IN3020/4020 – Database Systems
Spring 2021, Week 2.2

INDEXING

Dr. Egor V. Kostylev, IFI, UiO

Based upon slides by E. Thorstensen and M. N. Akkøk

UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

We´ll be looking at indices (indexes)

- Conventional indices
- B(+) trees
- Multidimensional indices
- Hash-like indices
- Bitmap indices

Index

- An **index** on an attribute A (of a table) is a data structure that facilitates finding the elements with a certain value in A (A is called the **search key**)
- The simplest type is probably the hash-index
- There are other (more advanced) indexing techniques based upon binary search

Index

Search keys are created to **search** for a single entry or a set of entries in an **index**. **Search keys** may only be constructed for the **key** columns in the **index**, and may contain one or more column values.

- An **index** on an attribute A is a data structure that facilitates finding the elements with a certain values in A (A is called the **search key**)
- The index is **organised** (e.g., sorted) on the search key
- For each value of the search key, the index has a list of pointers to the corresponding records
- More than one index in the same table (or file) means
  - Faster search
  - More complexity (changes will lead to updated indices as well)
  - Increased storage requirement, larger files

Types of indices

- Dense vs. sparse indices
- Primary Index
  - The data file is sorted (physically) on the search key
  - Maximum one data entry for each search-key value
- Cluster index
  - The data file is (still) sorted (physically) on the search key
  - Allows more than one data entries with same search-key value
- Secondary index
  The data file is NOT sorted (physically) on the related search key

Overview of index types

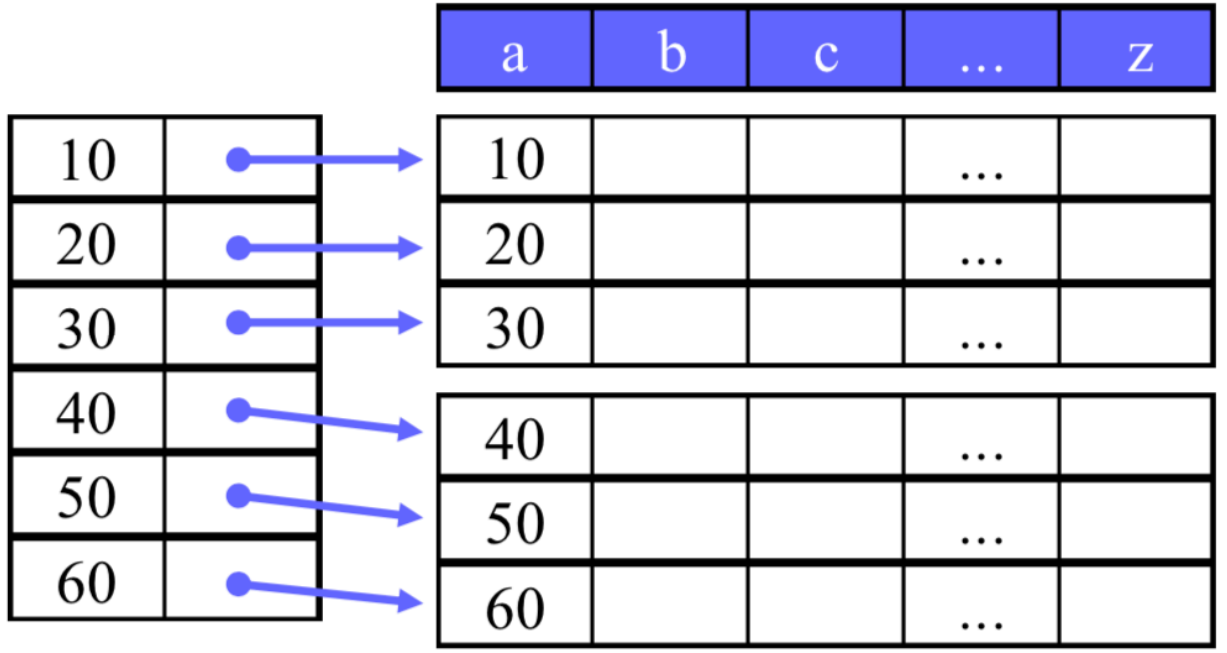
A **candidate key** is a combination of attributes that uniquely identify a database record without referring to any other data. Each table may have one or more **candidate**. One of these **candidate keys** is the table's **primary key**.

Data file	Search key is a candidate key	Search key is not a candidate key
Sorted on search key	Primary index dense or sparse	Clustering dense or sparse
Not sorted on search key	Secondary index dense	Secondary index dense

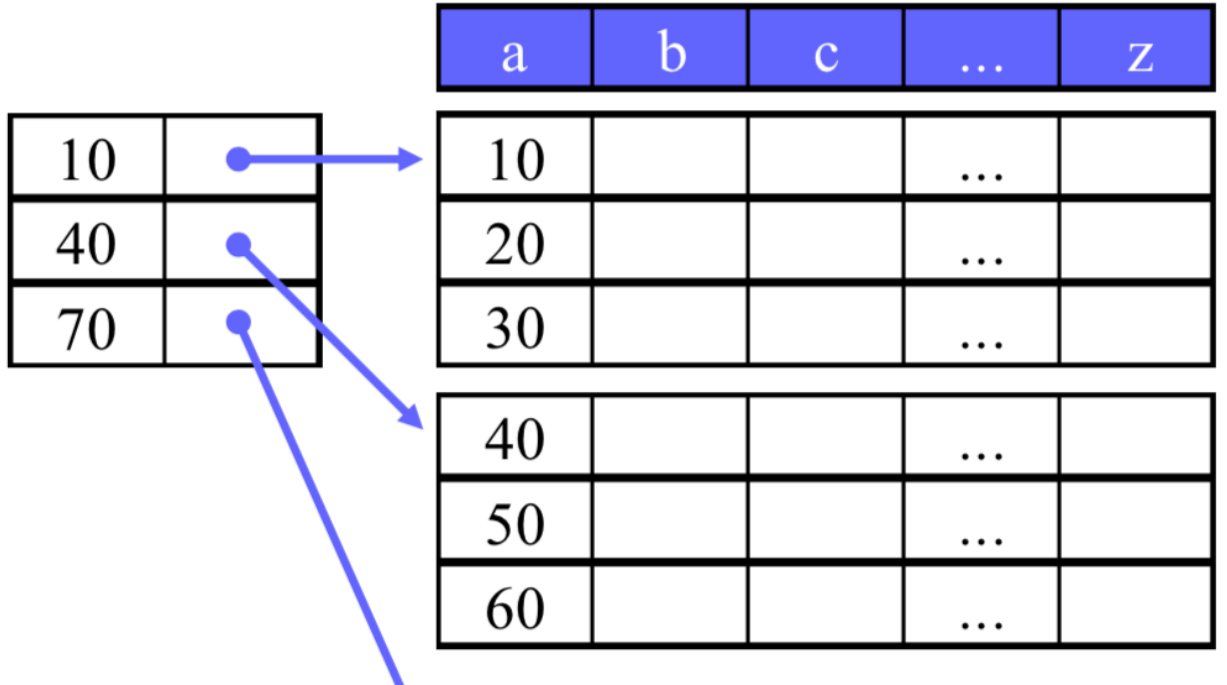
Quick overview:
https://en.wikipedia.org/wiki/Database_index
Less quick but more detailed overview (for geeks):
https://www.geeksforgeeks.org/indexing-in-databases-set-1/

Primary index:
Dense vs. Sparse indices

A **dense index** has one lookup for each value of the search key



A **sparse index** has one lookup for each data block



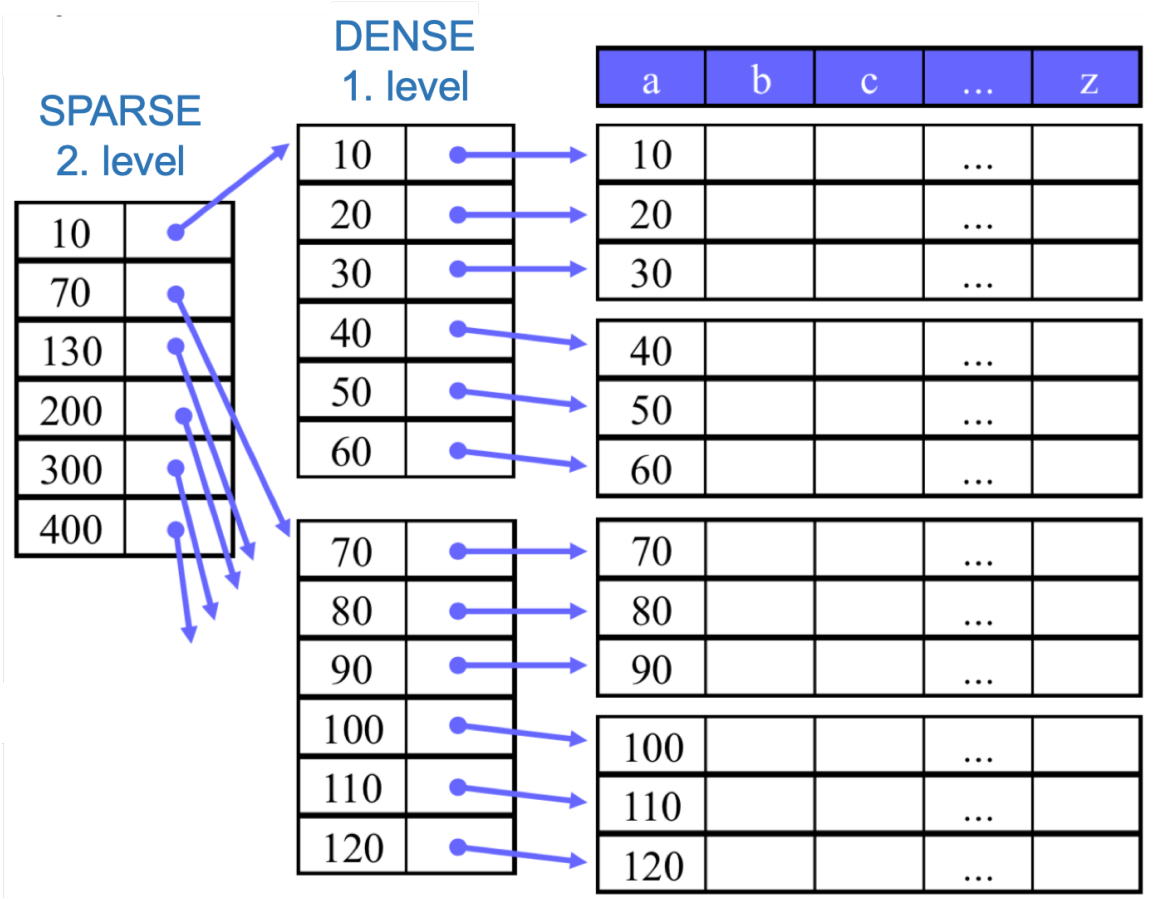
A simple comparison

- Assume that we have
- 1,000,000 entries of 300B, 4B search key, 4B pointers
- 4K block size, average 5.6ms for fetching a block
- 13.6 records per block, i.e., 76924 blocks with data
- 512 indices per block, i.e., 1954 blocks for a dense index and 151 blocks for a sparse index

