

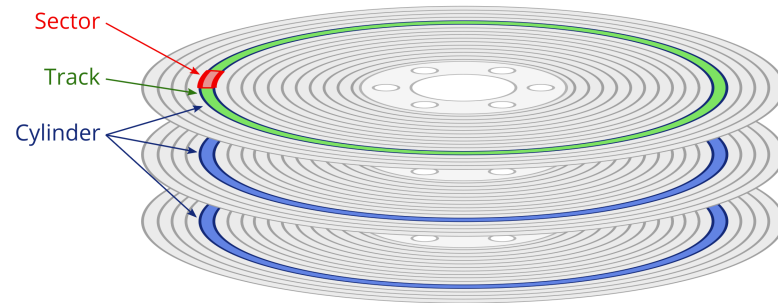
Unit 8
Physical Design
And Query Execution Concepts

Database Design

- ◆ ***Logical*** (ER model, normalization, etc.) vs ***physical*** (DBMS implementation) database design
- ◆ Main issues addressed generally in physical design
 - Storage media
 - File structures
 - Indexes
- ◆ We concentrate on
 - Centralized (not distributed) databases
 - Database stored on a disk using a “standard” file system, not one “tailored” to the database
 - Database using unmodified general-purpose operating system
 - Indexes
- ◆ The only criterion we will consider: ***performance***

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
- ◆ 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 or 4 096 B
- ◆ We assume: a virtual memory page is the same size as a block
- ◆ **A physical unit of access is always a block**



What Is A File?

- ◆ 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 or last block
 - Find the next or previous block
- ◆ 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
---	------

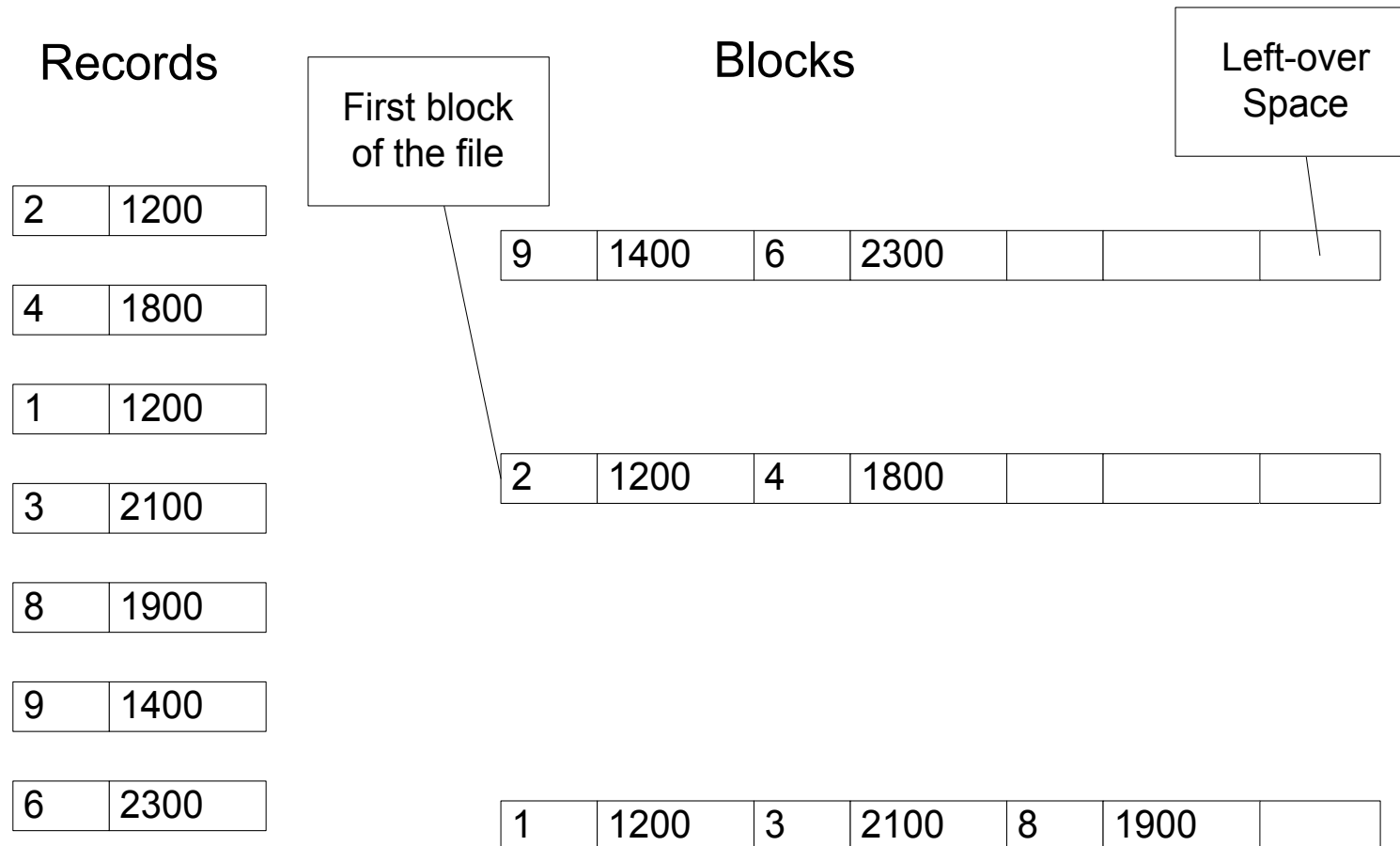
3	2100
---	------

8	1900
---	------

9	1400
---	------

6	2300
---	------

Example: Storing A Relation (Physical File)



Processing A Query

◆ 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***
- ◆ 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

Back to 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	

Back to 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
 - 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, somewhere in between

Review Of Data Structures To Store N Numbers

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

Heap (Assume Contiguous Storage)

- ◆ Finding (including detecting of non-membership)

Takes between 1 and N operations

- ◆ Deleting

Takes between 1 and N operations

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

- ◆ 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.

