Indexing

Contains slides by
Hector Garcia-Molina, Vera Goebel

Overview

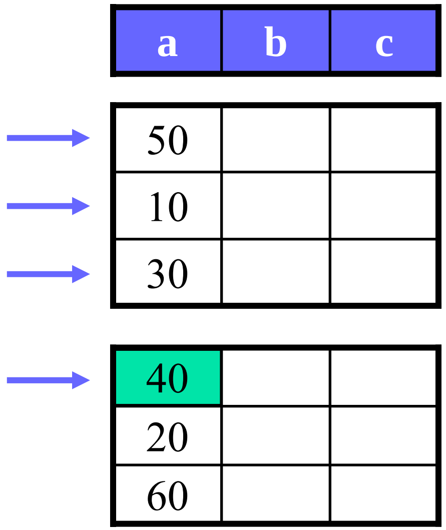
- ✓ Conventional indexes
- ✓ B-trees
- ✓ Hashing schemes
- ✓ Multidimensional indexes
  - tree-like structures
  - hash-like structures
  - bitmap-indexes

Search - I

- How do you represent a relation or an extent?
  - just scatter records among the blocks
  - order records
  - clustering
  - ...

- Today:
  How do we find an element quickly?

- Example 1 – linear search:
  `SELECT * FROM R WHERE a = 40`
  - read the records one-by-one
  - we must in average read 50 % of the records and thus 50 % of the disk blocks
  - ***expensive***

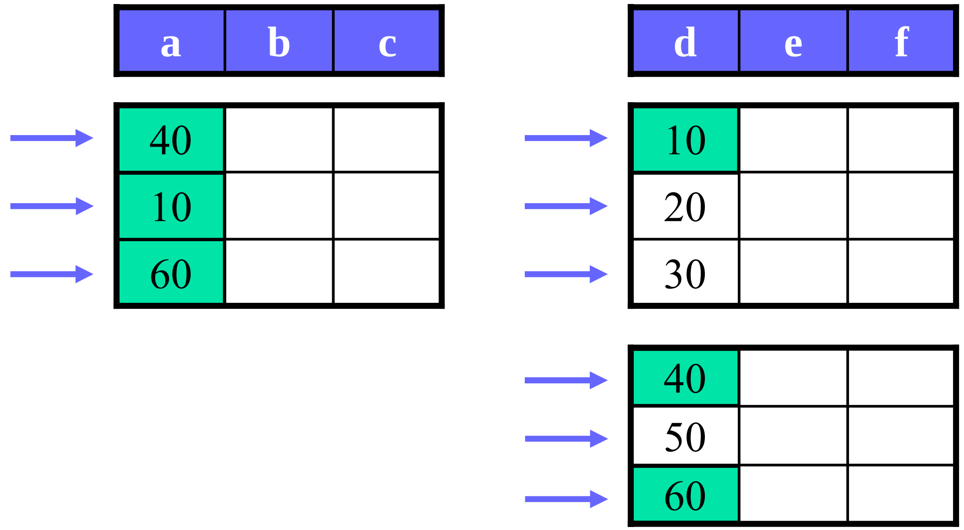


Search - II

- ✓ Example 2 – linear search:

  `SELECT * FROM R,S WHERE a = d`

  - ➢ for each tuple in R, we read on average 50 % of the tuples in S
  - ➢ ***expensive***



- ✓ We need a mechanism to *speed up the search* for a tuple with a particular value of an attribute ...

  ... ***INDEXES***

Conventional Indexes

Indexes – I

- An index on an attribute A is a data structure that makes it easy to find those elements that have a fixed value for attribute A
- Each index is specified on field(s) of a file
  - search key (indexing field/attribute)
  - the index stores each value of the search key ordered along with a list of pointers to the corresponding records
  - search an index to find a list of addresses
- We still may need to read many index fields to find a given search key, but indexes are more efficient than it may seem:
  - *the index fields are usually much smaller than the records*, i.e., less data blocks must be accessed – may even be small enough to pin in memory
  - the keys are *sorted*, e.g., make a binary search to find the key

Indexes – II

- ✓ A file (e.g., relation or extent) can have several indexes
- ✓ Selection of indexes is a tradeoff
  - ☺ improve searches and number of block accesses, e.g., in queries
  - ☹ increase complexity, e.g., modifications are harder and more time consuming as the indexes also must be updated
  - ☹ increase storage requirement
- ✓ One must analyze the typical usage pattern, i.e., if a set of elements are more frequently ...
  - ➢ ... queried than modified, then an index is useful
  - ➢ ... modified than queried, then the cost of also modifying the index must be considered
- ✓ Indexes are specially useful for attributes that are used in the `WHERE` clause of queries and for attributes that are used in join

Indexes – III

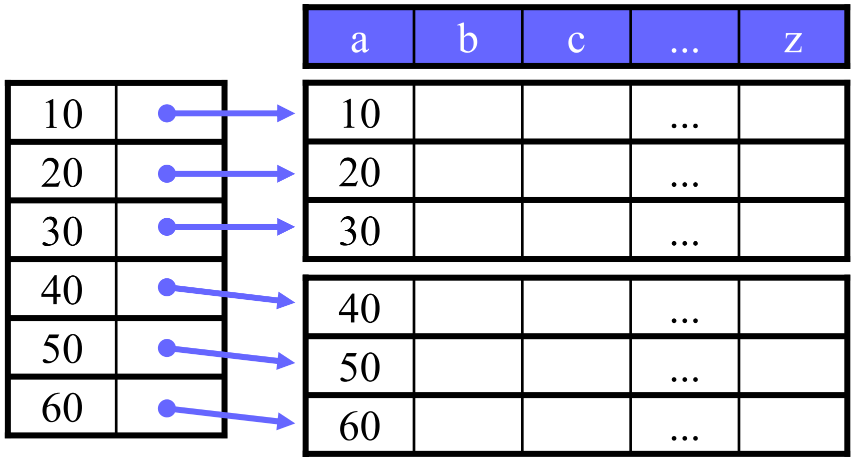
- ✓ An index can be dense or sparse

Note:
can at most have *one* primary or *one* clustering index, not both or many of one kind. Why?

- ✓ Different types of indexes:
  - primary indexes:
    specified on ordering key field of an ordered file
    NB! not the same as index on primary key
  - clustering indexes:
    specified on ordering field of an ordered file, but ordering field is not a key field (i.e., search key not unique - multiple records with same value of field may exist)
  - secondary indexes:
    can be specified on any non-ordering field

Dense Indexes – I

✓ A dense index has one index entry for every search key value



- every data record is represented in the index
- an existence search of a record may be done via index only
- the record is directly found in a block using the pointer, i.e., no search within the block
- if index fits in memory, a record can be found using a given search key with a maximum one disk I/O

Dense Indexes – II

- Example:
  - 1.000.000 records of 300 B (including header)
  - the search key is a 4 byte integer and a pointer requires 4 bytes
  - 4 KB blocks, no block header, random placement,
    average retrieval time ~5.6 ms (Seagate Cheetah X15)
  - no time to find a record within a block when in memory

⇨ 13.6 records per block → 76924 blocks (unspanned) to store data

⇨ 512 indexes per block → 1954 blocks (unspanned) to store index

no index:

⇨ (76924 / 2) = 38462 block accesses (average)
time to find a record = 38462 * 5.6 ms = 215.4 s

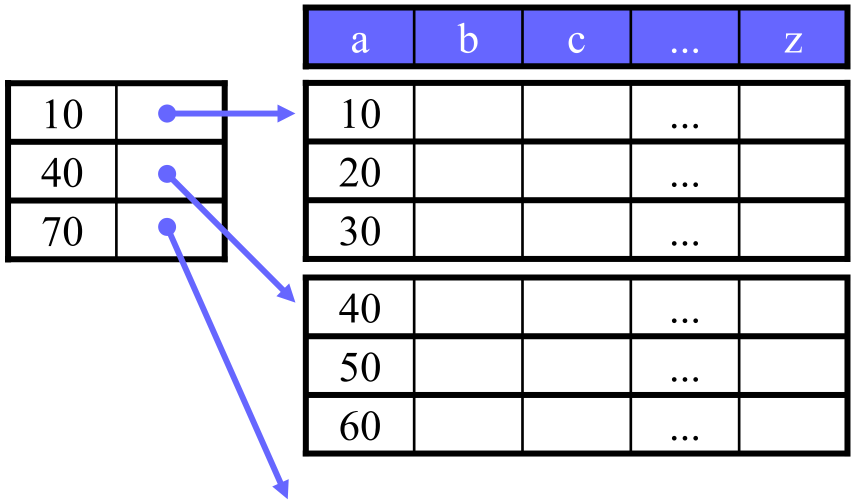
dense index, binary search:

⇨ $\lceil \log_2(1954) \rceil + 1 = 11 + 1 = 12$ block accesses (maximum)
time to find a record = 12 * 5.6 ms = 67.2 ms

⇨ using a dense index is 3205 times faster

Sparse Indexes – I

- ✓ A sparse index has one index entry for every data block, i.e., the key of the first record

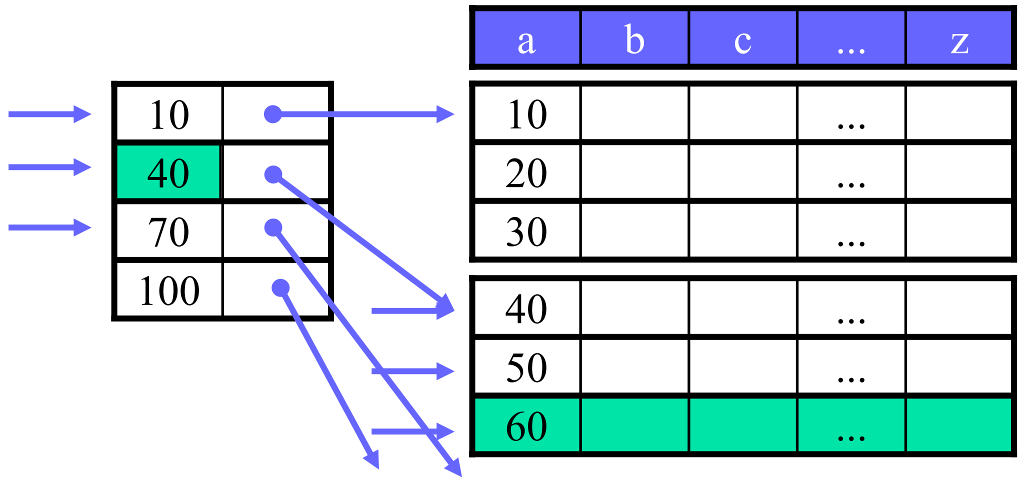


- ➢ only one index field per block, i.e., less index data
- ➢ cannot find out if a record exists only using the index
- ➢ to find a record K
  - search the index for the largest key less than or equal to K
  - retrieve the block pointed to by the index field
  - search within the block for the record

Sparse Indexes – II

- ✓ Example:

```
SELECT *
FROM   R
WHERE  a = 60
```



- ✓ Example (cont.):
  - in our previous dense index example, we needed 1954 blocks for the index (#records / # index fields per block)
  - using a sparse index we need only 151 (#data blocks / # index fields per block)

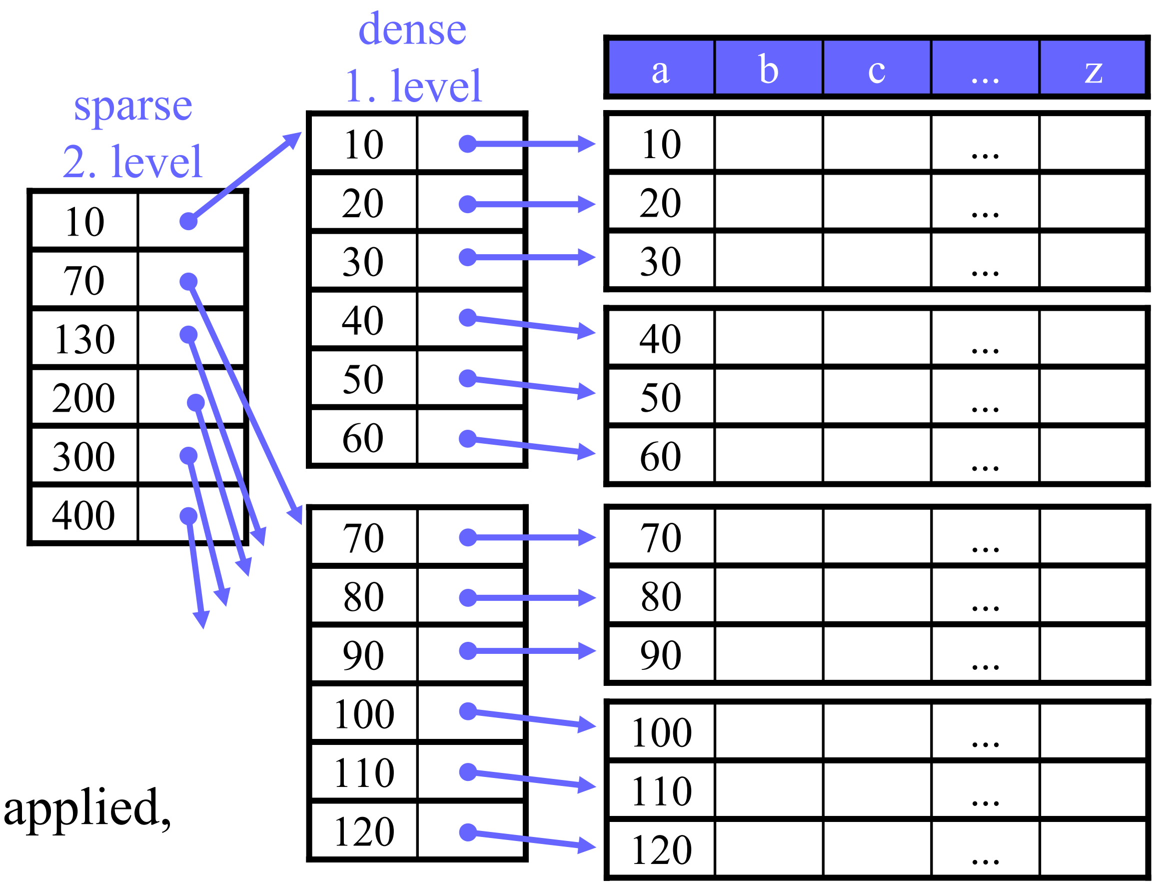
sparse index, binary search:

- ⇨ $\lceil \log_2(151) \rceil + 1 = 8 + 1 = 9$ block accesses (maximum)
  time to find a record = 9 * 5.6 ms = 50.4 ms
- ⇨ using a sparse index is 4272 times faster than using no indexes,
  1.33 times faster than dense indexes
- ⇨ However, sparse indexes must access the data block to see if a record exists

Multi-Level Indexes

- An index itself can span many blocks, i.e., still many block accesses
- Using a multi-level index is one approach to increase efficiency, i.e., using an index on the index

- Example (cont.):
  - need only 1954 / 512 = 4 for 2. level index blocks
  - need only $\lceil \log_2(4) \rceil + 1 + 1 =$ $2 + 1 + 1 = 4$ block accesses
  - time to find a record = 4 * 5.6 ms = 22.4 ms
  - 2.25 times faster than one-level sparse, 3 times faster than one-level dense

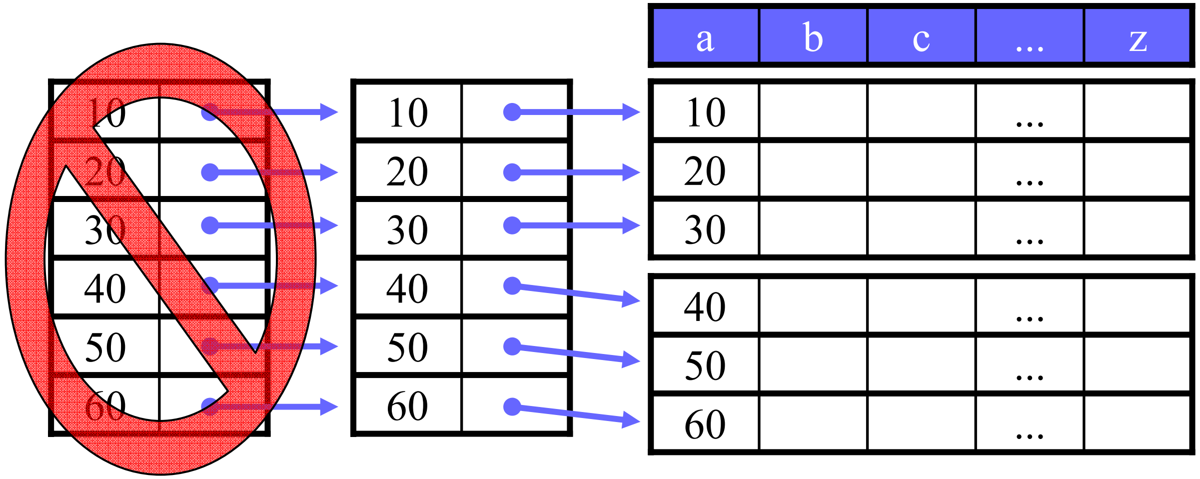


- Any levels of indexes may be applied, but the *idea has its limits*

Questions

✓ Can we build a dense, second level index for a dense index?

