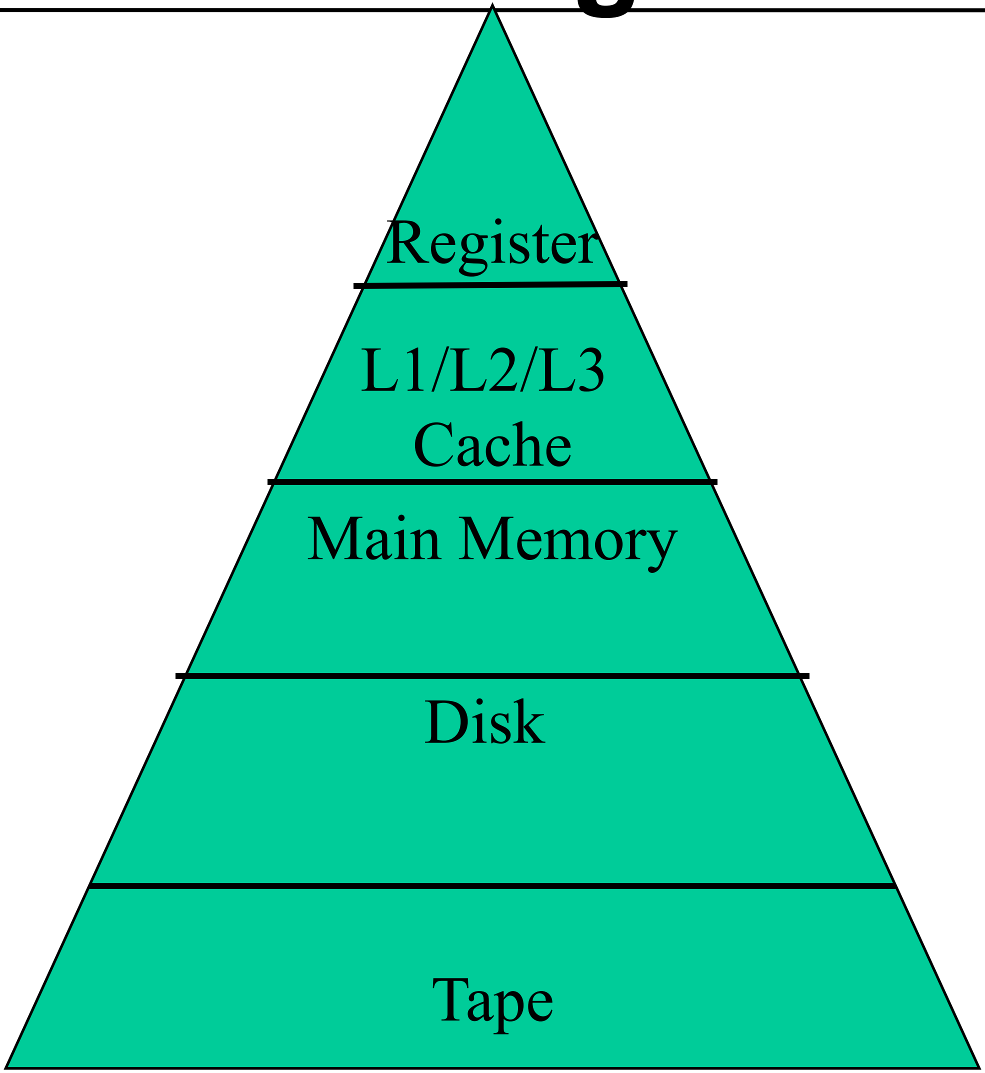
Physical Data Organisation

Topics:

- Storage hierarchy
- External storage
- Storage structures
- ISAM
- B-Trees
- Hashing
- Clustering

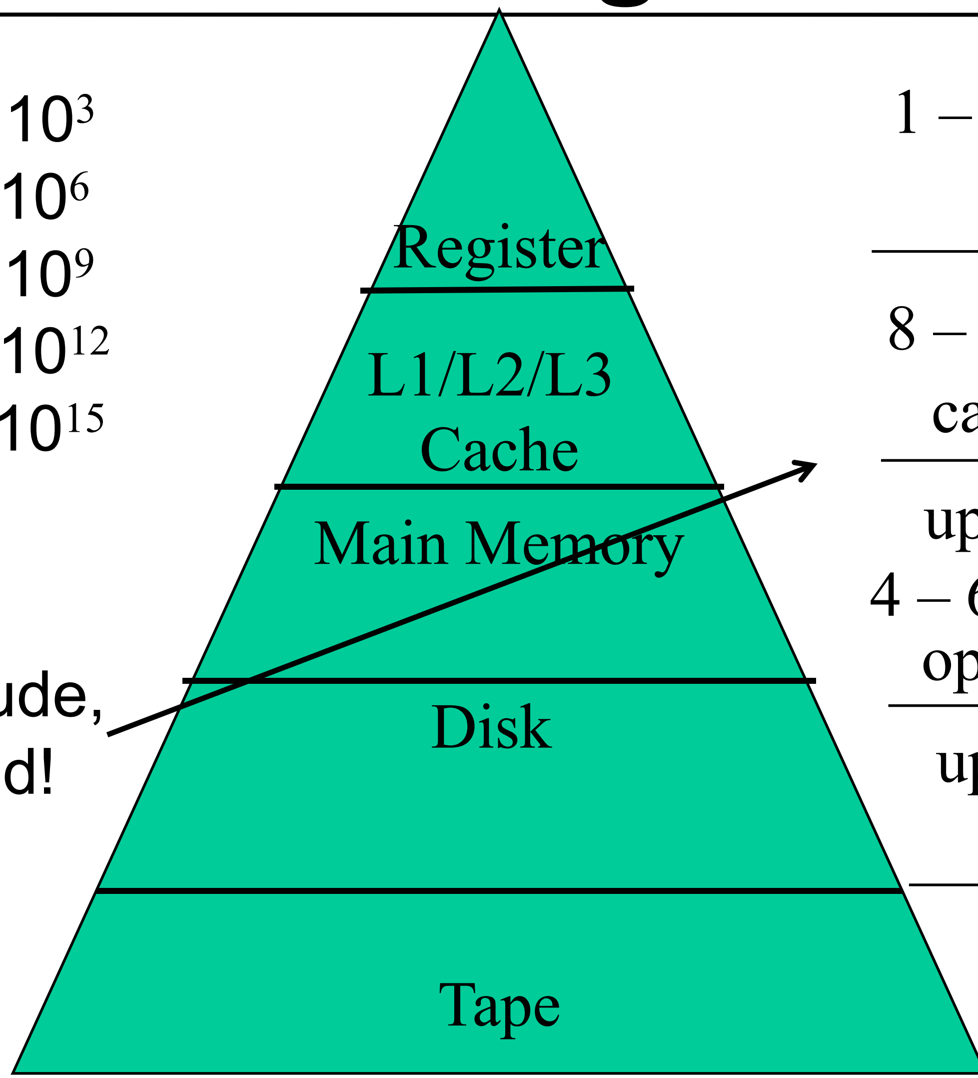
Overview: Storage Hierarchy



Overview: Storage Hierarchy

1 K (Kilo) $= 10^3$
1 M (Mega) $= 10^6$
1 G (Giga) $= 10^9$
1 T (Tera) $= 10^{12}$
1 P (Peta) $= 10^{15}$

Rough magnitude,
rapidly outdated!



