B^+-*Trees*

Fundamentals of Database Systems
Elmasri/Navathe
Chapter 6 (6.3)

Database System Concepts
Silberschatz/Korth/Sudarshan
Chapter 11 (11.3)

The Art of Computer Programming
Sorting and Searching, Vol 3
Knuth
p. 473-479 (Advanced)

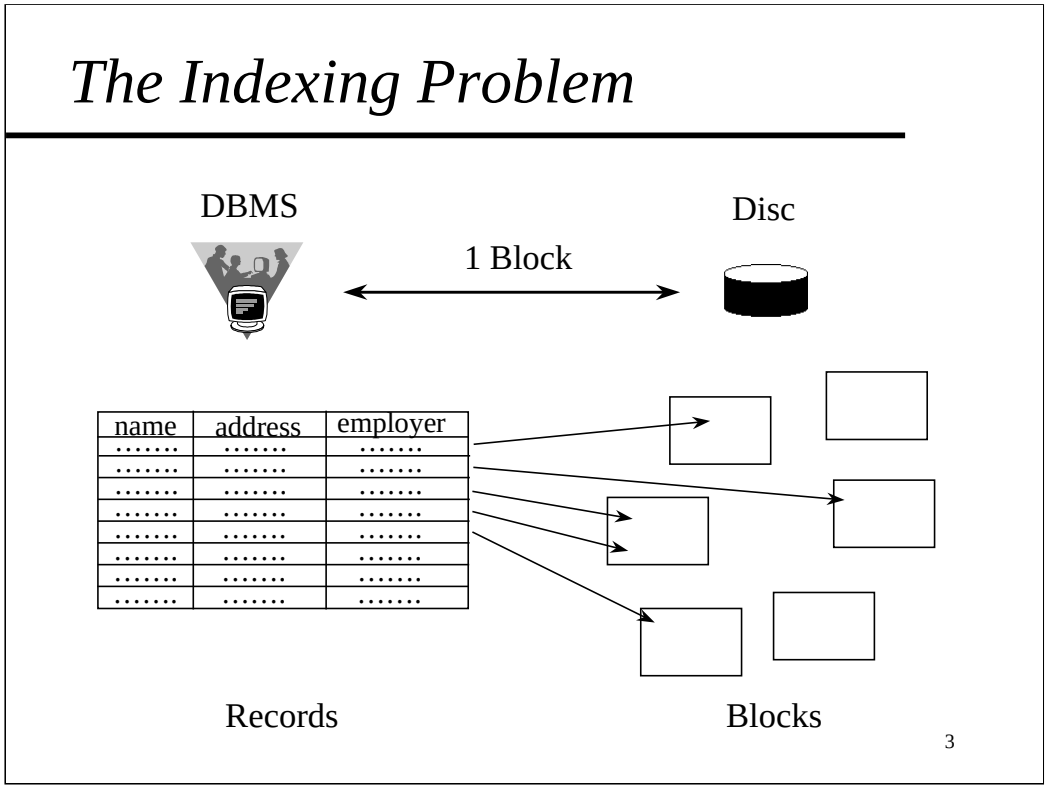
© Stephen Mc Kearney, 2001.

Overview

- ***The Indexing Problem***
  - **Problem**
  - **Simple index file**
  - **Multi-level indexes**
- *B^+-Tree Structure*
- *Algorithms*
  - Searching a B^+-Tree
  - Inserting in a B^+-Tree
  - Deleting from a B^+-Tree
- *B^+-Tree Size*
  - Order of a B^+-Tree
  - Size of a B^+-Tree
- *Properties of a B^+-Tree*
  - Size
  - Height
  - Growth
- *B-Trees in Oracle*

2

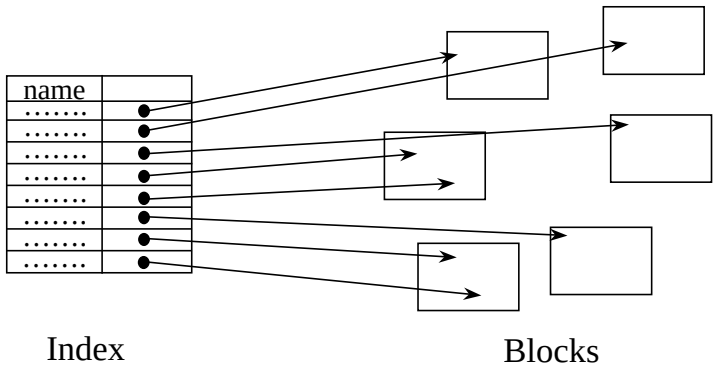
© Stephen Mc Kearney, 2001.



