Indexing

CPS 216
Advanced Database Systems

Outline

- The basics
- ISAM
- B^+-tree
- Next time
  - R-tree
  - Inverted lists
  - Hash indexes

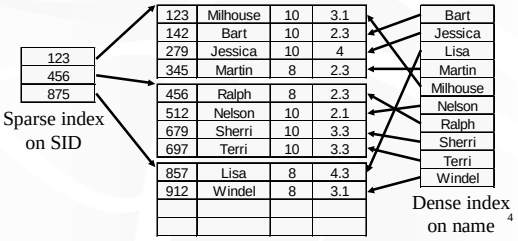
Indexing

- 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 [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 search key



Dense versus sparse indexes

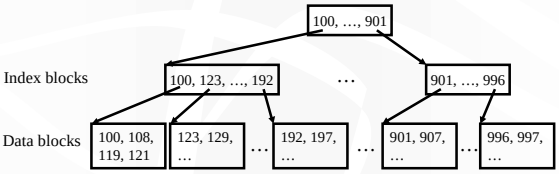
- Index size
  - Sparse index is smaller
- Requirement on records
  - Records must be clustered for sparse index
- Lookup
  - Sparse index is smaller and may fit in memory
  - Dense index can directly tell if a record exists
- Update
  - Easier for sparse index

Primary and secondary indexes

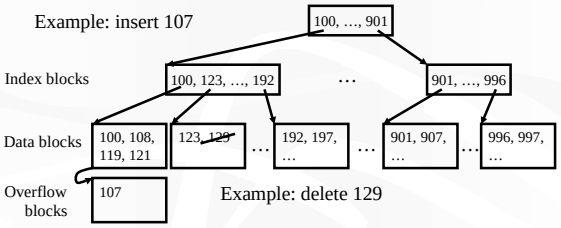
- Primary index
  - Created for the primary key of a table
  - Records are usually clustered according to the primary key
  - Can be sparse
- Secondary index
  - Usually dense
- SQL
  - PRIMARY KEY declaration automatically creates a primary index, UNIQUE key automatically creates a secondary index
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

ISAM

- What happens if you put a sparse index on top of another sparse index?
  - Indexed Sequential Access Method (more or less)



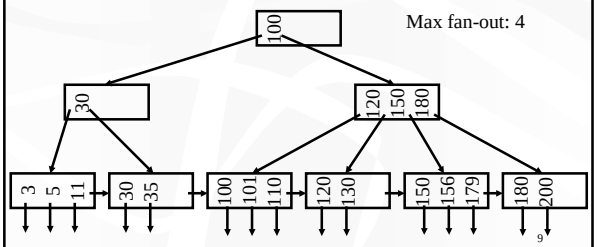
Updates with ISAM



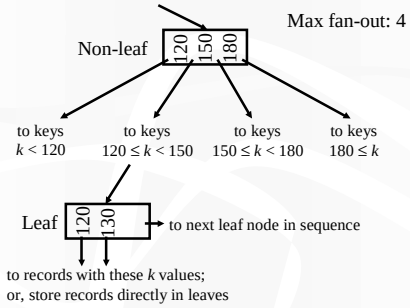
- Overflow and empty data blocks degrade performance

B$^+$-tree

- Balanced: good performance guarantee
- Disk-based: one node per block; large fan-out



Sample B$^+$-tree nodes



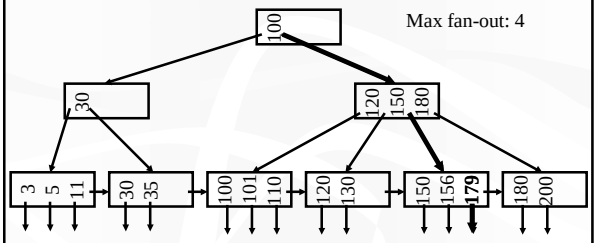
B$^+$-tree rules

- 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	$f-1$	ceil(f / 2)	ceil(f / 2) – 1
Root	f	$f-1$	2	1
Leaf	f	$f-1$	floor(f / 2)	floor(f / 2)

Query examples

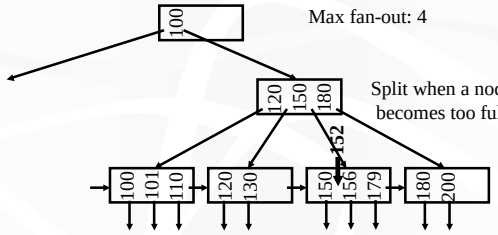
- Lookup: SELECT * FROM R WHERE k = 179;



- Range query: … WHERE k > 179 AND k < 200;
  - Find 179 and follow next-leaf pointers

An insertion example (slide 1)

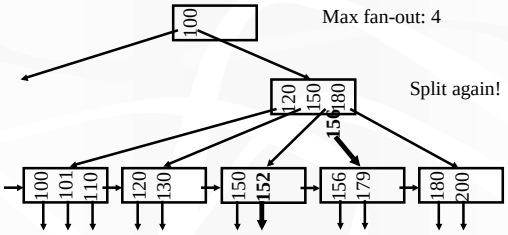
- Insert a record with search key value 152



13

An insertion example (slide 2)

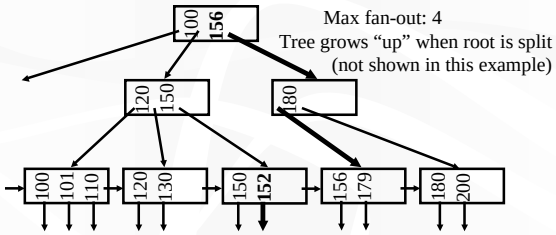
- Insert a record with search key value 152



