Indexing

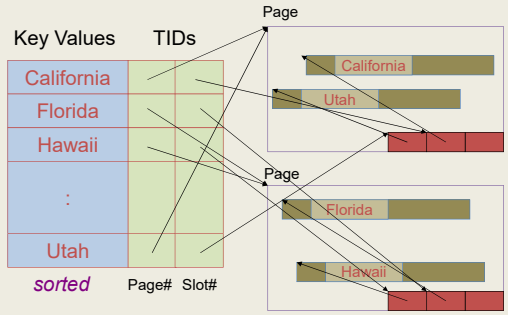
- Consider an index prepared in some textbook
  - When you want to find places related to a keyword
  - You will find pages of the index in the book at first
  - Then search the keyword in the index assuming all keywords are sorted
  - If you find the keyword, you will check page number written in the same row of the keyword, and go to the page
  - If the textbook has no index, you have to read all pages to find the keyword
- The concept of indexing in a database system is similar to indexing in textbooks

Basic Structures of Indices

- Tuples can be stored and retrieved based on the value of their key attribute
  - Associative Access
- Location of a tuple is indicated by a TID
  - Prepare data structures to derive a TID from a value
- Inverted Table (Inverted File)
  - Store a table for mapping TIDs with values of a key
  - Entries are sorted for the binary search
  - The size of the table is grown by increase of the number of tuples
    - Increase Search and Maintaining Cost

Inverted Table

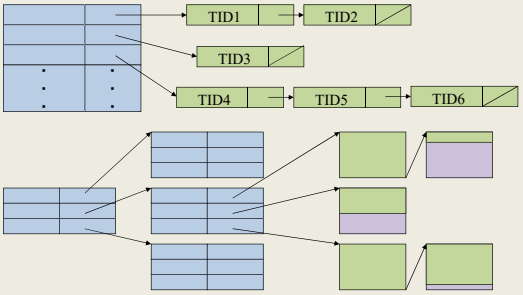
contents → address



Problems of Inverted Tables

- The inverted table is also stored into a disk
- If the number of tuples (n) increases
  - The inverted file use multiple disk pages
- Cost for key value search
  - $\log_2(n)$ by the binary search method
    - The binary search for multiple disk page is inefficient
- Cost for insert a new key value
  - To migrate entries between disk pages or to sort all entries again and save them into disk papges

Inverted List and Index Tree



Problem of skews

B-tree

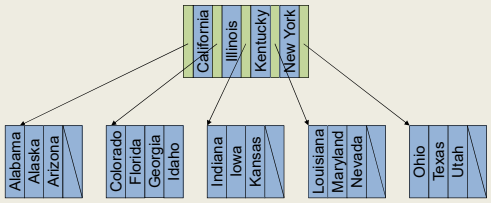
- First described in a paper by Bayer and McCreight [1972] (Proposed by Bayer)
- A B-tree of order F is a tree which satisfies the following properties:
  - Every node has $\leq F+1$ sons
  - Every node, except for the root and the leaves, has $\geq F/2+1$ sons
  - The root has at least two sons (unless it is a leaf)
  - All leaves appear on the same level
  - A nonleaf node with j sons contains j-1 keys

Implementation of B-tree

- A tree node is stored into a disk page
- It guaranties the maximum number of disk accesses to search for a key value
  - Because paths from the root to all leaves have the same length
- It guaranties the minimum occupancy of 50%
  - The average occupancy of 69%
  - Because a node has at least $F/2$ entries
- B-tree can be used as file organization and clustering as well as access path

Illustrations of a B-tree

$F = 4$

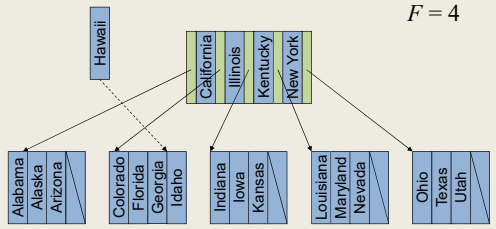


- Illustrate only key values for simplicity
  - Also contain TIDs or body of tuples

