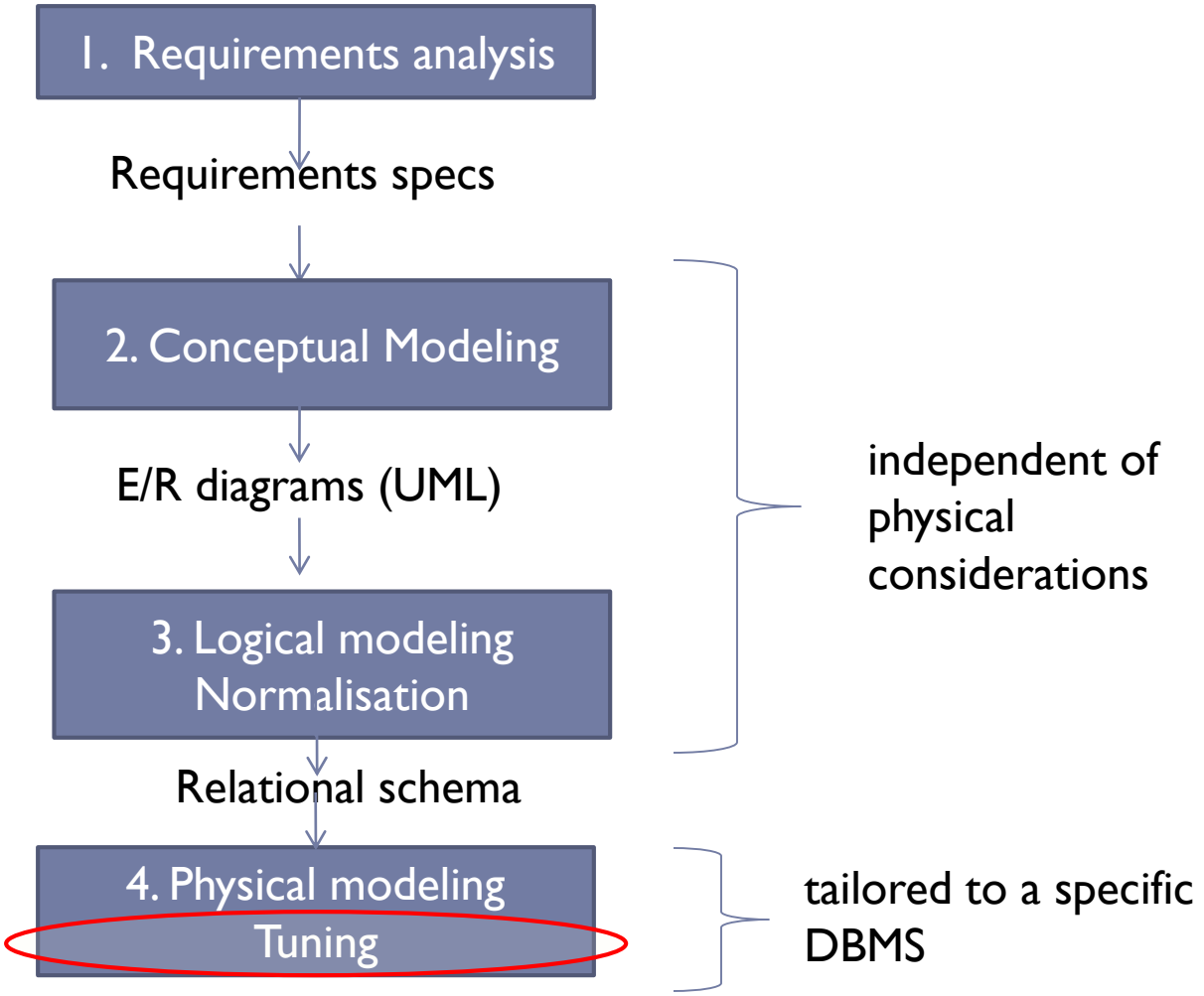
CREATE BITMAP INDEX *idx* ON

Databases (topic);

SELECT * FROM *Databases* WHERE *topic* = '**INDEXES**';