- **Without index:**
  - 76924/2 = 38462 average block access
  - Takes 38462 * 5.6ms = 215,4s
- **With dense index and binary search:**
  - $[\log_2(1954)] + 1 = 11 + 1 = 12$ block accesses (max)
  - Takes 12 * 5.6ms = 67.2ms
  - 3205 times faster than the one without indices!
- **With sparse index and binary search:**
  - $[\log_2(151)] + 1 = 8 + 1 = 9$ block accesses (max)
    Takes 9 * 5.6ms = 50.4ms
  - 4272 times faster as compared to no index, and 1.33 times faster than dense index

Multi-level indices

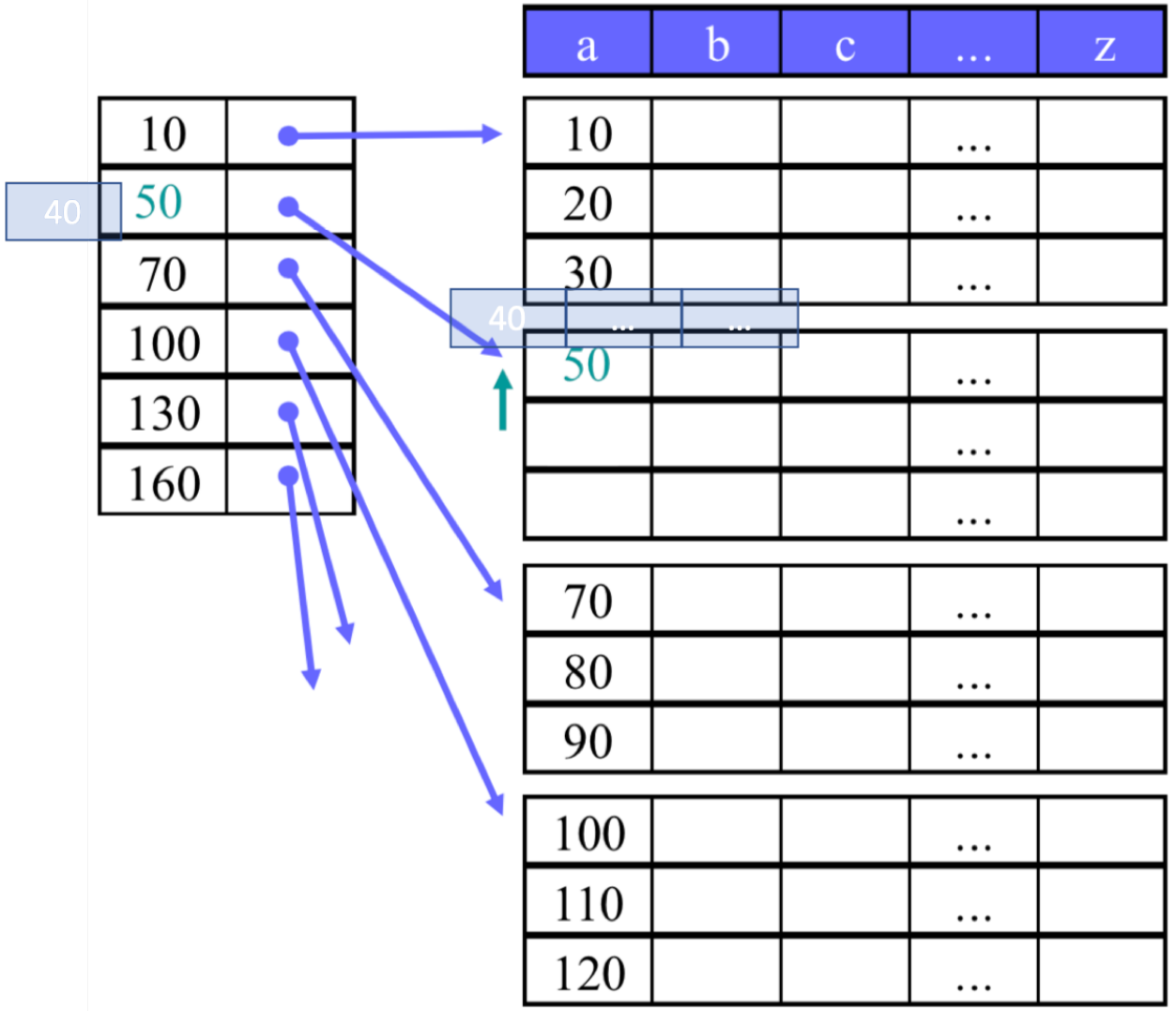
- An index can occupy several blocks
- A multi-level index (i.e., an index on an index) can improve performance
- Continuing the example:
  - Needs only 1954 / 512 = 4 blocks for 2 level
  - $[\log_2(4)] + 1 + 1 = 2 + 1 + 1 = 4$ block access, takes 4 * 5.6ms = 22.4ms
  - 2.25 faster than a simple sparse index, 3 times faster than a dense index
- One can in principle have any number of indices



Deletion with sparse index

- Delete entry where a = 60
  - No change necessary for the index

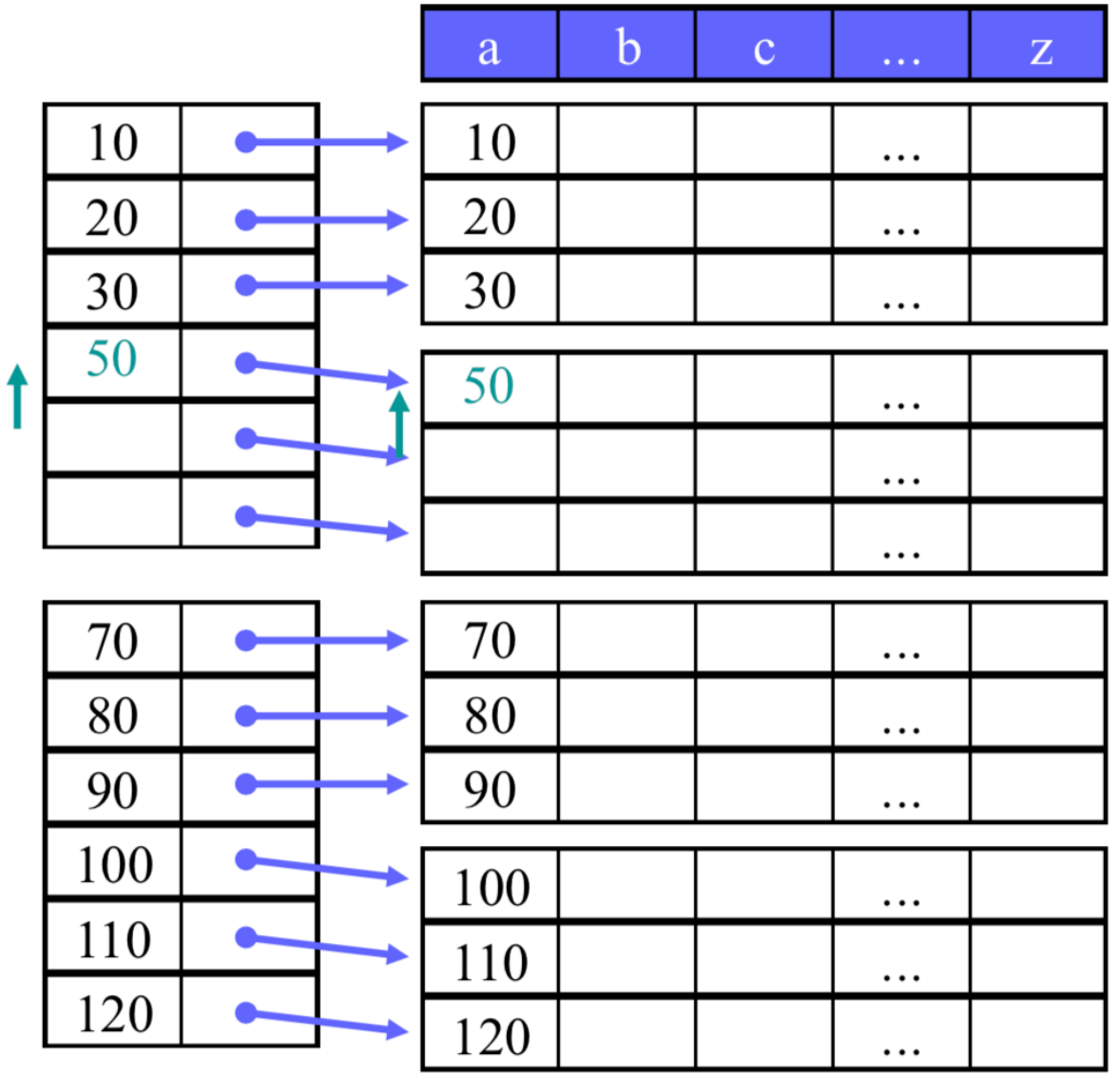
- Delete entry with a = 40
  - The first entry in the block is updated, which means that the index needs to be updated



