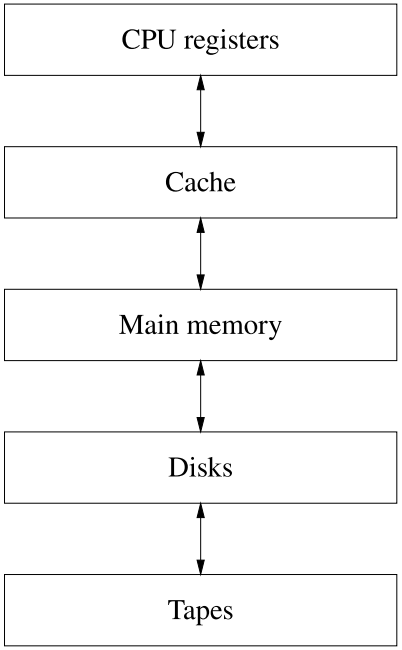
Physical Data Organization

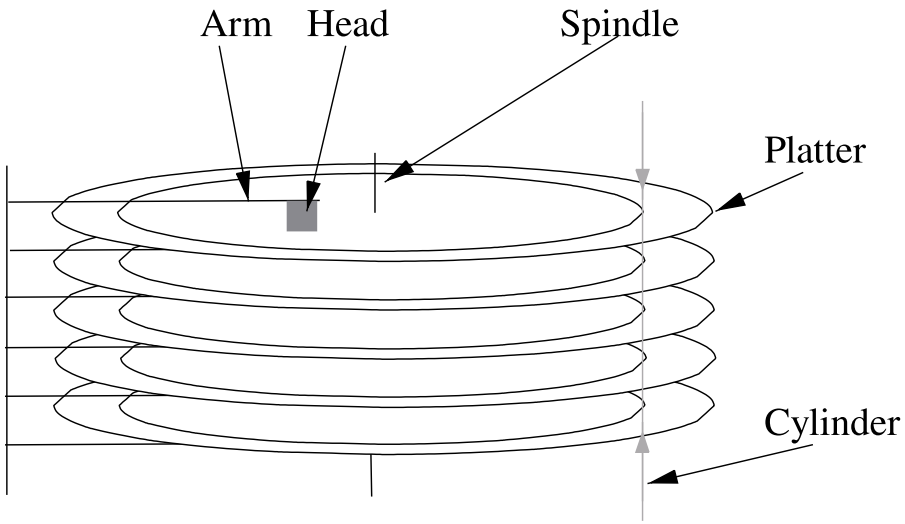
Physical Data Organization

Storage Hierarchy

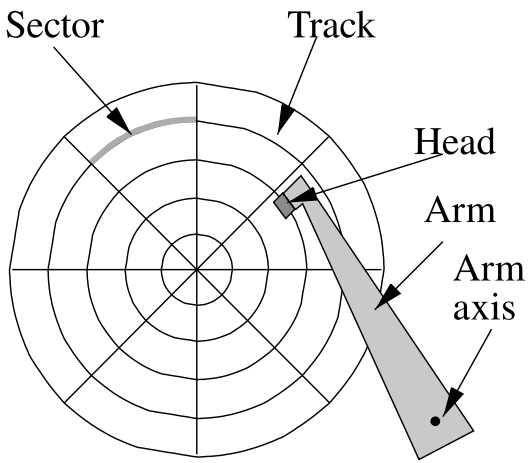
- Different layers of storage
- The higher in the hierarchy, the faster, more expensive, and smaller
- Differences are mostly orders of magnitude
- Most important for a DBMS: main memory and disks



Structure of a Disk Drive



a. Side view



b. Top view

Structure of a Disk Drive(2)

- Reading a block:
  - Positioning the head (seek time)
  - Rotation to the start of the block (latency)
  - Reading the block (transfer time)

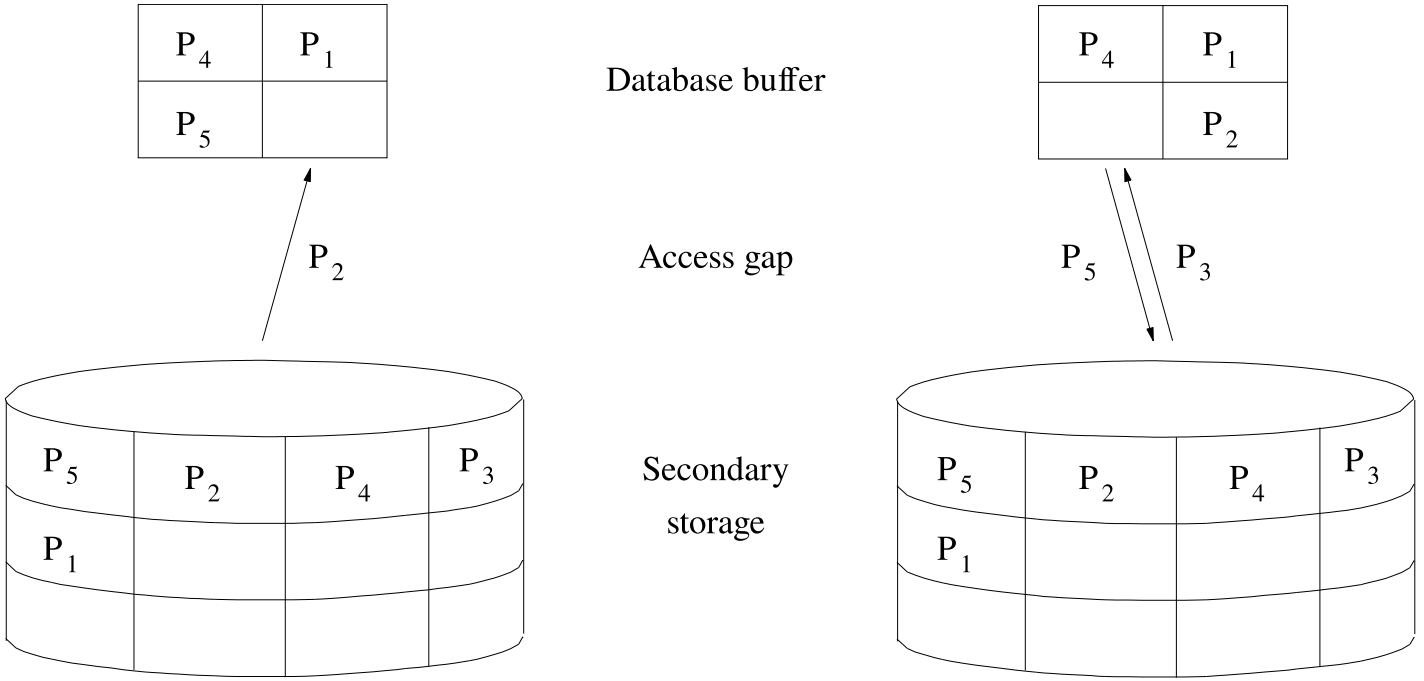
Data is usually large and valuable, therefore RAID (redundant array of inexpensive disks) is often used.