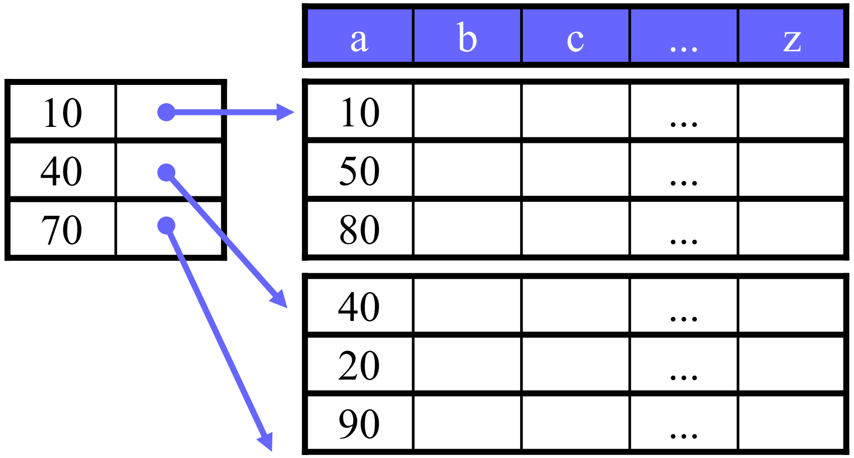
⇨ YES,
but it does ***not***
make sense



✓ Does it make sense to use a sparse index on an unsorted file?

⇨ NO,
how can one find records
that are not in the index

⇨ BUT,
one might use a sparse index on
a dense index on an unsorted file



Modifications

- An index file is a sequential file and must be treated in a similar way as a file of sorted records:
  - use overflow blocks
  - insert new blocks
  - slide elements to adjacent blocks
- A *dense index* points to the records, i.e.:
  - modified if a record is created, deleted, or moved
  - no actions must be done on block operations
- A *sparse index* points to the blocks, i.e.:
  - *may* be modified if a record is created, deleted or moved
  - no action must be done managing overflow blocks (pointers to primary blocks only)
  - must insert (delete) pointer to new (deleted) sequential block

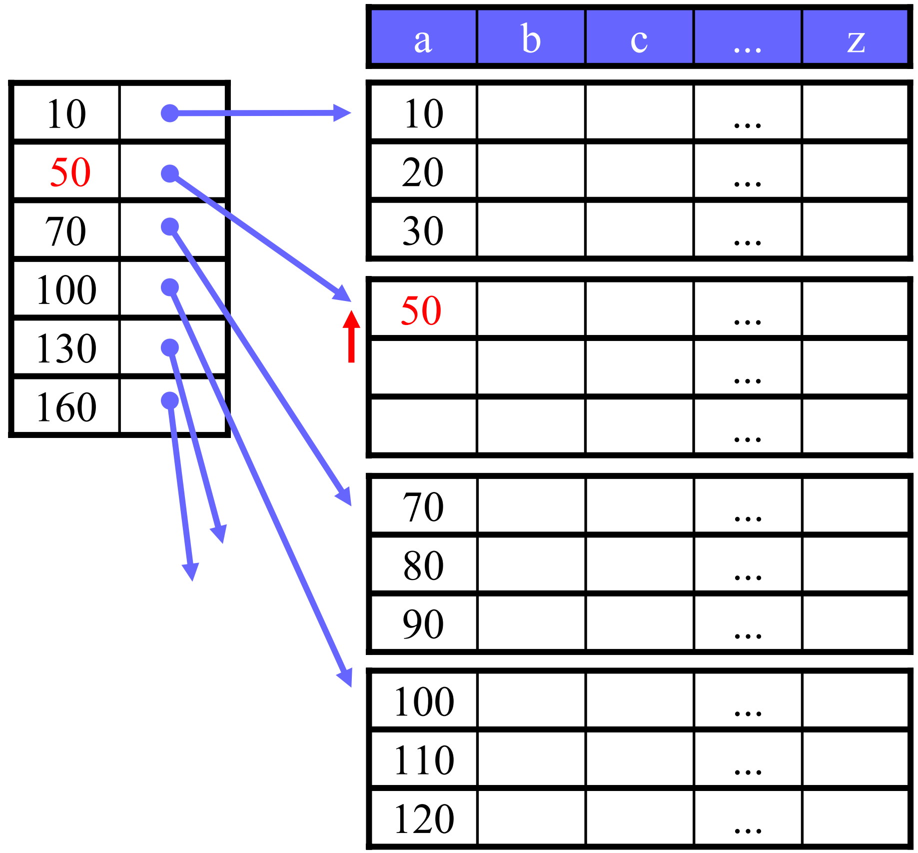
Modifications: Deletion Example 1

- ✓ Example – deletions using a sparse index:
  - ➢ delete record a = 60

    Note 1:
    as a sparse index points to the block, no action is required

  - ➢ delete record a = 40

    Note 2:
    the first record of the block has been updated, i.e., the index must also be updated



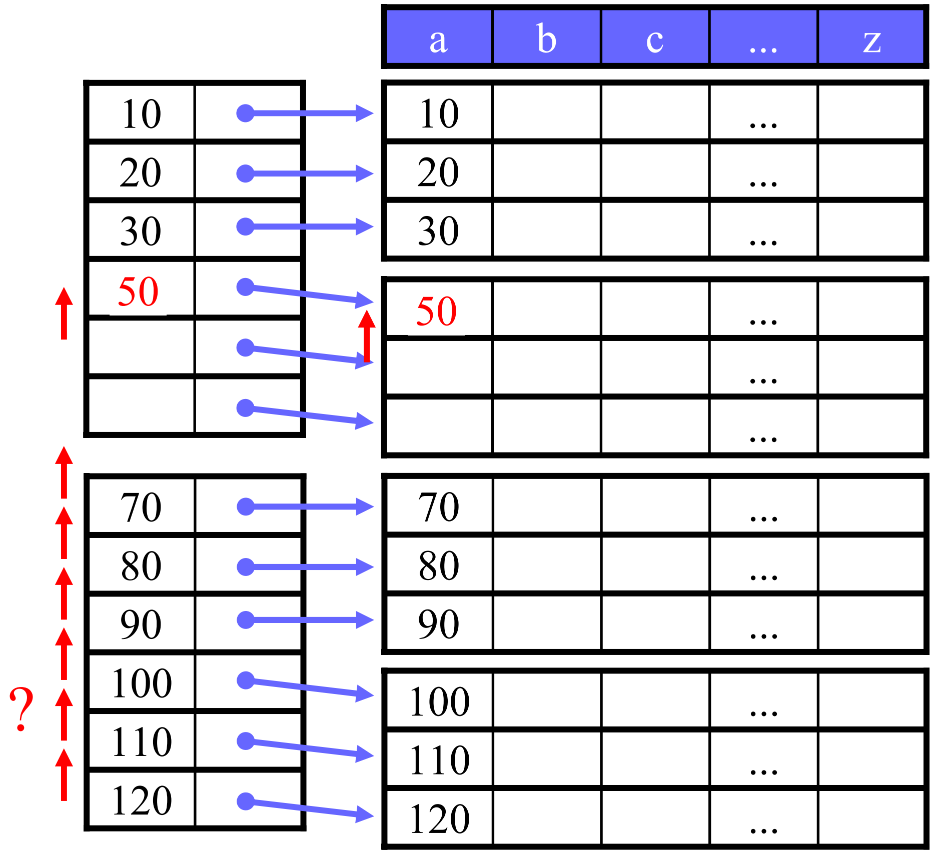
Modifications: Deletion Example 2

- Example – deletions using a dense index:
  - delete record a = 60
  - delete record a = 40

    Note 1:
    in many cases it is convenient to "compress" data in the blocks (optional)

    Note 2:
    one might compress the whole data set, but one usually keep some free space for future evolution of the data
  - delete record a = 50

    Note 3:
    the data block being empty might be deallocated or kept to enable faster insertions of new records



Modifications: Insertion Example 1

- ✓ Example – insertions using a sparse index:
  - ➢ insert record a = 60

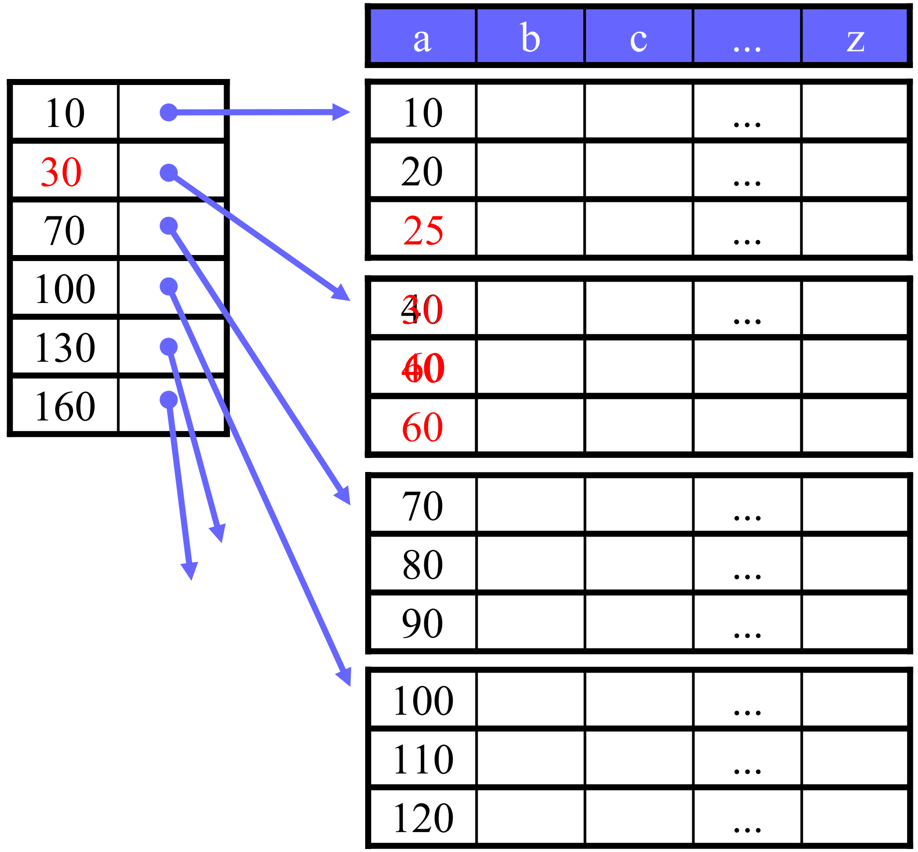
    Note 1:
    we are lucky – free space where we need it

  - ➢ insert record a = 25

    Note 2:
    record a = 25 should go into first block, has to move record a = 30 to second block where we have room

    Note 3:
    first record of block 2 has changes, must also update index

    Note 4:
    instead of sliding record a = 30, we might have inserted a new block or an overflow block



Modifications: Insertion Example 2

- ✓ Example – insertions using a sparse index:
  - ➢ insert record a = 95

    Note 1:
    no available room – insert overflow block or new sequential block

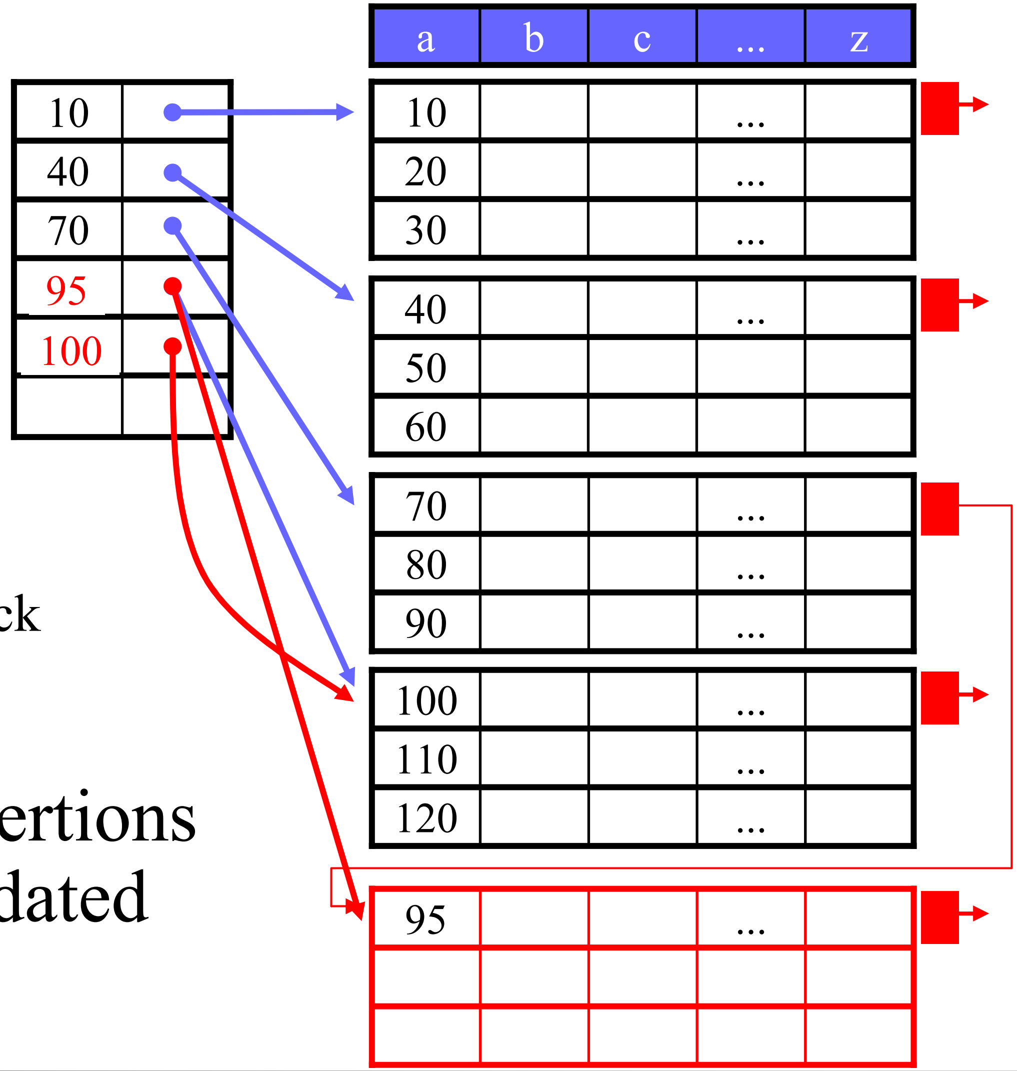
    - overflow block

      Note 2:
      no actions are required in the index, sparse indexes only have pointers to primary blocks

