

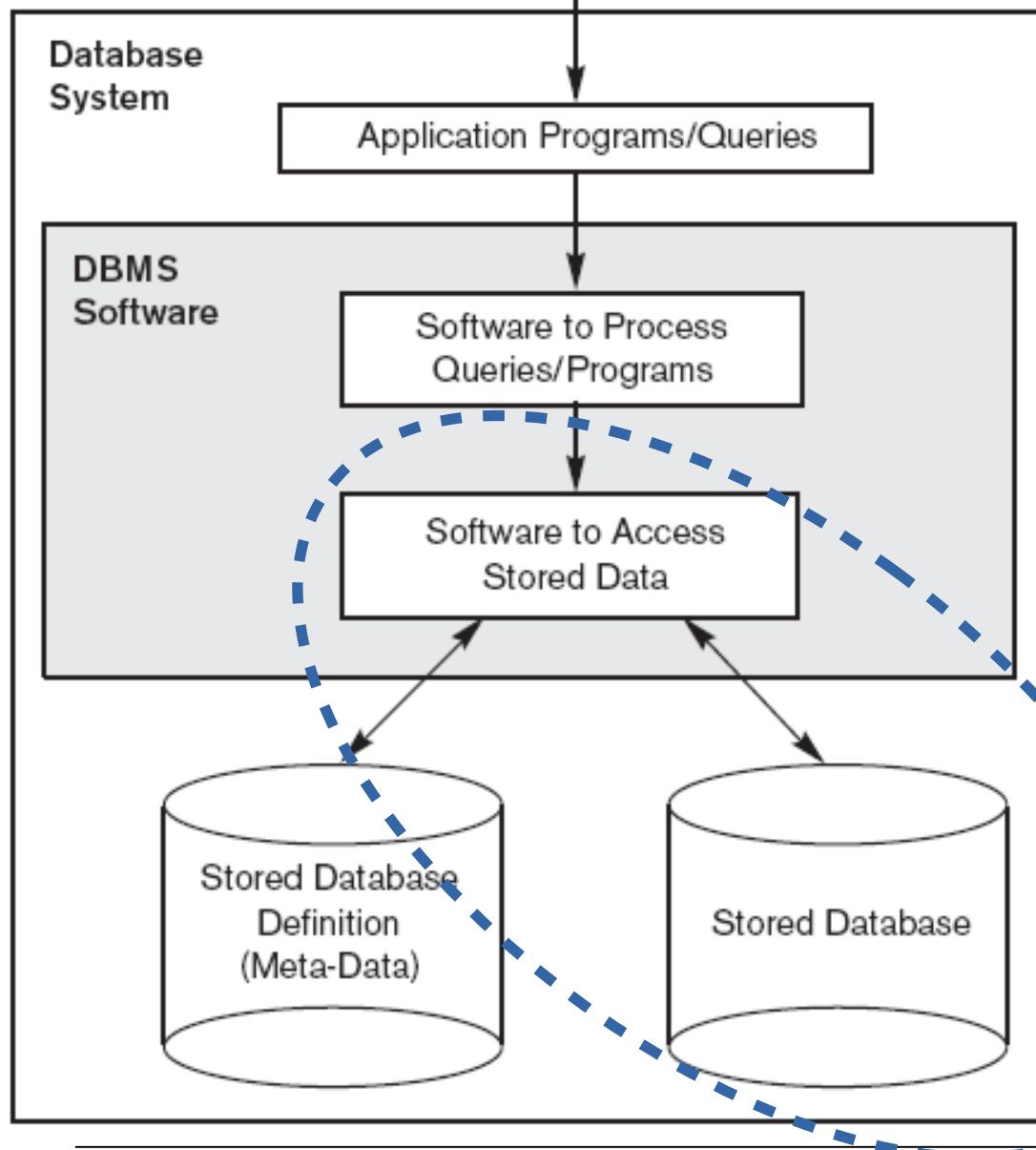
# Database Technology

## Topic 8: Data Structures for Databases

Olaf Hartig

[olaf.hartig@liu.se](mailto:olaf.hartig@liu.se)

