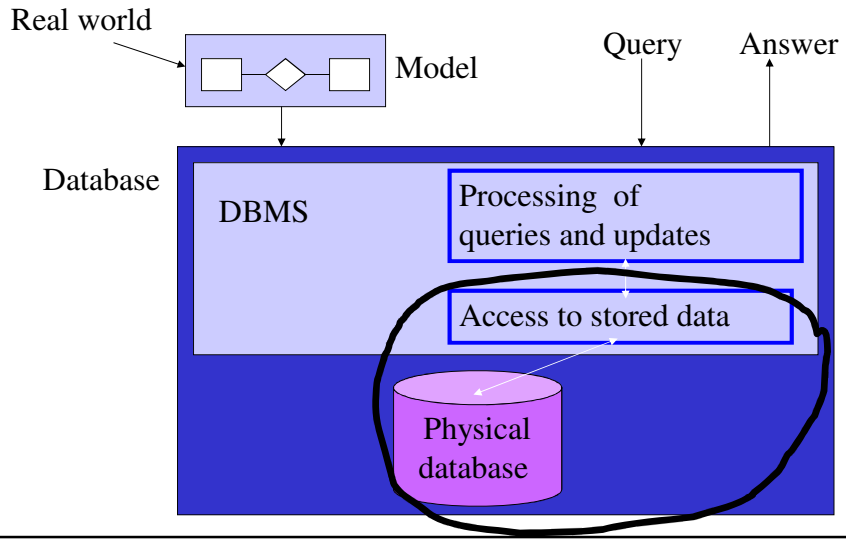
Lecture 8: Data structures for databases II

Jose M. Peña
jose.m.pena@liu.se

1

Database system



2

Indexes

- Previous lecture: File organization or **primary** access method (think in the chapters, sections, etc. of a book).
- This lecture: Indexes or **secondary** access method (think in the index of a book).
- Goal: To speed up the primary access method under **certain** query conditions.

3

Primary index

Primary access method: Binary search !

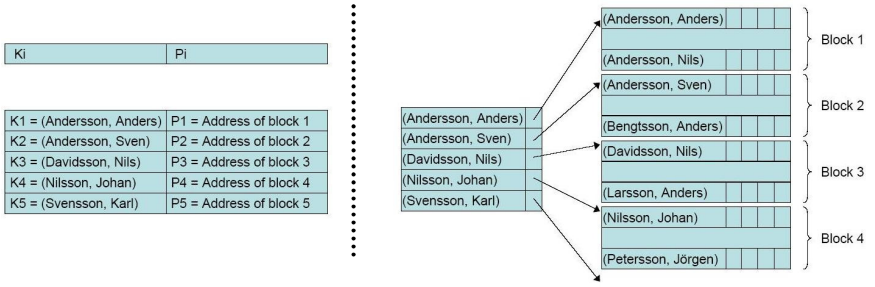
- Let us assume that the data file is **sorted**.
- Let us assume that the ordering field is a **key**.
- Primary index = **sorted file** whose records contain two fields:
  - One of the ordering key values.
  - A pointer to a disk block.

Primary access method: Binary search !

- There is one **index record** for each **data block**, and the record contains the ordering key value of the first record in the data block plus a pointer to that block.

4

Primary index



- Why is it faster to access a random record via a binary search in the index than in the data file ?
- What is the cost of maintaining an index ? If the order of the data records changes…

5

Clustering index

Primary access method: Binary search !

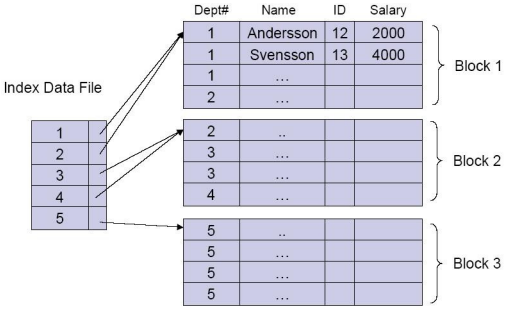
- Let us assume that the data file is **sorted**.
- Let us assume that the ordering field is a **non**-key.
- Clustering index = **ordered file** whose records contain two fields:
  - One of the ordering field values.
  - A pointer to a disk block.

Primary access method: Binary search !

- There is one **index record for each distinct value** of the ordering field, and the record contains the ordering field value plus a pointer to the **first data block** where that value appears.

6

Clustering index



- Efficiency gain ? Maintenance cost ?

