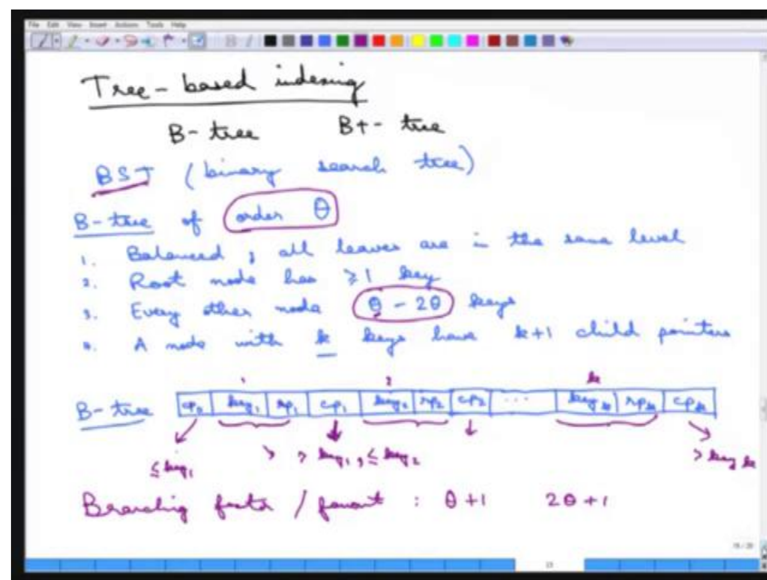
Fundamentals of Database Systems
Prof. Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 19
Database Indexing: Tree-Based Indexing

Next we will start on one very important type of indexing, which is a Tree Based Indexing.

(Refer Slide Time: 00:15)



And we will cover two important structures in this, the B tree and the B plus tree. So, these are for indexing the data on the disk. So, these are secondary storage or the disk is sometimes called the secondary storage. So, these are disk aware or designs and I am assuming that you are familiar a little bit with a binary search tree. So, this is essentially a binary search tree built for the disk.

So, if you recall the main design of a binary search tree is that corresponding to a node, the data is divided into two parts and the left sub tree of that node, the left child gets all the data that is less than the key that is stored in the node and the right sub tree or the right child gets all the data that is larger than the key that is stored in the node. So, that is the binary search tree and we will build upon the BST, so that is the Binary Search Tree and the B tree and the B plus tree builds upon that and how is it stored.

So, the B tree let us take, the B tree there is an order corresponding to a B tree, B tree of a particular order we will see what the order is, suppose the order is theta has the following properties. First of all the B tree is a balanced structure and balanced and all leaves are in the last level and final level, all leaves are in the... So, balanced meaning all leaves are in the same level, so this is the balanced. Then, there is a root node has at least one key it must contain of course, at least one key. Any other node, every other node will have between theta to 2 theta keys. So, it will have number of keys which is between theta to 2 theta.

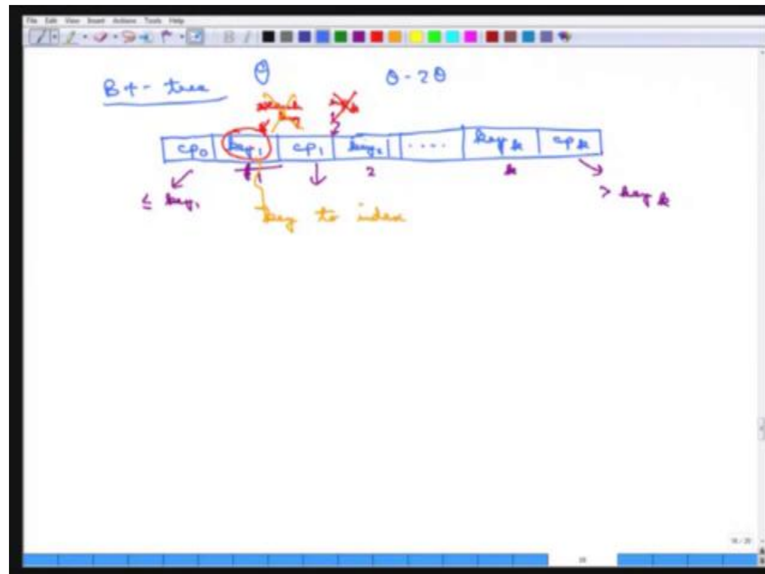
And a node with k keys will have k plus 1 child pointers, so the internal node design of a B tree looks like the following is that there is a key, let us say key 1. Now, corresponding to key 1 there is a child pointer 0 and then there is a child pointer, there is a record pointer 1 corresponding to the key 1 and then there is a child pointer 1. So, this goes on, so child pointer 1 then there is a key 2, then there is a record pointer 2, then there is a child pointer 2 etcetera, etcetera it goes on like this till there is about key k let us say, then there is a record pointer k and then there is a child pointer k. So, this is the structure, this is the B tree internal node with k keys.