Hashing

- ◆ Pick a number B “somewhat” bigger than N
- ◆ Pick a “good” pseudo-random function h
 $h: \text{integers} \rightarrow \{0, 1, \dots, B - 1\}$
- ◆ Create a “bucket directory,” D , a vector of length B , indexed $0, 1, \dots, B - 1$
- ◆ 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

$$N = 7$$

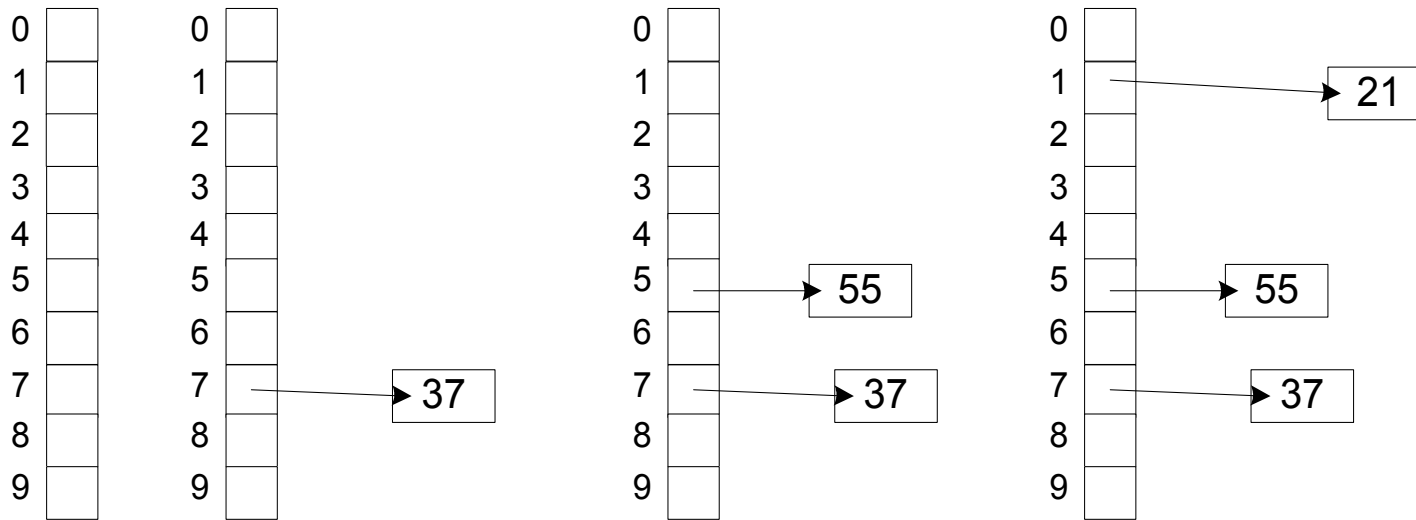
$$B = 10$$

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

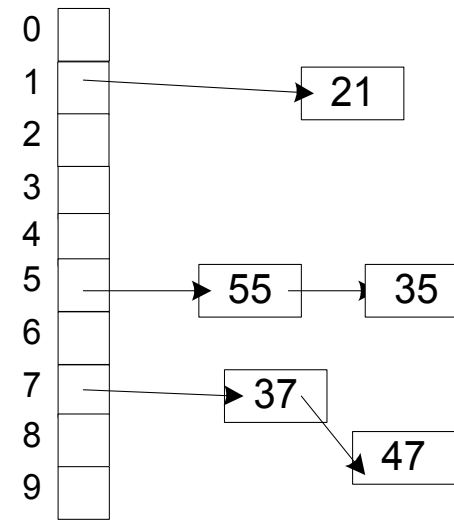
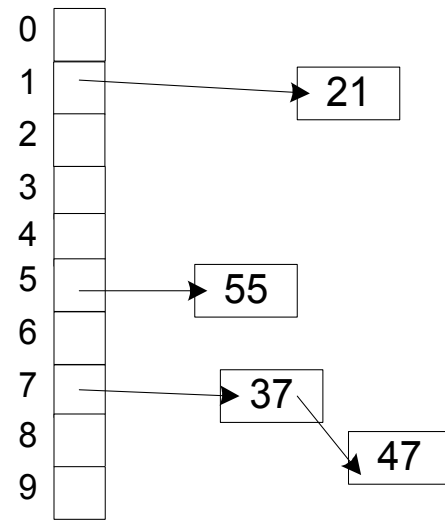
Integers arriving in order:

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

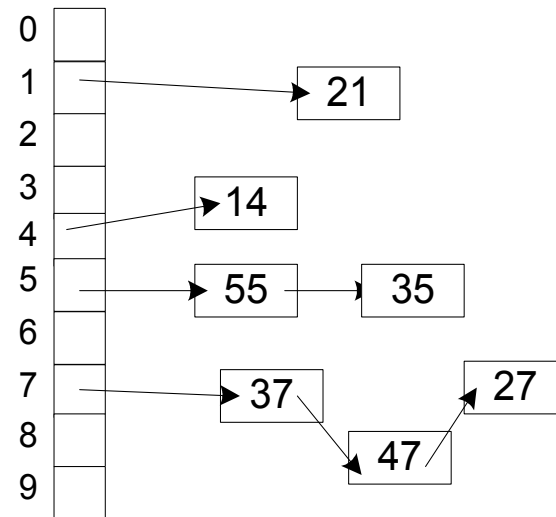
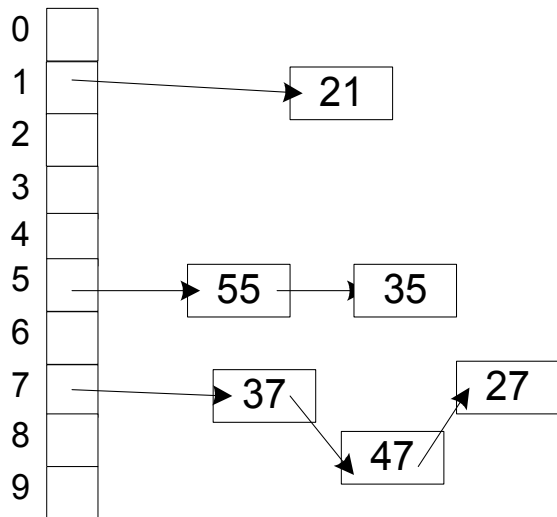
Hashing: Example Of Insertion



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"

But sometimes N is "too close" to B (bucket table becomes too small)