Deletion with dense index

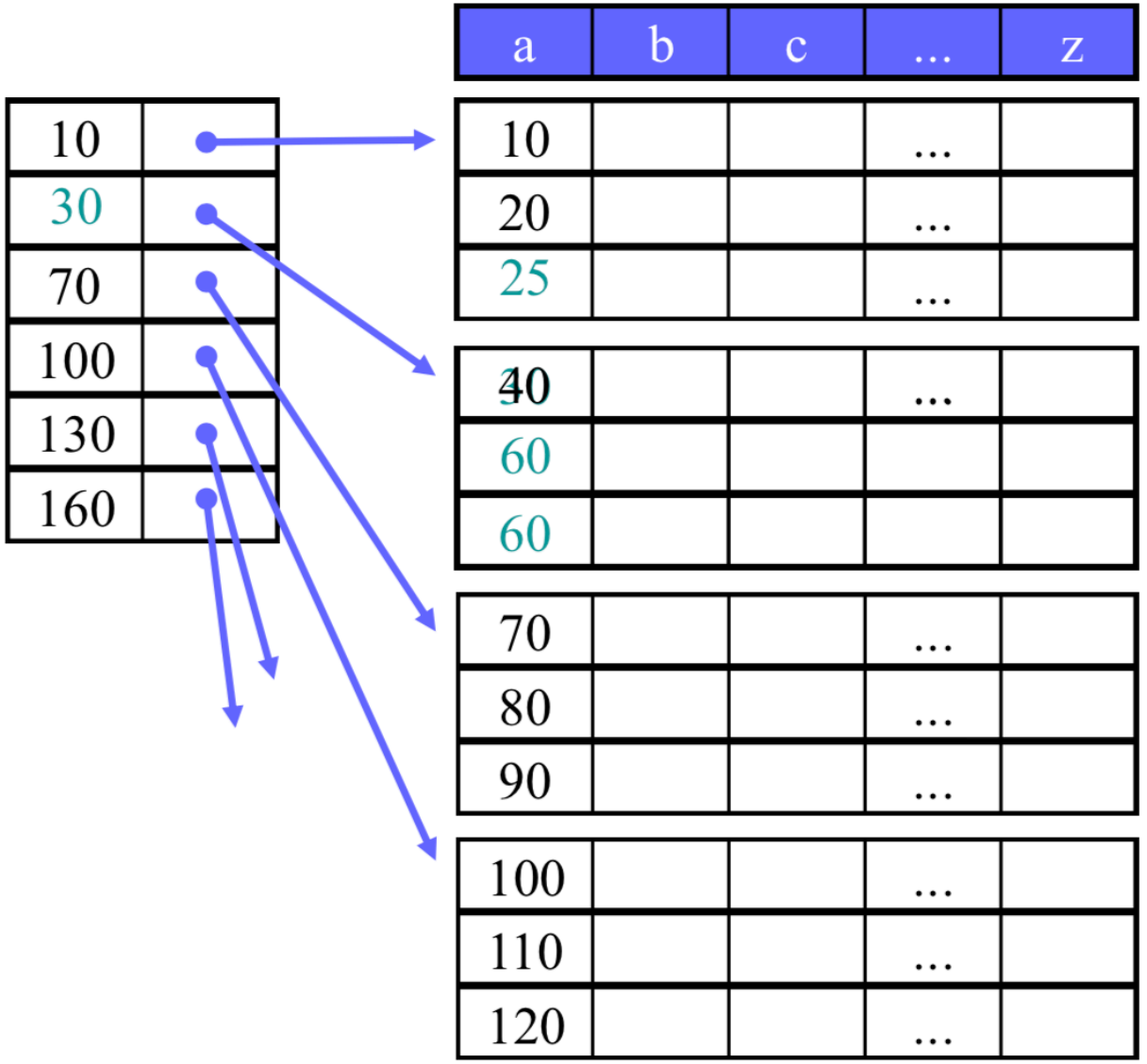
- Delete entry where a = 60
- Delete entry where a = 40

- One usually prefers to compress the data in the blocks
- One can also compress the whole data set, but we may like to keep free space for future inserts



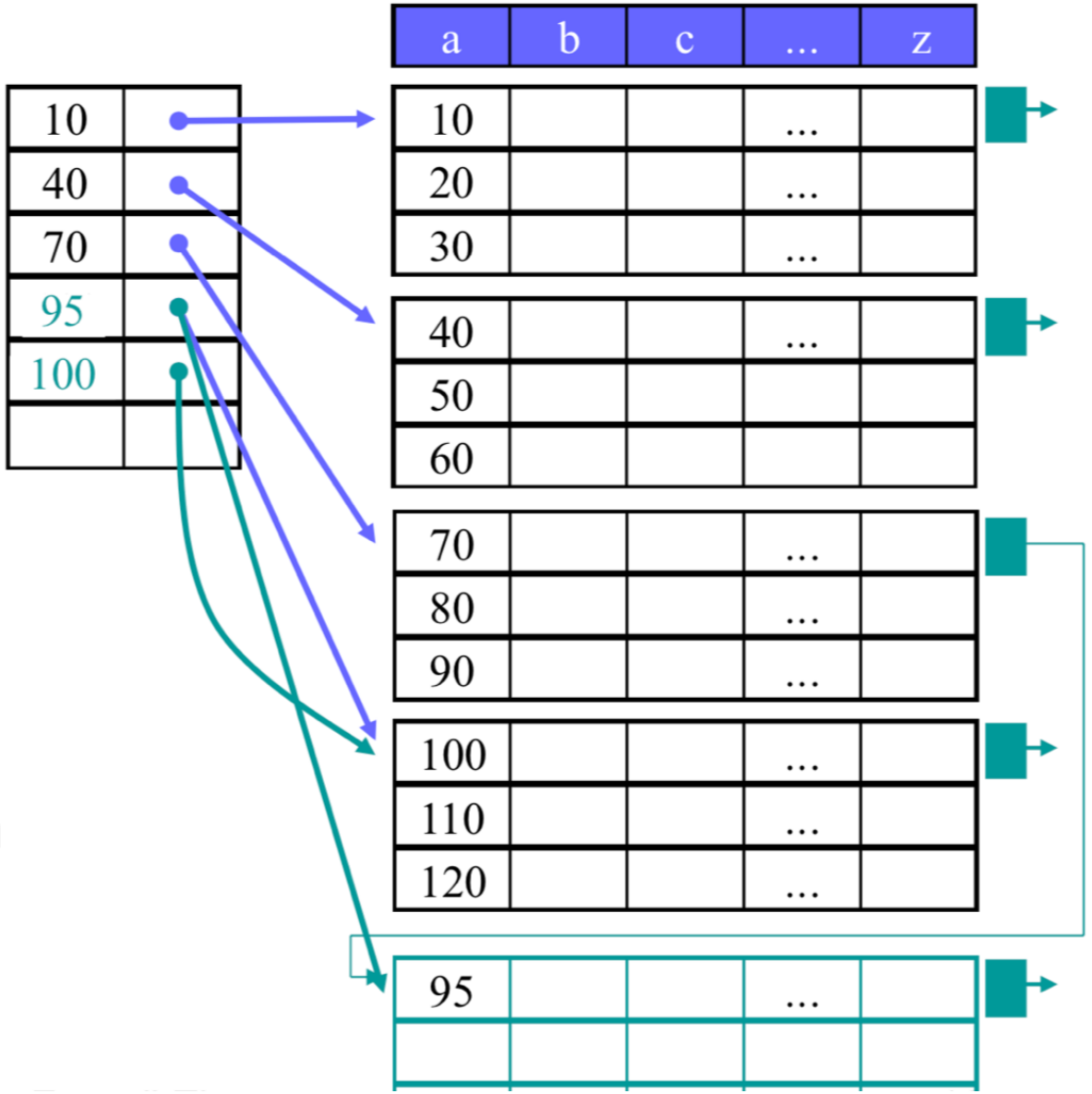
Insertion with sparse index

- Insert entry with a = 60
  - Lucky! Place available exactly where we need it!
- Insert entry with a = 25
  - Must move entry with a = 30 to next block to make space for 25
  - The first entry on block two is changed, and the index must be updated
  - NOTE: We could have inserted a new block or overflow block



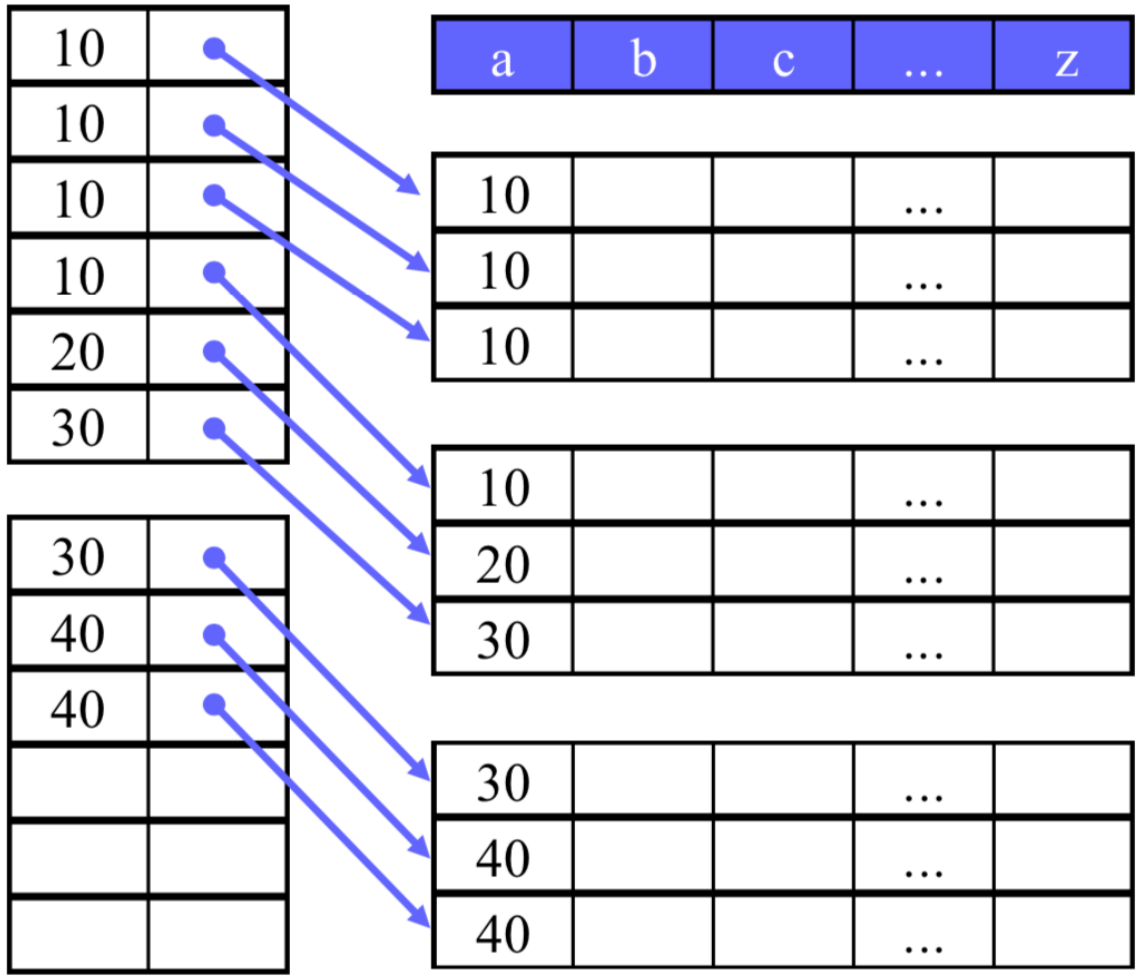
Insertion with sparse (and dense) index

- Insert entry with a = 95
  - No space. Insert a new/overflow block
  - Overflow block: No need to do much with the indices. Need only a pointer to the main blocks
  - New block: The index needs to be updated too
- Insertion in the case of dense indices is the same, except that the index must be updated every time