Most important RAID types:

- RAID0: striping (not a “real” RAID)
- RAID1: mirroring
- RAID0+1: striping and mirroring
- RAID3/4: striping with parity disk
- RAID5: striping with distributed parity

There are more RAID types, e.g., RAID6 for more fault tolerance.

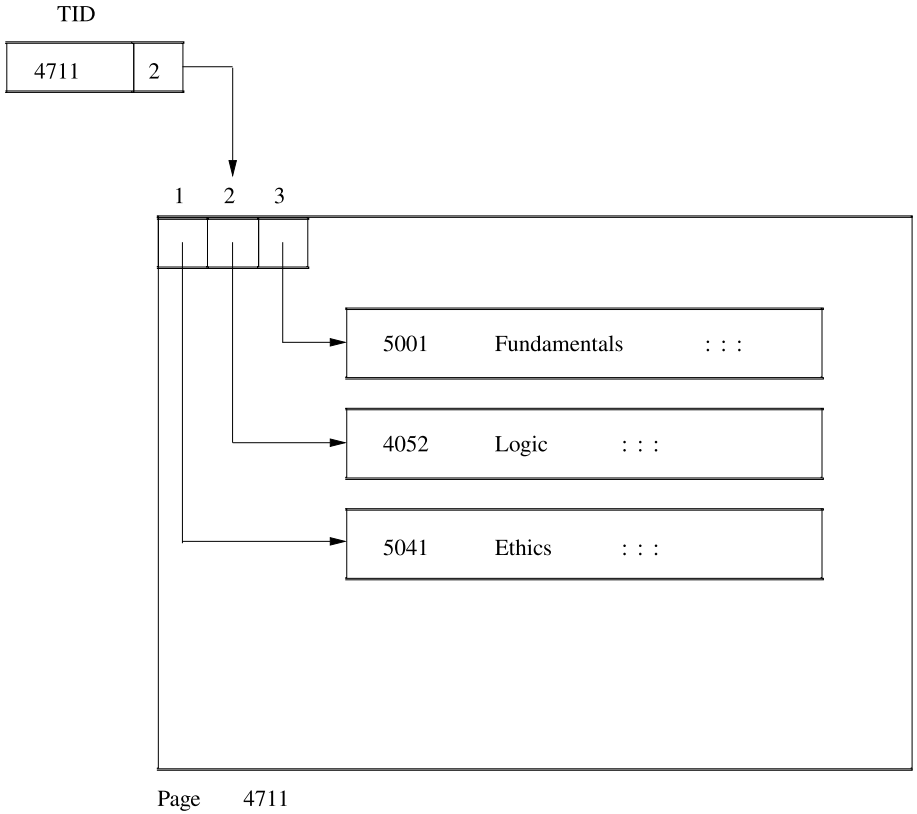
Database Buffer



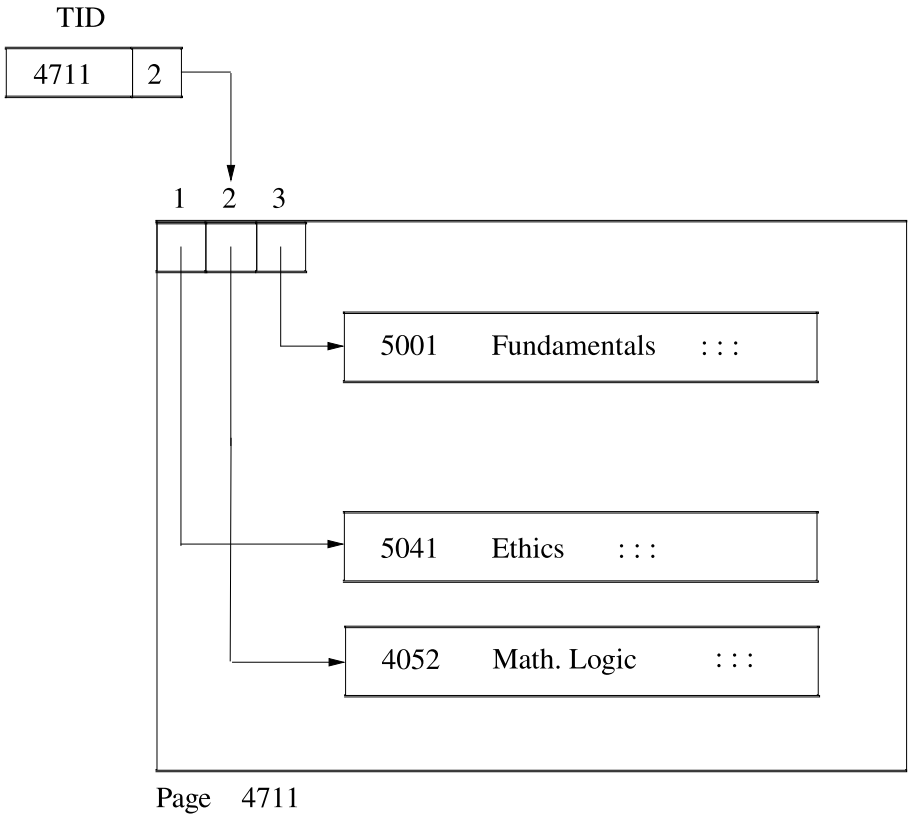
Storing Relations

- The tuples of a relation are stored on multiple pages in secondary storage
- Each page contains an internal record table with references to the tuples within the page (slotted pages)
- Tuples are referenced via *tuple identifiers* (TIDs)

