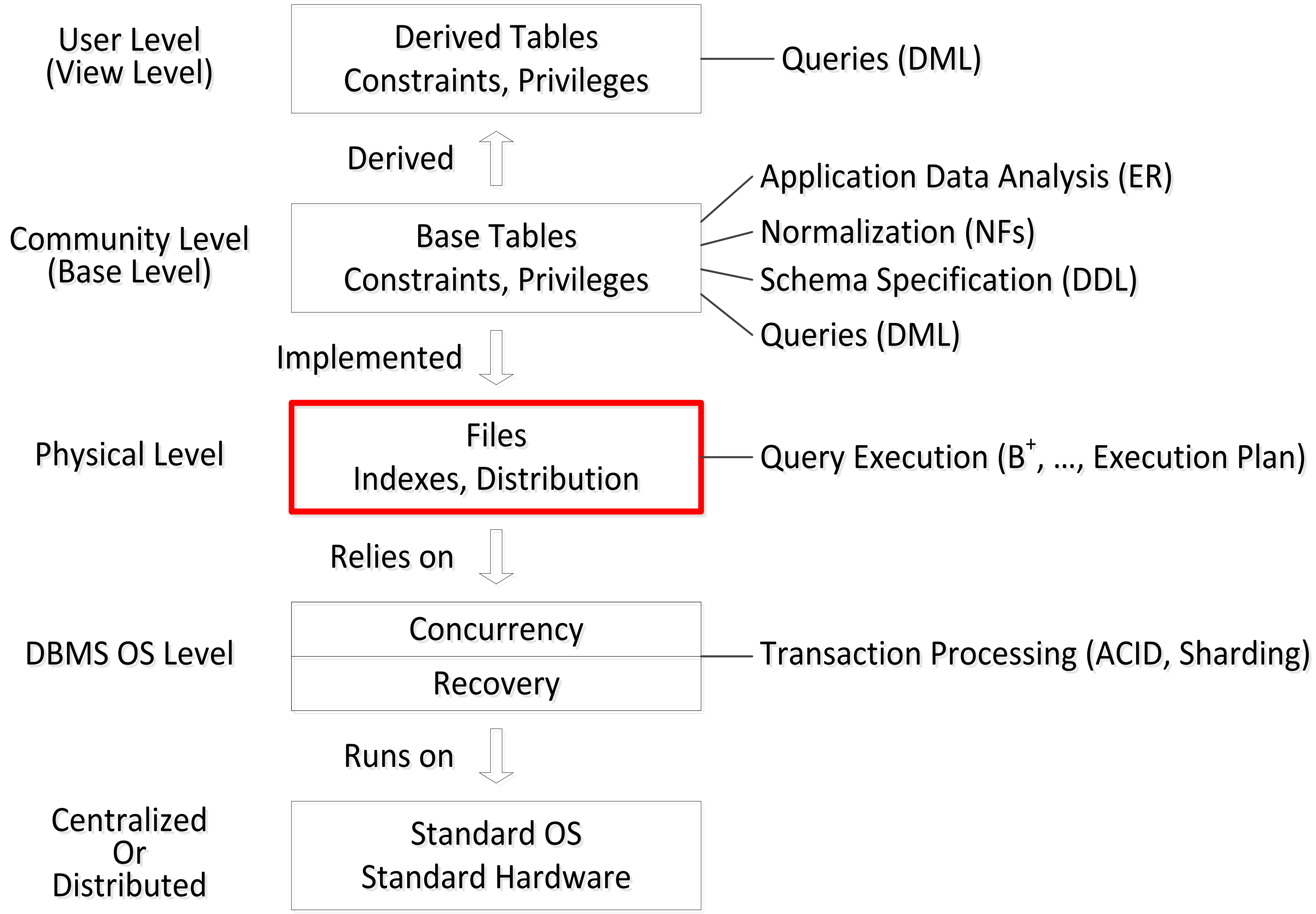
Unit 8
Physical Design
And Query Planning And Execution Concepts

Physical Design and Query Execution in Context



User Level (View Level)

Derived Tables
Constraints, Privileges

Queries (DML)

Derived

Community Level (Base Level)

Base Tables
Constraints, Privileges

Application Data Analysis (ER)

Normalization (NFs)

Schema Specification (DDL)

Queries (DML)

Implemented

Physical Level

Files
Indexes, Distribution

Query Execution (B^+, …, Execution Plan)

Relies on

DBMS OS Level

Concurrency
Recovery

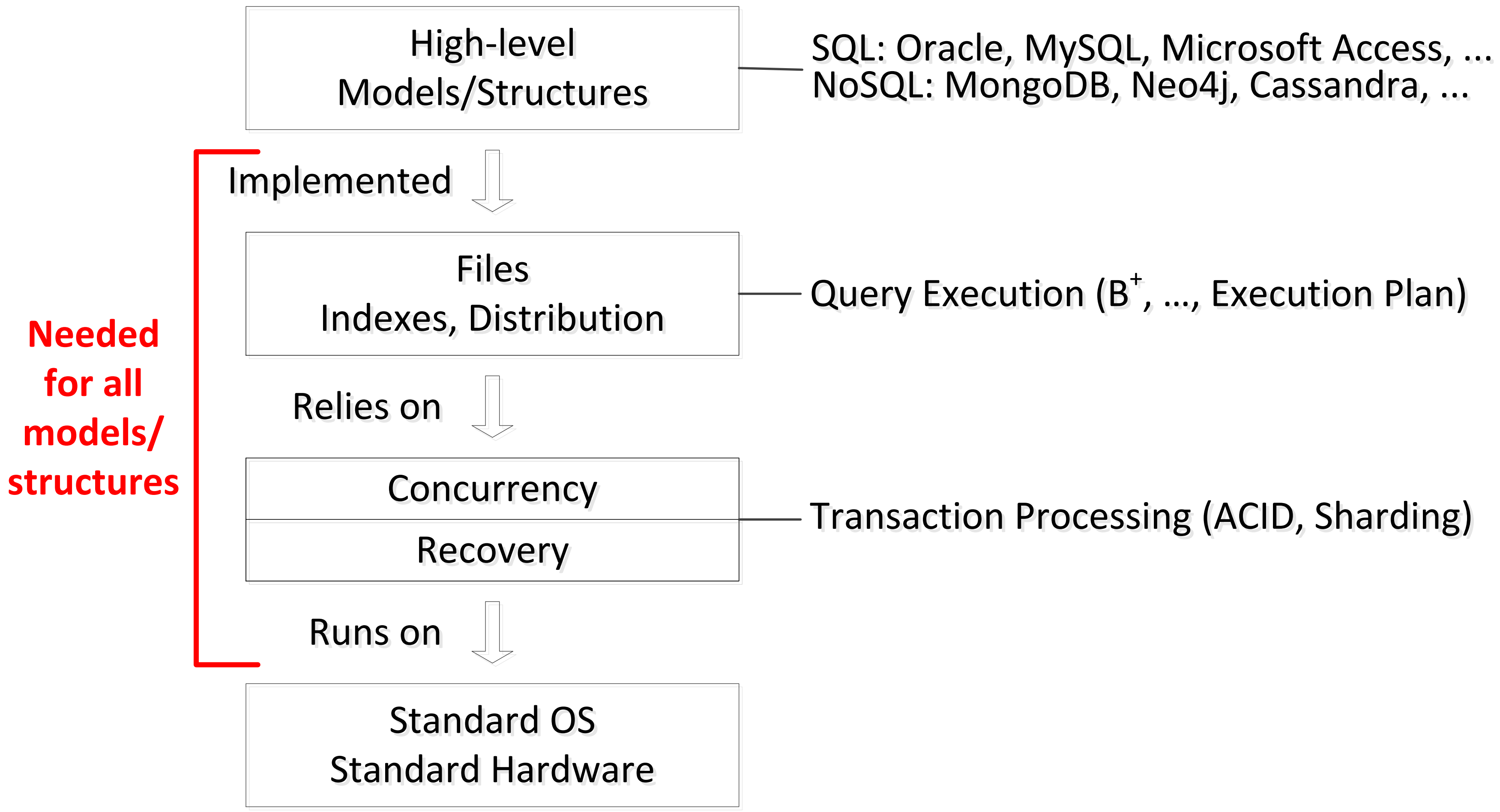
Transaction Processing (ACID, Sharding)

Runs on

Centralized
Or
Distributed

Standard OS
Standard Hardware

Lower Levels Needed All DBMSs
Not Only For Relational DBMSs



High-level
Models/Structures

SQL: Oracle, MySQL, Microsoft Access, ...
NoSQL: MongoDB, Neo4j, Cassandra, ...

Implemented

Files
Indexes, Distribution

Query Execution (B^+, …, Execution Plan)

Relies on

Concurrency

Recovery

Transaction Processing (ACID, Sharding)

Runs on

Standard OS
Standard Hardware

Needed for all models/ structures

An Important Note: Wider Applicability

- The material in this unit is applicable to Relational Database Management Systems, key-value stores, just data-intensive systems in general.
- A system is data-intensive if the data is large.

Database Design

- 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

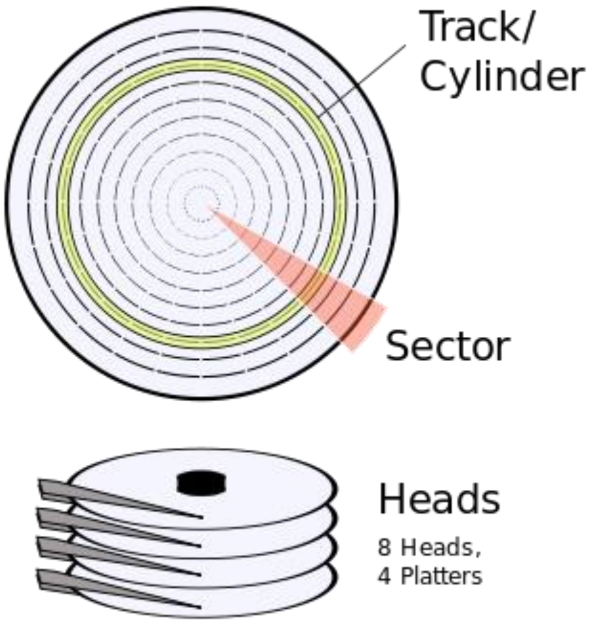
Physical Design: Basic Concepts

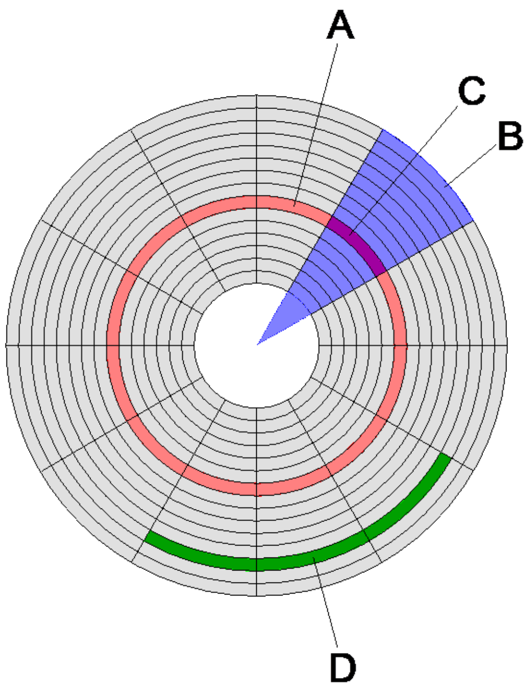
Issues Addressed In Physical Design

- ***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
- 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 Disk?