Mihaela Elena Breabăn

Relational Database Design Methodology

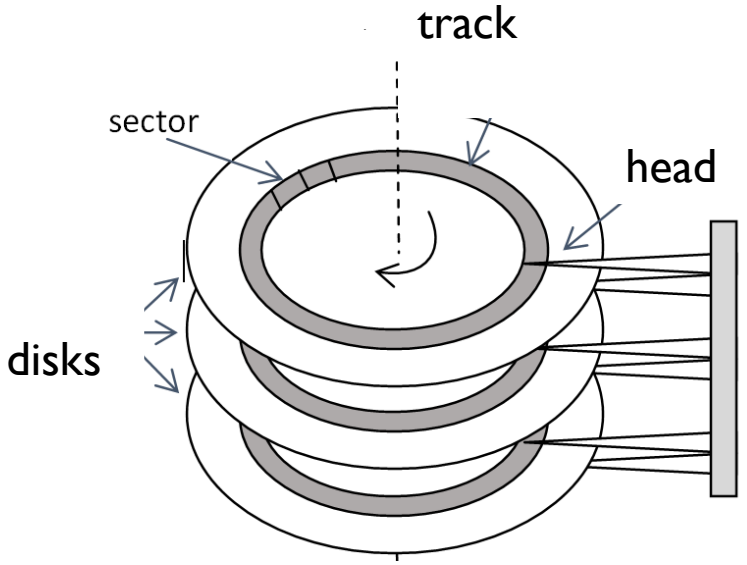


Outline

- Physical storage and access
- Indexing - motivation
- Ordered structures
  - Sequential indexes
  - B^+-Trees
  - Multi-key indexes
- Hashing
  - Static Hashing
  - Dynamic Hashing
- Multi-key access and Bitmap indexes
- Indexes in SQL
- Indexes in Oracle

Physical storage and access

sID	sName	grade
20	Ioana	9.5
40	Andrei	8.66
10	Tudor	8.55
30	Maria	8.33
70	Alex	9.33



- The time to bring a data block into memory is determined by:
  - *seek time* (time to move the drive head to the track)
  - *rotational latency* (time for the data/disk to rotate under the drive head)
  - *transfer time* (time to transfer the data to main memory)

Indexing – Motivation (1)

- Usually, the DBMS spends most of the time solving queries (searching)

SELECT * FROM Student

WHERE sID=40;

Search key

How can we retrieve the desired data?

a) Random ordering

sID	sName	grade
20	Ioana	9.5
40	Andrei	8.66
10	Tudor	8.55
30	Maria	8.33
70	Alex	9.33

Sort key

b) Ascending ordering of sID

sID	sName	grade
10	Tudor	8.55
20	Ioana	9.5
30	Maria	8.33
40	Andrei	8.66
70	Alex	9.33

Indexing – Motivation (2)

- Binary search
  - Time complexity: $O(\log_2(N))$
    ($\log_2(100\ 000)=17$)

SELECT * FROM Student

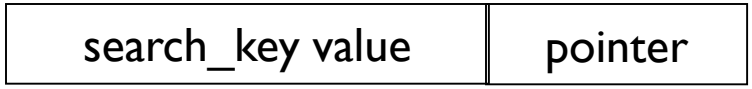
WHERE sName='Ioana';

How can we solve this problem efficiently?

- Solution: build an index file

Basic concepts in indexing

- **Data file** – the sequence of blocks containing the records in a table
- **Search key** – an attribute (or a set of attributes) used to search records in a data file
- **Sort key -** an attribute that decides the order of the records in a data file
- **Index file** – is associated to a search key for a data file and contains **index records** of the form



- **Dense index** – stores one entry for each existing value of the search key
- **Sparse index** – stores fewer entries

Observations;

- A data file can have associated more than 1 index file
- Index files are usually smaller than the data file

Questions…

- HOW many indexes in practice ?
- WHEN should we (NOT) create indexes ?
- HOW should we index (in terms of data structures)?