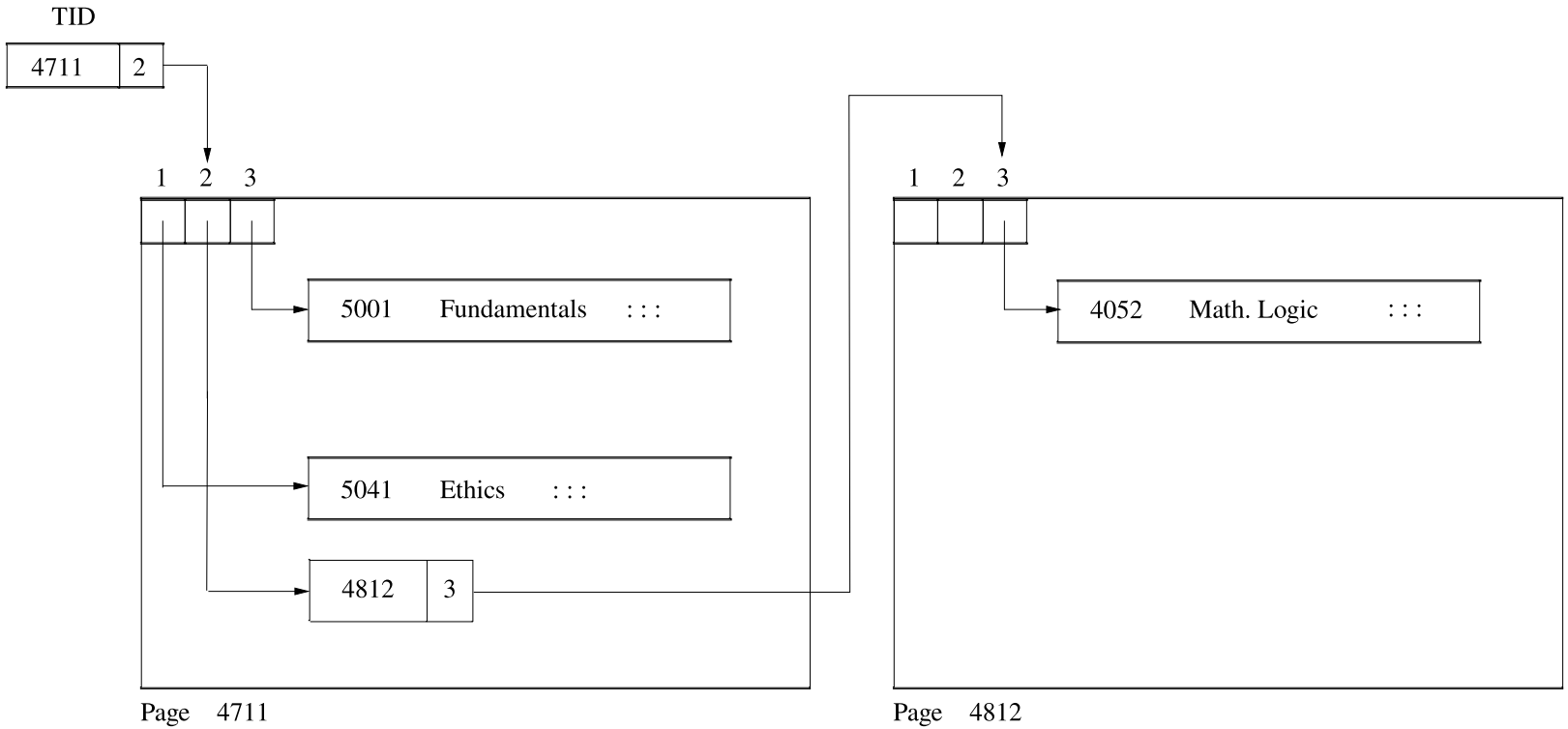
The TID Concept



Relocation within a Page



Relocation to Another Page



- Fetching the tuples of all relations into main memory one after another is the simplest way to process a query
- Unfortunately, it is also among the most expensive
- On closer inspection, one observes the following:
  - Often only a fraction of the tuples satisfies the query conditions
  - Queries often have similar predicates
  - Disks allow random access

Index Structures

- Index structures exploit these properties of queries to keep the transferred data volume small
- They allow fast associative access to the data
- Only the part of the data that is really needed to answer the query is fetched into main memory
- Two major indexing approaches
  - Hierarchical (trees)
  - Partitioning (hashing)