- 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 (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, usually at least 4 096 B (bytes)
- We assume: a virtual memory page is the same size a block

What Is A Disk?

- 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?

- 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

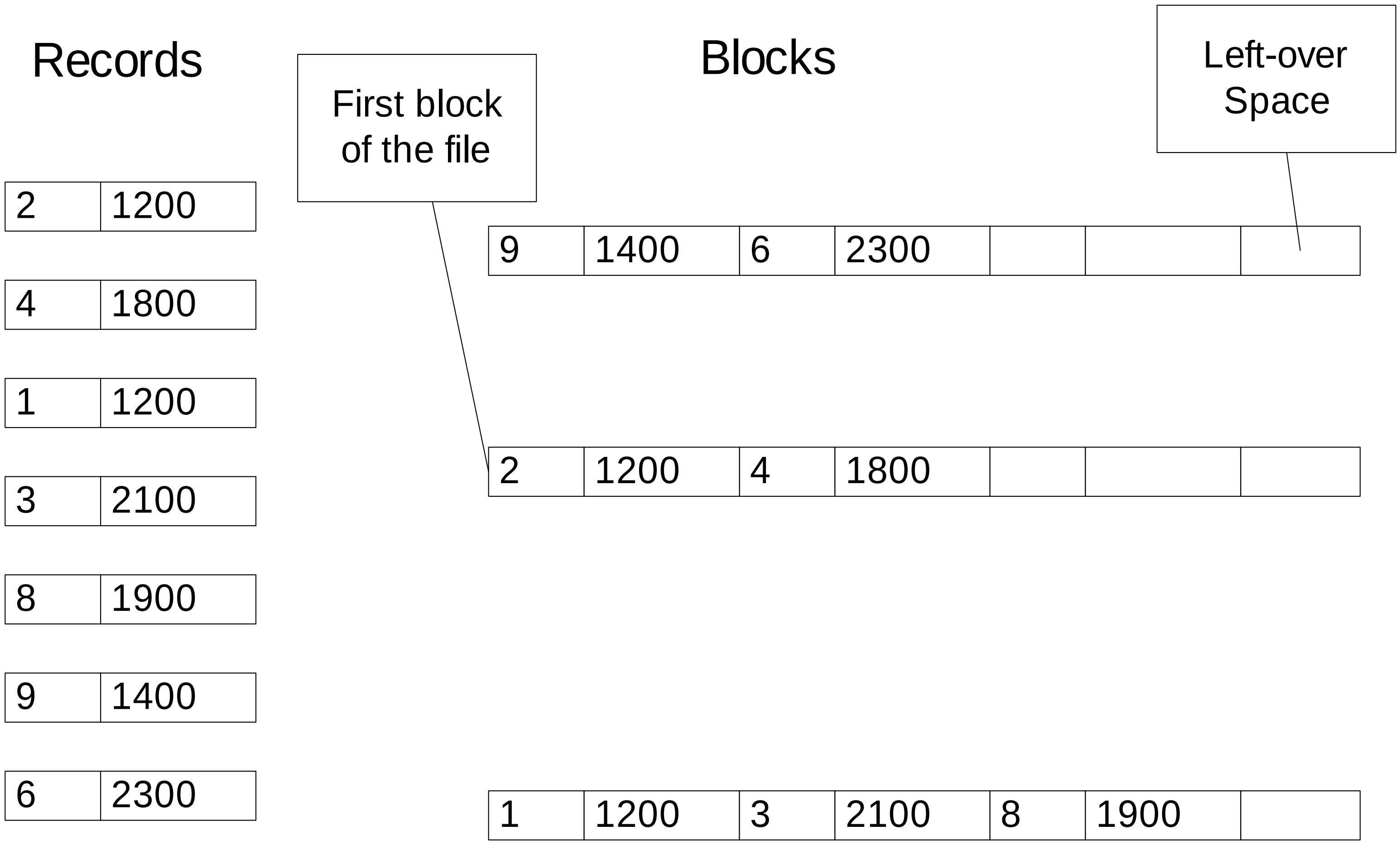
3	2100

8	1900

9	1400

6	2300

Example: Storing A Relation (Physical File)



Records

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

Blocks

First block of the file

Left-over Space

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

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 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***
- ***Processing in RAM is free***
- 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).

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

- 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
  - More indexes usually better.
- Extreme case: database is used only for INSERT queries
  - Indexes only for constraints, otherwise…
  - just append new records to the end of the file
- In general, there is a tradeoff.

Data Structures In RAM

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)

- 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 more if check for duplicates.

Sorted Sequence
(never 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

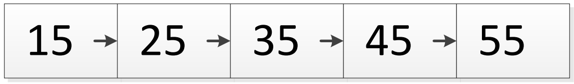
- Inserting

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

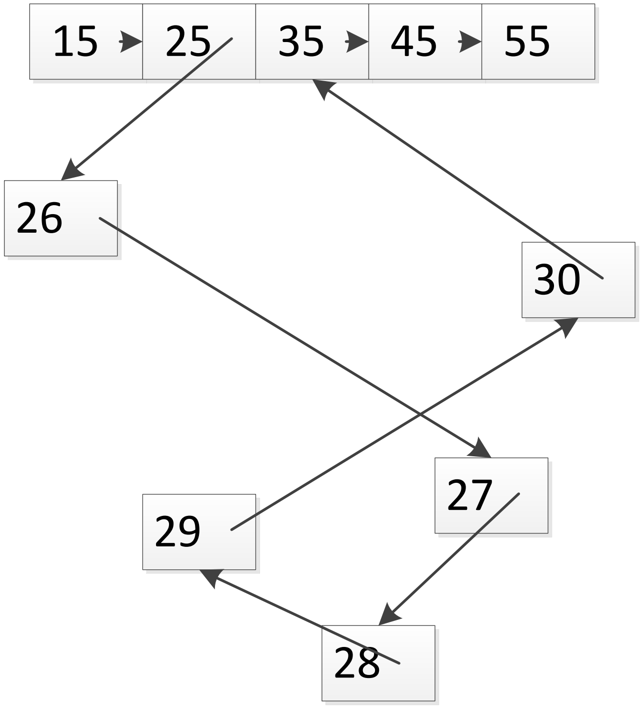
  Depends on the variant also. For instance, if “overflow” values can be put in the middle by means of a linked list, then this causes the “binary search to be unbalanced, resulting in possibly up to N operations for a Find.

Sorted Sequences

- 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)]
- Adjust the hash structure so most of the locations D[h(k)], will be pointing at a linked list of length 1 only