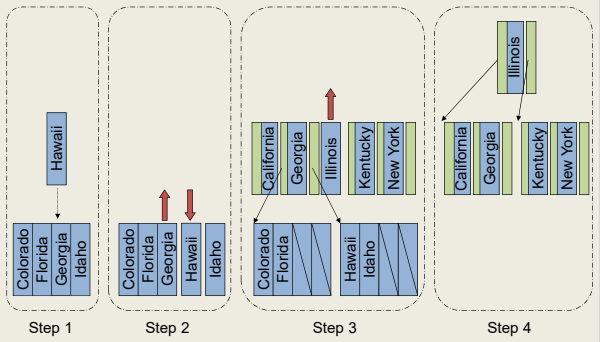
Inserting into a B-tree

- If a new tuple has to be inserted into a page that already holds the maximum number of entries
  - This page split into two pages: the old one and a newly allocated page
  - The existing entries are distributed across the two pages; one gets the lower half, the other one the upper half of the tuples
  - The dividing key between these two pages is propagated up to the upper page holding the pointer to the split page
  - When the split escalates up to the root node, height increases

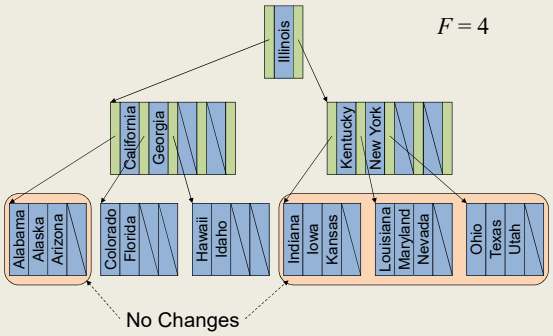
Insertion Process (1)



Insertion Process (2)



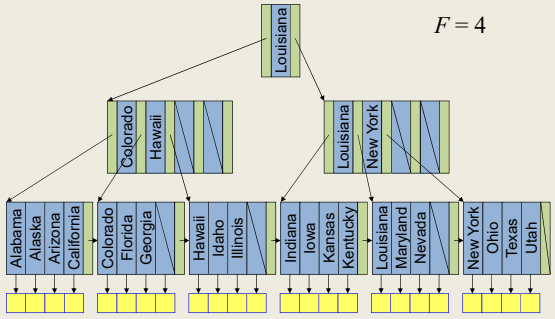
After the Insertion



B+-tree

- Data or TIDs are stored only at the leaf nodes
- The leaf nodes have an entry for every value of the key
- The leaf nodes are linked to provide ordered access
  - The links are useful for the rage queries
- The most popular variations of the B-tree
  - All commercial products (such as ORACLE and DB2) have adopted the B+-tree variation
  - The B+-tree is often referred to by the simpler name B-tree

Illustrations of B+-tree



Performance Aspects of B-trees

N: Number of tuples in the database
F: Maximum number of entries in an index node; by F^* we denote the average number of entries in an index node
C: Maximum number of entries in a leaf node; by C^* we denote the average number of entries in a leaf node
k: (Average) length of a key value
t: (Average) length of a tuple
p: Length of a pointer or a TID
B: Effective storage capacity of a page (page size minus length of administrational data)
u: Average node occupancy; we assume the same average occupancy for both leaf and index nodes

Entries and Leaves

- Calculate the average number of entries
  $C^* = \lfloor (B/(k+t)) \times u \rfloor$ (Storing tuples in leaves)
  $C^* = \lfloor (B/(k+p)) \times u \rfloor$ (Storing TIDs in leaves)
  $F^* = \lfloor (B/(k+p)) \times u \rfloor$
- To store N tuples,
  $\lceil N/C^* \rceil$
  leaf pages are required
- Since each index node can point to F^* successors, the first level above leaves has
  $\lceil \lceil N/C^* \rceil / F^* \rceil$

Height of the B-tree

- Calculate how often
  $\lceil N/C^* \rceil$
  can be divided by F^* until the result is less than 1
  $H = 1 + \lceil \log_{F^*}(\lceil N/C^* \rceil) \rceil$
- or
  $N = C^* \times F^{*(H-1)}$

An Example

	Storing tuples (C*=43)	Storing TIDs (C*=300)
H	*N (max)*	*N (max)*
2	12,900	90,000
3	3,870,000	27,000,000
4	1,161,000,000	8,100,000,000
5	348,300,000,000	2,430,000,000,000