Hierarchical Indexes

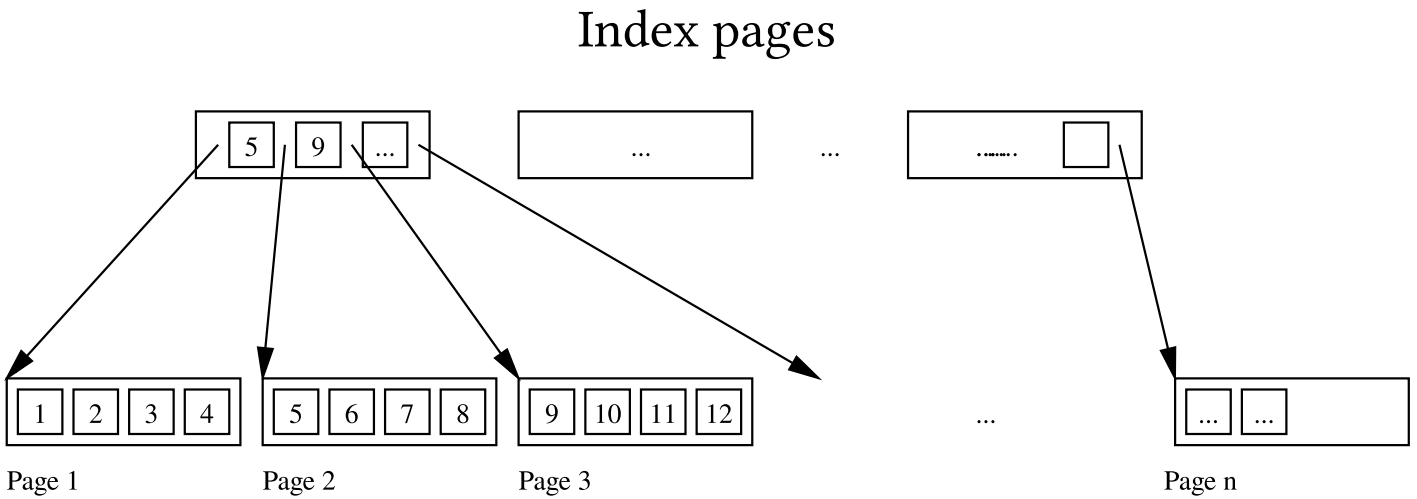
- We consider two hierarchical index structures:
  - ISAM (Index-Sequential Access Method)
  - B-trees

ISAM

- ISAM was the predecessor of B-trees
- The main idea is to sort the tuples on the indexed attribute and to create an index file on top of them
- Similar to a thumb index on the side of a book, which lets you leaf through it quickly

Example

- We are looking for the student with student number 13542
- All tuples of the relation `Students` are sorted by StudNr



Example(2)

- During query processing, one goes through the index pages and looks for the position where 13542 fits
- From there, the referenced data page is fetched
- Advantage: the number of index pages is usually much smaller than the number of data pages, i.e., I/O is saved
- Range queries can also be answered (e.g., when searching for all StudNr between 765 and 1232: first find the first matching data page and from there scan sequentially through the data pages up to StudNr 1232)

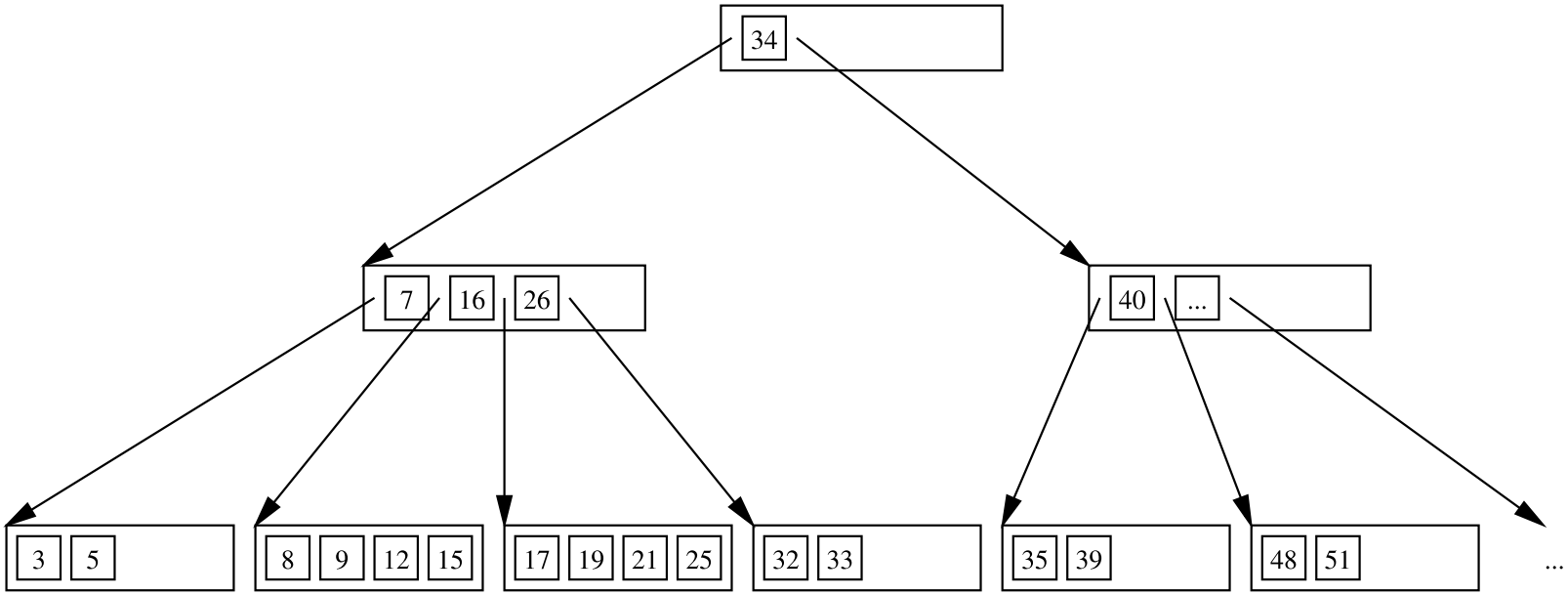
Problems with ISAM

- Although searching with ISAM is simple and fast, maintaining the index can become expensive
- If a tuple is to be inserted into a full data page, space has to be made: the data page is split into two pages (we have to maintain the sort order)
- This in turn creates a new entry on an index page
- If there is no space left on the index page either, the entries have to be shifted to make room

