

Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture - 28
Indexing and Hashing/3 : Indexing/3

Welcome to module 28 of Database Management Systems. We have been discussing about indexing and hashing.

(Refer Slide Time: 00:28)

The image shows a presentation slide titled "Module Recap" in red text. In the top left corner, there is a small icon of a sailboat. Below it, vertical text reads "NOC MOOCs Instructor: Prof. P. P. Das, IIT Kharagpur, Jan-Apr, 2018". The main content of the slide is a bulleted list with two items: "Balanced Binary Search Trees" and "2-3-4 Tree", each preceded by a red square bullet. In the bottom left corner, there is a small video inset showing a man in a suit and glasses. At the bottom of the slide, there is a footer with the text "Database System Concepts - 9th Edition" on the left, "28.2" in the center, and "©Silberschatz, Korth and Sudarshan" on the right. A navigation bar with various icons is visible at the bottom right of the slide.

This is the third module; in that continuation.

(Refer Slide Time: 00:38)

PPD

Module Objectives

- To understand the design of B⁺-Tree Index Files as a generalization of 2-3-4 Tree
- To understand the fundamentals of B-Tree Index Files

Database System Concepts - 6th Edition

28.3

©Silberschatz, Korth and Sudarshan

In the last module we have taken a quick look at the balanced BST and specifically a and different kind of inline data structure called 2-3-4 tree, which can be of very good use in terms of understanding B plus tree, which we want to study in this module and we will also take a quick look at the B tree.

(Refer Slide Time: 00:50)

PPD

Module Outline

- B⁺-Tree Index Files
- B-Tree Index Files

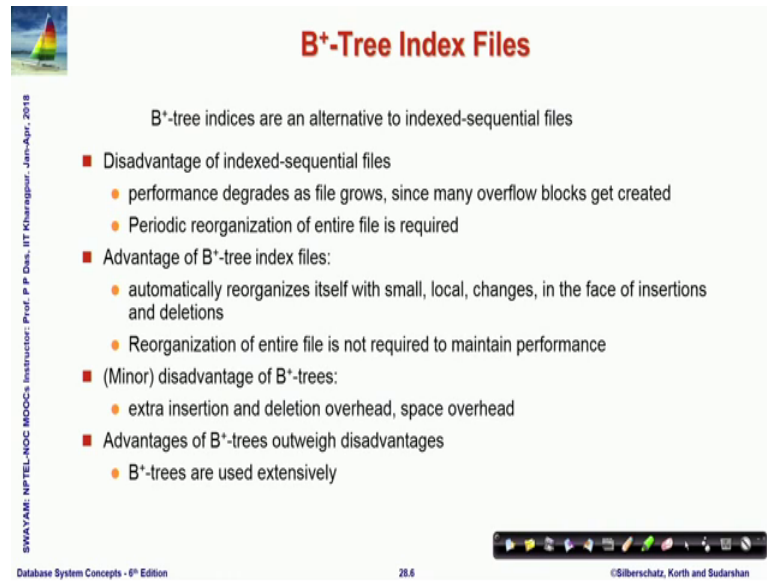
Database System Concepts - 6th Edition

28.4

©Silberschatz, Korth and Sudarshan

So, now, B plus tree is the main data structure is or one of the main data structures to be used for index files.

(Refer Slide Time: 01:00)



B⁺-Tree Index Files

B⁺-tree indices are an alternative to indexed-sequential files

- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created
 - Periodic reorganization of entire file is required
- Advantage of B⁺-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions
 - Reorganization of entire file is not required to maintain performance
- (Minor) disadvantage of B⁺-trees:
 - extra insertion and deletion overhead, space overhead
- Advantages of B⁺-trees outweigh disadvantages
 - B⁺-trees are used extensively

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P. P. Das, IIT Khargpur, Jan-April, 2018

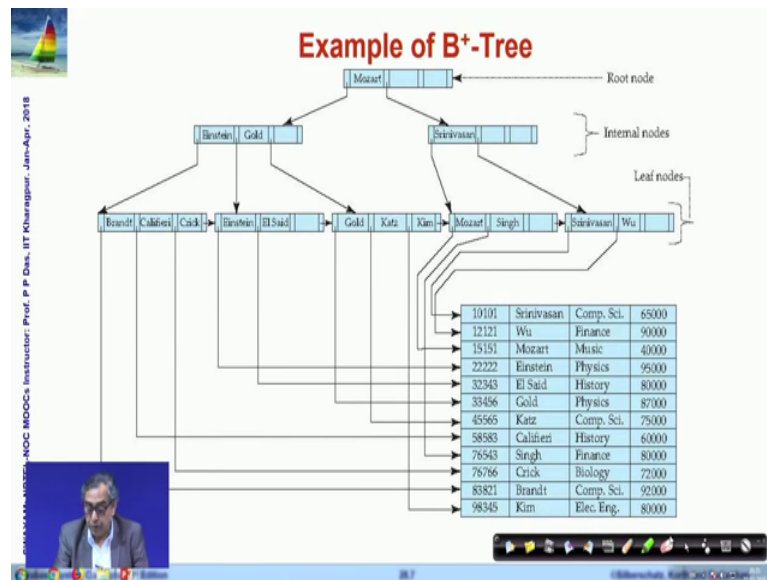
Database System Concepts - 6th Edition 28.6 ©Silberschatz, Korth and Sudarshan

So, B plus tree has now; what we have seen we have seen the ordered indexes. We have seen the index sequential files, where you could keep the index file in a sorted manner in the primary index you could build secondary index on that and so, on, but that is not an efficient way of doing things, because the performance keeps on degrading as the file grows.

Since many overflow blocks will get created, because certainly if you if you are growing, then naturally you have created say sparse index on uncertain values and if there are more records in that bucket. Then naturally you need to have linked buckets. So, periodic reorganization of the entire file becomes required which is a very costly affair.

In contrast advantage of B plus tree is it automatically reorganizes itself in small bits and pieces with local changes and so, on; whenever insertions and deletions happen and the reorganization of the entire file is not required for the purpose of maintenance. Of course, there are a little bit of disadvantage the extra insertion and deletion overhead exist for the small you know micro reorganization there is little bit of space over it, but in the face of the advantage that we get it outweighs the advantage is outweighs the disadvantages and B plus trees are used quite extensively.