1 – 8 Byte/Register
Compiler

8 – 128 Byte/Cache
cache-controller

upper GB-range,
4 – 64 KB block size
operating system

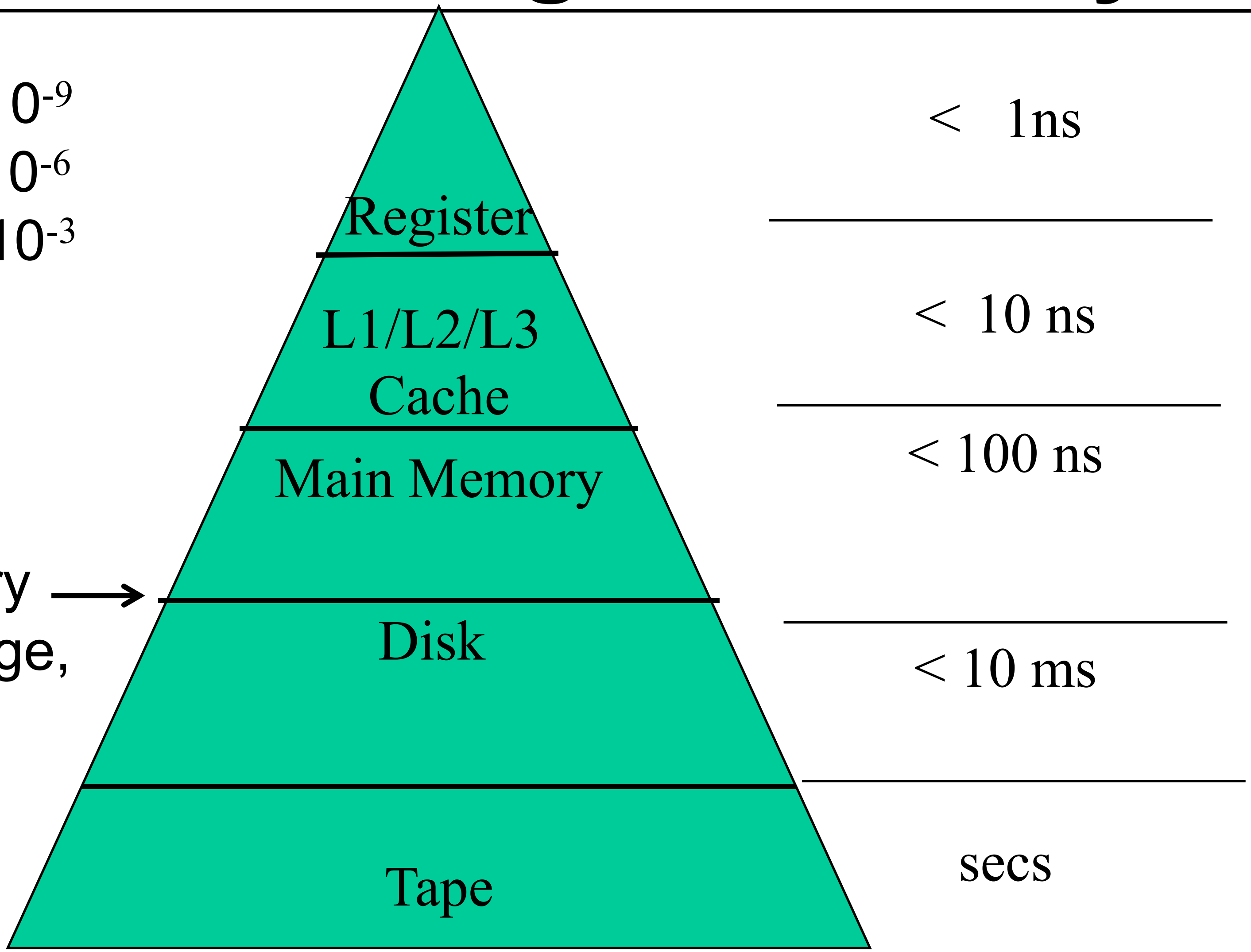
upper TB-range
user

PB-range
user

Overview: Storage Hierarchy

1 n (nano) = 10^{-9}
1 μ (micro) = 10^{-6}
1 m (milli) = 10^{-3}

(Flash-Memory →
Lower TB-range,
< 100 μs)



Overview: Storage Hierarchy

Idea (1min)

Building (10min)

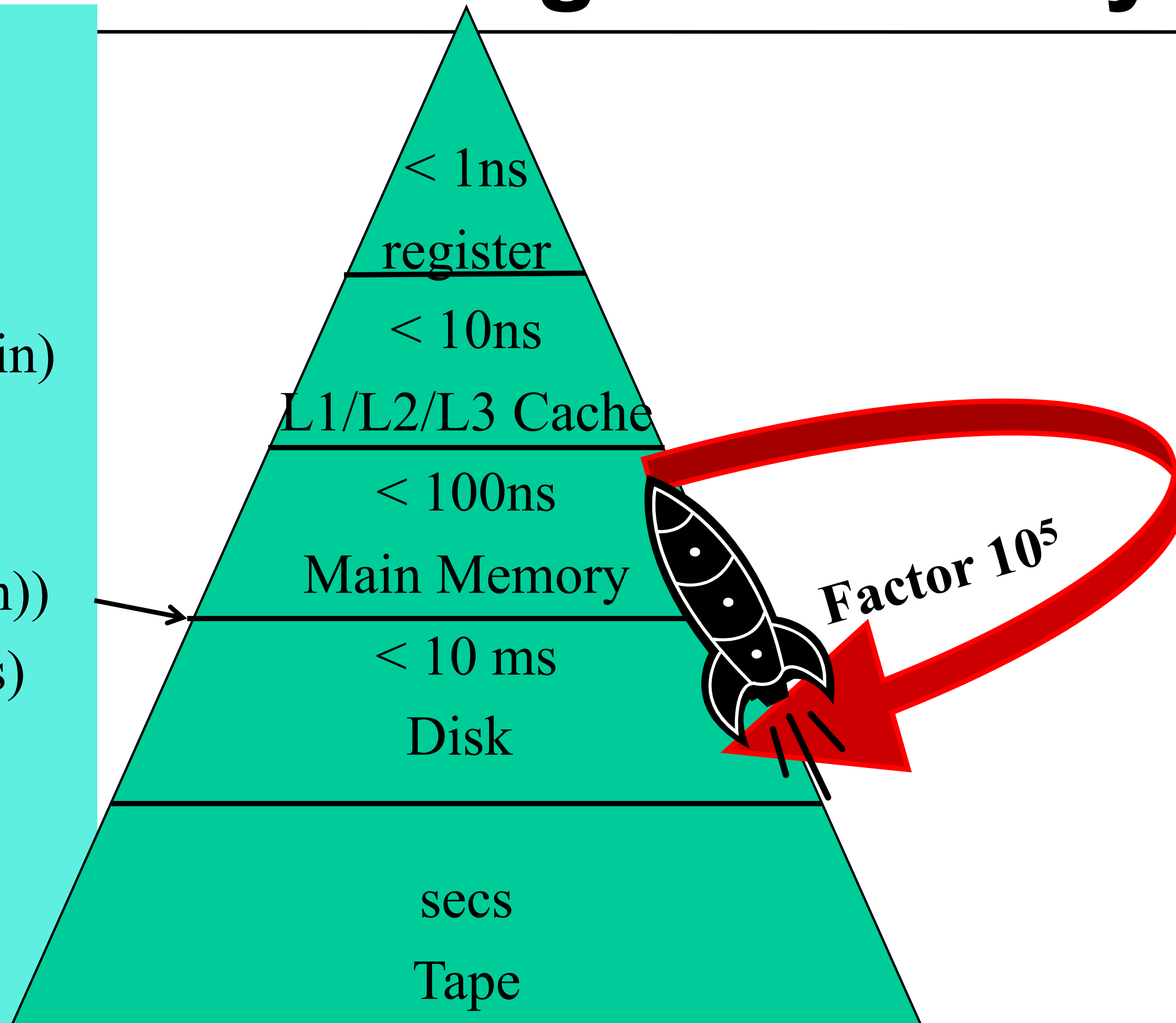
City (1.5h)

(Mars (2 month))

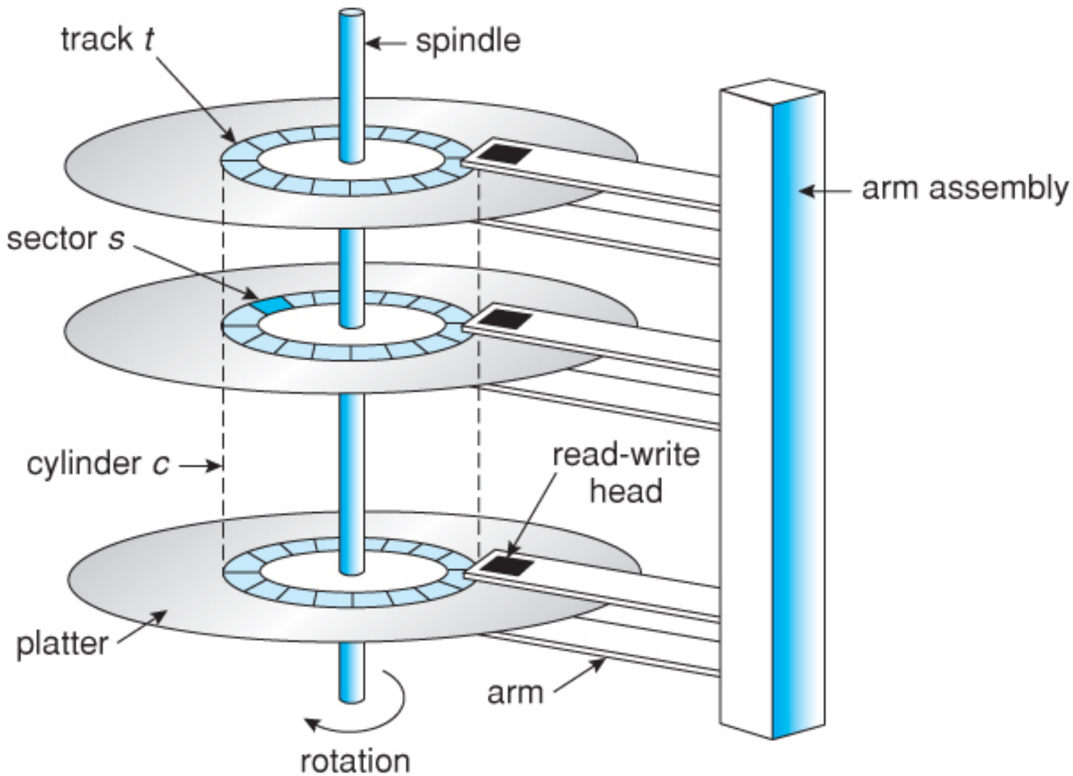
Pluto (2 years)