- Considerations:
  - Necessary storing space
  - Access time
  - Insertion time
  - Deletion time
  - The types of supported queries

Index types

- Ordered indexes: the values of the search key are ordered
  - Sequential indexes
  - B^+-trees
- Hash indexes: the values of the search key are uniformly distributed in buckets based on a hash function
  - Bucket: a storage unit containing one or more records
- Bitmap indexes: associated to discrete attributes, encode its distribution as a binary matrix

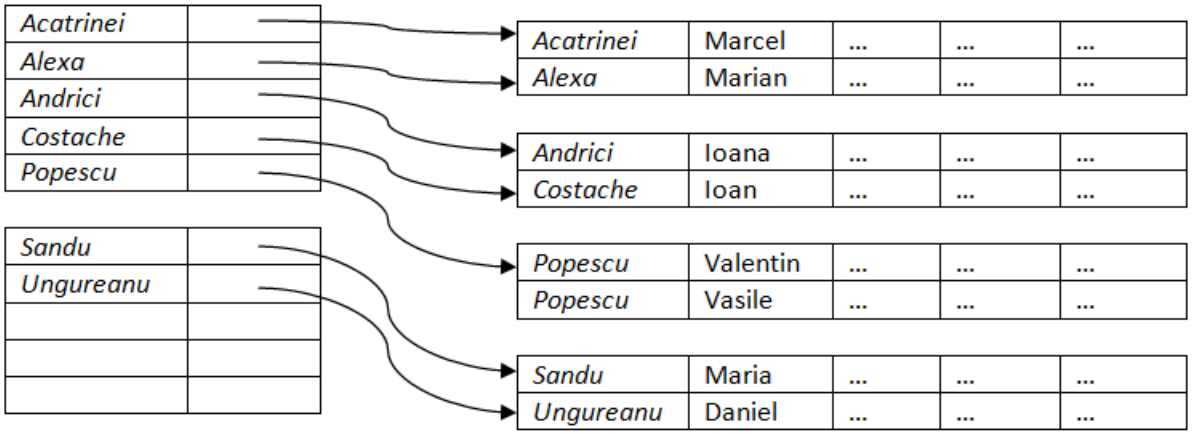
Ordered indexes: sequential files

Sequential indexes

- Index entries are sorted based on the search key
  - Ex: the catalogue with authors in a library
- Primary index: the search key is also a sort key for the data file
  - called also clustering/clustered index
  - The search key is usually the primary key of the table – but not mandatory
  - A table may have at most one primary index
- Secondary index (nonclustering/nonclustered): the search key gives an ordering different than that of the data file

Dense indexes

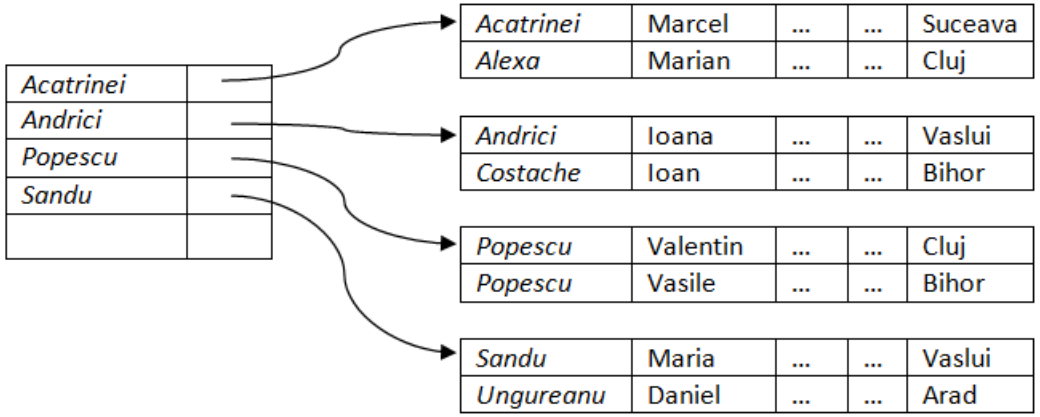
- Dense index: there exist entries for each value of the search key existent in the data file
- If the index is primary, it maintains a single pointer for the entries having identical values in the data file (a pointer to the first entry in the series)
- If the index is secondary, several entries may be necessary for a value of the search key.