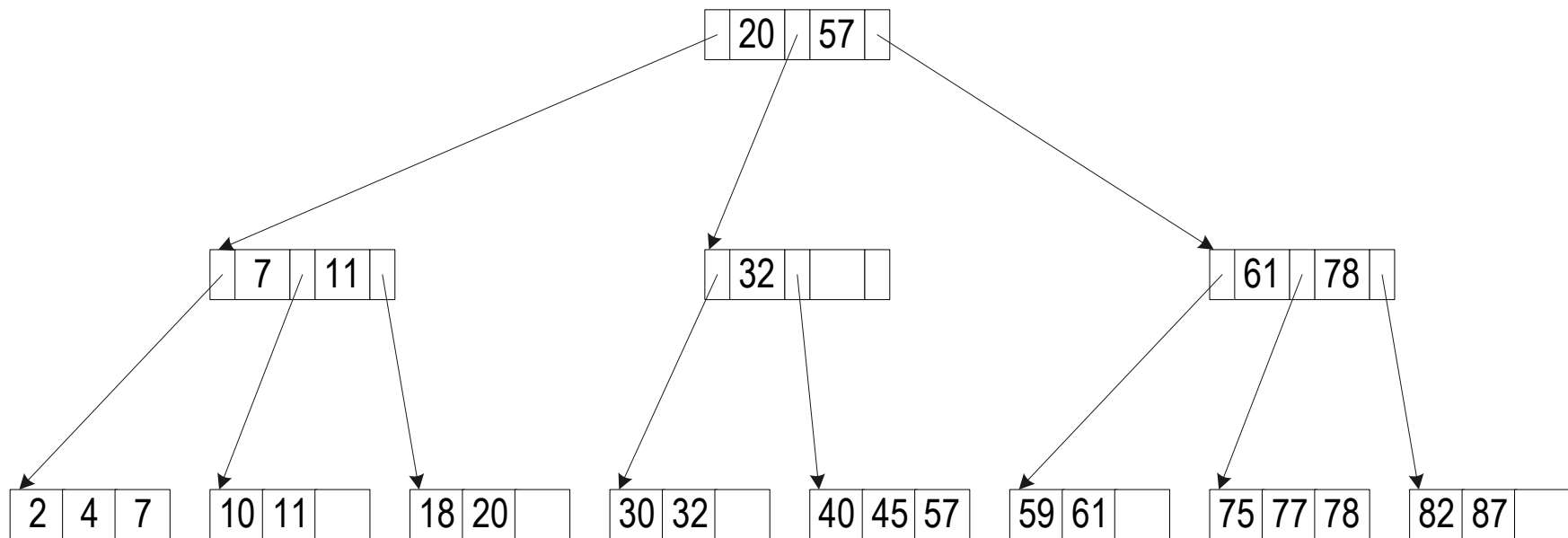
- Then, increase the size of the bucket table and rehash
- Number of operations linear in N
- Can amortize across all accesses (ignore, if you do not know what this means)

◆ Deleting

Obvious modification of "Finding"

Sometimes bucket table becomes too large, act "opposite" to Inserting

2-3 Tree (Example)



2-3 Trees

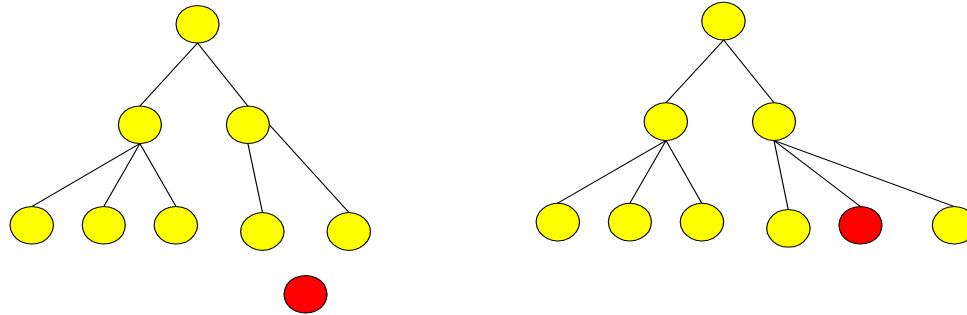
- ◆ A 2-3 tree is a rooted (it has a root) directed (order of children matters) tree
- ◆ All paths from root to leaves are of same length
- ◆ Each node (other than leaves) has between 2 and 3 children
- ◆ For each child of a node, other than the last, there is an index value stored in the node
- ◆ For each non-leaf node, the index value indicates the largest value of the leaf in the subtree rooted at the left 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: $O(\log N)$)
- ◆ We show by example of an insertion

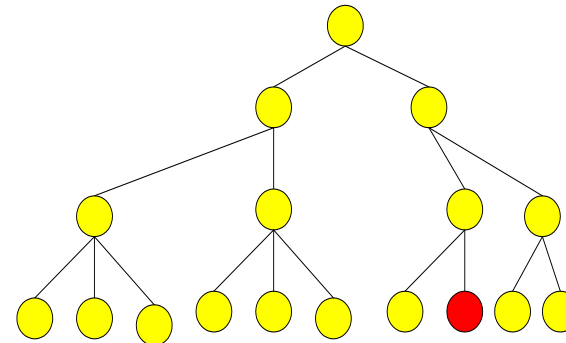
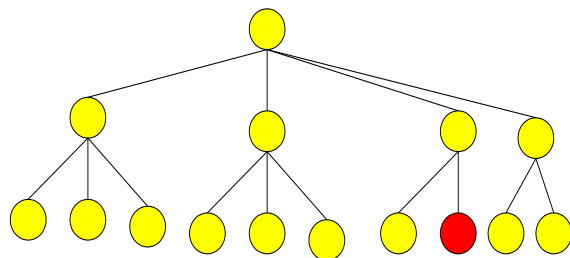
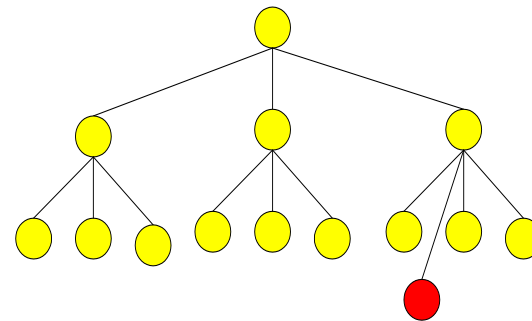
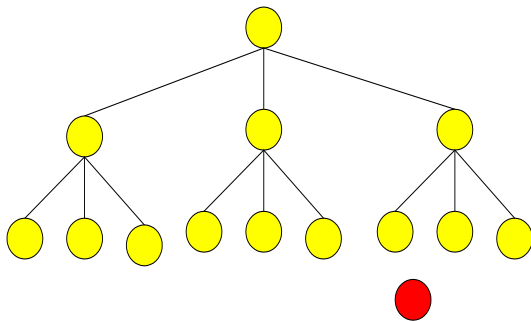
Insertion Of A Node In The Right Place

First example: Insertion resolved at the lowest level



Insertion Of A Node In The Right Place

Second example: Insertion propagates up to the creation of a new root



2-3 Trees

- ◆ Finding (including detecting of non-membership)