- Elamasri et al describes an index as being “used to speed up the retrieval of records in response to certain search conditions”.
- An index speeds up certain queries or searches because it stores information about where data is stored on the disc. The index points directly to the location of a record on the disc and can be used to avoid searching a large file.
- The DBMS represents data as records in a table. However, a disc stores data in blocks, or pages. Many records may be placed in one block or one record may be placed across many blocks.
- The computer can only transfer one block at a time between main memory and the disc. This means that to retrieve one record a whole block of records must also be retrieved. Therefore, it is important to know in which block a record is stored and to retrieve the minimum number of blocks when looking for a record.
- The problem for the DBMS is to decide in which block each record should be placed and what information should be stored in addition to the record to allow the record to be retrieved easily.

© Stephen Mc Kearney, 2001.

Simple Index File



Smaller index file is quicker to search

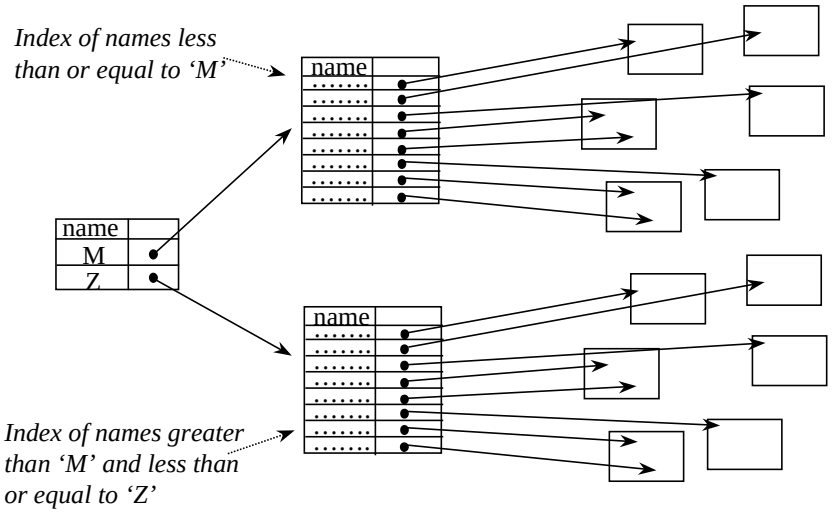
4

- A simple index file consists of a table with two columns (attributes). The first column is a list of values from the records, for example, the primary key, and the second column is a pointer to the block containing the record identified by the value.
- The two values are called an *index record*.
- For example, a simple index on *name* would consists of a list of index records. Each index record contains a name and a pointer to the block containing a record identified by the name.
- The index is stored in one or more blocks on the disc. Hence, the index must also be retrieved from the disc one block at a time.
- However, because each index record only contains a single attribute (the primary key) and a pointer, the index is physically smaller than the data file. Therefore, more index records than ordinary records can be placed in a single block.
- If more index records can be placed in a single block then the number of blocks required to store the whole index is less than the number of blocks required to store the original file. Hence the index will be quicker to search because less blocks must be read.

Ref: Elmasri, sec 6.1/6.2; Silberschatz, sec. 11.2.

© Stephen Mc Kearney, 2001.

Multi-Level Index



An index of an index

5

- When the number of indexed values is large, the index will not fit in one block. Therefore, the contents of the index must be placed in two or more blocks.
- As the number of blocks required to store the index grows, so searching the index becomes a major problem. In the same way that a data file takes a long time to search, a large index (occupying many blocks) will also take a long time to search.
- The solution to this problem is to create an *index of an index*. That is, the single index is split into a number of blocks and a new index is created that indexes each block.
- For example, to index a set of names we might split the name index between two blocks. All names beginning with 'M' or less are placed in block 1 and all names beginning with letters greater than 'M' are placed in block 2. A third block is created points to blocks 1 and 2.
- The B^+-Tree structure is an index of an index, called *multi-level index*. The algorithms that are used to manipulate the B^+-Tree control the number of blocks in each index and the number of indexes required.