So, let us try to understand this, so there are these k keys, this is key 1, key 2 up to k key, corresponding to each key the record pointer. So, this is a pointer that stores the record of the corresponding key is stored next to it. So, there are this k such record pointer, key pointer and then there are k plus 1 child pointers and the child pointers this follows the same idea as the BST. So, this child pointer contains all the keys that is less than this key 1, this child pointer contains all the keys that is greater than key 1, but less than key 2 and so on, so forth it goes on all the way.

So, this is everything that is less than key 1 and this is everything that is greater than key k technically correct, this has to be less than equal to so on, so forth, but that is the whole idea of a B plus tree and it is organized in that entire manner and then there is a order theta that is mentioned as part of the B tree. So, every internal node must have between theta to 2 theta keys cannot have anything; other than it cannot have less than theta things, theta key that cannot have more than 2 theta keys. So, there is a term called the branching factor or the fan-out of a B plus tree is therefore, it must have between theta to 2 theta keys.

So, the branch out must be at least theta plus 1 and it is atmost 2 theta plus 1. So, the branching factor the fan-out is the number of children that one node can have. So, in this case of k things there are k plus 1 children, so that is a B tree. Now, let us move on to what is a B plus tree, the B plus tree is very similar, but with one important difference.

(Refer Slide Time: 05:49)

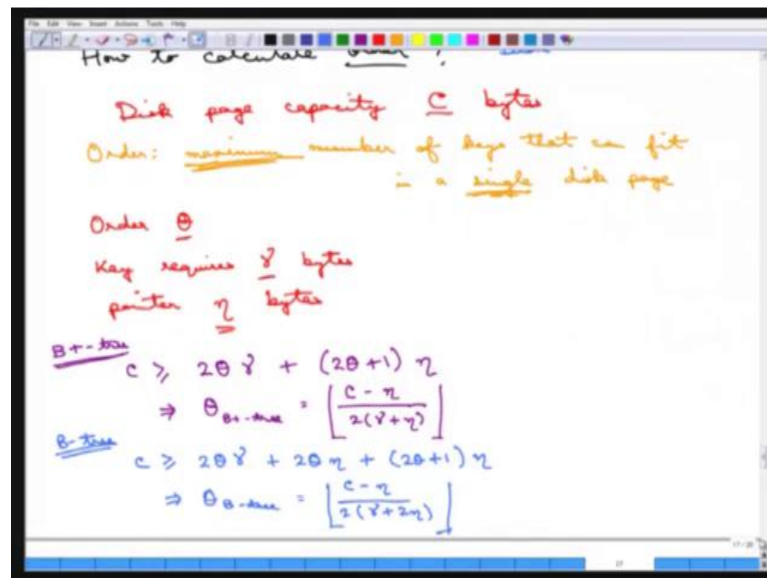


So, the B plus tree is a very similar structure, but there is one important difference with the B tree. So, if one draws the internal node structure of a B plus tree, one can understand the difference. So, this is a key 1 then corresponding to this there is a c p 0, then there is no record pointer. So, that is the very important difference, so then this all goes on, then there is key k and then there is a child pointer k, then there is no record pointer corresponding to a key in B plus tree. So, there is just a key and there is no record pointer. So, this is 1, 2 again up to k and then there are this again k plus 1 children and this again follows the same idea key 1 etcetera.

But, it does not have the corresponding record pointer, so this record pointer k is not maintained inside the internal node. So, what does this mean is that these keys cannot be actual search keys, because if it is an actual search key the record pointer must be present. So, even if it is an actual search key, this actual search keys must be stored only at the leaf level, this cannot be an actual search key. Because, no key is stored, no record pointer is stored, this is not a search key this is just what is called a key to index.

So, these are not real keys that are present in the database, this may not be real keys that is present in the database, these are just keys to index. So, that is a big difference between a B tree and a B plus tree. Now, again the, everything else remains the same as the B tree. So, it is balanced and everything is in the leaf and it is of an order, it can have an order theta so; that means, it can have between theta to 2 theta keys etcetera, etcetera of the same thing.