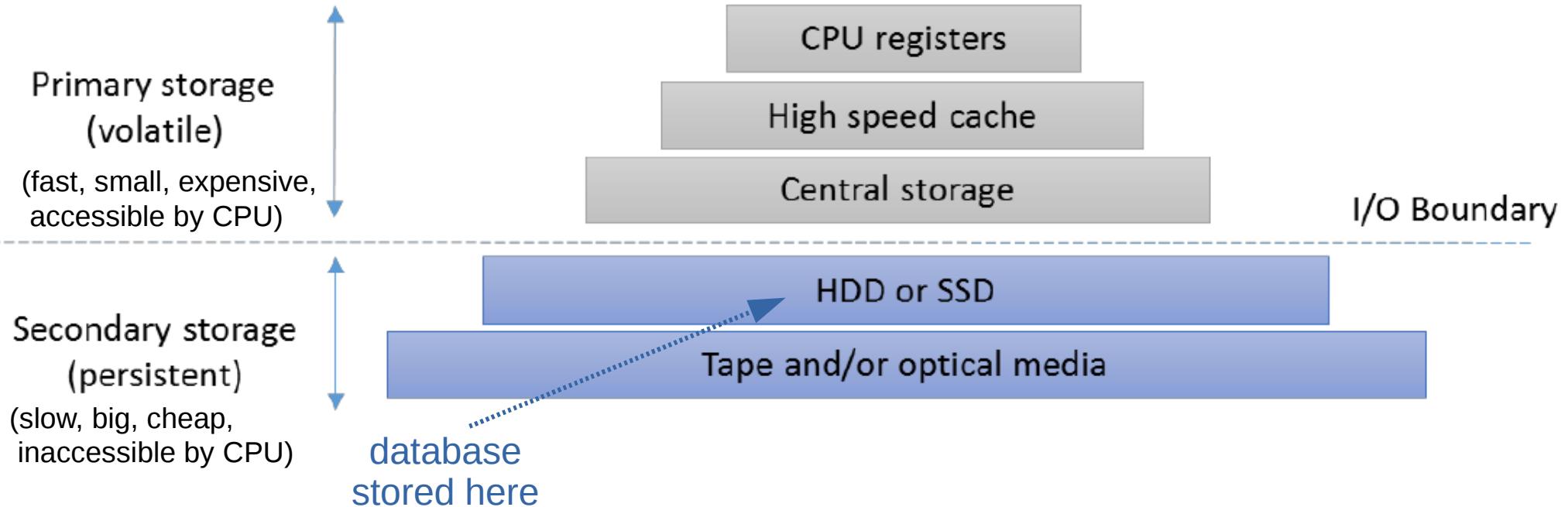
# Database System



**Figure 1.1**  
A simplified database system environment.

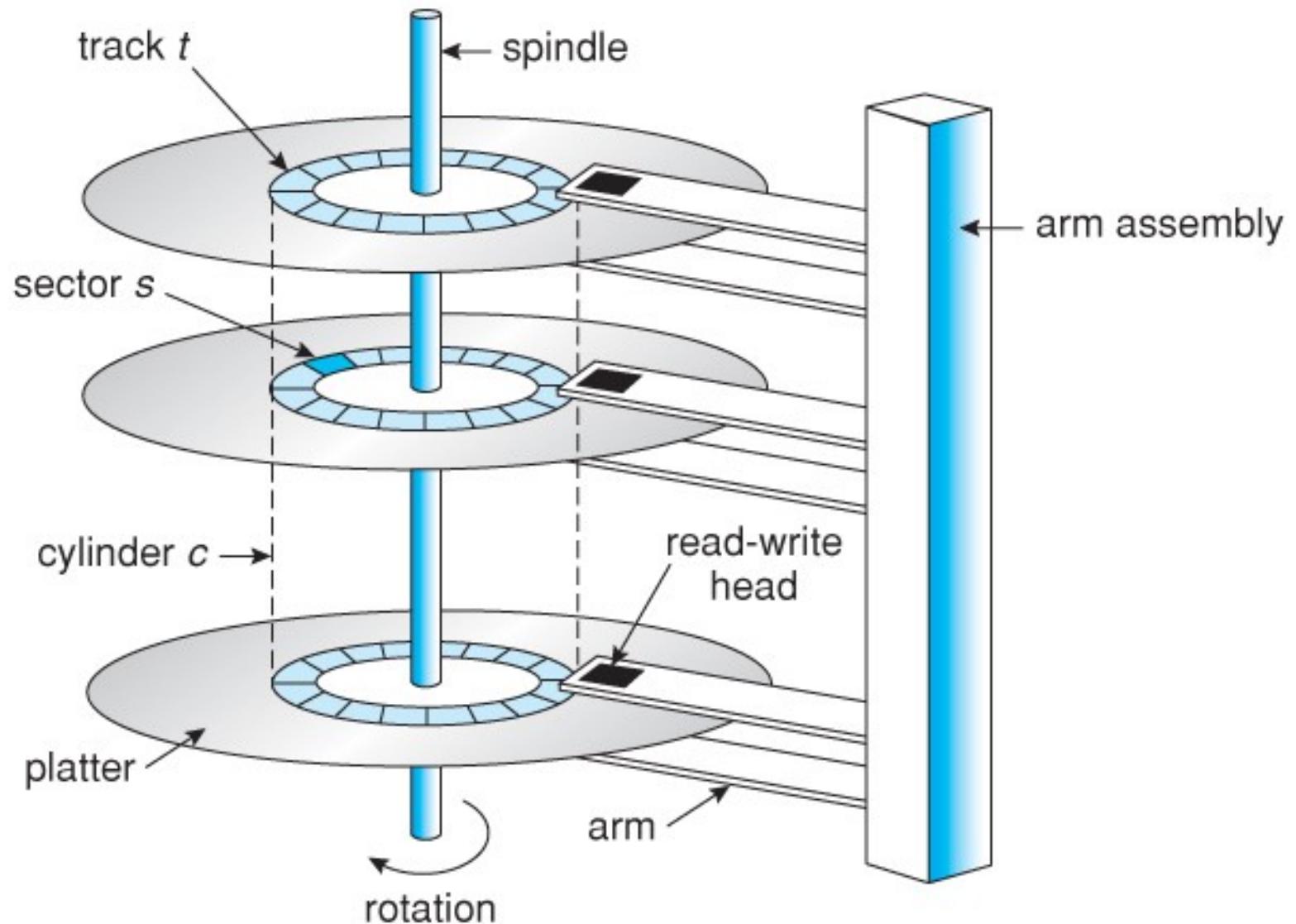
# Storage Hierarchy

# Storage Hierarchy

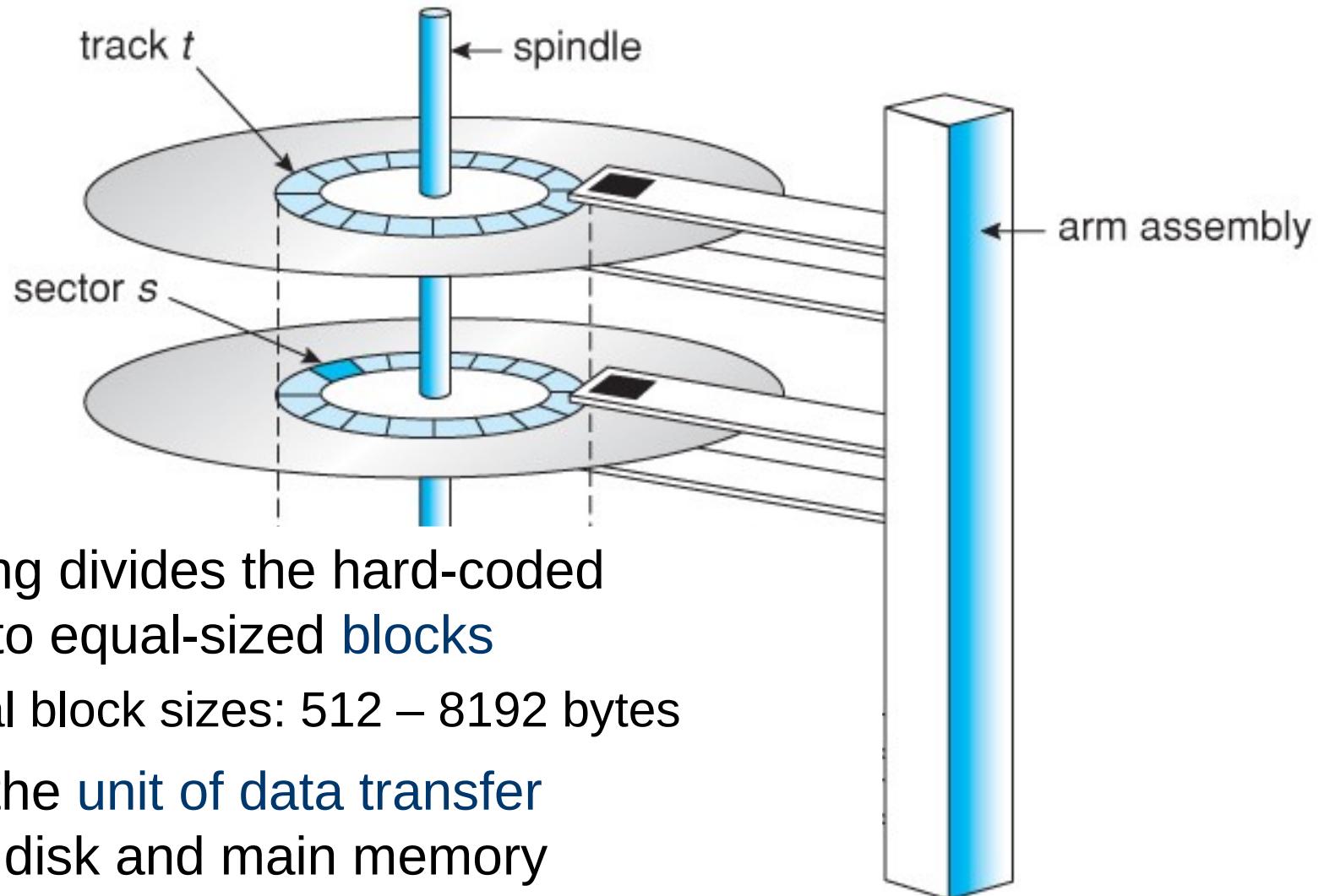


- Reading from / writing to disk is a major **bottleneck!**
  - CPU instruction: ca. 1 ns ( $10^{-9}$  secs)
  - Main memory access: ca. 10 ns ( $10^{-8}$  secs)
  - Disk access: ca. 1 ms (1M ns,  $10^{-3}$  secs)

# Magnetic Hard Disk Drives



# Magnetic Hard Disk Drives



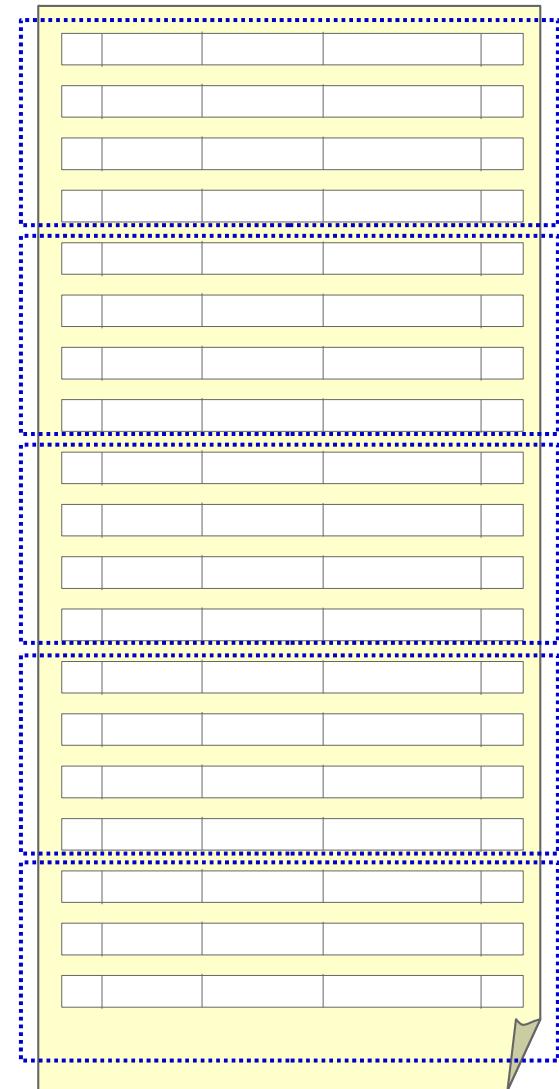
- Formatting divides the hard-coded sector into equal-sized **blocks**
  - Typical block sizes: 512 – 8192 bytes
- Block is the **unit of data transfer** between disk and main memory

# Files and Records

- Block is the unit of data transfer between disk and main memory

# Terminology

- Data stored in files
- File is a sequence of records
- Record is a set of field values
- For instance,
  - file = relation
  - record = row
  - field = attribute value
- Block is the unit of data transfer between disk and main memory
  - Records are allocated to **file blocks**



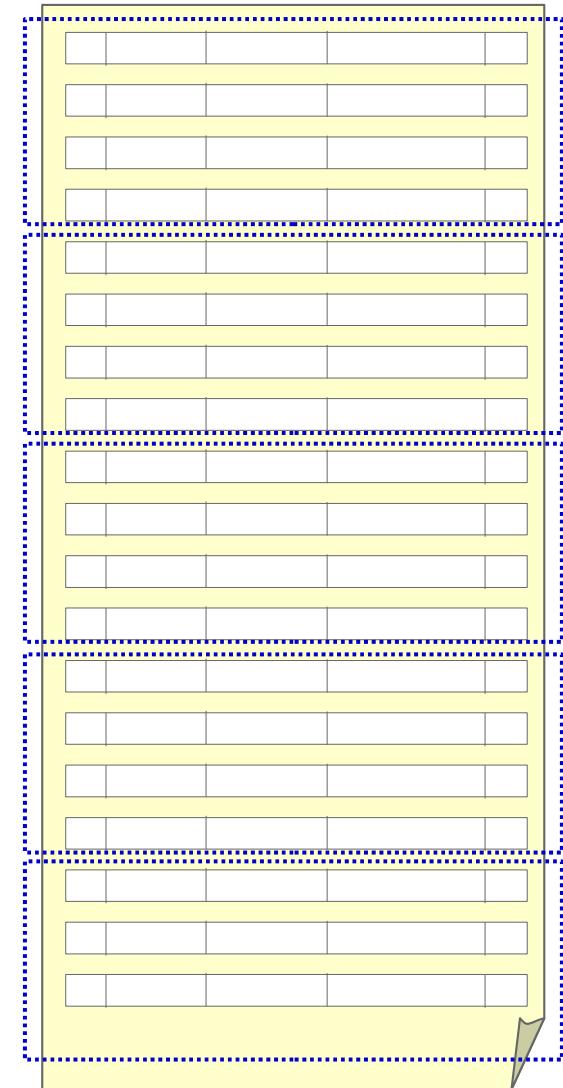
# Blocking Factor

- Blocking factor ( $bfr$ ) is the number of records per block
- Assume
  - $r$  is the number of records in a file,
  - $R$  is the size of a record, and
  - $B$  is the block size in bytes,

then:

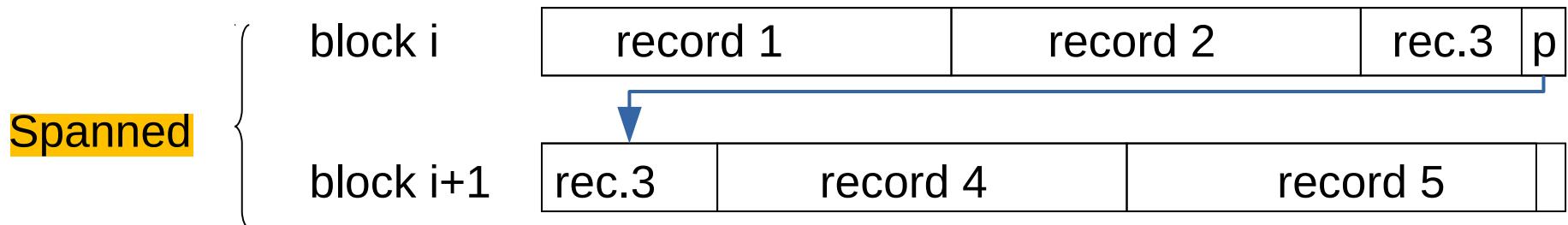
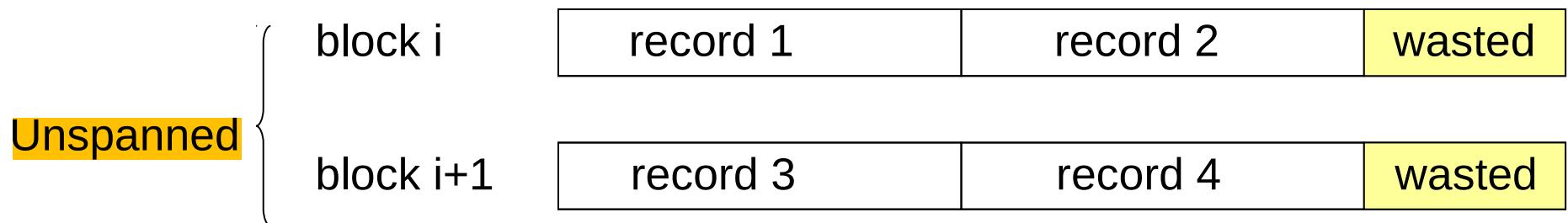
$$bfr = \left\lfloor \frac{B}{R} \right\rfloor$$