Ref: Elmasri, sec 6.2;

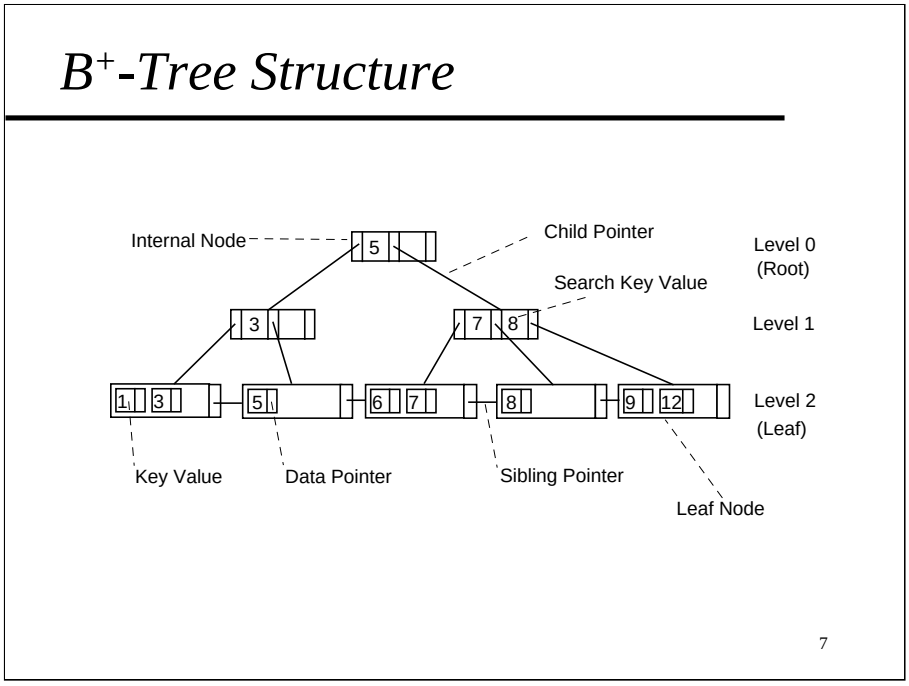
© Stephen Mc Kearney, 2001.

Overview

- *The Indexing Problem*
  - Problem
  - Simple index file
  - Multi-level indexes
- ***B$^+$-Tree Structure***
- *Algorithms*
  - Searching a B$^+$-Tree
  - Inserting in a B$^+$-Tree
  - Deleting from a B$^+$-Tree
- *B$^+$-Tree Size*
  - Order of a B$^+$-Tree
  - Size of a B$^+$-Tree
- *Properties of a B$^+$-Tree*
  - Size
  - Height
  - Growth
- *B-Trees in Oracle*

6

© Stephen Mc Kearney, 2001.