Takes $O(\log N)$ operations

- ◆ Inserting while keeping trees balanced

Takes $O(\log N)$ operations

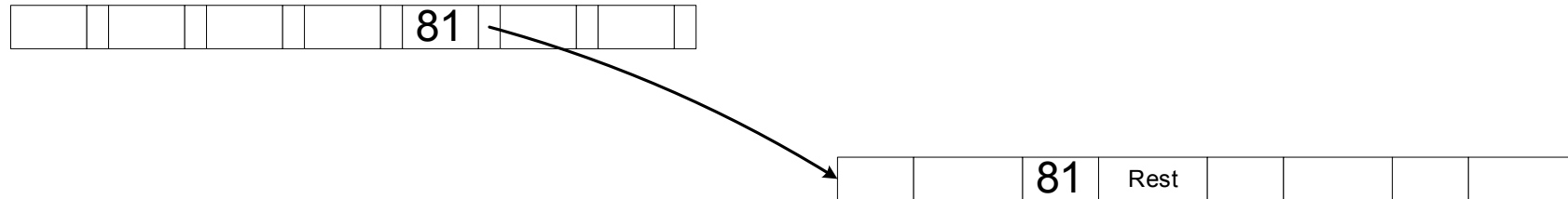
- ◆ Deleting while keeping trees balanced

Takes $O(\log N)$ operations. We do not show this as it is more complicated and sometimes balancing is not considered while doing deletions

Index And File

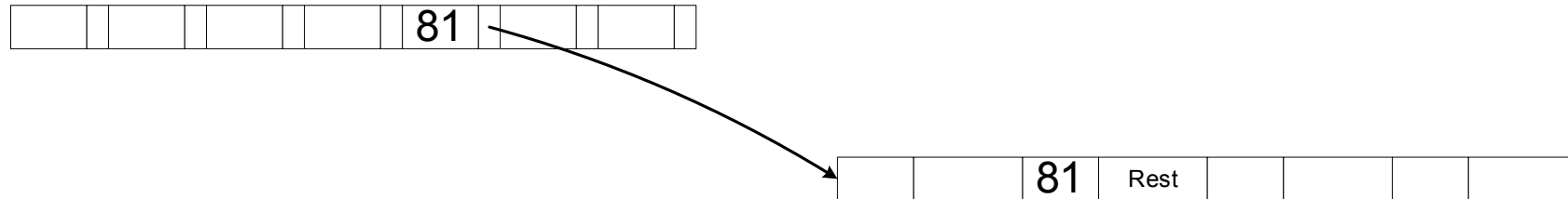
- ◆ In general, we have a data file of blocks, with blocks containing records
- ◆ Each record has a field, which contains a key, uniquely labeling/identifying the record and the “rest” of the record
- ◆ There is an index, which is another file which essentially (we will be more precise later) consists of records which are pairs of the form (key, block address)
- ◆ For each (key, block address) pair in the index file, the block address contains a pointer to a block of the file that contains the record with that key

Index And File



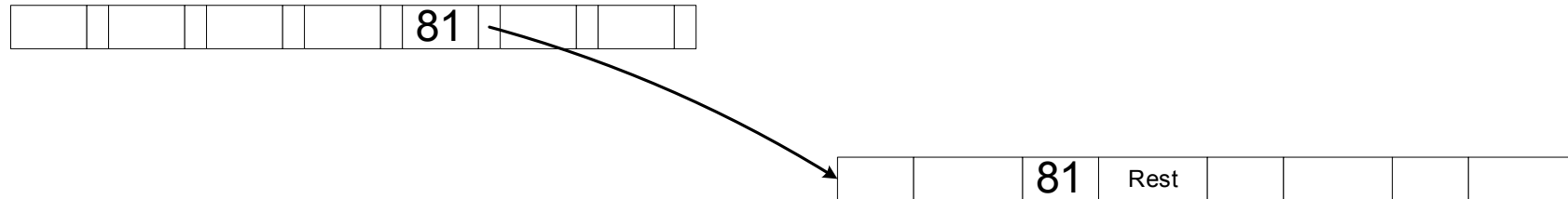
- ◆ Above, a single block of the index file
- ◆ Below, a single block of the data file
- ◆ The index record tells us in which data block some record is

Dense And Sparse



- ◆ An index is **dense** if for every key appearing in (some record) of the data file, a dedicated pointer to the block containing the record appears in (some record) of index file
- ◆ Otherwise, it is **sparse**

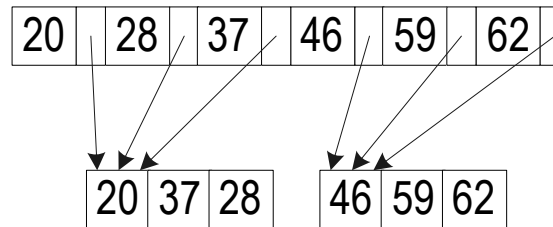
Clustered And Unclustered



- ◆ A data file is **clustered** if whenever some block B contains records with key values x and y , such that $x < y$, and key value z , such that $x < z < y$ appears anywhere in the data file, then it must appear in block B
- ◆ You can think of a clustered file as if the file were first sorted with contiguous, not necessarily full blocks, and then the blocks were dispersed
- ◆ Otherwise, it is **unclustered**

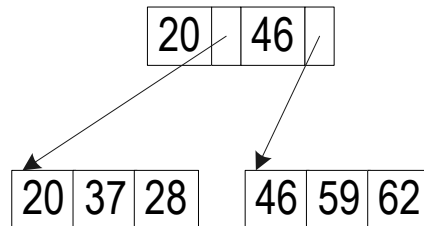
Dense Clustered Index (Dense Index And Clustered File)

- ◆ To simplify, we stripped out the rest of the records in data file



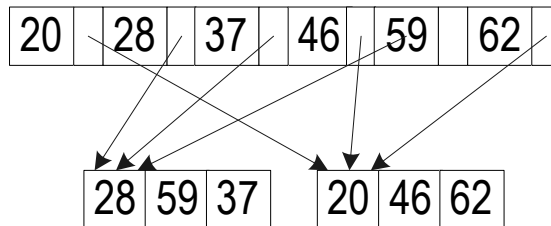
Sparse Clustered Index (Sparse Index And Clustered File)

