course:

**Database Systems** (NDBI025)

SS2017/18

lecture 11:

# Implementation of database structures

doc. RNDr. Tomáš Skopal, Ph.D.
RNDr. Michal Kopecký, Ph.D.

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague

# Today's lecture outline

- disk management, paging, buffer manager
- database files organization
- indexing
  - single-attribute indexes
    - $B^+$-strom, bitmaps, hashing
  - multi-attribute indexes

# Introduction

- relations/tables stored in files on the disk
- need to organize table records within a file (efficient storage, update and access)

<u>Example:</u>
Employees(name char(20), age integer, salary integer)

# Paging

- records stored in disk pages of fixed size (a few kB)
- the reason is the hardware, still assuming magnetic disk based on rotational plates and reading heads
  - the data organization must be adjusted w.r.t. this mechanism
- the HW firmware can only access entire pages (I/O operations – reads, writes)
- realtime for I/O operations =
  = seek time + rotational delay + data transfer time
- sequential access to pages is much faster than random access (the seek time and rotational delay not needed)

Example: reading 4 KB could take 8 + 4 + 0,5 ms = 12,5 ms;
i.e., the reading itself takes only 0,5 ms = 4% of the realtime!!!

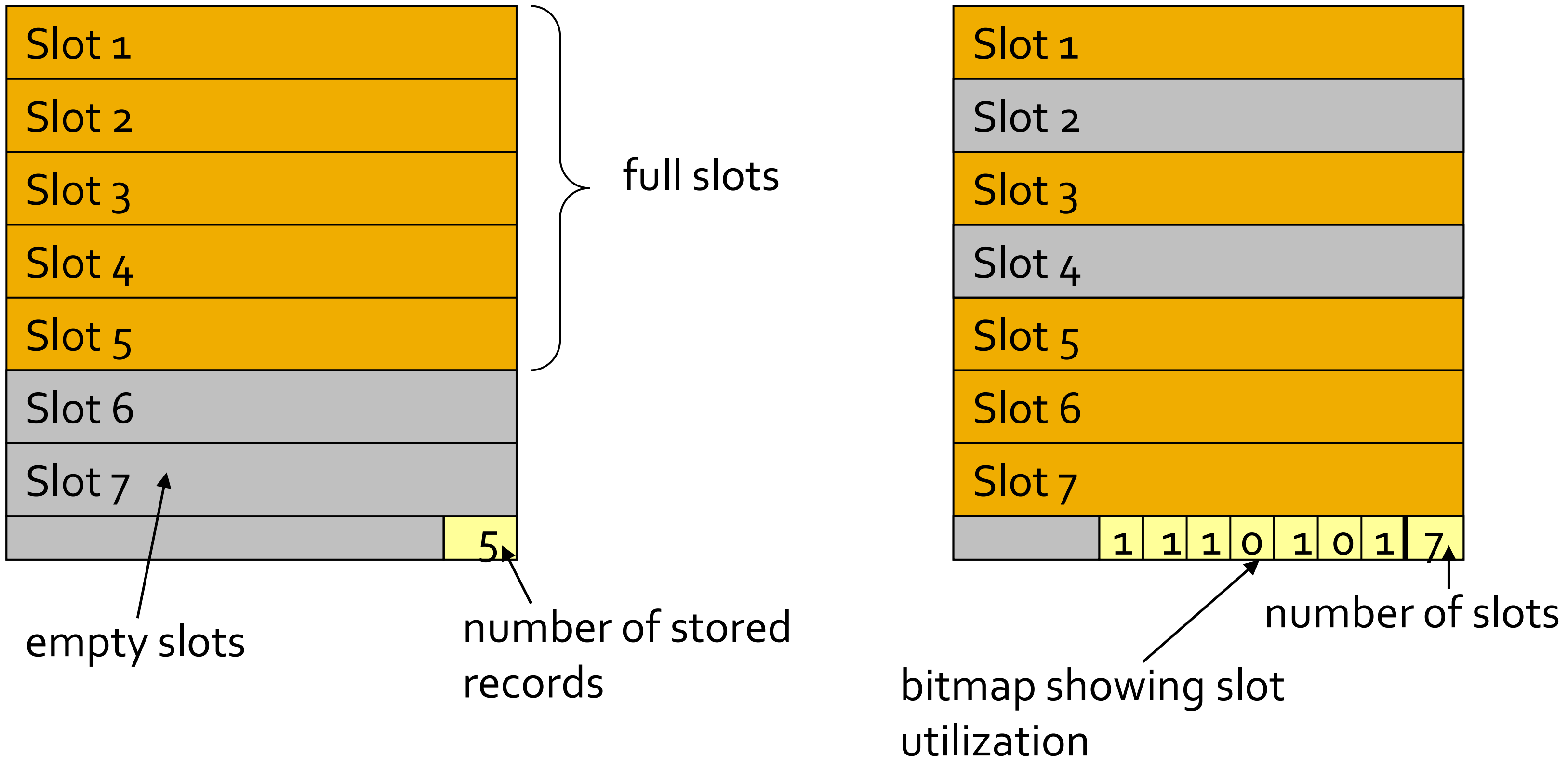
## Paging, cont.