    - new sequential (primary) block

      Note 3:
      must update index

- ✓ Dense indexes manage insertions similarly – but must be updated each time



Dense vs. Sparse Indexes

	Dense	Sparse
space	one index field per record	one index field per data block
block accesses	"many"	"few"
record access	direct access	must search within block
"exist queries"	use index only	must always access block
use	anywhere (not dense-dense)	not on unordered elements
modification	always updated if the order of records change	updated only if first record in block is changed

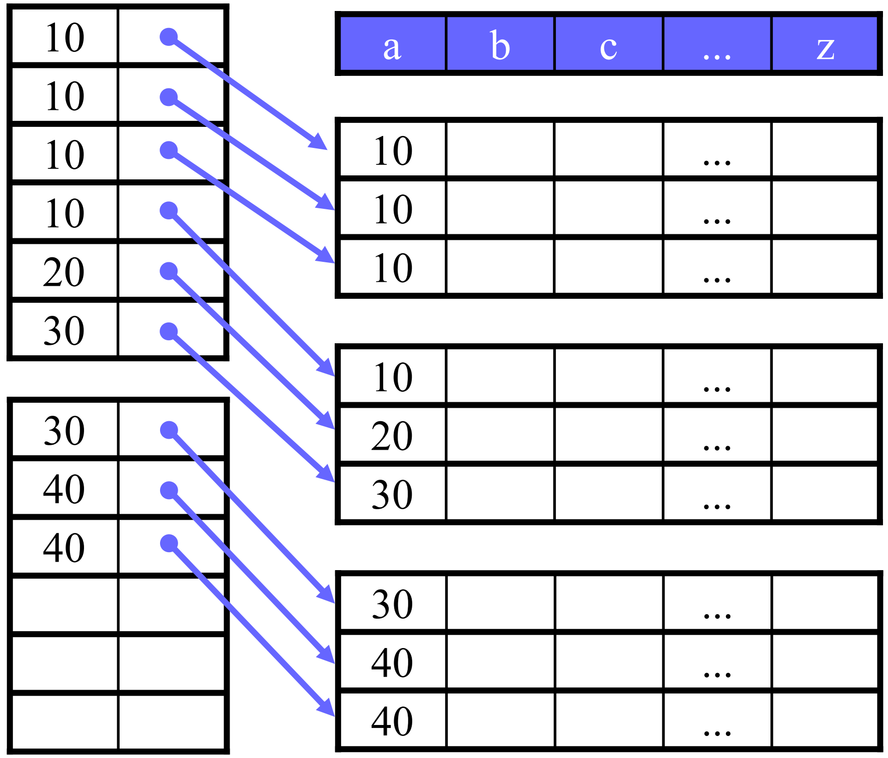
Duplicate Search Keys (Cluster Indexes) – I

- So far we have looked at indexes where the search key has been unique – if records also are sorted, the index is called a *primary* index
- Indexes are also used on non-key attributes where duplicate values are allowed – if records in addition are sorted, the index is called a *cluster* index
- In general, if the records are sorted by the search key, the previous ideas may be applied
- Many ways to implement such an index:
  - dense index with one index field
    - per record (pointer to all duplicates)
    - unique search key (pointer to only the first record)
  - sparse index

Duplicate Search Keys (Cluster Indexes) – II

✓ Example 1 – dense index:

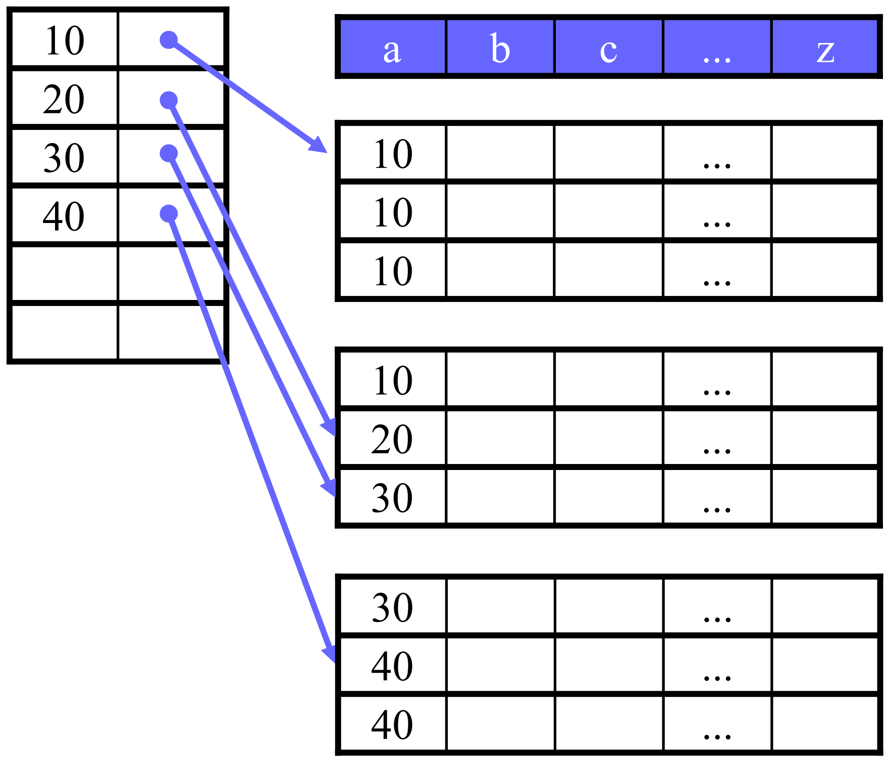
- ➢ one index field per record
- ☺ easy to find records and how many
- ☹ more fields than necessary?? – index itself spans more disk blocks



Duplicate Search Keys (Cluster Indexes) – III

✓ Example 2 – dense index:

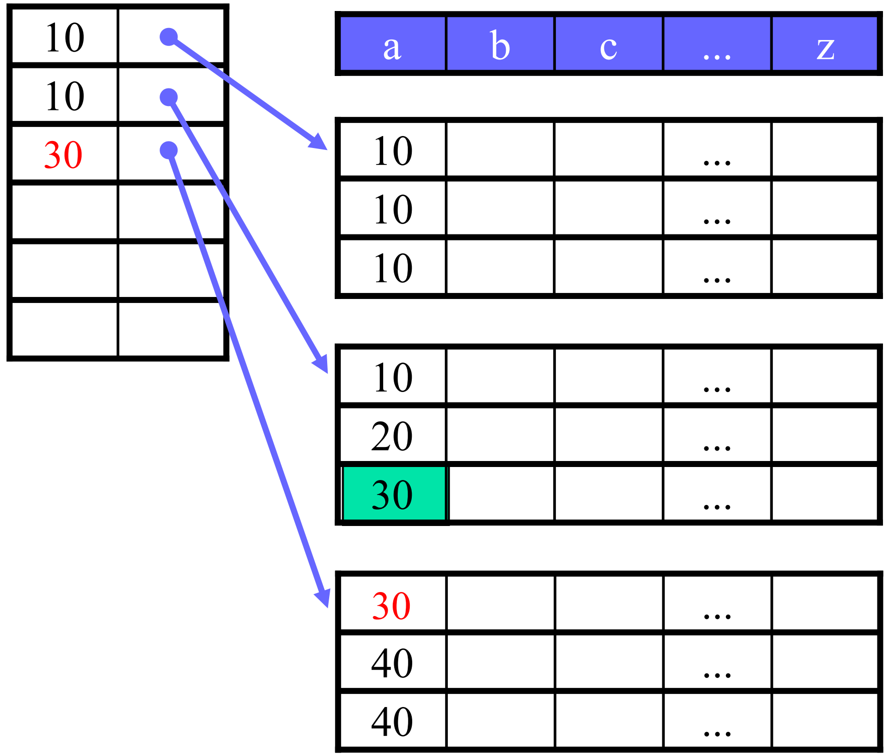
- ➢ only one index field per unique search key
- ☺ smaller index – quick search
- ☹ more complicated to find successive records



Duplicate Search Keys (Cluster Indexes) – IV

✓ Example 3 – sparse index:

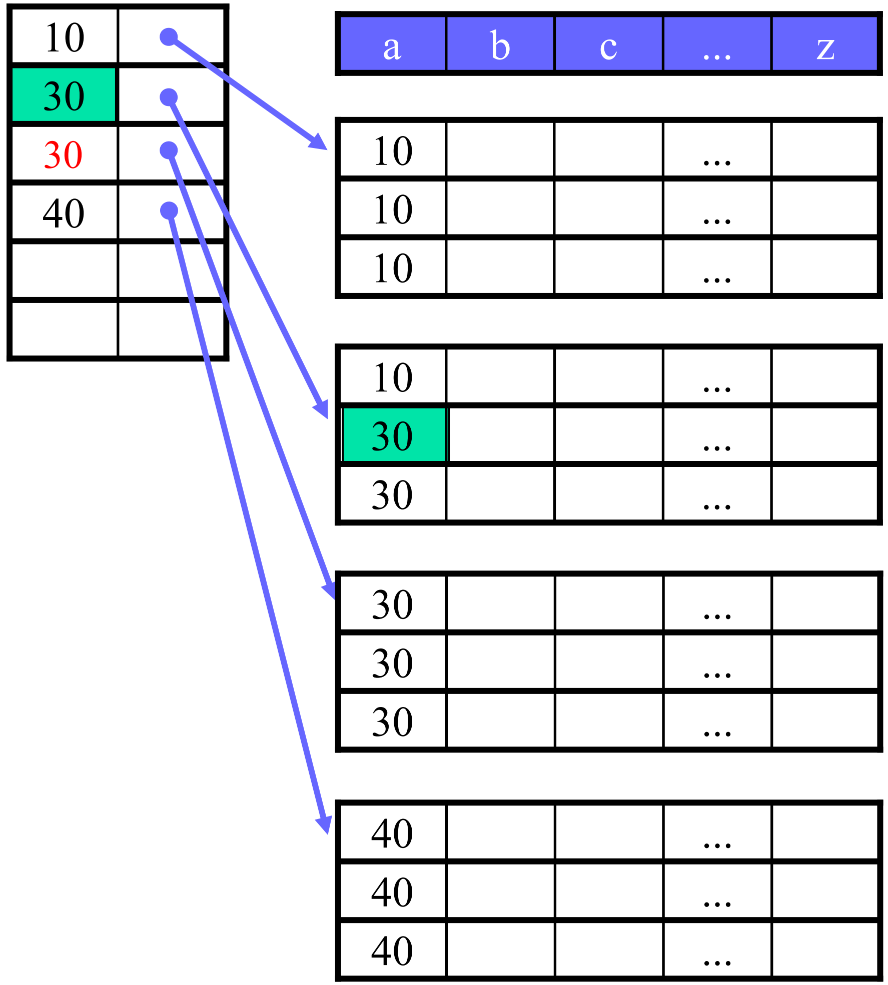
- ➢ index field is first record in each block, pointer to block
- ☺ small index – fast search
- ☹ complicated to find records
- ➢ e.g., must be careful if looking for 30



Duplicate Search Keys (Cluster Indexes) – V

✓ Example 4 – sparse index:

- index field is first *new* record in each block, pointer to block
- ☺ small index – fast search
- ☹ complicated to find records
- can we even remove the second index entry of 30

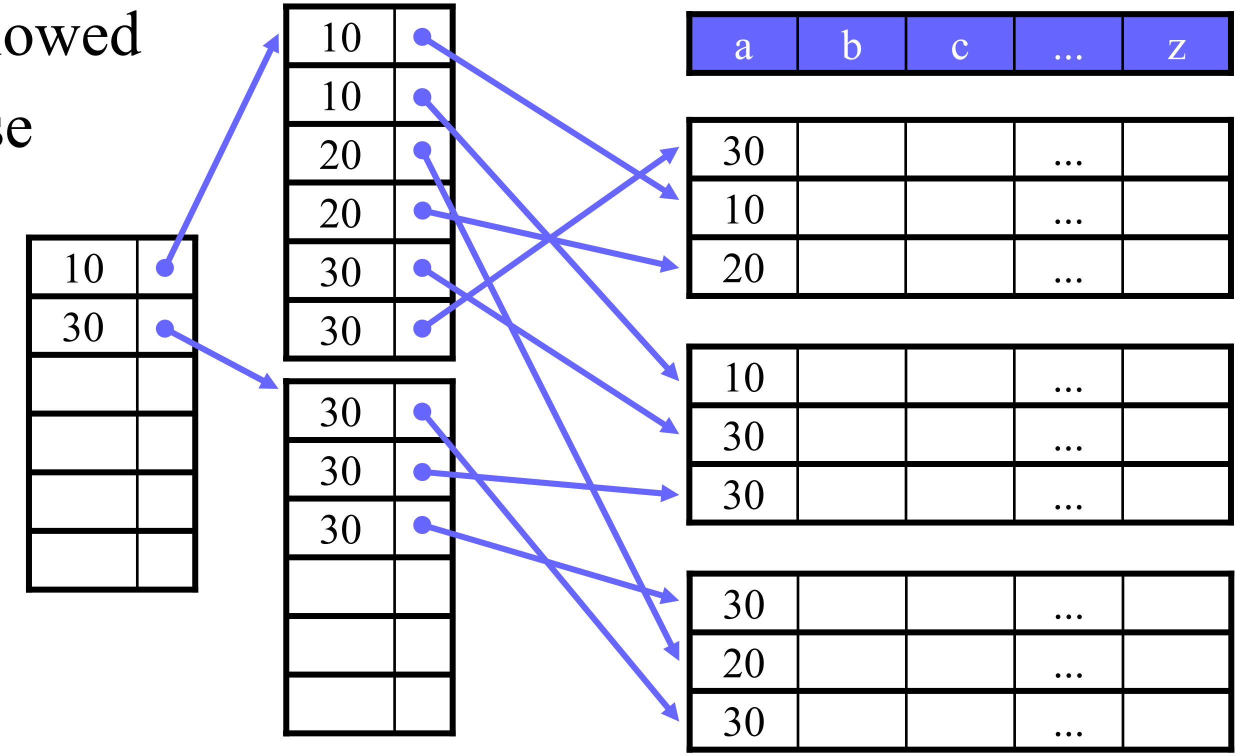


Unsorted Record Indexes (Secondary Indexes) – I

- ✓ Both primary and cluster indexes work on sorted files, i.e., underlying file is sorted on the search key
- ✓ What if we want several indexes on same file?
- ⇨ *Secondary* indexes
  - ➢ works on unsorted records, i.e.,
    does not determine placement of records in file
  - ➢ works like any other index – find a record fast
  - ➢ first level is always dense – any higher levels are sparse
  - ➢ duplicates are allowed
  - ➢ index itself is sorted on the search key value – easy to search

