	0
83	0	1	0	0	1	0
96	0	1	1	0	0	0
93	1	0	1	0	0	0
30	1	0	0	0	0	1

Bitmap Index (continued)

- Useful when cardinality of an attribute is small:
  - This means: the number of distinct values is small
  - Which implies that the number of columns in the index is small
- Example: Find all IDs for people for F and S; i.e., people for which Sex = “F” and Region = “S”
  - Just do Boolean AND on the columns labeled F and S; where you get 1, you look at the ID, and it satisfies this condition
- Example: How many males live in the northern region
  - Just do Boolean AND and count the number of 1’s
- Can use Boolean OR, NEGATION, if needed
- The idea is to make the bitmap index table so small that it can fit in the RAM (or still be relatively small)
  - So operations are very efficient
- But, the ID field could be large
- Solution: two tables, each smaller than the original one
  - And maybe store only the bitmap table in RAM

Optimization: Maintain 2 Smaller Structures with Implicit Row Pairing

ID
1000000040056768782345
3100000040056768782345
9900000040056768782345
8100000040056768782345
6200000040056768782345
5200000040056768782345
4700000040056768782345
4400000040056768782345
5100000040056768782345
8300000040056768782345
9600000040056768782345
9300000040056768782345
3000000040056768782345

F	M	E	N	S	W
0	1	0	0	0	1
1	0	1	0	0	0
1	0	0	0	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	0	1	0
1	0	0	1	0	0
1	0	0	0	1	0
1	0	0	1	0	0
0	1	0	0	1	0
0	1	1	0	0	0
1	0	1	0	0	0
1	0	0	0	0	1

Columnar DBMS

- SQL databases traditionally store a table in a file as a sequence of records, one per tuple
- Columnar databases typically store a table as a collection of files
  - » Typically, each file stores one column of the table
- Columnar databases are likely more efficient in answering OLAP queries dealing with a few columns from very wide tables.
- But they can use a standard SQL syntax as they essentially deal with tables

Example: Row-oriented Storage
(Typical SQL DBMS Implementation)

Employee	ID#	Name	Salary
	1	Alice	425
	2	Bob	300
	4	Carol	
	5	David	126
	6	Bob	720

- 1 file with 5 records (drawn in two lines here)

1	Alice	425	2	Bob	300	4	Carol	NULL

5	David	126	6	Bob	720

Example: Column-oriented Storage
(Typical Columnar DBMS Implementation)

Employee	ID#	Name	Salary
	1	Alice	425
	2	Bob	300
	4	Carol	
	5	David	126
	6	Bob	720

- 3 files with 5 records each

1	2	4	5	6
Alice	Bob	Carol	David	Bob
425	300	NULL	126	720

Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

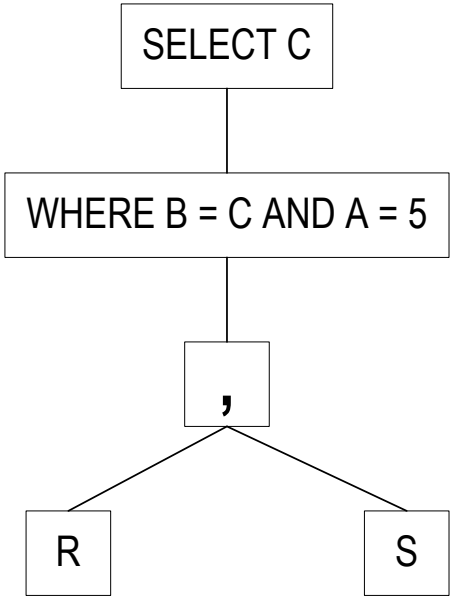
Query Optimization and Query Execution Concepts

