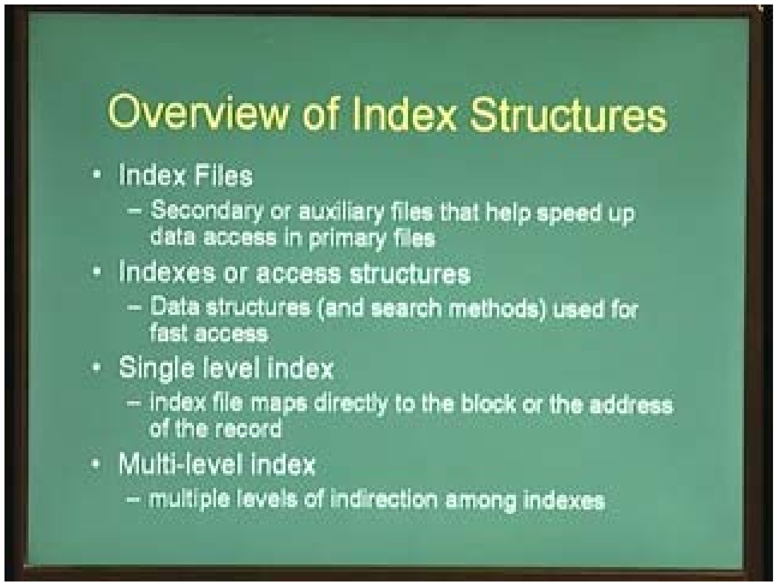
Database Management System
Dr. S. Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 12

Indexing Techniques Multi-Level and Dynamic Indexes

Hello and welcome. In the previous session we had looked into different varieties of index structures. Index structures are auxiliary files that are used in the database storage that are used to help in accessing data which are stored in the primary files. Index structures are extremely important especially when database sizes have been growing exponentially in the recent past and the value of index structure is also more important because the main problem in database today is not the storage of the large amounts of data but retrieval of them. Rather in searching data elements based on some certain criteria, key values and so on. We saw different levels of, different kinds of index structures namely primary indexes, clustering indexes, secondary indexes on key attributes and secondary indexes on non-key attributes and so on. Let us briefly summarize them today in this session before we move on to more complex index structures.

(Refer Slide Time: 02:28)



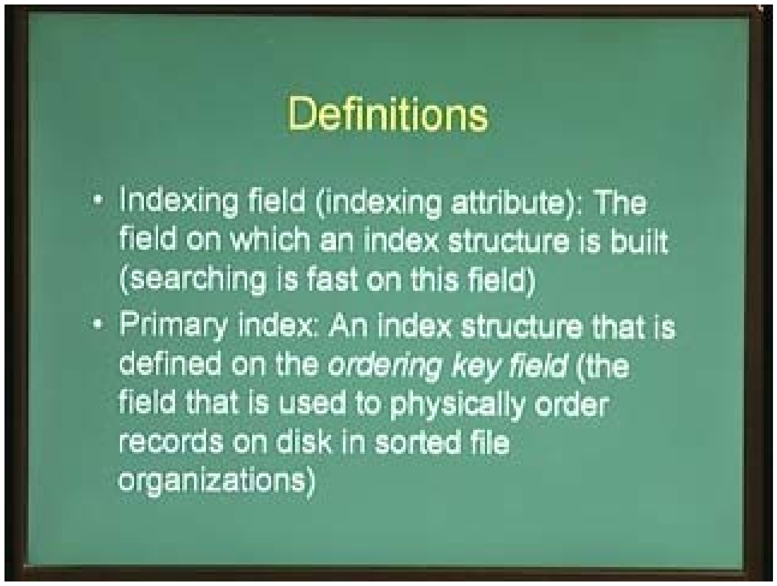
Overview of Index Structures

- Index Files
  - Secondary or auxiliary files that help speed up data access in primary files
- Indexes or access structures
  - Data structures (and search methods) used for fast access
- Single level index
  - index file maps directly to the block or the address of the record
- Multi-level index
  - multiple levels of indirection among indexes

Some key definitions in index structures are shown here. An index file is a secondary or auxiliary file that is used to help speed up data access that is contained in the primary files or the data file. An index or an access structure is the data structure that is used in these secondary files which help in this retrieval process and of course the data structure is associated with its corresponding search methods and algorithms using which we can access these data elements as quickly as possible.

We said that there are two kinds of index structure primarily, namely this single level index and the multi-level index structures. However until now we have just covered this single level index structures. A single level index structure is a single auxiliary file that directly maps to addresses in the primary file. And these addresses could be either block addresses which stores physical blocks on disk or any other storage medium or they could be recorded results where the address of or a record is directly stored or a record can be directly accessed within a block. That is the block address is augmented with the offset value which gives us the record address.

(Refer Slide Time: 03:57)



Definitions

- Indexing field (indexing attribute): The field on which an index structure is built (searching is fast on this field)
- Primary index: An index structure that is defined on the *ordering key field* (the field that is used to physically order records on disk in sorted file organizations)

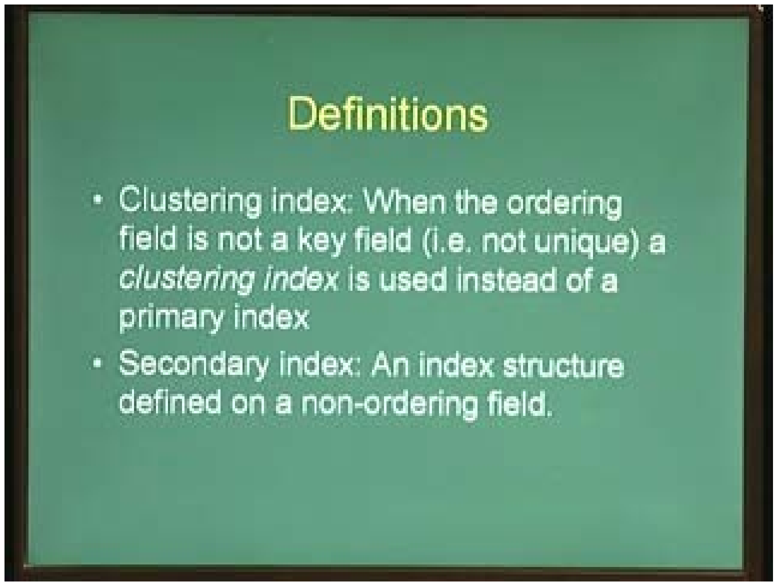
Some more definitions which are again important for looking into multi-level indexes which we are going to be exploring in this session. The indexing field is the field or the attribute on which the index is maintained and usually the field could be either an ordering field or a non-ordering field. It could either be a key field or an non-key field and the corresponding index structure for each of these fields changes depending on what kind of fields or what is the characteristics of the field that we are indexing.

A primary index is an index that is defined on the ordering key field of the data element that is the field should not only be an ordering field, it should also be a key field of the data element. What are the properties of an ordering field and a key field? Key field has a property that it is unique, that it has a uniqueness constraint or that no two fields in the database, no two key fields in the database have the same value. Similarly if the field is an ordering field we can have an assurance that the primary data file is physically sorted based on this field.

Therefore whenever we have a key of a given value i, we know that for all key values greater than i we have to search forward that is we have to search in the forward direction of the file. We don’t have to search the reverse of the file or all the key values until now and so. Therefore these properties the property that the file is ordered based on the ordering field and the field is a key field helps us in building a primary index, there is a sparse index which can help access data in the primary file as efficiently as possible. We also look into clustering index which is the index structure that is used when the field that is to be indexed is an ordering field but not a key field.

If the clustering field is not a key field then it is no longer constrained by the uniqueness constraints. That is there is no longer requirement that each of these, each element in this key field has to be unique. This poses a particular problem in the sense that a given key value may correspond to more than one addresses. The last kind of index that we saw was the secondary index. A secondary index is an index that is defined over some non-ordering field.

(Refer Slide Time: 6:45)

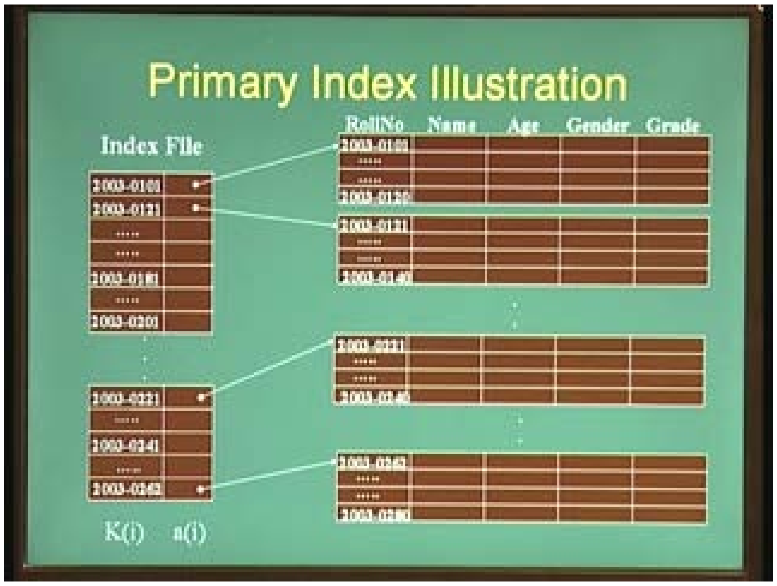


Definitions

- Clustering index: When the ordering field is not a key field (i.e. not unique) a *clustering index* is used instead of a primary index
- Secondary index: An index structure defined on a non-ordering field.

That is there is no assurance that the primary data file is ordered based on this field. If the primary data file is not ordered based on this field then it is not possible for us to store a sparse index. This is because we don’t know where to search the next record from. Therefore the index has to be dense index structure. However there is still, there is a further dimension to the secondary index data structure that is, is the key field or is the indexing field on which the secondary index is based upon is it a key field or a non-key field. If it is a key field then we have a particular kind of index structure and if it is a non key field then the index structure changes.