Hashing: Example Of Insertion

N = 7

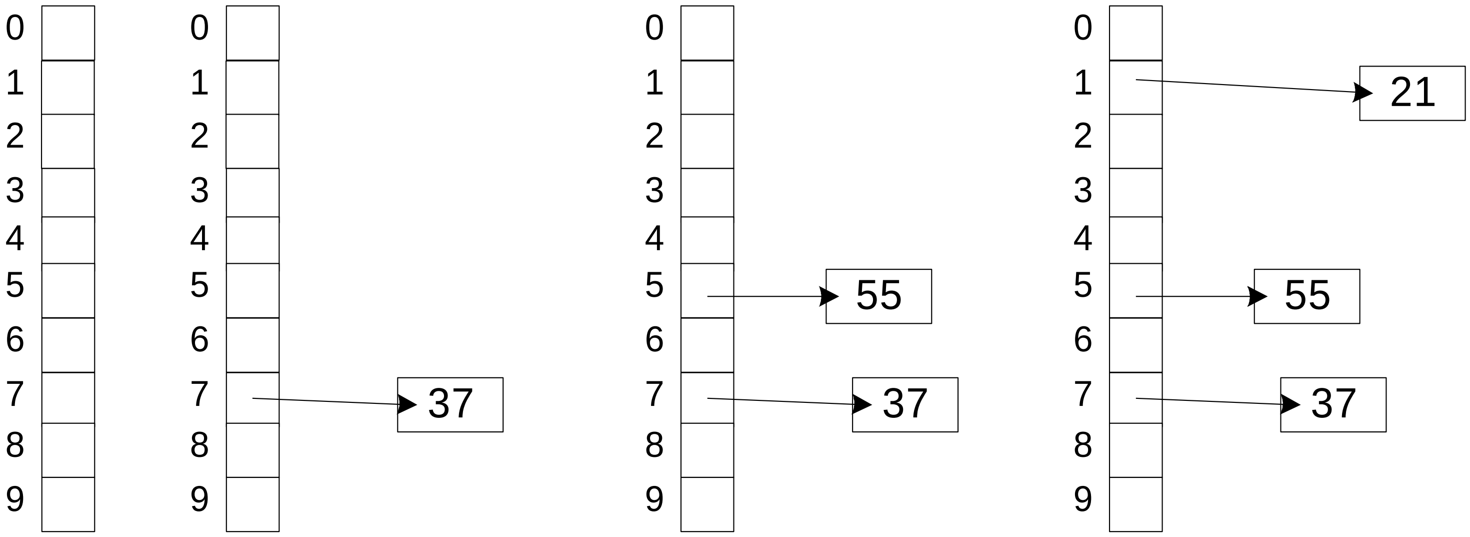
B = 10

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

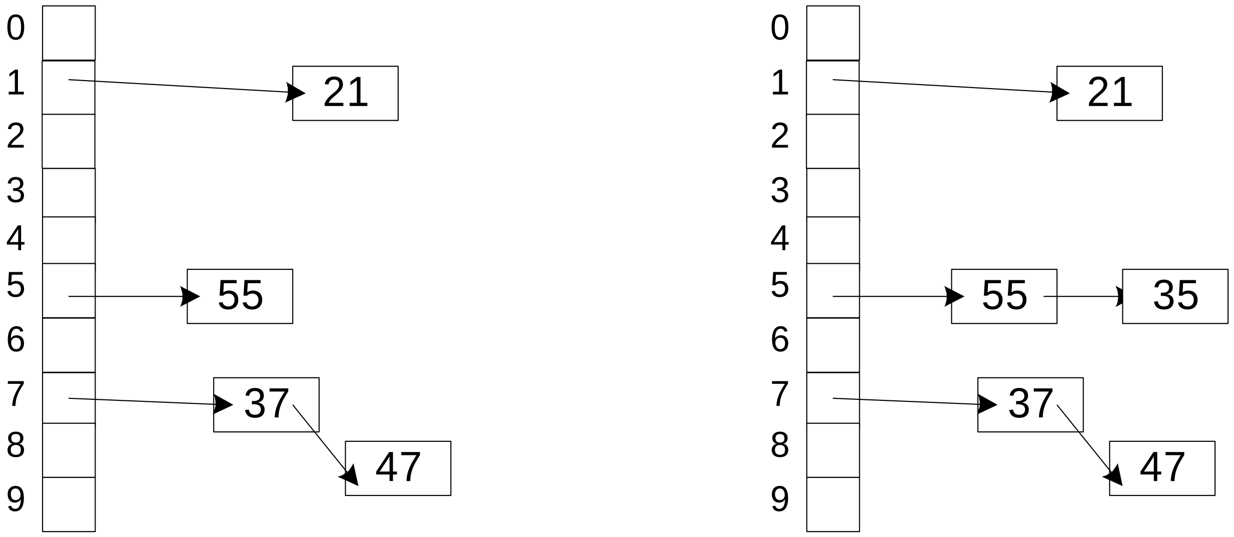
Integers arriving in order:

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

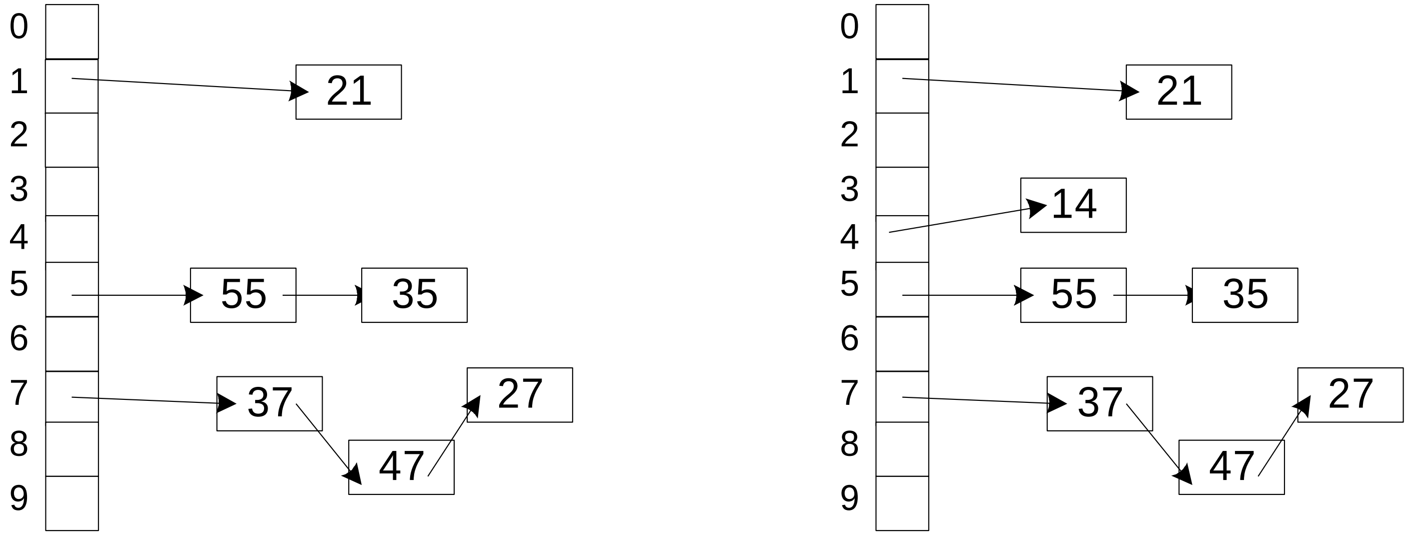
Hashing: Example Of Insertion



Hashing: Example Of Insertion



Hashing: Example Of Insertion



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

- Inserting

  Obvious modifications of ”Finding”

  Sometimes N becomes is “too close” to B (bucket table becomes too small)

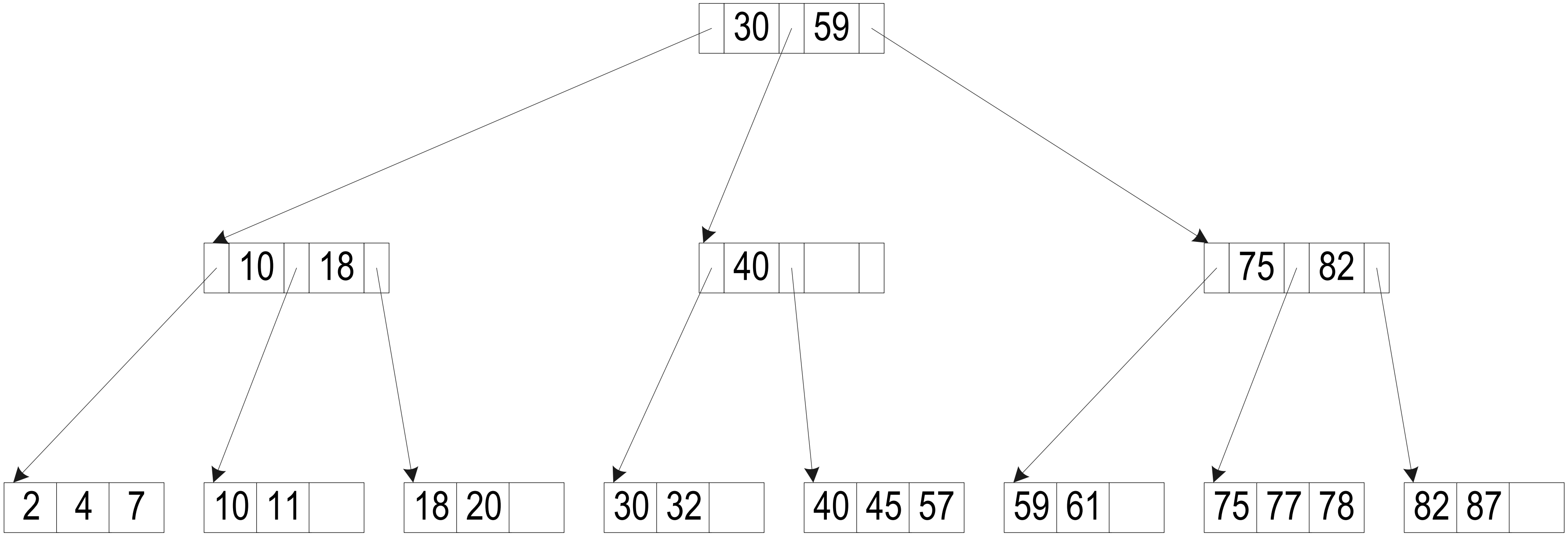
  - Then, increase the size of the bucket table and rehash (can do it incrementally)
  - Number of operations linear in N
  - Can amortize across all accesses (ignore, if you do not know what this means), but know that it is still constant operations per access even when taking into account rehashing

- Deleting

  Obvious modification of “Finding”

  Sometimes N too small for B, act “opposite” to Inserting

2-3 Tree (Example)



2-3 Trees

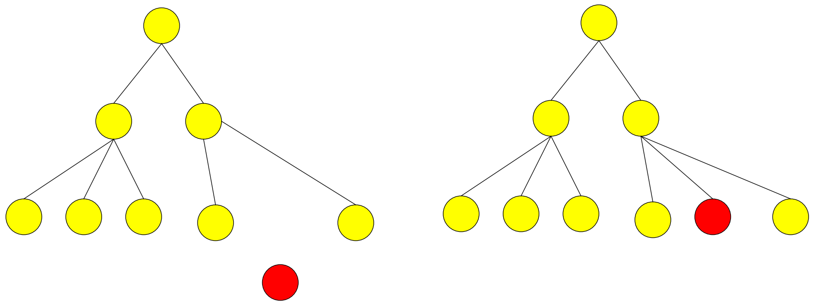
- A 2-3 tree is a rooted (it has a root) directed (order of children matters) tree
- All paths from root to leaves are of same length
- Each node (other than leaves) has between 2 and 3 children
- For each child of a node, other than the last, there is an index value stored in the node
- **For each non-leaf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

