

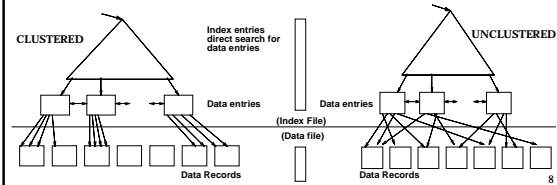
Index Classification

- ❖ **Primary vs. secondary:** If search key contains primary key, then called primary index.
 - *Unique index:* Search key contains a candidate key.
- ❖ **Clustered vs. unclustered:** If order of data records is the same as, or 'close to', order of data entries, then called clustered index.
 - A file can be clustered on at most one search key.
 - Cost of retrieving data records through index varies greatly based on whether index is clustered or not!

7

Clustered vs. Unclustered Index

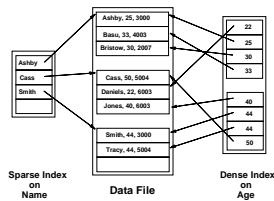
- ❖ Suppose that the data records are stored in a Heap file.
 - To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
 - Overflow pages may be needed for inserts. (Thus, order of data recs is 'close to', but not identical to, the sort order.)



8

Index Classification (Contd.)

- ❖ **Dense vs. Sparse:** If there is at least one data entry per search key value (in some data record), then dense.
 - Every sparse index is clustered!
 - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



9

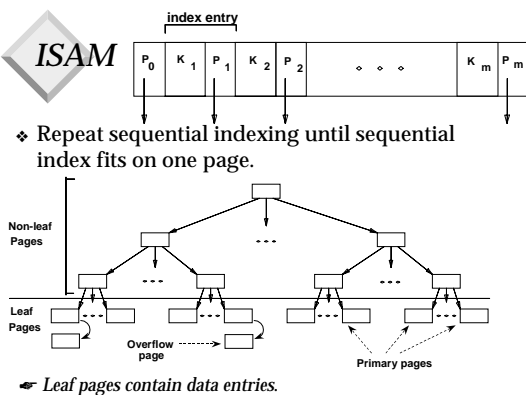
Tree-Structured Indices

- ❖ Tree-structured indexing techniques support both *range searches* and *equality searches*.
- ❖ **ISAM:** static structure; **B+ tree:** dynamic, adjusts gracefully under inserts and deletes.

10

ISAM

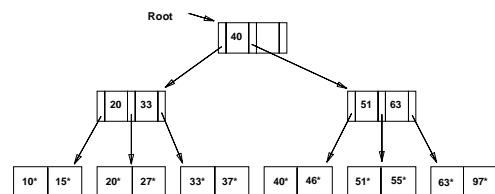
- ❖ Repeat sequential indexing until sequential index fits on one page.



11

Example ISAM Tree

- ❖ Each node can hold 2 entries; no need for 'next-leaf-page' pointers. (Why?)



12

Comments on ISAM

Data Pages
Index Pages
Overflow pages

- ❖ **File creation:** Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- ❖ **Index entries:** <search key value, page id>; they 'direct' search for *data entries*, which are in leaf pages.
- ❖ **Search:** Start at root; use key comparisons to go to leaf. Cost $\propto \log_F N$; $F = \# \text{ entries/index pg}$, $N = \# \text{ leaf pgs}$
- ❖ **Insert:** Find leaf data entry belongs to, and put it there.
- ❖ **Delete:** Find and remove from leaf; if empty overflow page, de-allocate.

☛ **Static tree structure:** inserts/deletes affect only leaf pages.

13

After Inserting 23*, 48*, 41*, 42* ...

14

... Then Deleting 42*, 51*, 97*

☛ Note that 51 appears in index levels, but not in leaf!

15

B+ Tree: The Most Widely-Used Index

- ❖ Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. ($F = \text{fanout}$, $N = \# \text{ leaf pages}$)
- ❖ Minimum 50% occupancy (except for root). Each node contains $d \leq m \leq 2d$ entries. The parameter d is called the *order* of the tree.
- ❖ Supports equality and range-searches efficiently.