14

An insertion example (slide 3)

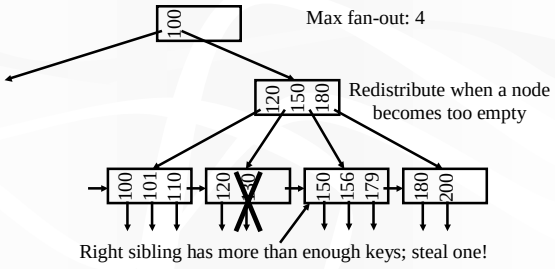
- Insert a record with search key value 152



15

A deletion example (slide 1)

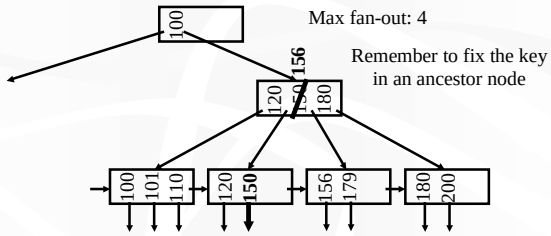
- Delete the record with search key value 130



16

A deletion example (slide 2)

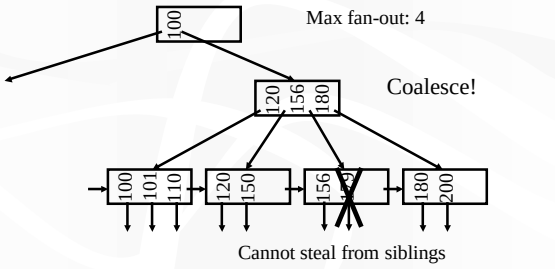
- Delete the record with search key value 130



17

Another deletion example (slide 1)

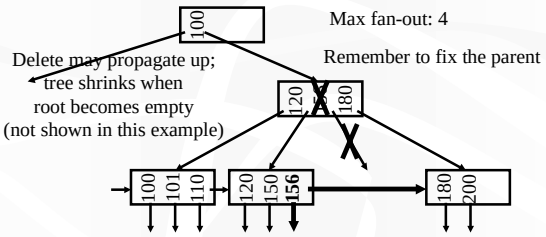
- Delete the record with search key value 179



18

Another deletion example (slide 2)

- Delete the record with search key value 179



19

Performance analysis

- How many I/Os are required for each operation?
  - *h* (more or less), where *h* is the height of the tree
  - Plus one or two to manipulate actual records
  - Plus *O*(*h*) for reorganization (very rare if *f* is large)
  - Minus one if we cache the root in memory
- How big is *h*?
  - Roughly $\log_{\text{fan-out}} n$, where *n* is the number of records
  - Fan-out is large (in hundreds)—many keys and pointers can fit into one block
  - A 4-level B$^+$-tree is enough for typical tables

20

B$^+$-tree in practice

- The index of choice in most commercial DBMS
- Complex reorganization for deletion often is not implemented (e.g., Oracle, Informix)
- Next
  - Bulk-loading
  - Concurrency control

21

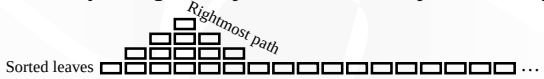
Building a B$^+$-tree from scratch

- Naïve approach
  - Start with an empty B$^+$-tree
  - Process each record as a B$^+$-tree insertion
- Problem
  - Every record require *O*(*h*) random I/Os

22

Bulk-loading a B$^+$-tree

- Sort all records (or record pointers) by search key
  - Just a few passes (assuming a big enough memory)
  - Can have more sequential I/Os
  - ➢Now we already have all the 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



23

Concurrency control for B$^+$-trees

- Naïve approach
  - Treat nodes as data objects; use 2PL
- Problem: low concurrency
  - Every read/write starts from the root—root becomes bottleneck for locking
  - That's the same as locking the entire table!

24

A simple B+-tree locking protocol

- A lookup transaction can release its lock on the parent once it gets a lock on the child
- An insert/delete transaction can do the same, provided that its modification cannot propagate up to the parent
- Never lock a node twice (even if its parent is locked all the time)
- ➢ More reading in Red Book: "Efficient Locking for Concurrent Operations on B-Trees"

25

Remember the phantom?

T1:

```
INSERT INTO Student
VALUES(512, "Nelson", 10, 2.1);
COMMIT;
```

T2:

```
SELECT * FROM Student
WHERE age = 10;

SELECT * FROM Student
WHERE age = 10;
COMMIT;
```

- T2 first locks all existing rows with age 10
- T1 inserts a new row with age 10
- T2 then sees the new row—phantom!

26

Predicate locking with B+-tree

- If there is a B+-tree on Student(age)
  - T2 will lock the B+-tree node containing age value 10
  - T1 has to wait for this lock to update the B+-tree
  - No more phantom!
- Predicate locking can be generalized to range predicates, e.g., age > 18 AND age < 20
  - Lock the B+-tree node (possibly non-leaf) containing this range

27

B+-tree versus ISAM

- ISAM is more static; B+-tree is more dynamic
- Performance
  - ISAM is more compact (at least initially)
    - Fewer levels and I/Os than B+-tree
  - Overtime, ISAM may not be balanced
    - Cannot provide guaranteed performance as B+-tree does
- Concurrency control
  - Much easier with ISAM
    - Because index blocks are never updated!

28

B+-tree versus B-tree

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

29