- I/O is a unit of time cost
- the page is divided into *slots*, that are used to store records,
  - identified by *page id*
- a record can be stored
  - in multiple pages = better space utilization but need for more I/Os for record manipulation
  - in single page (assuming it fits) = part of page not used, less I/Os
  - ideally, records fit the entire page
- record identified by **rid** (record id) – tuple *page id* and *slot it*

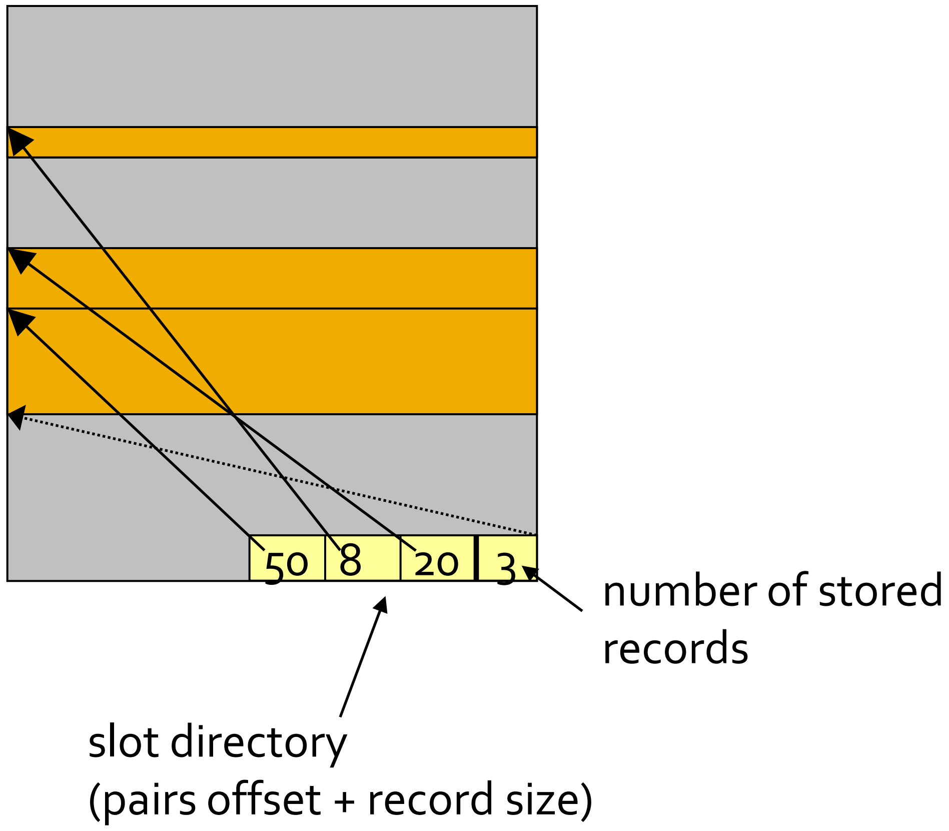
## Paging, cont.

- if only fixed-size data types are used in the record
  $\longrightarrow$ fixed record size
- if also variable-size data types are used in the record
  $\longrightarrow$ variable size of the records, e.g., types varchar(X), BLOB, ...
- fixed-size records = fixed-size slots
- variable-size records = need for slot directory in the page header

## Fixed-size page organization, example



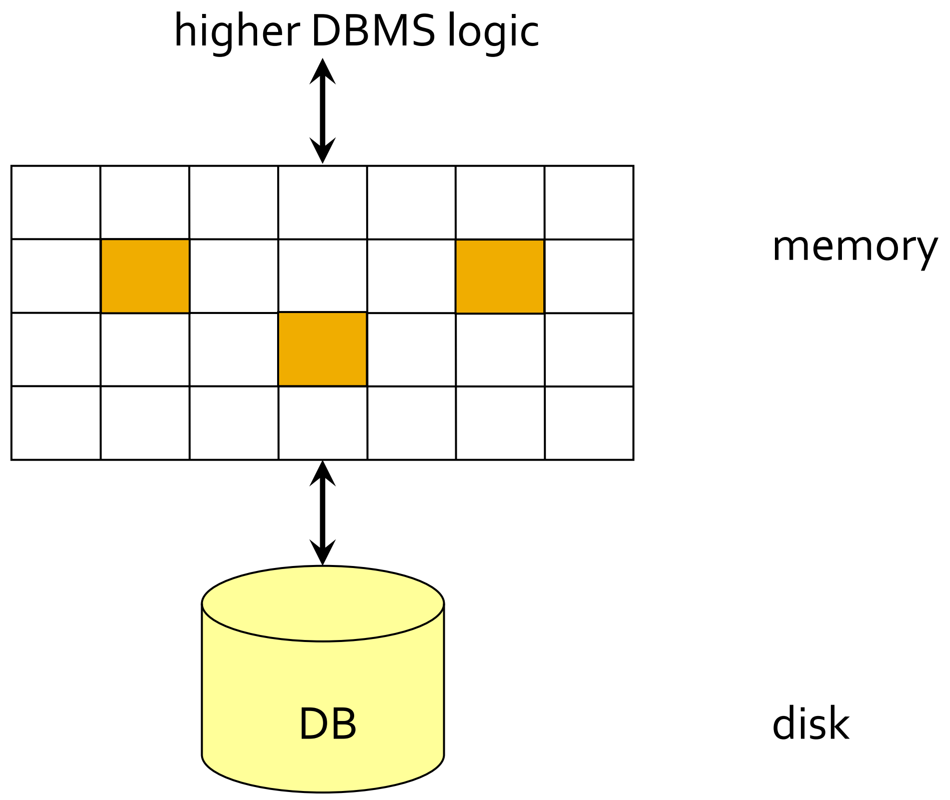
# Variable-size page organization, example



## Buffer management

- buffer = piece of main memory for temporary storage of disk pages, disk pages are mapped into memory frames 1:1
- every frame has 2 flags: **pin_count** (number of references to a page in frame) and **dirty** (modified record)
- serves for speeding repeated access to pages – buffer manager implements the **read** and **write** operations
- abstraction of the higher DBMS logic from disk management
- implementation of **read** retrieves the page from buffer, if it is not there, it is first fetched from the disk, increasing **pin_count**
- implementation of **write** puts the page into the buffer, setting **dirty**
- if the buffer is full (during read or write), some page must be replaced $\rightarrow$ various policies, e.g., LRU (least-recently-used),
  - if the replaced page is **dirty**, it must be stored

## Buffer management



## Database storage

- data files
  (consisting of table data)
- index files
- system catalogue – contains metadata
  - table schemas
  - index names
  - integrity constraints, keys, etc.