- Blocks needed to store the file:  $b = \left\lceil \frac{r}{bfr} \right\rceil$
- Space wasted per block =  $B - bfr * R$



# Spanned Records

... avoid wasting space



# Allocating File Blocks on Disk

- **Contiguous allocation:** file blocks allocated consecutively (one after another)
  - Fast sequential access, but expanding is difficult
- **Linked allocation:** each file block contains a pointer to the next one
  - Expanding the file is easy, but reading is slower
- **Linked clusters allocation:** hybrid of the two above
  - i.e., linked clusters of consecutive blocks
- **Indexed allocation:** index blocks contain pointers to the actual file blocks

# File Organization

(Organizing Records in Files)

# Heap Files

- Records are added to the end of the file
- Adding a record is cheap
- Retrieving, removing, and updating a record is expensive because it implies *linear search*
  - Average case:  $\left\lceil \frac{b}{2} \right\rceil$  block accesses\*
  - Worst case:  $b$  block accesses
- Record removal also implies waste of space
  - Periodic reorganization

\*recall,  $b$  is the number of blocks of the file

# Sorted Files

- Records ordered according to some field
- Ordered record retrieval is cheap (i.e., on the ordering field, otherwise expensive)
  - All the records: access the blocks sequentially
  - Next record: probably in the same block
  - Random record: *binary search*; hence,  $\lceil \log_2 b \rceil$  block accesses in the worst case\*
- Adding a record is expensive, but removing is less expensive (deletion markers and periodic reorganization)

\*recall,  $b$  is the number of blocks of the file

# Hash Files

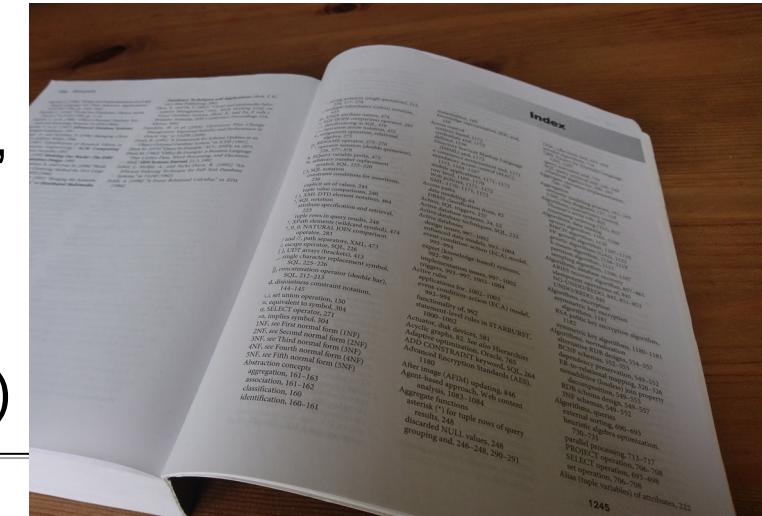
- File is logically split into “buckets”
  - Bucket: one or several contiguous disk blocks
  - Table converts bucket number into address of block
- Choose a field of the records to be the *hash field*
- Given a record, which bucket does it belong to?
  - apply hash function  $h$  to the value  $x$  that the record has in its hash field
  - resulting hash value  $h(x)$  is the number of the bucket into which the record goes
- Cheapest random retrieval (when searching for equality)
- Ordered record retrieval is expensive

# Indexes

## (Secondary Access Methods)

# Motivation

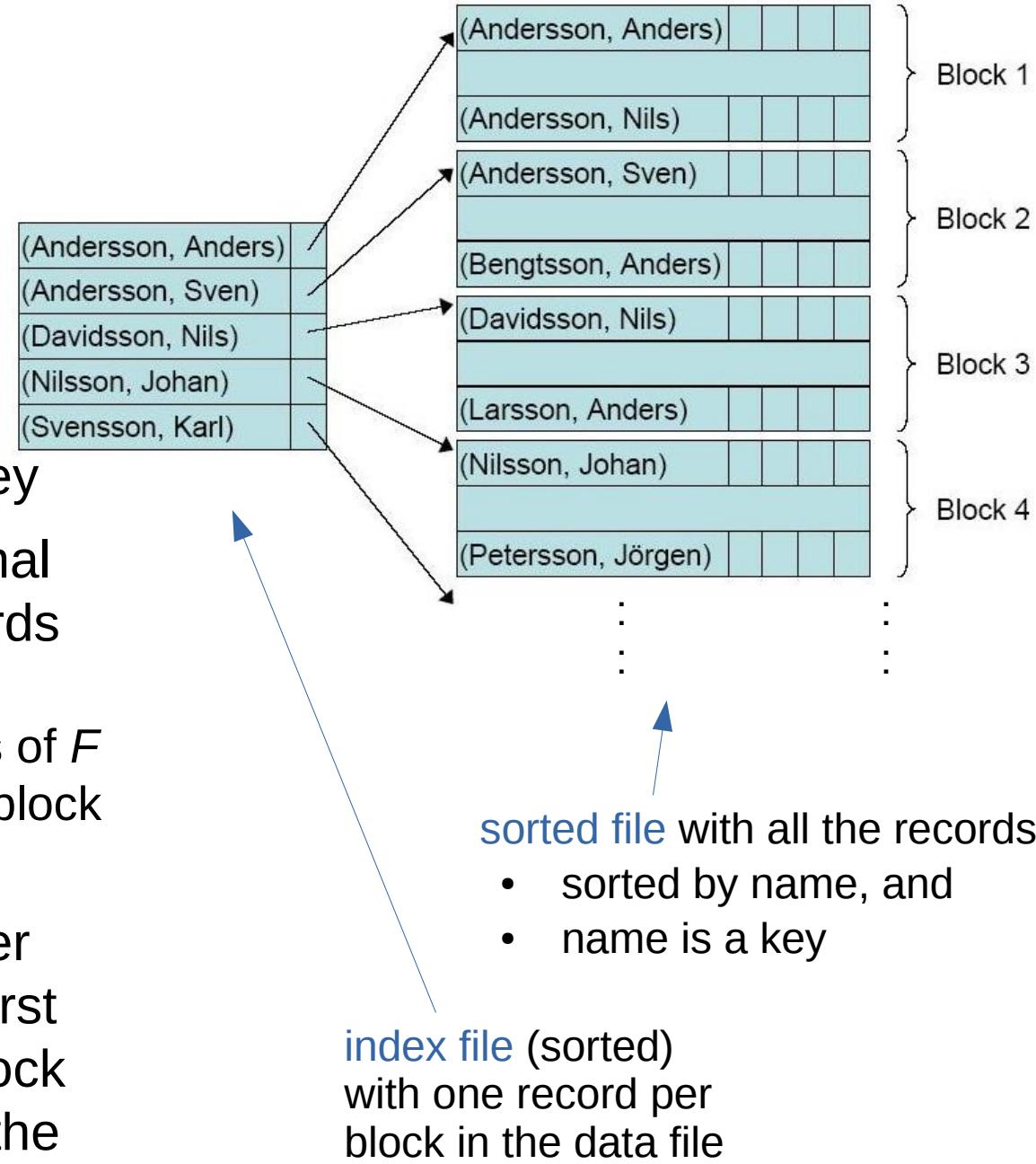
- File organization (heap, sorted, hash) determines primary method to access data in a file
  - e.g., sequential search, binary search, hash-based
- However, this may not be fast enough
- Solution: index files
  - introduce secondary access methods
  - goal: speed up access under specific conditions
  - there exist various types of index structures
- Outline:
  - 1) Single-level ordered indexes (primary, secondary, and clustering indexes)
  - 2) Multilevel indexes
  - 3) Dynamic multilevel indexes (B+-trees)



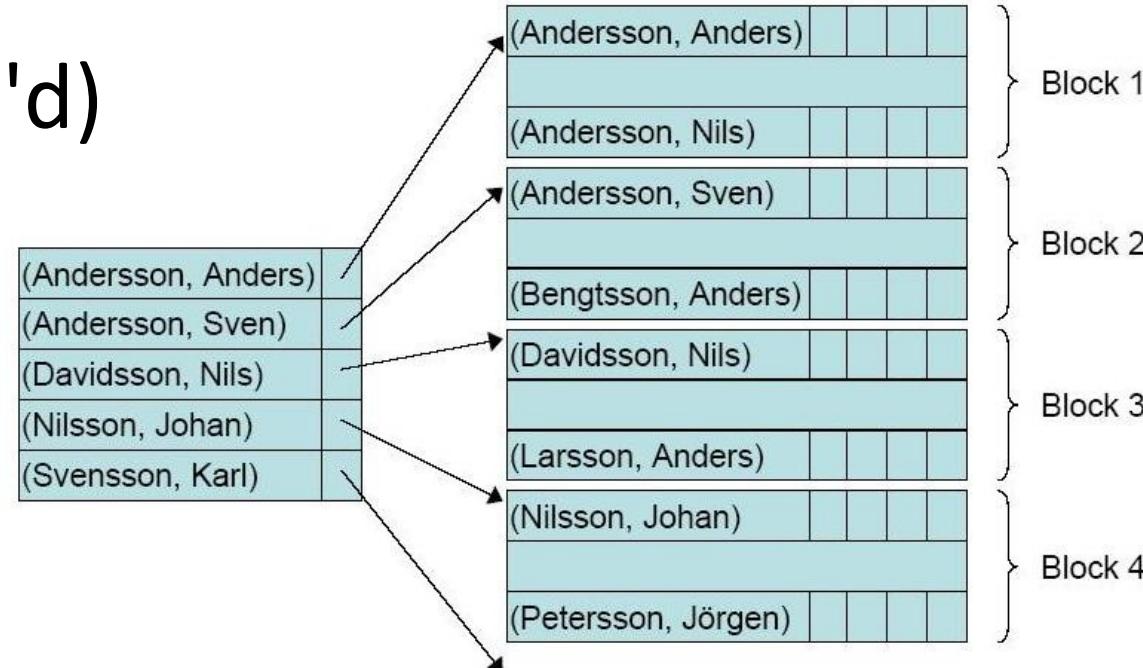
# Single-Level Ordered Indexes

# Primary Index

- Assumptions:
  - Data file is sorted
  - Ordering field  $F$  is a key
- Primary index: an additional *sorted file* whose records contain two fields:
  - $V$  - one of the values of  $F$
  - $P$  - pointer to a disk block of the data file
- One index record  $(V,P)$  per data block such that the first data record in the data block pointed to by  $P$  has  $V$  as the value of the ordering key  $F$