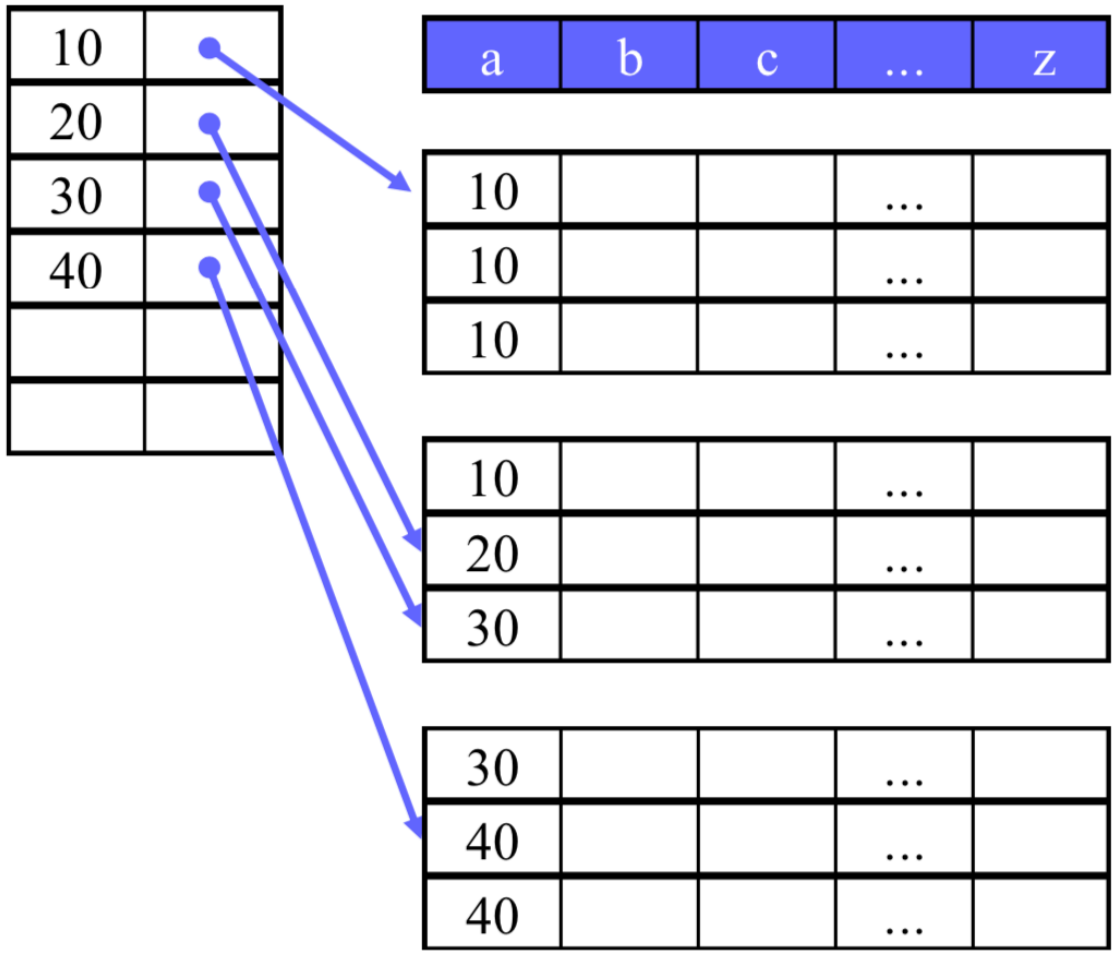
Cluster Index - Duplicate (or Multiple) Search Keys

- If the file is sorted, a cluster index can be used even if the search key is not unique
- Example 1 – dense index:
  - One index field per record
  - Makes it easy to find entries and how many there are of each ☺️
  - Too many fields? More than necessary? 🧐



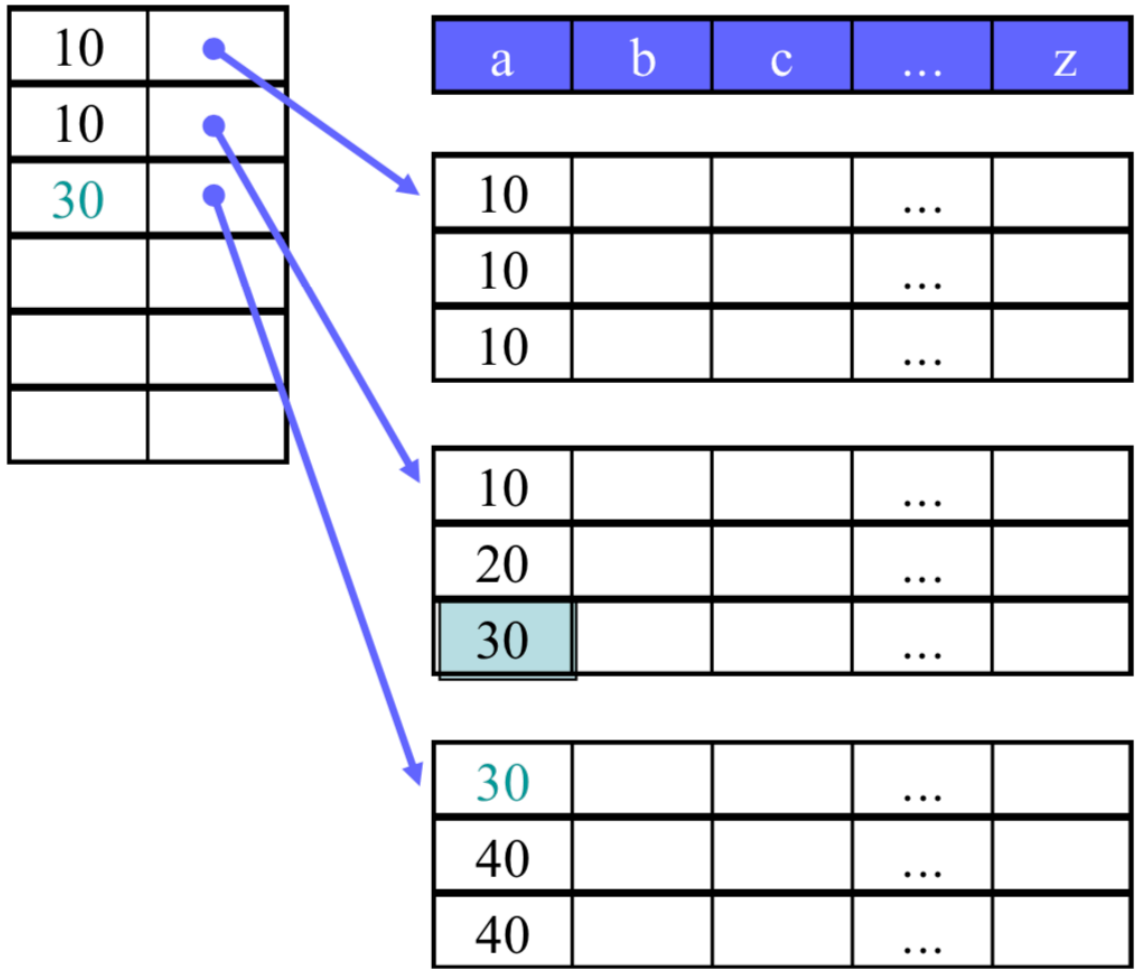
Cluster Index – Sparse Index (a)

- Only one index field per unique search key
- Less index – faster search ☺️
- Finding subsequent (intermediary) records is more complex 🙃



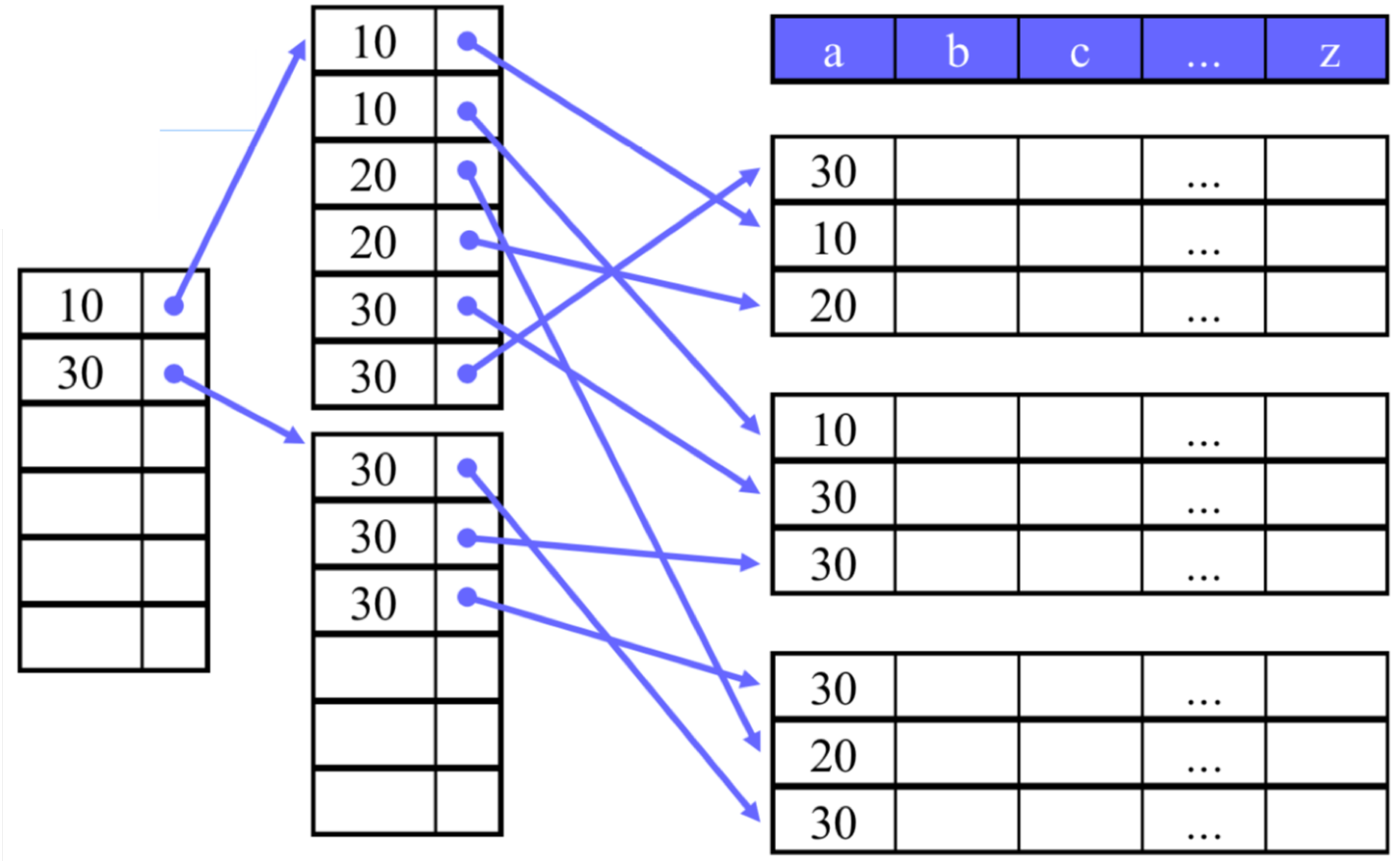
Cluster Index – Sparse Index (b)

