Lecture 21: Indexes

Monday, November 13, 2000

Outline

- Addresses (3.3)
- Index Structures (4.1, 4.2, 4.3)

Physical Addresses

- Each block and each record have a physical address that consists of:
  - The host
  - The disk
  - The cylinder number
  - The track number
  - The block within the track
  - For records: an offset in the block
    - sometimes this is in the block's header

Logical Addresses

- Logical address: a string of bytes (10-16)
- More flexible: can blocks/records around
- But need translation table:

Logical address	Physical address
L1	P1
L2	P2
L3	P3

Main Memory Address

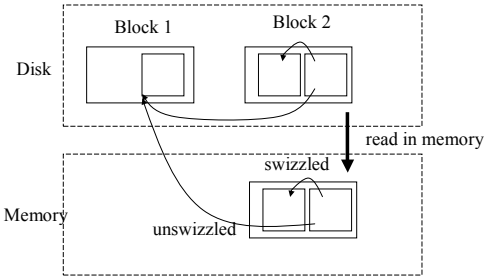
- When the block is read in main memory, it receives a main memory address
- Need another translation table

Memory address	Logical address
M1	L1
M2	L2
M3	L3

Optimization: Pointer Swizzling

- = the process of replacing a physical/logical pointer with a main memory pointer
- Still need translation table, but subsequent references are faster

Pointer Swizzling



Pointer Swizzling

- *Automatic*: when block is read in main memory, swizzle all pointers in the block
- *On demand*: swizzle only when user requests
- *No swizzling*: always use translation table

Pointer Swizzling

- When blocks return to disk: pointers need unswizzled
- Danger: someone else may point to this block
  - Pinned blocks: we don't allow it to return to disk
  - Keep a list of references to this block

Indexes

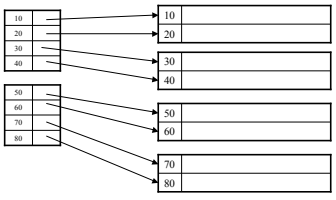
- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries with a given key value **k**.

Index Classification

- Primary/secondary
- Clustered/unclustered
- Dense/sparse
- B+ tree / Hash table / …

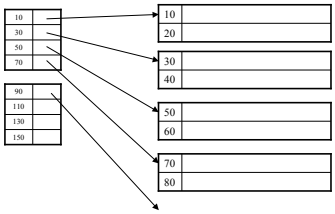
Primary Index

- File is sorted on the index attribute
- *Dense* index: sequence of (key,pointer) pairs



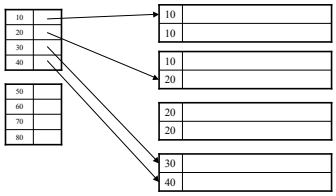
Primary Index

- *Sparse* index



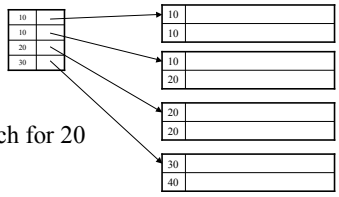
Primary Index with Duplicate Keys

- Dense index:



Primary Index with Duplicate Keys

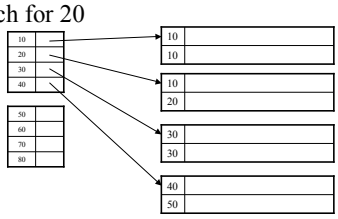
- Sparse index: pointer to lowest search key in each block:



- Search for 20

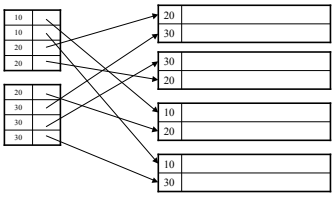
Primary Index with Duplicate Keys

- Better: pointer to lowest new search key in each block:
- Search for 20



Secondary Indexes

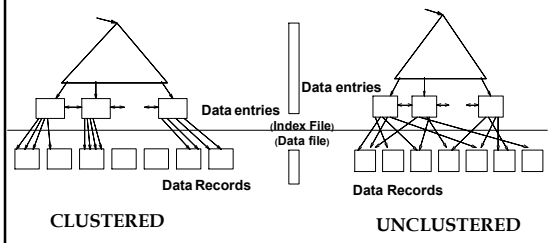
- To index other attributes than primary key
- Always dense (why ?)



Clustered/Unclustered

- Primary indexes = usually clustered
- Secondary indexes = usually unclustered

Clustered vs. Unclustered Index



Secondary Indexes

- Applications:
  - index other attributes than primary key
  - index unsorted files (heap files)
  - index clustered data

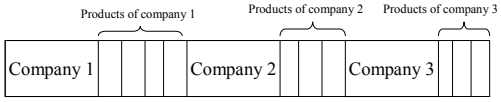
Applications of Secondary Indexes

- *Clustered data*

  Company(name, city), Product(pid, maker)

Select city
From Company, Product
Where name=maker
and pid="p045"

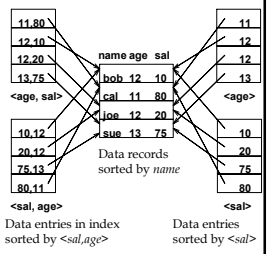
Select pid
From Company, Product
Where name=maker
and city="Seattle"



Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10

Examples of composite key indexes using lexicographic order.