7

Secondary index**es**

Primary access method: Linear search !

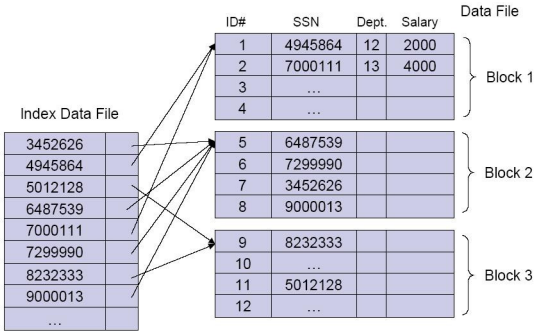
- Index on a **non**-ordering **key** field.
- The data file may be sorted or not.
- Secondary index = **ordered file** whose records contain two fields:

Primary access method: Binary search !

  - One of the non-ordering field values.
  - A pointer to a disk record or block.
- There is one **index record** per **data record**.

8

Secondary indexes

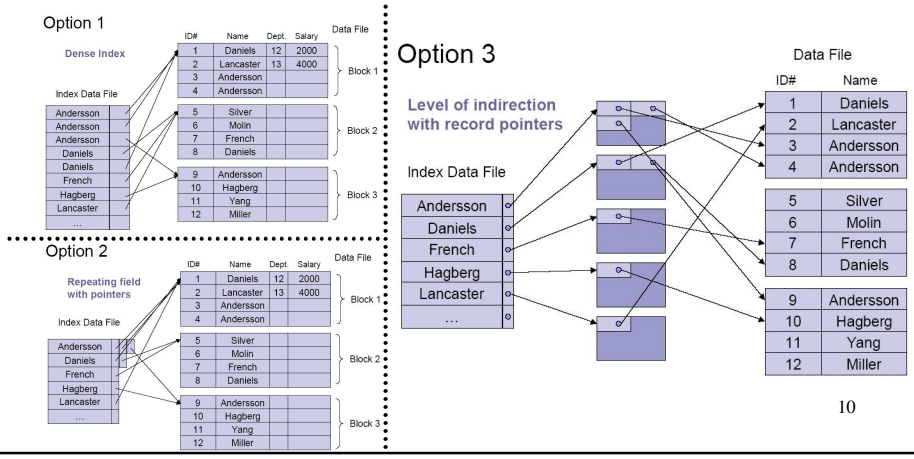


- Efficiency gain ? Maintenance cost ?
- Slower random access than a primary index on the same field, but higher relative gain on other fields. Check this claim.

9

Secondary indexes

- Index is on a **non**-ordering **non**-key field.



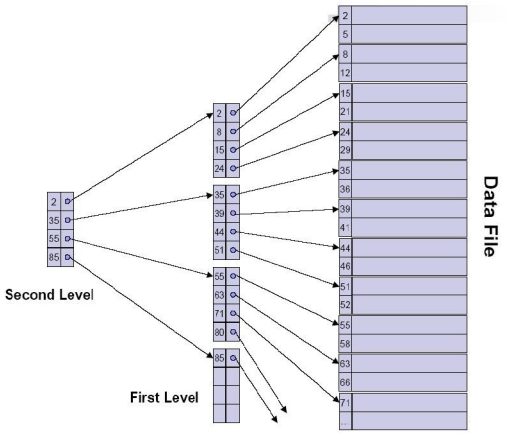
10

Multilevel indexes

- Index on index (first level, second level, etc.).
- Works for primary, clustering and secondary indexes as long as the first level index has a **distinct** index value for every entry.
- How many levels ? Until the last level fits in a **single** disk block.
- How many disk block accesses to retrieve a random record ? The number of index levels plus one.

11

Multilevel indexes



- Efficiency gain ? Maintenance cost ?

12

Exercise

- Assume an sorted file whose ordering field is a key. The file has 1000000 records of size 1000 bytes each. The disk block is of size 4096 bytes (unspanned allocation). The index record is of size 32 bytes.
- How many disk block accesses are needed to retrieve a random record when searching for the key field
  - Using no index ?
  - Using a primary index ?
  - Using a multilevel index ?

13

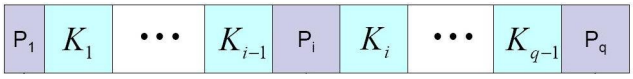
Dynamic multilevel indexes

- When using a (static) multilevel index, record insertion, deletion and update may be expensive operations, because all the index levels are **sorted** files.
- Solutions:
  - Overflow area + periodic reorganization.
  - Empty records + dynamic multilevel indexes, based on B-trees and B+-trees.