## Data files

- heap
- sorted file
- hashed file

Observing average I/O cost of simple operations:

1) sequential access to records
2) searching records based on equality (w.r.t search key)
3) searching records based on range (w.r.t search key)
4) record insertion
5) record deletion

Cost model:

N = number of pages, R = records per page

## Simple operations, SQL examples

- sequential reading of pages
  SELECT * FROM Employees
- searching on equality
  SELECT * FROM Employees  WHERE age = 40
- searching on range
  SELECT * FROM Employees
  WHERE salary > 10000 AND salary < 20000
- record insertion
  INSERT INTO Employees VALUES (...)
- record deletion based on **rid**
  DELETE FROM Employees WHERE rid = 1234
- record deletion
  DELETE FROM Employees WHERE salary < 5000

## Heap file

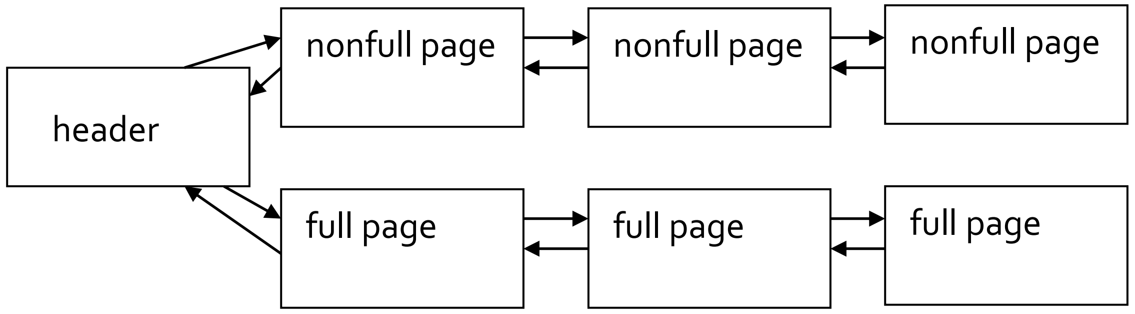
- stored records in pages are not ordered, resp., they are stored in the order of insertion
- page search can only be achieved by sequential scan (GetNext operation)
- quick record insertion (at the end of file)
- deletion problems → „holes“ (pieces of not utilized space)

## Maintenance of empty heap pages

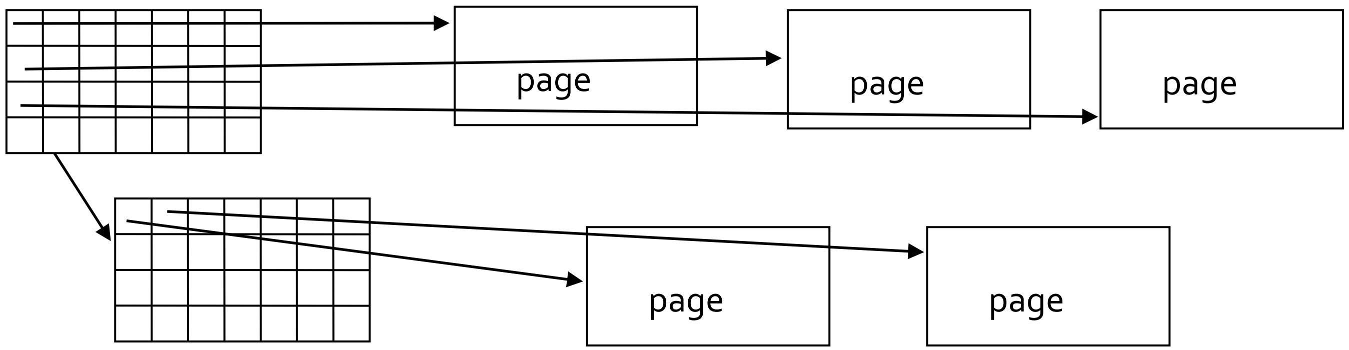
- double linked list
  - header + lists of full and nonfull pages
- page directory
  - linked list of directory pages
  - every item in the directory refers to a data page
  - flag bit of each item utilization

# Maintenance of empty heap pages

double linked list



page directory



## Heap, cost of simple operations

- sequential access = N
- search on equality = 0,5*N or N
- search on range = N
- record insertion = 1
- record deletion
  - 2, assuming **rid** based search costs 1 I/O,
  - N or 2*N,
    if deleted based on equality or range

# Sorted file

- records stored in pages based on an ordering according to a search key (single or multiple attributes)
- file pages maintained contiguous,
  i.e., no „holes“ with empty space
- allows fast search on equality and/or range
- slow insertion and deletion, „moving“ with the rest of pages
- a tradeoff used in practice
  - sorted file at the beginning
  - each page has an overhead space where to insert;
  - if the overhead space is full, update pages are used (linked list).
  - a reorganization needed from time to time, i.e., sorting