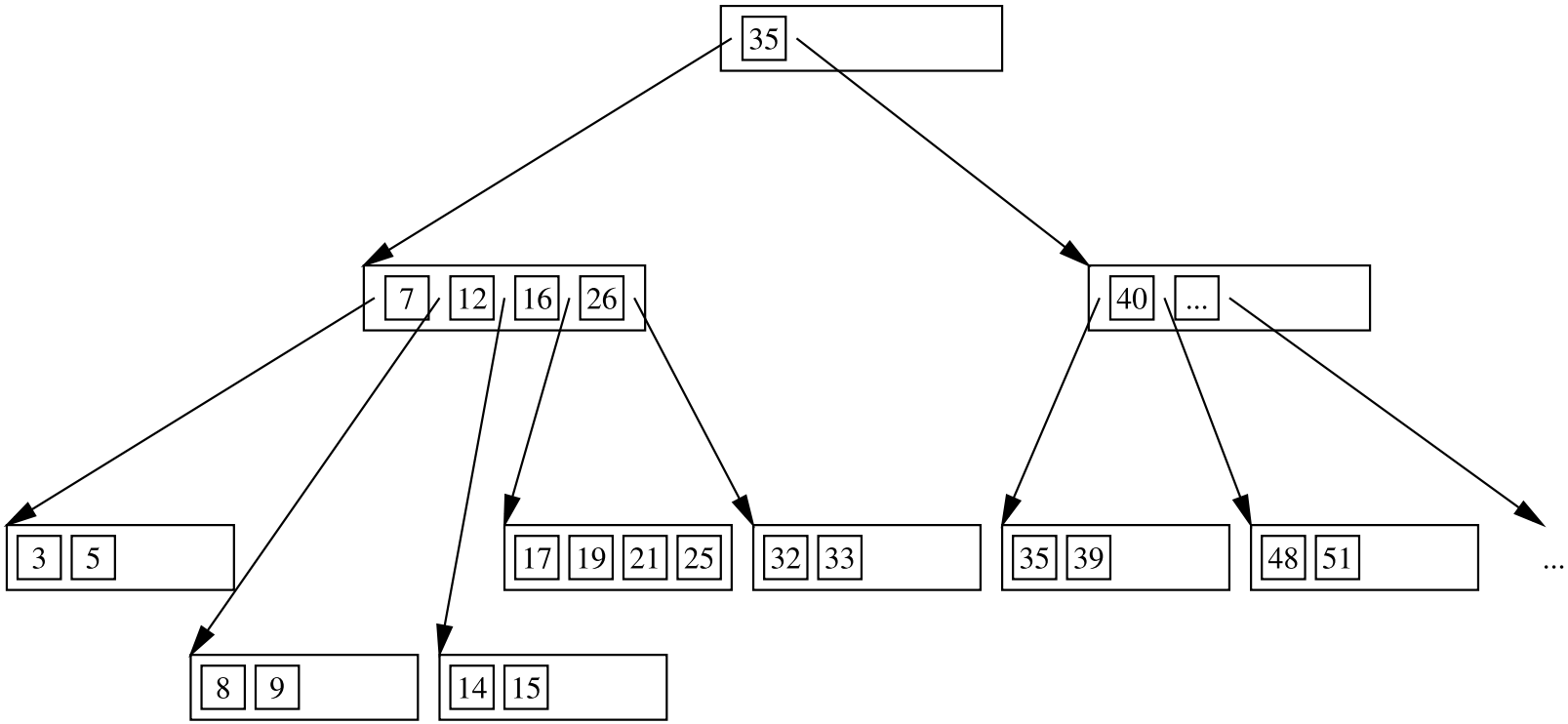
Further Problems

- Although the number of index pages is smaller than the number of data pages, scanning the index pages can still take a long time
- Idea: why not set up index pages for the index pages?
- That is, in principle, the idea of a B-tree

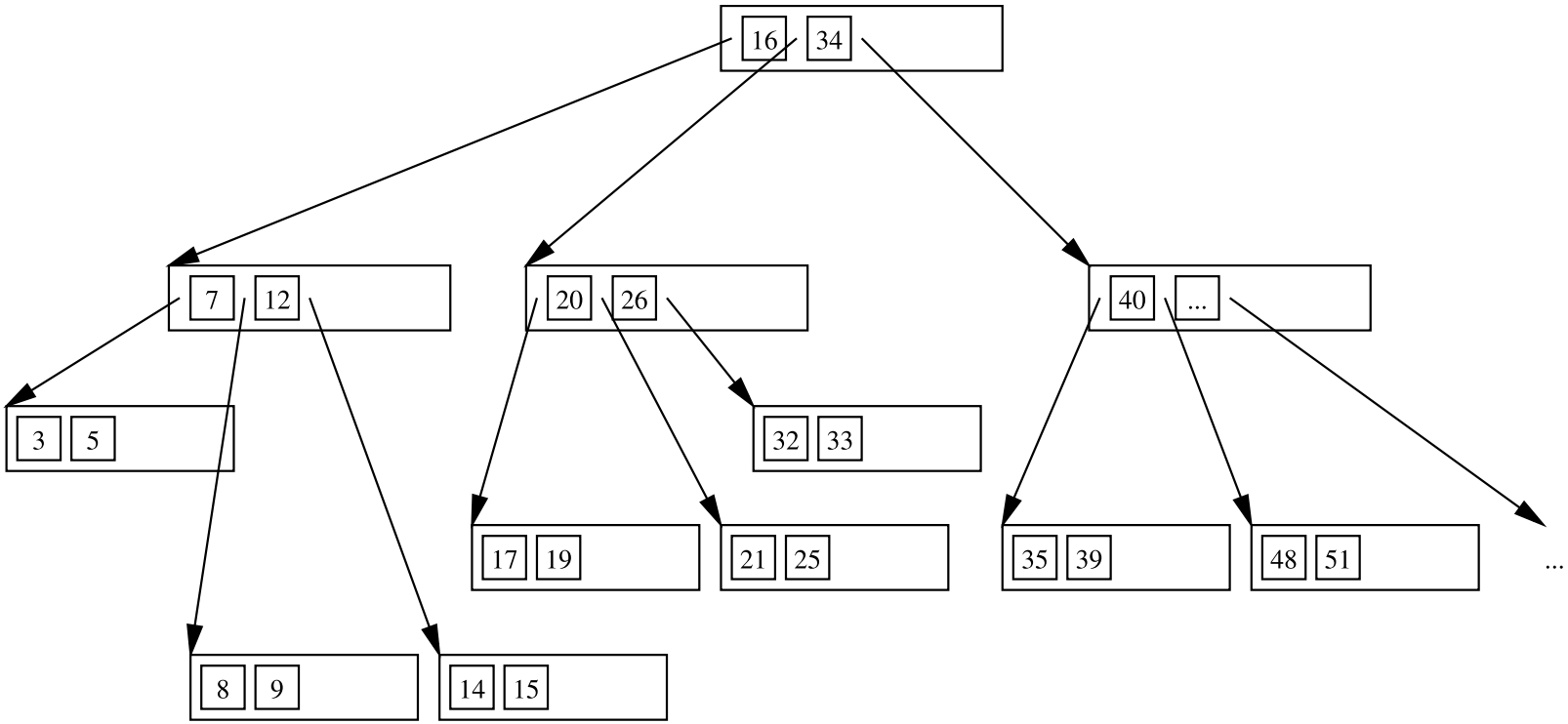
B-Tree (Example)