16

Example B+ Tree

- ❖ Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- ❖ Search for 5*, 15*, all data entries $\geq 24^*$...

☛ Based on the search for 15*, we know it is not in the tree!

17

B+ Trees in Practice

- ❖ Typical order: 100. Typical fill-factor: 67%.
 - average fanout = 133
- ❖ Typical capacities:
 - Height 4: $133^4 = 312,900,700$ records
 - Height 3: $133^3 = 2,352,637$ records
- ❖ Can often hold top levels in buffer pool:
 - Level 1 = 1 page = 8 Kbytes
 - Level 2 = 133 pages = 1 Mbyte
 - Level 3 = 17,689 pages = 133 Mbytes

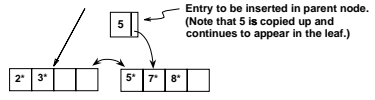
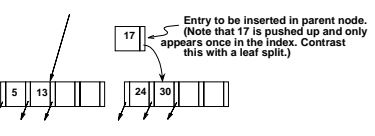
18

Inserting a Data Entry into a B+ Tree

- ❖ Find correct leaf L .
- ❖ Put data entry onto L .
 - If L has enough space, *done!*
 - Else, must *split* L (into L and a new node $L2$)
 - ♦ Redistribute entries evenly, **copy up** middle key.
 - ♦ Insert index entry pointing to $L2$ into parent of L .
- ❖ This can happen recursively
 - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- ❖ Splits “grow” tree; root split increases height.
 - Tree growth: gets *wider* or *one level taller at top*.

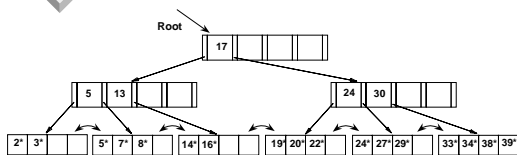
19

Inserting 8^* into Example B+ Tree

- ❖ Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
 
- ❖ Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.
 

20

Example B+ Tree After Inserting 8^*



- ❖ Notice that root was split, leading to increase in height.
- ❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

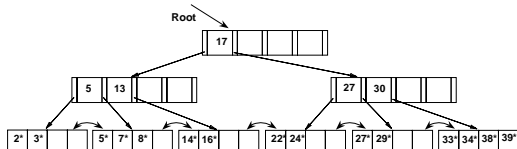
21

Deleting a Data Entry from a B+ Tree

- ❖ Start at root, find leaf L where entry belongs.
- ❖ Remove the entry.
 - If L is at least half-full, *done!*
 - If L has only $d-1$ entries,
 - ♦ Try to re-distribute, borrowing from *sibling* (adjacent node with same parent as L).
 - ♦ If re-distribution fails, *merge* L and sibling.
- ❖ If merge occurred, must delete entry (pointing to L or sibling) from parent of L .
- ❖ Merge could propagate to root, decreasing height.

22

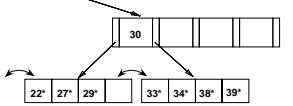
Example Tree After (Inserting 8^* , Then) Deleting 19^* and 20^* ...



- ❖ Deleting 19^* is easy.
- ❖ Deleting 20^* is done with re-distribution. Notice how middle key is *copied up*.

23

... And Then Deleting 24^*

- ❖ Must merge.
 - ❖ Observe ‘toss’ of index entry (on right), and ‘pull down’ of index entry (below).
 
-

24

Summary

- ❖ Indexes support efficient retrieval of records based on the values in some fields.
- ❖ Index is a collection of data entries plus a way to quickly find entries with given key values.
- ❖ Can have several indexes on a given file of data records, each with a different search key.
- ❖ Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.

25

Summary

- ❖ Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ❖ ISAM is a static structure.
 - Performance can degrade over time.
- ❖ B+ tree is a dynamic structure.
 - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
 - High fanout (**F**) means depth rarely more than 3 or 4.
 - Almost always better than maintaining a sorted file.

26

Summary (Contd.)

- Typically, 67% occupancy on average.
- Usually preferable to ISAM, modulo *locking* considerations; adjusts to growth gracefully.
- ❖ Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

27