Execution Plan

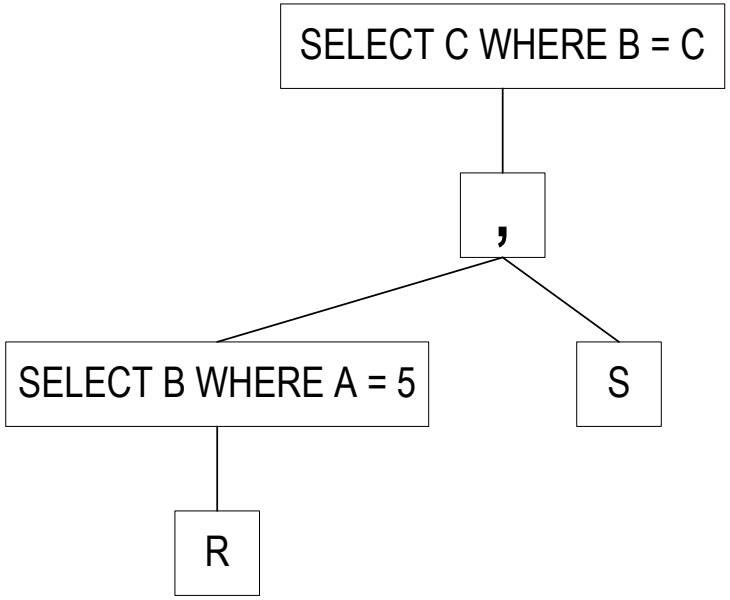
- Given a reasonably complex query there may be various ways of executing it, which can change performance from a few seconds to many minutes or vice versa
- Good Database Management Systems
  - ***Maintain statistical information about the database***
  - ***Use this information to decide how to execute the query***
- Given a query, they decide on an execution plan
- Recall that SQL queries are (mostly) ***expressions*** (in our algebra) and not programs
- ***Expression trees***, showing execution from leaves to the root, can be created
- ***Expression trees can be optimized***

Improving the Execution of a Simple Query:
Optimize/Improve the Expression Tree

- Given R(A, B) and S(C)
- SELECT C
  FROM R, S
  WHERE B = C AND A = 5;



Original expression tree



More efficient expression tree

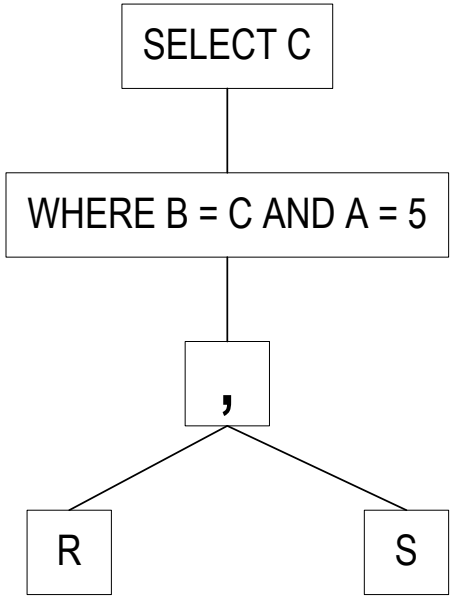
Two Execution Plans - The Right One Uses INNER JOINs

```
CREATE TABLE R(
       A INTEGER,
       B INTEGER
);

CREATE TABLE S(
       C INTEGER,
       D INTEGER
);

SELECT S.C AS C
FROM R, S
WHERE B = C AND A = 5;
```