Inserting 14



Inserting 20



Properties of a B-Tree

- 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 $i = 2$)
- The entries in each node are sorted
- Every node (except leaves) with n entries has $n + 1$ children

Properties of a B-Tree(2)

- Let $p_0, k_1, p_1, k_2, ...k_n, p_n$ be the entries in a node (p_j are pointers, k_j keys)
- Then the following holds:
  - The subtree referenced by p_0 contains only keys smaller than k_1
  - p_j points to a subtree with keys between k_j and k_{j+1}
  - The subtree referenced by p_n contains only keys greater than k_n

Insertion Algorithm

1. Find the correct leaf node in which to insert the new key
2. Insert the key there
3. If there is no space left
   3.1. Split the node and pull out the median
   3.2. Put all keys smaller than the median into the left node, all greater than the median into the right node
   3.3. Insert the median into the parent node and adjust pointers

Insertion Algorithm(2)

1. If there is no space in the parent node
   - If it is the root node, create a new root node, insert the median, and adjust pointers
   - Otherwise, repeat 3. with the parent node

Deletion Algorithm

- In a leaf node, a key can simply be deleted
- In an inner node, the connection to the children must be preserved
  - Therefore, the next larger (or next smaller) key is searched for (in the corresponding child node)
  - This key is written in place of the deleted key

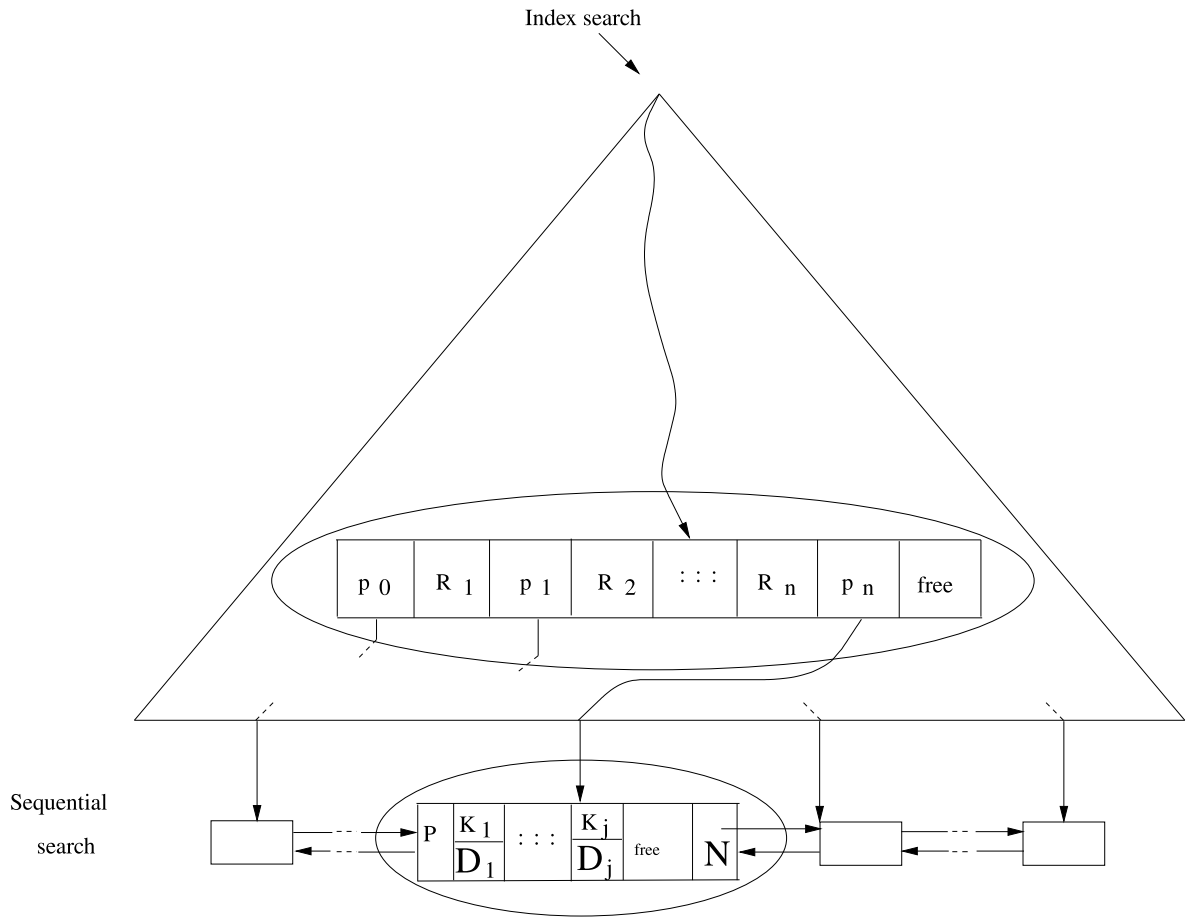
Deletion Algorithm(2)