14

Search trees

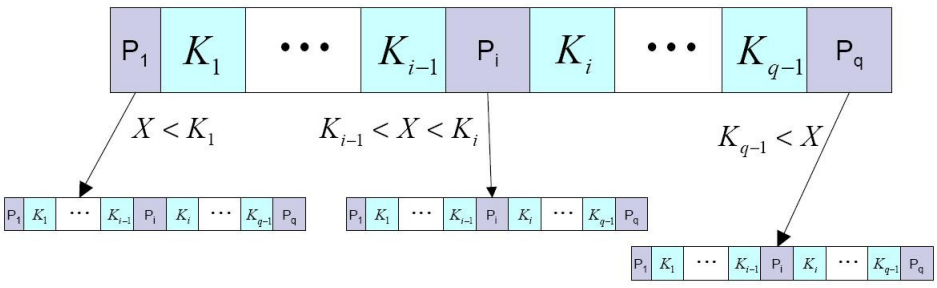
- Used to guide the search for a record.
- Generalization of binary search.
- The nodes of a search tree of **order p** look like



  - $q \leq p$
  - P_i is a **node pointer**.
  - K_i is a **key value**.

15

Search trees

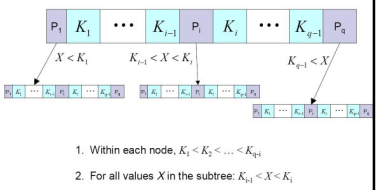


1. Within each node, $K_1 < K_2 < \ldots < K_{q-i}$
2. For all values X in the subtree: $K_{i-1} < X < K_i$

Note the cost of inserting, deleting and updating a record.

16

B-trees



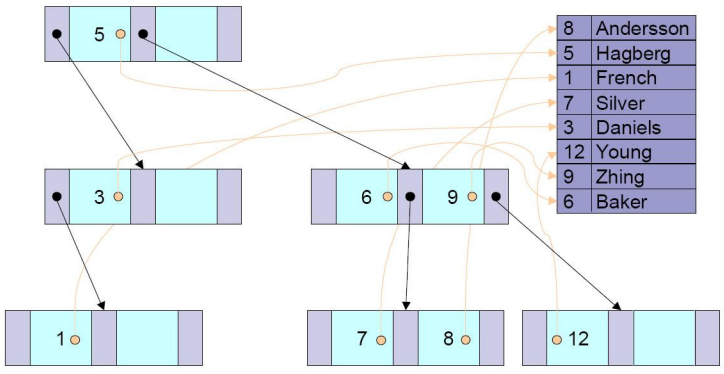
- B stands for **balanced**, i.e. all the leaves are at the same level. Why is this good ?
- A B-tree of order p is a balanced search tree, but each K_i has associated a pointer Pr_i to the **disk record** with key value K_i, in addition to the **node pointer** P_i .
- Moreover, each node in a B-tree (except the root and leaf nodes) has at least $\lceil \frac{p}{2} \rceil$ **node pointers**.

17

B-trees

order p=3

Order of insertion:
8, 5, 1, 7, 3, 12, 9, 6



The tree above is actually not a B-tree. Why ?
Note the cost of inserting, deleting and updating a record.

18

B-trees: Order

Thus, the node pointers are pointers to **disk blocks** !!

One **node** must fit in one block:

$$p \cdot P_{block} + (p-1) \cdot (P_{record} + K) \leq B \Rightarrow p \leq \frac{B + P_{record} + K}{P_{block} + P_{record} + K}$$

p	order, number of block pointer entries in a node
P_{block}	size of a block pointer
P_{record}	size of a record pointer
K	size of a search key field