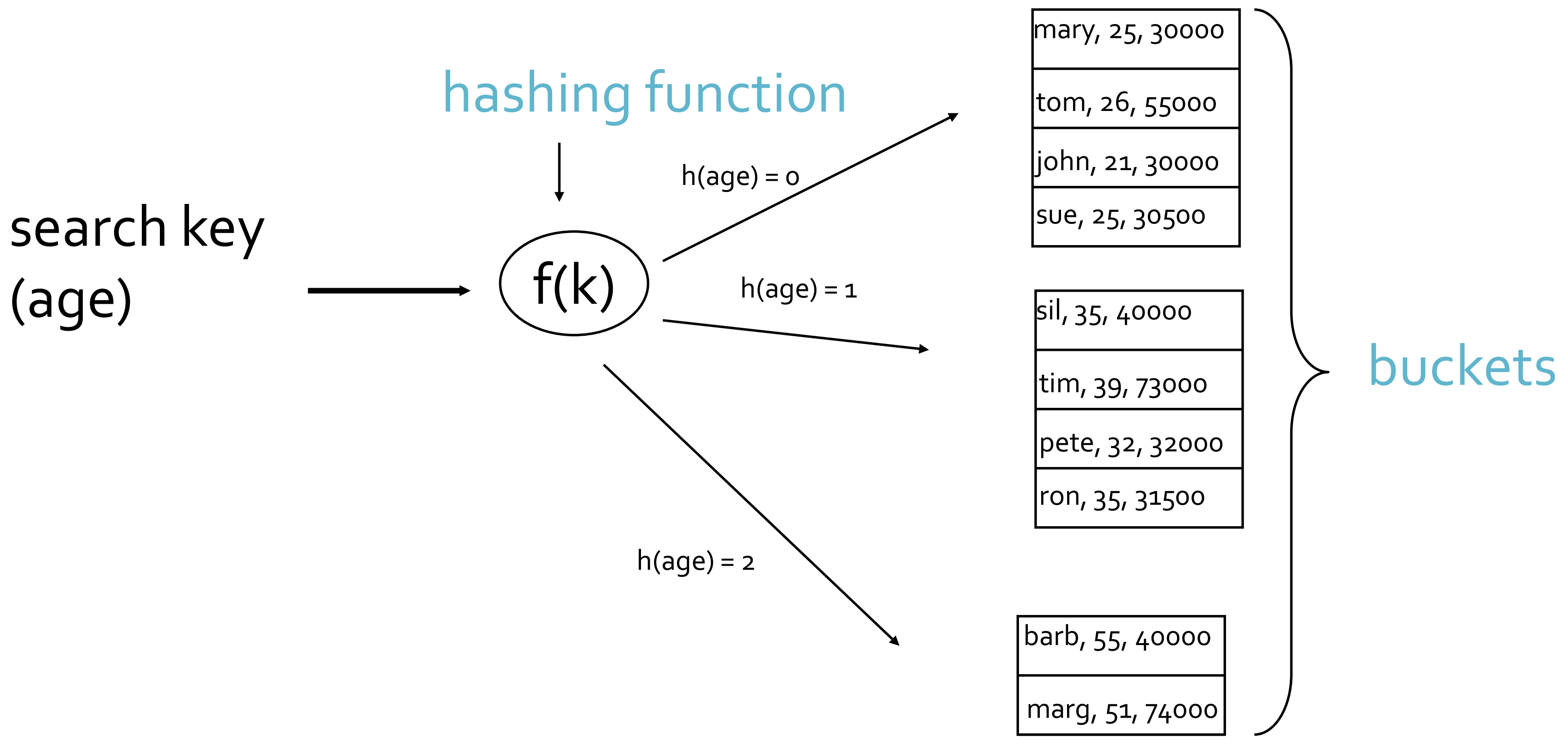
## Sorted file, cost of simple operations

- sequential access = N
- search on equality = $\log_2 N$ or N
- search on range = $\log_2 N + M$
  (where M is number of relevant pages)
- record insertion = N
- record deletion = $\log_2 N + N$ based on key, otherwise 1,5*N (**rid**)

## Hashed file

- organized in K buckets, a bucket is extensible to multiple disk pages
- record is inserted into/read from bucket determined by a hashing function and the search key
  - bucket id = f(key)
- if the bucket is full, new pages are allocated and linked to the bucket (linked list)
- fast queries and deletion on equality
- higher space overhead, problems with chained pages (solved by dynamic hashed techniques)

## Hashed file


hashing function
search key
(age)
f(k)
h(age) = 0
h(age) = 1
h(age) = 2
mary, 25, 30000
tom, 26, 55000
john, 21, 30000
sue, 25, 30500
sil, 35, 40000
tim, 39, 73000
pete, 32, 32000
ron, 35, 31500
barb, 55, 40000
marg, 51, 74000
buckets

## Hashed file, cost of simple operations

- sequential access = N
- search on equality = N/K (best case)
- search on range = N
- record insertion = N/K (best case)
- deletion on equality = N/K + 1 (best case), otherwise N

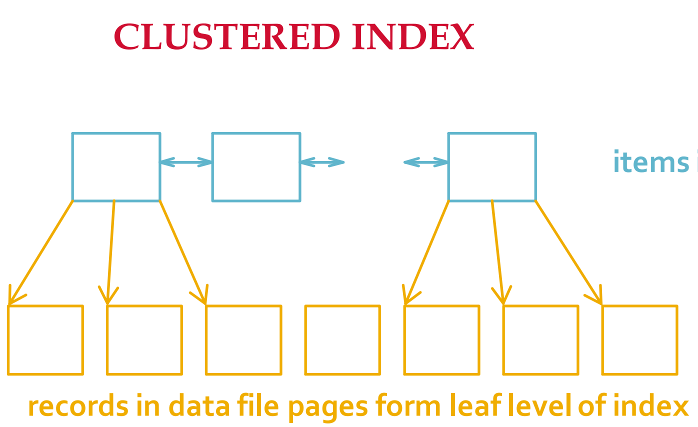
## Indexing

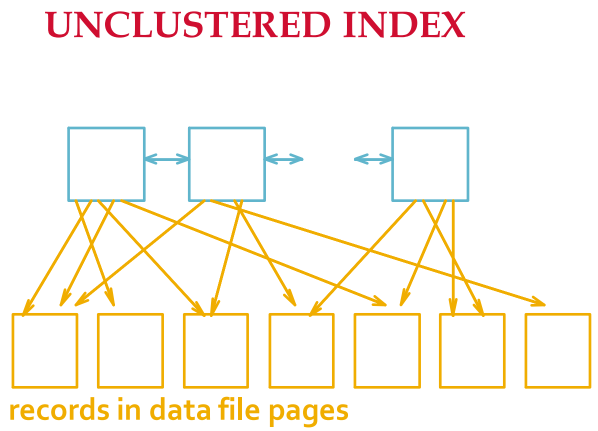
- index is a helper structure that provides fast search based on search key(s)
- organized into disk pages (like data files)
- usually different file than data files
- contains only search keys and links to the respective records (i.e., rid)
- need much less space than data files (e.g., 100x less)