(Refer Slide Time: 02:28)



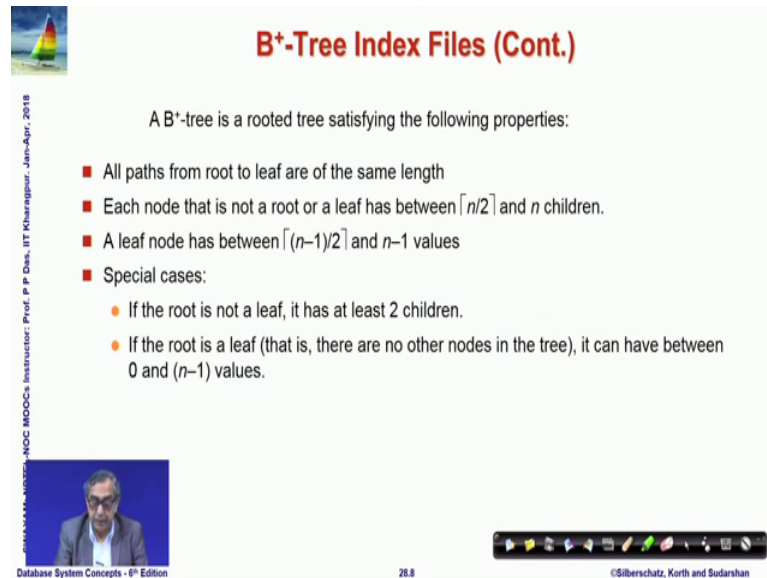
So, just recall the notion of 2-3-4 tree that we had discussed and look at this diagram. So, 2-3-4 tree have different types of node 2, node 3 node and 4 node. So, we said that there could be a node which can be only partially filled and it has a different number of children pointing to the; conditions of how different keys are ordered in that particular node.

So, here we I show an instance of a B plus tree, which is basically trying to represent this file in terms of the creating indexes. So, if the index is actually based on the name. So, this is the root node that you have and for an instance; we are taking a structure where every node can have 3 data items and 4 links and it could be it could be more it could be less, but this is just for an example. So, as you can see; so if we have this link, then on the left of Mozart, then it means all keys which are less than Mozart will be available on this link below; the link that exists here is for all keys which are greater than Mozart and less than right. Now there is nothing.

So, those will occur here. So, as you can see that Einstein gold brand all these will come on this length Srinivasan Singh wu all this come on this side the Mozart itself comes on this side. Now, if I look at this node the next level loads. Now this link has values which are less than Einstein as you can see this has values which are between Einstein and gold. So, Einstein and I set these are values which are more than gold.

So, this is this is a and as you can as you can see that though all nodes are shown to be of the same type as we had mentioned at the end of the 2-3-4 tree discussion, but it has variable number of entries. So, the number of links are between $n-1$ and n . So, n here is 4. So, you have at least either at least two entries or maximum up to 4 entries that can go on here.

(Refer Slide Time: 05:08)



B⁺-Tree Index Files (Cont.)

A B⁺-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
 - If the root is not a leaf, it has at least 2 children.
 - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.

Database System Concepts - 6th Edition 28.8 ©Silberschatz, Korth and Sudarshan

So, this is the basic observe definition of a B plus tree. All paths from root two leaf are of the same length. This is again something you should observe here, because if you if you see all of these paths all of them have the same length here then the length is 2. So, that is a basic property of 2-3-4 tree generalized into B plus tree.

So, each node that is not a root is a leaf level has between $n/2$ to n children. Leaf node has $n-1$ by 2 to $n-1$ value. And the if the root is not a leaf, then it has at least 2 children and if the root is a leaf there is no other nodes in the tree then it can have between 0 to $n-1$ values which are quite obvious.

(Refer Slide Time: 05:54)

B⁺-Tree Node Structure

- Typical node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-----	-----------	-----------	-------

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$
 (Initially assume no duplicate keys, address duplicates later)

Database System Concepts - 9th Edition

©Silberschatz, Korth and Sudarshan

So, naturally a typical node will look like this, where the pointers and key values alternate starting with a pointer P_1 , then key K_1 and so on, and ending with a pointer P_n . And the search keys are strictly ordered $K_1 < K_2 < K_3 < \dots < K_{n-1}$. These are facts that we have seen for 2-3-4 tree.

(Refer Slide Time: 06:14)

Leaf Nodes in B⁺-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, pointer P_i points to a file record with search-key value K_i .
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than or equal to L_j 's search-key values
- P_n points to next leaf node in search-key order

Brandt	Califeri	Crick	
--------	----------	-------	--

→ Pointer to next leaf node

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califeri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	60000
98345			

Database System Concepts - 9th Edition

©Silberschatz, Korth and Sudarshan

So, for a leaf node the pointer P_i points to the file record with the search key K_i and if there are two leaf nodes L_i and L_j and $i < j$, then L_i search key values are less

than or equal to the L_j search key values. So, this is the basic ordering that we had seen in 2-3-4 tree, that is what is getting generalized for a non leaf node.

(Refer Slide Time: 06:40)

Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
 - All the search-keys in the subtree to which P_1 points are less than K_1
 - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
 - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

$P_1 \quad K_1 \quad P_2 \quad \dots \quad P_{n-1} \quad K_{n-1} \quad P_n$

Database System Concepts - 6th Edition 28.11 ©Silberschatz, Korth and Sudarshan

Similarly all search-keys in the subtree which P_1 points to are less than K_1 , then for all that P_n points to are greater than K_{n-1} . And in the other cases they are between the two consecutive key values that exist between the pointers.

(Refer Slide Time: 07:01)

Example of B+-tree

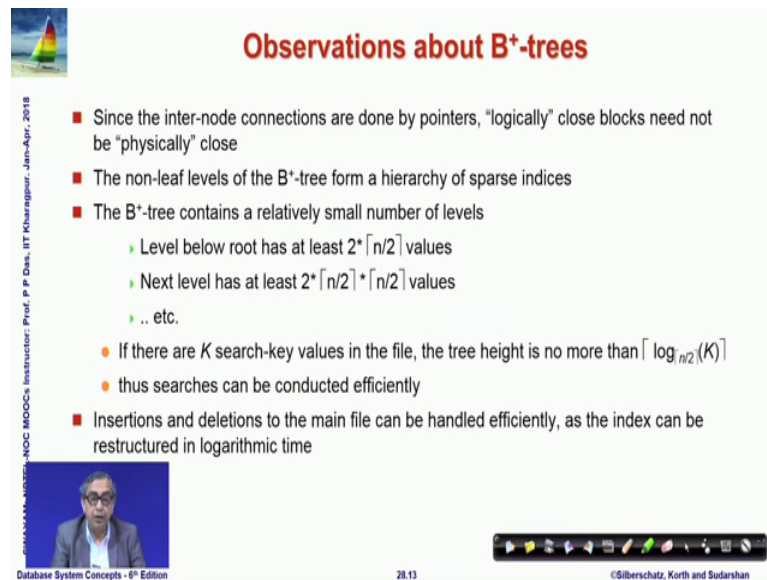
B+-tree for instructor file ($n = 6$)

- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$)
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil n/2 \rceil$ and n with $n = 6$)
- Root must have at least 2 children

Database System Concepts - 6th Edition 28.12 ©Silberschatz, Korth and Sudarshan

So, this is an example of a simple case which is n where n is equal to 6.