- Index fields point to the first entry in each block
- Even fewer indices – faster search ☺️
- Finding records is more complex 🙃



Unsorted Files and Secondary Indices

- One can use a **secondary index** if the files is unsorted (or sorted on another attribute)
  - Sorted on the search key: fast search
  - First level is always dense, higher levels are sparse
  - Duplicates are allowed



Dense vs. Sparse Indices

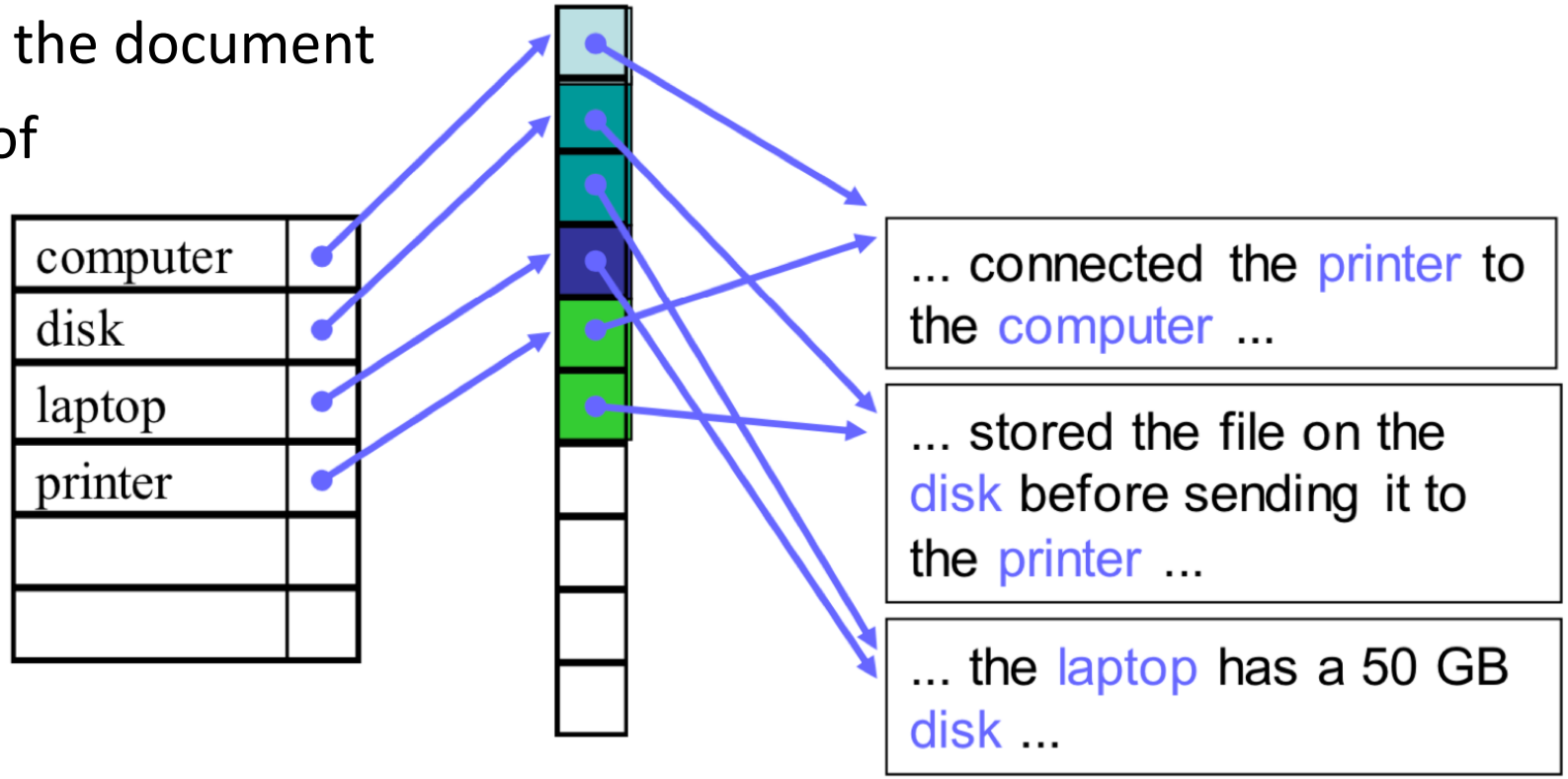
	DENSE	SPARSE
Space required	One index field per entry	One index field per data block
Block access	“Many”	“Few”
Access to entry	Direct access	Must search in the data block
Exists queries	Uses only the index	Must always access the data block
Usage	All cases	Not on unsorted elements
Updates/changes	Always updated if entry sequence is changed	Updated only if the first entry of the data block is changed

Inverted Indices

- What if we want to search for items within an attribute?
  - **SELECT** * **from** R **where** a **like** ´%cat%´
  - Search for documents that contain certain keywords, e.g., search engines like Google, Altavista, Excite, Lycos, AllTheWeb etc.

Inverted Indices (called “inverted” because…)

- **Direct Index:** Entries of the form (id1: document1), (id2: document2), ...
  - Look up an ID, access the document
- **Inverted index:** Entries of the form (computer: [id1]), (disk: [id2, id3]), ...
  - Look up a keyword, access all relevant document IDs



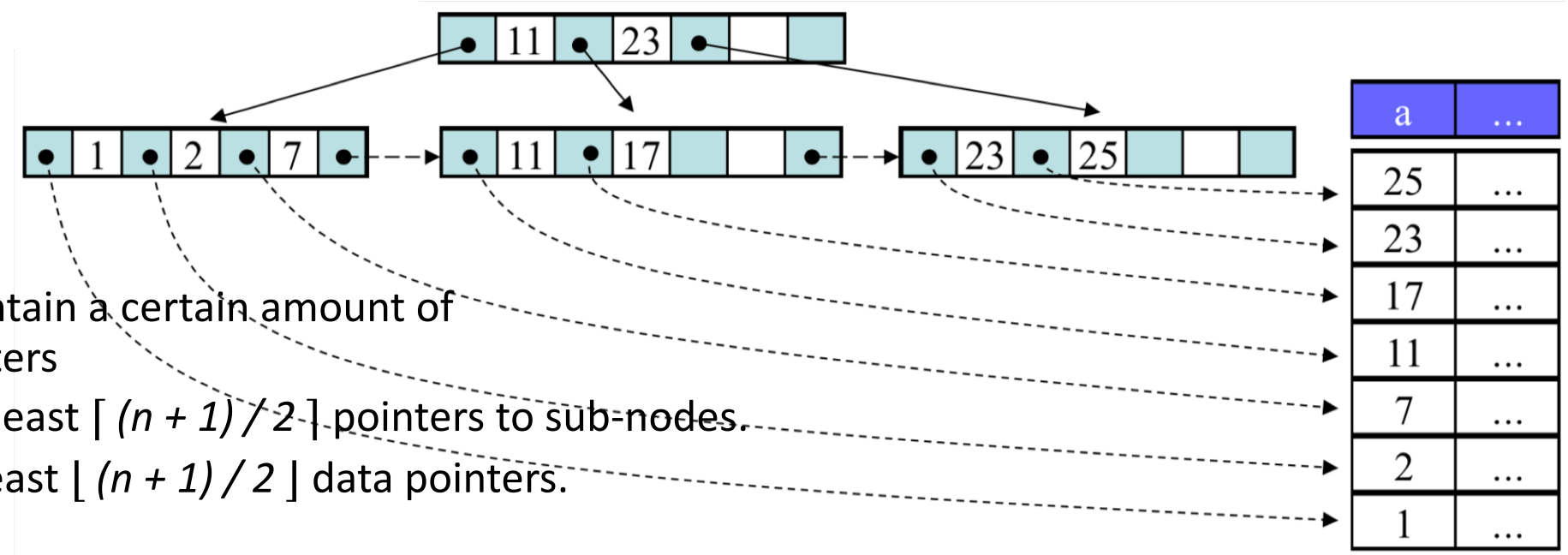
Data Structures for Indices

- So, how do we represent an index?

- Obvious idea: Sorted list.
- Better ideas?

B+ Trees

- The nodes are blocks, all leaf nodes are at the **same level**, each node has *n* search keys and *n + 1* pointers
  - Inner node: all pointers are to sub-nodes
  - Leaf node: *n* data-pointers and *1* next-pointer



- All nodes must contain a certain amount of search keys / pointers
  - Inner node: at least $\lceil (n + 1) / 2 \rceil$ pointers to sub-nodes.
  - Leaf node: at least $\lfloor (n + 1) / 2 \rfloor$ data pointers.

B+ Trees: Efficiency

- B+ trees:

  ☹ A search must always start from the root to a leaf node, i.e., the number of block accesses is equal to the height of the tree plus access to the records themselves in the data file.