# Indexing, principles

- index item could contain
  - the whole record (then index and data file are the same)
  - pair <key, rid>
  - pair <key, rid-list>, where rid-list is a list of links to records with the same search key value
- clustered vs. unclustered indexes
  - **clustered:** ordering of index items is (almost) the same as ordering in the data file, only tree-based indexes can be clustered + indexes containing the entire records (also hashed index)
    - primary key = search key used in clustered index (sorted/hashed data file)
  - **unclustered:** the order of search keys is not preserved

# Indexing, principles





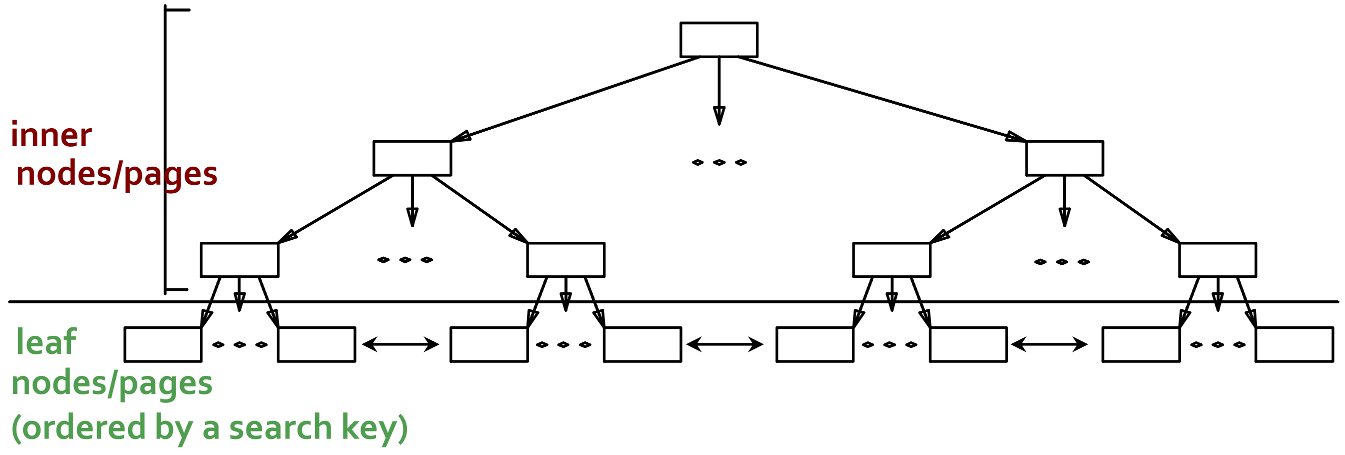
The pros of clustered index are huge speedup when searching on range, because the result record pages are read sequentially from data file. Unclustered index only allows read sequentially from index file, not from data file.

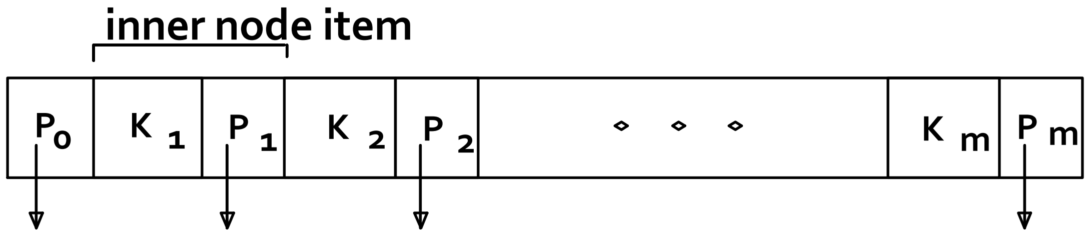
Cons: large overhead for keeping the data file sorted (even more expensive when applied to other indexes)

## $B^+$-tree

- follows B-tree - paged, balanced tree-based index (Rudolf Bayer, 1972).
- provides logarithmic complexity for insertion, search on equality (no duplicates), deletion on equality (no duplicates)
- guarantees 50% node (page) utilization
- $B^+$-tree extends B-tree by
  - linking leaf pages for efficient range queries
  - inner nodes contain indexed intervals, i.e., all keys are in the leaves

# B$^+$-tree, schema





Demo: [http://slady.net/java/bt/](http://slady.net/java/bt/)

## Hashed index

- similar to hashed data file
  - i.e., buckets and hashing function used
- only key values in the buckets together with the **rids**
- same pros/cons

# Bitmaps

- suitable for indexing attributes of low-cardinality data types
  - suitable for, e.g., attribute **FAMILY_STATUS** = {single, married, divorced, widow}
  - not suitable for, e.g., attribute **PRODUCT_PRICE** (many values), better B-tree
- for each value ***h*** of an indexed attribute ***a*** a bitmap (binary vector) is constructed, where 1 on ith position means the value ***h*** appears in the ith record (in the attribute atributu ***a***), while it holds
  - bitwise OR of all bitmap for an attribute forms only 1s (every attribute has a value)
  - bitwise AND of any two bitmaps for an attribute forms only zeros (attribute values are deterministic)

| Name | Address | Family status |
|---|---|---|
| John Smith | London | single |
| Rostislav Drobil | Prague | married |
| Franz Neumann | Munich | married |
| Fero Lakatoš | Malacky | single |
| Sergey Prokofjev | Moscow | divorced |