(Refer Slide Time: 07:10)



Observations about B⁺-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close
- The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices
- The B⁺-tree contains a relatively small number of levels
 - ▶ Level below root has at least $2^{\lceil n/2 \rceil}$ values
 - ▶ Next level has at least $2^{\lceil n/2 \rceil} * \lceil n/2 \rceil$ values
 - ▶ .. etc.
- If there are K search-key values in the file, the tree height is no more than $\lceil \log_{n/2}(K) \rceil$
- thus searches can be conducted efficiently
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time

Database System Concepts - 9th Edition

©Silberschatz, Korth and Sudarshan

So, since the inter-node connections are done by pointers, “logically” closed blocks are not “physically” close. So, that is a key idea there is a key observation about the B plus tree. So, 2 nodes the records which are logically closed are may not actually be physically close, because the pointers actually define the closeness in terms of the ordering of the values.

So, B plus tree contains relatively small number of levels, we will see what that level would be? So, what will happen; if the level below root has two values at the most at least and the below that will have n by 2 values, because every node has to be at least half field. We have said every node we will have to have n by 2 lengths to n links it cannot be less than that less than n by 2 link.

So, the next level as n by 2 , then the next level has 2 into n by 2 into n by 2 and so, on. So, every time you every level you go down you can basically increasing by a factor of n by 2 , which as you all know simply means that the number of levels or the height is $\log K$ to the base n by 2 , where K is a number of search key values that exist on the tree. So, larger the end smaller is this value. So, larger the node size is smaller is a is a height and therefore, the number of insertion number of you know access operations that need to be performed.

So, insertion, deletions to the main file can be handled efficiently as the index can be restructured in logarithmic time as you have just seen.

(Refer Slide Time: 09:01)

Queries on B+-Trees

- Find record with search-key value V
 - $C = \text{root}$
 - While C is not a leaf node {
 - Let i be least value s.t. $V \leq K_i$
 - If no such exists, set $C = \text{last non-null pointer in } C$
 - Else { if $(V = K_i)$ Set $C = P_{i+1}$ else set $C = P_i$ }}
 - Let i be least value s.t. $K_i = V$
 - If there is such a value i , follow pointer P_i to the desired record
 - Else no record with search-key value k exists

Database System Concepts - 6th Edition 28.14 ©Silberschatz, Korth and Sudarshan

So, search should be very simple, because its just an extension of what you did in 2-3-4 trees. So, algorithm is given here I will skip it, because we have already done this in detail.

(Refer Slide Time: 09:13)

Handling Duplicates

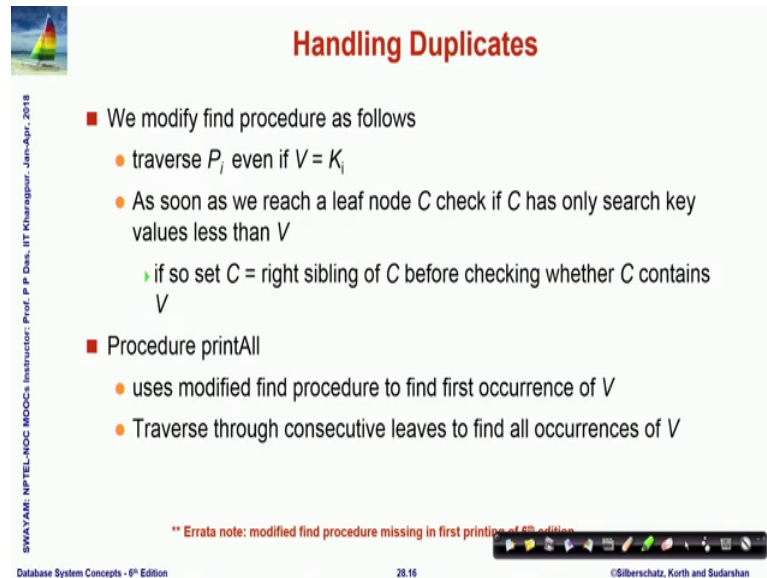
- With duplicate search keys
 - In both leaf and internal nodes,
 - we cannot guarantee that $K_1 < K_2 < K_3 < \dots < K_{n-1}$
 - but can guarantee $K_1 \leq K_2 \leq K_3 \leq \dots \leq K_{n-1}$
 - Search-keys in the subtree to which P_i points
 - are $\leq K_i$, but not necessarily $< K_i$
 - To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1} . Then in parent node K_i must be equal to V

Database System Concepts - 6th Edition 28.15 ©Silberschatz, Korth and Sudarshan

Now, what we introduced I started saying that there are no duplicates. So, the keys follow strict ordering, but the whole assumption will also hold good, if you allow the equality between the consecutive keys, but only difference is there could be multiple keys which are all equal; and if that happens then you have to use the same key value

present at the two leaf nodes and the parent will also have the same leaf node same value.

(Refer Slide Time: 09:43)



Handling Duplicates

- We modify find procedure as follows
 - traverse P_i even if $V = K_i$
 - As soon as we reach a leaf node C check if C has only search key values less than V
 - ▶ if so set $C =$ right sibling of C before checking whether C contains V
- Procedure printAll
 - uses modified find procedure to find first occurrence of V
 - Traverse through consecutive leaves to find all occurrences of V

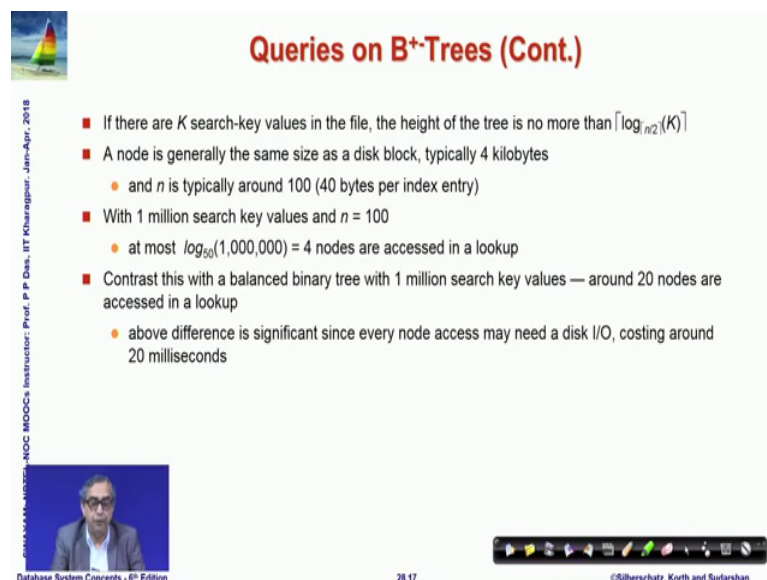
** Errata note: modified find procedure missing in first printing of 6th edition

SWAYAM: NPTEL-NOC MOCs Instructor: Prof. P. P. Das, IIT Kharagpur, Jan-Apr, 2018

Database System Concepts - 6th Edition 28.16 ©Silberschatz, Korth and Sudarshan

So, for doing in the case of such duplicates will have to a little bit modify the procedure for doing the search and say printing all values and so, on. So, you could go through that.