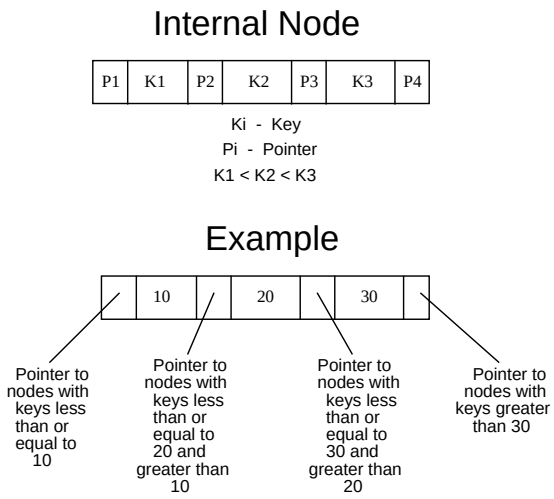


- A B^+-Tree consists of one or more blocks of data, called *nodes*, linked together by pointers. The B^+-Tree is a tree structure. The tree has a single node at the top, called the *root node*. The root node points to two or more blocks, called *child nodes*. Each child nodes points to further child nodes and so on.
- The B^+-Tree consists of two types of (1) *internal nodes* and (2) *leaf nodes*:
  - Internal nodes point to other nodes in the tree.
  - Leaf nodes point to data in the database using *data pointers*. Leaf nodes also contain an additional pointer, called the *sibling pointer*, which is used to improve the efficiency of certain types of search.
- All the nodes in a B^+-Tree must be at least half full except the root node which may contain a minimum of two entries. The algorithms that allow data to be inserted into and deleted from a B^+-Tree guarantee that each node in the tree will be at least half full.
- Searching for a value in the B^+-Tree always starts at the root node and moves downwards until it reaches a leaf node.
- Both internal and leaf nodes contain *key values* that are used to guide the search for entries in the index.
- The B^+-Tree is called a *balanced tree* because every path from the root node to a leaf node is the same length. A balanced tree means that all searches for individual values require the same number of nodes to be read from the disc.

Ref: Elmasri, sec 5.3, p116-117; Silberschatz, sec 11.3.

© Stephen Mc Kearney, 2001.

Internal Nodes



8

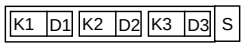
- An *internal node* in a B^+-Tree consists of a set of *key values* and *pointers*. The set of keys and values are ordered so that a pointer is followed by a key value. The last key value is followed by one pointer.
- Each pointer points to nodes containing values that are *less than or equal to* the value of the key immediately to its right. For instance, in the example, above, the first pointer points to a node containing key values that are less than or equal to 10.
- The last pointer in an internal node is called the *infinity pointer*. The infinity pointer points to a node containing key values that are greater than the last key value in the node.
- When an internal node is searched for a key value, the search begins at the leftmost key value and moves rightwards along the keys.
    - If the key value is less than the sought key then the pointer to the left of the key is known to point to a node containing keys less than the sought key.
    - If the key value is greater than or equal to the sought key then the pointer to the left of the key is known to point to a node containing keys between the the previous key value and the current key value.

Ref: Elmasri, sec 6.3; Silberschatz, sec 11.3

© Stephen Mc Kearney, 2001.

Leaf Nodes

Leaf Node



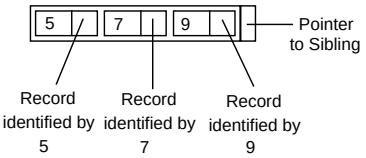
Ki - Key

Di - Data Pointer

S - Sibling Pointer

K1 < K2 < K3

Example



9

- A *leaf node* in a B^+-Tree consists of a set of *key values* and *data pointers*. Each key value has one data pointer. The key values and data pointers are ordered by the key values.
- The data pointer points to a record or block in the database that contains the record identified by the key value. For instance, in the example, above, the pointer attached to key value 7 points to the record identified by the value 7.
- Searching a leaf node for a key value begins at the leftmost value and moves rightwards until a matching key is found.
- The leaf node also has a pointer to its immediate *sibling node* in the tree. The sibling node is the node immediately to the right of the current node. Because of the order of keys in the B^+-Tree the sibling pointer always points to a node that has key values that are greater than the key values in the current node.
- Leaf nodes always appear at the bottom of the B^+-Tree structure.

Ref: Elmasri, sec 6.3; Silberschatz, sec 11.3.

© Stephen Mc Kearney, 2001.

Overview

- *The Indexing Problem*
  - Problem
  - Simple index file
  - Multi-level indexes
- *B^+-Tree Structure*
- ***Algorithms***
  - **Searching a B^+-Tree**
  - **Inserting in a B^+-Tree**
  - **Deleting from a B^+-Tree**
- *B^+-Tree Size*
  - Order of a B^+-Tree
  - Size of a B^+-Tree
- *Properties of a B^+-Tree*
  - Size
  - Height
  - Growth
- *B-Trees in Oracle*

10

See B+-Tree Handout

© Stephen Mc Kearney, 2001.