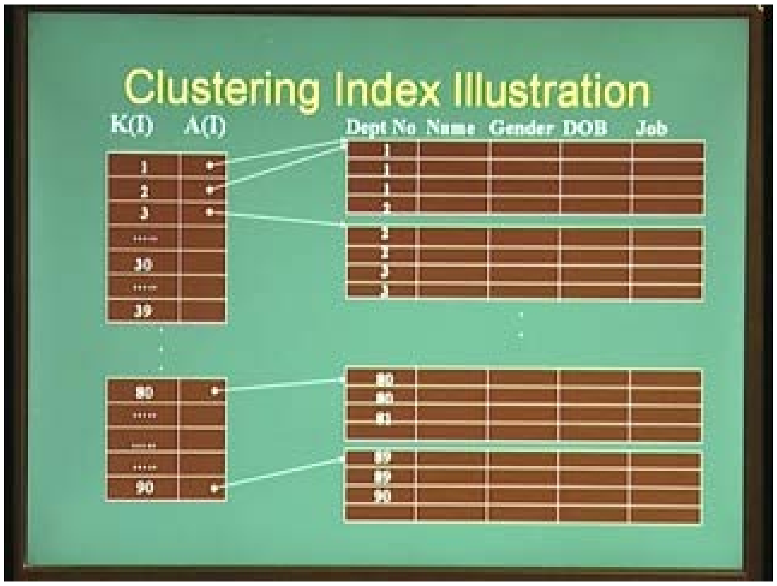
(Refer Slide Time: 07:27)



Let us briefly look at some illustrations of the three kinds of index structures that we are covered so far, so that it helps us in understanding the more complex index structures that we are going to cover in this session. The primary index structure is shown in this slide here, this slide shown an index file pointing to a different blocks in the data file. The data file comprises of different blocks that is records that are organized into different blocks and the blocks or the records are sorted within the blocks that is the indexing attribute is not only a key attribute that is it is not only unique, it is also the set of data records are also sorted based on this indexing field.

When this is the case, it is enough for us or it is sufficient if we are able to store or if we are able to index just the first attribute or the key value within a given block. This is called the anchoring record if you remember. So we just store the key value of the anchoring record in the index file and maintain a sparse index. The number of entries in this index file is equal to the number of blocks, the physical blocks that make up the primary file or the data file. And this index file is a sparse index because it does not store all attributes or all values of the key and this index file can afford to use fixed length records because the value of the record is, value of the key is known and the value of the address, block address is also known and we don't need anything else therefore the primary index file can afford to use fixed length records.

(Refer Slide Time: 09:15)

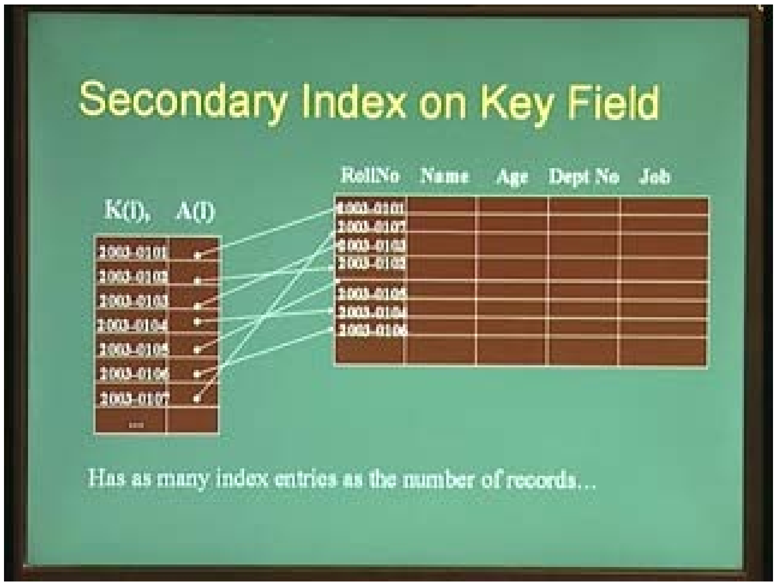


This slide shows an illustration of the clustering indexing structure. In a clustering index, the file or the primary data file is ordered based on the clustering field. However this clustering field is a non-key field. If it is a non key field then there is no guarantee or there is no requirement for the field to be unique. So, in this slide there are some records that are shown, ordered on the field called department number. And there are repetitions in the department number that is there are three number of ones for a given department number, three number of two’s and two number of three’s and so on.

However since the primary file or the data file is ordered based on this field, it is sufficient for us to know where does the first or where is a first occurrence of a given value and that is what we store in the index file. That is the index file stores a unique values that the ordering field takes up that is shown in the left hand side of the slide that is values like 1 2 3 and so on and a pointer that points to the first occurrence that is to the block that contains the first occurrence of this particular value because the primary file is sorted, this is sufficient for us.

However there is a problem with insertion of records which needs, which may need a clustering indexes to be altered and which can be rectified by assigning separate blocks for each distinct value of the ordering field.

(Refer Slide Time: 11:03)


Secondary Index on Key Field

Has as many index entries as the number of records…

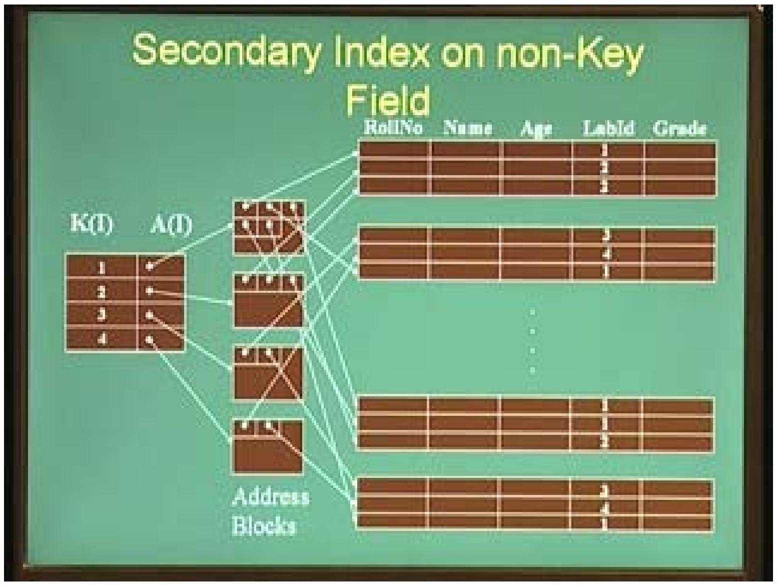
We also know that secondary index on key field attributes, if it is a key field note that secondary index or index structures that are maintained on non ordering fields that is a fields on which, fields which do not contribute to the physical ordering of data records in the primary data file because they do not contribute to the physical order of the records in the primary file, this has to be a dense index because we have to for each record we need to know where this index or where this is actually stored. Therefore all values of the index attribute has to be reflected in the indexed file.

In the slide here there are two files shown. The index file shown in the left side contains each roll number which is the indexing attribute and which is also the key attribute. That is the roll number is unique because the roll number is unique that is because the attribute is a key attribute, we don't have to worry about duplicates, we don't have to worry about repetitions and because repetition is not a problem we can afford to use fixed length record sizes for the indexed records. That is we know the length of the key attribute and we know the length of a record address and therefore we can afford to use fixed length records in the indexing file.

However the indexing file is dense and it contains as many records as there are records in the data file itself. If secondary index is maintained on a non key attribute then we no longer have the luxury of the uniqueness constraint on the attribute. That means this attribute not only, does not contribute to the physical ordering of records in the primary file, it also is not constraint by the uniqueness constraint that is there may be repetitions, there may be several different records having the same key value. If this is the case, then a given key value K of i may correspond to multiple addresses. In clustering index this was not a problem because the indexing attribute was an ordering attribute. That is physically the data records were ordered based on this attribute.

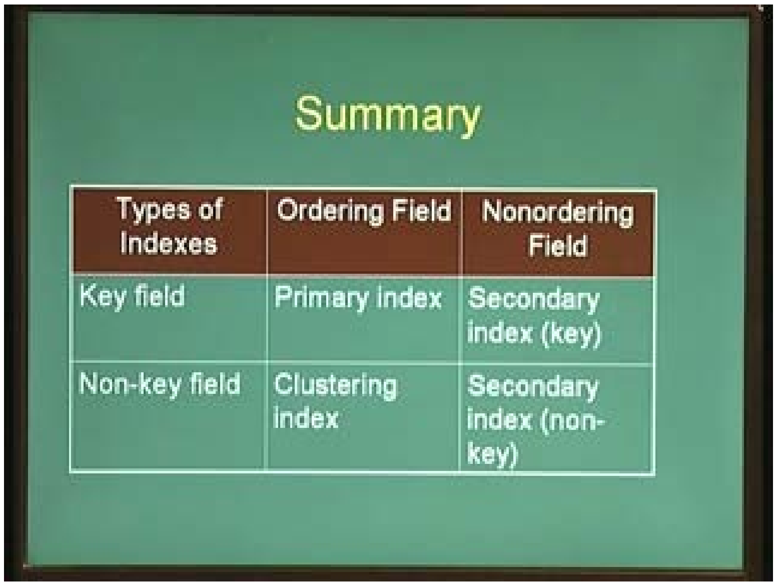
Therefore it was sufficient for us to know where is the first occurrence of this record. We do not have such a luxury in secondary indexes because this indexing field is not an ordering attribute. In such a case we use extra levels of indirection or extra level levels of redirection in order to reach the data record. In the slide here there are three different kinds of files that are shown.

(Refer Slide Time: 14:02)



The left most file is the secondary index file which shows K of i and A of i attributes that is key values each and every distinct key value and a block address for each key value. Disk block address is not the block address of the data file but in fact a block of addresses, a block address containing a block of addresses, many different addresses. So this block contains several addresses, one each for each record having this particular value and they are stored block wise and block overflows are handled by chaining so that each different key value occurs in a separate block by itself.

(Refer Slide Time: 14:53)

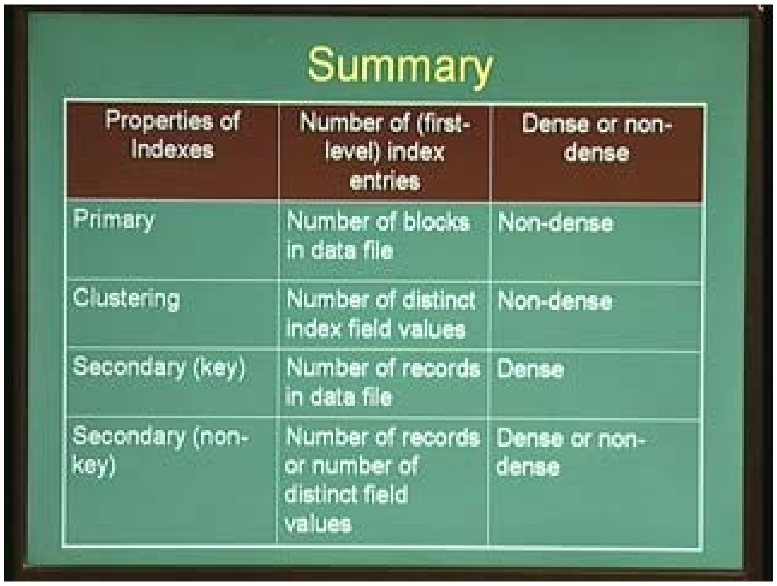


Summary

Types of Indexes	Ordering Field	Nonordering Field
Key field	Primary index	Secondary index (key)
Non-key field	Clustering index	Secondary index (non-key)

So let us briefly summarize the different characteristics of a single level indexes and see and motivate a need for multilevel indexes. If the field or the indexing field is a key field and an ordering field, we can use a primary index as shown in the slide here. If the indexing field is a key field and not an ordering field, we can use a secondary index on keys that is a dense index with fixed length records. If the indexing field is a non key field but it is an ordering field then we use the clustering index where we can store just the address of the first occurrence of every unique data value. If on the other hand, if the indexing attribute is neither a key field nor an ordering field then we have to resort to secondary indexes of non key varieties that is we have to use an extra level of indirection.