- 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

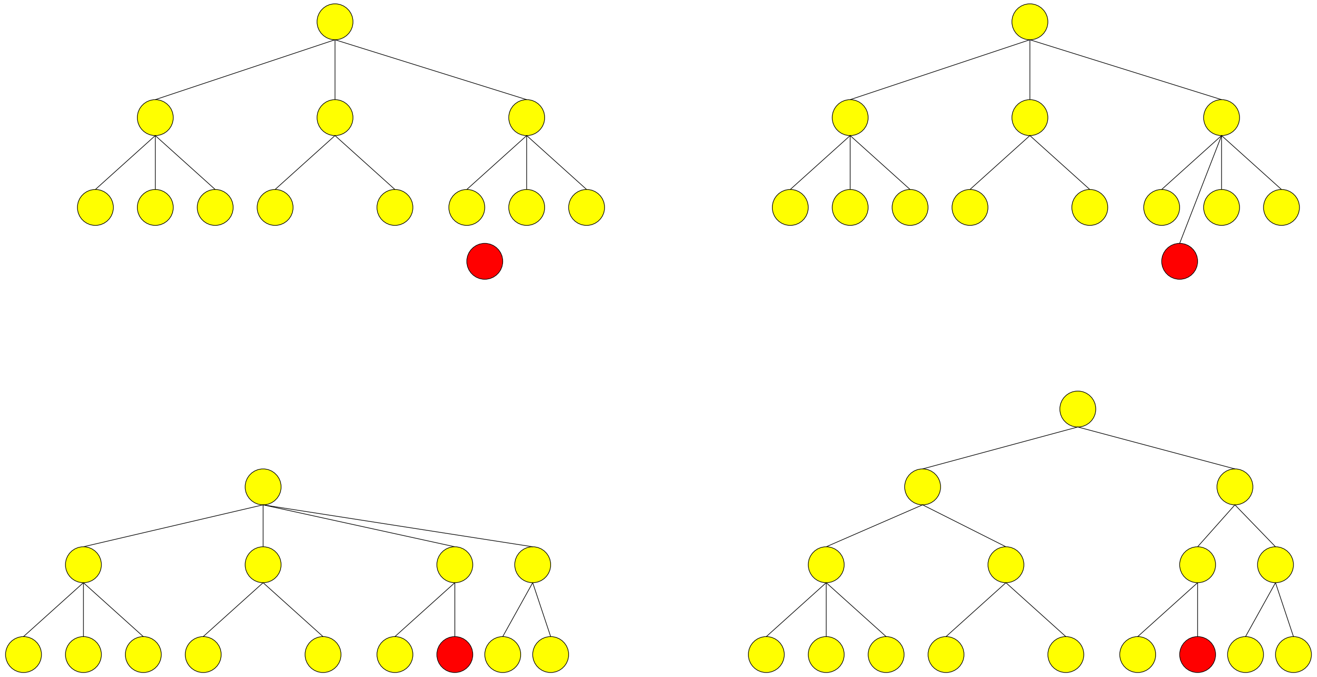
Insertion Of A Node In The Right Place (The Fastest Case)

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



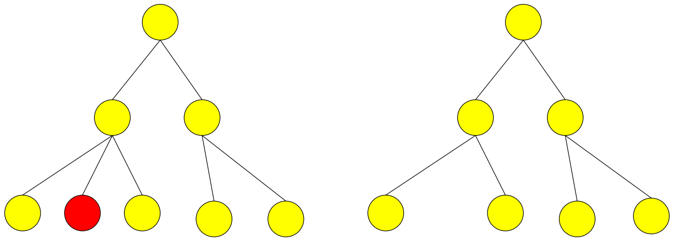
Insertion Of A Node In The Right Place (The Slowest Case)

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

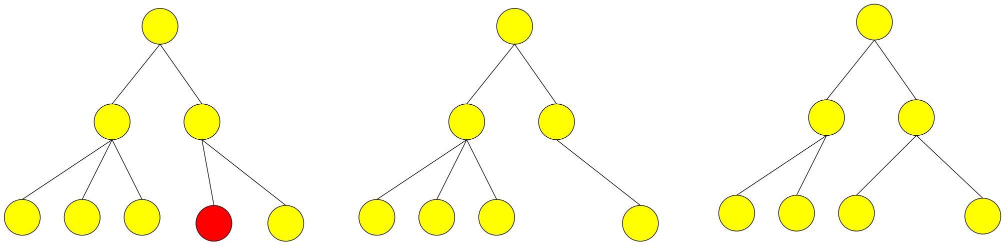


Deletion Of A Node (The Fastest Case)

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

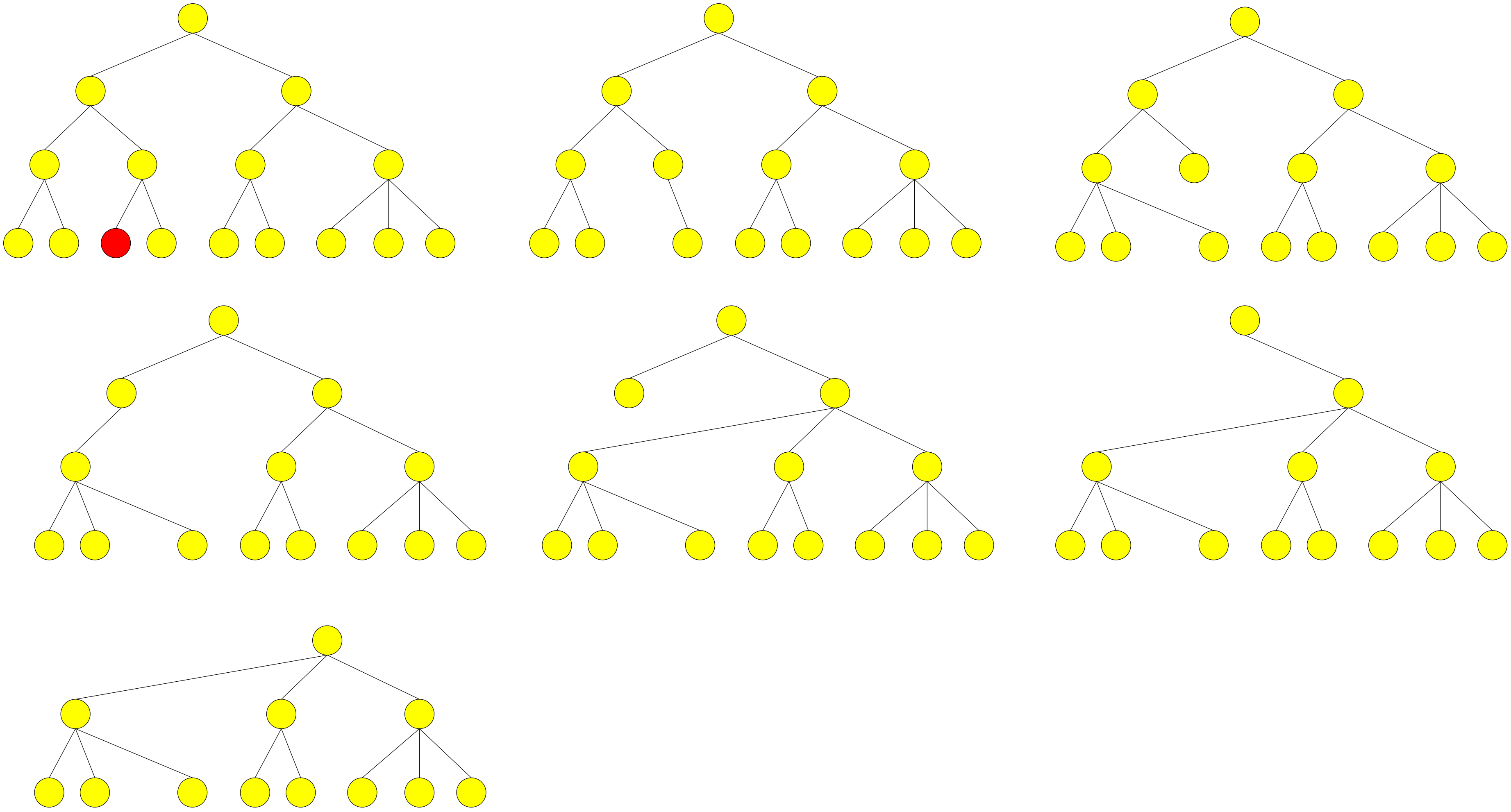


- First find the place to insert using Θ(log N) operations
- Deletion resolved at the leaf level with sibling stealing



Deletion Of A Node (The Slowest Case)

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



2-3 Trees

- Finding (including detecting of non-membership)

  Takes Θ(log N) operations

- Inserting while keeping trees balanced

  Takes Θ(log N) operations

- Deleting while keeping trees balanced

  Takes Θ(log N) operations

B^+-trees: Generalizes 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.

What To Use?

- If the set of integers is large, use either hashing or 2-3 trees
- Use 2-3 trees if "many" of your queries are range queries
- Find all the SSNs (integers) in your set that lie in the range 070430000 to 070439999
- Use hashing if your queries are on a key value.
- If you have a total of 10,000 integers randomly chosen from the set 0 ,..., 999999999, how many will fall in the range above?
- How will you find the answer using hashing, and how will you find the answer using 2-3 trees?

Data Structure On Disk: Basic Considerations

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

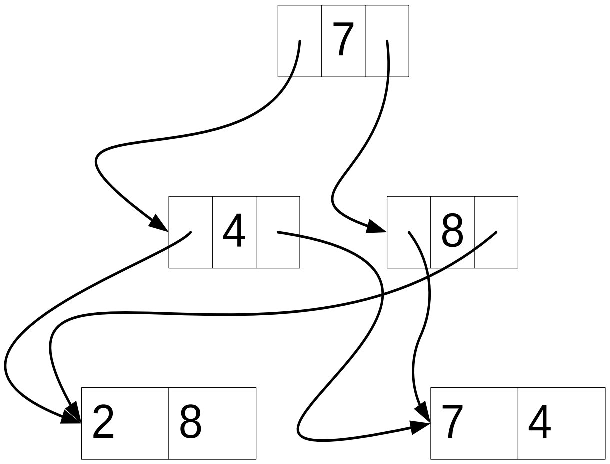
- Organize an index to a file so that you easily find the useful blocks

Index And File

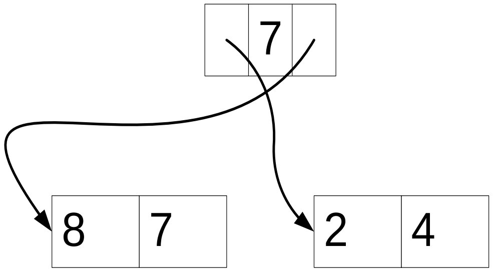
- In general, we have a data file of blocks, with blocks containing records
- Each record has a field, which contains a key, uniquely labeling/identifying the record and the “rest” of the record
- There is an index, which is another file which essentially (we will be more precise later) consists of records 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***