  ☺ The number of levels is usually very low (typically 3)

  ☺ Interval search is fast

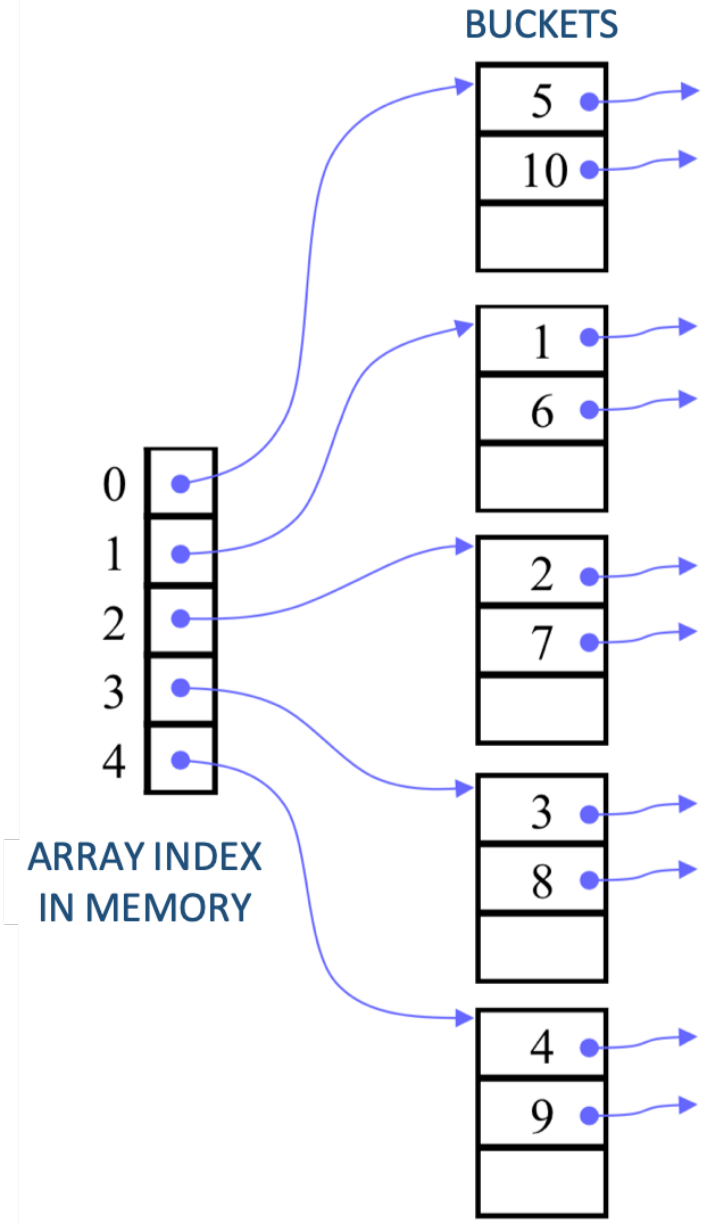
  ☺ For large n, it is rarely necessary to split / merge nodes

  ☺ Disk I / O can be reduced by keeping some of the index blocks in memory

- Example: 4B search keys, 8B pointers, 4KB blocks
  - How many values can be stored in each node?
    $4n + 8(n + 1) \leq 4096 \rightarrow n = \mathbf{340}$
  - The nodes are on the average 75% full. How many records can a 3-level B+ tree contain?
    $(340 \times 75\%)^3 = 16679103{,}875 \approx 16{,}6$ **million** records!

Hash Tables

- Uses a hash function on the search key to an array index with points to which bucket possibly contains information about the current record
- Each bucket is a block (with support for overflow blocks)
  - Array size is usually a prime number
  - Important for a hash-function!
    - Fast
    - A good distribution of the search keys to buckets
- Example: Array size B = 5; h(key) = mod(key, B)



Hash Table: Efficiency

- Ideally, the array size is large enough so that all elements of one hash value fit into one bucket block.

  ☺ Then we get significantly fewer disk operations than with regular indices and B+ trees

  ☺ Fast search for specific search key

  ☹ Multiple entries can lead to more blocks per bucket

  ☹ Poor on interval search

Dynamic Hash Tables

- Difficult to keep all items for one hash value within a bucket block if the number of records increases while the hash table remains static
- Dynamic hash tables allow the array size to vary so that it is sufficient with one block per bucket
  - Extensible hashing
  - Linear hashing

Sequential vs. Hash Indices

- Sequential indices such as B+ trees are good at interval search:

  select * **from** R **where** a > 5

- Hash indices are good when searching for a specific key:

  select * **from** R **where** a = 5

Indices in SQL

- Syntax (DBMS-dependent):
  - **create index** name **on** relationName (attribute)
  - **create unique index** name **on** relationName (attribute)
  - **drop index** name

- NOTE: Not all DBMSs allow us to specify
  - type of index, e.g., B-tree, hashing, etc.
  - parameters such as load factor, hash size, etc.

Indices in PSQL

Default index in Postgres is B-tree

“B-trees can handle equality and range queries on data that can be sorted into some ordering”

https://www.postgresql.org/docs/9.2/static/indices-types.html

Works for =, <, <=, BETWEEN and IN.
Also for LIKE, but limited.

Indices in PSQL

- There is also something called GiST. This is a mechanism for implementing indices for particular data types.
- Can be used in many cases. Some is included in Postgres by default.

Queries with several conditions

select ... **from** R **where** a = 30 **and** b < 5

- Strategy 1:
  - Use an index, e.g., on a
  - Find and *fetch* all records with a = 30
  - Search through these records to find records with b < 5

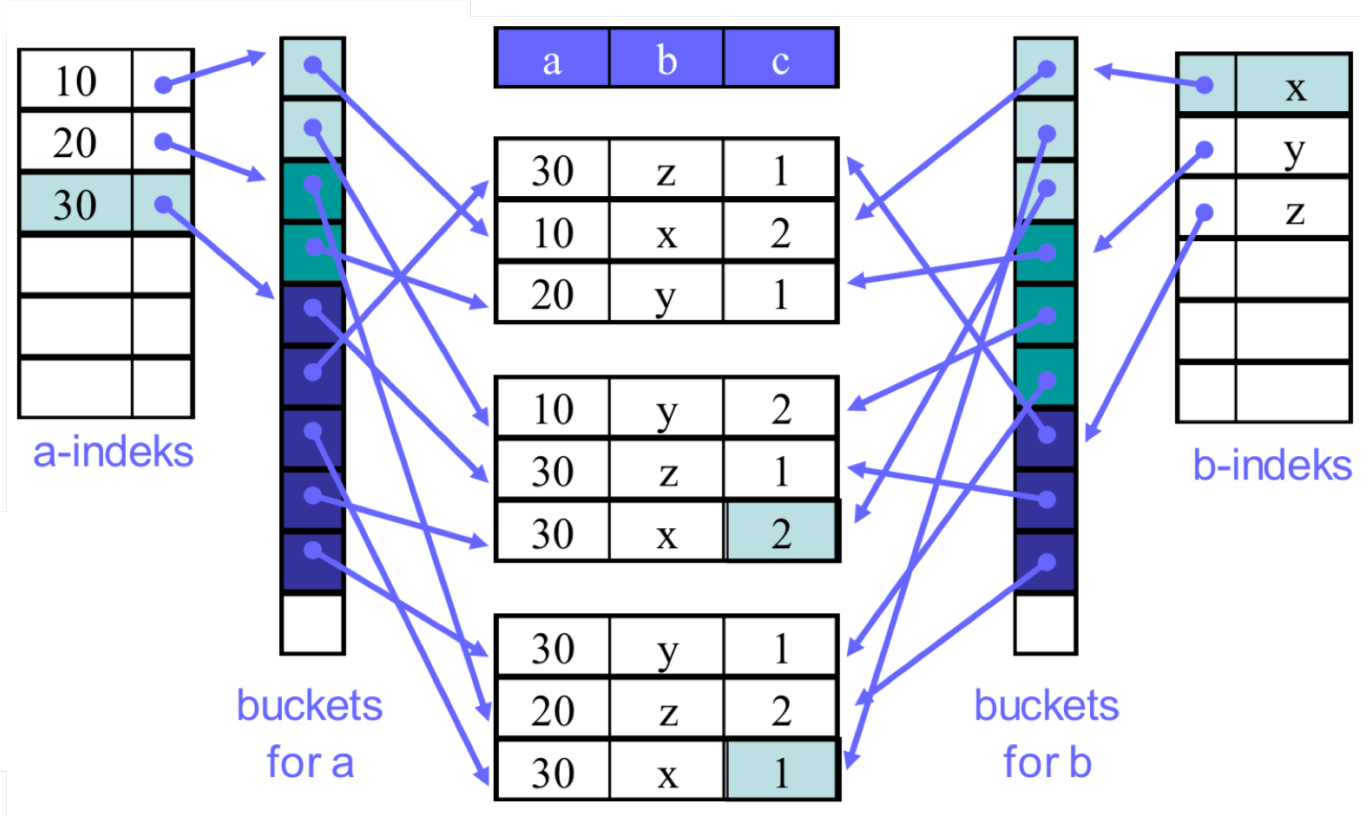
☺️ Simple strategy

☹️ Risks reading many unnecessary records from disk (storage)

Several conditions: Strategy 2

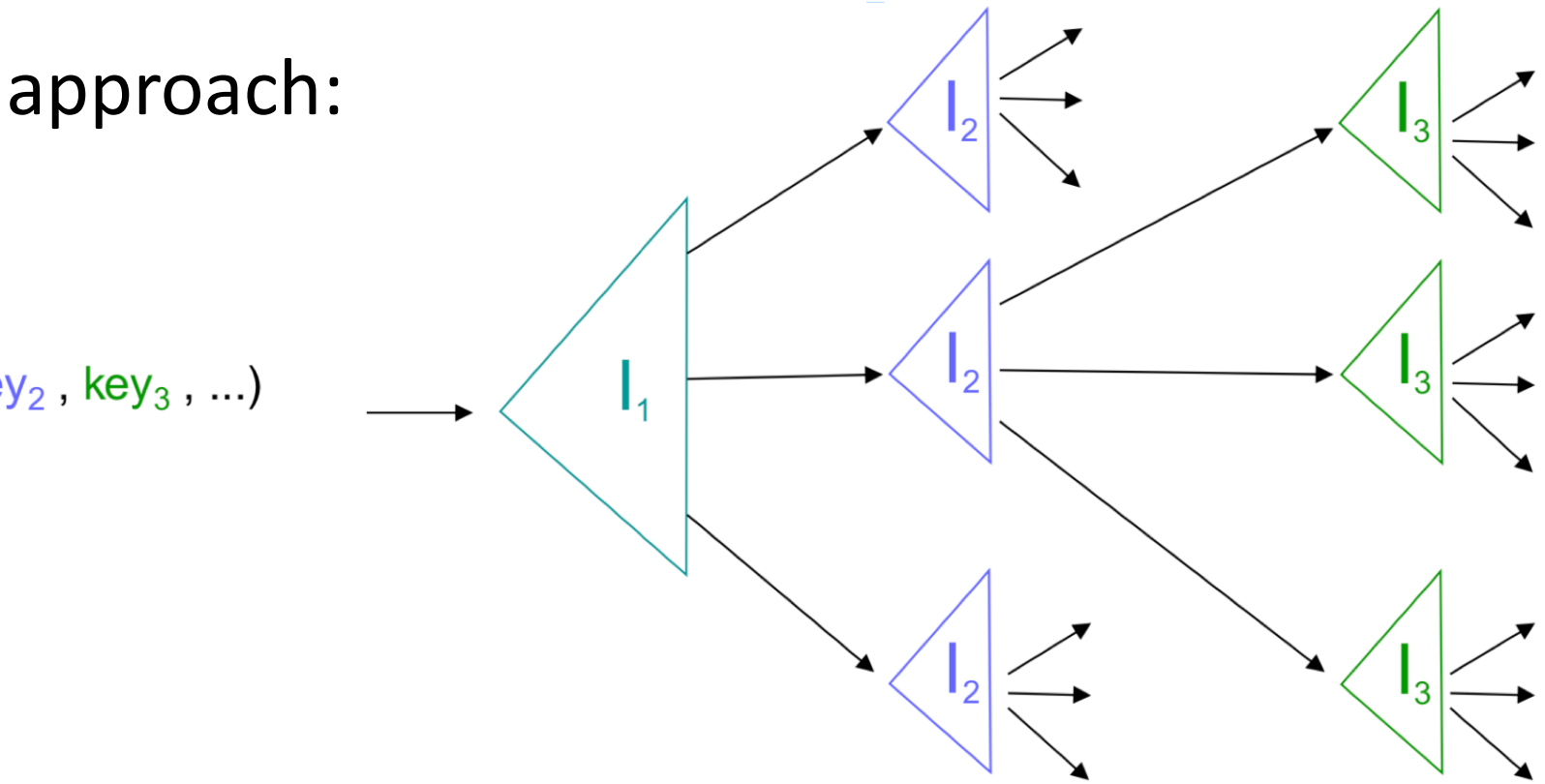
select c **from** R **where** a = 30 **and** b = ´x´