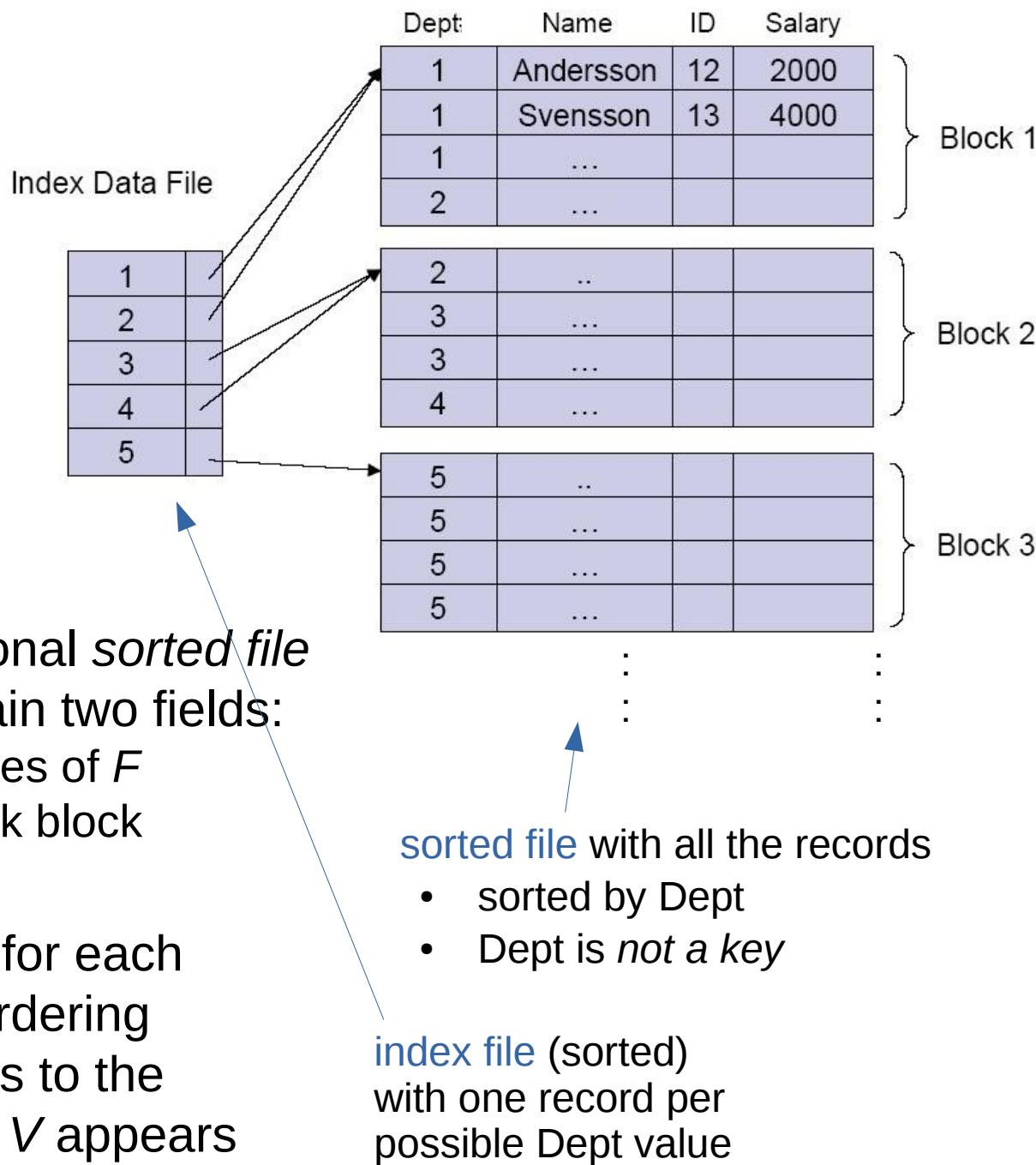
# Primary Index (cont'd)



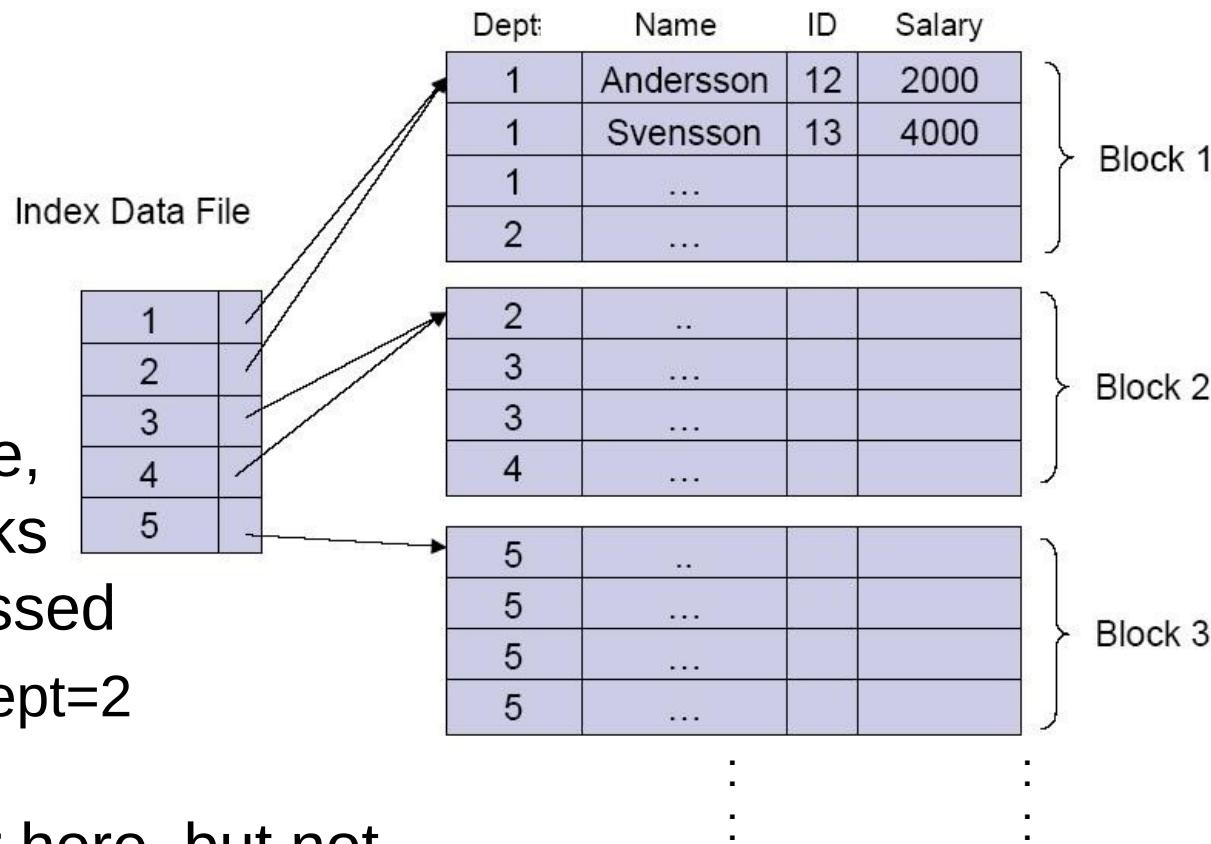
- Why is it faster to access a random record via a binary search in the index than in the data file?
  - Index file is much smaller than the data file because:
    - (a) Number of index records << number of data records
    - (b) Index records smaller than data records (i.e., higher blocking factor for the index file than for the data file)
  - Much smaller file → binary search converges must faster!
- There is a cost of maintaining the index!

# Clustering Index

- Assumptions:
  - Data file is sorted
  - Ordering field  $F$  is **not** a key (hence, we cannot assume distinct values)
- Clustering index: additional *sorted file* whose records contain two fields:
  - $V$  - one of the values of  $F$
  - $P$  - pointer to a disk block of the data file
- One index record ( $V,P$ ) for each distinct value  $V$  of the ordering field  $F$  such that  $P$  points to the first data block in which  $V$  appears



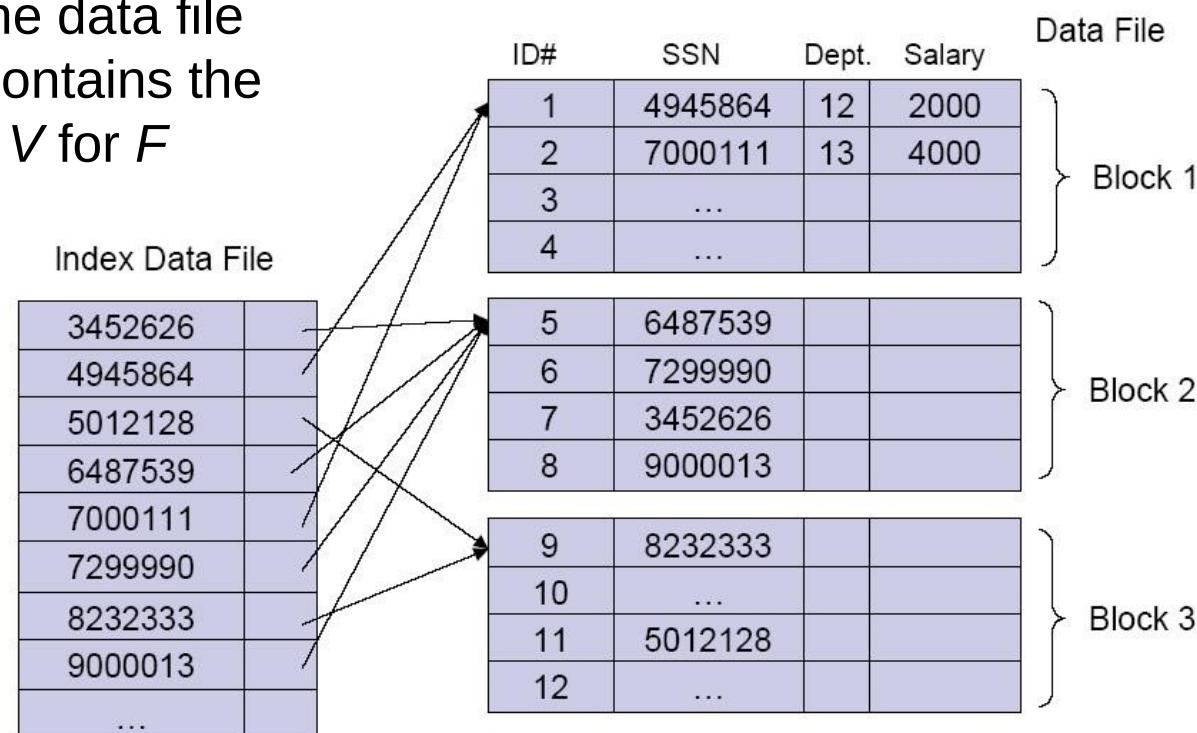
# Clustering Index



- Attention: after binary search in the index file, multiple data file blocks may need to be accessed
  - see, for instance, Dept=2
- Index file also smaller here, but not as much as for a primary index
  - number of index records  $\leq$  number of data records
  - at least, index records smaller than data records (like in a primary index)