Overview

- *The Indexing Problem*
  - Problem
  - Simple index file
  - Multi-level indexes
- *B^+-Tree Structure*
- *Algorithms*
  - Searching a B^+-Tree
  - Inserting in a B^+-Tree
  - Deleting from a B^+-Tree
- ***B^+-Tree Size***
  - **Order of a B^+-Tree**
  - **Size of a B^+-Tree**
- *Properties of a B^+-Tree*
  - Size
  - Height
  - Growth
- *B-Trees in Oracle*

11

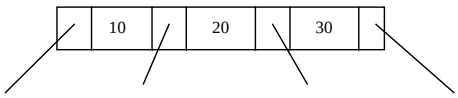
© Stephen Mc Kearney, 2001.

Order of a B$^+$-Tree

The *order* of a B$^+$-Tree is the maximum number of keys and pointers that can internal node can hold.

A B$^+$-Tree of order *m* can hold *m-1* keys and *m* pointers.

e.g. A node of order 4.



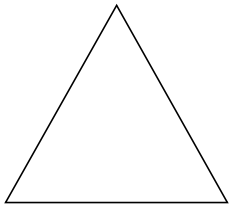
12

- The *order* of a B$^+$-Tree is the number of keys and pointers that an internal node can contain. An order size of *m* means that an internal node can contain *m-1* keys and *m* pointers.
- The order size is important because it determines how large a B$^+$-Tree will become.
- For example, if the order size is small then fewer keys and pointers can be placed in one node and so more nodes will be required to store the index. If the order size is large then more keys and pointers can be placed in a node and so fewer nodes are required to store the index.

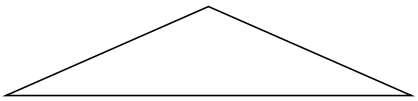
Ref: Elmasri, sec 6.3; Silberschatz, sec 11.3.

© Stephen Mc Kearney, 2001.

Order Size & Tree Height



Small Order Size



Large Order Size

When two B^+-Trees have the same number of keys,

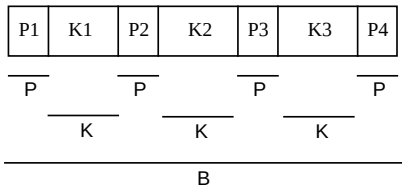
the order size determines the height.

13

- The size of a B^+-Tree is measured by:
  - The current number of keys and pointers in each node.
  - The order size.
  - The number of levels in the tree (the height).
  - The number of nodes in the tree.
- The size and shape of the tree determines the number of nodes in the tree which affects the number of nodes that must be read to execute a search. Therefore, it is important to understand how the order size and number of nodes affects the size and shape of the tree.
- A tree with a small order size will require more nodes to store a fixed number of keys. Having a small order size for nodes in the tree makes the tree taller and thinner. This is because the same number of keys must fit into smaller nodes which requires more nodes.
- A tree with a large order size will require fewer nodes to store a fixed number of keys. Having a large order size for nodes in the tree makes the tree smaller and wider. This is because the same number of keys must fit into larger nodes which requires fewer nodes.

© Stephen Mc Kearney, 2001.

Calculating the Order Size



B = Size of block
K = Size of Key
P = Size of Pointer
O = Order of tree

Block Size

$$B \geq (O \times P) + ((O-1) \times K)$$

Order Size

$$\frac{B+K}{P+K} \geq O$$

14

Working

$$B \geq (O \times P) + ((O-1) \times K)$$
$$B \geq OP + (OK - K)$$
$$B \geq OP + OK - K$$
$$B + K \geq OP + OK$$
$$B + K \geq O(P + K)$$
$$\frac{B+K}{P+K} \geq O$$

Example

$K = 9$
$P = 6$
$B = 512$

$$\frac{512+9}{6+9} \geq O$$
$$\frac{521}{15} \geq O$$
$$34 \geq O$$

Ref: Elmasri, p121 (example more complex).

© Stephen Mc Kearney, 2001.