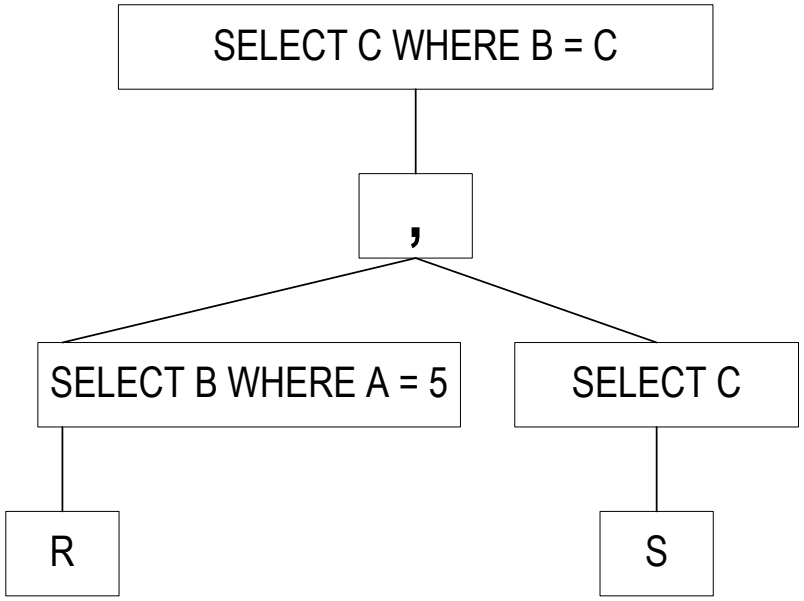
```
CREATE TABLE R(
       A INTEGER,
       B INTEGER
);

CREATE TABLE S(
       C INTEGER,
       D INTEGER
);

SELECT S_C AS C
FROM (SELECT B AS R_B FROM R
WHERE A = 5)
INNER JOIN (SELECT
C AS S_C FROM S)
ON R_B = S_C;
```

Left Execution Plan

```
CREATE TABLE R(
        A INTEGER,
        B INTEGER
);

CREATE TABLE S(
        C INTEGER,
        D INTEGER
);

SELECT S.C AS C
FROM R, S
WHERE B = C AND A = 5;
```



Right Execution Plan




```
CREATE TABLE R(
       A INTEGER,
       B INTEGER
);

CREATE TABLE S(
       C INTEGER,
       D INTEGER
);

SELECT S_C AS C
FROM (SELECT B AS R_B FROM R
WHERE A = 5)
INNER JOIN (SELECT
C AS S_C FROM S)
ON R_B = S_C;
```

Which Plan Is Better?

- Likely, the right plan
  - The INNER JOIN there works with smaller inputs than the Cartesian product in the left plan
  - INNER JOINs are frequently more efficient than Cartesian Products even with the same size inputs
- But we do not know until we investigate how the query planner in the DBMS works

- But we can think what we would do to execute the query
- Important observation, as we will see next, we do not need to produce the two intermediate tables

```
SELECT B FROM R WHERE A = 5;
SELECT C FROM S;
```

Pipelining For Better Performance - A New Execution Plan

1. Initialize a ***hash table*** for C values of S
2. Read S, for each tuple
   extract C and add it to the ***hash table*** unless it is already there (constant time for the tuple)
3. Initialize the ***result table*** for the query
4. Read R, for each tuple
   extract B and check if it is in the ***hash table*** (constant time per tuple)

   If yes, insert it as a new tuple in the ***result table***

- Total cost is proportional to the length(S) + length(R)

  Not quadratic in the sizes of the inputs!

Using or Not Using Indexes

When to Use Indexes to Find Records

- When you expect that it is cheaper than simply going through the file
- How do you know that? Make profiles, estimates, guesses, etc.
- Note the importance of clustering in greatly decreasing the number of block accesses
  - Example, if there are 1,000 blocks in the file, 100 records per block (100,000 records in the file), and
    - There is a hash index on the records
    - The file is unclustered (of course)
  - To find records of interest, we need to read at least all the blocks that contain such records
  - To find 50 (very small fraction of) records, perhaps use an index, as these records are in about (maybe somewhat fewer) 50 blocks, very likely
  - To find 2,000 (still a small fraction of) records, do not use an index, as these records are in “almost as many as” 1,000 blocks, very likely, so it is faster to traverse the file and read 1000 blocks (very likely) than use the index