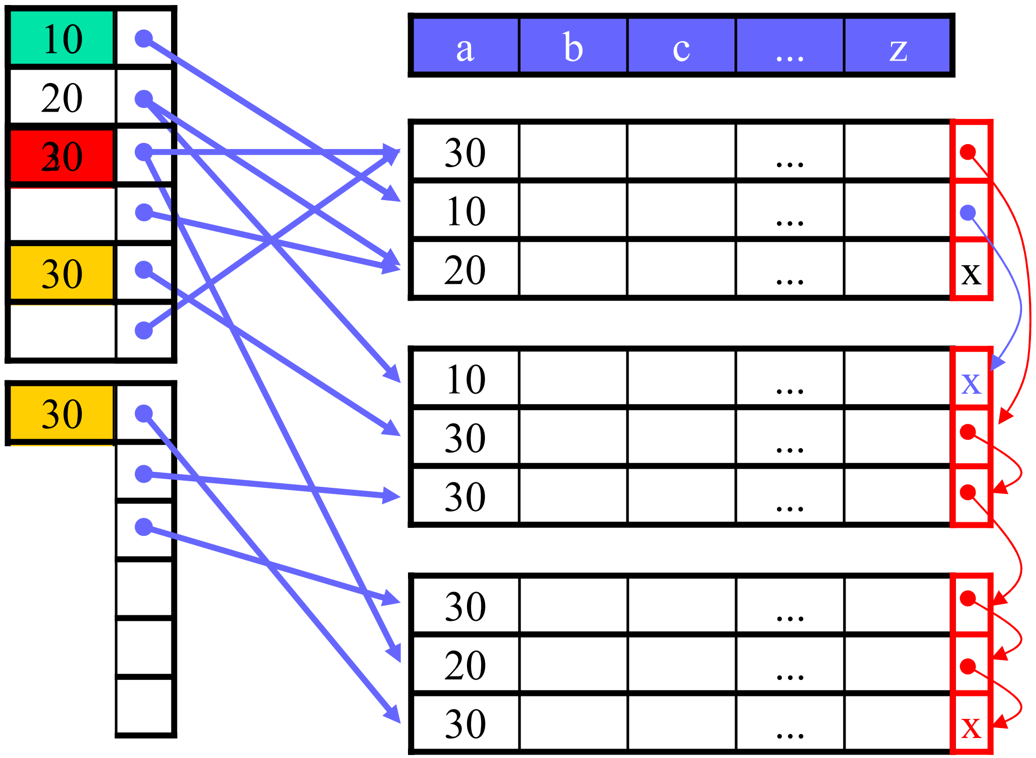
Unsorted Record Indexes (Secondary Indexes) – II

- Example:
  - duplicates are allowed
  - first level is dense
  - higher levels are sparse
  - index is sorted



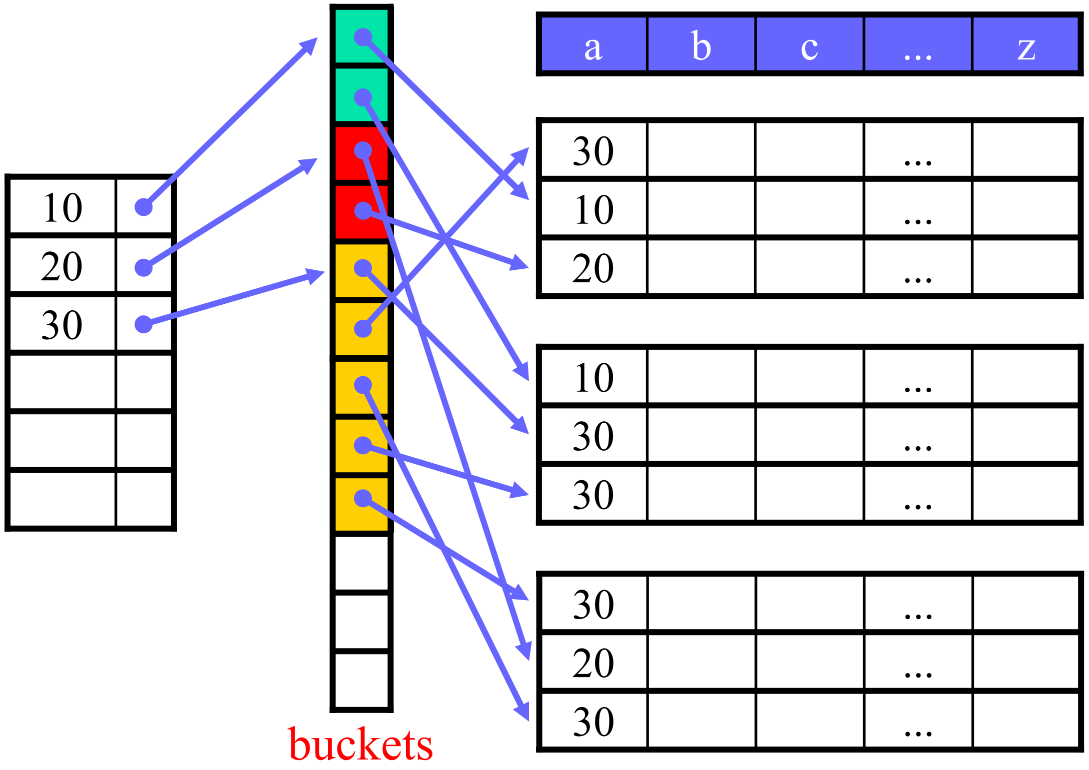
Unsorted Record Indexes (Secondary Indexes) – III

- ✓ Duplicate search keys may introduce overhead, both *space* and *search time*
- ✓ Variable sizes index fields
  - ☺ saves space in index
  - ☹ complex design and search
- ✓ Chain records with same search key
  - ☺ simple index, easy search
  - ☹ add fields to records header
  - ☹ follow chain to successive records



Unsorted Record Indexes (Secondary Indexes) – IV

- Buckets are a convenient way to avoid repeating values, by using indirection
  - designated bucket file
  - index entry for K points to first element in bucket for K
  - manage bucket file as other sorted files

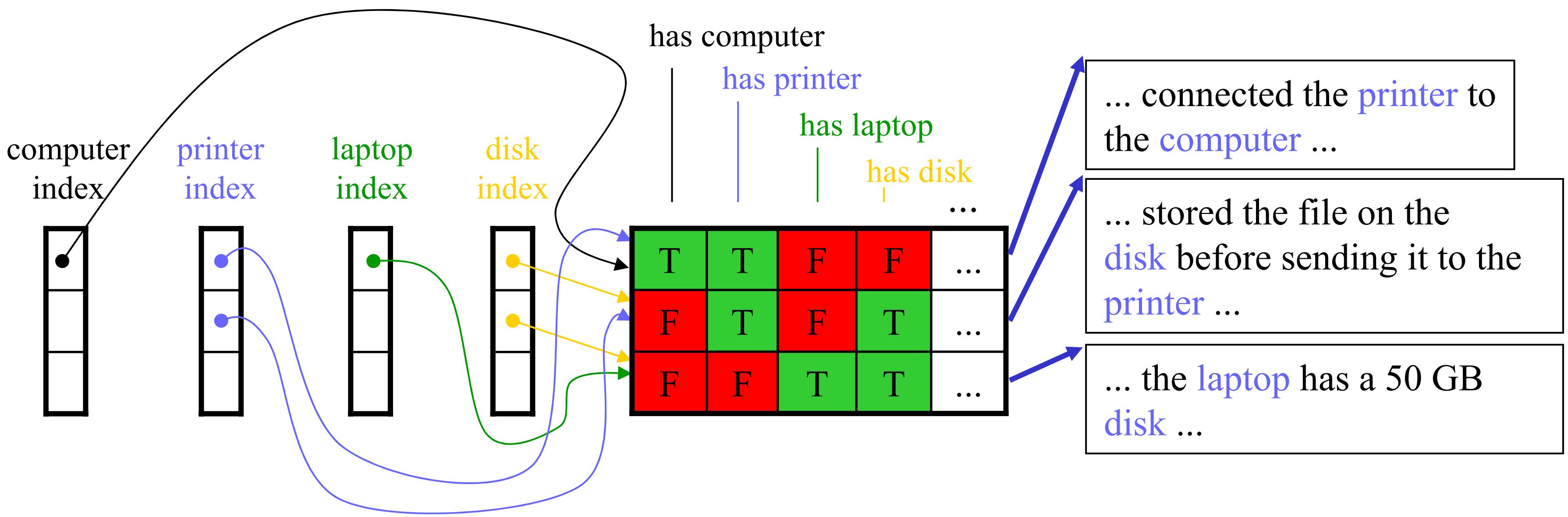


Inverted Indexes – I

- ✓ Indexes so far uses whole attribute as search key
- ✓ Do we need indexes on single elements within an attribute?
  - `SELECT * FROM R WHERE a LIKE '%cat%'`
  - searching for documents containing specific keywords, e.g., web search engines like Google, Altavista, Excite, Infoseek, Lycos, Fast (AllTheWeb), etc.
- ✓ How can one make such indexes?

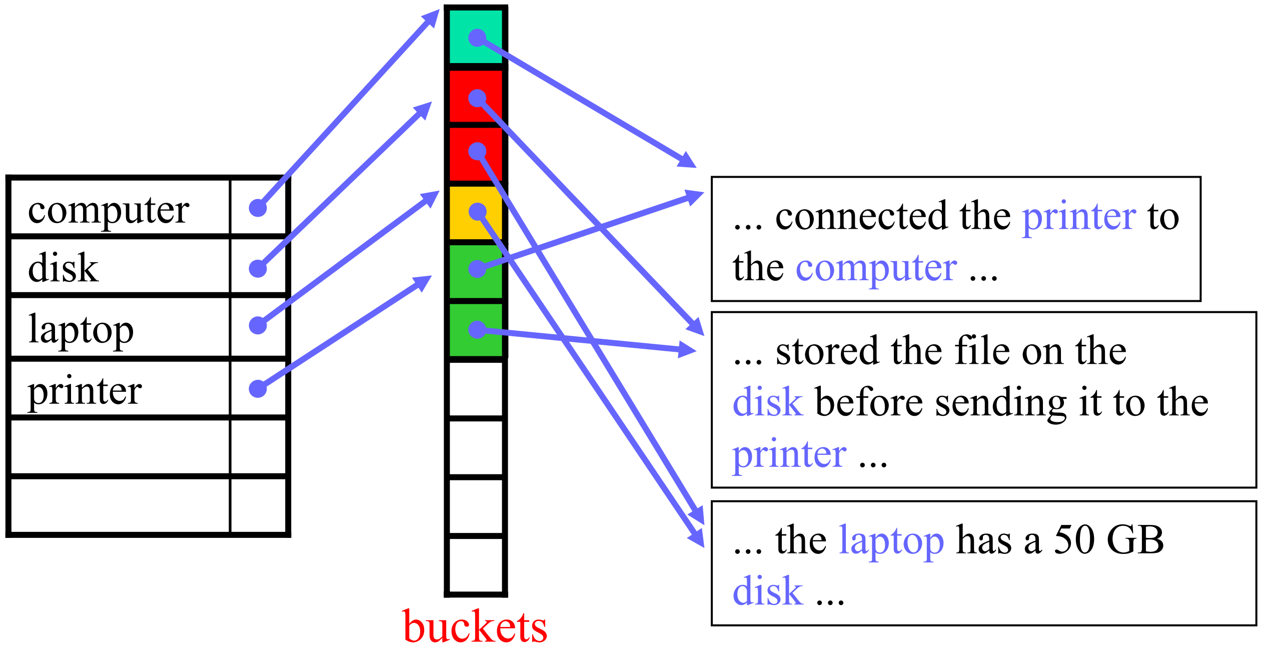
Inverted Indexes – II

- Approach 1 – true/false table:
  - define “all” keywords
  - make table with one boolean attribute for each keyword
  - make one tuple per document/text attribute
  - make index on all attributes including only `TRUE` values
- Example: allowed keywords – computer, printer, laptop, disk, ...



Inverted Indexes – III

- ✓ Approach 2 – inverted index:
  - make one (inverted) index for keywords
  - use indirection putting pointers in buckets
- ✓ Example:



Conventional Indexes

- ✓ Conventional indexes:
  - ☺ simple
  - ☺ index is a sequential file, good for scans
  - ☹ inserts and movements expensive
  - ☹ no balance – varying number of operations to find a record
- ✓ Conventional indexes – one or two levels – are often helpful speeding up queries, but they are *usually not used in commercial systems....*

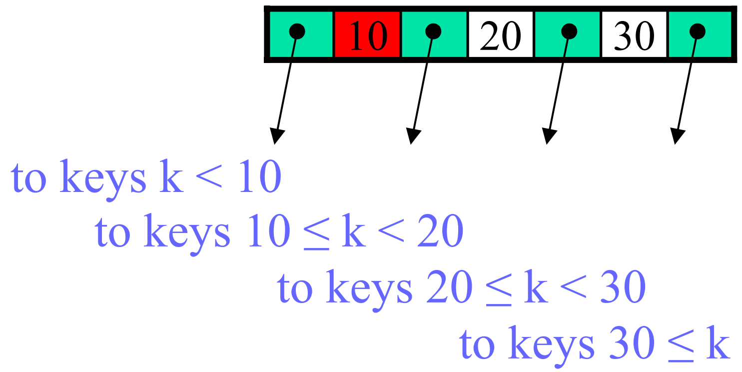
B-Trees

B-Trees

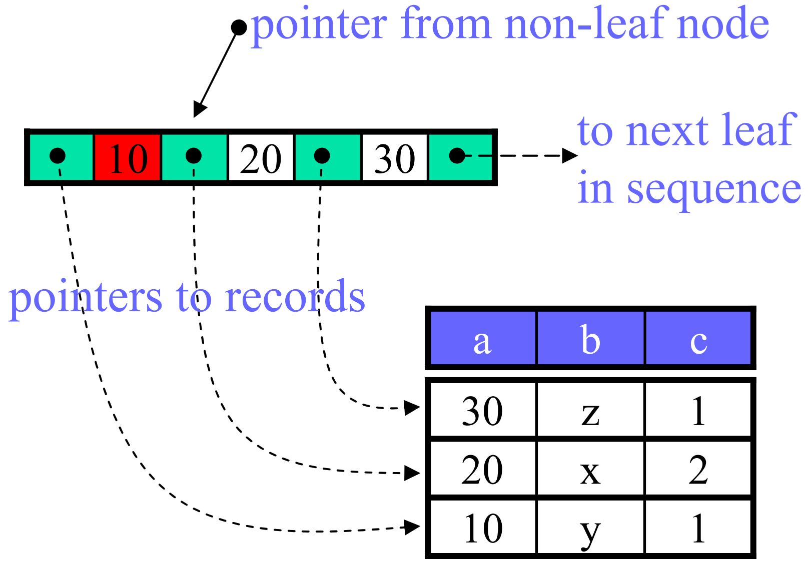
- B-trees is a general structure which is frequently used
  - automatically maintains an appropriate number of levels
  - manages the space in a block (usually between half used and full), i.e., no overflow blocks needed
  - balanced – all leaves at the same level
- Many different types of B-trees – B^+-tree
  - in a B-tree, all search keys and corresponding data pointers are represented somewhere in the tree, but in a B^+-tree, all data pointers appear in the leaves (sorted from left to right)
  - nodes have *n* search keys and *n + 1* pointers
    - in intermediate nodes, all pointers are to other sub-nodes
    - in a leaf node, there are *n* data pointers and *1* next pointer
  - nodes are not allowed to be empty, use at least
    - intermediate node: $\lceil (n+1)/2 \rceil$ pointers to subnodes
    - leaf node: $\lfloor (n+1)/2 \rfloor$ pointers to data