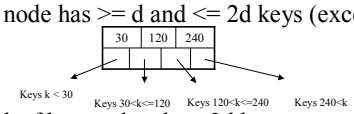


B+ Trees

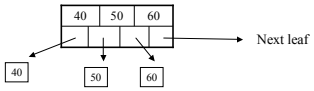
- Search trees
- Idea in B Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d
- Each node has >= d and <= 2d keys (except root)

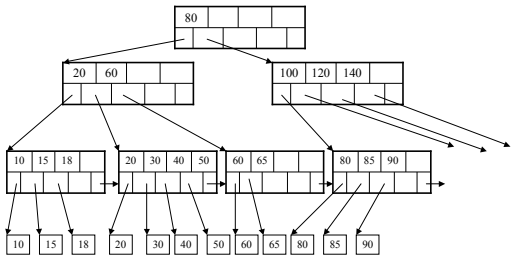


- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2



B+ Tree Design

- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
Select name
From people
Where age = 25
```

- Range queries:
  - As above
  - Then sequential traversal

```
Select name
From people
Where 20 <= age
  and  age <= 30
```

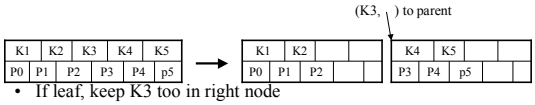
B+ Trees in Practice

- 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

Insertion in a B+ Tree

Insert (K, P)

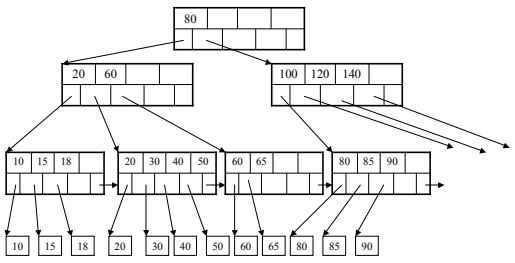
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



- If leaf, keep K3 too in right node
- When root splits, new root has 1 key only

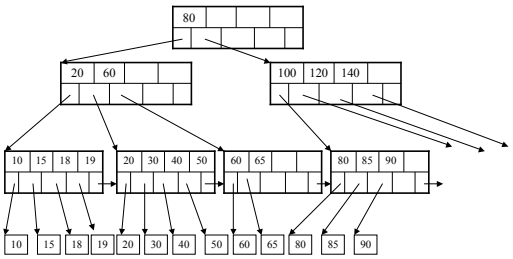
Insertion in a B+ Tree

Insert K=19



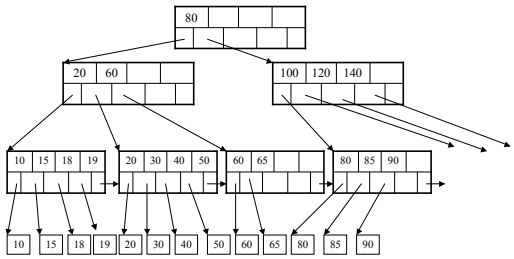
Insertion in a B+ Tree

After insertion



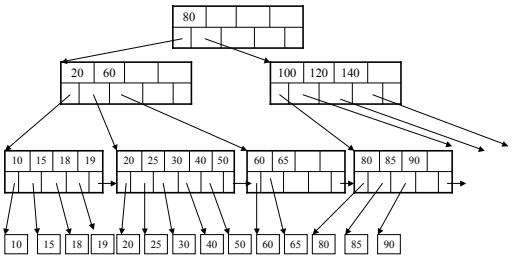
Insertion in a B+ Tree

Now insert 25



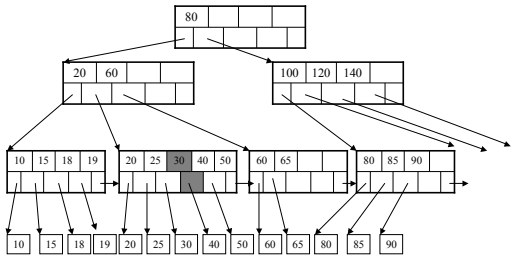
Insertion in a B+ Tree

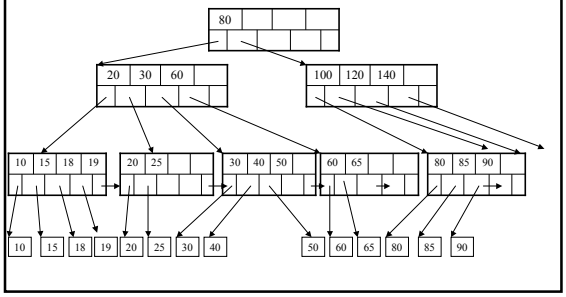
After insertion



Insertion in a B+ Tree

But now have to split !



Insertion in a B+ Tree

After the split