- B=8,000 Byte, k=10 Byte, t=100 Byte, p=6 Byte, u=0.6

Assignment 5

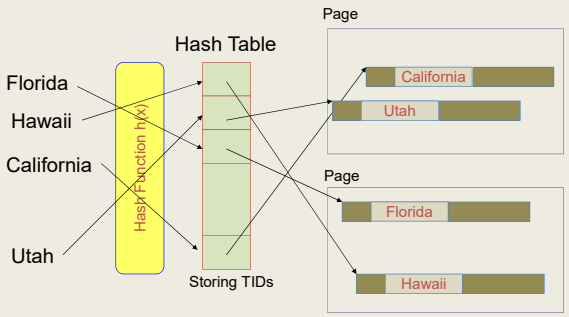
- Assuming:
  - Average seek time of the disk =2ms
  - Rotation Speed of the disk = 12,000 rpm
  - Data transfer bandwidth = 20MB/s
  - A page size 4KB (4096Byte)
  - Effective storage capacity (B) 4000Byte
  - k=20 Byte, t=100 Byte, p=4 Byte, u=0.6

1. Derive the maximum number of tuples for a B-tree storing TIDs, of which height is 2 and 3
2. Calculate access time to derive a TID of a tuple by the B-tree stored into the disk for 500,000 tuples

Hashing

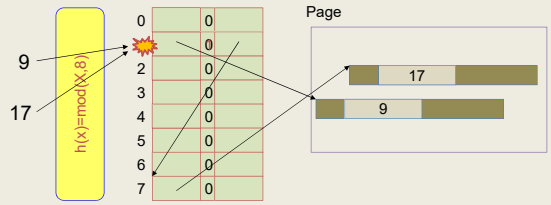
- Hash files are designed to provide fast access to one tuple via an attribute or concatenation of attributes
  - TID does just that, but it is a system-generated, internal identifier, which normally does not have any significance for the application
  - The attribute used for hash file access is an external attribute
- Use the same hash function for storing and retrieving
- An example of a hash function: h(X) = mod(X, n)
  - n is the size of the hash table
- Folding methods are used for character strings

Hashing



Hash Collision

- Different keys have the same hash value
  - Ex: h(X) = mod(X, 8), h(9) = h(17) = 1
- Use other entries for the keys
- Search cost is not constant



Extensible Hashing (1)

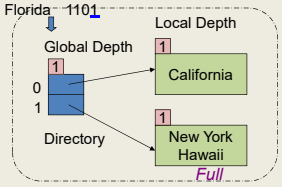
- A fixed number of entries in a hash table becomes a problem for the scalability
- By the extensible hashing, the number of entries in a hash table can grow and shrink with usage (a power of two)
- A hash function produces a bit string S for each key value
- Then d bits are taken out of S from a defined position
  - d is called as a depth

Extensible Hashing (2)

- The hash table has a global depth, and each hash bucket has a local depth
- An example:
  - Assuming two tuples can be store into a bucket

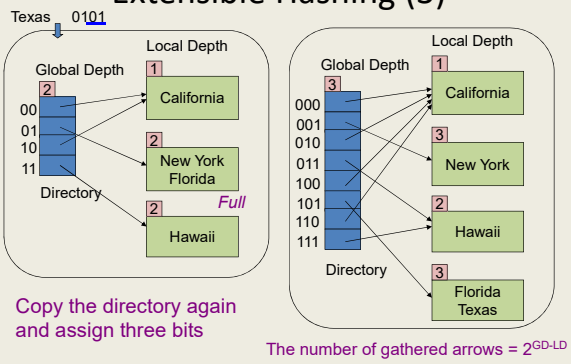
Key value	S
New York	1001
California	0100
Hawaii	0011
Florida	1101
Texas	0101

Take out d bits from LSB



Copy the directory and assign two bits

Extensible Hashing (3)



Copy the directory again and assign three bits

The number of gathered arrows = 2^{GD-LD}

Extensible Hashing (4)

- An inverted table or a smaller hash table is placed into a hash bucket
- To access a record, probe the directory of the extensible hashing at first, access a page corresponding to the hash bucket, and derive the TID by the inverted table or the small hash table
- The directory may use a number disk pages but can be distinguished by the bit string
- The number of disk access is two to derive a TID
- However, costs for extensions are high because of requiring whole copy of the directory