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

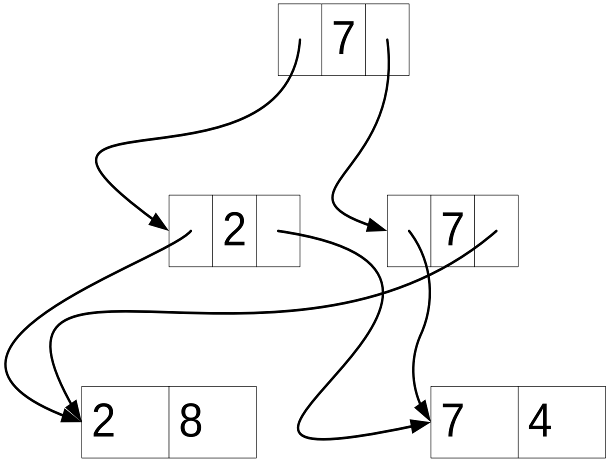


sparse index and clustered file

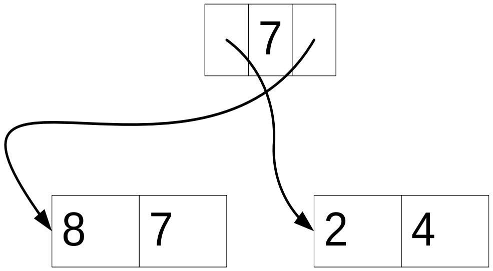
File: Clustered or Unclustered

- 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

2	4	7	8



dense index and unclustered file

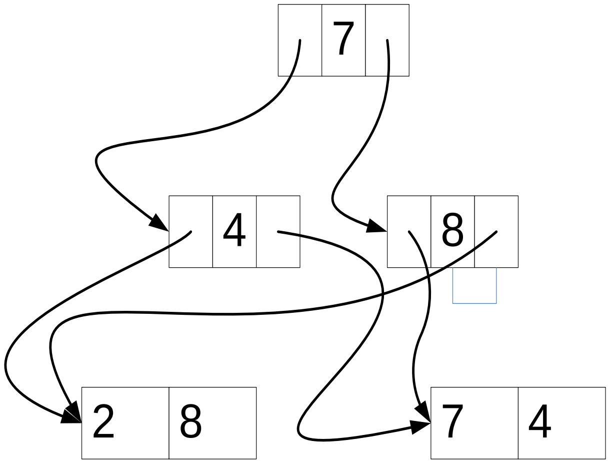


sparse index and clustered file

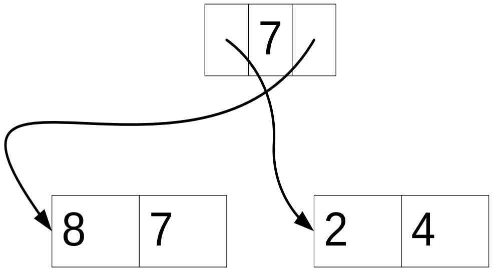
Fewest levels if accessing entire records: Clustered File + Sparse Index

- 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



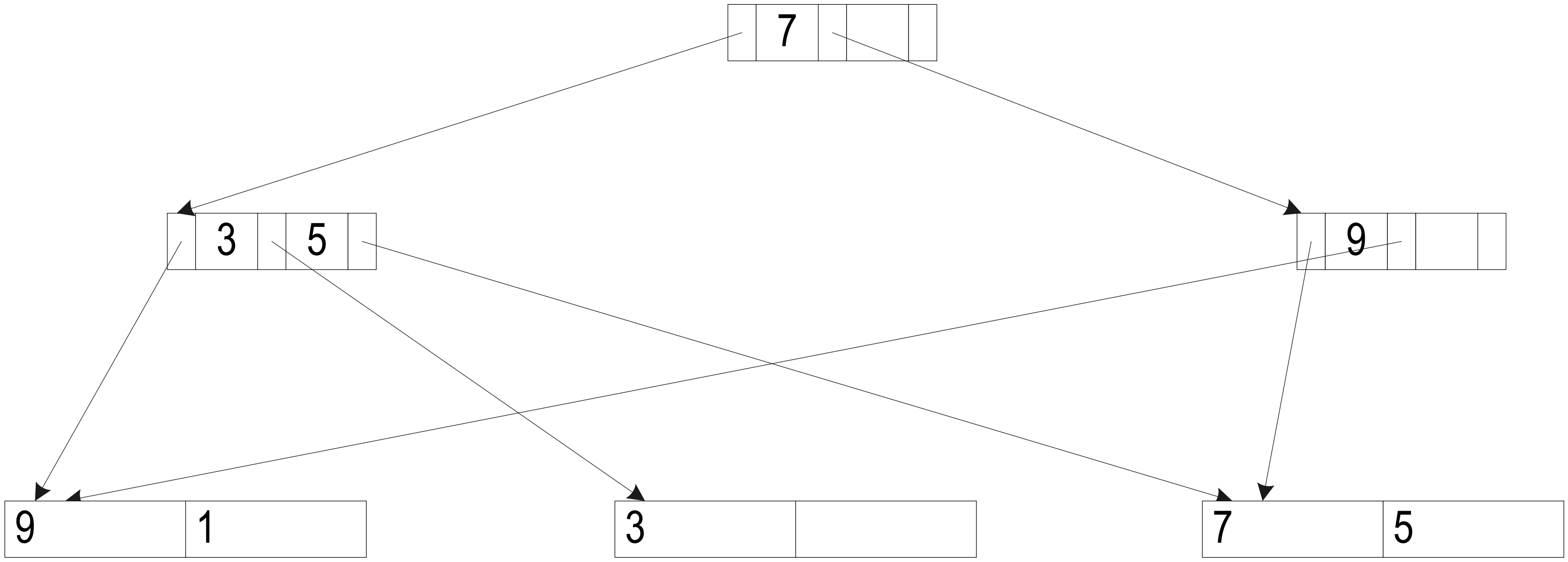
dense index and unclustered file



sparse index and clustered file

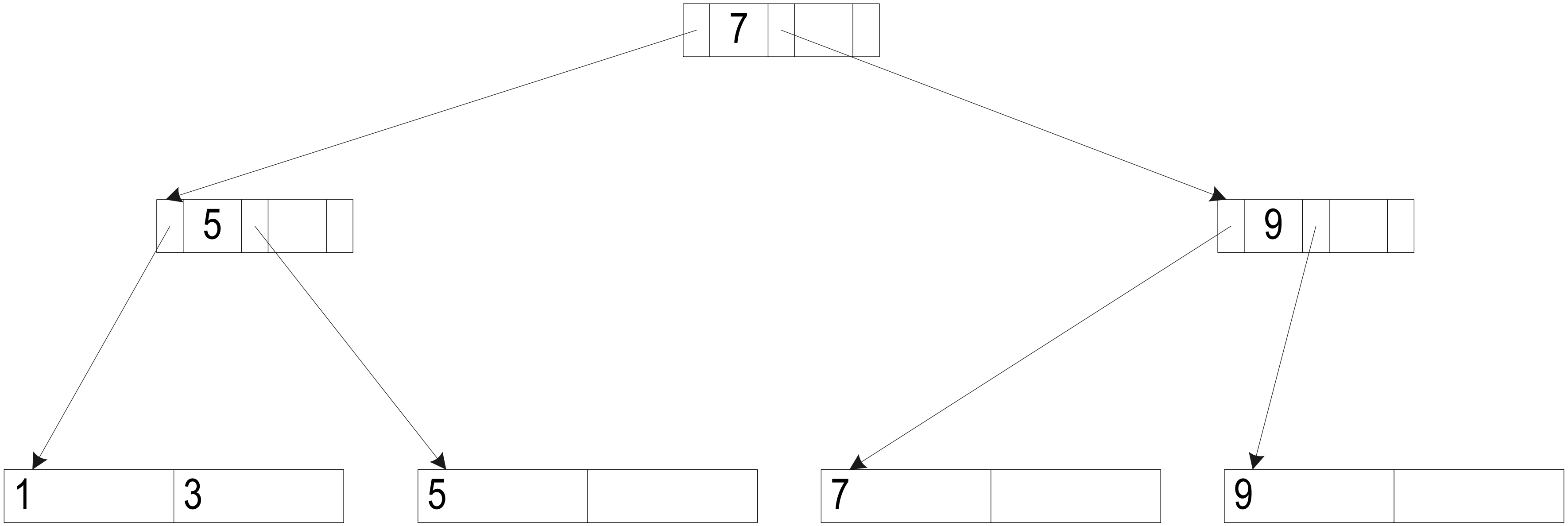
Dense Index And Unclustered File

- Pointers point from the leaf level of the tree to blocks (not records) of the file
  - Multiple pointers point at a block
- 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-text

- 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.
- Dennis class skip to page 95

Searching On Key Values

B^+-Trees:
Search Trees For Files On Disk

Search Trees

- We will study in greater depth search trees for finding records in a file
- They will be a generalization of 2-3 trees
- For very small files, the whole tree could be just the root
  - For example, in the case of 2-3 trees, a file of 1 record, for instance
- So to make the presentation simple, we will assume that there is at least one level in the tree below the root
  - The case of the tree having only the root is very simple and can be ignored here
- ***We start with a simplifying assumption that we index on a key value: that is, for each possible index value there is at most one record corresponding to the index***
  - So we could index on SSN
  - So we could not index on Salary