# Secondary Indexes on Key Field

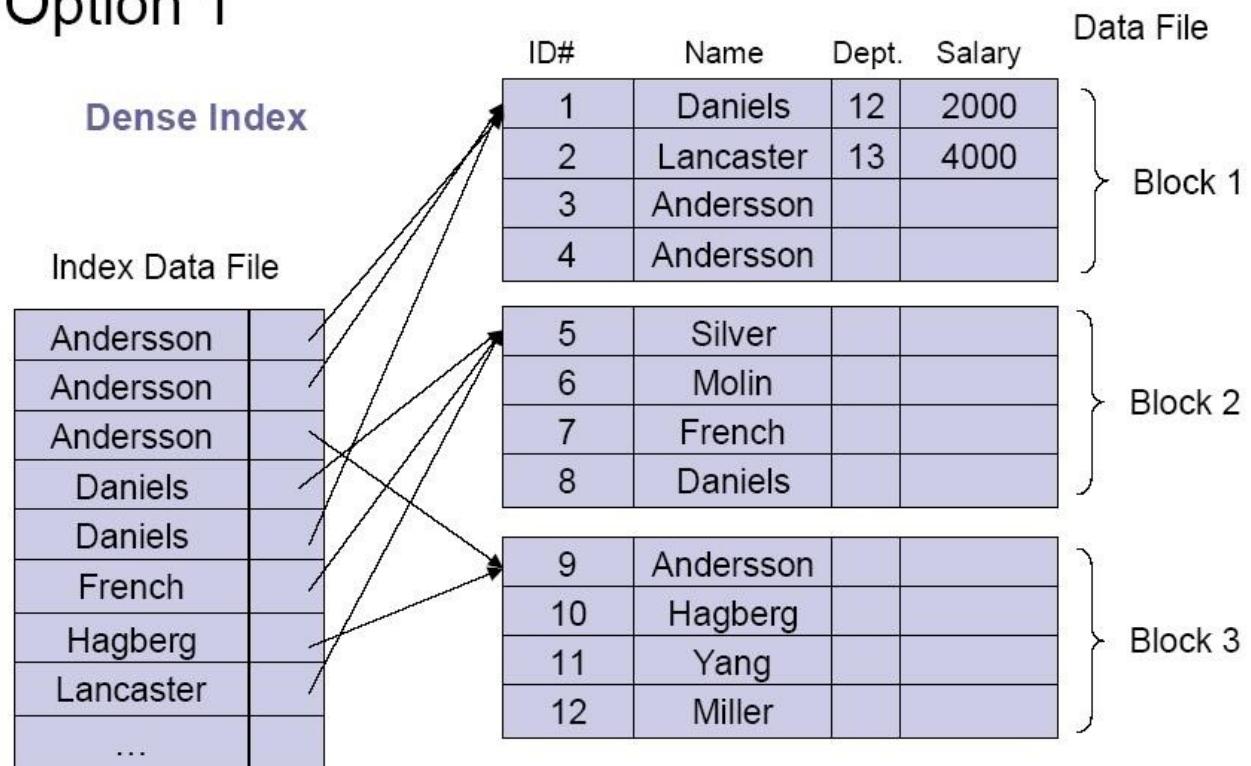
- Index on a *non-ordering key field F*
  - Data file may be sorted or not
- Secondary index: additional *sorted* file whose records contain two fields:
  - $V$  - one of the values of  $F$
  - $P$  - pointer to the data file block that contains the record with  $V$  for  $F$
- One index record per data record
- Searching based on a value of  $F$  can now be done with a binary search in the index



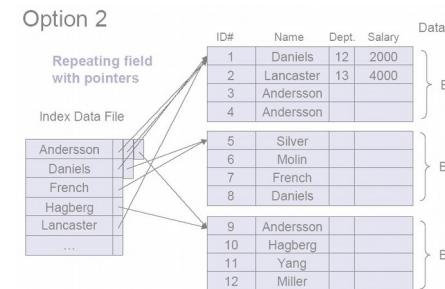
# Secondary Indexes on Non-Key

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

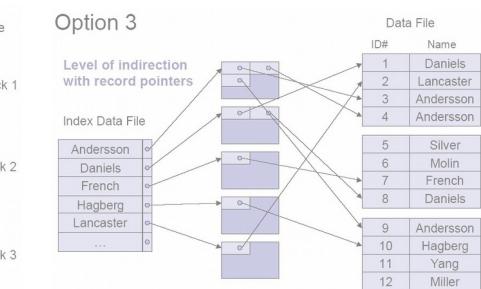
Option 1



Option 2



Option 3

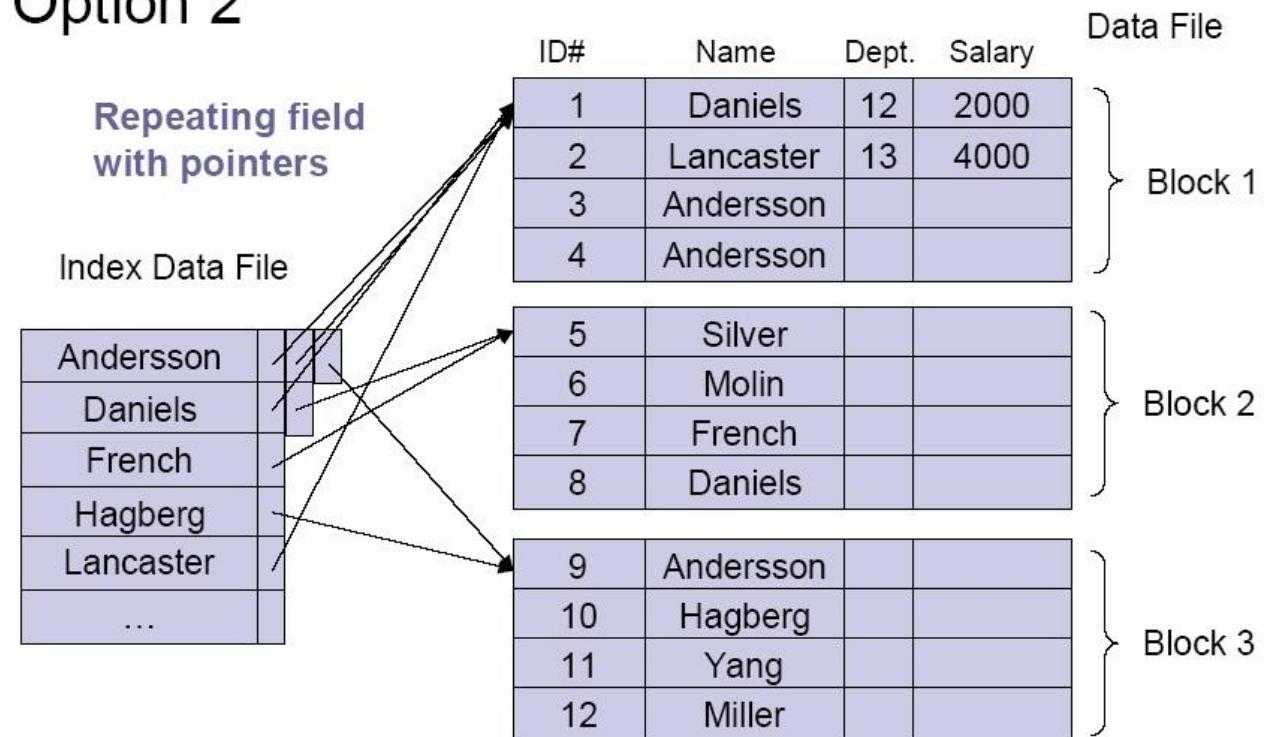


# Secondary Indexes on Non-Key

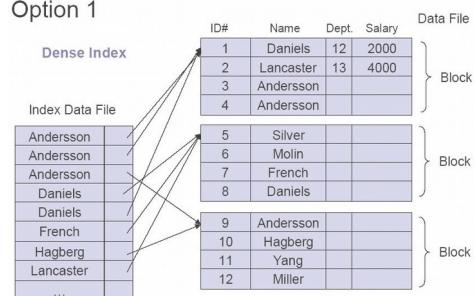
- Index on a *non-ordering non-key field*

同上，样本数据没有发生变化，但是在index中进行了有序的排列

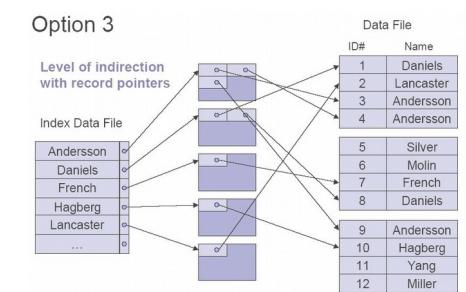
## Option 2



## Option 1



## Option 3



# Secondary Indexes on Non-Key

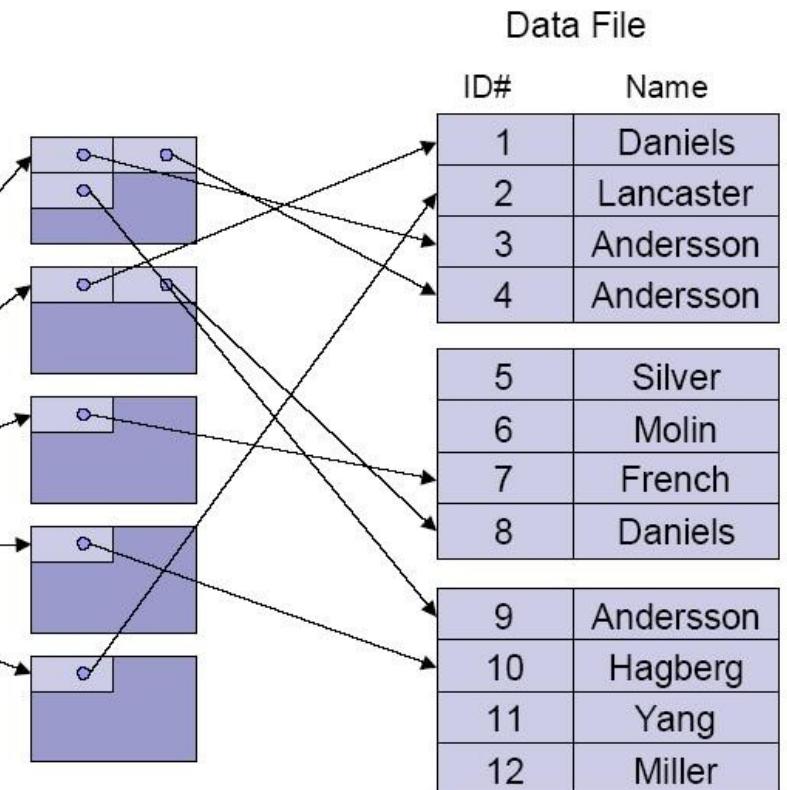
- Index on a *non-ordering non-key field*
- also called *inverted file*

## Option 3

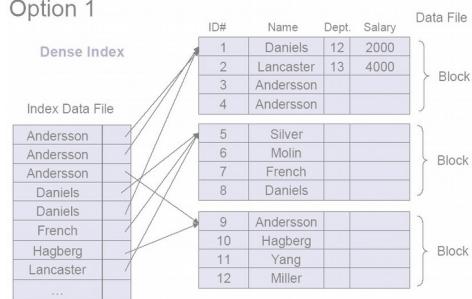
**Level of indirection with record pointers**