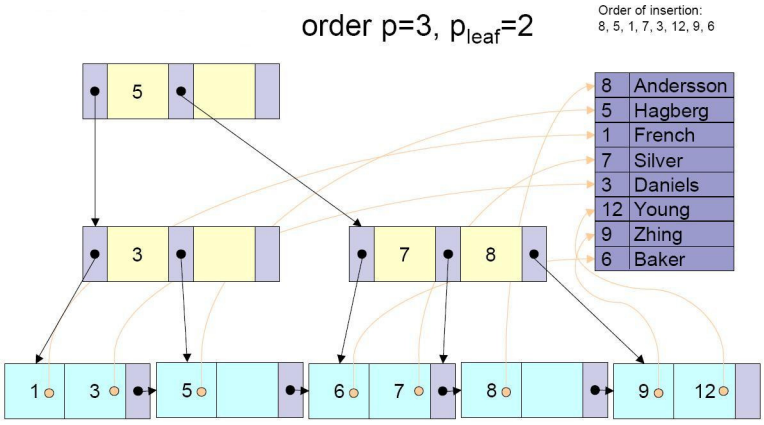
19

B+-trees

- Variation of B-trees. Most commonly used. Resembles very much a multilevel index.
- **Only the leaves** have pointers to disk records.
- The leaves contain an entry for **every** key value.
- The leaves are usually **linked** to provide ordered access.
- Of course, B+-trees are balanced.

20

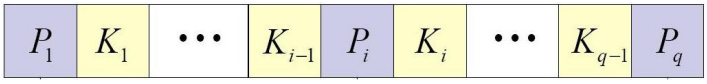
B+-trees



21

B+-trees: Internal nodes

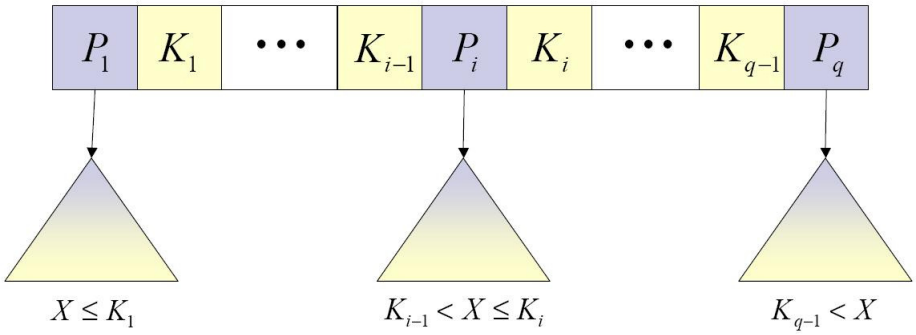
- The **internal** nodes of a B+-tree of order p look like



  - $q \leq p$
  - P_i is a **node pointer**.
  - K_i is a **key value**.
  - Every node (except the root) has **at least** $\lceil \frac{p}{2} \rceil$ node pointers.

22

B+-trees: Internal nodes



1. Within each node, $K_1 < K_2 < \ldots < K_{q-i}$
2. For all values X in the subtree: $K_{i-1} < X \le K_i$

23

B+-trees: Leaves

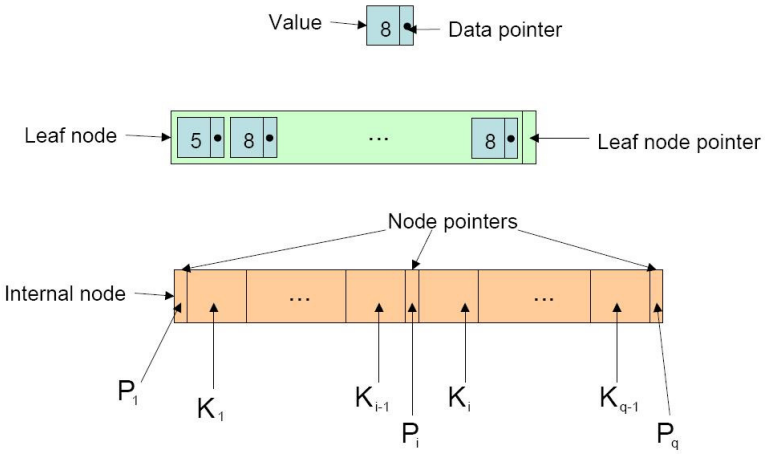
- The **leaves** of a B+-tree of order p look like

K_1	Pr_1	...	K_i	Pr_i	...	K_q	Pr_q	P

  - $q \le p$
  - Within the leaf, $K_1 < K_2 < \ldots < K_q$
  - Pr_i is a pointer to the **disk record** with key value K_i .
  - P is a pointer to the disk next leaf.
  - The leaf has **at least** $\lceil \frac{p}{2} \rceil$ key values.

24

B+-trees



25

Thus, the node pointers are pointers to **disk blocks** !!