- ◆ To simplify, we stripped out the rest of the records in data file



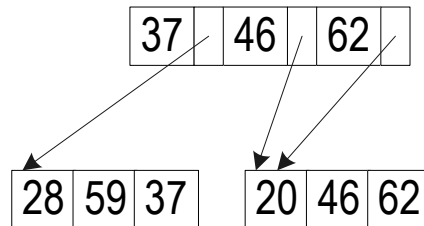
Dense Unclustered Index (Dense Index And Unclustered File)

- ◆ To simplify, we stripped out the rest of the records in data file



Sparse Unclustered Index (Sparse Index and Unclustered File)

- ◆ To simplify, we stripped out the rest of the records in data file

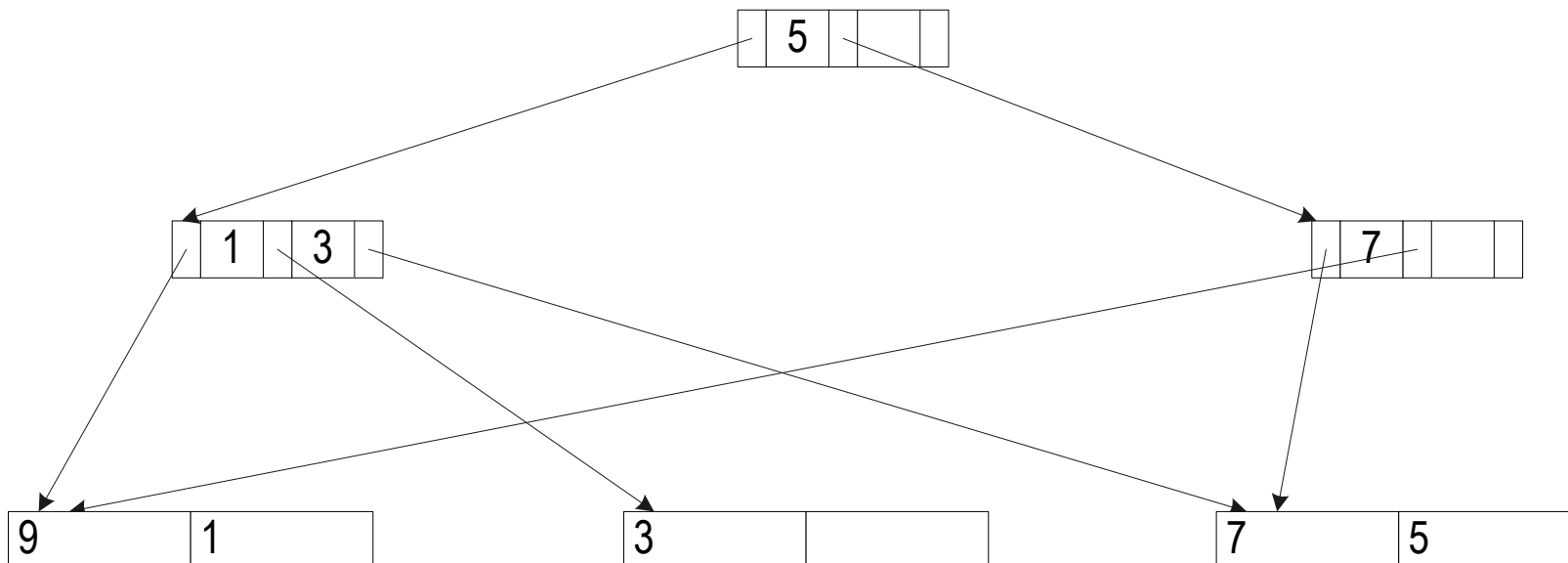


2-3 Trees Revisited

- ◆ We will now 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

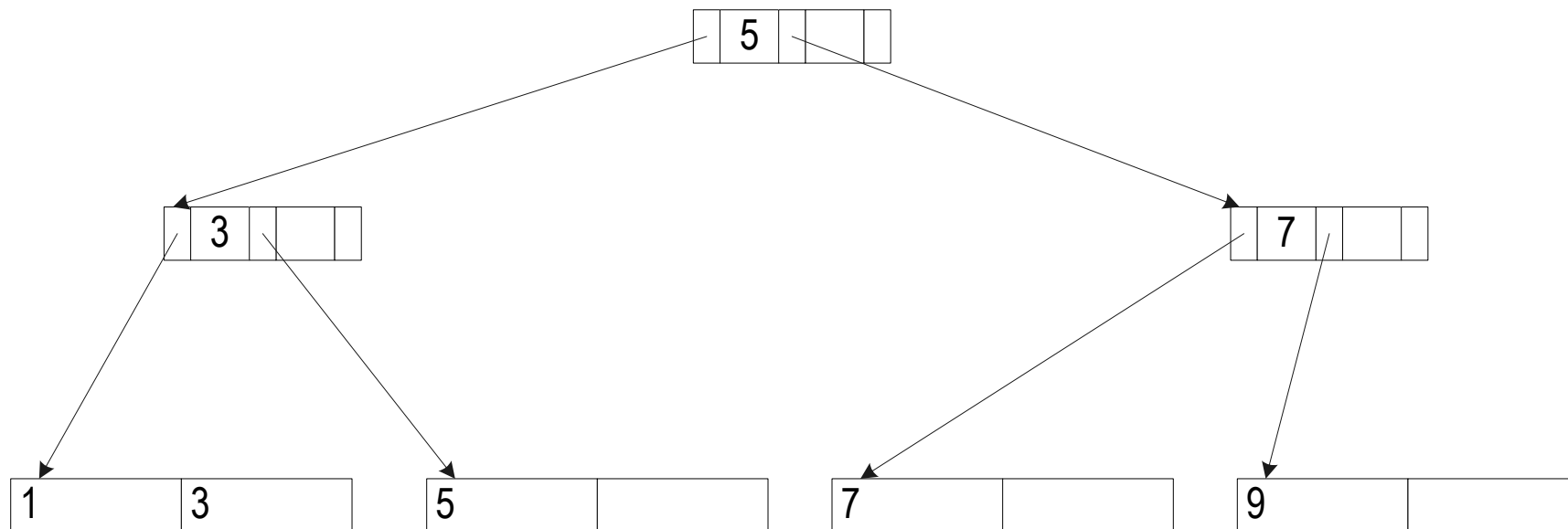
Dense Index And Unclustered File

- ◆ Pointers point from the leaf level of the tree to blocks (not records) of the file
 - Multiple pointers point at a block
- ◆ For each value of the key there is a pointer for such value
 - Sometimes the value is explicit, such as 1
 - Sometimes the value is implicit, such as 5 (there is one value between 3 and 7 and the pointer is the 3rd pointer coming out from the leftmost leaf and it points to a block which contains the record with the key value of 5)