Andromeda

(2000 years)



Magnetic Disks



Sector:
Unit to read or write, 1-8 KB

Track:
Formed of sectors of equal size

Read Data from Disk

Seek Time: positioning of arm and head to the track

Latency: Rotation to the beginning of the sector ½ rotation of the disk (on average)

Transfer Time: Transfer sector from disk to main memory

Increasing range of disk transfer rates from the inner diameter to the outer diameter of the disk

Random versus Chained IO

Random I/O

- Every time positioning of the arm, head, and rotation

Chained IO

- Positioning, then read sectors track-wise

Chained IO is one to two maginitude faster than random I/O

→ **Need to consider this gap in algorithms!**

Random versus Chained IO

Time to read **1000 blocks** of size **8 KB**?

t_s:4ms; t_r:2ms; t_{tr}:0.1ms; $t_{\text{track-to-track seek time}}$:0.5ms (63 sectors per track)

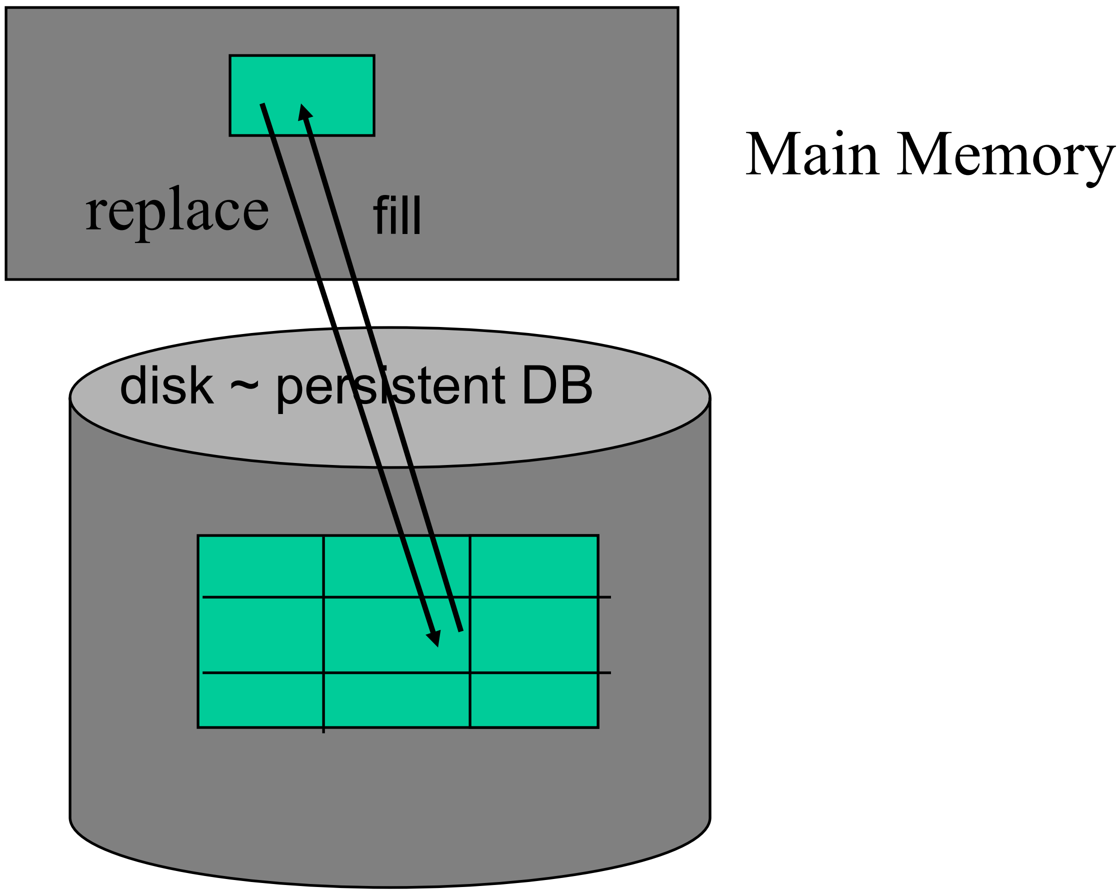
Random access:

$$t_{rnd} = 1000 * t$$
$$= 1000 * (t_s + t_r + t_{tr}) = 1000 * (4 + 2 + 0.1)$$
$$= 1000 * 6.1 = \mathbf{6100\ ms}$$

Sequential access:

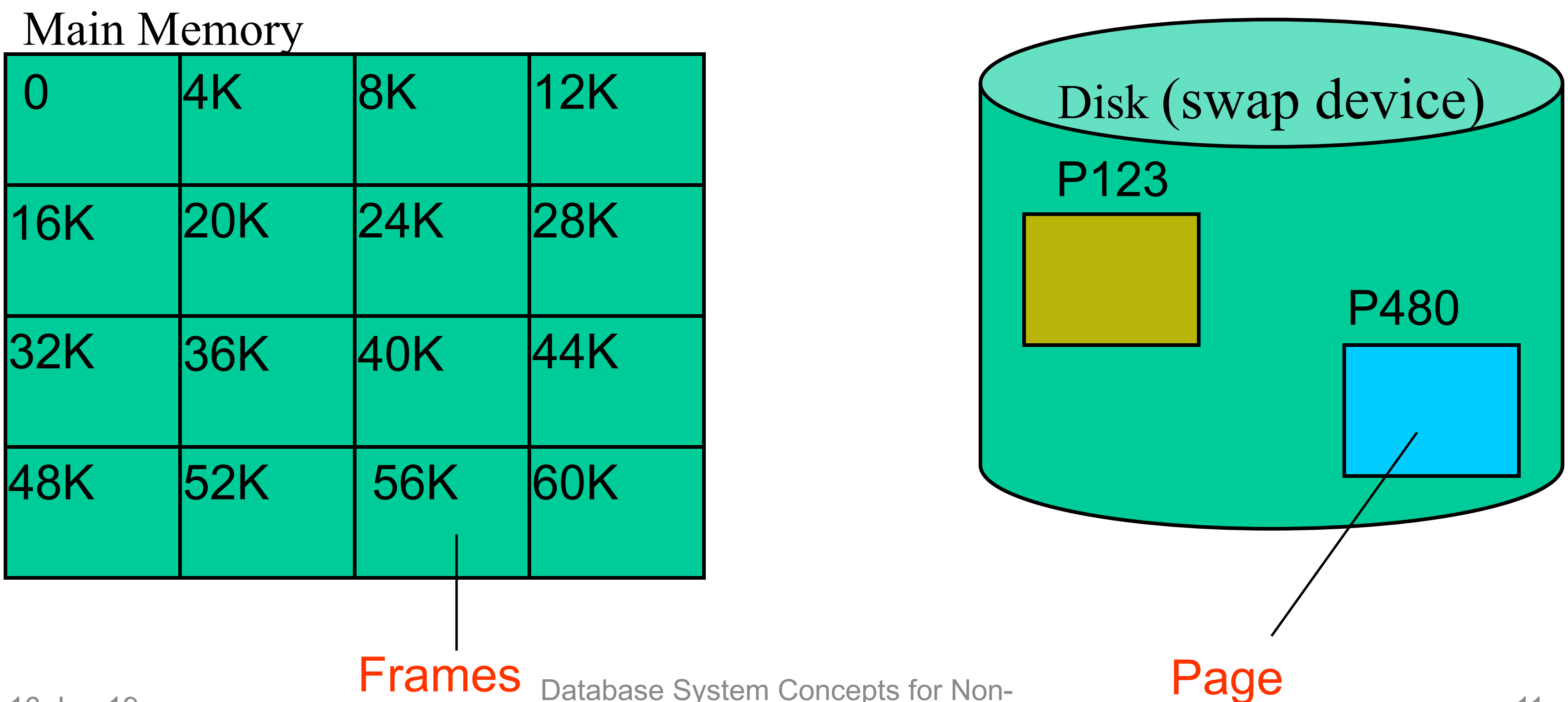
$$t_{seq} = t_s + t_r + 1000 * t_{tr} + N * t_{\text{track-to-track seek time}}$$
$$= t_s + t_r + 1000 * 0.1 + (16 * 1000)/63 * 0.5$$
$$= 4 + 2 + 100 + 126 = \mathbf{232\ ms}$$