- After deleting a key, a node can be underfull (have fewer than i entries)
- Then this node is merged with a neighboring node
- This can cause an underflow in the parent node, i.e., the parent node has to be merged as well
- Since this procedure is relatively costly and databases tend to grow rather than shrink, this merging is often not implemented

B^+-Trees

- The performance of a B-tree depends heavily on the height of the tree, therefore we want a high fan-out of the inner nodes
- Storing data in inner nodes reduces the fan-out
- B^+-trees store only reference keys in inner nodes, the data itself is stored in leaf nodes
- Usually the leaf nodes are additionally linked to allow fast sequential scans

Schematic Representation



Prefix B^+-Trees

- A further improvement is the use of reference key prefixes, e.g., for long strings
- We only need to find some reference key that separates the left from the right subtree:
  - Müller $\leq$ P $<$ Schmidt
  - Systemprogram $\leq$? $<$ Systemprogrammer

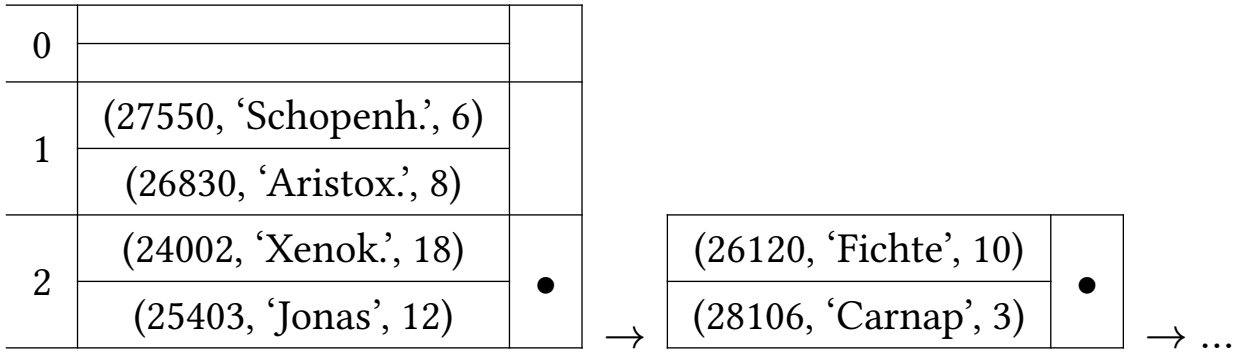
- Trees need on average $\log_k(n)$ page accesses to read a data item (k=fan-out, n=number of indexed records)
- Hash tables (partitioning methods) need on average two page accesses

- Hash function $h(x) = x \mod 3$

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

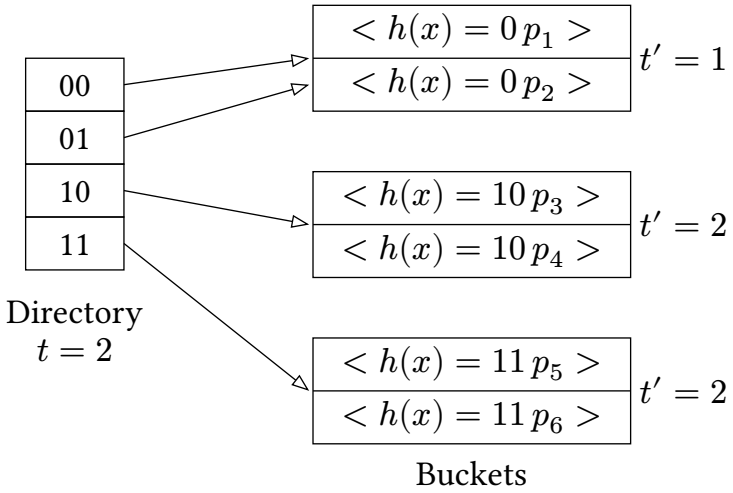
Hashing(2)

- Collision handling



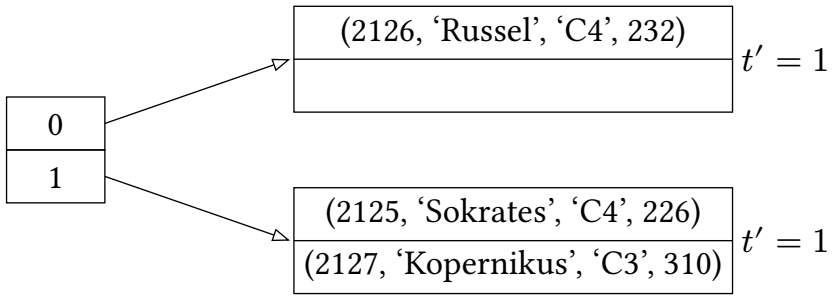
- Inefficient for unpredictable data volumes