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 largest in a block there is a pointer for such value
 - Sometimes the key value is explicit, such as 3
 - Sometimes the key value is implicit, such as 5
- ◆ Because of clustering we know where 1 is



“Quality” Of Choices In General

- ◆ Sparse index and unclustered file: generally bad, cannot easily find records for some keys
- ◆ Dense index and clustered file: generally unnecessarily large index (we will learn later why)
- ◆ Dense index and unclustered file: good, can easily find the record for every key
- ◆ Sparse index and clustered file: best (we will learn later why)

B⁺-Trees: Generalization Of 2-3 Trees

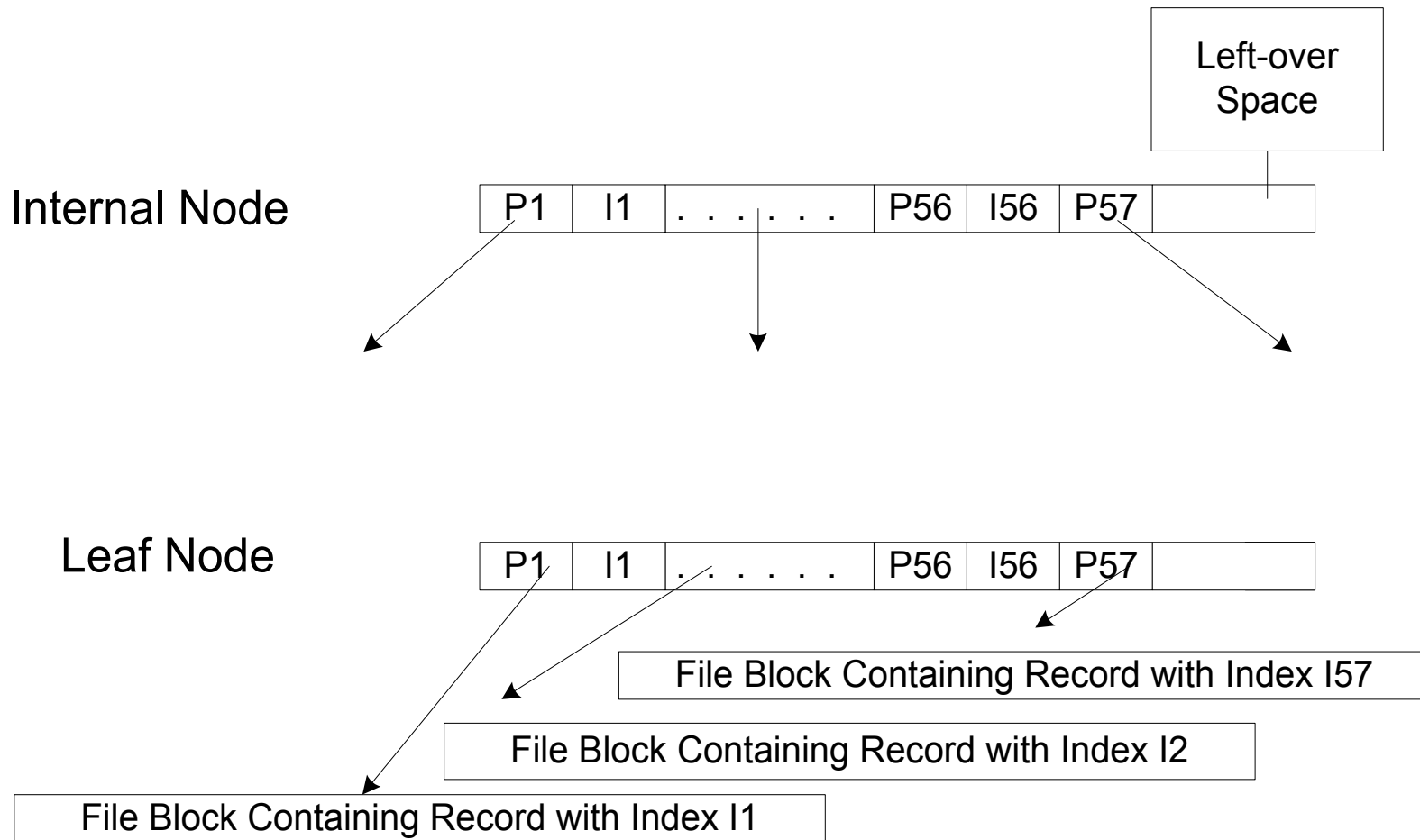
- ◆ A B⁺-tree is a tree satisfying the conditions
 - It is rooted (it has a root)
 - It is directed (order of children matters)
 - All paths from root to leaves are of same length
 - For some parameter m:
 - All internal (not root and not leaves) nodes have between $\lceil m/2 \rceil$ and m children
 - 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

- ◆ Each node consists of a sequence (P is **address** or **pointer**, I is **index** or **key**):
 $P_1, I_1, P_2, I_2, \dots, P_{m-1}, I_{m-1}, P_m$
- ◆ I_j 's form an increasing sequence.
- ◆ I_j is the largest key value in the leaves in the subtree pointed by P_j
 - Note, others may have slightly different conventions

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



(note that I57 is not listed in the index file)

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 $O(\text{depth of the tree})$ 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:
 - Disk of 16 GiB (GiB or gibibyte means 2^{30} bytes, GB or gigabyte means 10^9 bytes); very small, but a better example
 - Each block of 512 bytes; very small, but a better example
 - File of 20 million 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

- ◆ There are 2^{34} bytes on the disk, and each block holds $2^9 = 512$ bytes.
- ◆ Therefore, there are 2^{25} blocks
- ◆ Therefore, a block address can be specified in 25 bits
- ◆ We will allocate 4 bytes to a block address
 - We may be “wasting” space, by working on a byte as opposed to a bit level, but simplifies the calculations
- ◆ A node in the B-tree will contain some m pointers and $m - 1$ keys, so what is the largest possible value for m , so a node fits in a block?