Primary dense index: the search key is the same as the sort key of the data file

Sparse index

- Sparse indexes do NOT contain entries for all the values of the search key
  - Applicable only for records which are sequentially ordered based on the search key (when the search key is the same as the sort key)
  - Time-space trade-off
  - Usually an entry in the index points to a block in a data file
- To locate a record having value k for the search key:
  - Find the index entry having the largest value of the search smaller than k
  - Look sequentially in the data file starting with the record indicated by the index entry



Sparse index: the index search key must be identical with the data file sort key

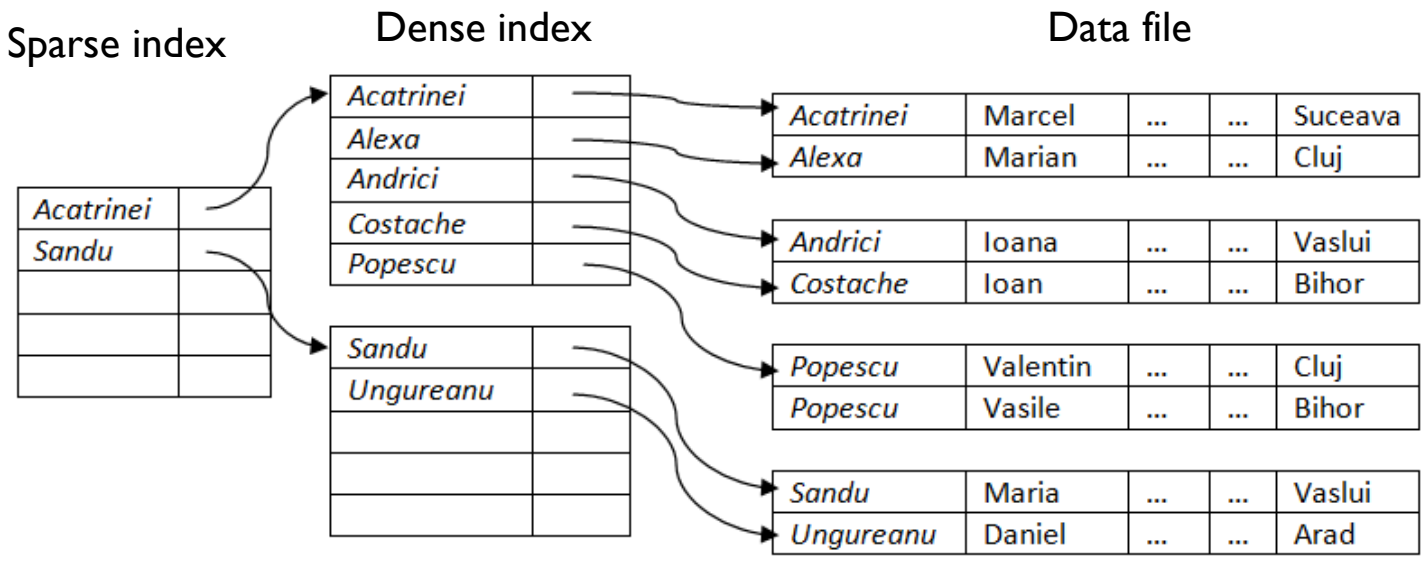
Multi-level indexes

- Multi-level index: an index associated to another index
  - Necessary when the primary index does not fit into the main memory
  - As solution: the primary index is stored on the disk as a data file and a sparse index is attached to it
- External index – a sparse index over the primary index
- Internal index – the primary index

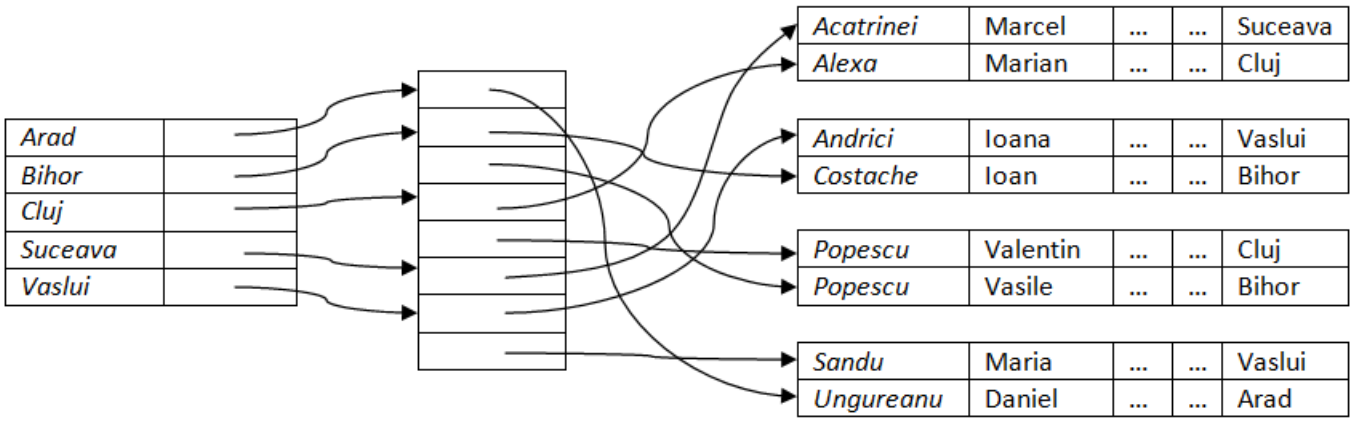
- When the external index is still too large to fit into the memory, another sparser index may be built over it, etc…