A Sharp Example

- We have a relation Employee(SSN,BirthDate,Sex)
- We have a primary index on SSN and secondary indexes on BirthDate and Sex
- We have a query
  SELECT SSN
  FROM Employee
  WHERE BirthDate = '1982-05-03' AND Sex = 'Female';
- Two choices
  - Bring all the blocks containing '1982-05-03' and then pick out the records containing 'Female'
  - Bring all the blocks containing 'Female' (which could be all the blocks) and then pick out the records containing '1982-05-03'
- If the systems knows that about 50% of the employees are Female, the second choice is very bad, but the first choice is likely to be very good
- But the system must profile the database, or the programmer needs to set/influence the execution plan

Computing Conjunction Conditions

- A simple “more general” example: R(A,B)
- SELECT *
  FROM R
  WHERE A = 1 AND B = 'Mary';

- Assume the database has indexes on A and on B
- This means, we can easily find
  - All the blocks that contain at least one record in which A = 1
  - All the blocks that contain at least one record in which B = ‘Mary’

- A reasonably standard solution
  - DB picks one attribute with an index, say A
  - Brings all the relevant blocks into memory (those in which there is a record with A = 1)
  - Selects and outputs those records for which A = 1 and B = ‘Mary’

But There Are Choices

- Choice 1:
  - DB picks attribute A
  - Brings all the relevant blocks into memory (those in which there is a record with A = 1)
  - Selects and outputs those records for which A = 1 and B = 'Mary'
- Choice 2:
  - DB picks attribute B
  - Brings all the relevant blocks into memory (those in which there is a record with B = 'Mary')
  - Selects and outputs those records for which A = 1 and B = 'Mary'

- It is more efficient to pick the attribute for which there are fewer relevant blocks, i.e., the index is more ***selective***

But There Are Choices

- Some databases maintain ***profiles***, statistics helping them to decide which indexes are more selective
- But some have a more static decision process
  - Some pick the first in the sequence, in this case A
  - Some pick the last in the sequence, in this case B (Oracle used to do that)
- So depending on the system, one of the two below may be much more efficient than the other
  - SELECT *
    FROM R
    WHERE A = 1 AND B = 'Mary';
  - SELECT *
    FROM R
    WHERE B = 'Mary' AND A = 1;
- So it may be important ***for the programmer*** to decide which of the two equivalent SQL statements to specify

Using Both Indexes

- DB brings into RAM the identifiers of the relevant blocks (their serial numbers
- Intersects the set of identifiers of blocks relevant for A with the set of identifiers of blocks relevant for B
- Reads into RAM the identifiers in the intersections and selects and outputs records satisfying the conjunction

Joins

Computing A Join Of Two Tables

- We will deal with rough asymptotic estimates to get the flavor
  - » So we will make simplifying assumptions, which still provide the intuition, but for a very simple case
- We have available RAM of 3 blocks
- We have two tables R(A,B), S(C,D)
- Assume no indexes exist
- There are 1,000 blocks in R
- There are 10,000 blocks in S
- We need (or rather DB needs) to compute

```
SELECT *
FROM R, S
WHERE R.B = S.C;
```

A Simple Approach

- Read 2 blocks of R into memory
  - » Read 1 block of S into memory
  - » Check for join condition for all the “resident” records of R and S
  - » Repeat a total of 10,000 times, for all the blocks of S
- Repeat for all subsets of 2 size of R, need to do it total of 500 times
- Total reads: 1 read of R + 500 reads of S = 5,001,000 block reads
- Essentially quadratic in the size of the files

A Simple Approach (continued)