B^+-Trees: Generalization Of 2-3 Trees

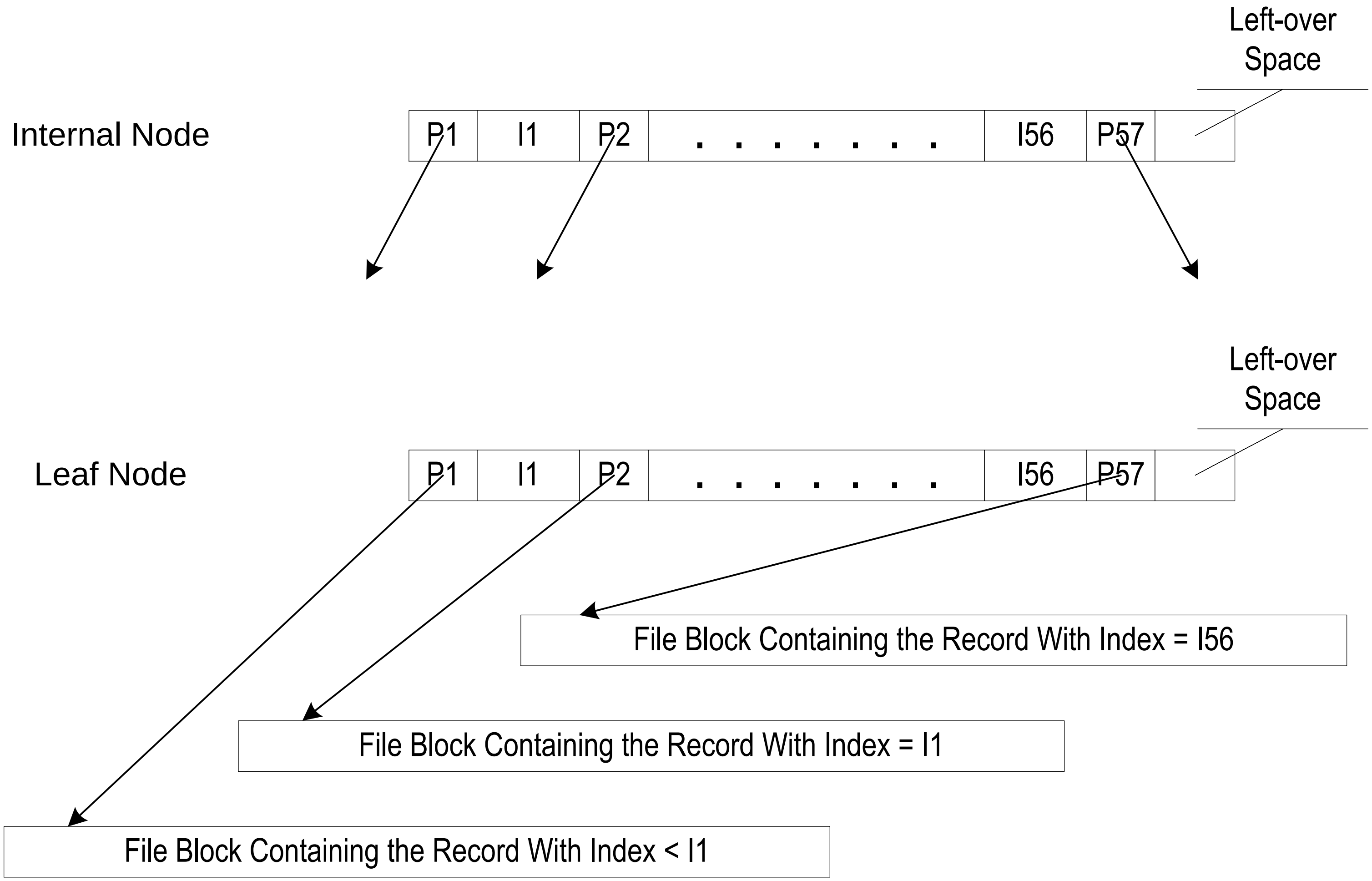
- A B^+-tree is a tree satisfying the conditions
  - It is rooted (it has a root)
  - It is directed (order of children matters)
  - All paths from root to leaves are of same length
  - For some parameter m:
    - All internal (not root and not leaves) nodes have between ceiling of m/2 and m children
    - The root between 2 and m children
- We ***cannot***, in general, avoid the case of the root having only 2 children, we will see soon why

B^+-Trees: Generalization Of 2-3 Trees-text

- Each node consists of a sequence (P is ***address*** or ***pointer***, I is ***index*** or ***key***): $P_1, I_1, P_2, I_2, \ldots, P_{m-1}, I_{m-1}, P_m$
- 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


Left-over Space
Internal Node
P1
I1
P2
. . . .
I56
P57
Left-over Space
Leaf Node
P1
I1
P2
. . . .
I56
P57
File Block Containing the Record With Index = I56
File Block Containing the Record With Index = I1
File Block Containing the Record With Index < I1

B^{+}-tree vs. Binary Search (then skip to p. 92)

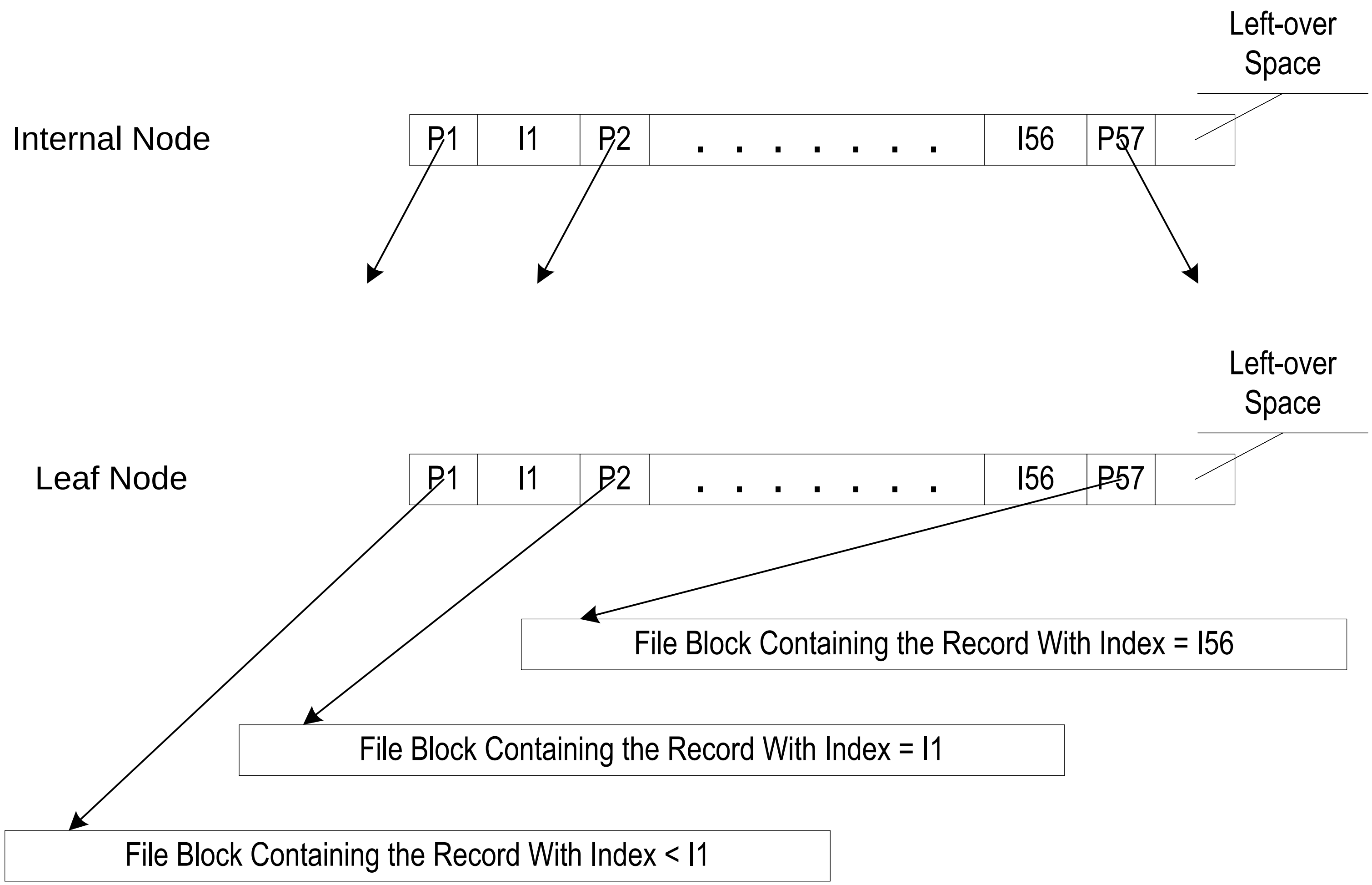
- For simplicity, assume that the leaves are just integers
- If can fit 1 gig in memory but there are 1 trillion integers, then binary search tree will have 10 levels on disk.
- B tree with fanout of 1000, just one or two.

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”

Example
Dense Index And Unclustered File

- 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)


Left-over
Space
Internal Node
P1
I1
P2
. . . .
I56
P57
Left-over
Space
Leaf Node
P1
I1
P2
. . . .
I56
P57
File Block Containing the Record With Index = I56
File Block Containing the Record With Index = I1
File Block Containing the Record With Index < I1

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

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

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

- 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

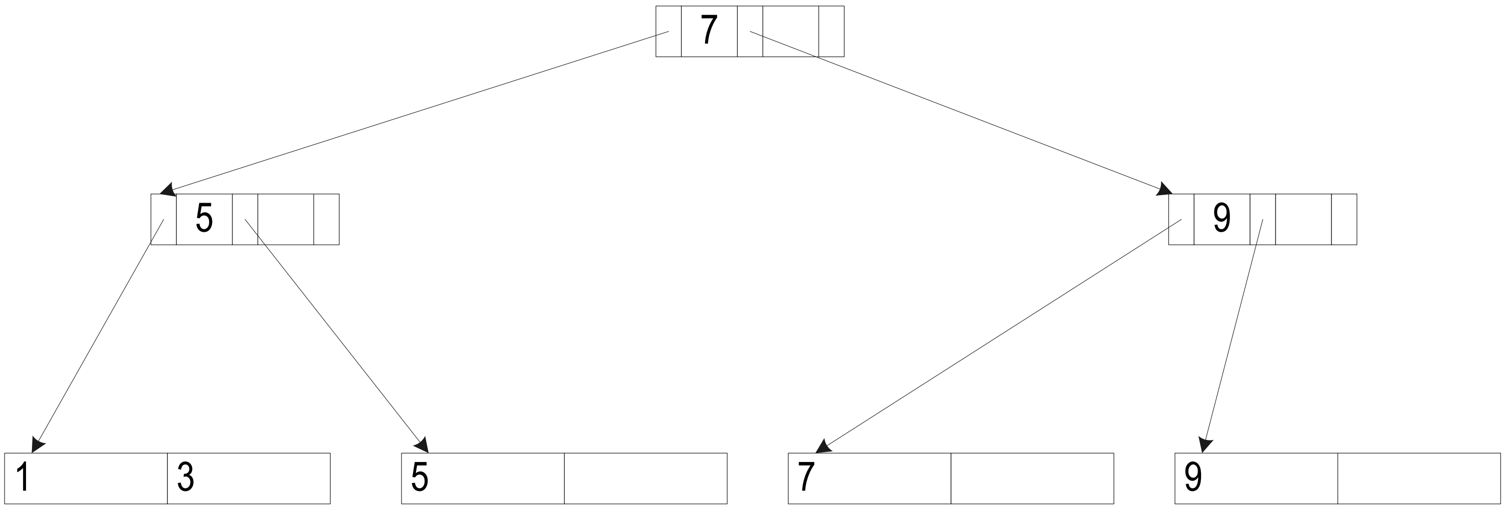
- A 2-3 tree, better than sorting but not by much