Buffer Management

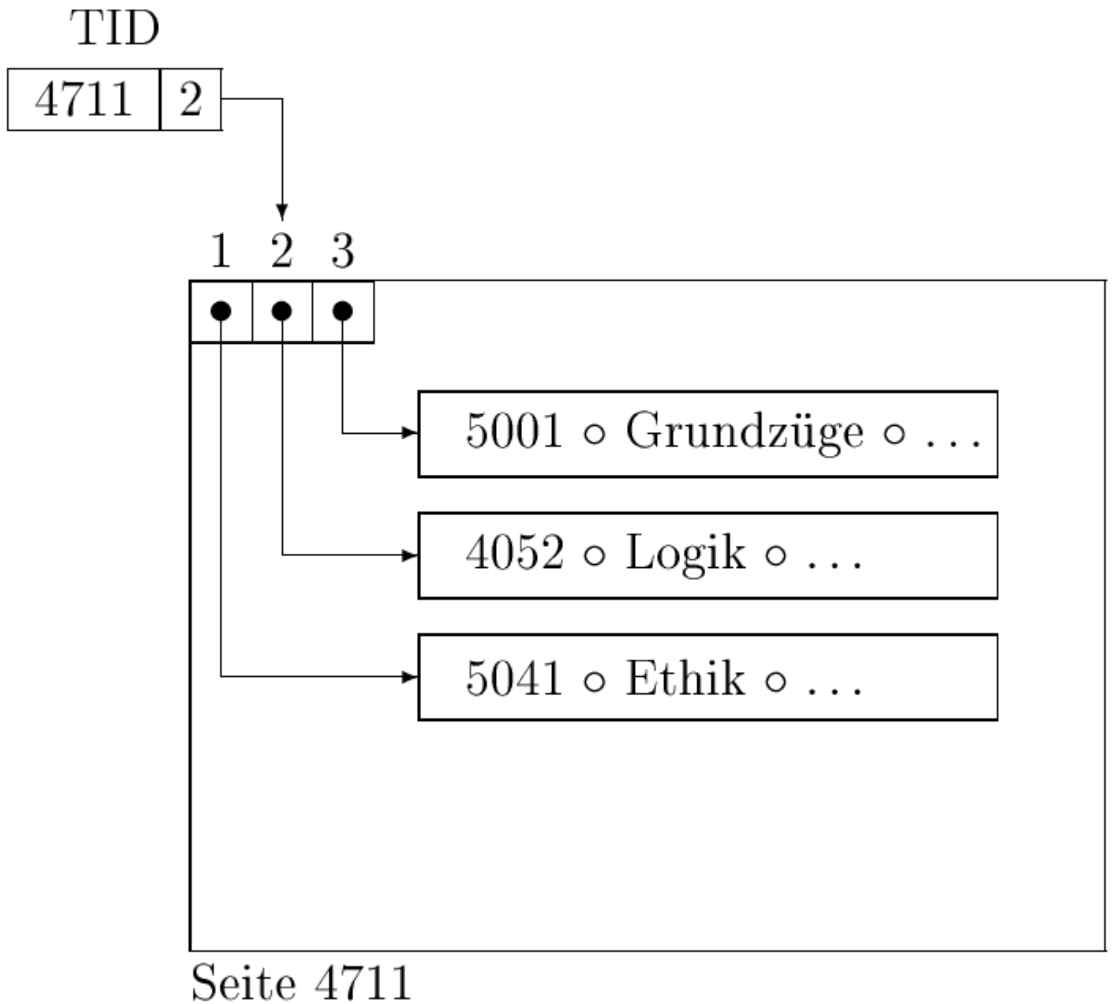


Fill and replace pages

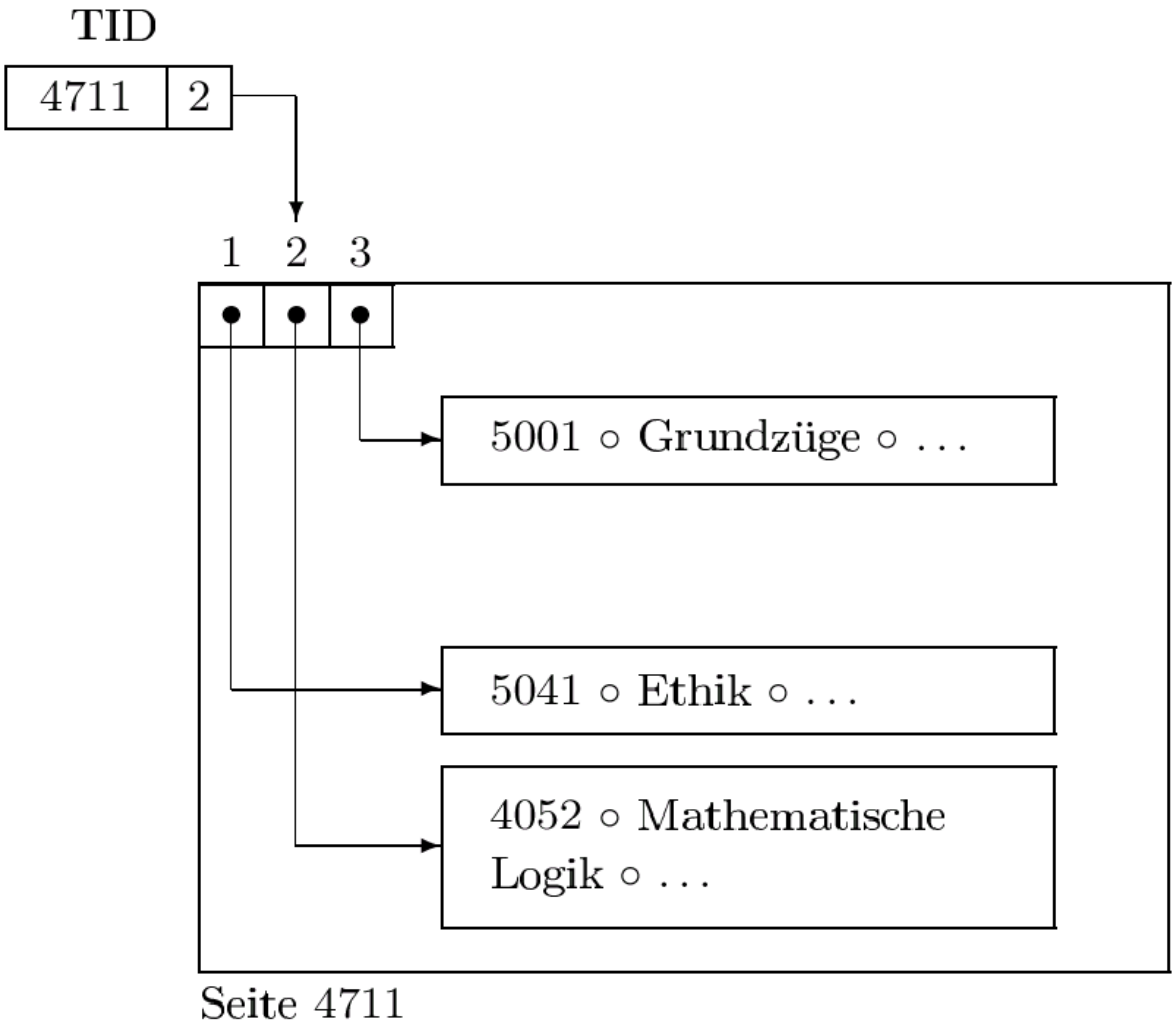
- System buffer is divided in frames of equal size
- A frame can be filled with one page (block, sector)
- Overflow pages are swapped on disk