(Refer Slide Time: 07:49)



Now, how is the order of a B tree or a B plus tree, how to calculate the order of a B tree or a B plus tree. So, I mentioned that these are disk based design. So, where is the disk based design coming? How to calculate order is where the disk based design comes, this depends on the disk that is there. So, if you remember the OS cannot access everything from the disk, it can only access data in terms of a page or block, this is called disk page or a block.

So, suppose the disk page or a block is some 4 kilobytes of data, so it cannot access only 1 byte of data, it must access the entire 4 kilobyte of data from the disk to the main memory and then, process that 1 byte or how many whatever number of bytes that is required to do that. And if you also remember, the main point of the physical design that we showed is that is to reduce the number of this disk accesses, the number of random IOS or the number of sequential IOS is the main point.

So, it does not matter what is the time spent in the main memory, but the point is to optimize on the number of blocks that is read or written by the algorithm, so the number of blocks that is accessed. So, since a particular page must be accessed from the disk and since accessing the page meaning, accessing all 4 kilobytes of data, it makes sense to packing the disk page with as much data as possible. So, as much of means as much of 4 kilobytes of data whatever is the size there, such that once the page is accessed, once the page is brought into memory, the entire data that is stored in the page can be utilized and it does not require another access to bring in another data point.

So, what I mean to say is that, suppose the disk page capacity, so a disk page suppose the disk page has a disk page capacity is of C bytes. So, then it makes sense to put in as many search keys as possible within those C bytes, because this... So, one has to access this C bytes, one has to bring this all this C bytes anyway even if one search key is accessed. So, it makes sense to put in as many search keys as possible within this C bytes.

So, now, how to calculate the order? So, order is the maximum number of keys that can fit, maximum number of search keys that can fit in a disk page, the order is calculated using that can fit in a single disk page. So, that is the maximum that disk can fit in a single disk page. Then, how to compute it? Let us say the disk page capacity is C bytes, let us say the order is theta and we will calculate what theta is, this is theta and suppose key consumes key requires some gamma bytes to store a pointer that the pointer to where the data is requires some eta bytes.

So, for a B plus tree what will happen is that this C must be greater than or equal to 2 theta. So, if the order is theta then there will be 2 theta keys that can be stored, so 2 theta times gamma plus there are no record pointers for a B plus tree. So, this we are doing for a B plus tree, there is no record pointer. But, there are 2 theta plus 1 child pointers, so which is eta. So, this is the definitive equation for a B plus tree, so one if you calculate this comes out to be theta for a B plus tree, then comes out to be the following it is C minus eta by twice of gamma plus eta and of course, one has to take the floor function of it is the maximum number that can fit, this is how to calculate the order for a B plus tree.

For a B tree, the same thing can be done except there is something more. So, there is a 2 theta of gamma, this is the number of bytes that is required to store the 2 theta keys, then

there are 2 theta record pointers. So, one has to add this 2 theta times eta plus of course, the 2 theta plus 1 times eta for the child pointers, so this then comes out to be theta of B tree then comes out to be C minus eta, but this is 2 of gamma plus 2 eta.

So, these are the two important formula that we need to remember on how to compute the order of a B plus tree, how did we come to this conclusion to repeat is that it makes sense to read an entire disk space. So, it makes sense to pack in as much data as possible within a disk space, because that is the least that the OS will read anyway and the disk accesses are the costliest operations. So, it makes sense to reduce the number of disk access operations.

And once a disk, once a block or once a disk space is accessed it makes sense to pack in as much of that C bytes as possible, if theta is the maximum number of keys that can be done then this two equations give a way of how to find out this theta.

(Refer Slide Time: 13:46)

And we can take a simple example of how to do it, let us compute this example there is a page size, typical page size is of the order of 4 kilobytes, 8 kilobytes, etcetera, let us say it is 4 kilobytes. So, this is 4 kilobytes the page size, let us say the keys take 8 bytes and pointers suppose it takes 4 bytes, let us assume this is the situation. Now, how do we find out? The B plus tree can be used the formula that is there, so this is the capacity. So, if one computes this, this comes out to be 170 while the theta for B tree.

Now, I have one to highlight one important point here, you see that the order of a B plus tree is more than the order of a B tree. So, it is in this particular example it is 170 versus 127, so which means that for a particular node there can be much more data packs for a B plus tree than a B tree. So, which also means that a height of a B plus tree can be lesser than that of a height of a B tree, because assuming the same amount of data since more amount of data can be fit within a node, less amounts of nodes are required and the branching factor is also larger for this node, so which means that the height of the B plus tree can be less.

