

# Indexing: Part I

CPS 216  
Advanced Database Systems

## Announcements

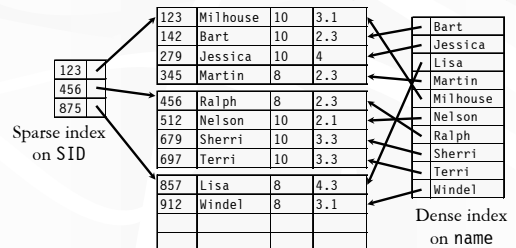
- ❖ Reading assignment
  - B<sup>+</sup>-tree tricks by Lomet
  - R-tree by Guttman
  - GiST by Hellerstein et al.
- ❖ Homework #1 due today (February 9)
- ❖ Homework #2 will be assigned Wednesday (February 11) and due in two weeks (February 26)
- ❖ No recitation session this Friday (February 14)
- ❖ Guest lecture next Monday (February 17)
  - Jennifer Widom on stream data processing
  - 4-5PM 130 North Building
  - No regular lecture on that day

## Basics

- ❖ Given a value, locate the record(s) with this value
  - SELECT \* FROM R WHERE A = value;
  - SELECT \* FROM R, S WHERE R.A = S.B;
- ❖ Other search criteria, e.g.
  - Range search
  - SELECT \* FROM R WHERE A > value;
  - Keyword search

## Dense and sparse indexes

- ❖ Dense: one index entry for each search key value
- ❖ Sparse: one index entry for each block
  - Records must be clustered according to the search key



## Dense versus sparse indexes

- ❖ Index size
  - Sparse index is smaller
- ❖ Requirement on records
  - Records must be clustered for sparse index
- ❖ Lookup
  - Sparse index is smaller and may fit in memory
  - Dense index can directly tell if a record exists
- ❖ Update
  - Easier for sparse index

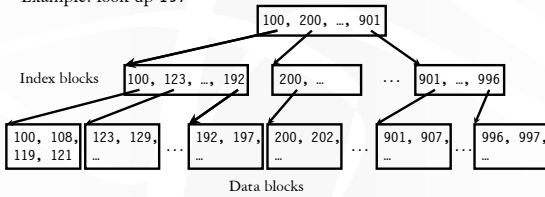
## Primary and secondary indexes

- ❖ Primary index
  - Created for the primary key of a table
  - Records are usually clustered according to the primary key
  - Can be sparse
- ❖ Secondary index
  - Usually dense
- ❖ SQL
  - PRIMARY KEY declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index
  - Secondary index can be created on non-key attribute(s)  
CREATE INDEX StudentGPAIndex ON Student(GPA);

# ISAM

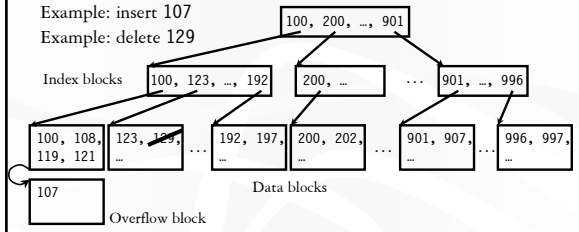
- ❖ What if an index is still too big?
  - Put a another (sparse) index on top of that!
  - ☞ ISAM (Index Sequential Access Method), more or less

Example: look up 197



# Updates with ISAM

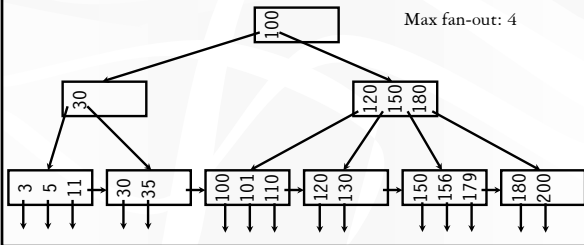
Example: insert 107  
Example: delete 129