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

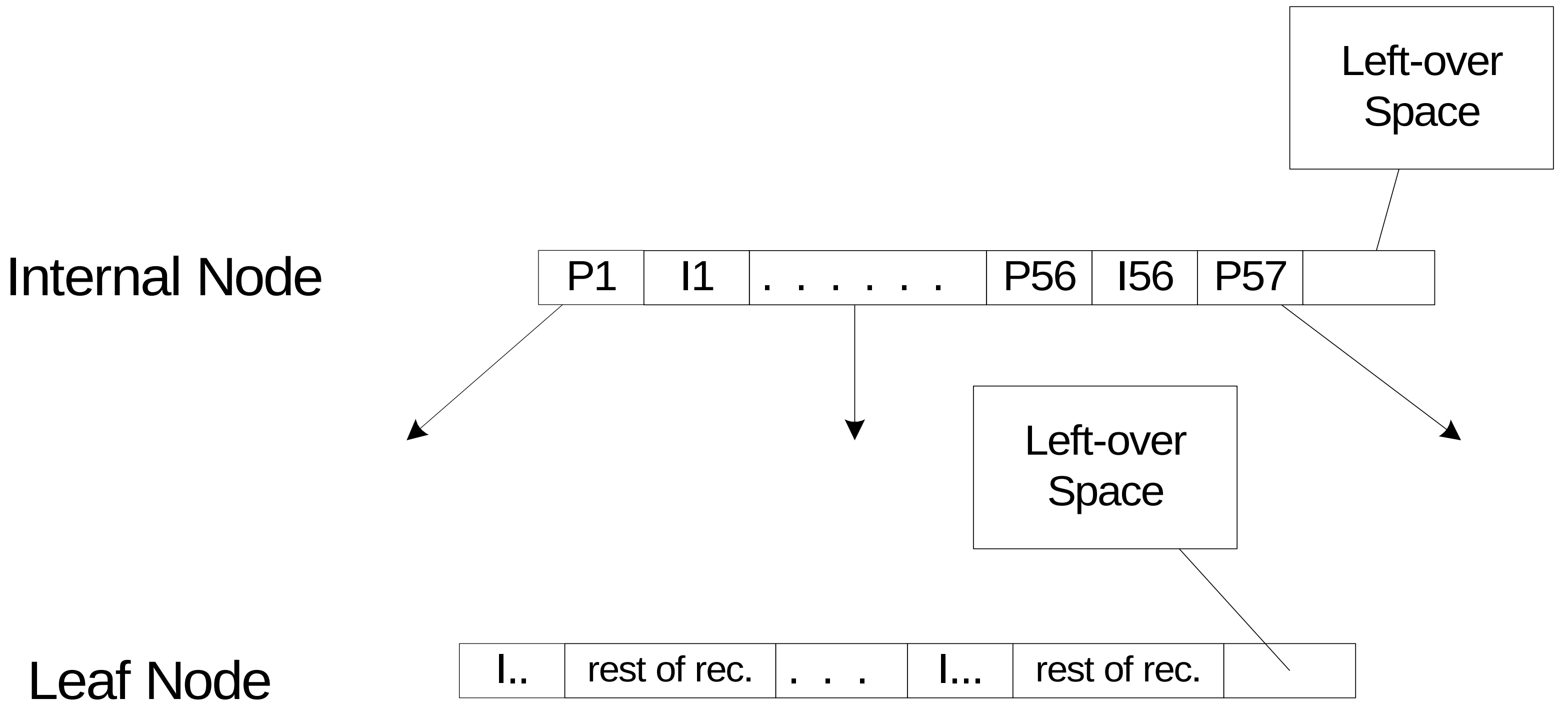
- We will now assume that we ***are*** permitted to reorganize the data file and therefore we decide to cluster it
- We will in fact treat the file as the lowest level of the tree
- The tree will have two types of nodes:
  - nodes containing indexes and pointers as before
  - nodes containing the data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



An Example

- 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-drawing-text



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

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

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

- 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

Hashing On A Disk

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

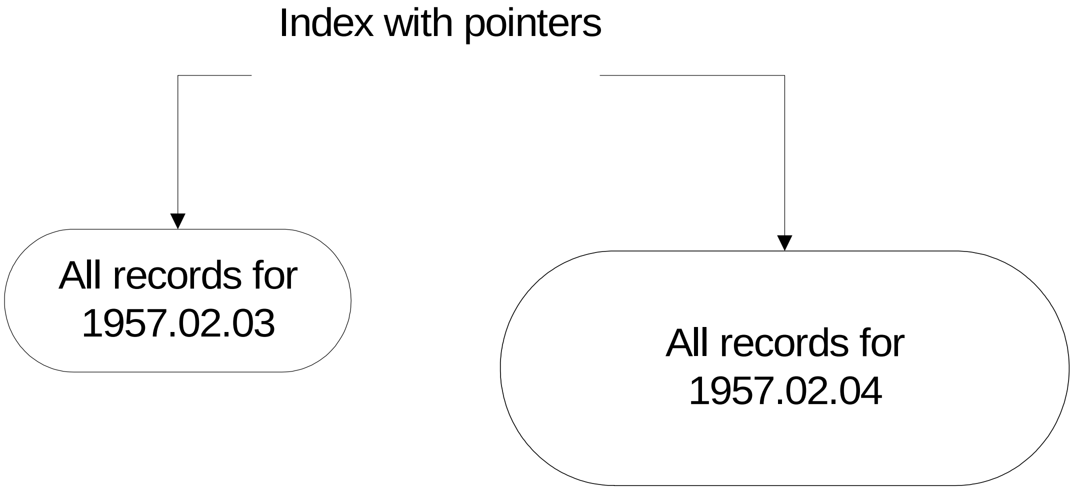
- Then, from each bucket, "very likely" we have a linked list consisting of only one block

- But how do we store the bucket array? In memory.
- So total cost to fetch is one block.

Searches On Non-Key Values

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.