(Refer Slide Time: 16:06)

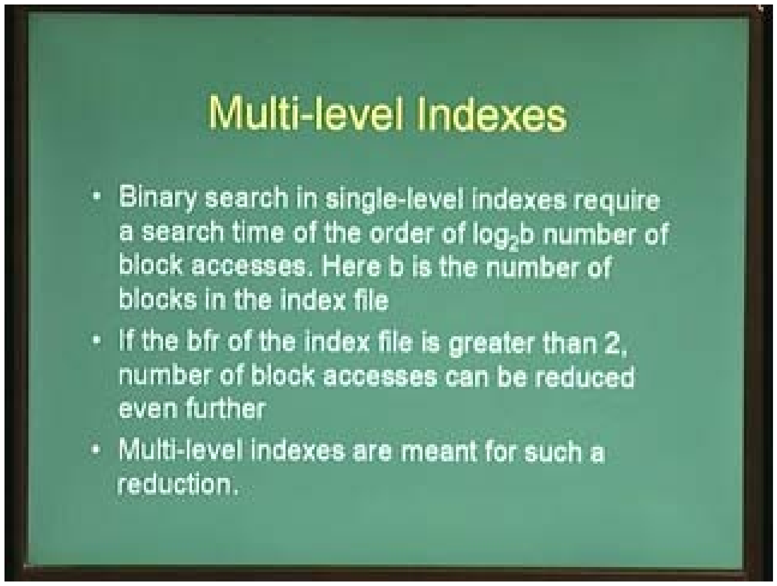


Summary

Properties of Indexes	Number of (first-level) index entries	Dense or non-dense
Primary	Number of blocks in data file	Non-dense
Clustering	Number of distinct index field values	Non-dense
Secondary (key)	Number of records in data file	Dense
Secondary (non-key)	Number of records or number of distinct field values	Dense or non-dense

There also other properties of the index structures that is primary index is the sparse index where the number of records in the index file is equal to the number of blocks in the primary file. Similarly a clustering index is also a sparse index where the number of records is equal to the number of distinct values that are present for the attribute. A secondary key index is a dense index which contains as many number of records as there are records in the database itself. While the secondary non-key index is either a dense or a sparse index depending on weather there are repetitions in the non key attribute and the number of records in the key attribute is simply the number of distinct values that are present in the indexing attribute in a data file.

(Refer Slide Time: 16:46)



Multi-level Indexes

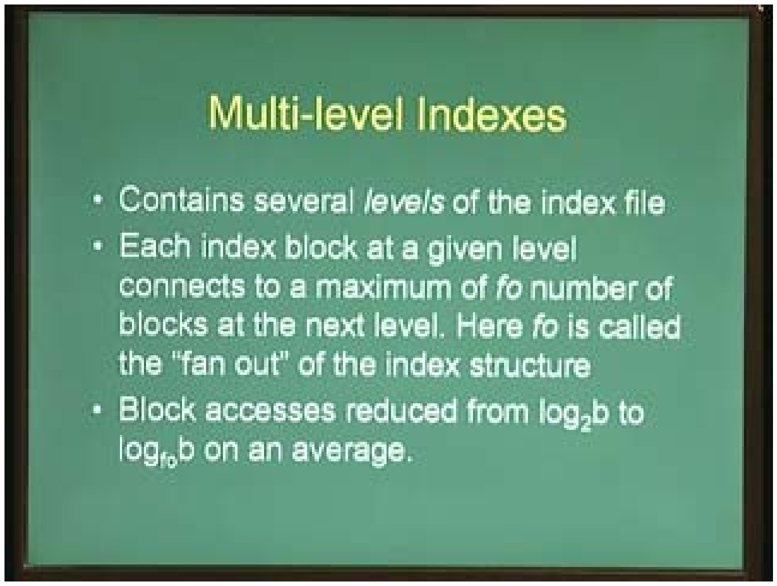
- Binary search in single-level indexes require a search time of the order of $\log_2 b$ number of block accesses. Here b is the number of blocks in the index file
- If the bfr of the index file is greater than 2, number of block accesses can be reduced even further
- Multi-level indexes are meant for such a reduction.

One of the main advantages of index structures like say primary index or dense secondary indexes and so on. Is that index files are ordered files that is they are ordered on their key values because they are ordered on their key values; it is possible to search them based on binary search. A binary search is a search technique which reduces the search space by half in each iteration. Therefore in the average case or in a ideal case, one can reach the particular key or the address in log n number of times, log n to base 2 where n is the number of records in the file.

On the other hand a linear search requires times of the order of n or n by 2 rather, so n different memory accesses or n different record accesses. However we can note that there are three different entities that we are concerned with during physical storage. These are the file itself which is a sequence of logical records, a record which is a logical equivalent of tuple in a relational schema and the block which is purely of physical nature that is which is meant for efficient data transfer between the storage media and the computer and whose size is determined by physical characteristics.

Now between block and records, we have defined a notion of the term blocking factor where blocking factor is the number of records per block or how many number of records can we store in the block. Now if the blocking factor of an index file that is of blocks storing an index file is greater than two that means if a block, if a disk block can store more than two addresses or more than two index records then we can actually come out with a even better method of searching where the method of searching is of the, reduces by the order of blocking factor rather than by the order of 2 which is constant in binary searches. These are explored in multi level indexes.

(Refer Slide Time: 19:20)

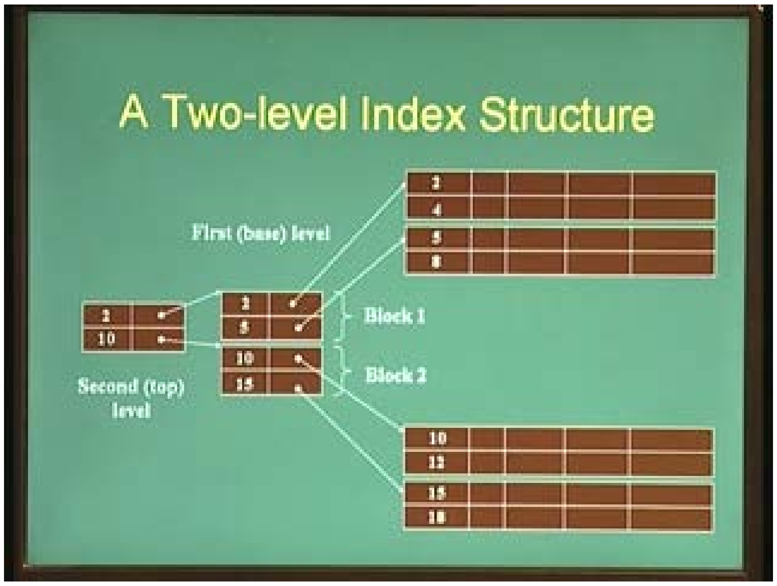


Multi-level Indexes

- Contains several *levels* of the index file
- Each index block at a given level connects to a maximum of *fo* number of blocks at the next level. Here *fo* is called the "fan out" of the index structure
- Block accesses reduced from $\log_2 b$ to $\log_{fo} b$ on an average.

A multi level index is an index file which contains several different levels as the name suggests and each index block at a given level has a factor called the fan out. A fan out is typically derived from a blocking factor that which depends on the number of records that one can store in a block. So a fan out is the number of different records that or the number of different entries that a given index entry can point to and in a good implementation, block accesses can be reduced from log n or log b to base 2 where b is the number of blocks in the index file, so it can be reduced from log b to base 2 to log b to base fan out. So this is useful if fan out is obviously greater than 2. If fan out is greater than 2, we can reduce the number of block access by a tremendous factor.

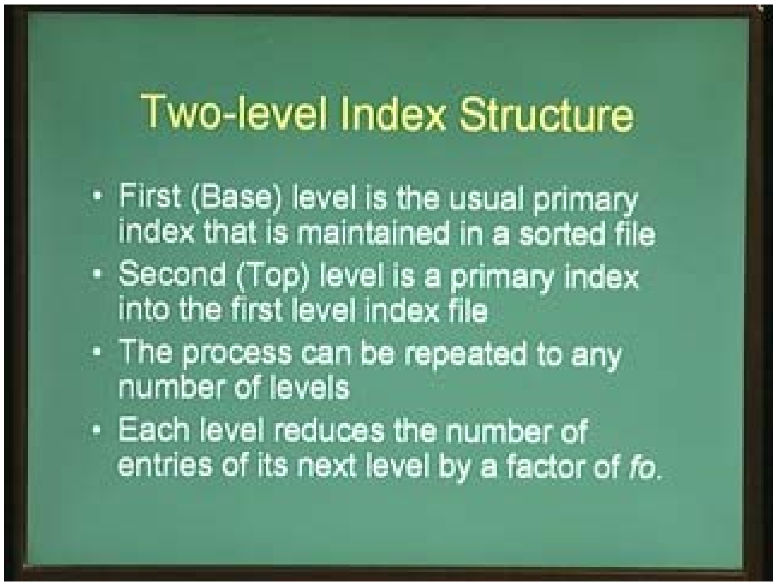
(Refer Slide Time: 20:31)



The figure in this slide shows a two level index structure where index structures are categorized into first level and second level. The first level is called the base level and the second level is called the top level and for the sake of clarity, I have also shown how index records are divided into blocks at the base level. It is divided into two, here the blocking factor is 2 but usually the blocking factor is more than 2. If you can notice the slide carefully, the way the slide or the way the index stores information is that there are different levels in which information is stored.

At the top level there are just 2 entries k of i entries 2 and 10. This entry says that everything between 2 and the next number can be found in this index file. That is every key value between 2 that is greater than or equal to 2 and less than 10 can be found in this index file. Similarly every key value that is greater than 10 and there is nothing below, therefore which is just greater than 10 can be found in the index file that is pointed to by this pointer. The other, the second level index file is also a replica of the top level index file in the sense that everything greater than or equal to the key value that is specified here and less than the key value of the next record is pointed to by the present address. That is this is an indexing scheme that indexes an ordering field and also a key field. Therefore all records starting from 2 to less than 5 can be found here and all records starting at 5 and less than 10 can be found here and so on.