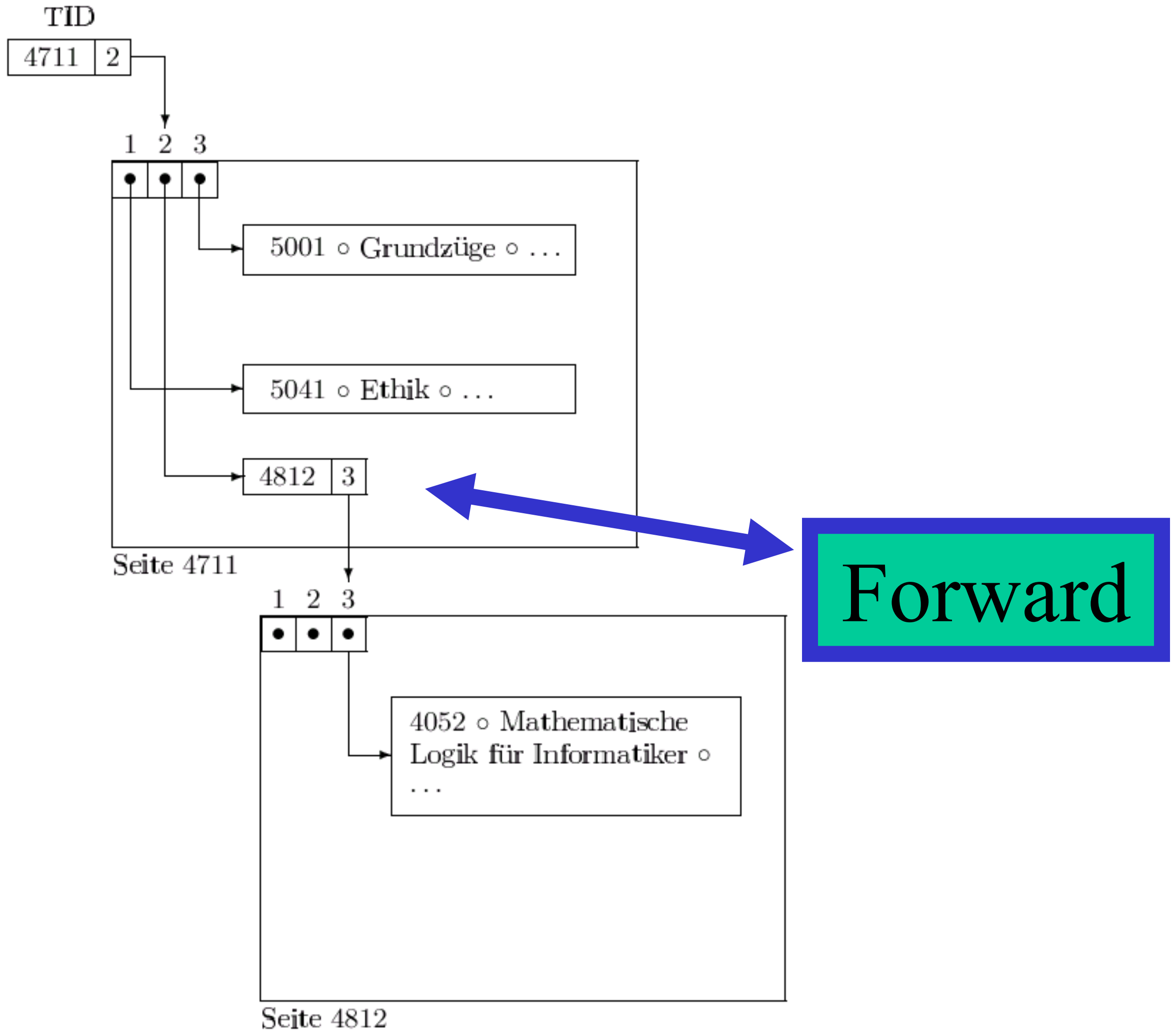
Addressing tuples on disk



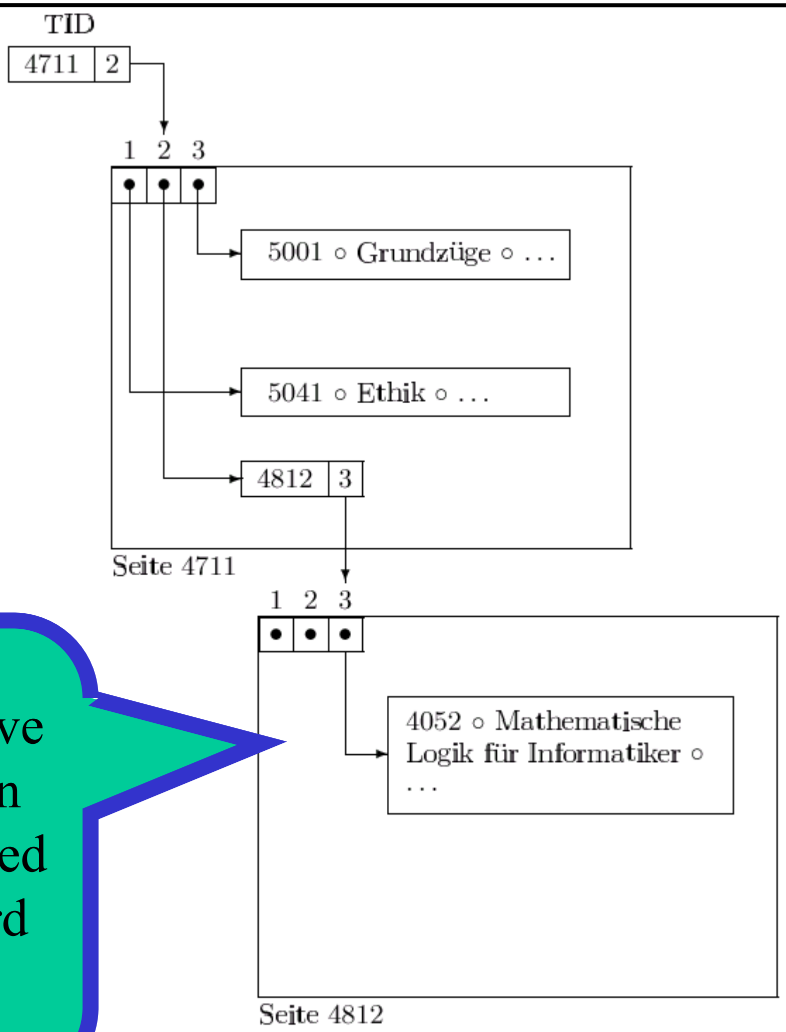
Moving within a page



Moving from one page to another



Moving from one page to another



With the next move the „Forward“ on page 4711 is altered (no more Forward to page 4812)

Data Transfer

Simple query execution:

select * **from** students **where** studNr=26120;

Get one tuple after the other to the main memory and evaluate predicates.

→ Most expensive way ☹

→ Mostly only a small fraction of the tuples fulfills the query

Index Structures

- Index structures are used to keep the data volume to be transferred from disk to main memory small
- Only that part of the data which is really needed to answer the query is transferred
- Two main indexing methods:
  - Hierarchical (trees)
  - Partitioning (Hashing)

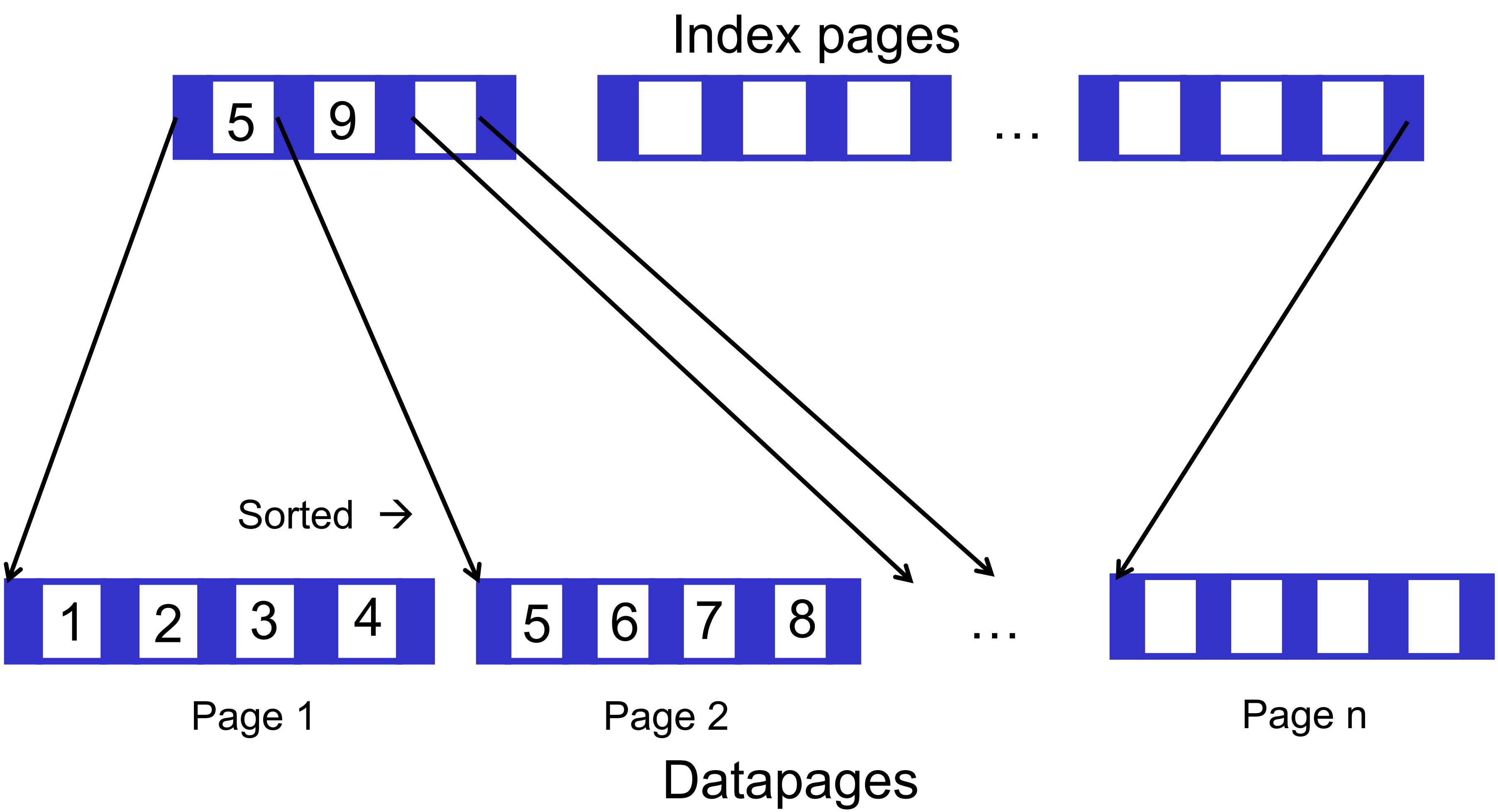
Hierarchical Indexes

We consider two hierarchical index structures:

- ISAM (Index-Sequential Access Method)
- B-Trees

- ISAM is the predecessor of B-Trees
- Main idea: sort tuples on the indexed attribute and create an index file on it
- Similar to a thumb index in a book

Example



Index pages
5
9
…
Sorted →
1
2
3
4
5
6
7
8
…
Page 1
Page 2
Page n
Datapages

Example cont.