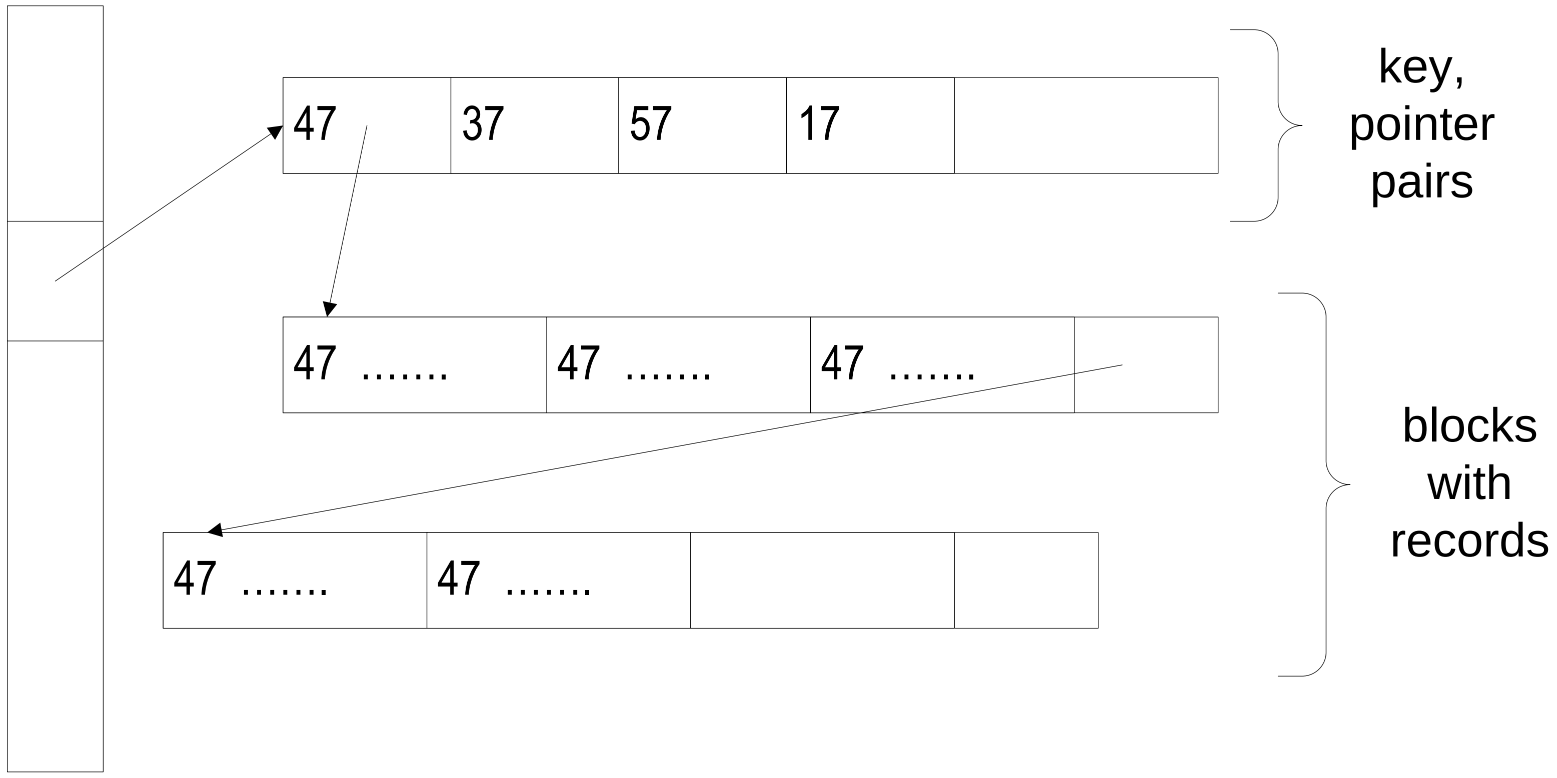
Index with pointers

All records for 1957.02.03

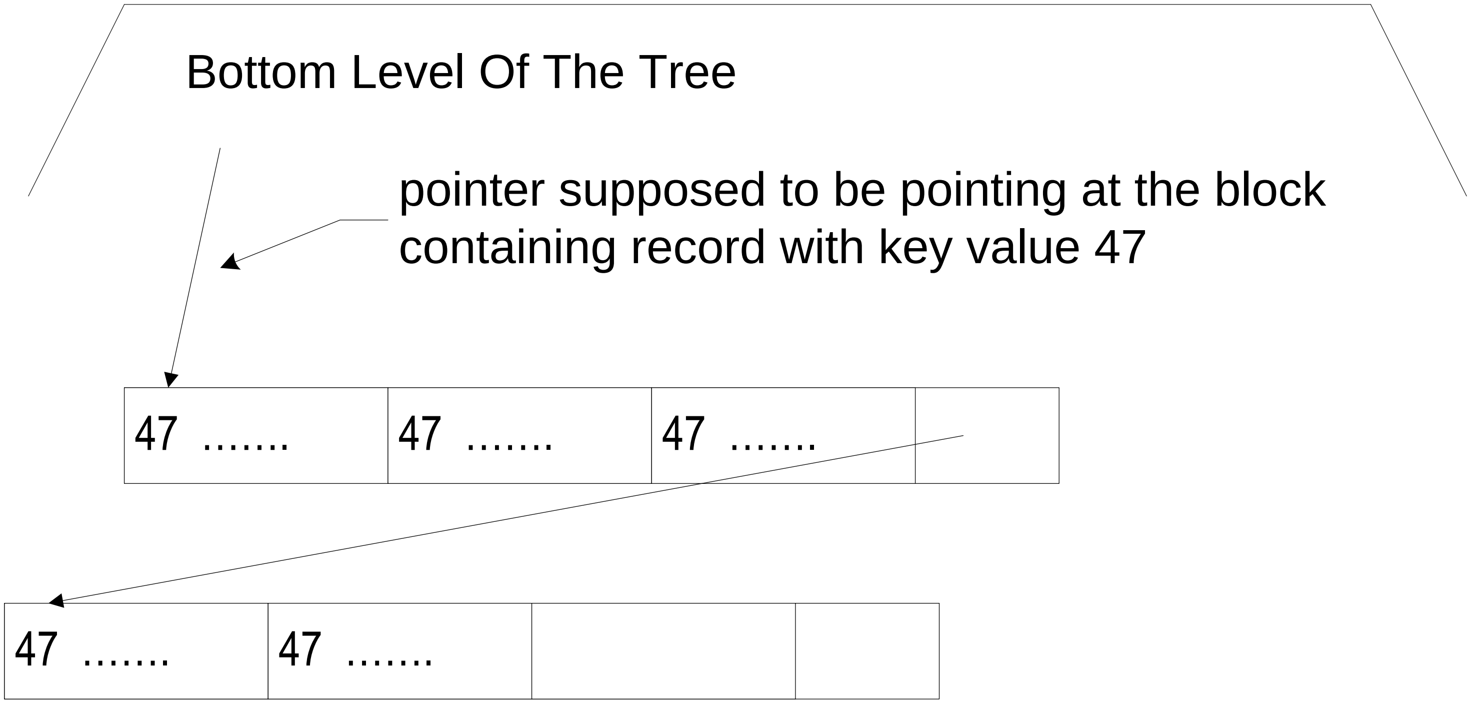
All records for 1957.02.04

Index On A Non-Key Field: Hashing

- The number of (key, pointer) pairs on the average should be “somewhat” fewer than what can fit in a single block
- Very efficient access to all the records of interest



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



Bottom Level Of The Tree

pointer supposed to be pointing at the block containing record with key value 47

Secondary Indexes

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 On Salary

- The file’s primary organization is on SSN (social security number)
- In many systems, it will be 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

- A general note: 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

Secondary Index On Salary

- 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).

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
- Revisit the idea of a covering index, especially for read-mostly data.

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

SQL And Oracle

Dennis class skip to 113

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
- Most systems allow you to specify B-tree or Hash or even more exotic indexes like R-trees or bitmaps

Oracle SQL

- 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

- 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

- 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)
  - Operations are very efficient (show circuit for Boolean ops)
- But, the ID field could be large
- Solution ideas: discretize.

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	<u>ID#</u>	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

Query Execution Concepts

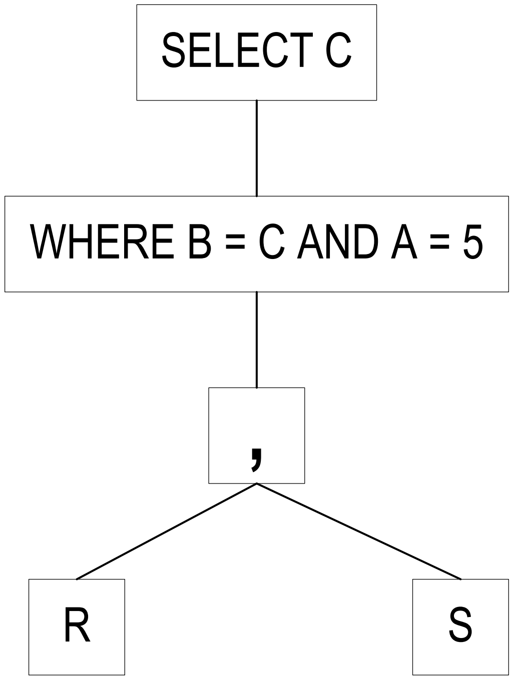
Execution Plan

- 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*** 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;



SELECT C

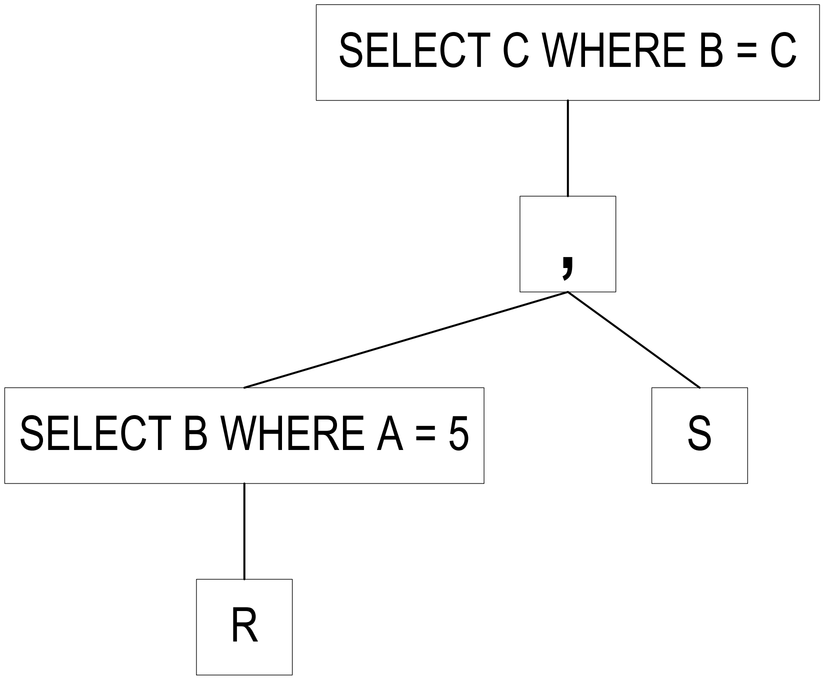
WHERE B = C AND A = 5

,

R

S

Original expression tree



SELECT C WHERE B = C

,

SELECT B WHERE A = 5

S

R

More efficient expression tree

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

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

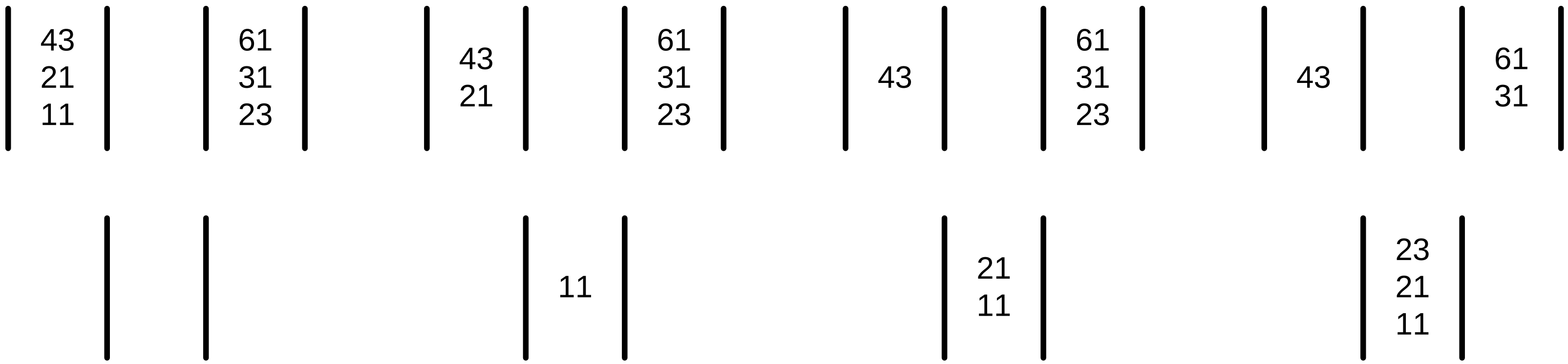
- Read 2 blocks of S into memory
  - Read 1 block of R into memory
  - Check for join condition for all the “resident” records of R and S
  - Repeat a total of 1,000 times, for all the blocks of R
- Repeat for all subsets of 2 size of S, need to do it total of 5,000 times

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

- Essentially quadratic in the size of the files

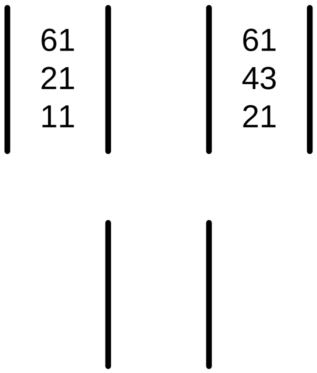
Reminder On Merge-Sort

- 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



61
21
11

61
43
21

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.

Order Of Joins Matters

- Consider a database consisting of 3 relations
  - Lives(Person, City) 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
- How would you answer the question, trying to do it most efficiently “by hand”?

- 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?

Order Of Joins Matters

- SELECT Oscar.Person Person1, Nobel.Person Person2
  FROM Oscar, Lives Lives1, Nobel, Lives Lives2
  WHERE Oscar.Person = Lives1.Person
  AND Nobel.Person = Lives2.Person
  AND Lives1.City = Lives2.City

- 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

Key Ideas

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