- Indexes on all the levels must be updated when the data file is modified by DML operations

Multi-level indexes Example



Secondary index



- *Query: Find the students living in Cluj (the data file is sorted based on students' names)*
- Solution: build a secondary index (dense!)
- To implement efficiently the one-to-many relation between the index and the data file, buckets of pointers may be used

Updating sequential indexes
Deletions

- Start by retrieving the record to be deleted using the index
- Deleting a record in the data file requires updating the index:
  - If the deleted record is the only one corresponding to a given search key, this is also deleted from the index

- Delete search key value *k* from
  - Dense indexes: is similar to deleting from the data file
  - Sparse indexes:
    - If there exists an entry *k* in the index, this is replaced by the next value of the search key (with respect to the ordering of the search key)
    - If the next value of the search key already exists in the index, *k* is simply deleted.

Updating sequential indexes Insertions

- Search in the index for the value of the search key corresponding to the inserted tuple
- Dense indexes: if the value does not appear in the index, it will be inserted
- Sparse indexes: if the index maintains an entry for each block in the data file, only when a new block is created in the data file, a new entry will be added in the index, pointing to the first record in the block)

- Insertions in the data file and in the index file may necessitate the creation of excess blocks -> the sequential structure may degenerate

- Insertion and deletion in multi-level indices are simple extensions of the discussed cases

Ordered indexes: B^+-trees

B^+-tree based indexes
Motivation

- Sequential structures degrade after many DML operations
- Reconstructing the indexes is necessary but costly
- B^+-tree
  - Speeds up retrieval and eliminates the constant need for reorganization
  - Is used extensively for data indexing in relational DBMSs

The structure of a B$^+$-tree (1)

- A balanced tree such that all the leaves are on the same level
- Structure of a typical node:

P_1	K_1	P_2	K_2	...	P_m	K_m	P_{m+1}

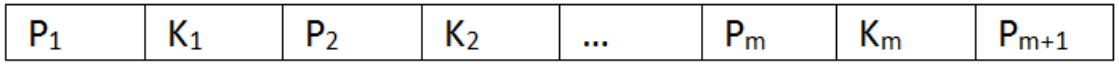
  - K_i – values of the search key
  - P_i pointers to
    - Internal descending nodes
    - (If in a leaf node) to single records in the data file or buckets of records
- The tree is characterized by a constant *m* specifying the maximum number of values in a node (the maximum number of pointers or descending nodes is *m+1*)
  - Usually m is computed such that the dimension of a node is equal to that of a block
- The values of the search key are ordered within a node

$$K_1 < K_2 < K_3 < \ldots < K_m$$

The structure of a B^+-tree (2)