Extendible Hashing



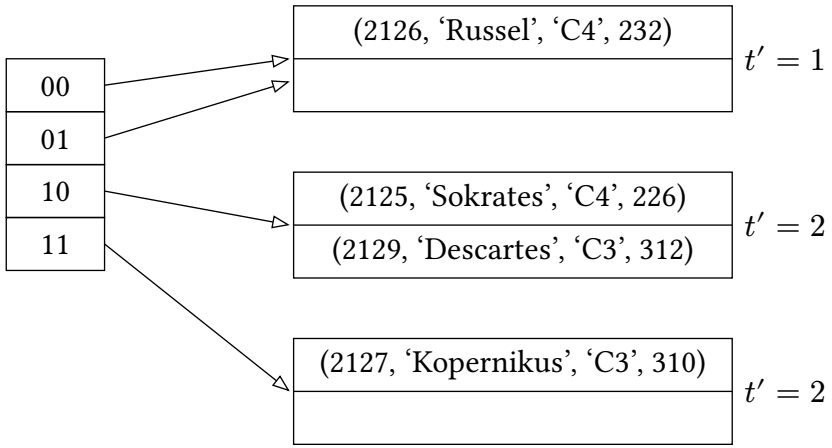
Concrete Example

	$h(x)$	
x	d	p
2125	1	01100100001
2126	0	11100100001
2127	1	11100100001



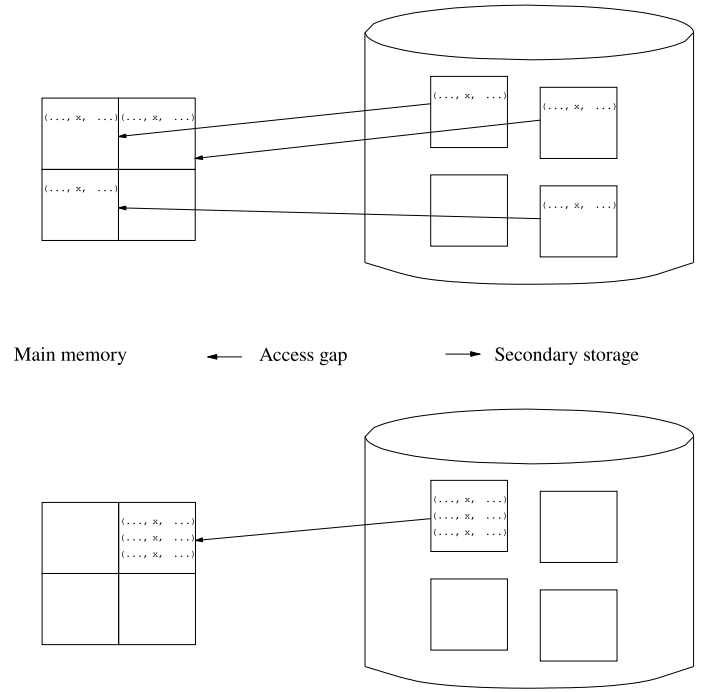
Inserting (2129, Descartes, ...)

	$h(x)$	
x	d	p
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001

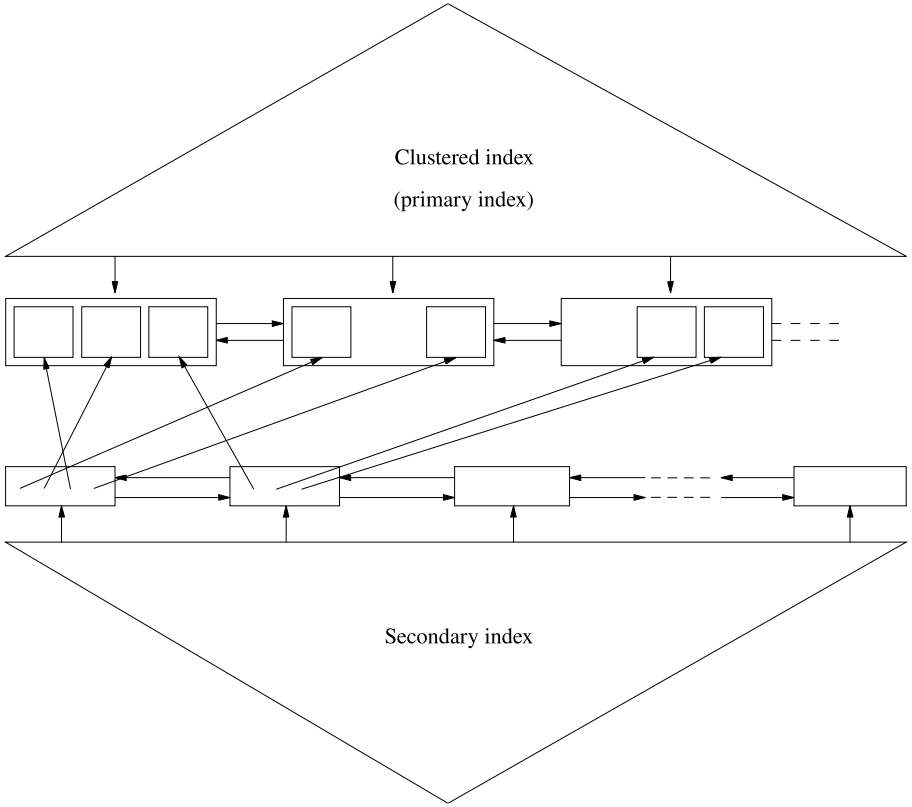


Clustering

```
select *
from   R
where  A = x;
```



Indexes and Clustering



Interleaved Clustering

			Page P_i
2125	Sokrates	C4	226
5041	Ethics	4	2125
5049	Maieutics	2	2125
4052	Logic	4	2125
2126	Russel	C4	232
5043	Epistemology	3	2126
5052	Philosophy of Science	3	2126
5216	Bioethics	2	2126
			Page P_{i+1}
2133	Popper	C3	52
5259	The Vienna Circle	2	2133
2134	Augustinus	C3	309
5022	Faith and Knowledge	2	2134

Summary

- TIDs are a clever mapping of relations to pages in secondary storage
- Indexes speed up access to data items (at the expense of update operations)
- B^+-trees are the standard index structures in relational DBMSs, suitable for both point and range queries