Index Data File

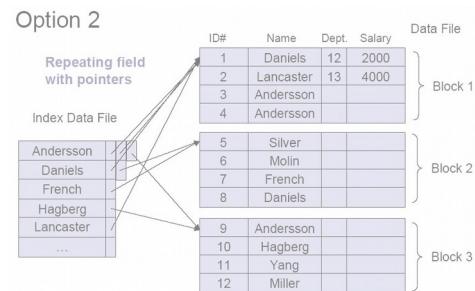
Andersson	o
Daniels	o
French	o
Hagberg	o
Lancaster	o
...	o



Option 1



Option 2



# Summary of Single-Level Indexes

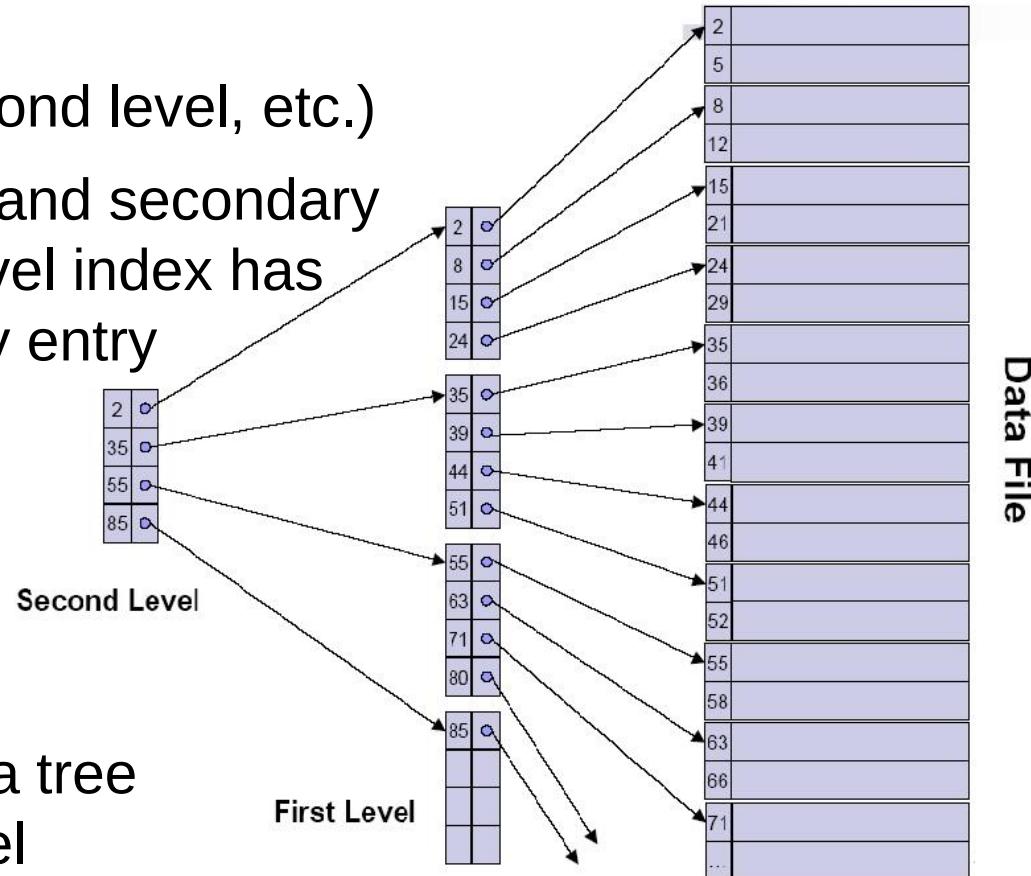
	Index field used for sorting the data records	Index field <b>not</b> used for sorting the data records
Index field is a key	Primary index	Secondary index (key)
Index field is not a key	Clustering index	Secondary index (non-key)

Type of index	Number of index entries
Primary	Number of blocks in data file
Clustering	Number of distinct index field values
Secondary (key)	Number of records in data file
Secondary (non-key)	Number of records or number of distinct index field values

# Multilevel Indexes

# 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 highest level fits into a single block
- Such a full multilevel index is a tree
  - single block of highest level is the root node in this tree
- How many block accesses to retrieve a random record?
  - number of index levels + 1



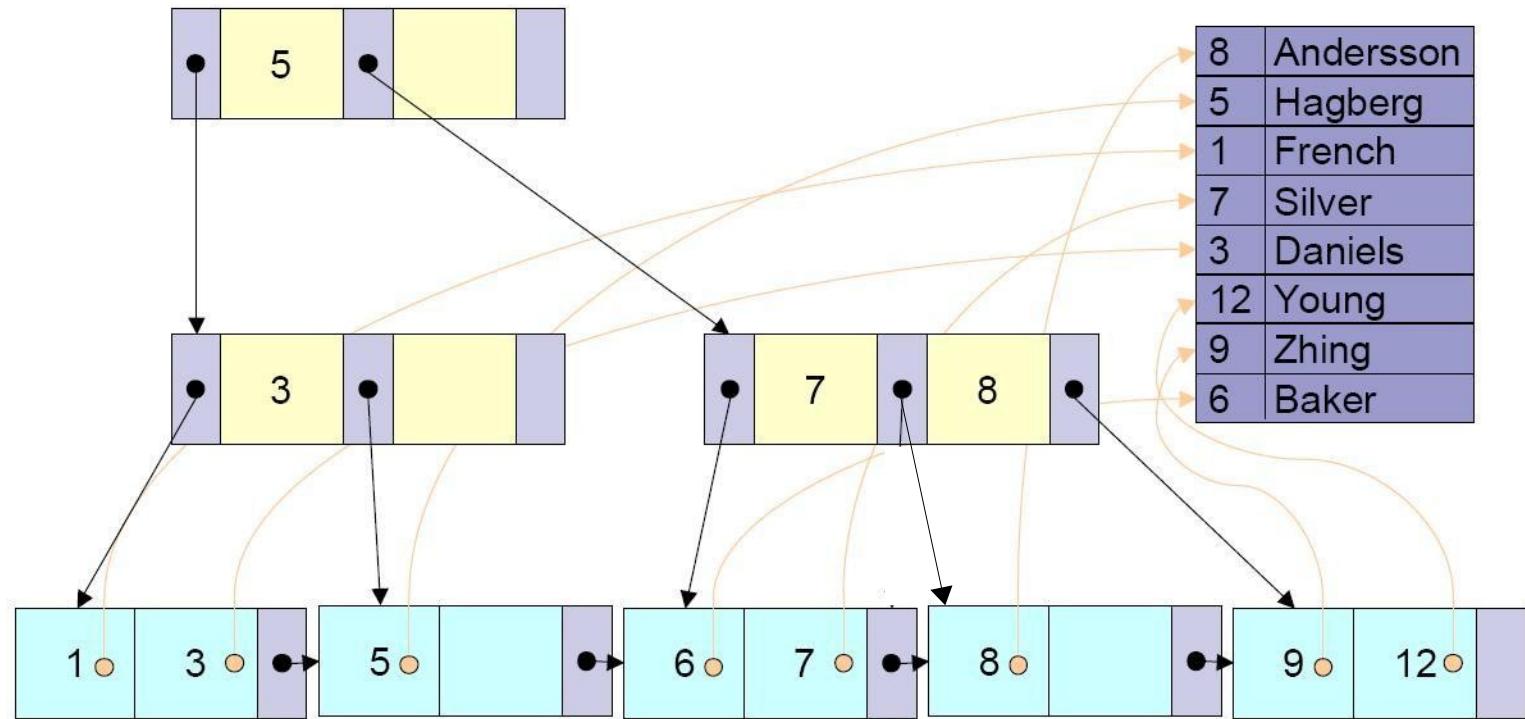
# Multilevel Indexes (cont'd)

- When using a (static) multilevel index, record insertion, deletion, and update may be expensive because all the index levels are sorted files
- Solutions:
  - Overflow area + periodic reorganization
  - Dynamic multilevel indexes that leave some space in index blocks for new entries (e.g., B-trees and B+-trees)

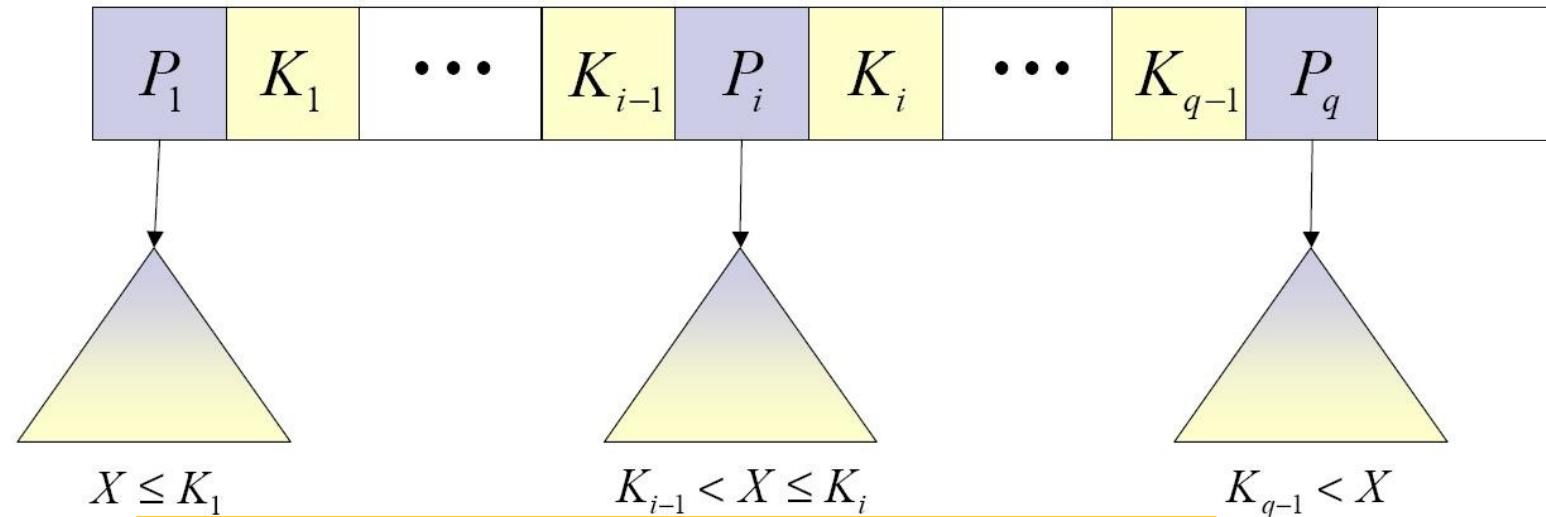
# B<sup>+</sup>-Trees

Dynamic Multilevel Indexes

# Example B<sup>+</sup>-Tree



# Internal Nodes of a B<sup>+</sup>-Tree



- $q \leq p$  (where  $p$  is the order of the B<sup>+</sup>-tree)
- Every  $K_i$  is an index value, every  $P_i$  is a tree pointer
- Within each node:  $K_1 < K_2 < \dots < K_{q-1}$
- For every value  $X$  in the  $P_i$  subtree:  $K_{i-1} < X \leq K_i$
- Each internal node (except the root) must be at least half full
  - i.e., there must be at least  $\left\lceil \frac{p}{2} \right\rceil$  tree pointers