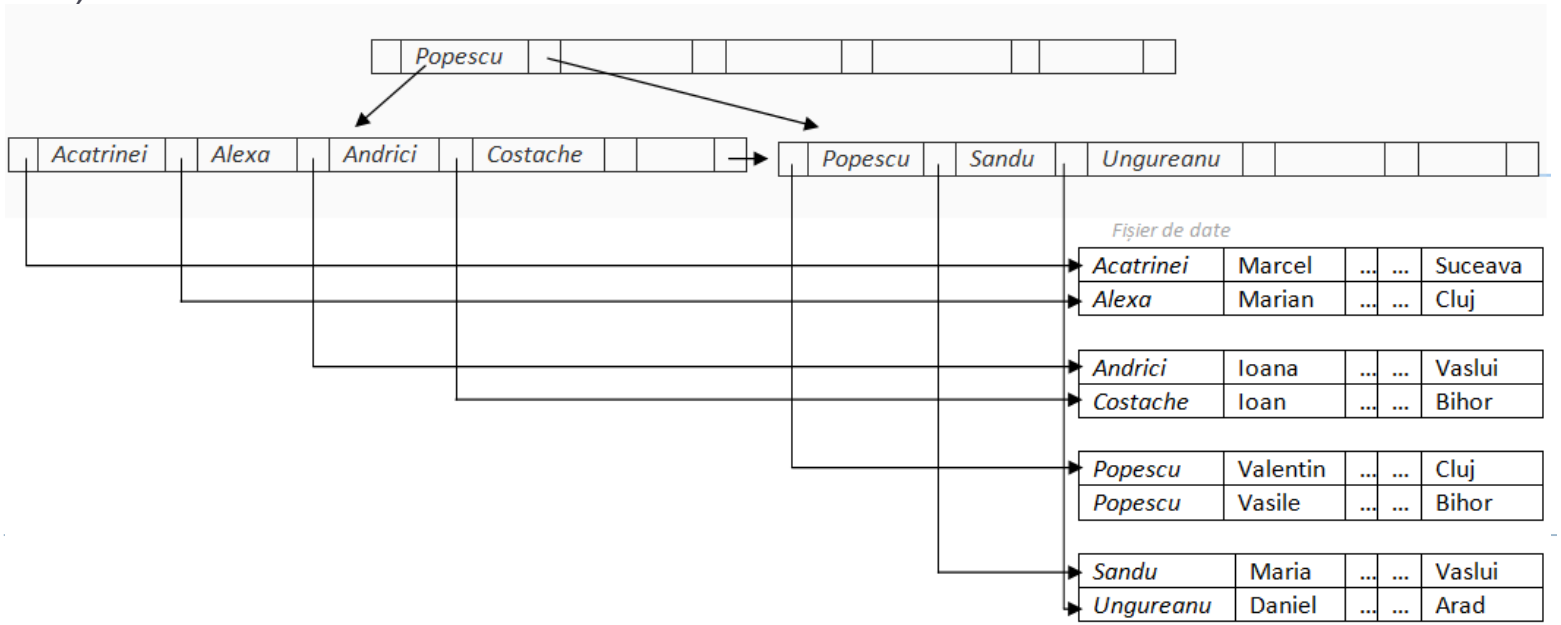
- Internal levels of the tree form a multi-level (sparse) index for the leaf nodes

- For a node with n pointers:
  - All the values of the search key in the subtree pointed by P_1 are smaller than K_1
  - For P_i, $2 \leq i \leq n-1$, all values of the search key in the subtree to which it points are larger than or equal to K_{i-1} and smaller than K_i
  - All values of the search key in the subtree pointed by P_n are larger than or equal to K_{n-1}



Rules for filling in the nodes

- Nodes are not fully filled in:
  - The root node has at least 2 and at most $m + 1$ *pointers/descendents (*correspondingly, at least 1 and at most m values, ordered ascending*)*
  - Every node on an internal level has at least $\lceil (m+1)/2 \rceil$ *and at most* $m + 1$ pointers/descendents (correspondingly, at least $\lceil m/2 \rceil$ and at most m values, ordered ascending)
  - Every leaf node has at least $\lceil m/2 \rceil$ *and at most* m values; all the pointers point to the data file, excepting the last pointer which points to the next leaf node (with larger values).