(Refer Slide Time: 22:39)



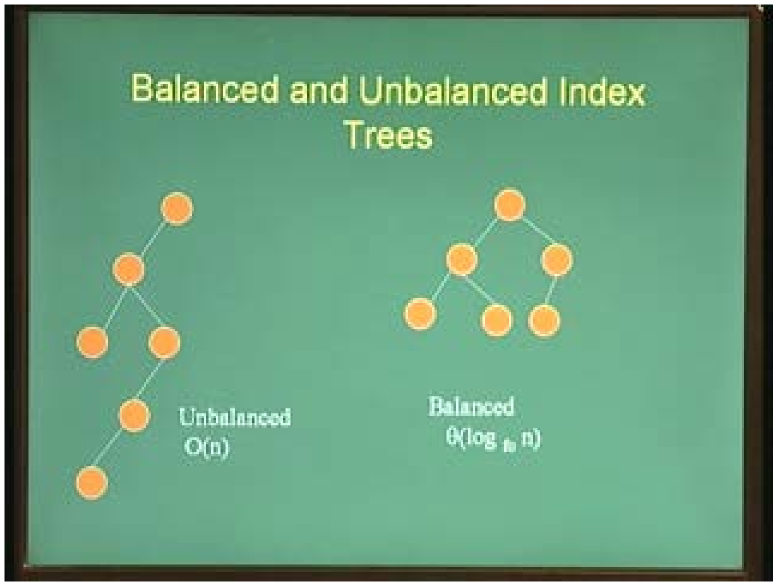
Two-level Index Structure

- First (Base) level is the usual primary index that is maintained in a sorted file
- Second (Top) level is a primary index into the first level index file
- The process can be repeated to any number of levels
- Each level reduces the number of entries of its next level by a factor of *fo*.

The first level or the base level is a usual primary index that is maintained on a sorted file. The second level is a actually a primary index on the primary index because the index file itself is a sorted file and it is sorted based on the key attribute, we can store another primary index on this index file. And we can continue this process to any number of levels depending on the size of our database. So we could have a third level that stores an index or that stores a primary index on the second level and so on and at each level, the number of entries in the next level is determined by the fan out or the blocking factor.

Here the fan out was 2 therefore what we ended up seeing was a binary tree structure. However in general the fan out is usually much more, a block could contain many number of records, many number of index entries much more than 2. Therefore at each level from level 2 to level 3 for example fan out number of records that is a blocking factor number of records can be indexed using just one index structure. Therefore the number of entries starts reducing exponentially by a factor of fan out. So this again depicts the same thing that is at each level, the number of index entries is getting reduced by half.

(Refer Slide Time: 24:19)

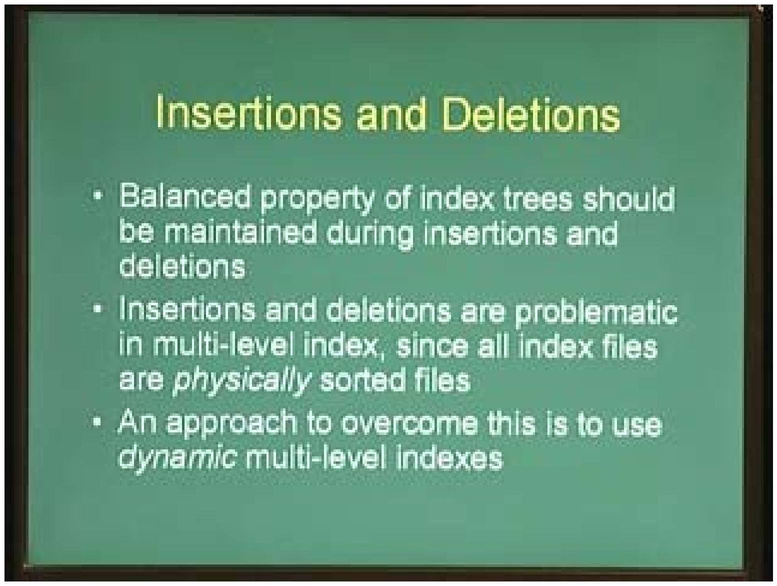


In order for a multi level index to be efficient enough or to be more efficient than primary index, there is an important consideration of the multi-level index structure. If you are familiar with data structures, you might have come across a data structure called tree which is a data structure used for representing hierarchies. A multi level index structure is useful only when the tree structure that is specified by the multi level index structure is balanced. What is meant by a balance tree? It is well possible for us to have a multi-level index structure of the kind that is shown in the left hand side of this slide shown in this figure.

In the left hand side, as you can see there are several levels to the index structure and in the worst case one has to make 4 different 1 2 3 and 4 different traversal's of different index files in order to find a given record. On the other hand the right hand side of the figure has the same number of nodes or here each node or each circle here represents an index file and the right side of the figure has the same number of index files, however a smaller number of levels. And the load or the number of index files is more or less evenly balanced across the entire tree.

In such a case in the average, the average behavior of the such a tree or even the worst case behavior of a balance tree is only log n to base fan out while the worst case behavior of an unbalanced tree is of the order of n different block accesses and which is no different from performing a linear search over these different index files. Therefore an order for multi level indexes to be useful, they have to they have to form a balanced tree.

(Refer Slide Time: 26:42)

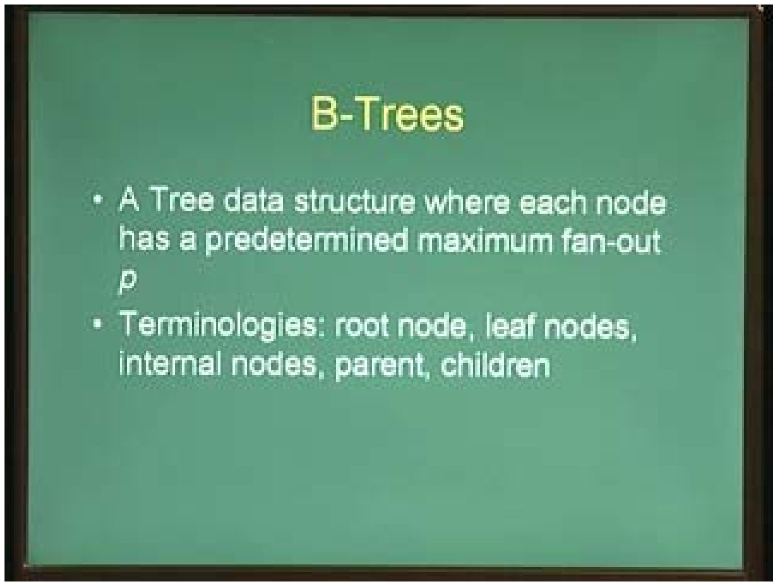


Insertions and Deletions

- Balanced property of index trees should be maintained during insertions and deletions
- Insertions and deletions are problematic in multi-level index, since all index files are *physically* sorted files
- An approach to overcome this is to use *dynamic* multi-level indexes

So whenever insertions and deletions happen in a data file containing a multi level index, the balanced property of the index trees should be maintained. And this is especially problematic in multi level indexes because all index files are physically sorted files and we need to make a number of different adjustments at number of different levels, if you are just storing several different primary index structures in order to maintain the balance tree property of the index structure. An approach to overcome this is what is called as dynamic multi-level indexes. That is an index structure changes itself dynamically by as a little a number of operations as possible so that the balanced property of the index tree is maintained.

(Refer Slide Time: 27:41)



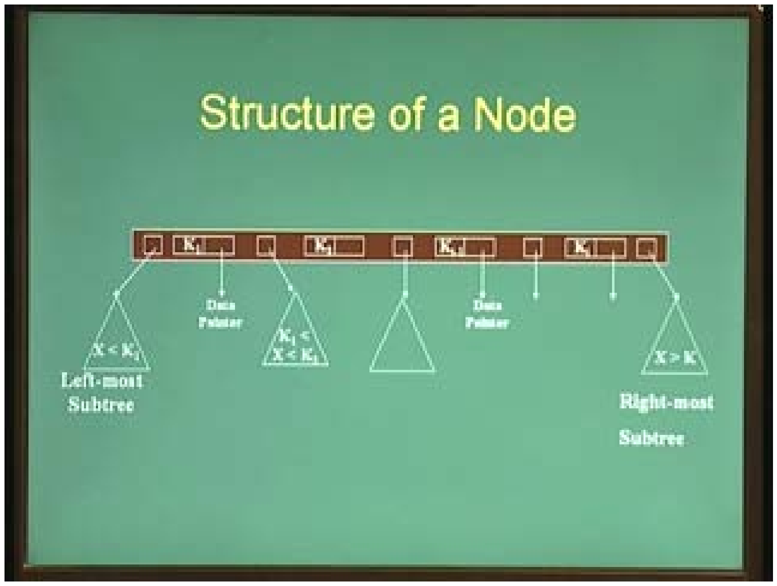
B-Trees

- A Tree data structure where each node has a predetermined maximum fan-out *p*
- Terminologies: root node, leaf nodes, internal nodes, parent, children

The most commonly used dynamic index structures are what are called b-trees and b plus trees. Let us have a look at these two index structures in a little more detail. Here a b tree is an index tree which is of course as the name suggest a tree data structure where each node has a pre determined maximum fan out given by p which is of course, when we are implementing it which would be related to bfr that is the blocking factor. There are several terminologies we use when we are talking about b tree. A given block that is allocated to a b tree is called a node in the b tree. As we will see later, a block corresponds to tree node in the logical tree structure that the b tree forms.

A special kind of node called the root node forms the access to the b tree that is it is the top most node in the tree and each node has a maximum of p children that is the fan out number of children and of course each node has a maximum of one parent. I am saying the maximum of one parent because the root node will not have any parent node. So it is either 0 or 1 parent depending on whether it is a root node or a non root node. And there are what are called as leaf nodes that are the lowest level nodes that is nodes which do not have any children. And any node in the tree that is not a root node, that is neither a root node nor a leaf node is called an internal node. Of course we have already defined the notion of parent and children that is when a node points to some other node then it is said to be a parent of the other node and the other node is said to be the child of the node which is pointing to it.