$(m) \times (\text{size of pointer}) + (m - 1) \times (\text{size of key}) \leq \text{size of the block}$

$$(m) \times (4) + (m - 1) \times (5) \leq 512$$

$$9m \leq 517$$

$$m \leq 57.4...$$

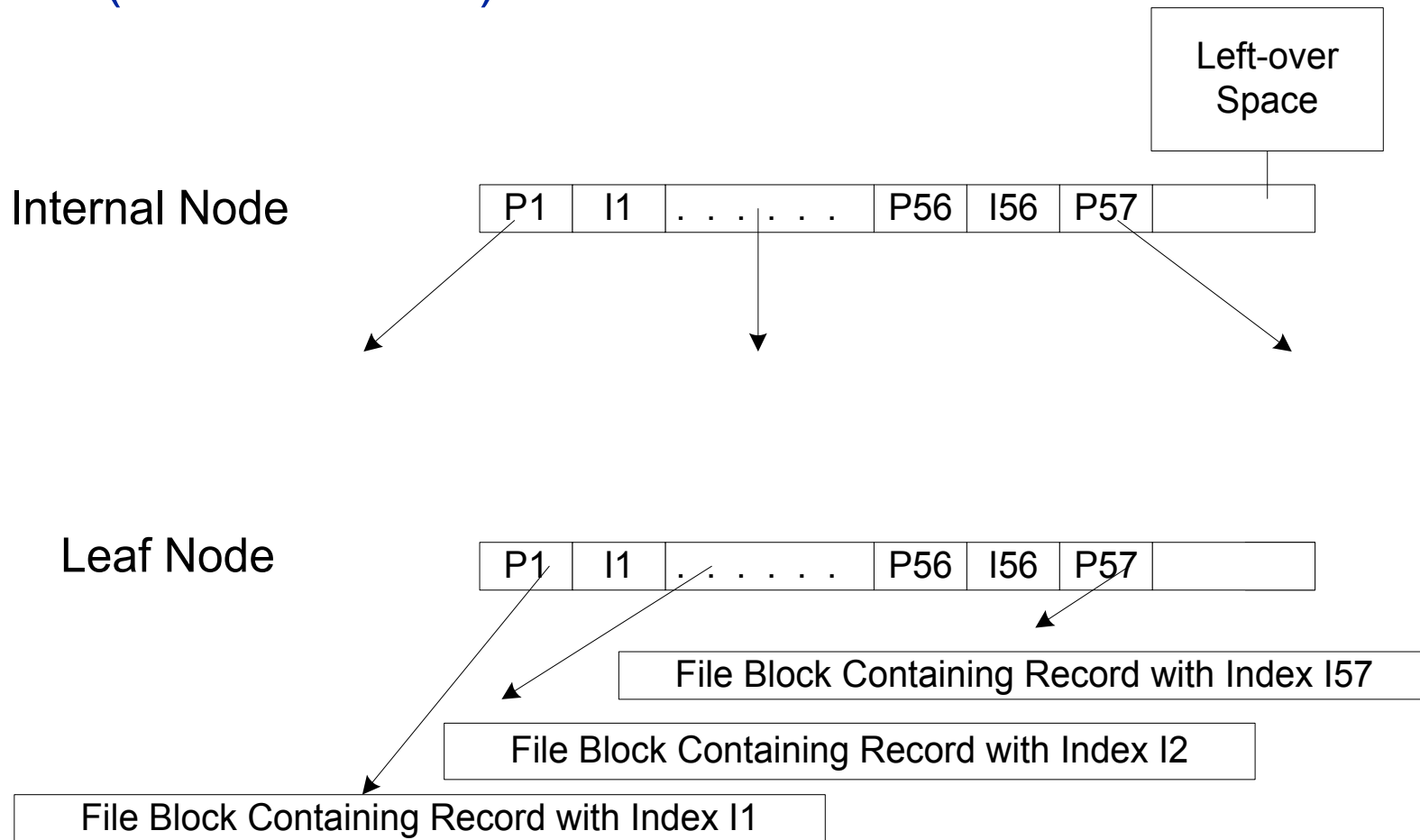
$$m = 57$$

An Example

- ◆ Therefore, the root will have between 2 and 57 children
- ◆ Therefore, an internal node will have between 29 and 57 children
 - $29 = \lceil 57/2 \rceil$
- ◆ 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 & 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)



(note that I57 is not listed in the index file)

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
- ◆ In the narrow tree, we would like the leaves to contain only 29 pointers, therefore we will have $20,000,000 / 29 = 689,655.1\dots$ leaves
- ◆ We must have an integer number of leaves
- ◆ If we round up, that is we have 689,656 leaves, we have at least $689,656 \times 29 = 20,000,024$ pointers: too many!
- ◆ So we must round down and have 689,655 leaves
 - If we have only 29 pointers coming out of each leaf we have $689,655 \times 29 = 19,999,995$ leaves: too few
 - But this is OK, we will have some leaves with more than 29 pointers “coming out of them” to get exactly 20,000,000

An Example

- ◆ In the wide tree, we would like the leaves to contain 57 pointers, therefore we will have $20,000,000 / 57 = 350,877.1\dots$ leaves
- ◆ If we round down, that is we have 350,877 leaves, we have at most $350,877 \times 57 = 19,999,989$ pointers: too few!
- ◆ So we must round up and have 350,878 leaves
 - If we have all 57 pointers coming out of each leaf we have $350,878 \times 57 = 20,000,046$ leaves: too many
 - But this is OK, we will have some leaves with fewer than 57 pointers “coming out of them” to get exactly 20,000,000

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 $1,414,562 \times 29 = 41,022,298$ pointers, but there are only 20,000,000 records
 - So it must be 5 levels
- ◆ 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	

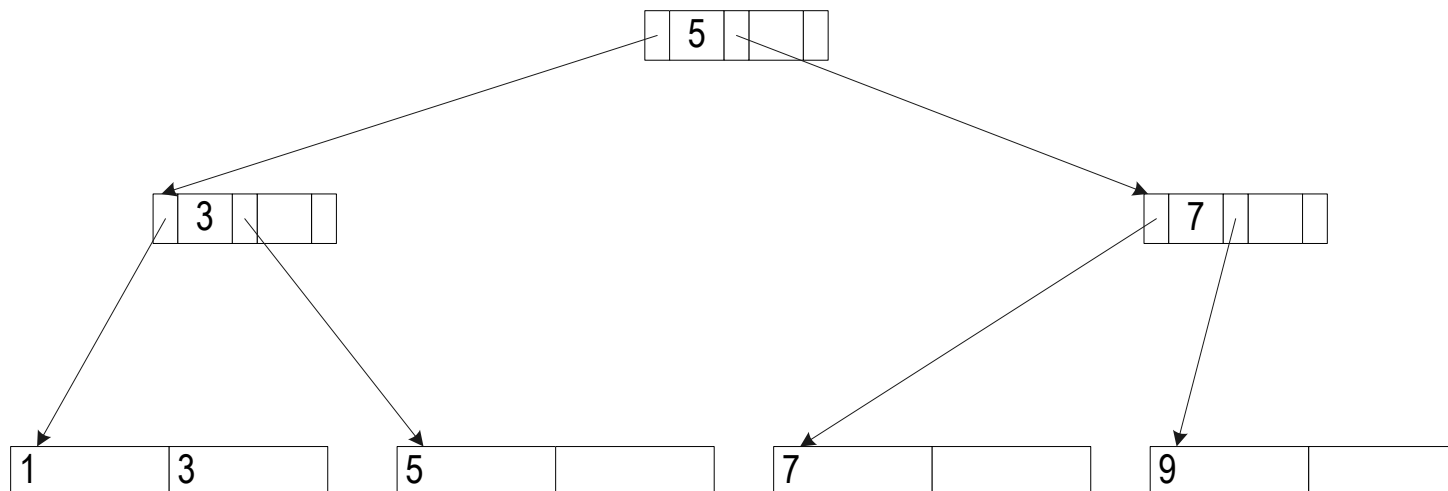
- ◆ For the wide tree, 4 levels is too little
 - If we had 4 levels there would be at most $185,193 \times 57 = 10,556,001$ pointers, but there are 20,000,000 records
 - So it must be 5 levels
- ◆ Note for the wide tree we round up the number of levels
- ◆ Conclusion: it will be 5 levels exactly in the 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