- Student with student number 13542 is searched
- During query execution you go through the **index pages** and look for the place where 13542 fits
- From there you get the referenced **data page**
- **Advantage**: Number of index pages is much less than number of data pages, i.e. you save I/O
- You can also answer **range queries**, e.g. all StudNr between 765 and 1232: find as a start the first fitting data page for 765 and from there on you can go **sequentially** through the data pages until StudNr 1232

Problems with ISAM

Simple and fast search but **maintenance of index** is expensive:

- Inserting a tuple in a full data page: need to make room in **dividing data page into two** → we need to keep the sorting
- This creates a **new entry** on an **index page**
- Inserting an entry in a full index page leads to **shifting the entries** to make room
- Although the number of index pages is smaller than the number of data pages **going through the index pages** can nevertheless **take a long time**

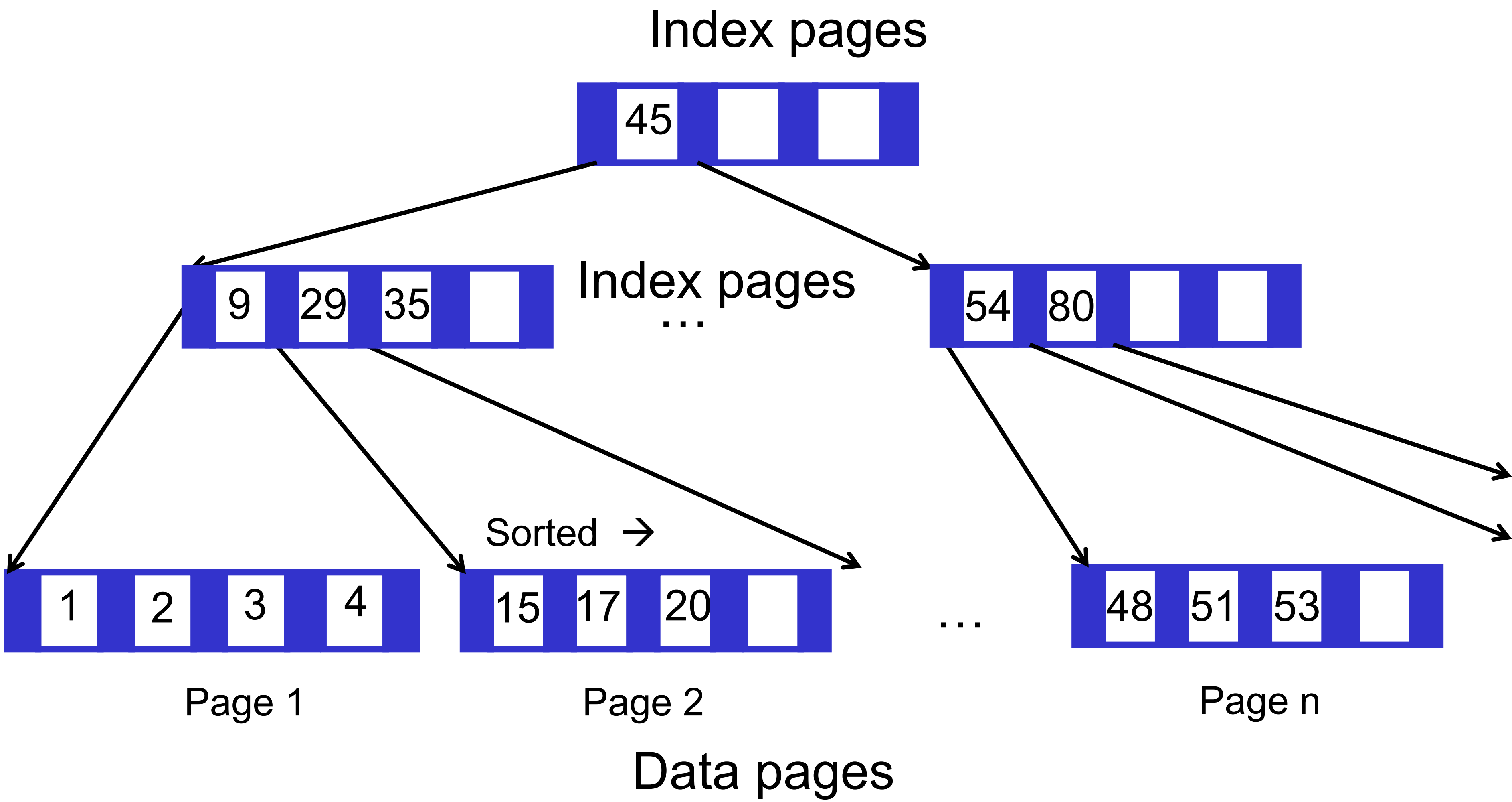
Advancement

Idea:

Why not have **index pages for the index pages**?

→ This is in principle the idea of a **B-Tree**

B-Tree


Index pages
45
Index pages
…
9
29
35
54
80
Sorted →
1
2
3
4
15
17
20
…
48
51
53
Page 1
Page 2
Page n
Data pages

B-Trees

Trees in Informatics

… have nodes

… have edges

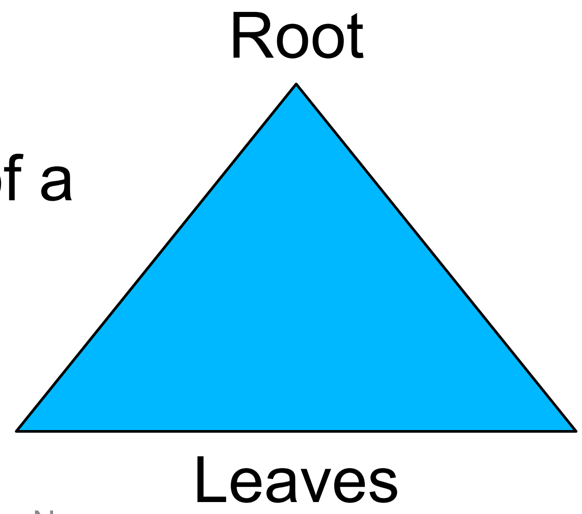
… have a root (at the top!)

… have leaves (at the bottom!)

… are often balanced

(otherwise in extreme cases rather a chain)

Schematic depiction of a balanced tree:



Properties of a B-Tree

B-Tree of degree *i* has following properties:

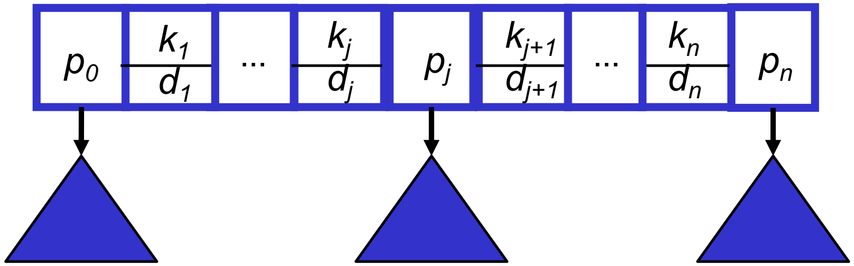
- Every path from the root to a leaf has the same length
- Every node – except the root – has at least *i* and at most *2i entries* (in the example above *i*=2)
- Entries in every node are sorted
- Every node – except the leaves – with *n* entries has *n+1* children