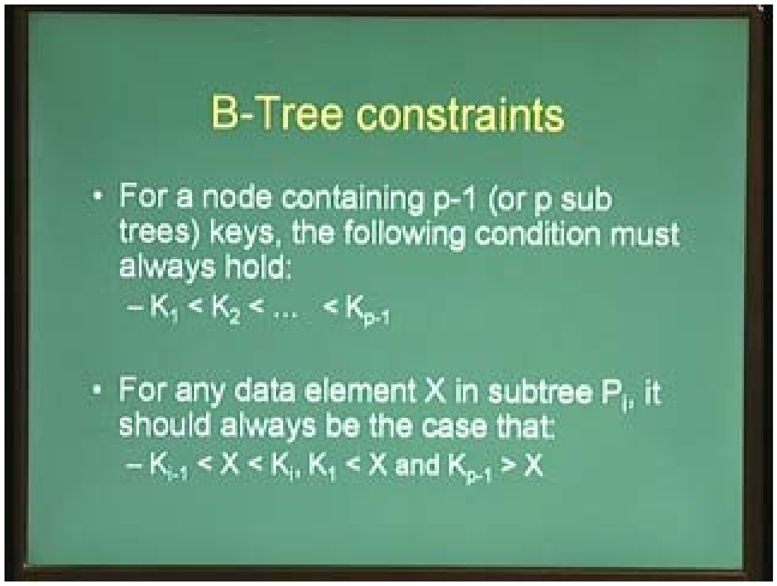
(Refer Slide Time: 29:41)



This slide shows a typical structure of a b tree node. This node is nothing but a block when it is implemented on disk. As you can see here there are several aspects or several fields to this node in a b tree. One can easily notice that there are several pointers in fact there are precisely maximum of p pointers in the b tree. Each pointer which points to a triangle represents a sub tree that can be pointed to by this node. Of course a sub tree can be null if this is a leaf node or if a pointer does not exist. Sub trees are filed in such a way that there are filled leftmost that is you cannot have a left most sub tree as null but some internal sub trees to be filled up and and left most sub tree being null. And in addition to the sub tree pointers or pointers to other nodes, there are several different key and data pointers. That is there are several blocks here that contain a key value under pointer to the record containing this key value.

Similarly there is another key value and and pointer to that record and so on. So there are several key values that are present and pointers to where the key values are present in the primary file in addition to maximum of p pointers to other nodes in the tree. What are the properties of these, what are the properties of such nodes or what are the constraints that a b tree has to adjure to?

(Refer Slide Time: 31:30)



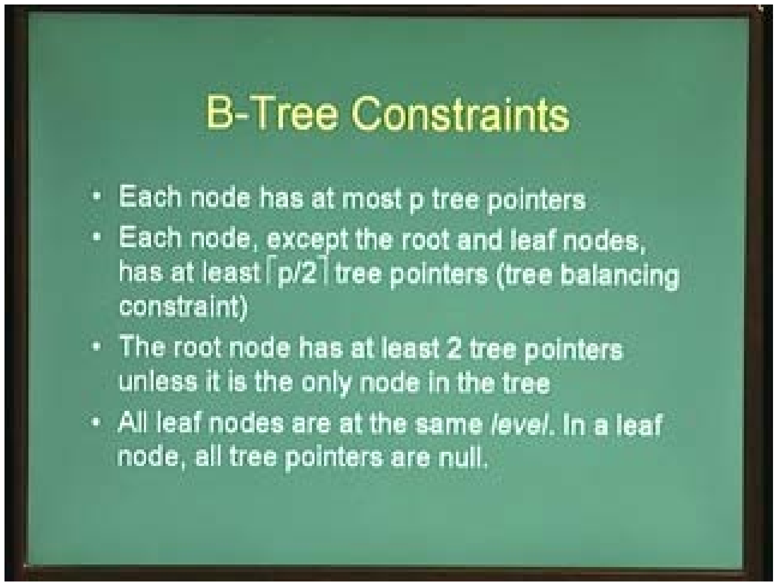
B-Tree constraints

- For a node containing p-1 (or p sub trees) keys, the following condition must always hold:
  - $K_1 < K_2 < \ldots < K_{p-1}$
- For any data element X in subtree P_i, it should always be the case that:
  - $K_{i-1} < X < K_i$, $K_1 < X$ and $K_{p-1} > X$

For a node containing p minus 1 keys, note that if a node can point to a maximum of p sub trees, it can contain a maximum of p minus 1 keys because as shown in the previous slide, their keys embed or this pointers embed the keys. That is there is one pointer on the left most side of the left most key and one at the right most side of the right most key and one pointer between every two keys in the node. The keys are always stored in a sorted sequence that is if there are p minus 1 keys then k_1 is less than k_2 is less than etc until K of p minus 1.

For any data element in a sub tree that is pointed to by one of the sub tree pointers, let us say in some sub tree P_i for any data element in that sub tree p_i that data element should be less than the data element of the left most key that is K of i minus 1 or rather it should be greater than the left most key, the right most left key that is K of i minus 1 and should be less than the leftmost right key that is K of i.

(Refer Slide Time: 33:02)



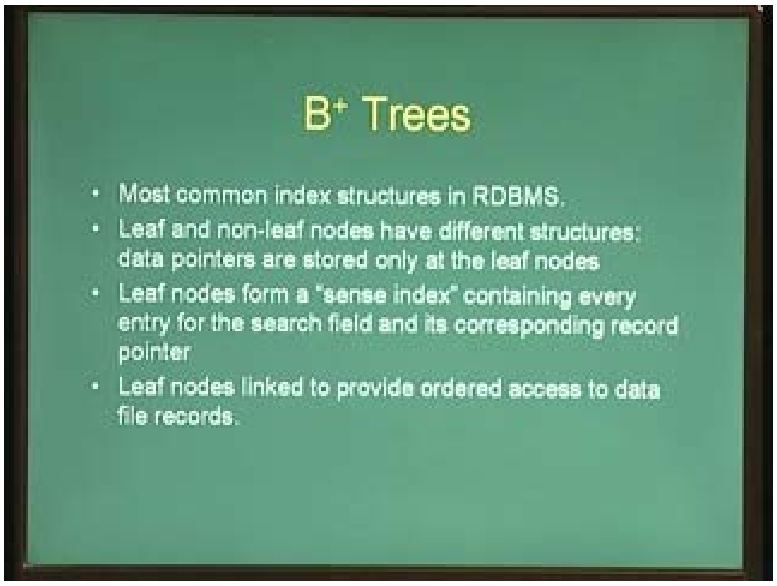
B-Tree Constraints

- Each node has at most p tree pointers
- Each node, except the root and leaf nodes, has at least $\lceil p/2 \rceil$ tree pointers (tree balancing constraint)
- The root node has at least 2 tree pointers unless it is the only node in the tree
- All leaf nodes are at the same *level*. In a leaf node, all tree pointers are null.

So more constraints of b tree nodes, each node can have at most p tree pointers of course and each node except the root node and the leaf nodes should have at least sealing of p by 2 tree pointers. What is sealing of p by 2? Divide p by 2 and take the upper integer value of this division. So they must have at least more than half of their pointers to be filled up as part of the tree building procedure. The root node should have at least two tree pointers, unless it is the only node in the tree that is unless root node is also a leaf node, it should have at least two children as part of the tree.

And all leaf nodes are the same level. What is the level of a node in a tree? The level is simply the distance in terms of the number of hops from the root node. All leaf nodes are maintained at the same level in the tree. So this is the constraint that has to be maintained and suitable algorithms have to be or suitable algorithms are created so that these constraints are maintained.

(Refer Slide Time: 34:22)



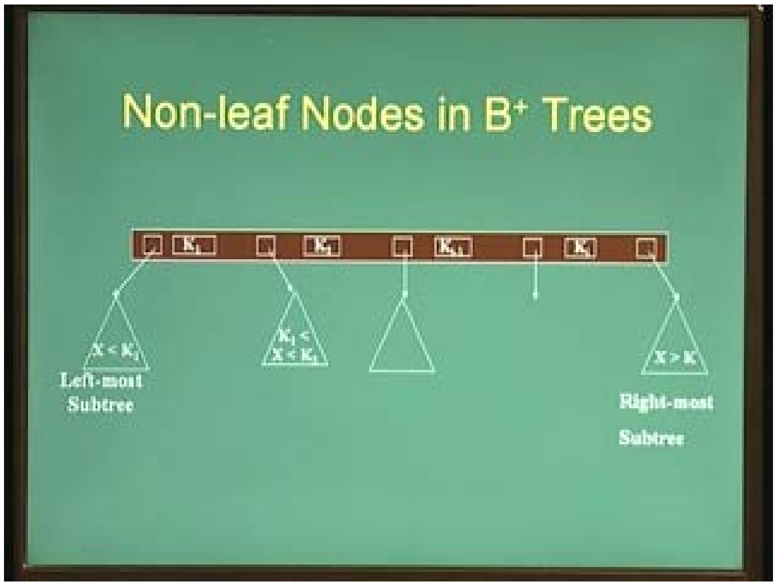
B⁺ Trees

- Most common index structures in RDBMS.
- Leaf and non-leaf nodes have different structures: data pointers are stored only at the leaf nodes
- Leaf nodes form a "sense index" containing every entry for the search field and its corresponding record pointer
- Leaf nodes linked to provide ordered access to data file records.

We shall be looking at insertion and deletion algorithm for b trees, after we have a look at the b plus trees. In fact the insertion and deletion algorithms for both of these tree structures are the same except that b plus tree has greater expressiveness or b plus trees allows for different kinds of accessing, different varieties of accessing the data elements in addition to B trees. Therefore let us have a look at some definitions of B plus trees and their constraints before we look at insertion algorithms. B plus trees is a most common index structure that is found in many of the commercial RDBMS.

It is very similar to a b tree except that the leaf and non leaf nodes have different structures. In a b tree, there is no difference between a leaf node and a non leaf node both of them have a same structure. That is they have a set of address pointers and the set of key values and data pointers. The leaf nodes form a separate kind of index containing each different key value in a sorted form and pointers to the corresponding data elements. That is leaf nodes are linked together so has to provide ordered access to the data file records.

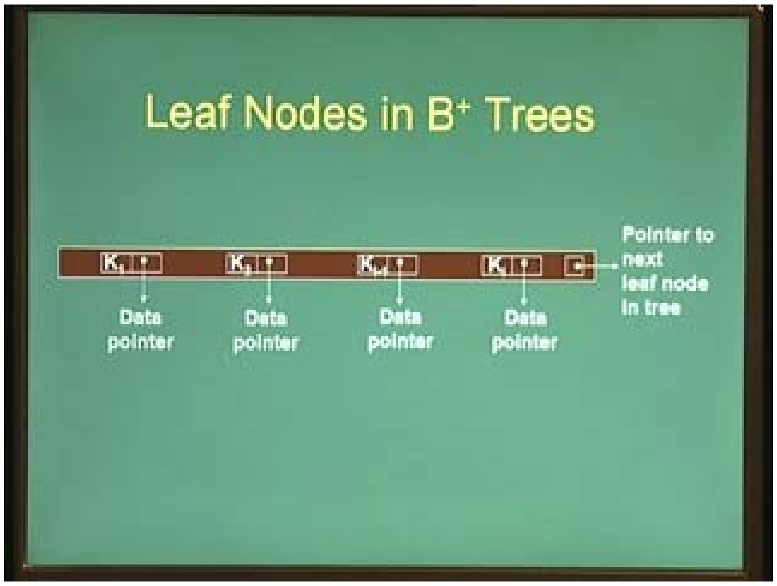
(Refer Slide Time: 35:48)



Non-leaf Nodes in B⁺ Trees

A non leaf node of a b plus tree is depicted in the following figure and as you can see it is quite similar to a non leaf node in a or it is quite similar to a node of a b tree. That is it contains a two kinds of entities that is sub tree pointers or block pointers and key values but the only difference is that there are no data values here. That is there are just key values k_1 k_2 etc, there are no data pointers as part of this node. And of course the same set of constraints hold that is for any x between k_1 and k_2 the value of all the keys in that sub tree x should be greater than k_1 that is the right most left key and less than k_2 which is the left most right key.