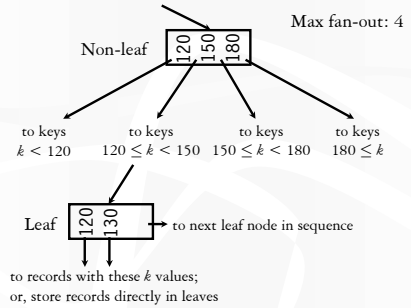
- ❖ Overflow chains and empty data blocks degrade performance
  - Worst case: most records go into one long chain

# B<sup>+</sup>-tree

- ❖ Balanced (more or less): good performance guarantee
- ❖ Disk-based: one node per block; large fan-out



# Sample B<sup>+</sup>-tree nodes



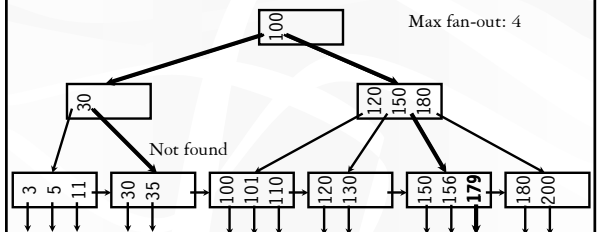
# B<sup>+</sup>-tree balancing properties

- ❖ All leaves at the same lowest level
- ❖ All nodes at least half full (except root)

	Max # pointers	Max # keys	Min # active pointers	Min # keys
Non-leaf	$f$	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil - 1$
Root	$f$	$f-1$	2	1
Leaf	$f$	$f-1$	$\lceil f/2 \rceil$	$\lceil f/2 \rceil$

# Lookups

SELECT \* FROM R WHERE  $k = 179$ ;  
SELECT \* FROM R WHERE  $k = 32$ ;



13

### Range query

SELECT \* FROM R WHERE  $k > 32$  AND  $k < 179$ ;

Max fan-out: 4

Look up 32...

And follow next-leaf pointers

14

### Insertion

❖ Insert a record with search key value 32

Max fan-out: 4

Look up where the inserted key should go...

And insert it right there

15

### Another insertion example

❖ Insert a record with search key value 152

Max fan-out: 4

Oops, node is already full!

16

### Node splitting

Max fan-out: 4

Yikes, this node is also already full!

17

### More node splitting

Max fan-out: 4

❖ In the worst case, node splitting can "propagate" all the way up to the root of the tree (not illustrated here)

- Splitting the root causes the tree to grow "up" by one level

18

### Deletion

❖ Delete a record with search key value 130

Max fan-out: 4

Look up the key to be deleted...

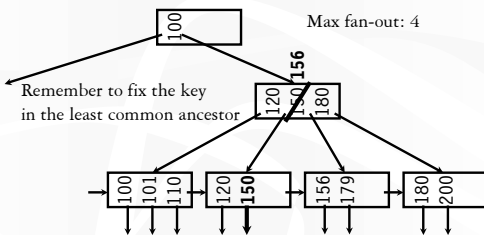
And delete it

Oops, node is too empty!

If a sibling has more than enough keys, steal one!

## Stealing from a sibling

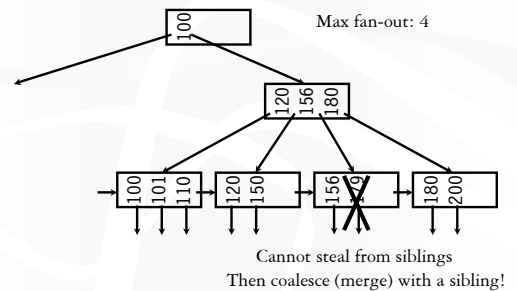
19



## Another deletion example

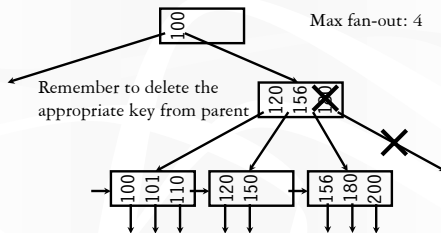
20

- ❖ Delete a record with search key value 179



## Coalescing

21



- ❖ Deletion can “propagate” all the way up to the root of the tree (not illustrated here)
  - When the root becomes empty, the tree “shrinks” by one level