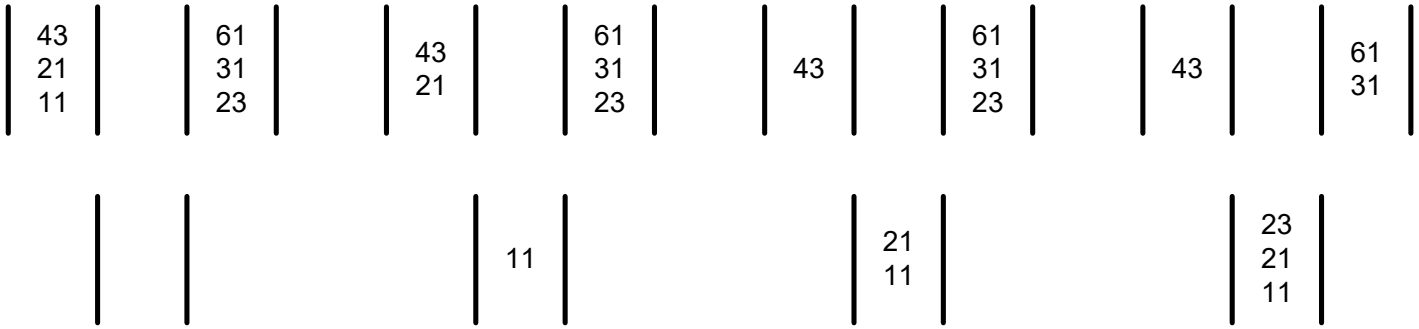
- Read 2 blocks of S into memory
  - » Read 1 block of R into memory
  - » Check for join condition for all the “resident” records of R and S
  - » Repeat a total of 1,000 times, for all the blocks of R
- Repeat for all subsets of 2 size of S, need to do it total of 5,000 times

- Total reads: 1 read of S + 5,000 reads of R = 5,010,000 block reads

- Essentially quadratic in the size of the files

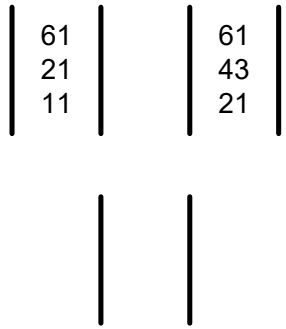
Reminder On Merge-Sort

- At every stage, we have sorted sequences of some length
- After each stage, the sorted sequences are double in length and there is only half of them
- We are finished when we have one sorted sequence
- Three blocks are enough: two for reading the current sorted sequences and one for producing the new sorted sequence from them
- In the example, two sorted sequence of length three are merged; only the first three steps are shown



Merge-Join

- If R is sorted on B and S is sorted on C, we can use the merge-sort approach to join them
- While merging, we compare the current candidates and output the smaller (or any, if they are equal)
- While joining, we compare the current candidates, and if they are equal, we join them, otherwise we discard the smaller
- In the example below (where only the join attributes are shown, we would discard 11 and 43 and join 21 with 21 and 61 with 61



Merge-Join

- The procedure:
  - Sort R (using merge sort)
  - Sort S (using merge sort)
  - Join R and S
- To sort R
  - Read 3 blocks of R, sort in place and write out sorted sequences of length of 3 blocks
  - Merge sort R
- To sort S
  - Read 3 blocks of S, sort in place and write out sorted sequences of length of 3 blocks
  - Merge sort S
- Then merge-join

Performance Of Merge-Join

- To sort R, ceiling of $\log_2 1000 = 10$ passes, where pass means read and write
- To sort S, ceiling of $\log_2 10000 = 14$ passes, where pass means read and write
- Once we have sorted R and S, to merge-join R and S, one pass on each of R and S (as we join on keys, so each row of R either has one row of S matching, or no matching row at all)
  - » Specifically we read the first block of R and the first block of S and join what we can and then read the next block of either R or S or both depending on which is completely processed
  - » When a block of the join is full, write it out and start a new block
- Cost
  - » 10 passes to sort R: 20000 block accesses (reads and writes)
  - » 14 passes to sort S: 280000 block accesses (reads and writes)
  - » joining: 1000 for reading R, 10000 for reading S, at most 1000 for writing the join

Hash-Join

- Sorting could be considered in this context similar to the creation of an index
- Hashing can be used too and works well provided hash blocks are substantially smaller than RAM size
  - » trickier to do in practice and therefore less popular

Order Of Joins Matters – Decomposing Joing?