However, what may happen is that in a B tree the search key is located higher up than the level of the leaf. So, a particular search key can be located much faster when going all the way up to the leaf level, for a B plus tree there is no way, because the internal nodes do not contain the search key, every search key is stored only at the leaf level. So, to find the search key for a B plus tree, the search has to process all the way up to the leaf level.

In general what happens is that the number of search keys that the B plus tree found without going all the way up to the leaf level is very low. And therefore, what the database practitioners actually do is to build up a B plus tree and not the B tree and even though one will find that the B tree index has been built it is actually the B plus tree index that is being built for commercial databases and for other implementations of database systems.

So, it is the B plus tree that is the defector standard and that completes the B tree and the B plus tree thing and that also completes about the tree based indexing, there are little more left about the indexing and one very interesting kind of indexing that can be done is called a bit map indexing.

(Refer Slide Time: 16:53)

So, what may happen is that when the attribute domain consist of a small number of values. So, a bit map or a bit vector can be employed to store all such values. So, let us take an example to highlight what I am trying to say, suppose there is a simple table which is gender and grade and suppose this is male, C, female, A, female, C, male, D, male, A. Now, each of this gender as a very small attribute domain, because it contains only male and female. So, is a grade it contains only A, B, C, D and suppose that is the case then one can employ a bit vector to encode all these values.

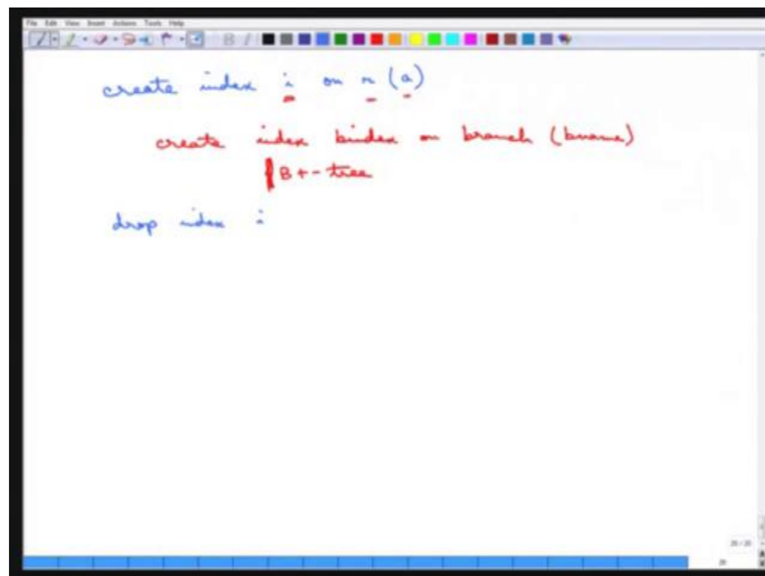
So, what one can do is to say that the male consists of the following manner, it is 1, 0, 0, 1, 1 why is it one, because the first one is a male, the second one is not a male, the third one is not a male, fourth one is a male, fifth one is a male. Now, note that one means it is a male, the 0 means it is not a male, it does not say what the value is and for that we require this female coding as well which is 0, 1, 1, 0, 0 and although this looks like a redundant thing, because it is complimentary, because it has got only two values, it does not it will not happen for A, because A will have this 0, 1, 0, 0, 1, B none of the values is b. So, it will have 0, 0, 0, 0, 0.

Now, note that for example, here it is not A, but it does not say what the value is and one has to go to C to find out that this is actually C and D is simply 0, 0, 0, 1, 0. So, how is this useful? So, suppose a particular query is find all males, find male students who got D. Now, this is one only need to do the take the bit map of m and do a bitwise and with

bitmap of D. So, that is easy to do and once that is being done, the answer comes out to be...

So, bitmap of m is this one and bitmap of D is this one and once that is being done, this is the one that is highlighted. So, it is the fourth student he who has got who is the male, who has got a D. So, bit map operations are generally much faster and more efficient, because OS packs them in nice byte sized order etcetera. So, that is why this is more can be faster, to complete the indexing scheme this is how things are being done in an SQL.

(Refer Slide Time: 20:06)



In an SQL one can say create index i on r of a so; that means, create the index which is named as i on the attribute a of the relation r. So, for example, one can say create index b index on branch b name, generally when only one attribute is being mentioned it is the B plus tree index that is being meant. So, that is the B plus tree index that is being meant and simply one can also say drop index just to delete the index that completes the part about indexing and we studied about hashing, static hashing and dynamic hashing and some tree based indexing.

Thank you.