- Use two dense indices: one for a, one for b
- Find all pointers to records with a = 30
- Find all pointers to entries with b=‘x’
- Compare (intersect) pointers and fetch the relevant records
- Pick the relevant attributes



Multidimensional indices

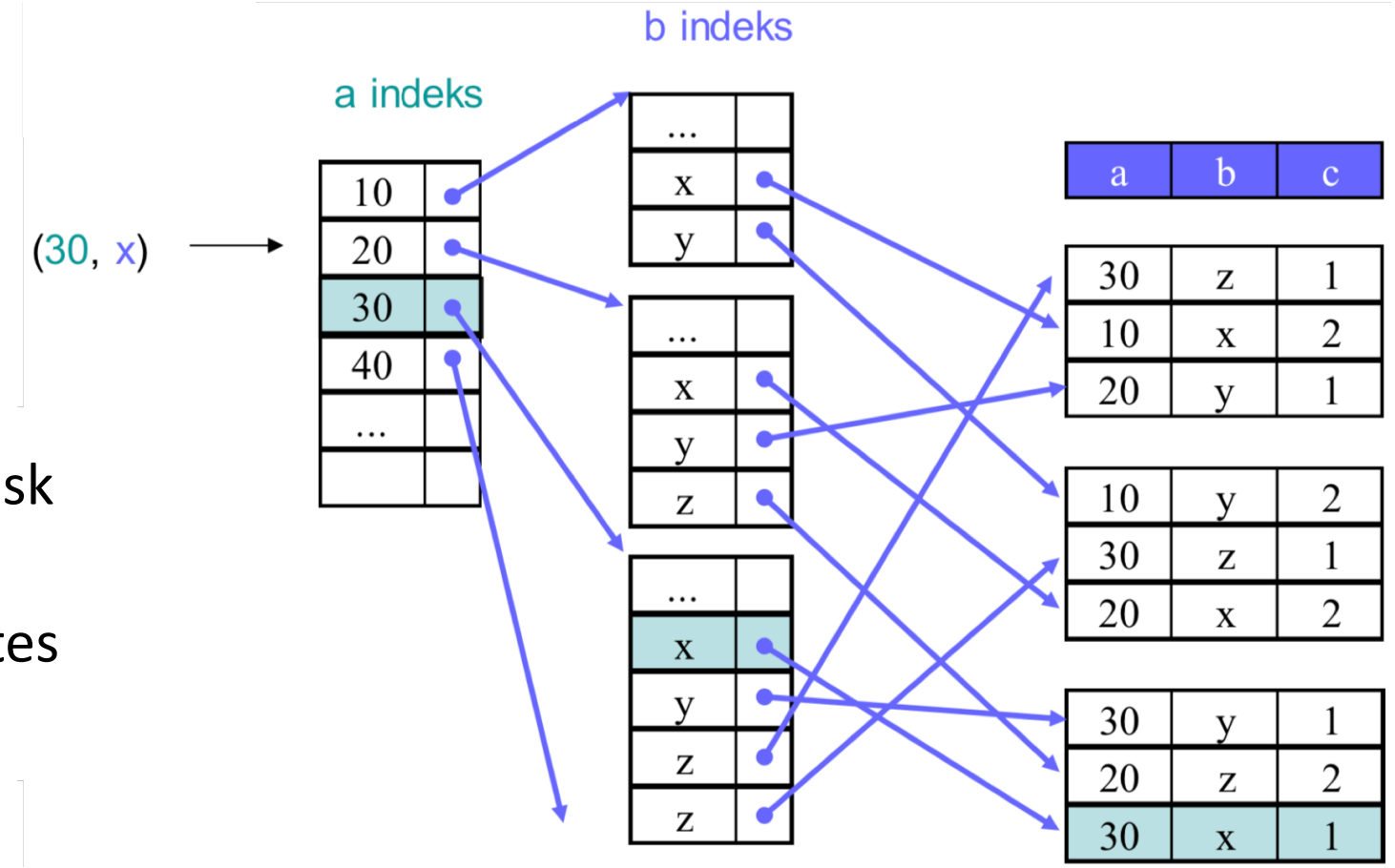
- A multidimensional index combines multiple dimensions in one index
- A simple tree-like approach:



Multidimensional indices: Example (with dense indices)

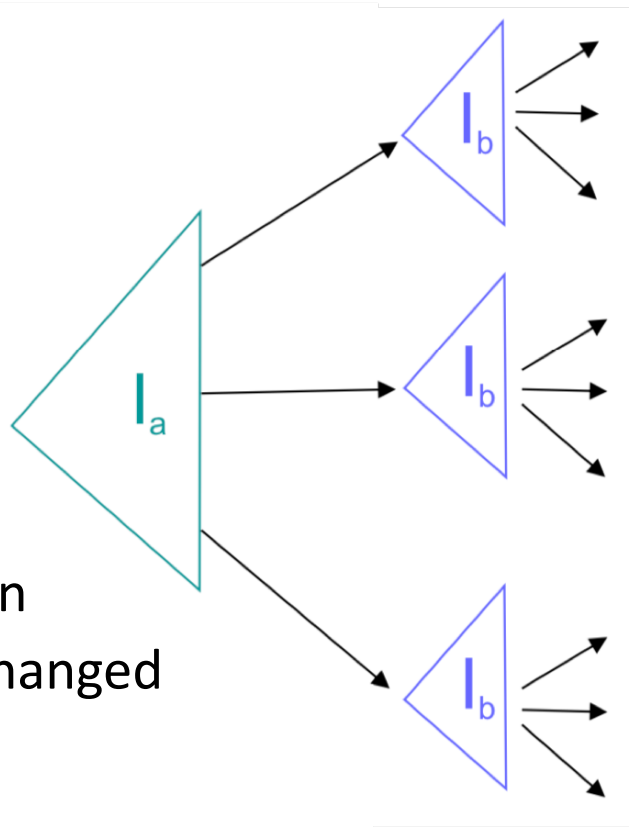
select c **from** R **where** a = 30 **and** b = ´x´

- Search key = (30, x)
- Read the a dimension
- Look for 30, find the related index for the b-dimension
- Search for x, read relevant disk block and fetch the record
- Pick out the relevant attributes



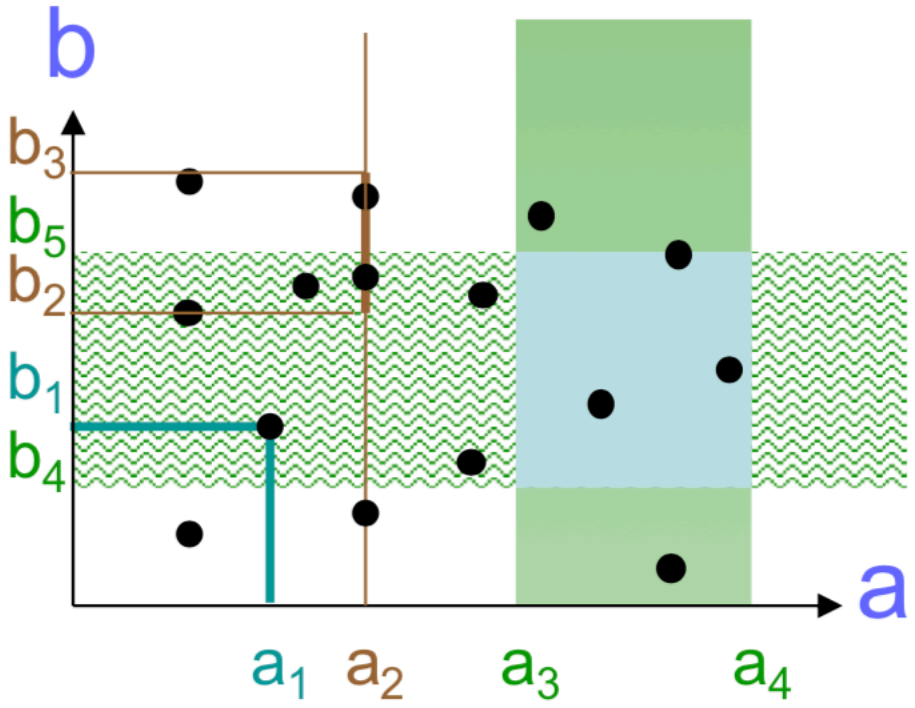
Multidimensional indices: For which queries is this a good index to use?