(Refer Slide Time: 09:58)



Queries on B+ Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{n/2}(K) \rceil$
- A node is generally the same size as a disk block, typically 4 kilobytes
 - and n is typically around 100 (40 bytes per index entry)
- With 1 million search key values and $n = 100$
 - at most $\log_{100}(1,000,000) = 4$ nodes are accessed in a lookup
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
 - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

SWAYAM: NPTEL-NOC MOCs Instructor: Prof. P. P. Das, IIT Kharagpur, Jan-Apr, 2018

Database System Concepts - 6th Edition 28.17 ©Silberschatz, Korth and Sudarshan

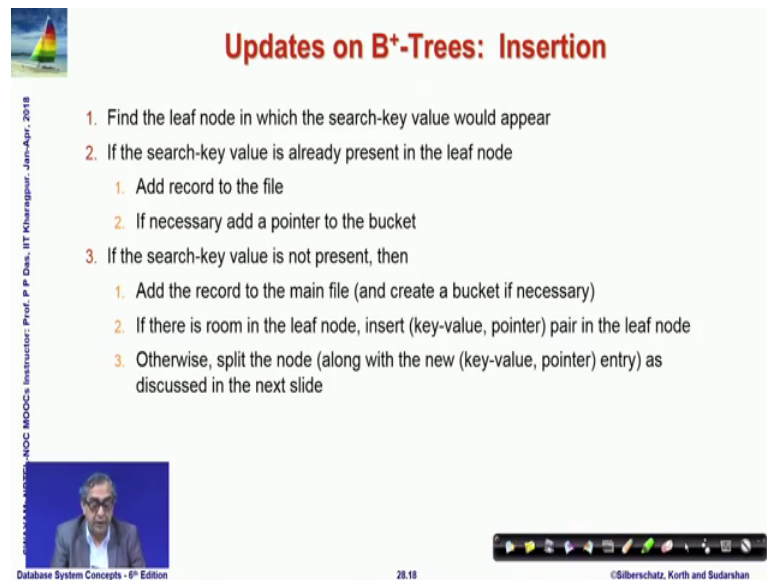
So, if there is a key search-key values in the file, then let us see what the cost is coming to actually, then the height of the tree is not more than log key to the base n by 2. So, if we say that the every node. So, how large would be the node. Now again I would remind

you that we are moving from 2-3-4 tree, which was a in memory data structure to a external data structure. So, our main cost is a disk axis. So, what would you like to make this node size, if we make the node size too small, then there will be too many nodes and every node will have to be accessed? So, as you can see this is \log to the base n by 2.

So, we benefit by making n larger the n this \log value or the height will be less, but can I make n arbitrary large then n will not fit into one disk block. So, it would it cannot be accessed in one fetch from the disk to the memory. So, we would typically like to make it is customary to make the node as the same size as the disk block, which is typically say 4 kilobyte or 8 kilobyte like that and therefore, the if that is a size then it the n will be typically around 100, because if 4 kilobytes is a is a total space and if I assume that 40 bytes per index entry, which is very typical, then n would be about 100.

So, if I assume that my index file has actually 1 million search key values to look for, then I will need 1 million to the base 100 by 250. So, 1 million \log 1 million to the base 50 which is approximately 4 node accesses in a lookup table. So, that is amazingly fast if you contrast this with binary balanced binary tree which will be \log 1 million to the base 2; which would be about 20 nodes accesses 20 disk accesses for this lookup. So, this is the co reason that B plus trees are preferred and with this if even, when you have couple of million records in a in a table you can actually manage with a very small number of node accesses for the lookup, which makes the realization of algorithms possible in the next couple of slides.

(Refer Slide Time: 12:23)



Updates on B⁺-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
 1. Add record to the file
 2. If necessary add a pointer to the bucket
3. If the search-key value is not present, then
 1. Add the record to the main file (and create a bucket if necessary)
 2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
 3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide

Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

I have discussed about how to update B plus trees talked about the insertion and the deletion process. I will skip them in the in the presentation, now because as we have discussed the process of insertion in depth in terms of the 2-3-4 tree the only difference here is that this is in a generalized framework, but follows exactly the same idea of node splitting and keeping in mind that in case of 2-3-4 tree you move from 2 to 3 and 3 to 4 node here. All that you will have to remember is you always make sure that you have every node half filled, because n by 2 is a minimum requirement.

So, you keep on inserting in a node till it becomes full, when it becomes full you cannot insert any more you divide it and split it into two nodes. So, that each one of the them become at least half filled and that is the simple logic and rest of it you can figured out by following on the 2-3-4 tree insertion. So, this is the first algorithm.

(Refer Slide Time: 13:33)

Updates on B⁺-Trees: Insertion (Cont.)

- Splitting a leaf node:
 - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lfloor n/2 \rfloor$ in the original node, and the rest in a new node
 - let the new node be p , and let k be the least key value in p . Insert (k,p) in the parent of the node being split
 - If the parent is full, split it and **propagate** the split further up
- Splitting of nodes proceeds upwards till a node that is not full is found
 - In the worst case the root node may be split increasing the height of the tree by 1

Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent

Database System Concepts - 6th Edition 28.19 ©Silberschatz, Korth and Sudarshan

Then we have shown here the strategy to splitting the node, which I have just you know discussed and the same notion of propagating the middle element of the split continues here go to next and here the examples shown in terms of the B plus tree.

(Refer Slide Time: 13:47)

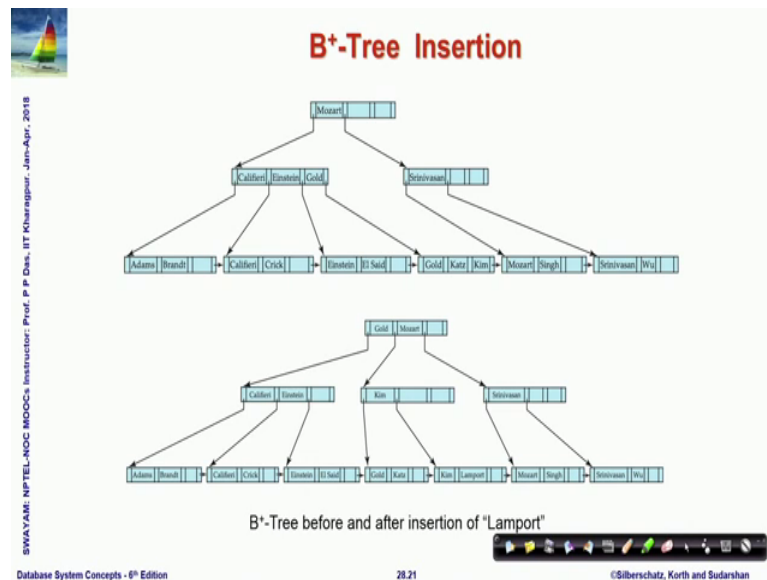
B⁺-Tree Insertion

B⁺-Tree before and after insertion of "Adams"