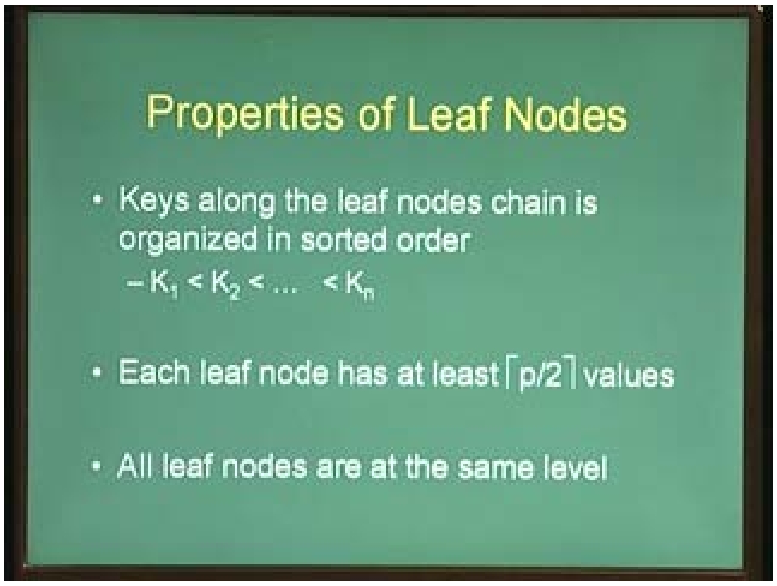
(Refer Slide Time: 36:41)



Leaf Nodes in B⁺ Trees

A leaf node in a b plus tree is shown in the following figure where there are no sub tree pointers because there are no children for the leaf nodes and there are only a set of keys and data pointers. That is there is key one and data pointer to the record containing key one, there is key two and the data pointer pointing to record containing key two and so on. And at the end of this block, there is a pointer pointing to next logical leaf node or the left logical block in this sequence. Therefore starting from the left most leaf node, we can access the entire database in a sorted form just by following the leaf nodes and the links to the next leaf node.

(Refer Slide Time: 37:37)

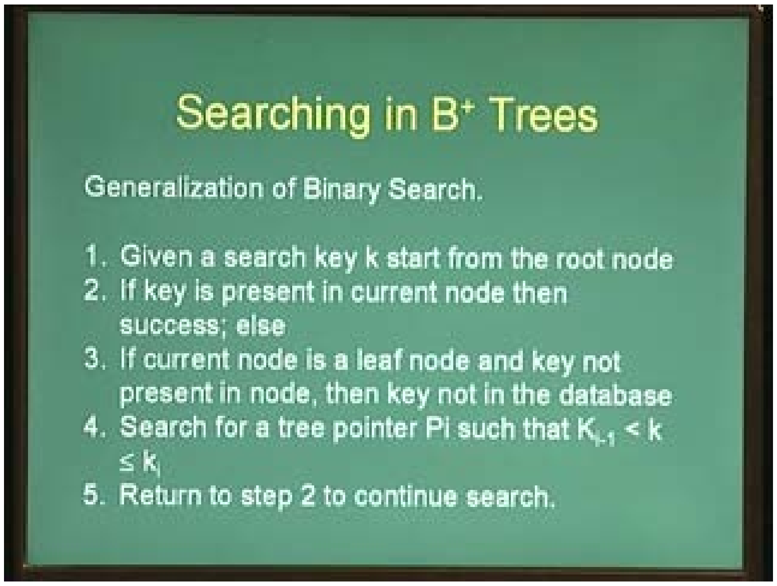


Properties of Leaf Nodes

- Keys along the leaf nodes chain is organized in sorted order
  - $K_1 < K_2 < \ldots < K_n$
- Each leaf node has at least $\lceil p/2 \rceil$ values
- All leaf nodes are at the same level

What are the properties of leaf nodes? Keys in a leaf node have to be ordered just like the property that we saw in b trees where keys have to be ordered within a leaf node. We should be able to access each leaf node in key sequence that is k_1 is less than k_2 less than k_n if there are n different keys in a leaf node. And just like the nodes in a b tree each leaf node should have at least half of its keys filled up. That is a sealing of p by 2 number of keys should be filled up and all leaf nodes should be at the same level as far as the overall b plus tree is concerned.

(Refer Slide Time: 38:23)



Searching in B⁺ Trees

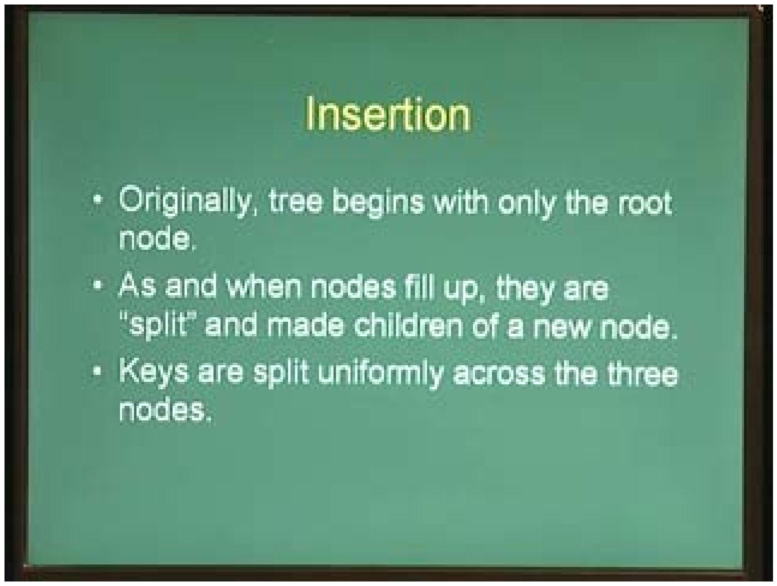
Generalization of Binary Search.

1. Given a search key k start from the root node
2. If key is present in current node then success; else
3. If current node is a leaf node and key not present in node, then key not in the database
4. Search for a tree pointer Pi such that $K_{i-1} < k \leq k_i$
5. Return to step 2 to continue search.

Let us first look at how we perform search in B trees and B plus trees. We are going to be looking at searches and additions and deletions in B plus trees and the corresponding algorithm for B tree can be derived in an analogous fashion. In fact the algorithm for B trees are little simpler than that of B plus trees. The searching algorithm over B plus trees is a generalization of binary search. Here it is a peary search in the sense that where P is a fan out of each block. So just like binary search we go about with a given key and starting from the root node of the tree, that is given a search key k start form the root node. If the key is present in the root node, in the current node itself then we are successful. That is the key corresponds to sorry, from the key we are able to end up, find the corresponding leaf node from where we can find a corresponding data pointer.

However if the current node is a leaf node in the B plus tree and key is not present then we can be sure that the key is not available in the database itself then we return not available or else what we do is we search for the different pointers such that the key value that we are looking for is embedded between the left and the right most keys. That is if we are searching for the first left, first sub tree that is P_1 then our key pointer should be lying between k_1 and k_2. If we are searching, if we want to search any P_i then we have to search or value, key value should lie between K_i and K_{i+1}. It is a matter of terminology, the slide shows K_{i-1} and K_i it is a matter of terminology whether how we use i minus 1 and i that is we can either say K_{i-1} and k_i or k_i and k_{i+1}. And we continue this search in the left sub tree in a recursive fashion by going back to step two and searching in that node and searching in a sub tree and so on.

(Refer Slide Time: 41:06)



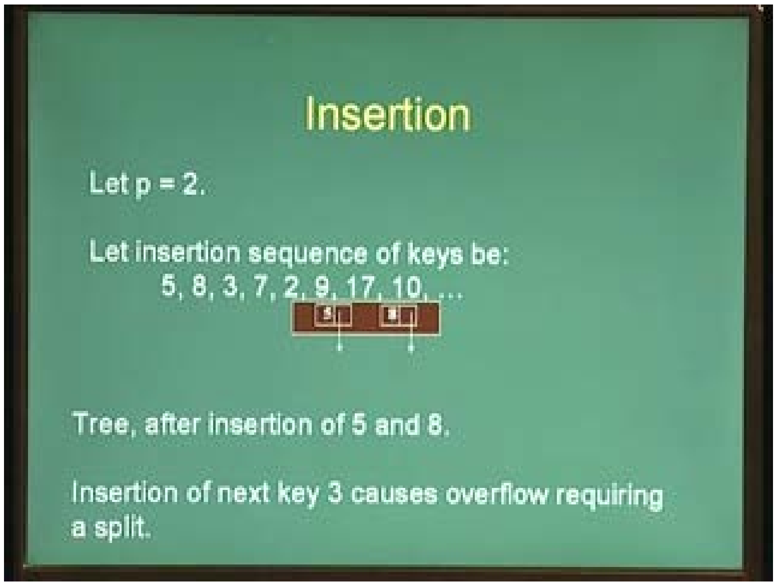
Insertion

- Originally, tree begins with only the root node.
- As and when nodes fill up, they are "split" and made children of a new node.
- Keys are split uniformly across the three nodes.

What about insertions? This is the main contribution of B plus trees in the sense that we will be able to insert records while maintaining the balanced property of the trees. Now we should be illustrating the process of insertion with an example and we shall not be going into the exact algorithm of insertion which can be referred to in any standard text books. However the illustration serves to help us clear the or help us to clarify the notion of how insertion happens within a B plus tree, the logic behind insertion in a B plus tree. To begin with B plus tree starts with a single node which is the root node and which is also leaf node. It has no parents and it has no children.

And the first key that is inserted into the database goes into the root node and because it is a leaf node, it just points to the corresponding data pointer. As and when nodes fill up that is as and when more and more records are inserted and more and more keys have to be incorporated into the tree, nodes get filled up. As and when they are filled up, nodes are split and this split nodes are made into children of a newly created node and the key values are split or also split correspondingly across these two nodes and the new parent node is updated accordingly. And this split operation is cascaded to levels above so that we end up with just one tree starting from one root node following until the leaf nodes and in a balanced fashion.

(Refer Slide Time: 43:11)



Insertion

Let p = 2.