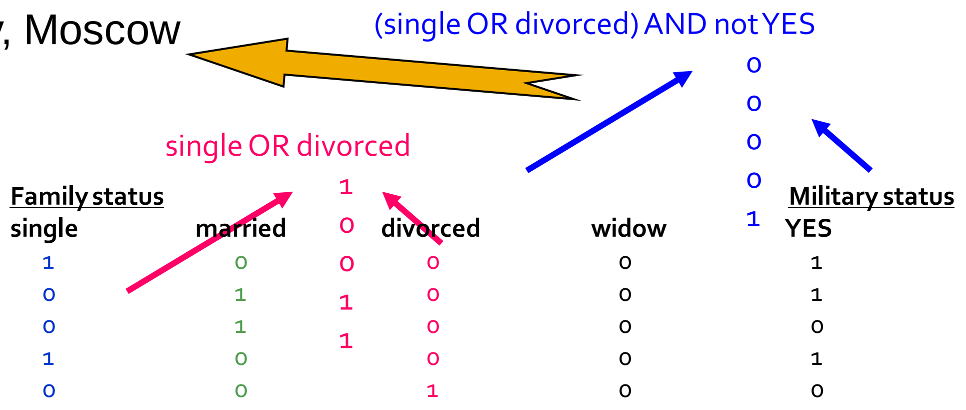
| single | married | divorced | widow |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |

# Bitmaps

- query evaluation
  - bitwise operations with attribute bitmaps
  - resulting bitmap marks the queried records
- example
  - **Which single or divorced people did not complete the military service?**
    (bitmap(**single**) **OR** bitmap(**divorced**)) **AND not** bitmap(**YES**)

**answer:** Sergey Prokofjev, Moscow

| Name | Address | Military service | Family status |
|---|---|---|---|
| John Smith | London | YES | single |
| Rostislav Drobil | Prague | YES | married |
| Franz Neumann | Munich | NO | married |
| Fero Lakatoš | Malacky | YES | single |
| Sergey Prokofjev | Moscow | NO | divorced |



## Bitmaps

- pros
  - efficient storage, could be also compressed
  - fast query processing, bitwise operations are fast
  - easy parallelization
- cons
  - suitable only for attributes with small cardinality domain
  - range queries get slow the linearly with the number of values in the range (bitmaps for all the values must be processed)

# Multi-attribute indexing, examples

- consider conjunctive range query
  SELECT * FROM Employees WHERE
  salary BETWEEN 10000 AND 30000 AND
  age < 40 AND
  name BETWEEN 'Dvořák' AND 'Procházka'
- simple solutions using B$^+$-tree
  (number of indexed attributes M = 3)
  1) M standalone indexes
  2) one index of M concatenated attributes
  - both solutions bad (the second is slightly better)

# Multi-attribute indexing, examples

Three standalone indexes:

...WHERE **salary BETWEEN 10000 AND 30000** AND **age < 40** AND **name BETWEEN 'Dvořák' AND 'Procházka'**

| | | | |
|---|---|---|---|
| r1 | Čech Jaroslav | 17000 | 27 |
| r2 | Dostál Jan | 21000 | 33 |
| r3 | Malý Zdeněk | 15000 | 45 |
| r4 | Mrázek František | 22000 | 37 |
| r5 | Novák Josef | 32000 | 25 |
| r6 | Novák Karel | 13000 | 19 |
| r7 | Oplustil Arnošt | 9000 | 36 |
| r8 | Papoušek Jindřich | 19000 | 50 |
| r9 | Richter Tomáš | 26000 | 41 |
| r10 | Zlámal Alois | 13000 | 52 |

{r3, r4, r5, r6, r7, r8}

| | | | |
|---|---|---|---|
| r7 | Oplustil Arnošt | 9000 | 36 |
| r6 | Novák Karel | 13000 | 19 |
| r10 | Zlámal Alois | 13000 | 52 |
| r3 | Malý Zdeněk | 15000 | 45 |
| r1 | Čech Jaroslav | 17000 | 27 |
| r8 | Papoušek Jindřich | 19000 | 50 |
| r2 | Dostál Jan | 21000 | 33 |
| r4 | Mrázek František | 22000 | 37 |
| r9 | Richter Tomáš | 26000 | 41 |
| r5 | Novák Josef | 32000 | 25 |

{r6, r10, r3, r1, r8, r2, r4, r9}

| | | | |
|---|---|---|---|
| r6 | Novák Karel | 13000 | 19 |
| r5 | Novák Josef | 32000 | 25 |
| r1 | Čech Jaroslav | 17000 | 27 |
| r2 | Dostál Jan | 21000 | 33 |
| r7 | Oplustil Arnošt | 9000 | 36 |
| r4 | Mrázek František | 22000 | 37 |
| r9 | Richter Tomáš | 26000 | 41 |
| r3 | Malý Zdeněk | 15000 | 45 |
| r8 | Papoušek Jindřich | 19000 | 50 |
| r10 | Zlámal Alois | 13000 | 52 |

{r6, r5, r1, r2, r7, r4}

intersection = {r4, r6}

# Multi-attribute indexing, examples

Index of concatenated attributes:

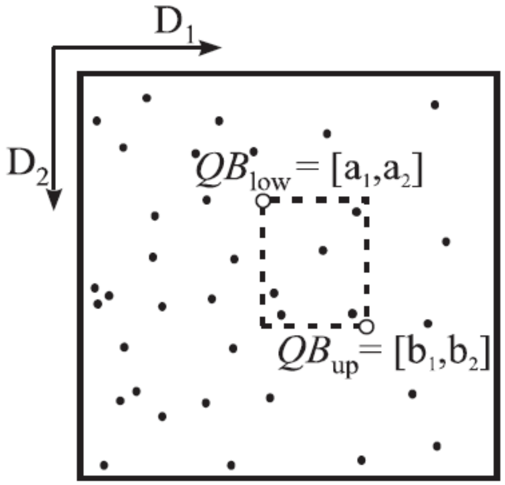
…WHERE **salary BETWEEN 10000 AND 30000** AND **age < 40** AND **name BETWEEN 'Dvořák' AND 'Procházka'**

| | | | |
|---|---|---|---|
| r1 | Čech Jaroslav | 17000 | 27 |
| r2 | Dostál Jan | 21000 | 33 |
| r3 | Malý Zdeněk | 15000 | 45 |
| r4 | Mrázek František | 22000 | 37 |
| r5 | Novák Josef | 32000 | 25 |
| r6 | Novák Karel | 13000 | 19 |
| r7 | Oplustil Arnošt | 9000 | 36 |
| r8 | Papoušek Jindřich | 19000 | 50 |
| r9 | Richter Tomáš | 26000 | 41 |
| r10 | Zlámal Alois | 13000 | 52 |

⇨ {r4, r6}

# Spatial indexing

- abstraction of M-tuples of keys as M-dimensional vectors
  <Novák Josef, 32000, 25> ⇒ [sig('Novák Josef'), 32000, 25]
  - key ordering must be preserved, e.g., for sig(*)
- "geometerization" of the problem as searching in M-dimensional space $R^M$
- conjunctive range query = (hyper)-rectangle QB in space $R^M$ delimited by two points, intervals in all dimensions



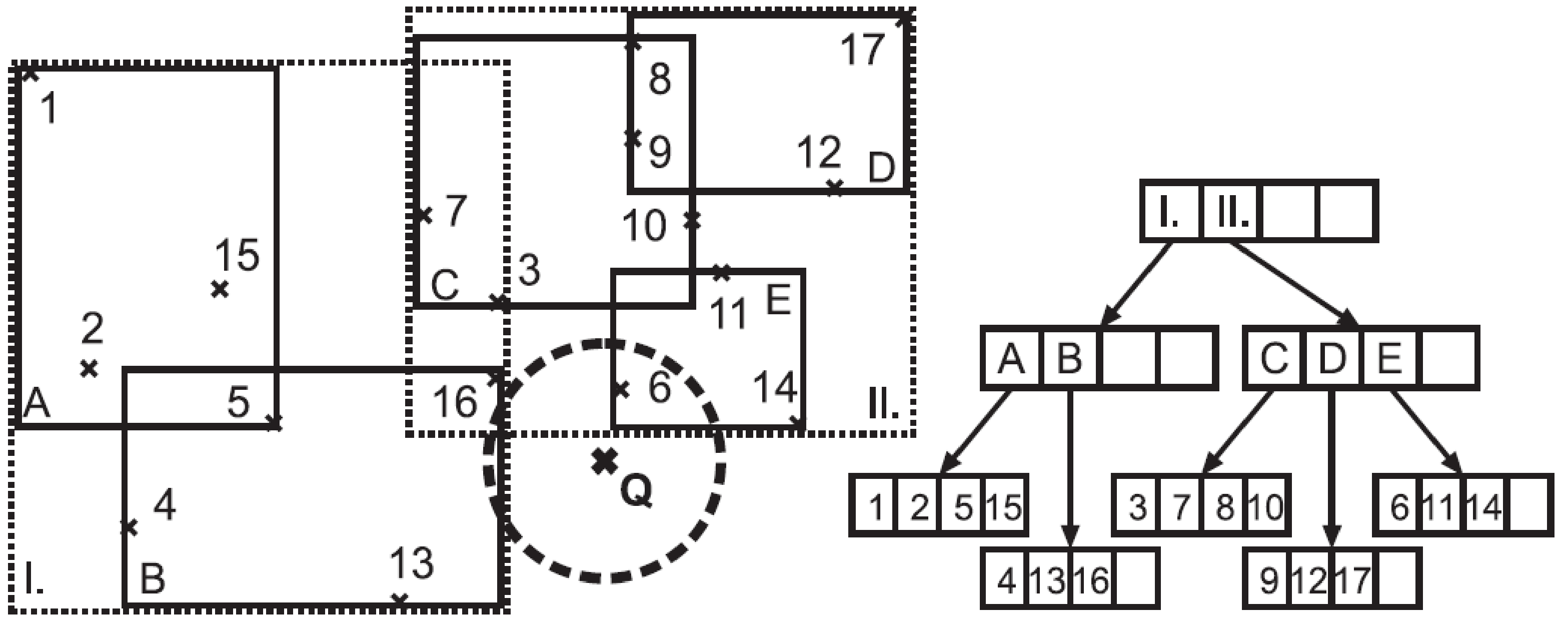
# Spatial indexing

- spatial access methods (SAM), points access methods (PAM)
  - various methods and indexing structures for speeding spatial queries
  - tree-based, hashing-based and sequential indexes
- (non-sequential) indexes use clustering of the close data within index pages
  - data close in space are also close in the index
- in ideal case, during range query only those index pages are accessed that contain the relevant index keys
- work well for a few dimensions (up to 10), for higher dimensions the sequential approaches are better