Database System Concepts - 6th Edition 28.20 ©Silberschatz, Korth and Sudarshan

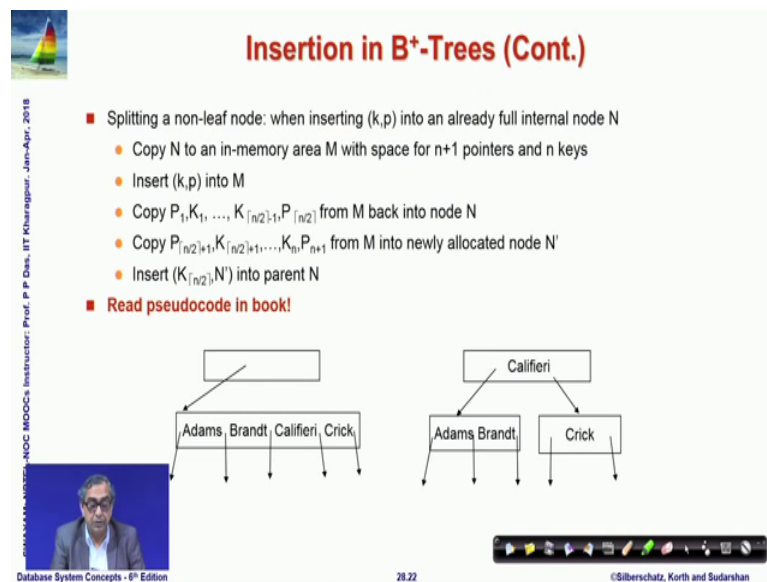
Before and after insertion of a certain key you can go through that and convince yourself.

(Refer Slide Time: 13:57)



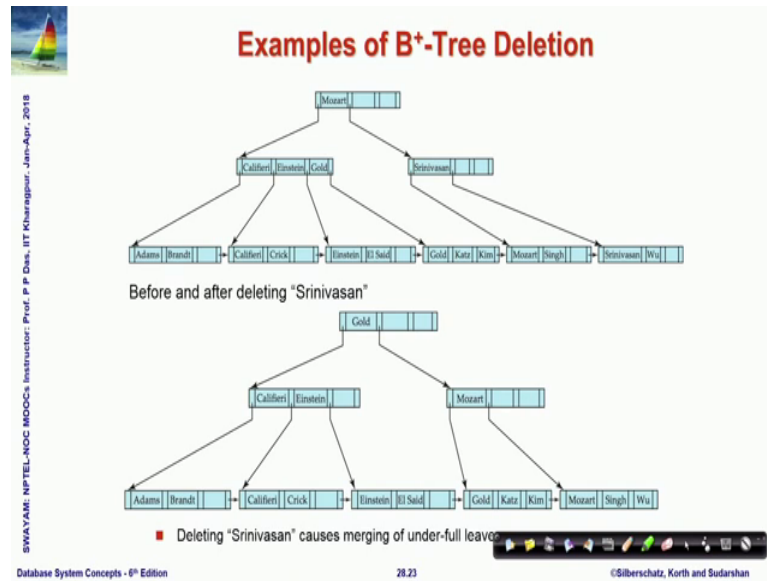
There is some more steps in the algorithm please go through them carefully and try to understand.

(Refer Slide Time: 14:04)



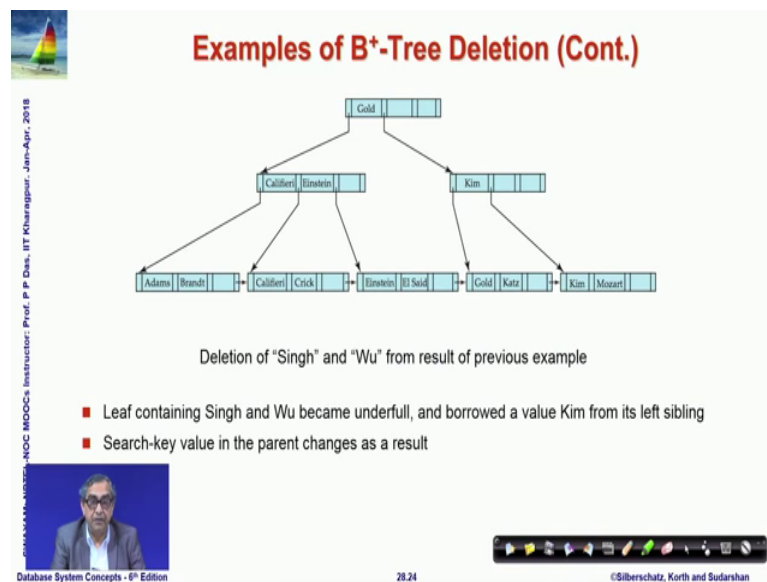
The whole process and then this is the basic algorithm written in a very cryptic pseudocode, I should say you should refer to the book actually 2, 4 and study the whole pseudocode to understand the algorithm better and work through examples as well.

(Refer Slide Time: 14:19)



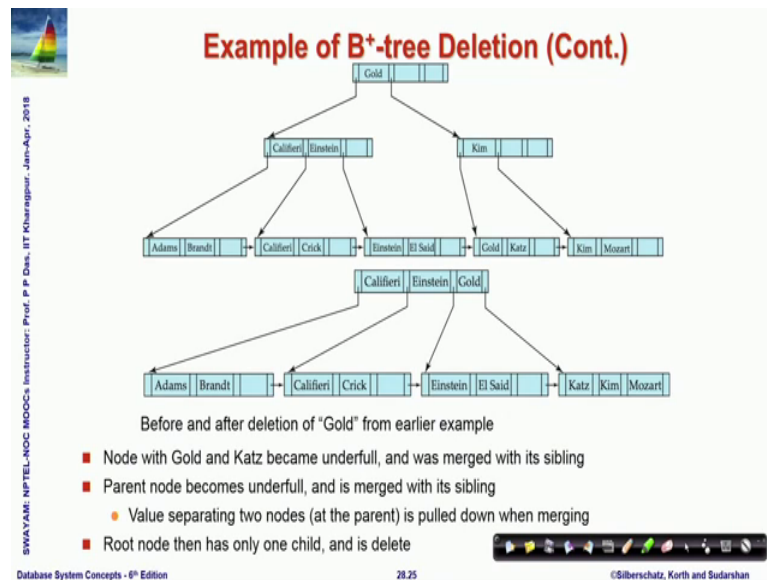
Similarly, examples of deletion in B plus tree; so the trees are shown before and after deletion of Srinivasan, then if we delete like that; now in case of in contrast to splitting.