Let insertion sequence of keys be:
5, 8, 3, 7, 2, 9, 17, 10, ...

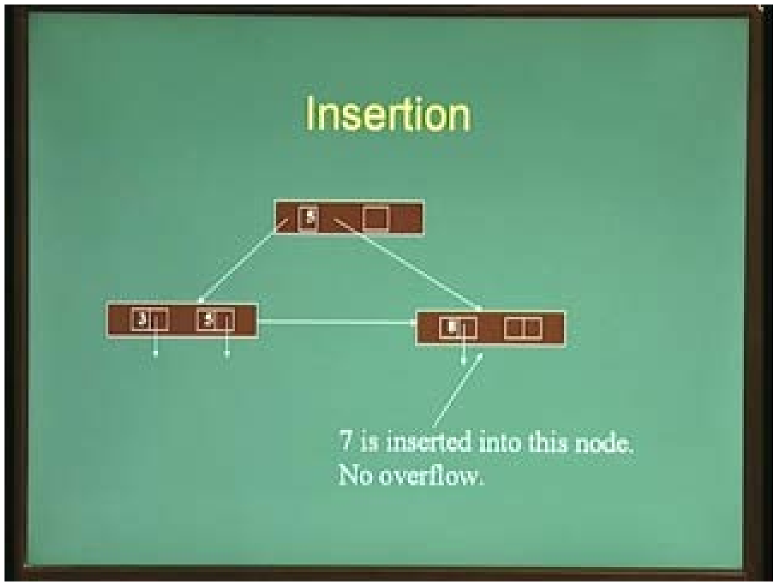
Tree, after insertion of 5 and 8.

Insertion of next key 3 causes overflow requiring a split.

Let us take an example to illustrate our point. Let p equal to 2, that is the fan out factor for just for the sake of simplicity we shall be assuming that the fan out factor is two. that is each node has only 2, has at most 2 children each internal node or root node has at most two children and let us consider a sequence in which records are different keys are inserted and a possible sequence is shown in the slide here that is keys are inserted in sequence 5 8 3 7 2 9 and so on.

So they appear in some arbitrary sequence that their sequence need not be ordered and we cannot ascribe any particular property in which keys are inserted. Initially when we insert five, we just have one root node and one key node and data pointer and nothing else in the tree. When we insert 8, it is still just one root node that is shown in the figure here and with two key pointers 5 and 8 and 2 data pointers, two corresponding data pointers. However when then next key is inserted that is key value 3 is inserted, this node overflows that because P equal to 2, we cannot accommodate any more keys in this node. Therefore we require a spilt in this key node. How do we split this node?

(Refer Slide Time: 44:47)



Insertion

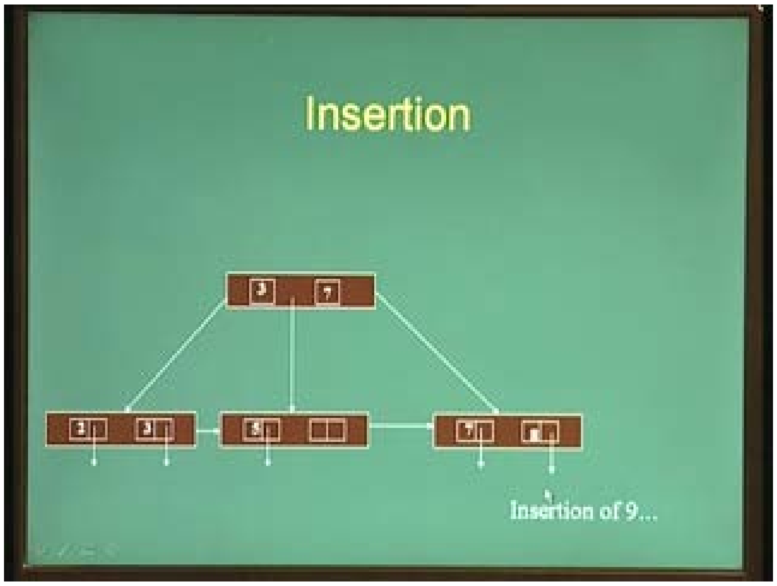
7 is inserted into this node.
No overflow.

Note how nodes have been split in this slide here. Initially we started with a root node which was also leaf node. When you spit this node, we get one more leaf node that is which is shown in right most side of this slide here and another intermediate node that is a node that points to two leaf nodes. In a B plus tree node that the intermediate node has a different structure than the leaf nodes, this has to be incorporated. So 5 and 8 were present in the tree and now key value 3 has to be inserted because 3 is less than 5 and note that the keys always, within any leaf nodes the keys always have to be in sorted form. Therefore the key 3 has to be inserted to the left of 5, to the left of this pointer called 5.

Therefore we get two different leaf nodes, one containing 3 and other containing 5 and 8. There is a small bug, there is a small error in this slide that is the left most node contains just the key 3 and the right most node contains the pointers 5 and 8 and the non-leaf node or the intermediate node is suitably updated so that 3 appears here that is everything less than or equal to 3 appears in this node and everything greater than 3 appears in this pointer. There is nothing else to be placed here because we don't have any other keys to begin. Therefore assume that we have got keys in the sequence 3 5 and 8, we would end up with a tree as shown in this figure here.

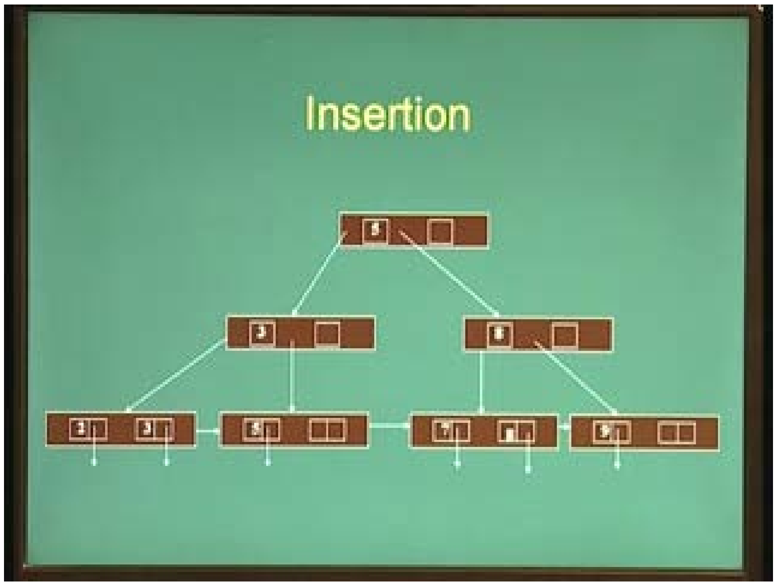
Now suppose 7 has to be inserted. Now, 7 can be inserted into this leaf node without any problem. That is 7 gets inserted here and the nodes and the key values are reordered. Earlier we had just 8 in this node and one 7 was inserted, the key values were reordered so that the keys are always sorted and there is no overflow. However the next key that is key value 2 causes another overflow that is key value 2 has to be inserted at the left most left most side here. This causes causes an overflow and this overflow has to be cascaded up or has to go up the level in the insertion or in the B tree.

(Refer Slide Time: 47:32)



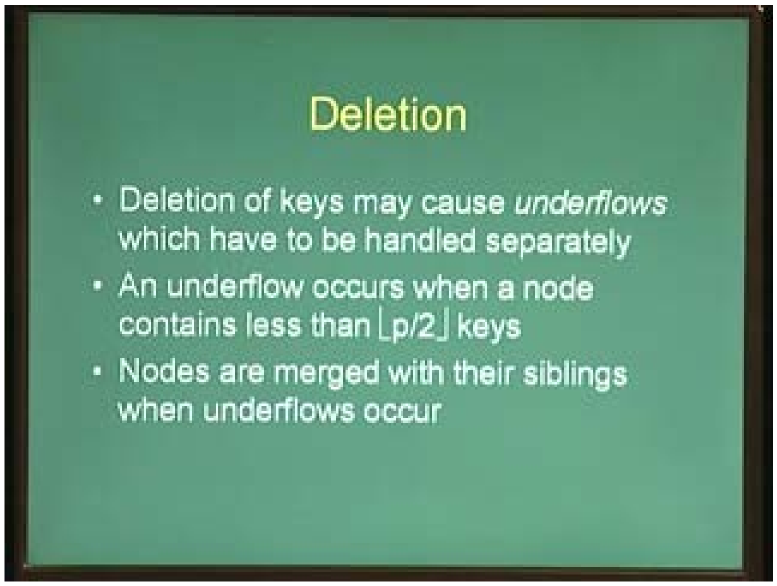
Therefore we get a B tree of the following form here that is both key values in the intermediate node now get filled and the intermediate node now points to three different leaf nodes. The insertion of 9 that is the next key value again forms a overflow because nine has to be inserted beyond the last block here. The algorithm tries to insert 9 into the last block which fails and then a new node is created.

(Refer Slide Time: 48:10)



Now, because a new node is created and we encounter another overflow. This overflow happens at the level above that is because a new node is created, we need 4 different pointers. However a node can accommodate only 3 different pointers here. Therefore we need to spit even the the node above and and introduce a new level into the tree. Therefore the corresponding tree that gets formed is shown in this figure here that is the next level node is also split so that a third level is created and then the keys are more or less uniformly distributed across the entire tree. As you can notice here the property or the balance property of the tree is maintained as and when the insertion is happening and all leaf nodes are at the same level. That is the height from the root of the tree is maintained at the same level for every leaf node in the index structure.

(Refer Slide Time: 49:16)



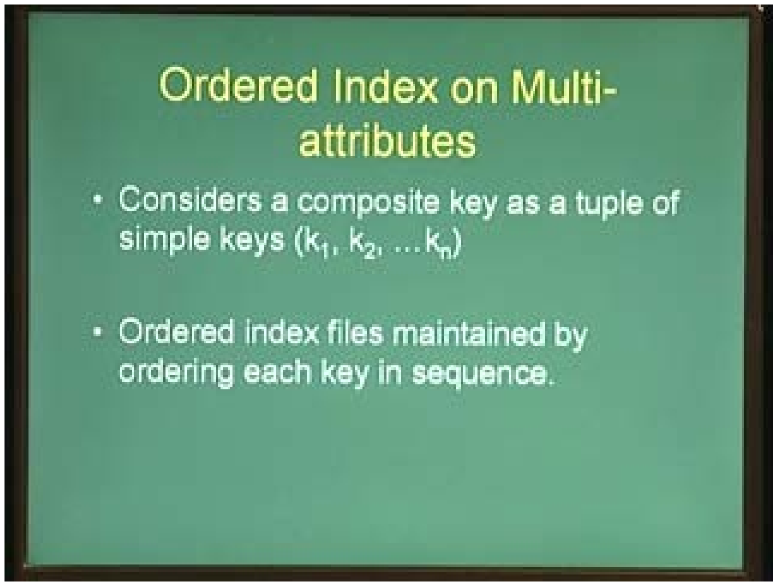
Deletion

- Deletion of keys may cause *underflows* which have to be handled separately
- An underflow occurs when a node contains less than $\lfloor p/2 \rfloor$ keys
- Nodes are merged with their siblings when underflows occur

So that was a brief illustration of how keys are inserted into a B plus tree and we shall not be going into the exact algorithm in order to insert keys in a B plus tree. Deletion of keys have to contend with an analogous problem that is the problem of underflow. In insertion we had the problem of overflow and in deletion we have the problem of underflow. An underflow happens when a node contains less than p by 2, floor of p by 2 keys less than or equal to floor of p by 2 keys. That is note that there was a constraint that alteast half of the keys in a node has to be filled up that is more than half rather. So if it contains less than half then of keys in a node then there is an underflow. Whenever there is a underflow a node is merged with its sibling in order to bring down number of levels in a tree. We shall not be going into deletion algorithms also in detail here.