Size of a B^+-Tree

Assume : 69% full (average)

Order = 34

Average number of pointers = 0.69 x 34 = 23

23 pointers per node
22 keys per node

	Nodes	Entries	Pointers
Root:	1	22	23
Level 1:	23	506	529
Level 2:	529	11,638	12,167
Leaf :	12, 167	267,674	

15

- In the example above, each node is assumed to be 69% full on average and to have an order size of 34. Therefore, the number of pointers and keys in each node is on average 23 pointers and 22 keys.
- This means that the root node contains pointers to 23 level 1 nodes. Each of the 23 level 1 nodes contains 23 pointers to level 2 nodes, and so on.
- This means that accessing *any* one record out of the 267,674 records in this tree requires four nodes to be read from the disc.

Ref: Elmasri, p174 (example more complex).

© Stephen Mc Kearney, 2001.

Size of B^+-Tree

Maximum Number of Nodes at each Level
(m = order of B^+-Tree)

Level	Nodes
0	1
1	2
2	2(m/2)
3	2(m/2)(m/2) or $2(m/2)^2$
…	…
l	$2(m/2)^{l-1}$

16

- The size of a B^+-Tree depends on the number of keys and pointers in each node. The B^+-Tree structure guarantees that each node cannot be less than half full.
- Therefore, the maximum size of a B^+-Tree can be identified by calculating the maximum number of nodes (of a particular order size) are required to store the keys. The maximum number of nodes in a B^+-Tree occurs when all the nodes are half full.
- When all the nodes are half full it is not possible to create a new node without adding new data. This is because creating a new node would involve moving some keys/pointers from an existing node into the new node. This would make the existing node less than half full which is not allowed.
- Therefore, by assuming that each node is half full (*m/2*) it is possible to calculate the theoretical maximum size of a B^+-Tree.
    - The first level in the tree, level 0, is the root node. The root node may contain a minimum of two pointers. Therefore, it must point to two or more nodes on level 1.
    - In the example above, level 0 has one node, the root, which points to two nodes on level 1. Therefore, level 1 contains two nodes. The two nodes on level 1 must be half full and so point to *m/2* nodes which each must point to *m/2* nodes.

Ref: Knuth, vol 3.

© Stephen Mc Kearney, 2001.

Height of a B⁺-Tree

"A B⁺-Tree with N keys has $N+1$ leaf nodes"

therefore, $N+1 \geq 2\left\lceil \frac{m}{2} \right\rceil^{l-1}$

therefore, $\underbrace{1+\log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)}_{\text{maximum number of levels}} \geq \underbrace{l}_{\text{number of levels}}$

If the number of keys, N, and the order size, m, are known then the maximum number of levels can be calculated.

17

Example

$$N = 1{,}999{,}998$$
$$m = 199$$
$$1+\log_{\lceil 199/2 \rceil}\left(\frac{1{,}999{,}998+1}{2}\right) \geq l$$
$$1+\log_{\lceil 199/2 \rceil}(999{,}999.5) \geq l$$
$$1+2.9999 \geq l$$
$$3.9999 \geq l$$

Therefore, to store 1,999,998 keys we need a B⁺-Tree of at most 3 levels.

Ref: Knuth, p476.

© Stephen Mc Kearney, 2001.

Overview

- *The Indexing Problem*
  - Problem
  - Simple index file
  - Multi-level indexes
- *B^+-Tree Structure*
- *Algorithms*
  - Searching a B^+-Tree
  - Inserting in a B^+-Tree
  - Deleting from a B^+-Tree
- *B^+-Tree Size*
  - Order of a B^+-Tree
  - Size of a B^+-Tree
- ***Properties of a B^+-Tree***
  - **Size**
  - **Height**
  - **Growth**
- *B-Trees in Oracle*

18

© Stephen Mc Kearney, 2001.

Properties of a B^+-Tree

- The root node points to at least two nodes.
- All non-root nodes are at least half full.
- For a tree of order *m*, all internal nodes have *m-1* keys and *m* pointers.
- A B^+-Tree grows upwards.
- A B^+-Tree is balanced.
- Sibling pointers allow sequential searching.