(Refer Slide Time: 14:43)



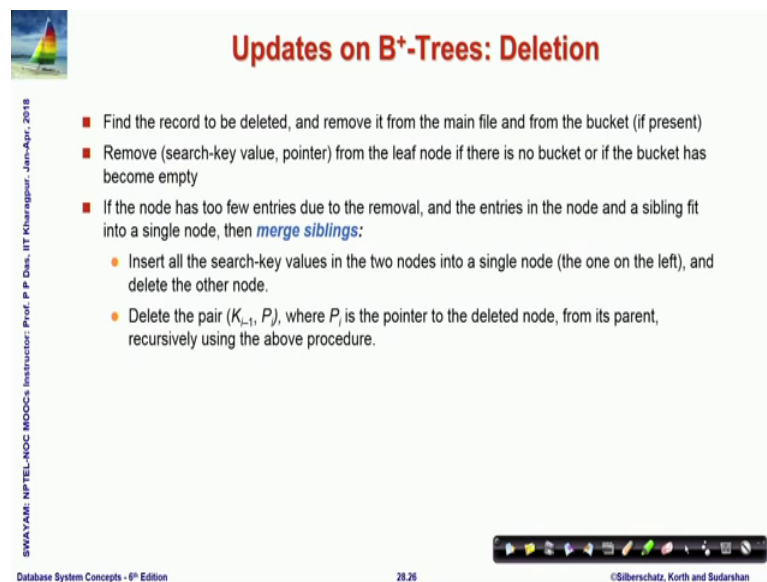
Now I will have merging of nodes which will start happening there are some more steps in the deletion shown here, please go through them and work this out they should not be you should not have any difficulty in understanding them given your background in the 2-3-4 tree.

(Refer Slide Time: 14:56)



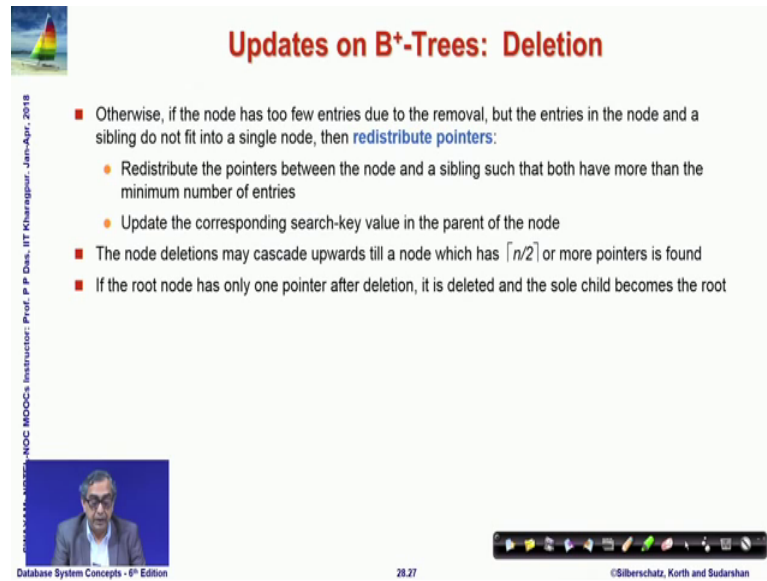
So, more steps in the deletion. So, this is the deletion process in terms of algorithmic steps and what you need to do for deletion.

(Refer Slide Time: 15:05)



So, this is all detailed here just. So, B plus tree file organization is takes care of the degradation problem.

(Refer Slide Time: 15:12)



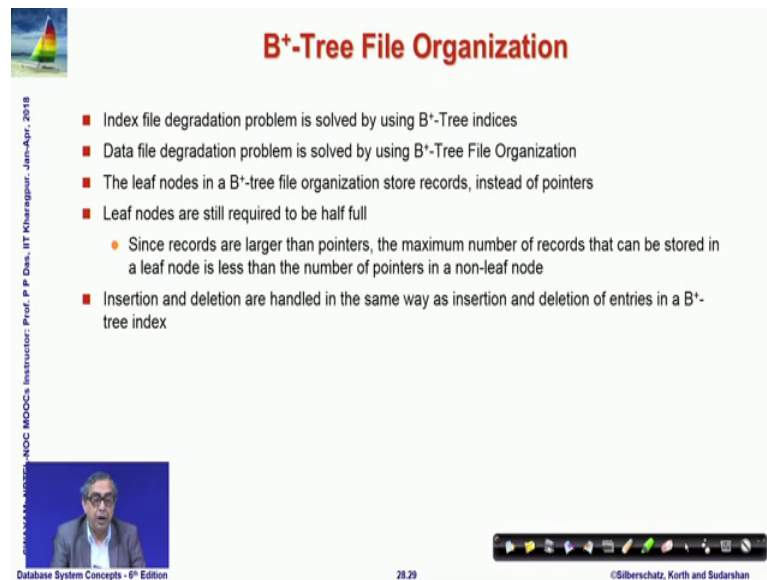
Updates on B⁺-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
 - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries
 - Update the corresponding search-key value in the parent of the node
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root

Database System Concepts - 8th Edition 28.27 ©Silberschatz, Korth and Sudarshan

In terms of the index files which would have happened, if we were used pure ordered indices like, the index sequential access method for storing the index files. So, that is now taken care of and even the data File degradation problem can also be solved by using B plus T organization.

(Refer Slide Time: 15:20)



B⁺-Tree File Organization

- Index file degradation problem is solved by using B⁺-Tree indices
- Data file degradation problem is solved by using B⁺-Tree File Organization
- The leaf nodes in a B⁺-tree file organization store records, instead of pointers
- Leaf nodes are still required to be half full
 - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B⁺-tree index

Database System Concepts - 8th Edition 28.29 ©Silberschatz, Korth and Sudarshan

So, it can be used for both maintaining the B the index as well as the actual data file and the leaf nodes in the B plus tree file organization stored the records instead of pointers. So, you finally, have the records there and the leaf nodes are still required to be half full

since they are records, but since records are larger than the maximum number of records that can be stored would be less than the number of pointers in a non leaf node insertion and deletions are handled in the same way as in the B plus tree index file.

So, here all that we are explaining that. So, far we have not explained the whole B plus tree in terms of index file organization and we are saying that you can do the same thing with the data file and only at the leaf level you will have to actually keep the data records for maintenance.

(Refer Slide Time: 16:47)

B⁺-Tree File Organization (Cont.)

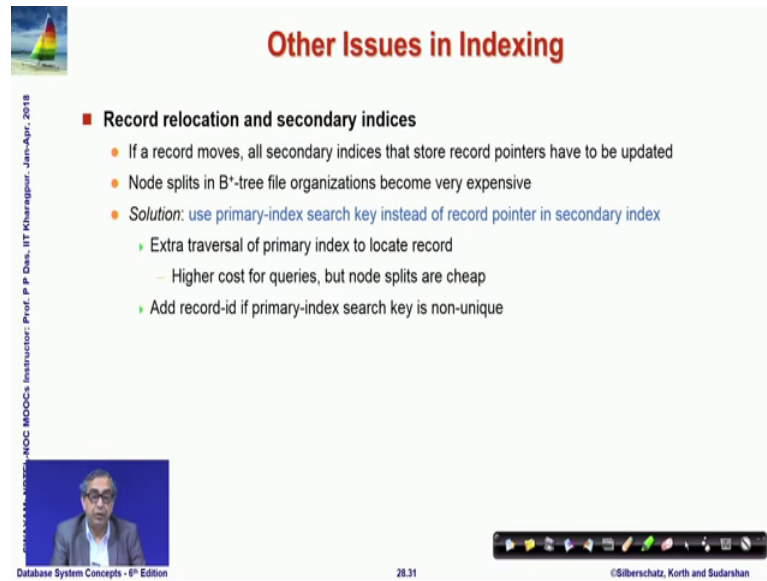
Example of B⁺-tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
 - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each having at least $\lfloor 2n/3 \rfloor$ entries

Database System Concepts - 9th Edition 28.30 ©Silberschatz, Korth and Sudarshan

So, this is showing some instances of the B plus tree organization.