B+-Trees – I

- ✓ Intermediate node (n = 3)
  (all pointers to sub-nodes)
  - ➢ left side pointer of key is pointer to sub-node with smaller keys
  - ➢ right side pointer of key is pointer to sub-node with equal or larger keys

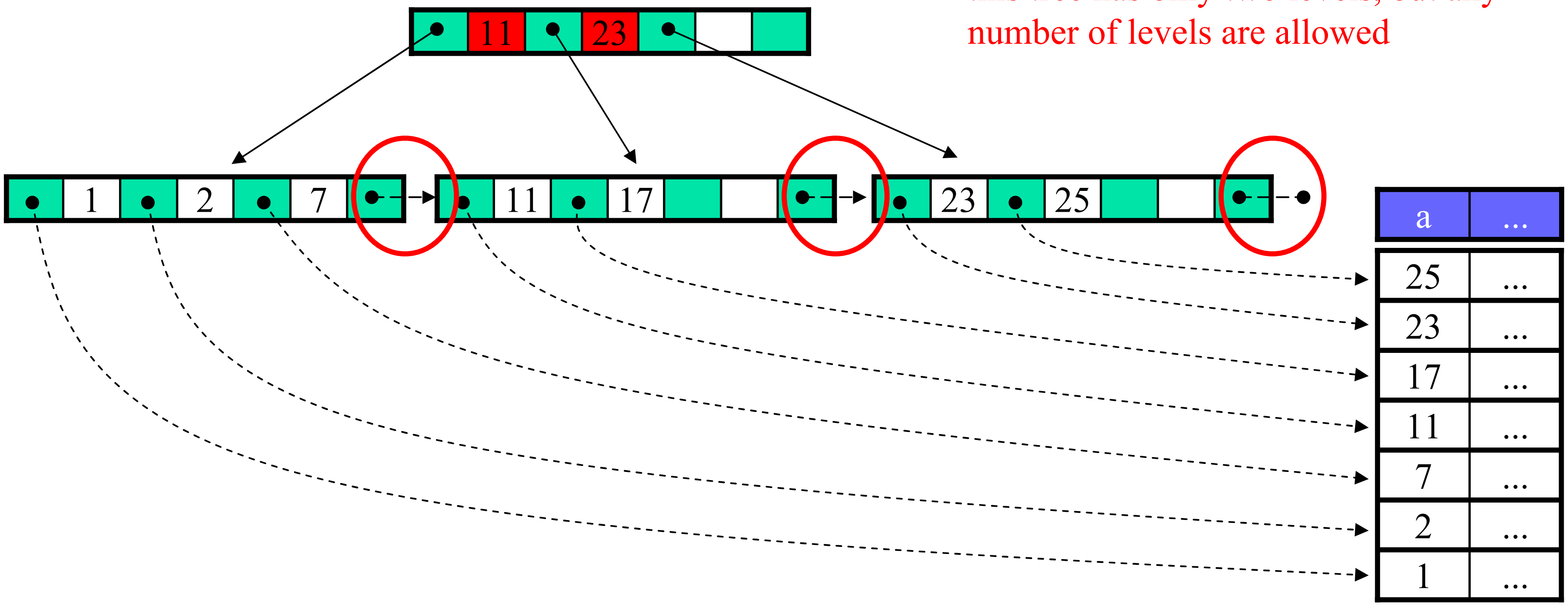


- ✓ Leaf node (n = 3)
  (n data pointers, 1 next pointer)
  - ➢ left side pointer of key is pointer to the key's record
  - ➢ last pointer is a pointer to next leaf in the sequence



B^+-Trees – II

- ✓ Total tree (n = 3):



Note 1:
this tree has only two levels, but any number of levels are allowed

Note 2:
since the leaves are linked it is easy to read sequences

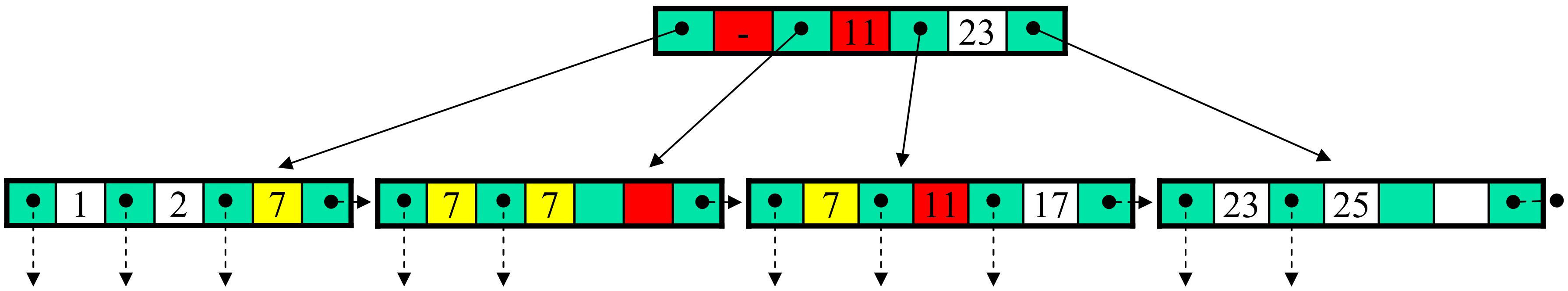
Note 3:
all leaves (and data pointers) are at the same level

B^+-Trees: Operations

- Lookup:
  - intermediate: use the pointer rules recursively to next child, i.e., left key pointer < K, right key pointer ≤ K
  - leaf:
    - dense index:
      - if the i-th key is K, then the i-th pointer points to requested record
      - if key K is not present, the record with search key K does not exist
    - sparse index:
      - find the largest search key less than or equal to K
      - retrieve the block pointed to by the index field
      - search within the block for the record
- Insertion/deletion (see textbook for details):
  - split/merge nodes if necessary to keep minimum requirement of pointers in each node

B^+-Trees: Duplicates

✓ Allowing duplicates:



Note 1:
an intermediate node points to the first occurrence of a duplicate search key, subsequent occurrences are found reading next node (leaves are sorted)

Note 2:
pointer K_i will now be the smallest "new" key that appear in the sub-tree pointed to by pointer *i + 1*

Note 3:
in some situations, pointer K_i can be null, e.g., cannot put 7 in first pointer in root

Note 4:
a node may not be full – even in a sequence of duplicate search keys

B^+-Trees: Applications

- B^+-trees may be used several ways - the sequence of pointers in the leaves can "implement" all the index types we have looked at so far

- *Leaf nodes* can for example act as a
  - dense or sparse primary index
  - dense or sparse cluster index – allowing duplicates
  - (dense) secondary index – unsorted records
  - ...

- *Intermediate nodes* are used to speed up search

B^+-Trees: Efficiency – I

- B^+-trees:
  - ☹ a search always needs to go from root to leaf, i.e., number of block accesses is the height of tree plus accesses for record manipulation
  - ☺ number of levels usually small – 3 is a typical number
  - ☺ range queries are very fast – find first, read sequentially
  - ☺ if *n* is large, splitting and merging will be rare, i.e., can usually neglect reorganization costs
  - ☺ disk I/O's may be reduced by pinning index blocks in memory, e.g., root is always available in main memory

⇨ Only a few disk I/O's (or block accesses) are needed for an operation on a record

B^+-Trees: Efficiency – II

- ✓ Example 1:
  assume integer keys (4 B) and 8 B pointers
  storage system uses 4 KB blocks – no headers
  how many values may be stored in each node?

  $4n + 8(n+1) \leq 4096 \;\Rightarrow\; n = \underline{340}$

- ✓ Example 2:
  a node is on average 75 % filled
  how many records does a 3-level B^+-tree hold?

  $(340 * 75\,\%)^3 = 165813375 \approx \underline{16.6 \text{ million}}$ records

Hash Tables

Hash Tables – I

- ✓ Hash tables is a another structure useful for indexes

  - a fixed size array containing the search keys-pointer pairs
  - secondary storage hash tables often have one bucket per block with "support" for overflow blocks
  - often main memory array of pointers to the bucket blocks
  - size of array usually a prime number
  - uses a *hash function* to map search key value to array index

Hash Tables – II

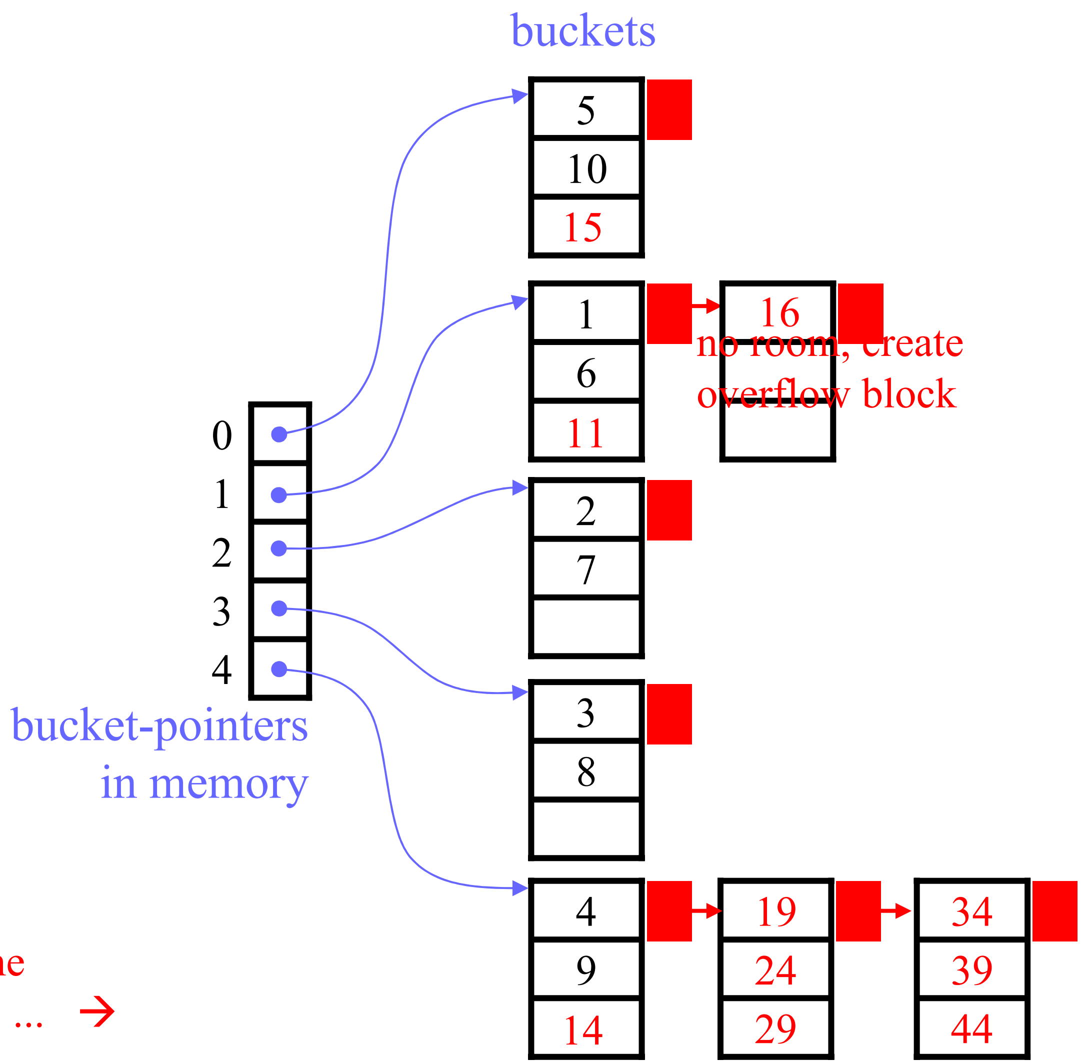
✓ A good hash functions is important

- should be easy to compute (fast)
- should distribute the search keys evenly – each bucket should have approximately the same number of elements
- common function examples:
  - integer keys:
    `mod(key value, array size)`
  - character keys:
    `mod(sum characters as integer, array size)`
- best function varies between different uses

Hash Tables – III

- ✓ Example:
  - array size B = 5
  - h(key) = mod(key, B)

insert: 11 → h(11) = 1
14 → h(14) = 4
15 → h(15) = 0
16 → h(16) = 1



Note:
a problem occurs if most keys map to same bucket, e.g., adding 19, 24, 29, 34, 39, 44, ... → not an appropriate hash function

Hash Tables – IV

- ✓ Should records be sorted within a bucket?
  - ➢ YES, if search time (CPU time) is critical
  - ➢ NO, if records are frequently inserted/deleted
- ✓ Operations are "straight forward" – calculate hash value, and ...
  - ➢ ... insert into corresponding bucket - manage overflow if needed
  - ➢ ... delete from corresponding bucket – may *optionally* consolidate blocks if possible
- ✓ Ideally, the array size is large enough to keep all elements in one bucket-block per hash value, i.e., size and hash function must be carefully chosen
  - ☺ if so, the number of disk I/O's significantly better compared to straightforward indexes and B-trees
  - ☺ fast on queries selecting one particular key value
  - ☹ however, as the number of records increase, we might have several blocks per bucket
  - ☹ range queries will be slow as subsequent keys go to subsequent buckets

Dynamic Hash Tables

- ✓ Hard to keep all elements within one bucket-block if file grows and hash table is *static*
- ✓ *Dynamic* hash tables allow the size of the table to vary, i.e., may keep one block per bucket for performance
  - extensible hashing
  - linear hashing

Extensible Hashing – I

- ✓ Extensible hashing:
  - always an array of pointers
  - pointer array may grow
    - length is power of 2
    - double size when more space needed
  - certain buckets may share a block to reduce space – if so, block header contains an indication of this
  - hash function computes a hash value which is a static bit-sequence of *b* bits– the number if used bits *i*, however, varies dynamically after the size of the pointer array

    e.g.:

    b

    h(key) → 1 1 0 1 0 1 0 1

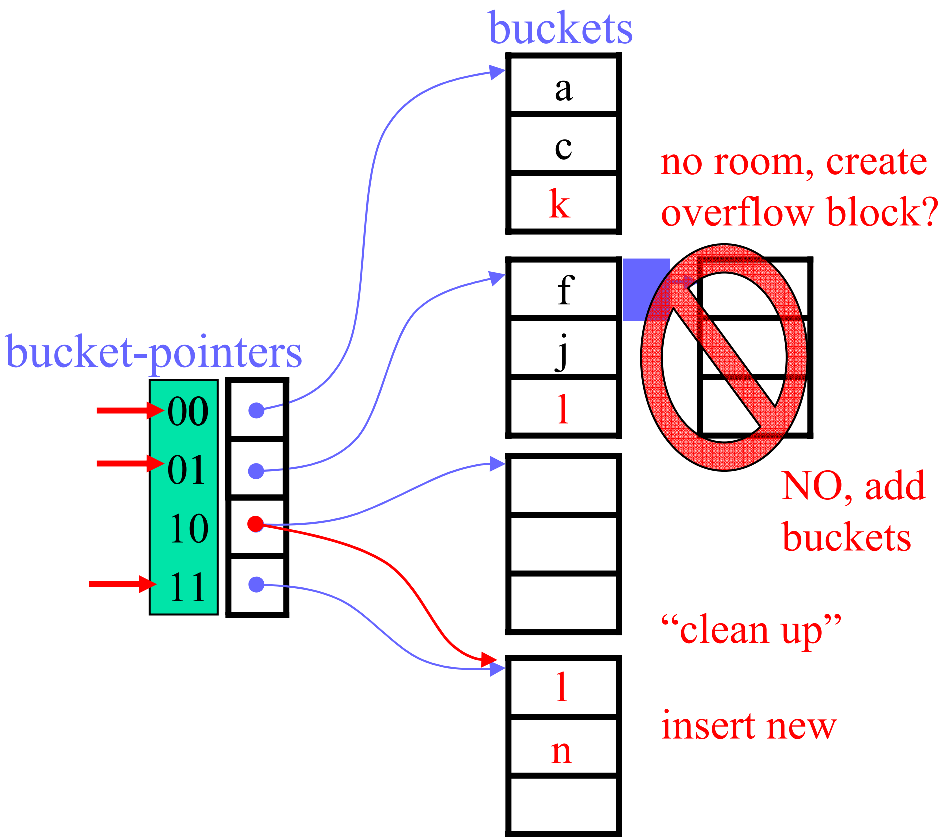
    i

Note 1:
b is static and in this example 8, i.e., we may maximum have an array size of 2^8.