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 (1 for file + 5 levels) accesses per record
 - In fact, somewhat less, as likely 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
- ◆ 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 can again examine how a tree can develop in two extreme cases:
 - “narrowest” possible
 - “widest possible

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, 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, 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, assuming 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

How About 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”

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
- ◆ 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

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

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

Index On Several Fields

- ◆ In general, a single index can be created for a set of columns
- ◆ So if there is a relation $R(A,B,C,D)$, and 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 SSN

- ◆ Instead we could have physical pointers

(SALARY)(block address)* and/or (NAME)(block address)*

- ◆ Here the block addresses point to 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

How About SQL?

- ◆ Most commercial database systems implement indexes
- ◆ Assume relation $R(\underline{A}, B, C, D)$ 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 is SQL DDL
 - 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

Execution Plan

- ◆ Given a reasonably complex query there may be various ways of executing it
- ◆ Good Database Management Systems (such as newer versions of Oracle)
 - 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

Computing Conjunction Conditions

- ◆ A simple 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 $B = \text{'Mary'}$
- ◆ 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 $B = \text{'Mary'}$
- ◆ 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

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

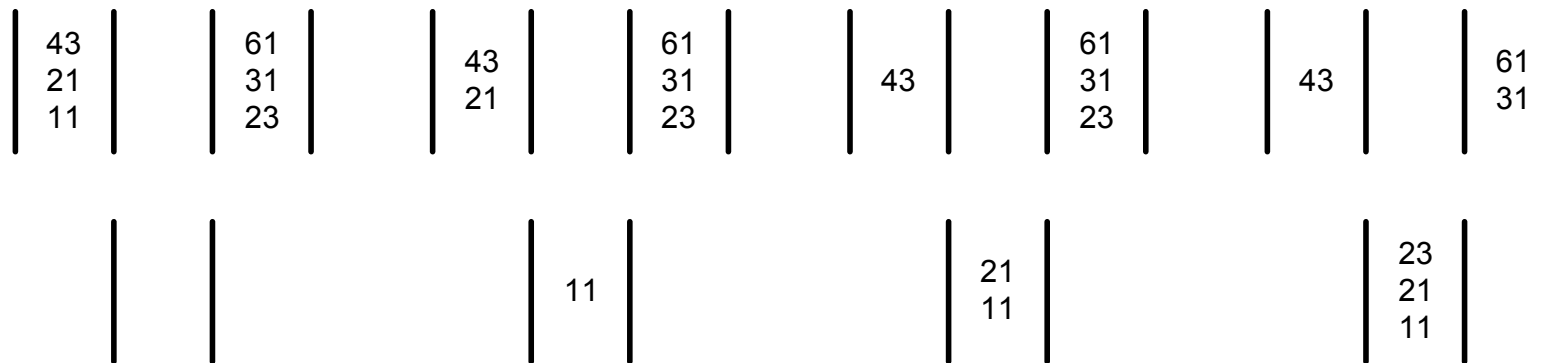
- ◆ Read 2 blocks of R into memory
 - Read 1 block of S into memory
 - Check for join condition for all the “resident” records of R and S
 - Repeat a total of 10,000 times, for all the blocks of S
- ◆ Repeat for all subsets of 2 size of R, need to do it total of 500 times
- ◆ Total reads: 1 read of R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

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

- ◆ Can we do better?

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 sequences 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	61
21	43
11	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 starting from 3-block sorted sequences, about 9 passes, where pass means read and write
 - $3 \times 2^x > 1000$? $x = 9$
 - Add 1 for initial 3-block sort
- ◆ To sort S (starting from 3-block sorted sequences) about 12 passes, where pass means read and write
 - $3 \times 2^y > 10000$? $y = 12$
 - Add 1 for initial 3-block sort
- ◆ To merge-join R and S, one pass (at most the size of R, as we join on keys, so each row of R either has one row of S matching, or no matching row at all)
 - Read R and S, write at most R
- ◆ Total cost, below 300,000 block accesses
 - $2 \times 10 \times 1000 + 2 \times 13 \times 10000 + 11000 + 1000 = 292,000$

Order Of Joins Matters

- ◆ Consider a database consisting of 3 relations
 - Lives(Person, City) about people in the US, about 330,000,000 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 following 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 (Our Old Example)

- ◆ `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`

Very inefficient

- ◆ 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

- ◆ Logical files: records
- ◆ Physical files: blocks
- ◆ Cost model: number of block accesses
- ◆ File organization
- ◆ Indexes
- ◆ Hashing
- ◆ 2-3 trees for range queries
- ◆ Dense index
- ◆ Sparse index
- ◆ Clustered file
- ◆ Unclustered file
- ◆ B⁺ trees
- ◆ Optimal parameter for B⁺ tree depending on disk and key size parameters

Key Ideas

- ◆ Best: clustered file and sparse index
- ◆ Hashing on disk
- ◆ Index on non-key fields
- ◆ Secondary index
- ◆ Use index for searches if it is likely to help
- ◆ SQL support
- ◆ Bitmap index
- ◆ Need to know how the system processes queries
- ◆ How to use indexes for some queries
- ◆ How to process some queries
- ◆ Merge Join
- ◆ Hash Join
- ◆ Order of joins
- ◆ Cutting down relations before joining them