(Refer Slide Time: 16:54)



Other Issues in Indexing

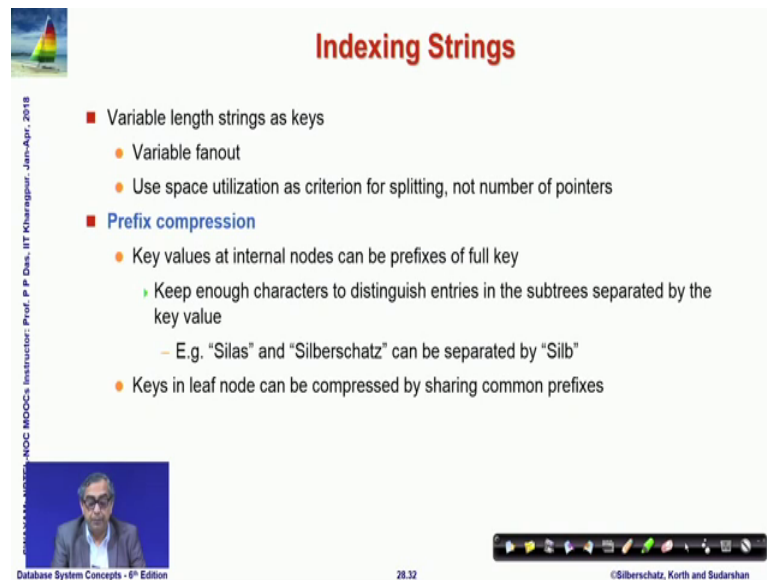
- **Record relocation and secondary indices**
 - If a record moves, all secondary indices that store record pointers have to be updated
 - Node splits in B*-tree file organizations become very expensive
 - **Solution:** use primary-index search key instead of record pointer in secondary index
 - Extra traversal of primary index to locate record
 - Higher cost for queries, but node splits are cheap
 - Add record-id if primary-index search key is non-unique

Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

So, there is a couple of other issues the record relocation and secondary index, if a record moves all secondary indices that store record pointers will also have to be updated node splits in B plus tree file organization is very expensive. So, what we do is? We use primary index search key instead of record pointer in the secondary index. So, in the secondary index we do not actually keep the direct record pointer instead, we keep the search-key of the primary index and we know that the primary index can be very efficiently searched. So, what happens is when in the secondary index when you have been able to actually find that you do not get a pointer directly to the record, but you get the search key through which you can use the primary index and actually go to that.

But with that you get yourself get rid of the requirement of maintaining different secondary index structures and getting into several record relocationess problems.

(Refer Slide Time: 18:05)



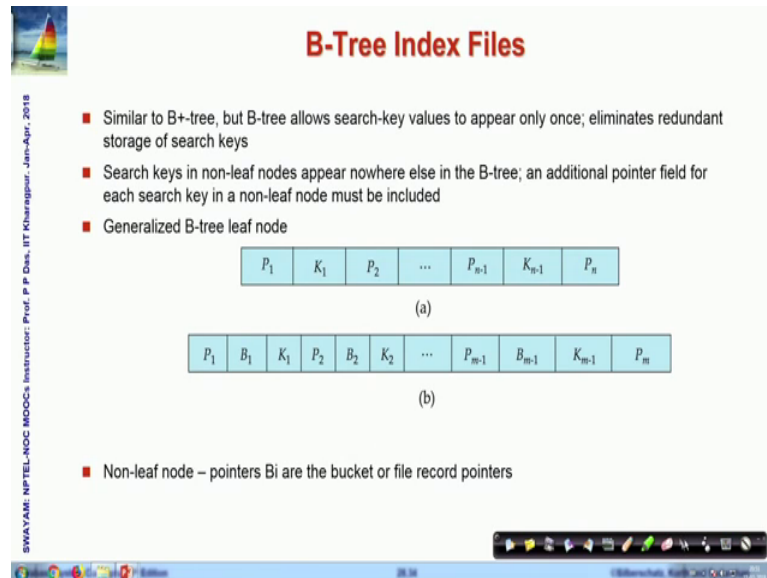
Indexing Strings

- Variable length strings as keys
 - Variable fanout
 - Use space utilization as criterion for splitting, not number of pointers
- Prefix compression
 - Key values at internal nodes can be prefixes of full key
 - ▶ Keep enough characters to distinguish entries in the subtrees separated by the key value
 - E.g. "Silas" and "Silberschatz" can be separated by "Silb"
 - Keys in leaf node can be compressed by sharing common prefixes

Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

There are your indexing also may need to take care of other issues of string your variable length string could be keys which are variable fan out and so, the general strategy in handling indexing with string is to do a kind of what is known as prefix compression. So, you kind of find out what is the shortest prefix which can distinguish between the strings. So, if you have Silas and Silberschatz then you can easily make out that Silb would be a separating string between these two. So, Silb will match with Silberschatz, but or not will match with the first one. So, you do not need to look beyond that so, we can just keep enough characters to distinguish entries in the subtree separated I by the key values and keys in the leaf node can be compressed by sharing common prefixes.

(Refer Slide Time: 19:12)



B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys
- Search keys in non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non-leaf node must be included
- Generalized B-tree leaf node

(a)

P_1	K_1	P_2	...	P_{m-1}	K_{m-1}	P_m
-------	-------	-------	-----	-----------	-----------	-------

(b)

P_1	B_1	K_1	P_2	B_2	K_2	...	P_{m-1}	B_{m-1}	K_{m-1}	P_m
-------	-------	-------	-------	-------	-------	-----	-----------	-----------	-----------	-------