Let us move on to the last aspect of indexing structures, namely how do we deal with index structures on multiple attributes. Until now we have assumed that indexing attributes are the fields on which indexes are maintained or simple indexes that is simple attributes. However the indexes could sometimes be maintained on composite attributes that is a set of attributes forming a key attribute. That is for example department id and employee number, to combine to form a key attribute is a composite attribute. How do we maintain indexes on a composite attribute? There are several different strategies that are variations or extensions of existing indexing structures and we shall briefly summarize some of the main techniques used for indexing multiple attributes.

(Refer Slide Time: 51:23)

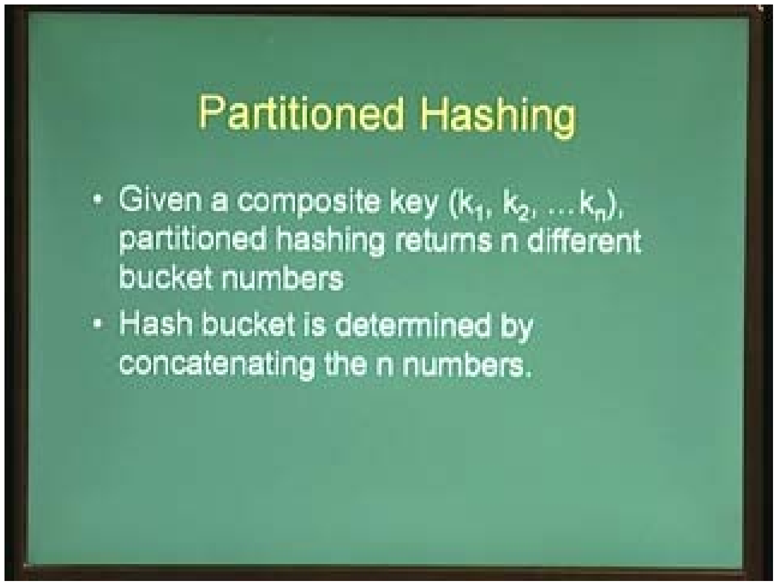


Ordered Index on Multi-attributes

- Considers a composite key as a tuple of simple keys (k_1, k_2, ... k_n)
- Ordered index files maintained by ordering each key in sequence.

One simple way is to index multiple attributes that is maintain a sorted file, a primary index of multiple attributes is to have an ordered index on multiple attributes. That is instead of sorting the file on just one attribute, we sort the file on two attributes that is sort it based on first attribute and among them sort based on second attribute which is the simplest solution possible but which can give us only a primary index as we have seen earlier that is it has to be a ordering attribute and a key attribute. The second strategy for dealing with composite attributes is to use what is called as partition hashing.

(Refer Slide Time: 52:13)

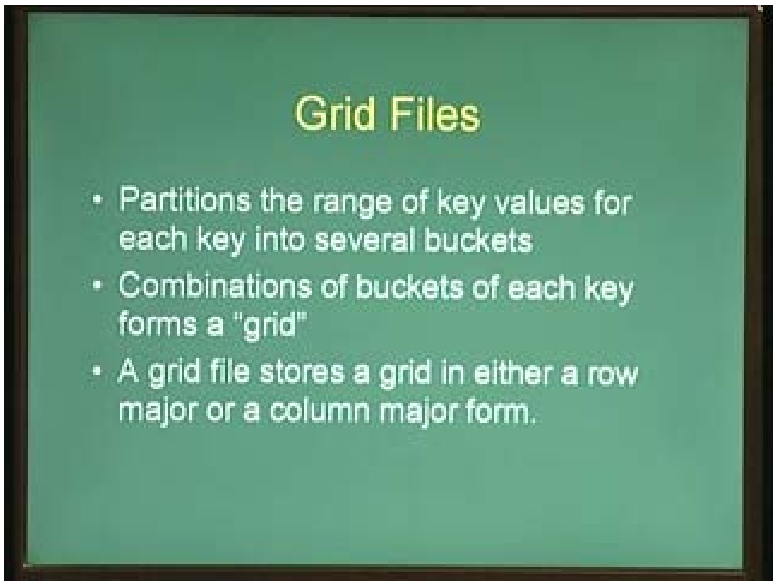


Partitioned Hashing

- Given a composite key (k_1, k_2, ... k_n), partitioned hashing returns n different bucket numbers
- Hash bucket is determined by concatenating the n numbers.

Partition hashing is a hashing technique which takes a composite attribute that is n different key elements pertaining to this attribute and returns n different bucket numbers. We then transform this into a single bucket address by concatenating all these bucket numbers to form the bucket address. So that is another technique for dealing with composite attribute. The third technique that is used especially in applications like data warehouses is to use the notion of grid files.

(Refer Slide Time: 52:50)

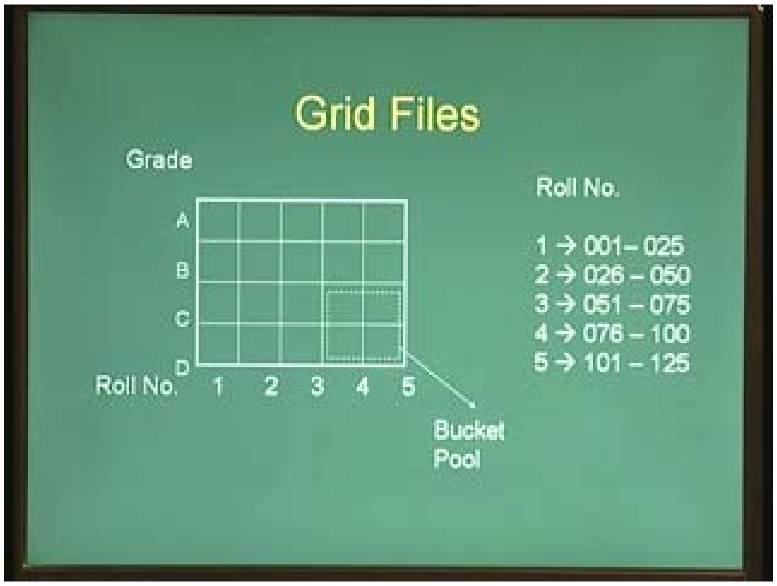


Grid Files

- Partitions the range of key values for each key into several buckets
- Combinations of buckets of each key forms a "grid"
- A grid file stores a grid in either a row major or a column major form.

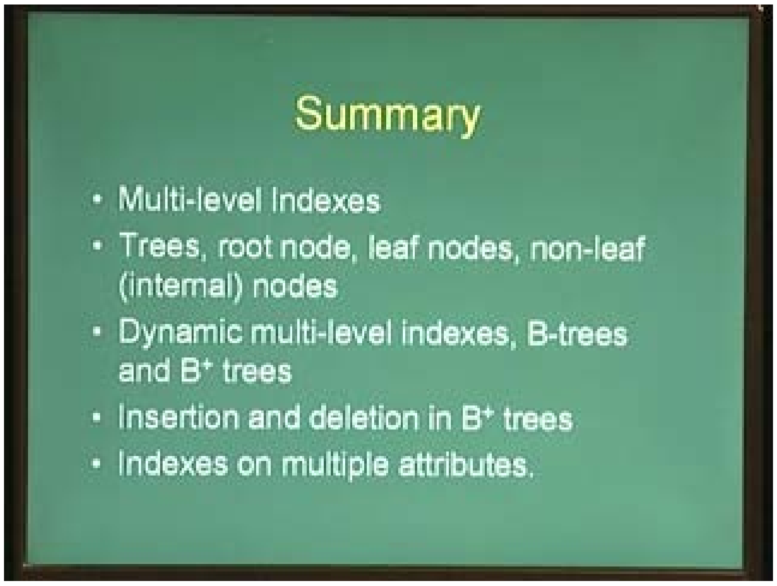
A grid file is a file that establishes grid structure. What is a grid structure? We can use a grid structure when we know the entire range in which a key value can be spread upon. For example if we know that the roll number of a student ranges form 01 to 150, we know the entire range of the set of all possible roll number. And if you are fairly sure that the distribution of key values in this range is fairly uniform then we can split this range into several different buckets. Now such splitting of key ranges into buckets and combining this different buckets forms a grid structure and these grid structures forms a matrix or a hypercube which can be stored within a single file using several techniques called row major techniques, column major techniques and other techniques called space filling curves and so on, using which they can be stored within a single file.

(Refer Slide Time: 54:03)


Grid Files

This slide shows such an illustration. That is there are two different key values roll number and grade. Roll numbers are ranging from 001 to 125 and they are divided into 5 different ranges, similar 5 different buckets. Similarly grades are divided into 5 different ranges A B C and D and the combinations of this form a grid and each cell in the grid corresponds to a set of bucket address or a set of block addresses which contain records that satisfy both this constraints of keys. So each pool or each cell corresponds to bucket pool.

(Refer Slide Time: 54:42)


Summary

- Multi-level Indexes
- Trees, root node, leaf nodes, non-leaf (internal) nodes
- Dynamic multi-level indexes, B-trees and B^+ trees
- Insertion and deletion in B^+ trees
- Indexes on multiple attributes.

This brings us to the end of this session. Let us quickly summarize the main topics that we have covered in this session. We covered several kinds of multi level indexes and multi level index has, index structures has several different levels and are usually organized in the form of a tree. A tree contains a root node which is the entry to the multi level index structure, several leaf nodes and many internal nodes that form the tree structure. And for a tree structure to be efficient, it has to be balanced and balancing or self balancing tree structures are what are implemented in dynamic multi level indexes in which we saw B trees and B plus trees. And, we also insertion how insertion and deletions are handled in B plus trees. Lastly, we looked at some strategies by using which we can maintain index structures on multiple attributes that brings us to the end of this session.