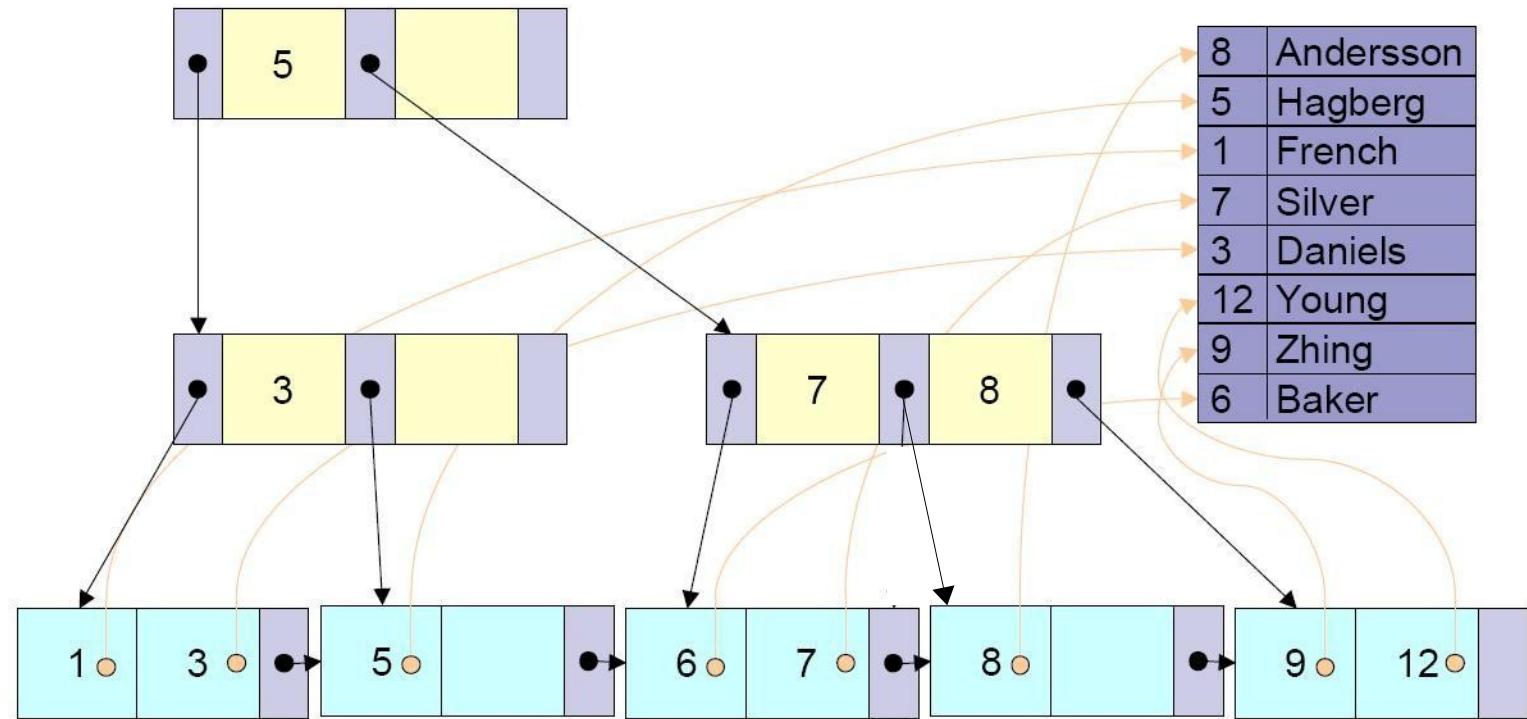
# Leaf Nodes of a B<sup>+</sup>-Tree

$K_1$	$Pr_1$	...	$K_i$	$Pr_i$	...	$K_q$	$Pr_q$	$P_{\text{next}}$
-------	--------	-----	-------	--------	-----	-------	--------	-------------------

- $q \leq p$  (where  $p$  is the order for leaf nodes of the B<sup>+</sup>-tree)
- Every  $K_i$  is an index value
- Every  $Pr_i$  is a data pointer to the data file block that contains the record with index value  $K_i$
- $P_{\text{next}}$  is a pointer to the next leaf node
- Within each node:  $K_1 < K_2 < \dots < K_q$
- Every leaf node must be at least half full
  - i.e., at least  $\left\lceil \frac{p}{2} \right\rceil$  index values in each leaf node

# Retrieval of Records in a B<sup>+</sup>-Tree

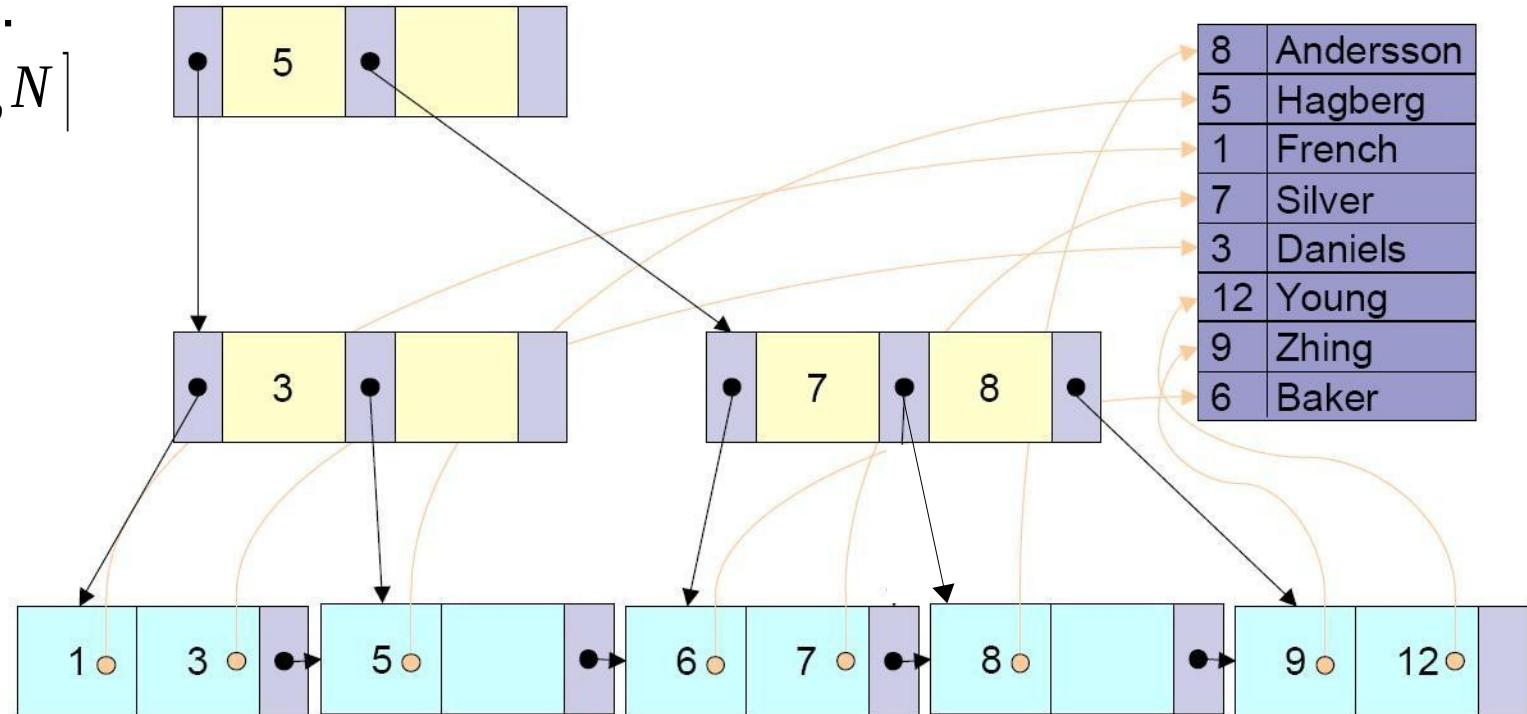
- Very fast retrieval of a random record
- Number of block accesses: depth of tree + 1



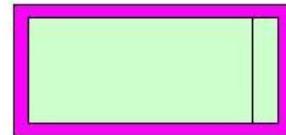
# Depth of a B<sup>+</sup>-Tree

- Given that internal nodes must have at least  $\left\lceil \frac{p}{2} \right\rceil$  children,
- For a depth of  $d$ , the number  $N$  of leaf nodes is at least  $\left\lceil \frac{p}{2} \right\rceil^d$
- Hence, in the worst case,  $d$  is at most  $\left\lceil \log_{\left\lceil \frac{p}{2} \right\rceil} N \right\rceil$
- Best case:

$$\left\lceil \log_p N \right\rceil$$

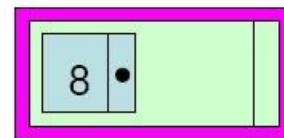


# B<sup>+</sup>-Tree Insertion



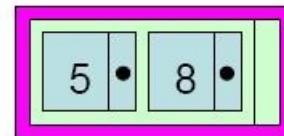
Insert: 8

# B<sup>+</sup>-Tree Insertion



Insert: 5

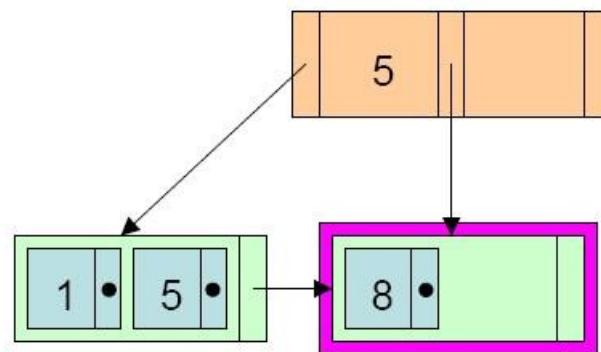
# B<sup>+</sup>-Tree Insertion



Overflow – create a new level

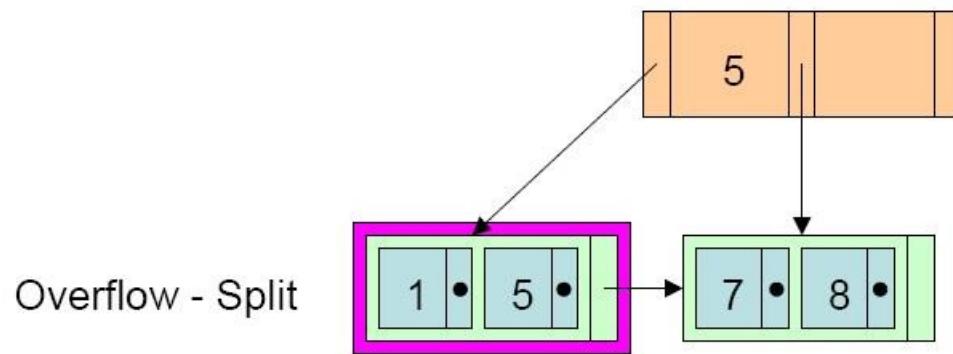
Insert: 1

# B<sup>+</sup>-Tree Insertion



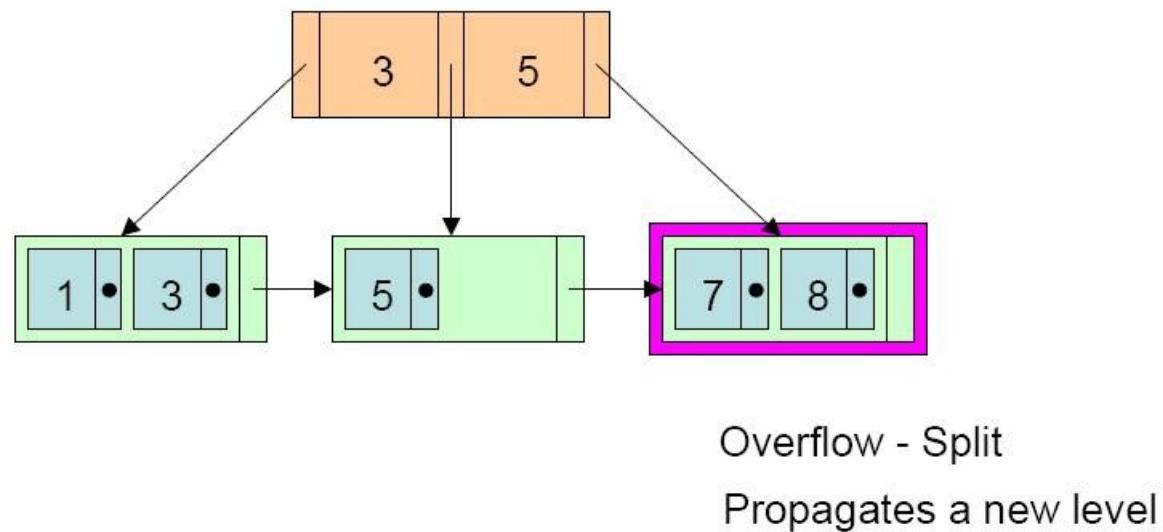
Insert: 7

# B<sup>+</sup>-Tree Insertion



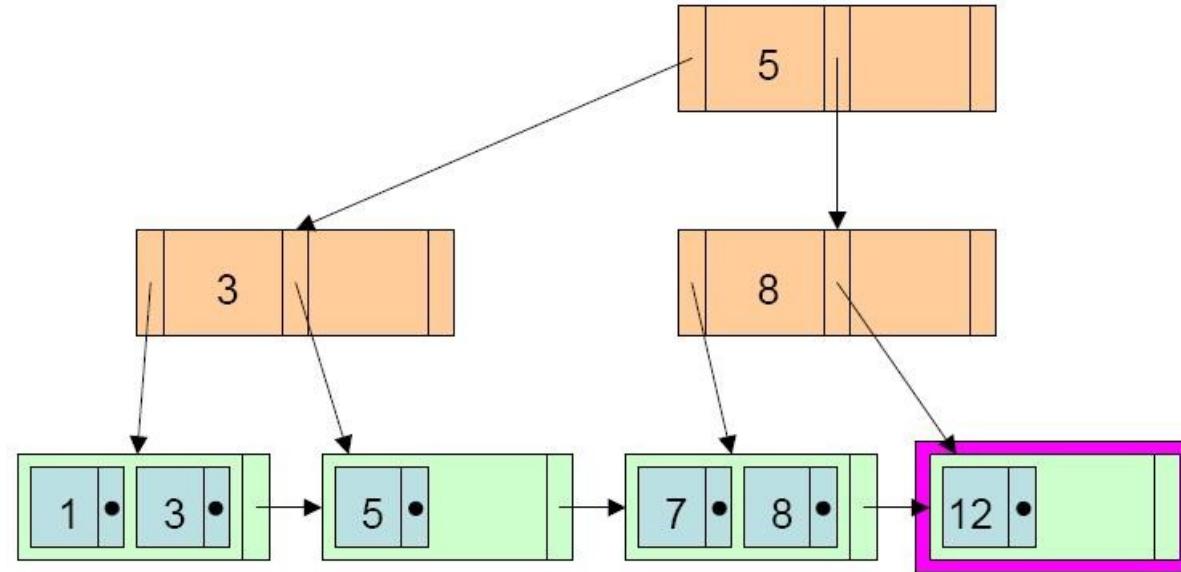
Insert: 3

# B<sup>+</sup>-Tree Insertion



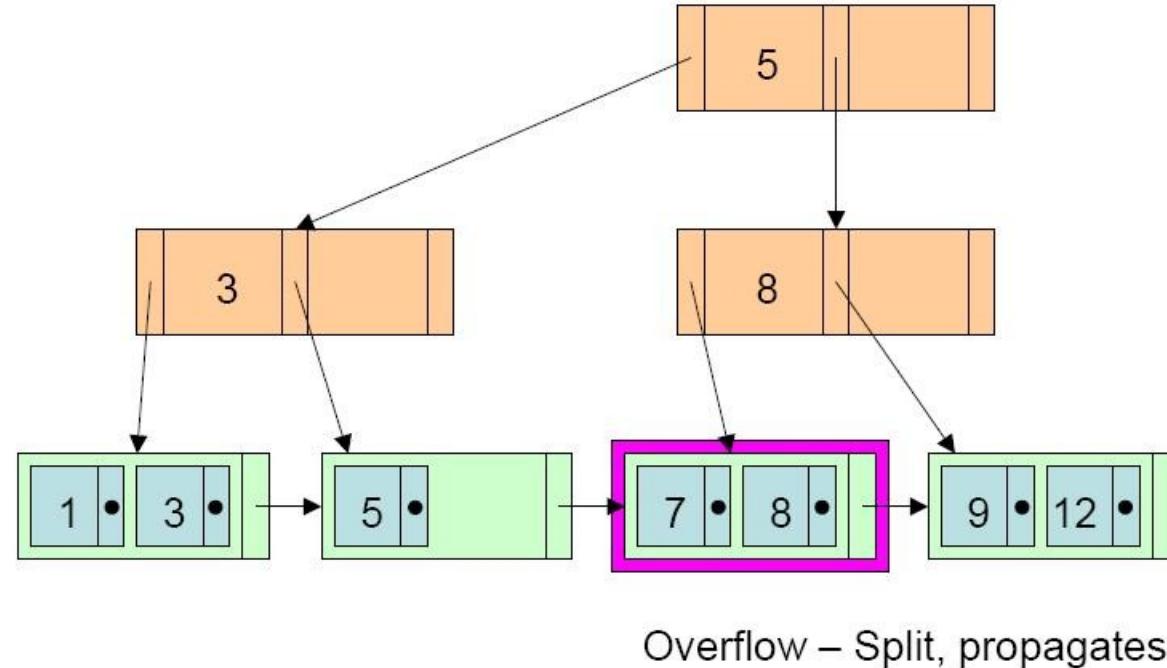
Insert: 12

# B<sup>+</sup>-Tree Insertion



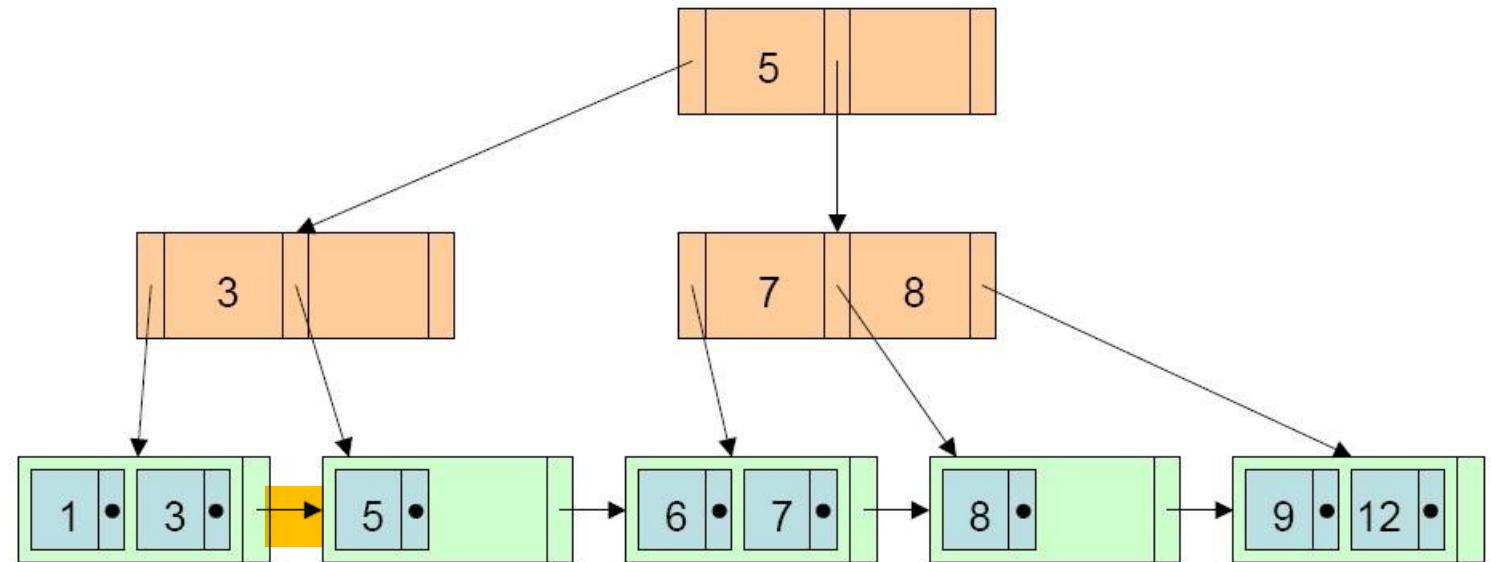
Insert: 9

# B<sup>+</sup>-Tree Insertion



Insert: 6

# B<sup>+</sup>-Tree Insertion



Resulting B+-tree

# Summary

# Summary

- Storage hierarchy
  - Accessing disk is major bottleneck
- Organizing records in files
  - Heap files, sorted files, hash files
- Indexes
  - Additional sorted files that provide efficient secondary access methods
- Primary, secondary, and clustering indexes
- Multilevel indexes
  - Retrieval requires reading fewer blocks
- Dynamic multilevel indexes
  - Leave some space in index blocks for new entries
  - B-tree and B+-tree

[www.liu.se](http://www.liu.se)