Properties of a B-Tree

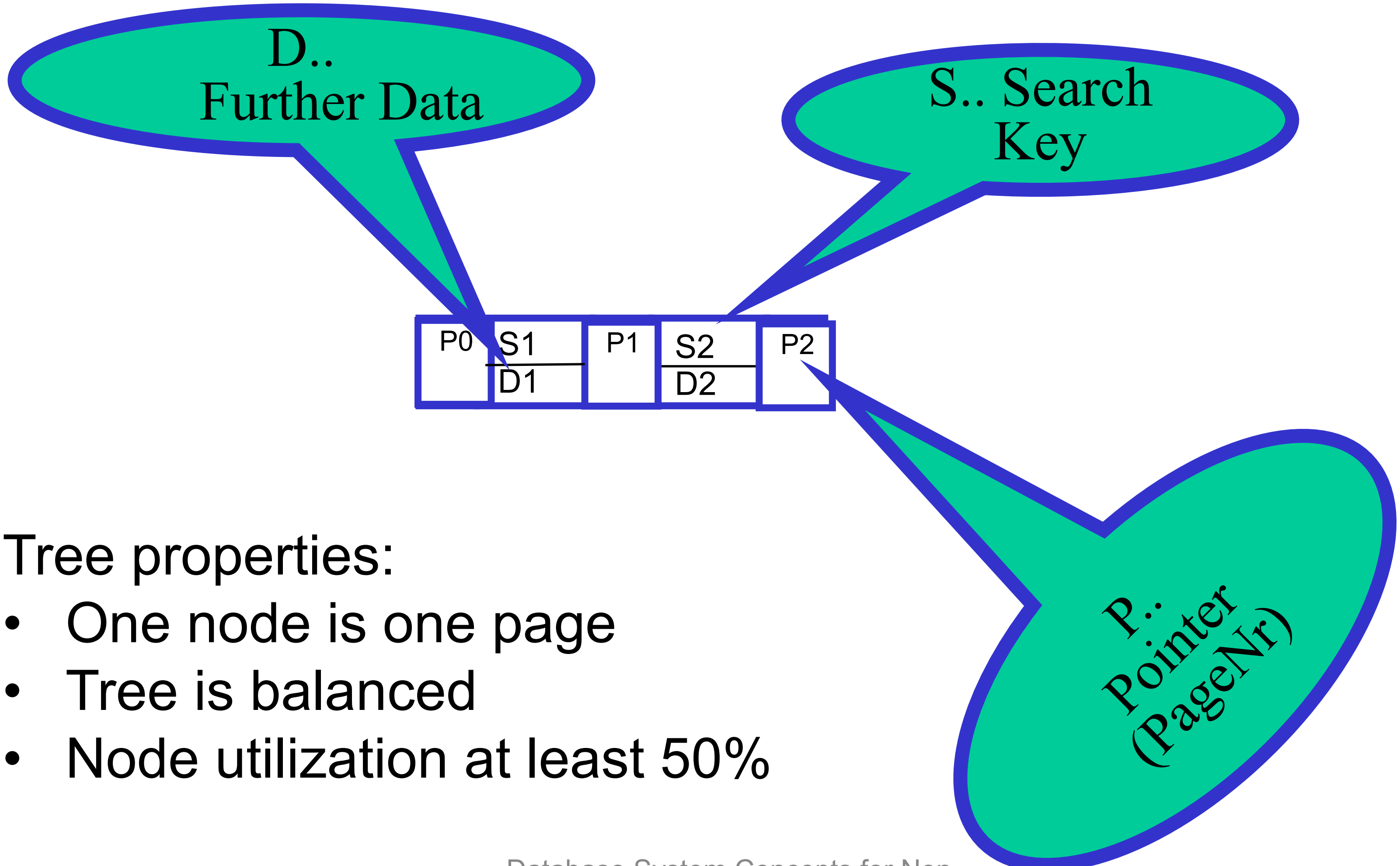
- *Let*
  $p_0, k_1, p_1, k_2, \ldots k_n, p_n$
  be entries in a node (p_j are page identifier, k_j keys)

Then the following holds:

1. Sub-tree in p_0 contains only keys smaller than k_1
2. p_j has a sub-tree with keys between k_j and k_j+1
3. Sub-tree being referenced by p_n contains only keys greater than k_n



Node Structure



Tree properties:

- One node is one page
- Tree is balanced
- Node utilization at least 50%

Insert Algorithm

1. Find the proper leaf node to insert new key
2. Insert key there
3. If node full
   i. Divide node into two and extract median
   ii. Insert all keys smaller than median into left node, all keys greater than median into right node
   iii. Insert median in parent node and adapt pointers
4. If parent node full
   i. If root node then create new root node, insert median, and adapt pointers
   ii. Otherwise repeat 3. with parent node

Delete Algorithm

Read the literature or example on lecture website

Gradual Assembly of a B-Tree of Degree i=2

See:
https://db.in.tum.de/teaching/ws1819/DBSandere/BTreeExample.pdf

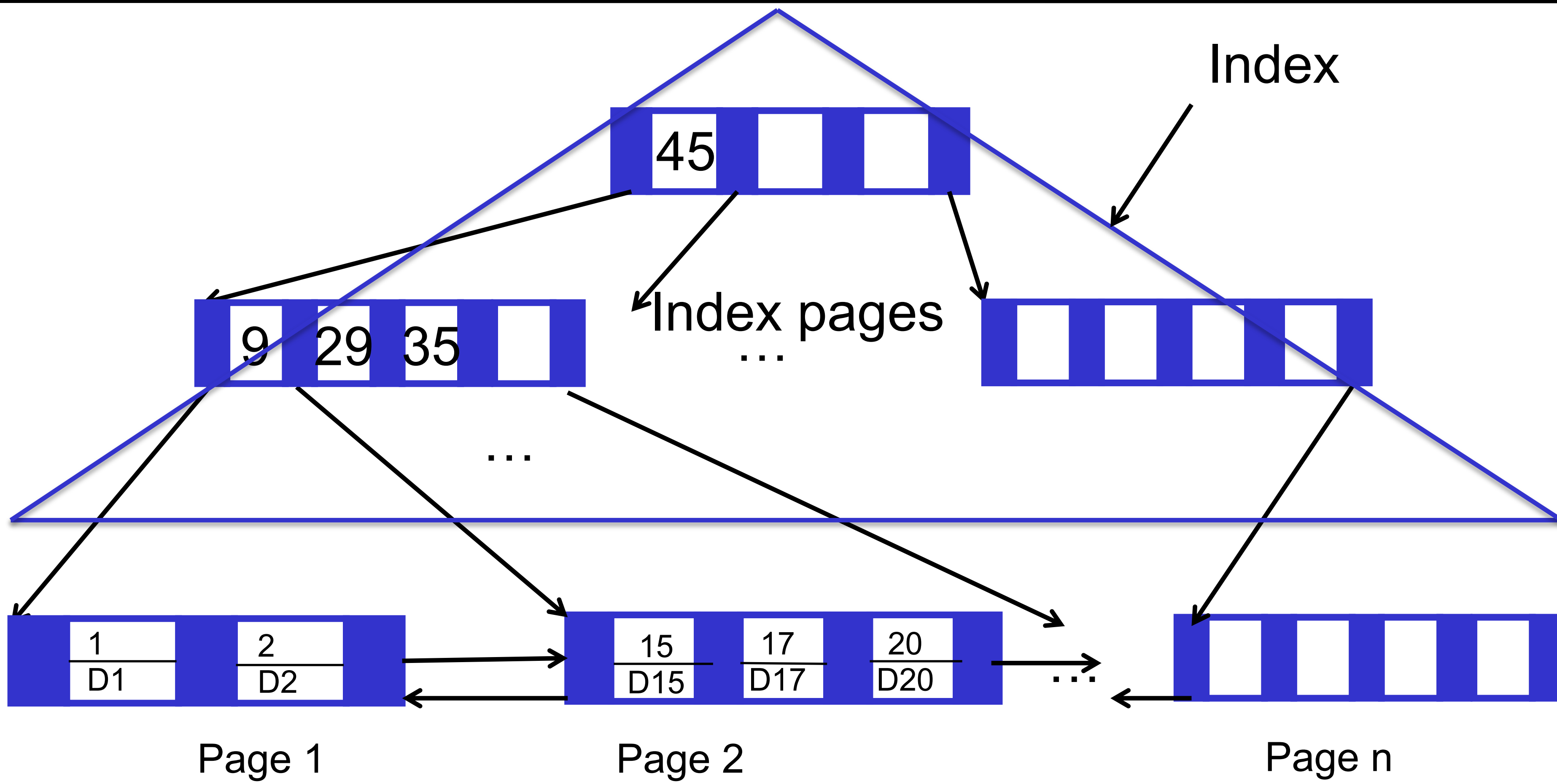
In the internet there are a number of animation programs for B-Trees – **no warranty!**

https://www.cs.usfca.edu/~galles/visualization/BTree.html
looks quite good, but uses a different notation for the maximal node size and does not handle node underflows.