- Consider a database consisting of 3 relations
  - Data(Person, Sex, …) about people in the US, about 300,000,000 tuples
  - Oscar(Person) about people in the US who have won the Oscar, about 1,000 tuples
  - Nobel(Person) about people in the US who have won the Nobel, about 100 tuples
- We are interested in a dating application
- Produce the relation Good_Match(Person1,Person2) where the two Persons live in the same city and the first won the Oscar prize and the second won the Nobel prize
- How would you do it using SQL?
- Recall the example of computing (Father, Daughter) from a previous Unit

Order Of Joins Matters

- CREATE Good_Match AS

  SELECT Oscar.Person Person1, Nobel.Person Person2
  FROM Oscar, Data Data1, Nobel, Data Data2
  WHERE Oscar.Person = Data1.Person
  AND Nobel.Person = Data2.Person
  AND Data1.City = Data2.City

- Very inefficient, the cartesian product has $300{,}000{,}000 \times 300{,}000{,}000 \times 1{,}000 \times 100 = 9{,}000{,}000{,}000{,}000{,}000{,}000$ tuples
- Using various joins we can specify easily the "right order," in effect producing
  - Oscar_PC(Person,City), listing people with Oscars and their cities
  - Nobel_PC(Person,City), listing people with Nobels and their cities
- Then producing the result from these two small relations
- Effectively we do some WHERE conditions earlier, rather than later
- This is much more efficient

Decomposing Joins?

- CREATE OscarSex AS
  SELECT Oscar.Person Person, Sex
  FROM Oscar, Data
  WHERE Oscar.Person = Data.Person;

- The cartesian product has
  $300{,}000{,}000 \times 1{,}000 = 300{,}000{,}000{,}000$
  tuples, and the result has $1{,}000$ tuples

- CREATE NobelSex AS
  SELECT Nobel.Person Person, Sex
  FROM Nobel, Data
  WHERE Nobel.Person = Data.Person;

- The cartesian product has
  $300{,}000{,}000 \times 100 = 30{,}000{,}000{,}000$
  tuples and the result has 100 tuples

Can Compute A Join Better

- CREATE OscarSex AS
  SELECT Oscar.Person Person, Sex
  FROM Oscar, Data
  WHERE Oscar.Person = Data.Person;

- A reasonable execution plan

- Hash Oscar on person: less than $10{,}000$ operations ($1{,}000$ times some small constant)

- Go through Data and for each tuple check if the Person is in Oscar_PC by hash lookup: slightly more than $300{,}000{,}000$ operations

Agenda

1. Session Overview
2. Physical Database Design
3. Data Structures in RAM
4. Data Structure On Disk
5. Searching On Key Values
6. B+ Trees – Search Trees For Files On Disk
7. Hashing On A Disk
8. Searches on Non-Key Values
9. Secondary Indexes and More
10. SQL and Oracle
11. Query Execution
12. Summary and Conclusion

Database Design Context

- Logical DB Design (what we have done so far):
  1. Create first a model of the enterprise (e.g., using ER)
  2. Create a logical "implementation" (using a relational model and normalization)
  - » Creates the top two layers: "User" and "Community"
  - » Independent of any physical implementation
  - » It is frequently called (regrettably) physical design
- Physical DB Design
  - » Uses a file system to store the relations
  - » Requires some knowledge of hardware and operating system's characteristics
  - » Addresses questions of distribution (distributed databases), if applicable
  - » Creates the third layer
- Query execution planning and optimization ties the three layers together; not the focus of this unit, but some issues discussed

Key Ideas

- The goal: efficiency
- Logical and physical files
- The cost model: disk access dominates
- Implications of cost model
- Optimize file organization and create indexes
- Sparse and dense indexes
- When to use hashing and when to use 2-3 trees
- Clustered and unclustered files
- Hashing and B^+ trees
- Optimal B^+ trees
- Indexes on non-key fields
- Secondary indexes
- Symbolic vs. physical pointers

Key Ideas

- SQL support for physical design
- Oracle support for physical design
- Query execution concepts

Any Questions?