19

- The root node in a B^+-Tree can have a minimum of one key and two pointers. This is because as the tree grows it is necessary to create a new root by splitting the current root. Splitting the current root creates two nodes which must be pointed to by the new root.
- All non-root nodes in the B^+-Tree, that is, internal and leaf nodes, can be at least half full. This is because when nodes are full they are split in half. When keys are deleted from the tree, nodes are reorganised to guarantee each node is half full.
- The order of a B^+-Tree is the maximum number of pointers in an internal node.
- Because the B^+-Tree always fills up from the leaf nodes it only possible to split the root node after all nodes between it and a leaf node have been split. Hence, the tree 'grows upwards'.
- As all paths from the root node to a leaf node are the same length, the tree is said to be balanced. A balanced tree guarantees that all searches for individual keys require the same number of nodes to be read because all the paths from the root to the leaf nodes are the same length.
- The sibling pointers between leaf nodes allow sequential searches to be carried out efficiently. The sibling pointer always points to a node that contains keys greater than the keys in the current node.

© Stephen Mc Kearney, 2001.

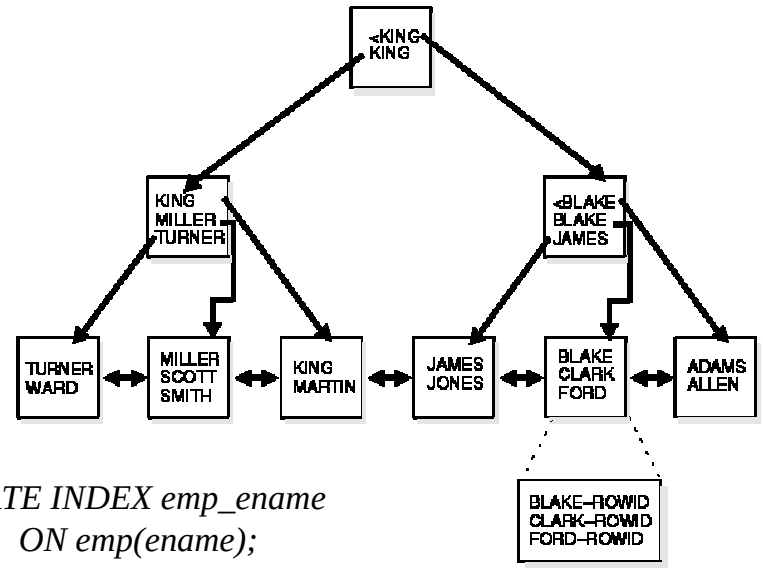
Overview

- *The Indexing Problem*
  - Problem
  - Simple index file
  - Multi-level indexes
- *B^+-Tree Structure*
- *Algorithms*
  - Searching a B^+-Tree
  - Inserting in a B^+-Tree
  - Deleting from a B^+-Tree
- *B^+-Tree Size*
  - Order of a B^+-Tree
  - Size of a B^+-Tree
- *Properties of a B^+-Tree*
  - Size
  - Height
  - Growth
- ***B-Trees in Oracle***

20

© Stephen Mc Kearney, 2001.

Oracle B-Tree*



CREATE INDEX emp_ename
ON emp(ename);

Oracle Concepts Manual

Advantages of B-Tree Structure*

- All leaf blocks of the tree are at the same depth, so retrieval of any record from anywhere in the index takes approximately the same amount of time.
- B*-tree indexes automatically stay balanced.
- All blocks of the B*-tree are three-quarters full on average.
- B*-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.
- Inserts, updates, and deletes are efficient, maintaining key order for fast retrieval.
- B*-tree performance is good for both small and large tables, and does not degrade as the size of a table grows.

Oracle Concepts Manual

© Stephen Mc Kearney, 2001.

Oracle Index-Organised Tables

Regular Tables	Index-Organised Tables
• ROWID uniquely identifies a row; primary key can be optionally specified	• Primary key uniquely identifies a row; primary key must be specified
• Implicit ROWID column; allows physical secondary indexes	• No implicit ROWID column; cannot have physical secondary indexes
• ROWID based access	• Primary key based access
• Sequential scan returns all rows	• Full-index scan returns all rows in primary key order
• UNIQUE constraint and triggers allowed	• UNIQUE constraint not allowed but triggers are allowed
• A table can be stored in a cluster containing other tables.	• An index-organized table cannot be stored in a cluster.
• Distribution, replication, and partitioning supported	• Distribution, replication, and partitioning not supported

© Stephen Mc Kearney, 2001.