# Spatial indexing

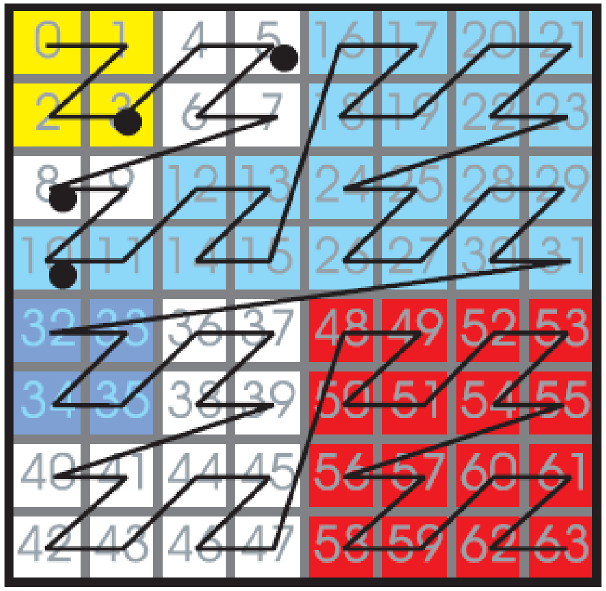
- tree indexes
  - R-tree, UB-tree, X-tree, etc.
- hashed indexes
  - Grid file
- sequential indexes
  - VA-file

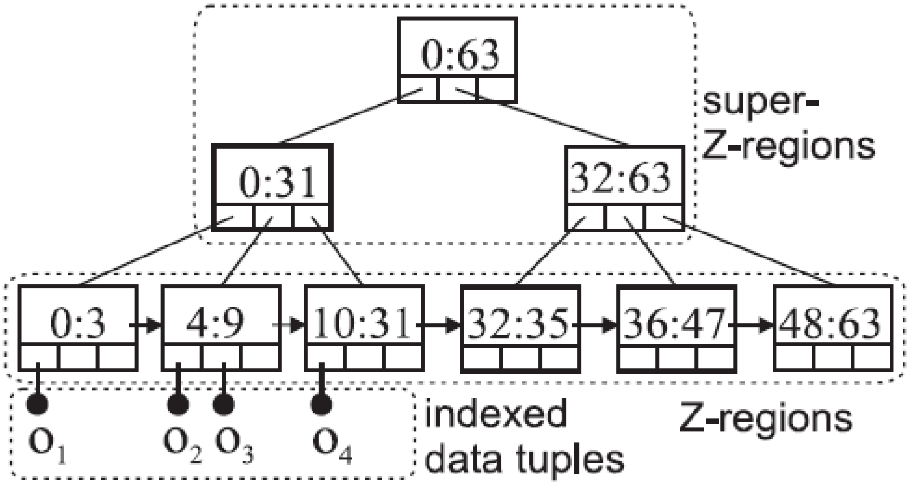
# R-tree



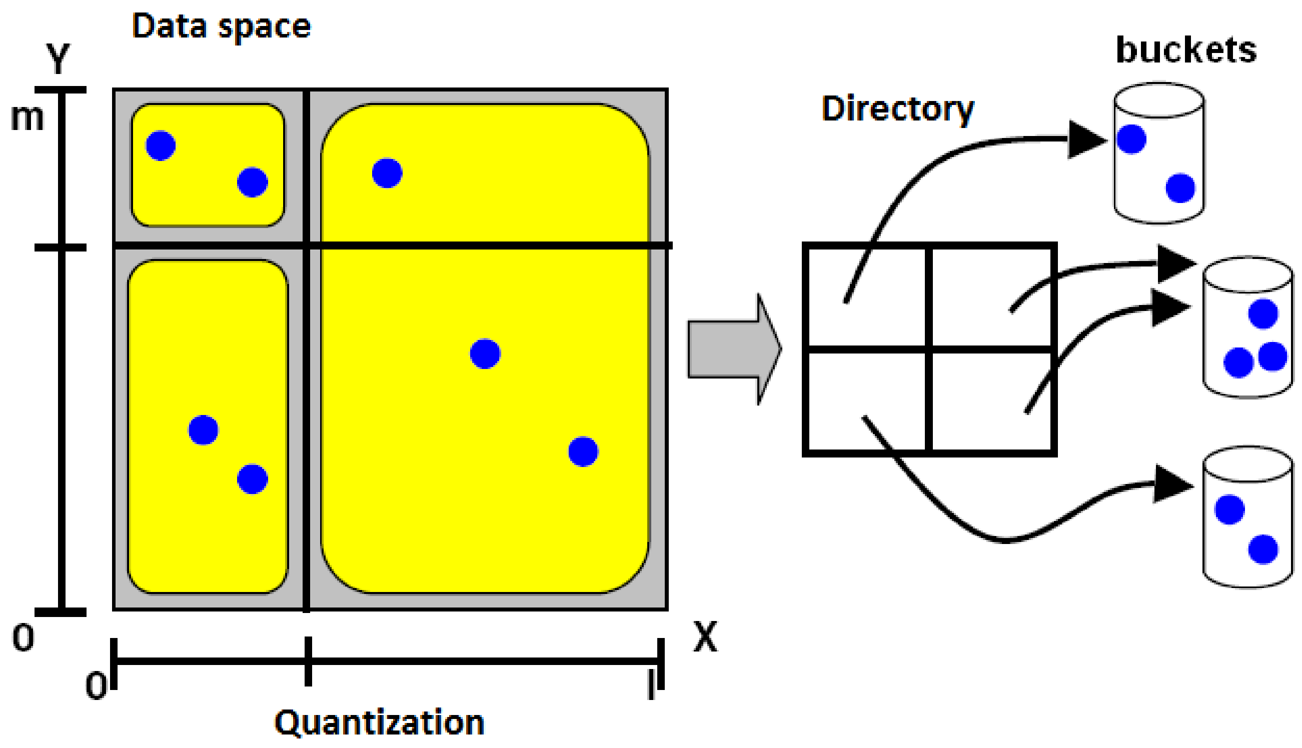
Demo: http://www.dbnet.ece.ntua.gr/~mario/rtree/

# UB-tree




0:63
super-Z-regions
0:31
32:63
0:3
4:9
10:31
32:35
36:47
48:63
o1
o2 o3
o4
indexed data tuples
Z-regions

# Grid file


Data space
Y
m
0
0
X
l
Quantization
Directory
buckets