Note 2:
i is dynamic and in this example 4, i.e., we currently using an array size of 2^4. However, if more buckets are needed, we double the size and increase i to 5.

Extensible Hashing – II

- ✓ Example:
  - b = 4, i = 1 increase i → 2
  - insert record with key k, h(k) → 0 0 1 0
  - insert record with key l, h(l) → 1 1 0 0
  - insert record with key n, h(n) → 1 1 0 1
  - bucket 10 and 11 may share a block to save space



Extensible Hashing – III

- ✓ Extensible hashing:
  - ☺ manage growing files
  - ☺ still one block per bucket (fast)
  - ☹ indirection expensive if pointer array is on disk
  - ☹ doubling size
    - much work to be done, especially if i is large
    - as size increases, it may no longer fit in memory

Linear Hashing – I

✓ Linear hashing:

- number of buckets are determined by the average fill level
- hash function similar to extensible hashing, but using the low order bits, i.e.,
  the $\lceil \log_2 n \rceil$ low order bits where n is number of buckets
- inserts:
  - find correct bucket and insert – use overflow block if necessary
  - if block do not exist, put element in bucket $m - 2^{i-1}$, i.e., change the first bit to 0, m is the value of the used bits form the hash value
  - if average fill level reaches limit, add one bucket (not related to inserted item)
  - if now n exceeds 2^i, i is incremented by one

Linear Hashing – II

✓ Linear hashing:

☺ manage growing files

☺ do not need indirection

☹ can still have overflow chains

☹ some work moving elements from “re-directed” bucket when creating a new

Sequential vs Hash Indexes

- ✓ *Sequential indexes* like B^+-trees is good for range queries:

  `SELECT * FROM R WHERE R.A > 5`

- ✓ *Hash indexes* are good for probes given specific key:

  `SELECT * FROM R WHERE R.A = 5`

Indexes in SQL

- Syntax:
  - `CREATE INDEX` name `ON` relation_name (attribute)
  - `CREATE UNIQUE INDEX` name `ON` relation_name (attribute)
    → defines a candidate key
  - `DROP INDEX` name
- Note: cannot specify
  - type of index, e.g., B-tree, hashing, etc.
  - parameters such as load factor, hash size, etc.

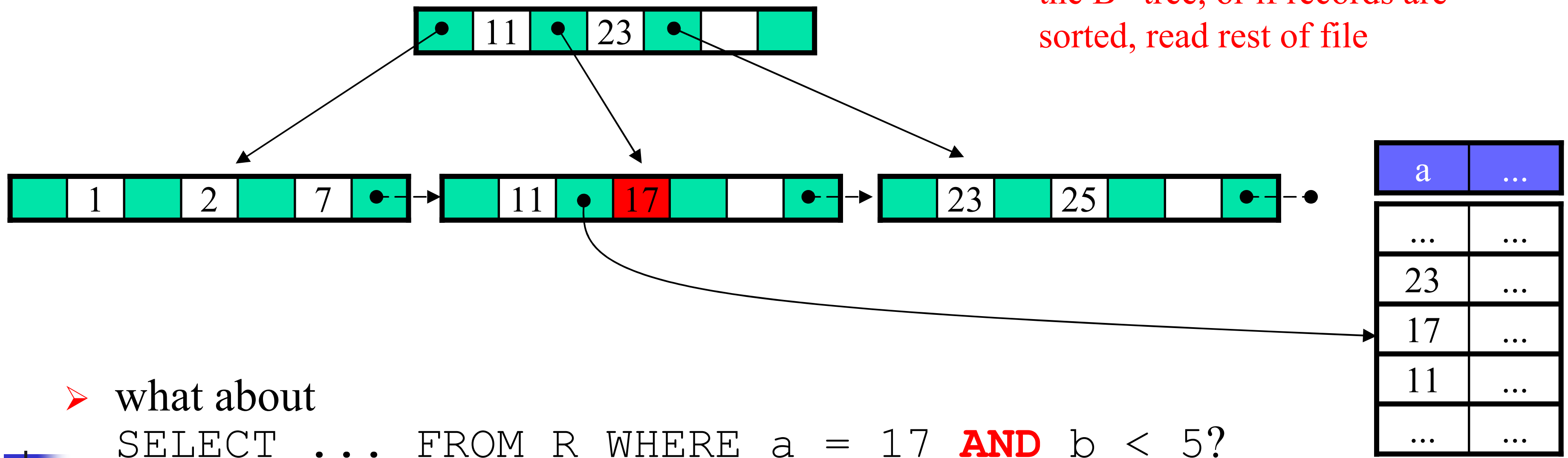
  ⇨ index type is chosen by the people implementing the system
  ... at least in SQL...

  ⇨ But, IBM's *DB2* have some choices (not according to the SQL standard)

Using Indexes in Queries – I

- Indexes are used to quickly find a record
- Some queries may even be solved without reading the records from disk, e.g., find the number of elements
  → count the number of pointers in the index (dense)
- Index structures discussed so far are *one dimensional*
  - one single search key
  - good for queries like
    `SELECT ... FROM R WHERE` **a ≥ 17**
    e.g.: using a B^+-tree index on `a`

Note:
in a range query, we may just read subsequent elements in the B^+-tree, or if records are sorted, read rest of file



  - what about
    `SELECT ... FROM R WHERE a = 17` **AND** `b < 5?`

Using Indexes in Queries – II

- Strategy 1:
  `SELECT ... FROM R WHERE a = 30` **`AND`** `b < 5`

  - use one index, say on a
  - find and *retrieve* all records where a = 30 using the index
  - search these records for b values less than 5

  ☺ simple and easy approach

  ☹ may read a lot of records not needed from disk

Using Indexes in Queries – III

- ✓ Strategy 2:
  `SELECT ... FROM R WHERE a = 30 AND b < 5`

  - use two dense indexes – on `a` and `b`
  - find all pointers in the first index where a = 30
  - find all pointers in the second index where b < 5
  - manipulate pointers – compare (intersect) pointers and retrieve the records where the pointers match

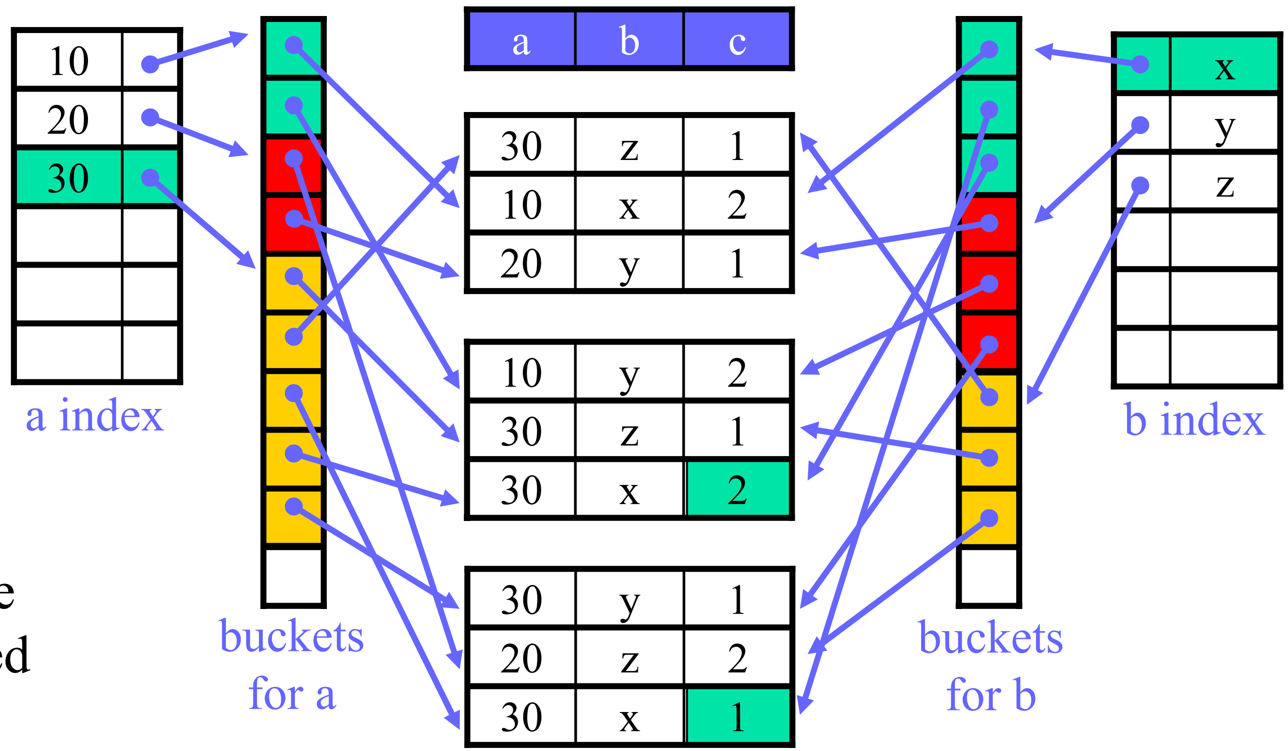
☺ may reduce data block accesses compared to strategy 1

☹ search two indexes (or more if more conditions)

☹ cannot sort records on two attributes (for two indexes)

Using Indexes in Queries - IV

- Example - strategy 2, using pointer buckets:
  `SELECT c FROM R WHERE a=30 AND b='x'`
  - find records where a is 30 using index on a
  - find records where b is 'x' using index on b
  - have two set of record-pointers
  - compare (intersect) pointers and retrieve the records requested
  - select specified attributes



Using Indexes in Queries - V

- ✓ *One dimensional* indexes can be uses many places, but in case of operations with several search conditions, it may be ....
  - ... many indexes to search
  - ... hard to keep efficient record order on disk for several indexes
  - ...
- ✓ This has lead to the idea of ...

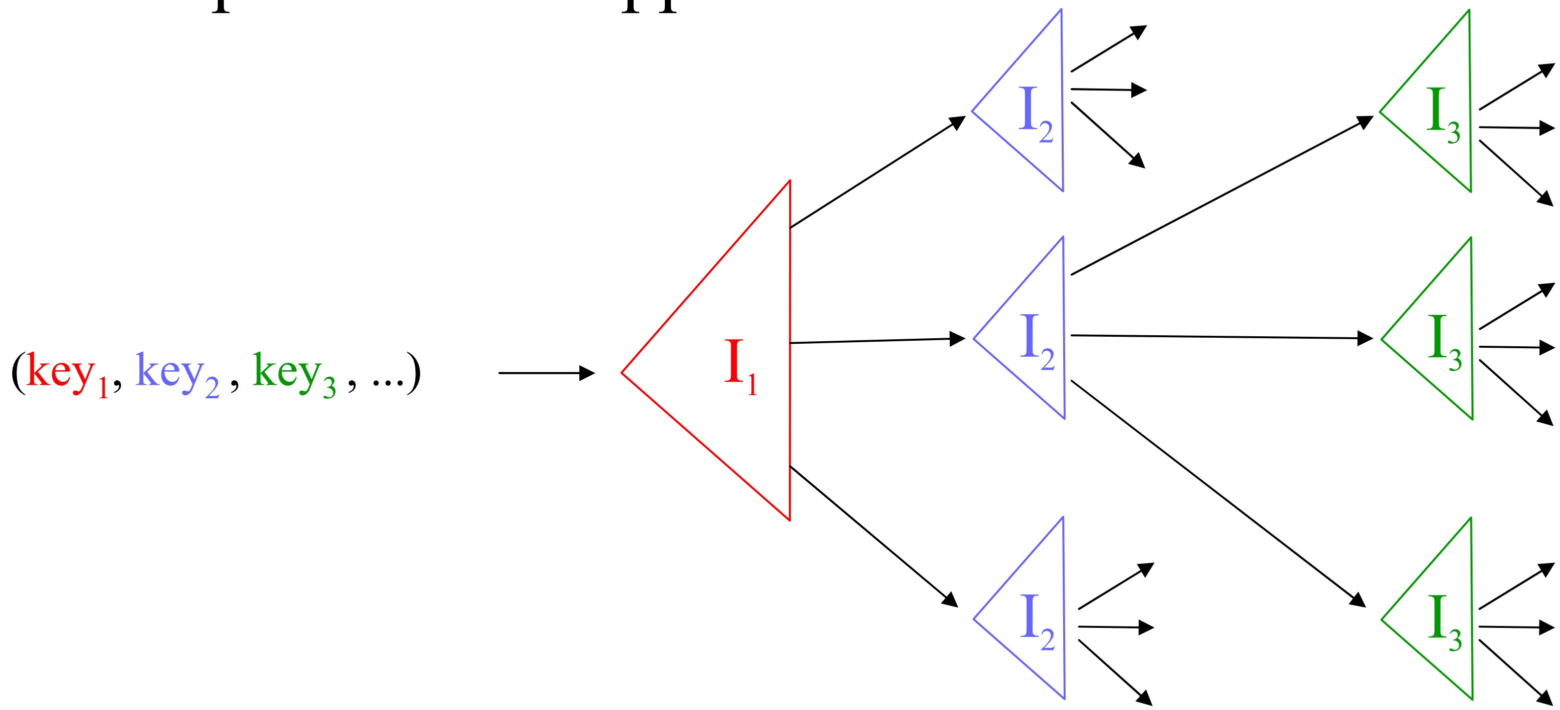
... ***MULTIDIMENSIONAL INDEXES***

Multidimensional Indexes

Multidimensional Indexes – I

- A multidimensional index combines several dimensions into one index

- One simple tree-like approach:

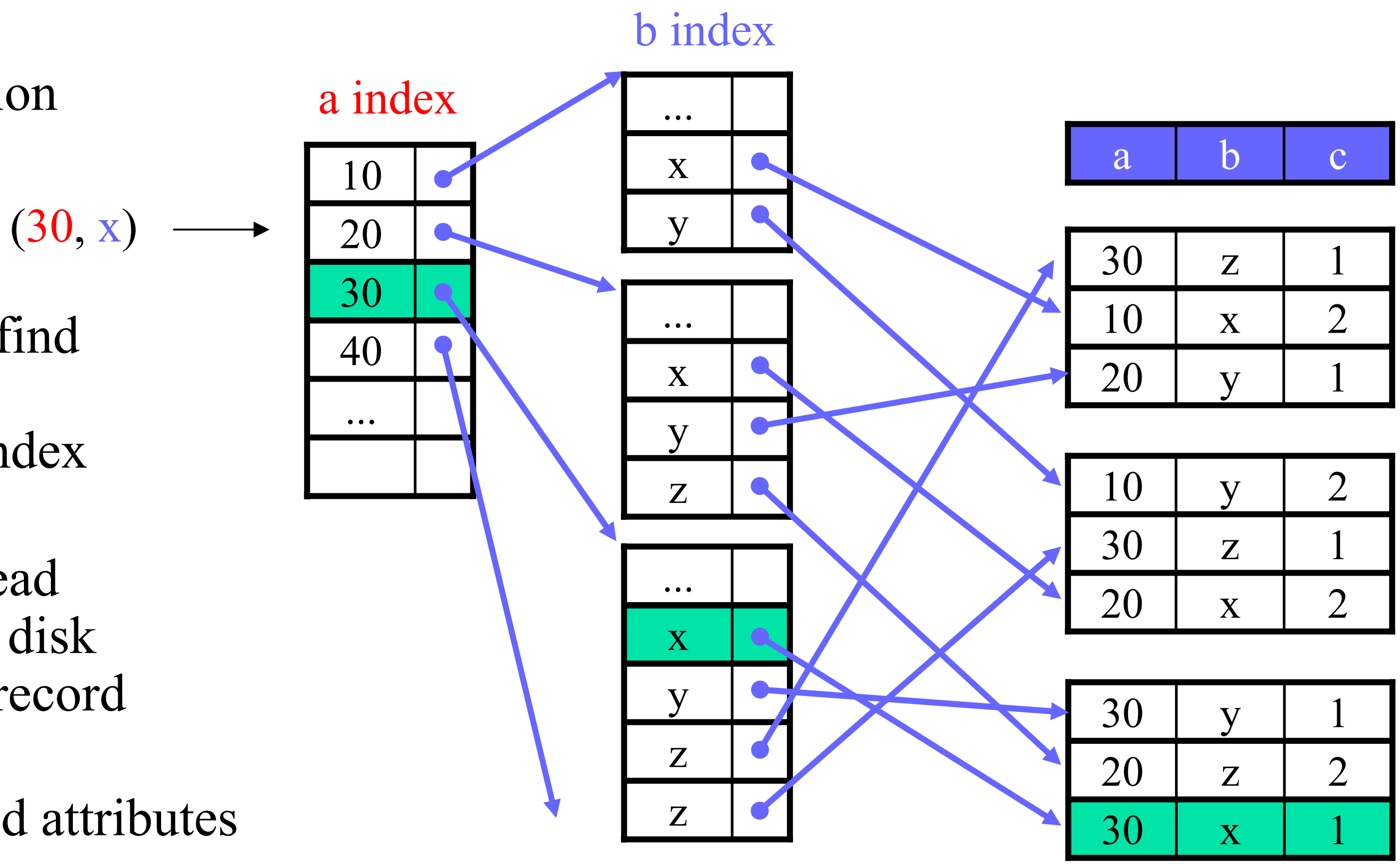


Multidimensional Indexes – II

✓ Example – multidimensional, dense index:

`SELECT ... FROM R WHERE a = 30 AND b = 'x'`

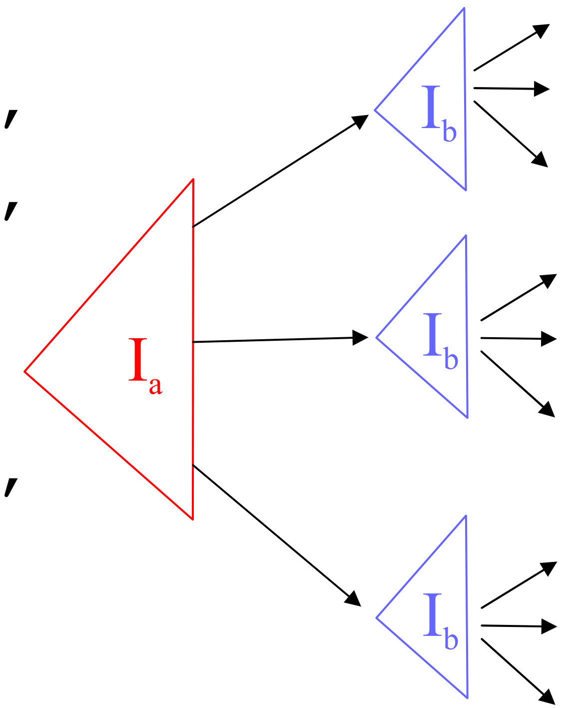
- search key = (30, x)
- read a-dimension
- search for 30, find corresponding b-dimension index
- search for x, read corresponding disk block and get record
- select requested attributes



Multidimensional Indexes – III

✓ For which queries is this index good?

- ☺ find records where `a = 10 AND b = 'x'`
- ☺ find records where `a = 10 AND b ≥ 'x'`
- ☹ find records where `a = 10`
- ☹ find records where `b = 'x'`
- ? find records where `a ≥ 10 AND b = 'x'`
  - may search several indexes in next dimension
  - better if dimensions changed order

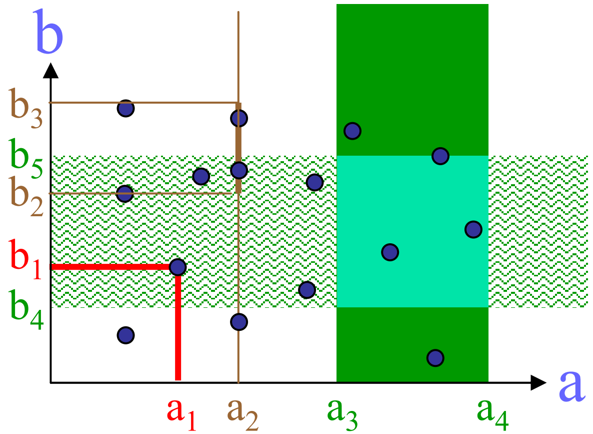


✓ Several other approaches....

- other tree-like structures
- hash-like structures
- (bitmap-indexes)

Map View

- A multidimensional index may have several dimensions
- If we assume only two (as in our previous example), we can imagine that our index is a geographical map:



- Now our search is similar to searching 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 several tree structures that works in a similar way to finding map-areas, e.g.,
  - kd-trees
  - quad-trees
  - R-trees

- However, these approaches *give up at least one* of the following important properties of B-trees:
  - balance of tree – all nodes at same level
  - correspondence of tree nodes and disk blocks
  - performance of modification operations

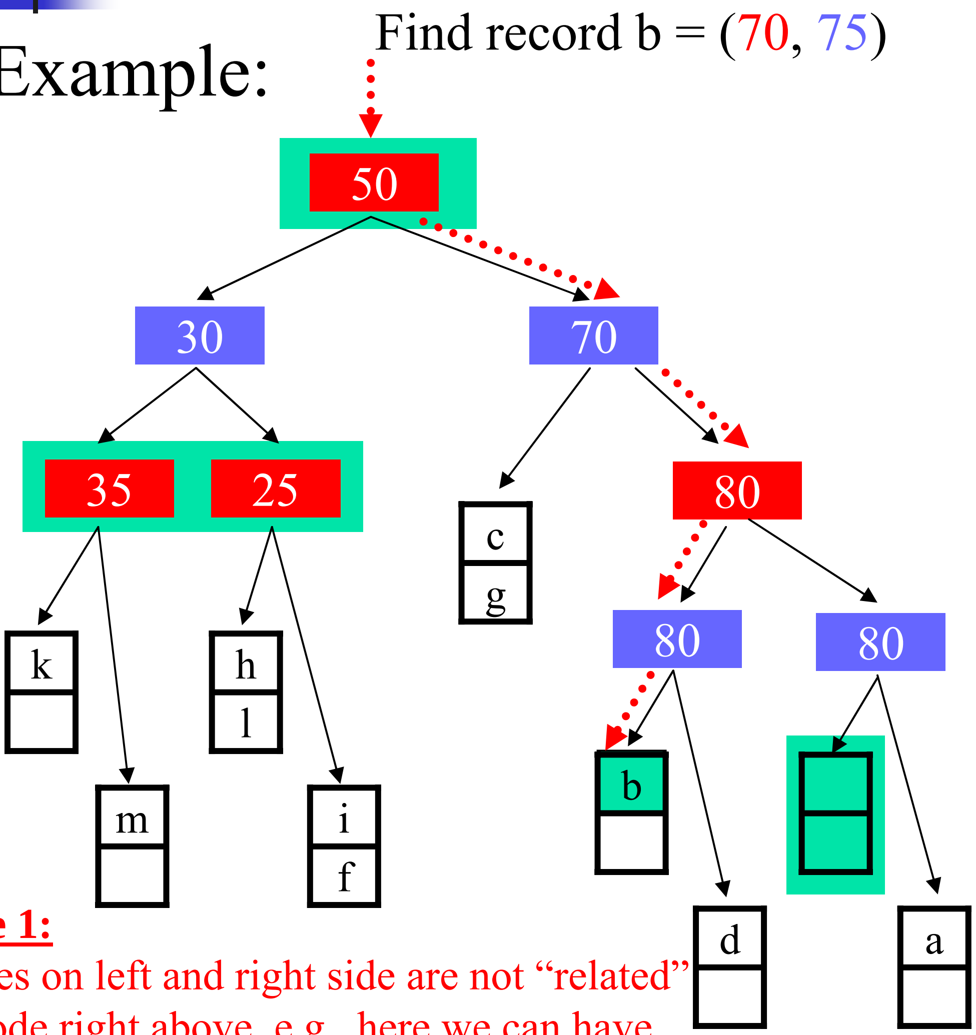
kd - trees – I

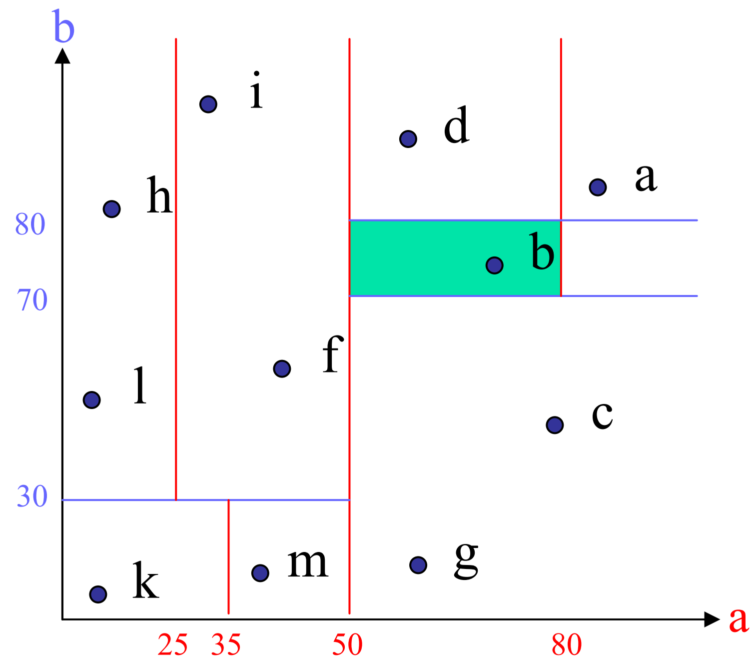
- A kd-tree (k-dimensional tree) is a binary tree having nodes with
  - an attribute a
  - a value V splitting the remaining data points in two
  - ⇨ data points are placed in nodes as in a traditional binary tree
- We will look at modification of the kd-tree where
  - interior nodes only have the V value
    - left child is have values less than V
    - right child have values equal or larger than V
  - leaves are blocks with room for as many record the block can hold
- The different dimensions are interleaved:

level 0	a-dimension
level 1	b-dimension
...	...
level n	a-dimension
level n+1	b-dimension
...	...

kd - trees – II

- ✓ Example:





Note 1:
values on left and right side are not "related" to node right above, e.g., here we can have any value as long as it is below 50

Note 2:
may have empty buckets

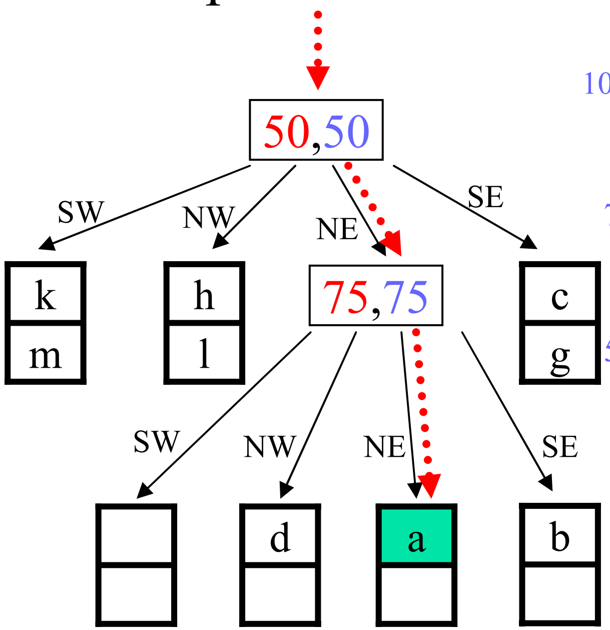
Note 3:
an easy extension is to have multiple values in each node – multiple branches

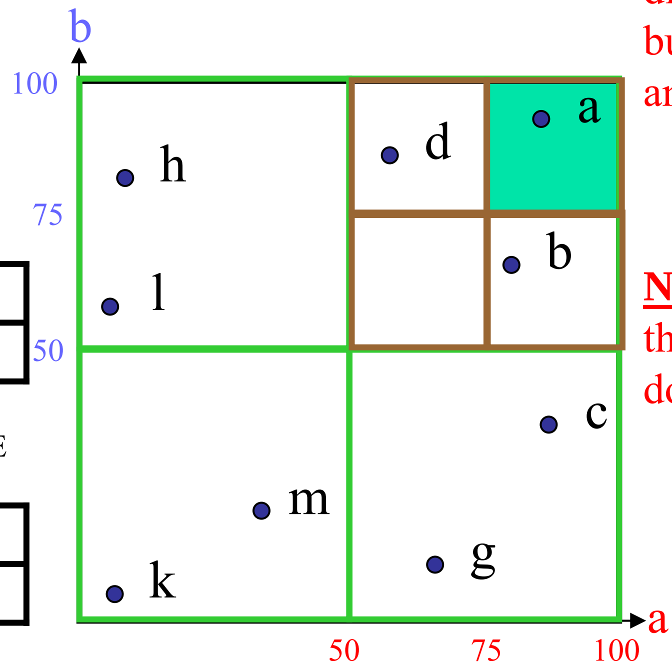
kd - trees – III

- Some “problems”:
  - may have to check both branches of the root, e.g., for b-dimension condition
  - may have to check both branches of the sub-tree in range queries, e.g., want values <10,30> and node value is 20
  - higher trees compared to B-trees
    - storing each node in an own block is expensive
    - much more block accesses compared to B-trees
  - number of block accesses is reduced if we
    - use multiway branches
    - group interior nodes into one block, e.g., node and several sub-levels of the tree in one block

Quad - trees

- Quad-trees:
  - the interior nodes corresponds to a *k*-dimensional cube in *k* dimensions and it has 2^k children
  - the data pointers is in the leaf nodes
- In our 2-dimensional example, a quad-tree's interior node divide the area in four equal square regions
- Example:





Note 1:
divide area further if no room in bucket - we have 2-element buckets and NE-square has 3 elements