B+-Trees

- Performance of a B-Tree heavily depends on height: on average $\log_k(n)$ page accesses to read one data element
  (k=degree of branching, n=number of indexed data elements)
  → preferably high degree of branching of the inner nodes
- Storing data in the inner nodes reduces branching degree
- B+-Trees only store reference keys in inner nodes – data itself is stored in leaf nodes
- Usually leaf nodes are bidirectionally linked in order to enable fast sequential search

Structure B+-Tree



Data pages, sorted, bidirectionally linked

Prefix B+-Trees

- Further Improvement by use of prefixes of reference keys, e.g. with long strings as keys
- You only have to find a reference key which separates the left and the right sub-tree:
  - Disestablishment <= E < Incomprehensibility
  - Systemprogram <= ? < Systemprogrammer

Several Indexes on the same Data

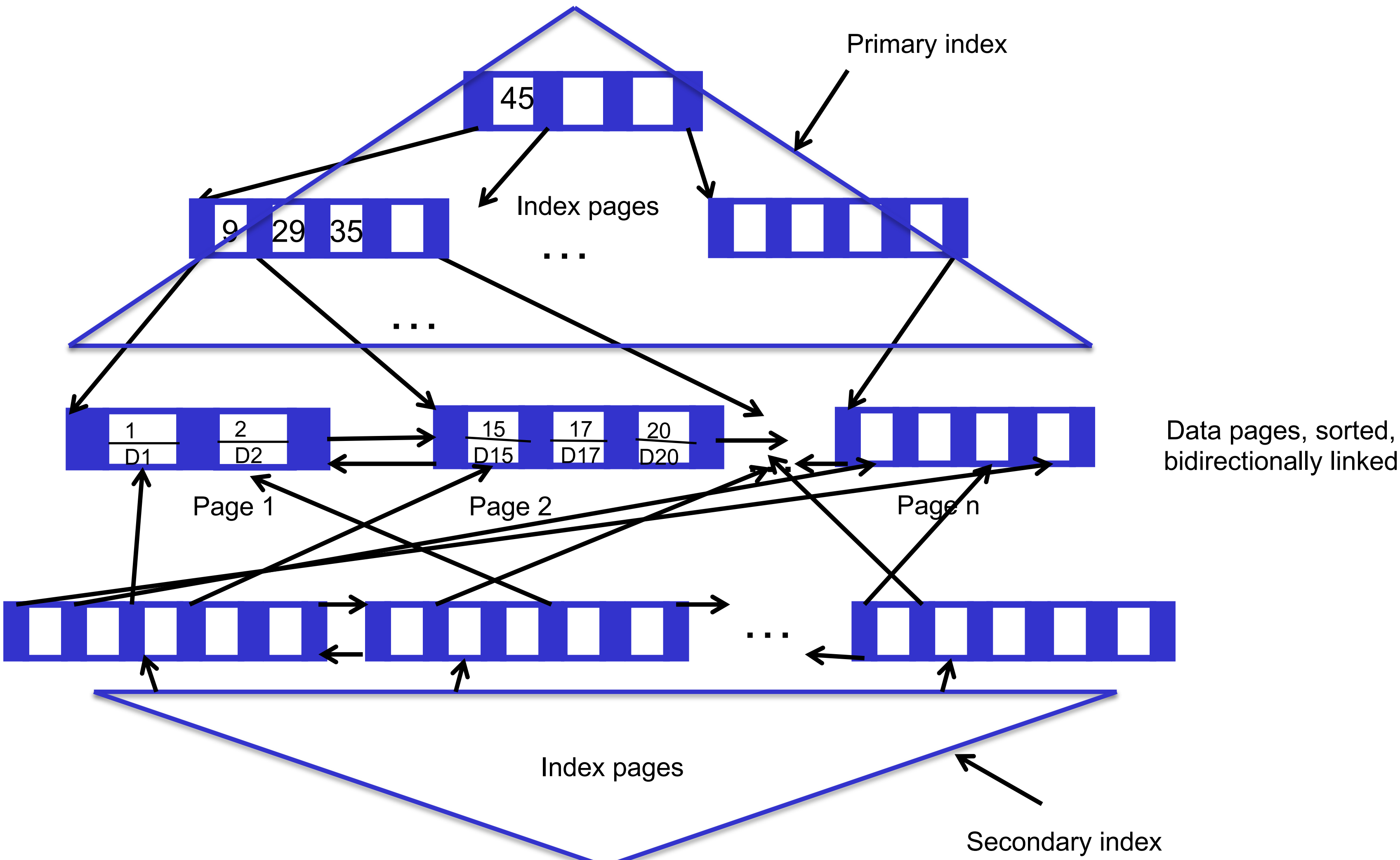
Primary index – Secondary index

Students		
StudNr	Name	Semester
25403	Jonas	12
29120	Theophrastos	2
29555	Feuerbach	2
27550	Schopenhauer	6
⋮	⋮	⋮

When

- Index on StudNr?
- Index on Name?
- Index on Semester?

Secondary indexes



DDL: Create Index

```
CREATE [UNIQUE] INDEX index_name
ON table_name (column_name1 [, column_name2, …])
```

Example:

```
CREATE INDEX full_name
ON Person (Last_Name, First_Name)
```

Partitioning
What is Hashing?

- (to hash = zerhacken)

- Storing tuples in a defined memory area

- Hash function: mapping tuples (key values) to a fixed set of function values (memory area)

- Optimal hash function:
  - injective (no identical function values for different arguments)
  - surjective (no waste of memory)

- Typical hash function h: $h(x) = x \bmod N$
  set of function values thereby $\{0, \ldots, N-1\}$

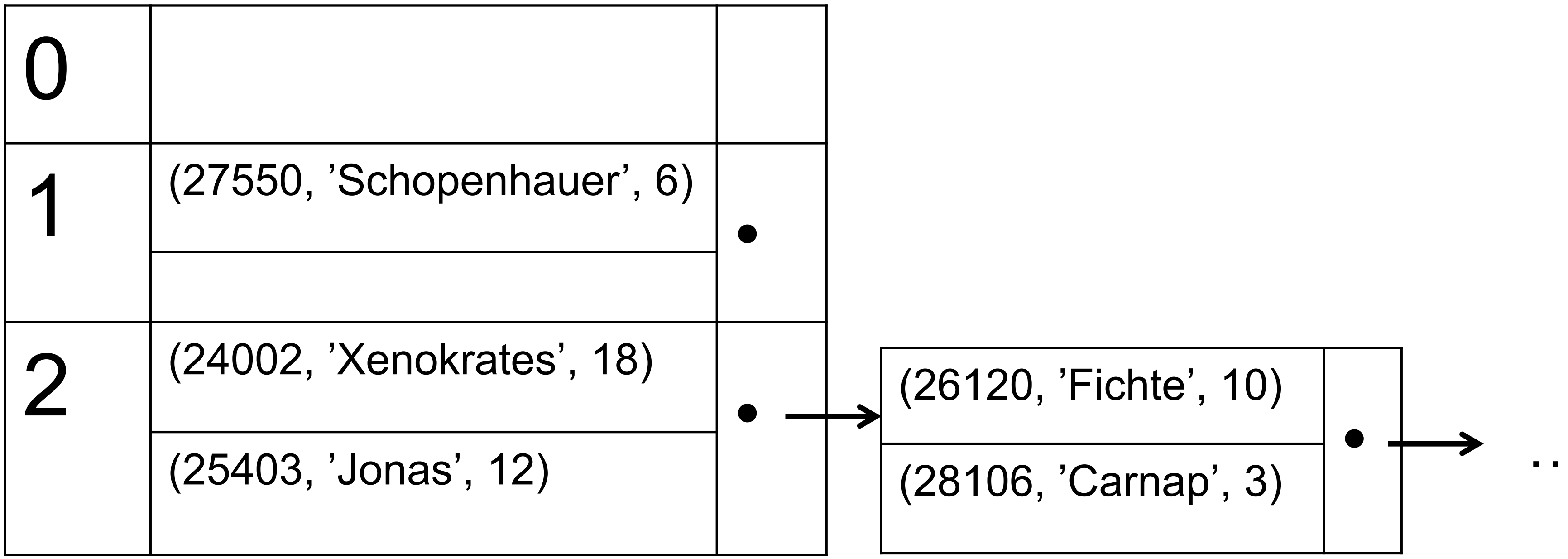
Example Hashing

- Example hash function *h(x) = x mod 3*

0	
1	(27550, ’Schopenhauer’, 6)
2	(24002, ’Xenokrates’, 18)
	(25403, ’Jonas’, 12)

Collisions

Collision handling



Inefficiently with not forseen quantity of data

Way out: extensible (dynamic) Hashing

→ further indirection via directory

Advantages / Disadvantages Hashing

+ Few accesses to external storage
  constant cost: O(1), generally 1-2
+ Simple implementation

- Collision handling necessary
- Pre-allocation of memory area
- Not dynamic resp. only with adjustment
- **No range queries, only point queries**

Interleaved Storing

Seite P_i

2125 ∘	Sokrates	∘ C4 ∘	226 •
5041 ∘	Ethik	∘ 4 ∘	2125 •
5049 ∘	Mäeutik	∘ 2 ∘	2125 •
4052 ∘	Logik	∘ 4 ∘	2125 •
2126 ∘	Russel	∘ C4 ∘	232 •
5043 ∘	Erkenntnistheorie	∘ 3 ∘	2126 •
5052 ∘	Wissenschaftstheorie	∘ 3 ∘	2126 •
5216 ∘	Bioethik	∘ 2 ∘	2126 •

Seite P_{i+1}

2133 ∘	Popper	∘ C3 ∘	52 •
5259 ∘	Der Wiener Kreis	∘ 2 ∘	2133 •
2134 ∘	Augustinus	∘ C3 ∘	309 •
5022 ∘	Glaube und Wissen	∘ 2 ∘	2134 •
2137 ∘	Kant	∘ C4 ∘	7 •
5001 ∘	Grundzüge	∘ 4 ∘	2137 •
4630 ∘	Die 3 Kritiken	∘ 4 ∘	2137 •
	⋮		