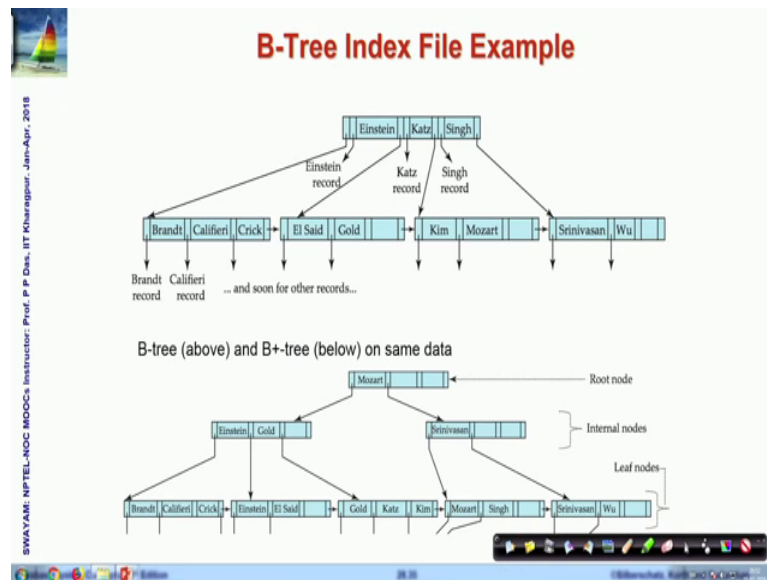
- Non-leaf node – pointers B_i are the bucket or file record pointers

SWAYAM: NPTEL-NOC MDOCS Instructor: Prof. P. P. Das, IIT Khargpur, Jan-April, 2018

So, that is a very common strategy next let us just take a quick look into the B tree index file which is another alternate possibility the basic difference between a B plus tree. And B tree allows search key values to appear only once, if you if you remember in the B plus tree your search key values where which occurs in an internal load keeps on occurring at multiple node levels also B plus B tree does not allow that the search key non leaf nodes appear nowhere else in the B tree.

So, if it does not then naturally the question is when where will the actual record value we found out for this key. So, what you do is in the node itself you introduce another field after along with the key which is the; pointer to the actual record. So, as you can see here let us get back. So, as you can see this is this was a general structure of the B plus tree node. And, now what we are doing is we are putting in separate pointers along with the key which will actually maintain the data for that key which will be pointers to the actual record, because this earlier in B plus tree all records. Finally, appear in terms of the leaf level nodes only they are their pointers come in the leaf level whereas, here the there is no repetition of the search key along the structure. So, they come wherever there.

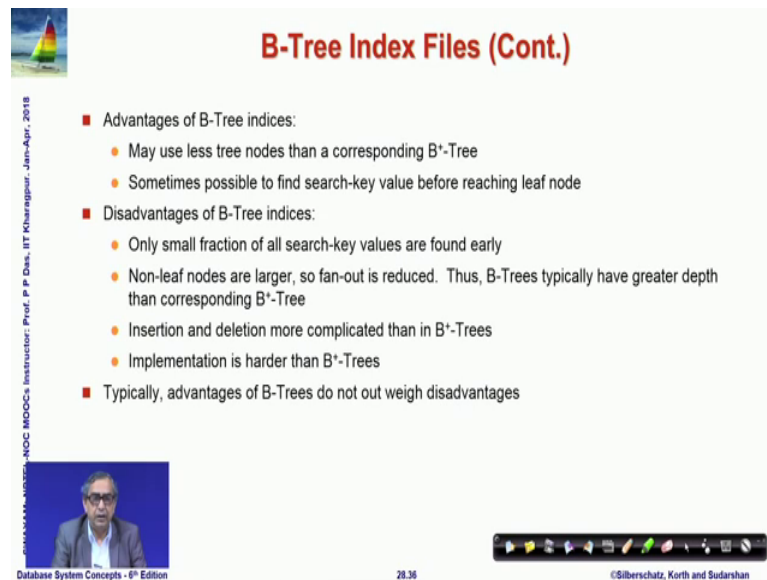
(Refer Slide Time: 20:43)



So, let me just show you an example. So, if you look into this carefully. So, this is what you have seen is a B plus tree. So, you can see that Mozart happened here, it also happened here and this is the leaf level. So, from here actually you get pointers to the; to the record for Mozart.

Similarly, Einstein happens here and it happens here Srinivasan happens here in. So, there are multiple times there happening this in contrast is a B tree representation where Einstein, if it happens then alongside with it the pointer to the Einstein's record exists, if brands happen here along with it the brands record exists and Einstein would not happen anywhere else in the tree. So, you do not have the second instance of the Einstein or this instance of the Mozart in the B tree. So, naturally that is the basic optimization that B tree does?

(Refer Slide Time: 21:48)



B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
 - May use less tree nodes than a corresponding B⁺-Tree
 - Sometimes possible to find search-key value before reaching leaf node
- Disadvantages of B-Tree indices:
 - Only small fraction of all search-key values are found early
 - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B⁺-Tree
 - Insertion and deletion more complicated than in B⁺-Trees
 - Implementation is harder than B⁺-Trees
- Typically, advantages of B-Trees do not outweigh disadvantages

Database System Concepts - 9th Edition
©Silberschatz, Korth and Sudarshan

. So, it is advantages it may use less notes than the corresponding B plus tree sometimes it is possible to find the search key value even before reaching the leaf node. So, search could be efficient, but it does have a lot of disadvantages, because what happens is only small fraction of all search key values are actually found early non leaf nodes are larger.

Now, because you have pointers to the data as well, so the fan out gets reduced which means that the number of children you can have is gets reduced. So, though you are expecting to get a benefit, because you are not having to go to the leaf every time, but you pay off because your fan out gets less. So, if your fan out get less naturally the tree has a greater depth. Now, because you can you are fanning out less number of children at every node. So, it has a greater depth. So, eventually your cost increases the naturally the deletions insertions are more complicated than in B plus tree and implementation is more difficult.

So, typically the advantages of B tree do not outweigh the disadvantages.

(Refer Slide Time: 23:01)

Module Summary

- Understood the design of B⁺-Tree Index Files in depth for database persistent store
- Familiarized with B-Tree Index Files

SWAYAM: NPTEL-NOC MDOCS Instructor: Prof. P P Das, IIT Kharagpur, Jan-Apr, 2018

Database System Concepts - 8th Edition 28.37 ©Silberschatz, Korth and Sudarshan

So, it is not very frequent that you will use b trees, but they are use at times, but that is not a very common thing and we stick to B plus tree for both of the data file as well as index file storage. So, in this module you have understood the design of B plus index B plus tree index files in depth for the purpose of data base persistent store and I would again remind you that whole discussion of how B plus tree is organized and how operations of access insert delete are done in B plus tree. I have introduced them in keeping in parallel with the simpler in memory data structure for this which is a 2-3-4 tree discussed in the last module.

So, while going through the insertion deletion processes of B plus tree, if you have difficulty following I would request that you go back to the 2-3-4 tree that is kind the simplest situation that can have that can occur and understand that and then you come back to the specific points in the B plus tree algorithm and also always keep in mind. When you refer to 2-3-4 tree for understanding also always keep in mind that in case of B plus tree all load types are same and the basic requirement is every node must be at least half full all the time except of course, for the root and in addition we have also familiarized with B tree and reason that B tree possibly is not a very powerful is not powerful enough it does not give enough advantages so, that to we would like to use it in place of B plus tree.