Note 2:
the value range of index attributes does not need to be the same

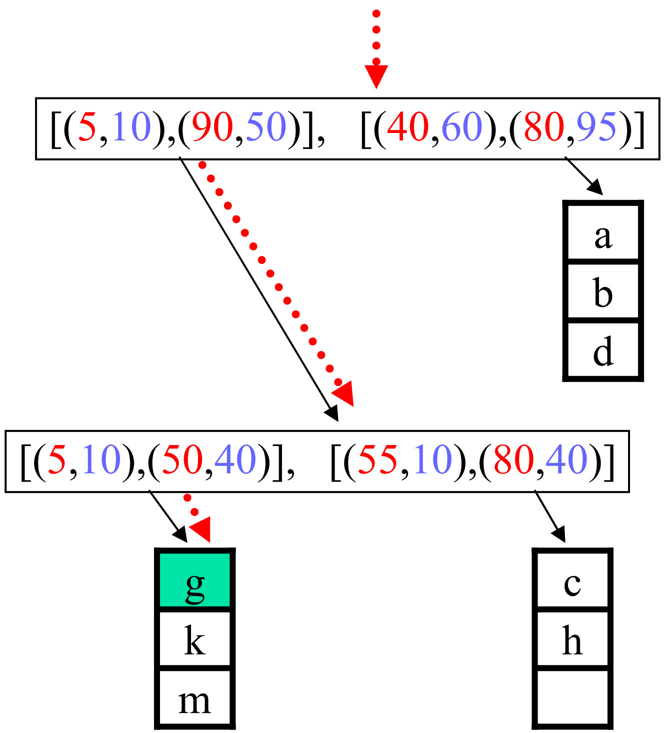
Find record a = (85, 90)

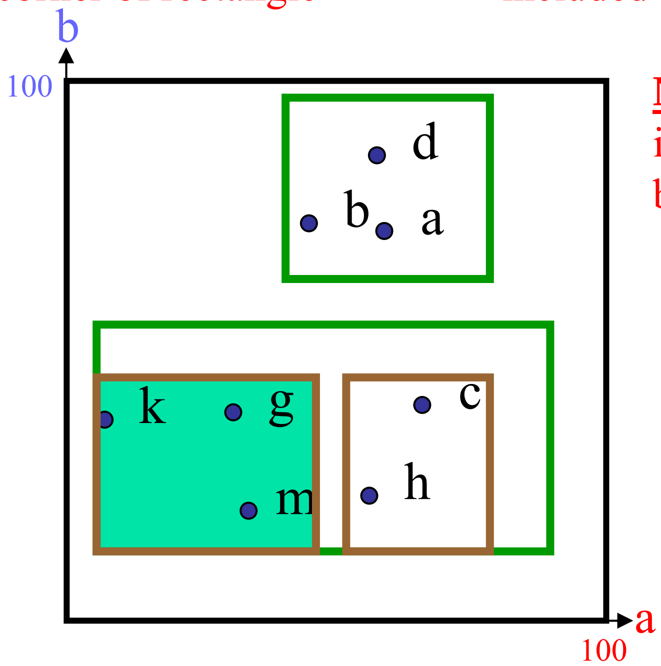
R - Trees

- An R-tree (region-tree) divide the k-dimensional region, into a varying number of sub-regions
- The regions may be of any shape (usually rectangles) and are allowed to overlap
- 2-dimensional example:

Note 1:
in this example the coordinates denote SW, NE corner of rectangle

Note 2:
all the area do not have to be covered as long as all elements are included into a region





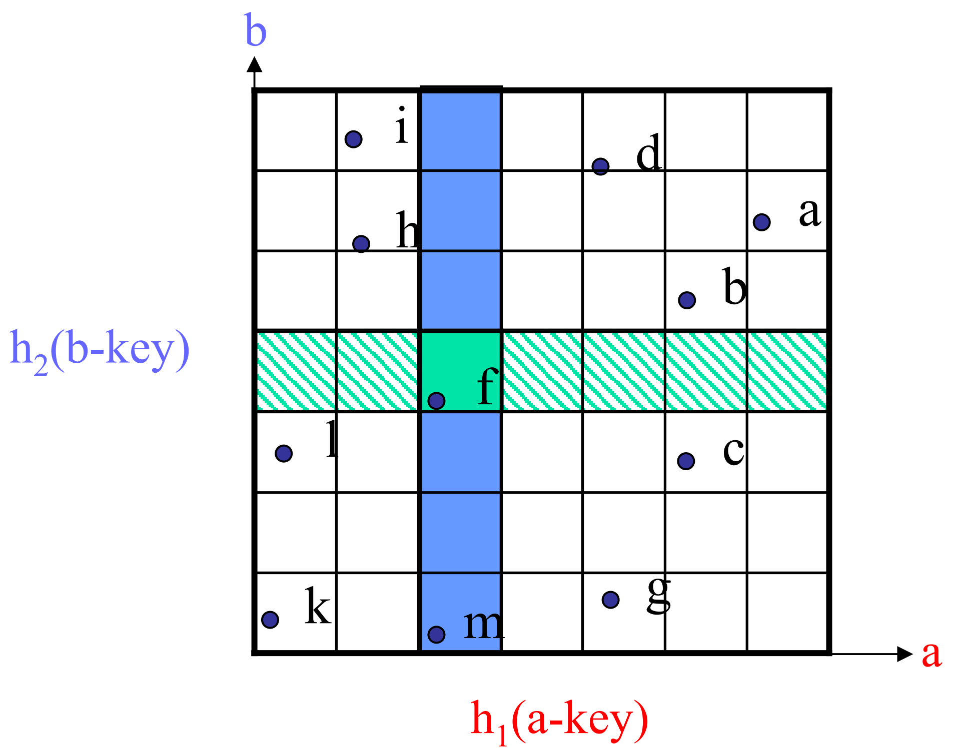
Note 3:
if not all elements fits in a bucket, divide further

Find record
g = (30, 35)

Hash-Like Structures: Grid Files – I

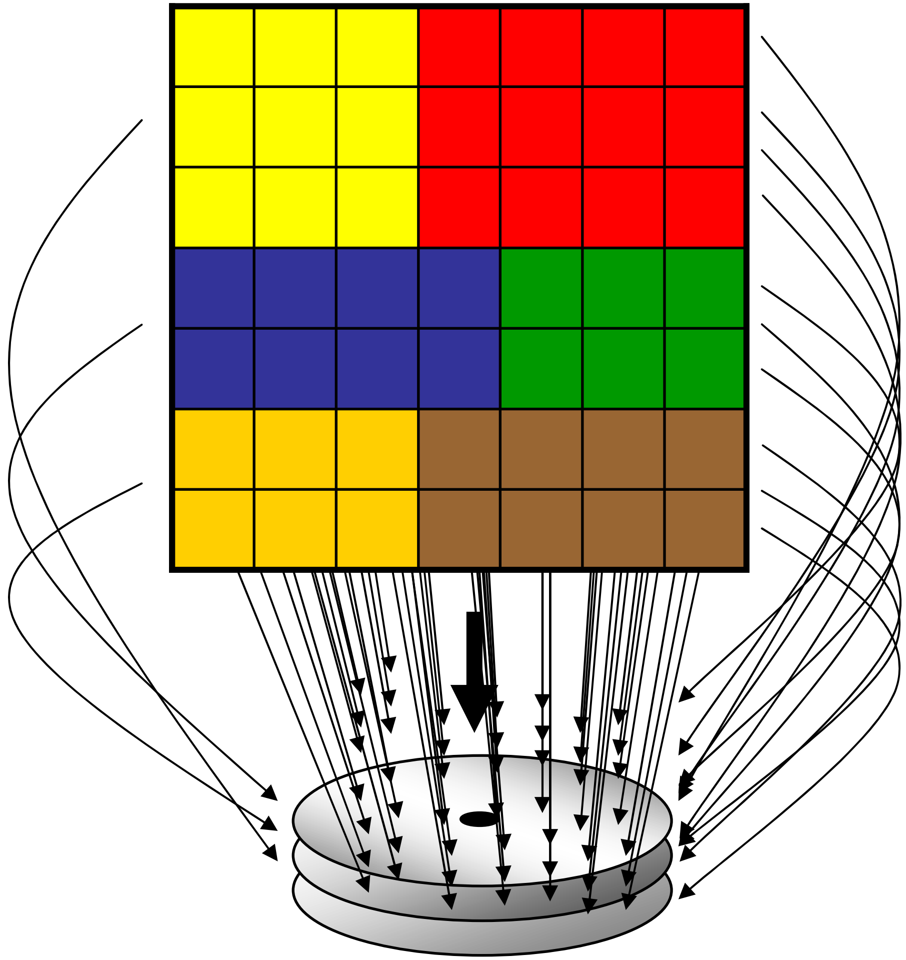
- Grid files extend traditional hashing indexes
  - hash values for each attribute in a multidimensional index
  - usually does not hash *values*, but *regions* – h(key) = <x,y>
  - grid lines partition the space into stripes

- 2-dimensional example:
  - find record (22, 31)
  - $h_1(22) = <a_x, a_y>$
  - $h_2(31) = <b_m, b_n>$
  - ⇨ record f



Hash-Like Structures: Grid Files – II

- ✓ How do we represent grid file on disk?
  - ➢ one bucket per sub-space
    → may waste lot of space
  - ➢ array in one direction
    → bad for range queries in "other" direction
  - ➢ areas
    → may chose appropriate areas
  - ➢ use *indirection*, as areas, but use buckets



Hash-Like Structures: Grid Files – III

- ✓ What can we do if a block does not have more room?
  - allow overflow blocks
  - add more grids
- ✓ Grid files can quickly find records with
  - key 1 = V_i AND Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j
  - key 1 $\geq V_i$ AND key 2 $< X_j$
- ✓ Grid files are
  - ☺ good for multiple-key search
  - ☹ space, management overhead
  - ☹ partitioning ranges evenly split keys

Hash-Like Structures: Partitioned – I

- ✓ Partitioned hash functions ...
  - ➢ ...allow several arguments
  - ➢ ...produce a single hash value
  - ⇨ $h(key_1, key_2, key_3, ...) = x$

  ☺ maps a multi-dimensional index to a single-dimension array

  ☹ only useful for searches involving *all* keys

- ✓ A more useful approach is to let the hash function produce a bit-string where each search key is represented by some of the bits

Hash-Like Structures: Partitioned – II

- Example – bit-string:
  - index on both department and salary of employees
  - h(...) produces a 3-bit string
    - $h_1(key_{department})$ produce the first bit
    - $h_2(key_{salary})$ produce the two last bits

  ⇨ $h(key_{department}, key_{salary}) = h_1(key_{department}) + h_2(key_{salary})$

$h_1()$

toys	0
clothes	0
admin	1
music	1
shoes	0
...	...

$h_2()$

100.000	01
200.000	00
300.000	10
400.000	10
500.000	11
...	...

Note:
several key values can map to the same bit-string

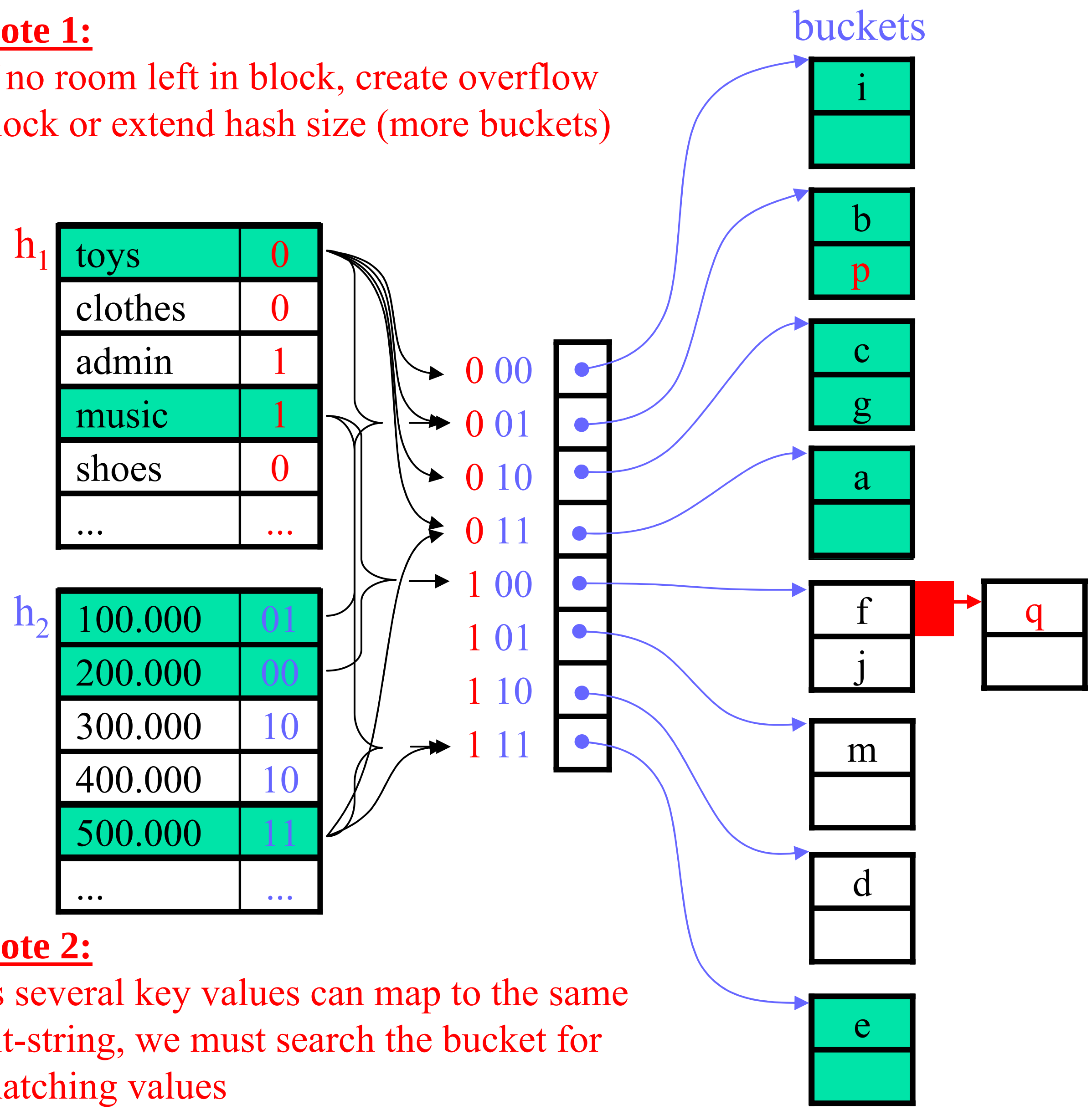
examples:

h(toys, 100000) = 0 01
h(toys, 500000) = 0 11
h(music, 300000) = 1 10
h(admin, 500000) = 1 11
h(shoes, 100000) = 0 01

Hash-Like Structures: Partitioned – III

- ✓ Example (cont.):
  - ➢ ***insert:***
    - ❖ p = (toys, 100000)
    - ❖ q = (music, 200000)
  - ➢ ***find employees with:***
    - ❖ department = music and salary = 500.000
    - ❖ department = toys
    - ❖ salary = 500.000

Note 1:
if no room left in block, create overflow block or extend hash size (more buckets)



Note 2:
as several key values can map to the same bit-string, we must search the bucket for matching values

Summary

- ✓ Searching is expensive
- ✓ Indexes
  - ☺ minimize the search costs and block accesses
  - ☹ increased storage requirements
  - ☹ more complex operations on inserts/updates/deletions of records
- ✓ Conventional indexes
- ✓ B-trees (B^+-trees)
- ✓ Hashing schemes
- ✓ Multidimensional indexes
  - ➢ tree-like structures
  - ➢ hash-like structures