

# Indexing

CPS 116  
Introduction to Database Systems

- ## Announcements (November 8)
- ❖ Project milestone #2 due today!
  - ❖ Homework #3 sample solution available
  - ❖ Homework #4 (last) to be assigned in 1½ weeks

- ## 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
 

database indexing

## 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

Sparse index on SID

123	
456	
857	

123	Milhouse	10	3.1
142	Bart	10	2.3
279	Jessica	10	4
345	Martin	8	2.3
456	Ralph	8	2.3
512	Nelson	10	2.1
679	Sherrri	10	3.3
697	Terri	10	3.3
857	Lisa	8	4.3
912	Windel	8	3.1

Dense index on name

Bart
Jessica
Lisa
Martin
Milhouse
Nelson
Ralph
Sherrri
Terri
Windel

- ## 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
    - Additional 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

❖ A hierarchy of intervals

❖ Balanced (more or less): good performance guarantee

❖ Disk-based: one node per block; large fan-out

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

to keys  $100 \leq k$

Non-leaf [120, 150, 180] Max fan-out: 4

to keys  $100 \leq k < 120$  to keys  $120 \leq k < 150$  to keys  $150 \leq k < 180$  to keys  $180 \leq k$

Leaf [120, 130] to next leaf node in sequence

to records with these  $k$  values; or, store records directly in leaves

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

❖ Height constraint: all leaves at the same lowest level

❖ Fan-out constraint: 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	$f$	$f-1$	$\lfloor f/2 \rfloor$	$\lfloor f/2 \rfloor$

### Lookups

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

### Range query

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

Max fan-out: 4

Look up 32...

And follow next-leaf pointers

### 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

### Another insertion example

❖ Insert a record with search key value 152

Max fan-out: 4

Oops, node is already full!

### Node splitting

Max fan-out: 4

Yikes, this node is also already full!

### 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 introduces a new root of fan-out 2 and causes the tree to grow "up" by one level

### Deletion

❖ Delete a record with search key value 130

Max fan-out: 4

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

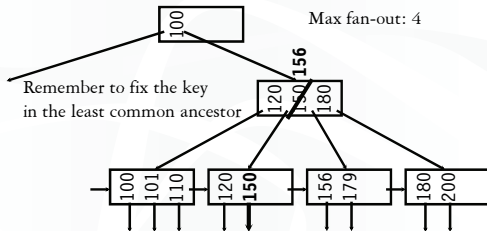
Look up the key to be deleted...

And delete it

Oops, node is too empty!

## Stealing from a sibling

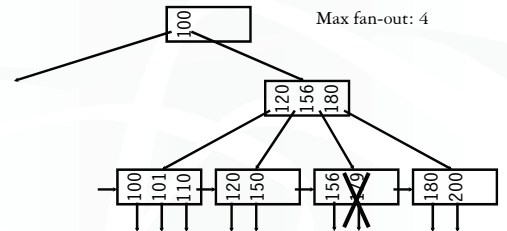
19



## Another deletion example

20

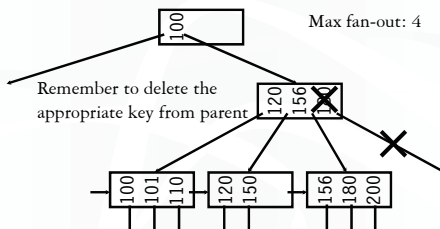
- ❖ Delete a record with search key value 179



Then coalesce (merge) with a sibling!

## 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?
  - $b$ , the height of the tree (more or less)
  - 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)
  - Leave nodes less than half full and periodically reorganize
- ❖ 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

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

25

- ❖ ISAM is more static; B<sup>+</sup>-tree is more dynamic
- ❖ ISAM can be 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

26

- ❖ 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!

## Beyond ISAM, B-, and B<sup>+</sup>-trees

27

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