- 😊 Find records with a = 10 **and** b = ´x´
- 😊 Find records with a = 10 **and** b >= ´x´
- 🤔 Find records with a = 10
- ☹️ Find records with b = ´x´
- 🤔 Find records with a >= 10 **and** b = ´x´
  - Risks searching through many indices in the next dimension
  - Would have been better if the dimension order could be changed
- But there are many other alternatives:
  - Other multidimensional tree-like structures
  - Multidimensional hash-like structures
  - Bitmap indices



Map View

- We can visualize a multidimensional index with two dimensions as a map
- Search is then equivalent to a search on the map for:
  - Points: a_1 and b_1
  - Lines: a_2 and $< b_2, b_3 >$
  - Areas: $< a_3, a_4 >$ and $< b_4, b_5 >$



Tree structures

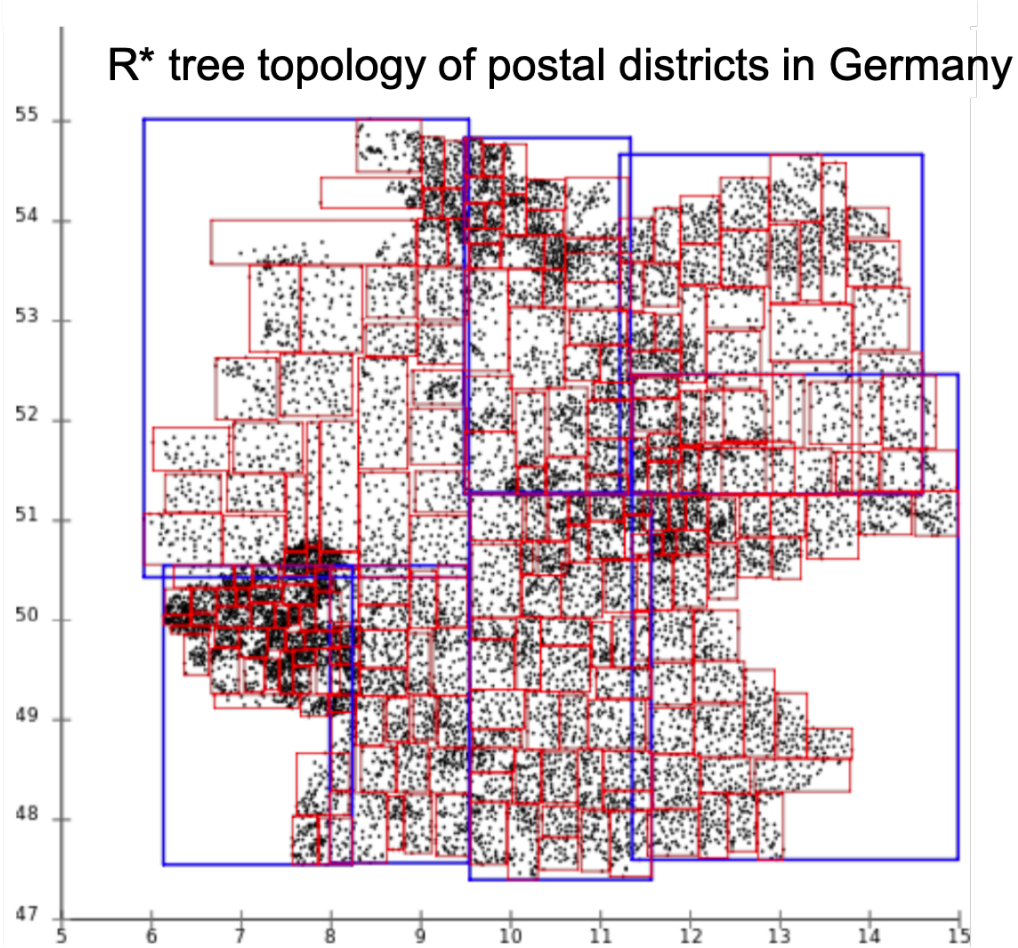
- There are many tree-like structures that correspond to searching for map areas:
  - **k-d trees:** a binary search tree in which *every* leaf node is a *k*-dimensional point. Very useful for range and nearest neighbour searches, and for multidimensional search keys
  - **Quad-trees:** a *tree* data structure in which each internal node has exactly four children. Often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions (used in image compression, also in spatial search, etc.)
  - **R-trees:** tree data structures used for spatial access methods, i.e., for indexing multi-dimensional information such as geographical coordinates, rectangles or polygons

Tree structures: Characteristics

- All these tree structures must satisfy at least one of the following characteristics of B(+) trees:
  - Balancing - all leaf nodes are at the same level (a **B+ tree** is a self-balancing tree)
  - Correspondence between tree nodes and disk blocks
  - Good performance for update operations

R-trees

- **The basic idea:**
  Group geometric objects that are close to each other
- Nodes are blocks. All leaf nodes are at the same level
  - Inner node: The smallest rectangle that covers all the objects in the subtree
  - Leaf node: An object
  - All inner nodes must contain a certain number of pointers
- Search (intersection, contained in, nearest neighbour) is easy
- Insertion is challenging
  - The tree should be completely balanced
  - Rectangles should not contain too much space
  - Rectangles on the same level should not overlap (may be relaxed)
  - May need to delete and re-insert objects for better placement



Hash-like structures: Grid files

Grid files extend traditional hash indices to multiple dimensions

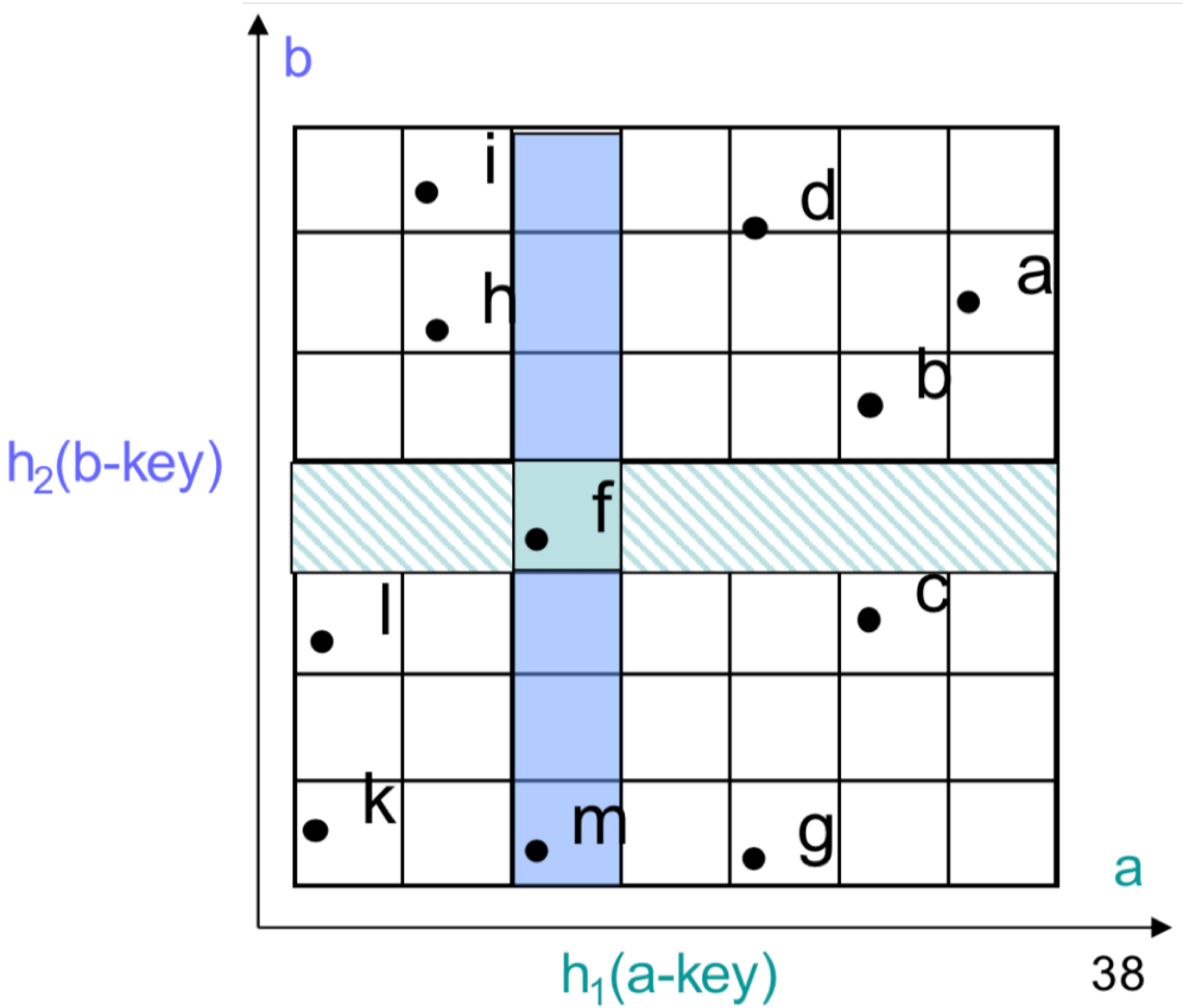
- Hashes values for each attribute in a multi-dimensional index
- Does not usually hash to individual values but to regions: h(key) = <x, y>
- Grid lines partition areas into stripes

Example (2 dimensions):
Find record with (a, b) = (22, 31)

h1(22) = <ax, ay>

h2(31) = <bm, bn>

→ enter f



Grid files

- Grid files are good for finding records/entries with
  - $key_1 = V_i$ **and** $key_2 = X_j$
  - $key_1 = V_i$
  - $key_2 = V_j$
  - $key_1 >= V_i$ **and** $key_2 < X_j$
- Grid files:
  - ☺ Are good for searches with multiple keys
  - ☹ Use too much space, and require some organizing

Bitmap indices: Example

File

record-nummer	F	G	H
1	30	foo	65
2	30	bar	43
3	40	baz	84
4	50	fou	43
5	40	bar	65
6	30	baz	65

Bitmap vectors for F

30:	1	1	0	0	0	1
40:	0	0	1	0	1	0
50:	0	0	0	1	0	0

Bitmap vectors for G

bar:	0	1	0	0	1	0
baz:	0	0	1	0	0	1
foo:	1	0	0	0	0	0
fou:	0	0	0	1	0	0

Bitmap indices

- Starting point: Each record is assigned an unchanging, unambiguous number
  - Numbering from 1 to n
  - The number can be considered a record ID and cannot be reused even if the record is deleted
- Select the field F to be indexed
  - For each value v used for F in one of the records, create a bit vector b_v of length n
  - If record nr. i has $F = v$, let $b_v[i]=1$
  - If record nr. i has $F \neq v$, let $b_v[i]=0$

Bitmap index characteristics

- Space requirements:
  - Total number of bits is #records * #values
  - In the worst case, n^2 bits is needed (but then each bit vector has only one 1-bit)
  - Bit vectors can be compressed; there is never more than n 1-bits total in the bit vectors
- Effective for:
  - Partial match queries (= state values for some fields, find all that have given values)
    - Calculate bit by bit **and** across the bitmap indices for the relevant attributes
  - Range queries (= enter intervals for some fields, find all that have values within the ranges)
    - Calculate bit by bit within the intervals