B+-trees: Order

One **internal node** must fit in one block:

$$p \cdot P_{block} + (p-1) \cdot K \leq B \quad \Rightarrow p \leq \frac{B+K}{P_{block}+K}$$

One **leaf node** must fit in one block:

$$p_{leaf} \cdot (P_{record} + K) + P_{block} \leq B \Rightarrow p_{leaf} \leq \frac{B-P_{block}}{P_{record}+K}$$

p	order, number of pointer entries in an internal node
p_{leaf}	number of record pointer entries in a leaf node
P_{block}	size of a block pointer
K	size of a search key field
P_{record}	size of a record pointer

26

B+-trees: Retrieval

- Very fast retrieval of a random record. At worst, $\left\lceil \log_{\lceil \frac{p}{2} \rceil} N \right\rceil + 1$
  - p is the order of the internal nodes of the B+-tree.
  - N is the number of leaves in the B+-tree.
- How would the retrieval proceed ?
- Insertion and deletion can be expensive.

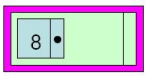
27

B+-trees: Insertion

Insert: 8

28

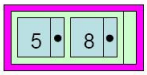
B+-trees: Insertion



Insert: 5

29

B+-trees: Insertion

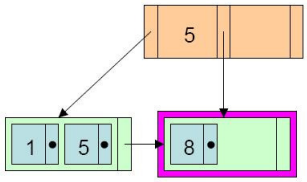


Overflow – create a new level

Insert: 1

30

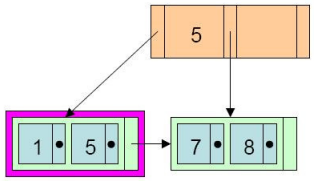
B+-trees: Insertion



Insert: 7

31

B+-trees: Insertion

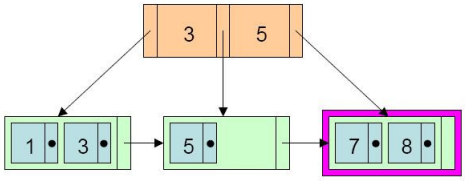


Overflow - Split

Insert: 3

32

B+-trees: Insertion

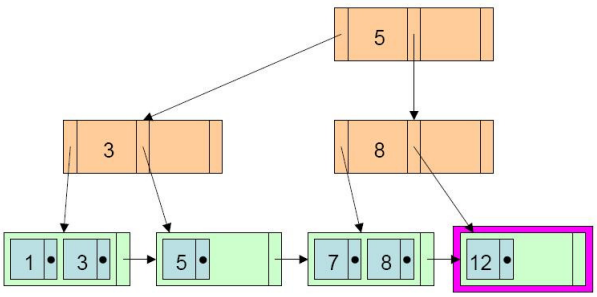


Overflow - Split
Propagates a new level

Insert: 12

33

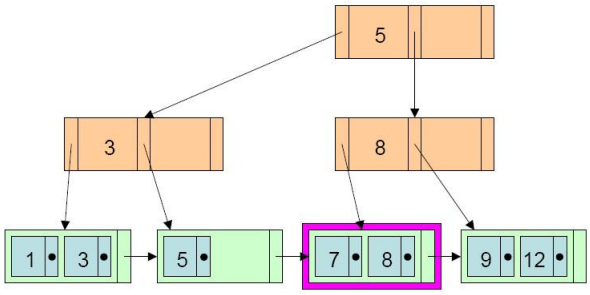
B+-trees: Insertion



Insert: 9

34

B+-trees: Insertion

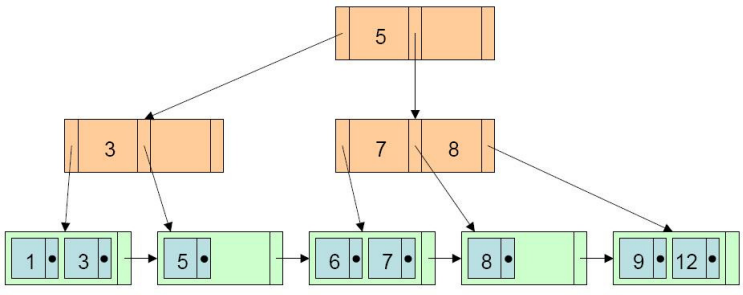


Overflow – Split, propagates

Insert: 6

35

B+-trees: Insertion



Resulting B+-tree

36

Exercise

- B=4096 bytes, P=16 bytes, K=64 bytes, node fill percentage=70 %.
- For both B-trees and B+-trees:
  - Compute the order p.
  - Compute the number of nodes, pointers and key values in the root, level 1, level 2 and leaves.
  - If the results are different for B-trees and B+-trees, explain why this is so.

37