## Performance analysis

22

- ❖ How many I/O's are required for each operation?
  - $b$  (more or less), where  $b$  is the height of the tree
  - Plus one or two to manipulate actual records
  - Plus  $O(b)$  for reorganization (should be very rare if  $f$  is large)
  - Minus one if we cache the root in memory
- ❖ How big is  $b$ ?
  - Roughly  $\log_{\text{fan-out}} N$ , where  $N$  is the number of records
  - B<sup>+</sup>-tree properties guarantee that fan-out is least  $f/2$  for all non-root nodes
  - Fan-out is typically large (in hundreds)—many keys and pointers can fit into one block
  - A 4-level B<sup>+</sup>-tree is enough for typical tables

## B<sup>+</sup>-tree in practice

23

- ❖ Complex reorganization for deletion often is not implemented (e.g., Oracle, Informix)
- ❖ Most commercial DBMS use B<sup>+</sup>-tree instead of hashing-based indexes because B<sup>+</sup>-tree handles range queries

## The Halloween Problem

24

- ❖ Story from the early days of System R...
 

```
UPDATE Payroll
SET salary = salary * 1.1
WHERE salary >= 100000;
```

  - There is a B<sup>+</sup>-tree index on *Payroll(salary)*
  - The update never stopped (why?)
- ❖ Solutions?
  - Scan index in reverse
  - Before update, scan index to create a complete “to-do” list
  - During update, maintain a “done” list
  - Tag every row with transaction/statement id

## Building a B<sup>+</sup>-tree from scratch

25

- ❖ Naïve approach
  - Start with an empty B<sup>+</sup>-tree
  - Process each record as a B<sup>+</sup>-tree insertion
- ❖ Problem
  - Every record require  $O(b)$  random I/O's

## Bulk-loading a B<sup>+</sup>-tree

26

- ❖ Sort all records (or record pointers) by search key
  - Just a few passes (assuming a big enough memory)
  - More sequential I/O's
- ☞ Now we already have all leaf nodes!
- ❖ Insert each leaf node in order
  - No need to look for the proper place to insert
  - Only the rightmost path is affected; keep it in memory



## Other B<sup>+</sup>-tree tricks

27

- ❖ Compressing keys
  - Head compression: factor out common key prefix and store it only once within an index node
  - Tail compression: choose the shortest possible key value during a split
  - In general, any order-preserving key compression
- ☞ Why does key compression help?
- ❖ Improving binary search within an index node
  - Cache-aware organization
  - Micro-indexing
- ❖ Using B<sup>+</sup>-tree to solve the phantom problem (later)

## B<sup>+</sup>-tree versus ISAM

28

- ❖ ISAM is more static; B<sup>+</sup>-tree is more dynamic
- ❖ ISAM is more compact (at least initially)
  - Fewer levels and I/O's than B<sup>+</sup>-tree
- ❖ Overtime, ISAM may not be balanced
  - Cannot provide guaranteed performance as B<sup>+</sup>-tree does

## B<sup>+</sup>-tree versus B-tree

29

- ❖ B-tree: why not store records (or record pointers) in non-leaf nodes?
  - These records can be accessed with fewer I/O's
- ❖ Problems?
  - Storing more data in a node decreases fan-out and increases  $b$
  - Records in leaves require more I/O's to access
  - Vast majority of the records live in leaves!

## Coming up next

30

- ❖ Other tree-based indexes: R-trees and variants, GiST
- ❖ Hashing-based indexes: extensible hashing, linear hashing, etc.
- ❖ Text indexes: inverted-list index, suffix arrays