B^+-tree Example

- 1 memory block = 1024 bytes
- Search key = a string of maximum 20 characters (1 character=byte)
- 1pointer = 8 bytes

- The maximum number of entries in a node?
  - The largest value m satisfying $20m + 8(m + 1) <= 1024$.
  - $m=36$
- Root: at least one, at most 36 values
- Internal node: at least19, at most 37 pointers
- Leaf node: at least18, at most 36 values = pointers to the data file

B^+-trees
Observations

- Because nodes are connected by pointers, blocks logically close are not necessarily physically close
- All the levels excepting the leaf level form a hierarchy of sparse indexes
- The last level = a sequential dense index
- The B^+-tree contains a relatively small number of levels
  - At most $\lceil \log_{\lceil (m+1)/2 \rceil}(K) \rceil$ for K values of the search key
    - Level two: at least 2 nodes
    - Level 3: $2*\lceil (m+1)/2 \rceil$ nodes
    - Level 4: at least $2*\lceil (m+1)/2 \rceil * \lceil (m+1)/2 \rceil$
    - etc…
- Insertions and deletions are efficiently processed: restructuring the index requires logarithmic time

Queries on B^+- trees Algorithm

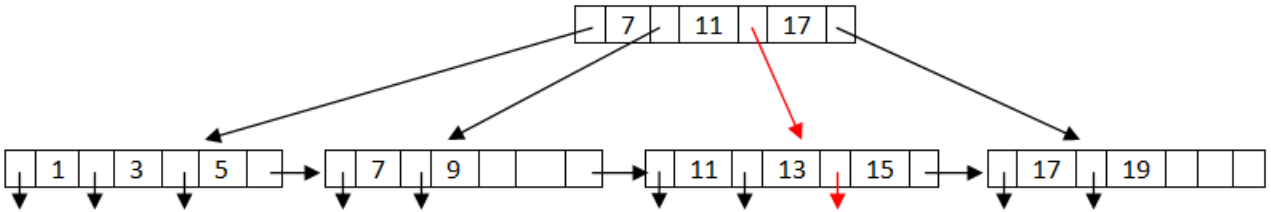
Goal: determine all the records in the data file having value k for the search key

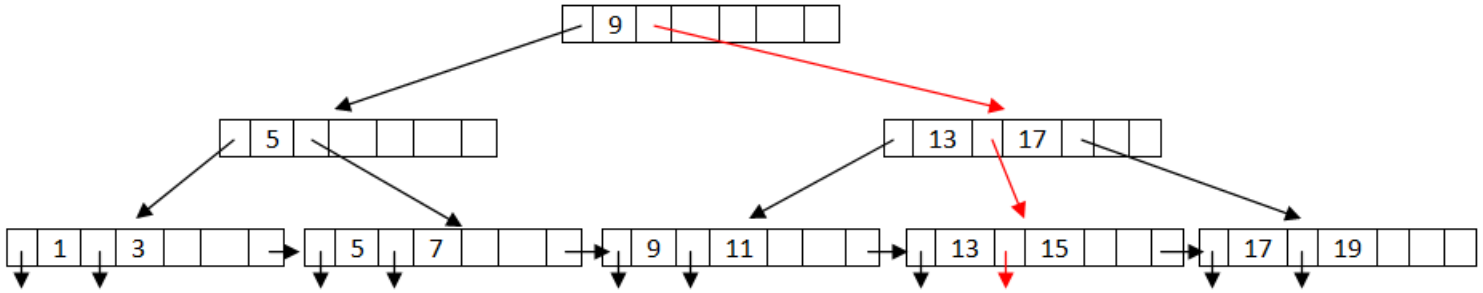
1. N=root
2. Repeat
   - Look in N for the smallest value of the search key which is largest than k
   - If such a value K_i exists, than $N = P_i$
     - else $N = P_n$ $(k \geq K_{n-1})$

   Until N is a leaf
3. If there exists $K_i = k$ in the leaf, pointer Pi points to the required record
4. Else, there is no record with search key = k

Queries

The search key – stores all the odd numbers between 1-19
Draw possible tree structures for m=3 and perform the query for value 15





Queries on B^+-trees
Quiz

- Given m=100 (each node stored on a block)
- *For 1 million values of the search key, how many nodes (blocks on the disk)are accessed when searching for a specific value in a B^+-tree? (4)*
- *What if a sequential index is used? (20)*

Updates in a B^+-tree Insertion

1. Find the leaf node that should contain the value to be inserted
2. If the value exists in the leaf node:
    1. Add the record in the data file
    2. Add a pointer in the bucket for that index entry
3. If the value does not exist
    1. Add the record in the data file
    2. If there is room in the leaf node in the index, insert the pair (pointer,key_value)
    3. Else divide the leaf node

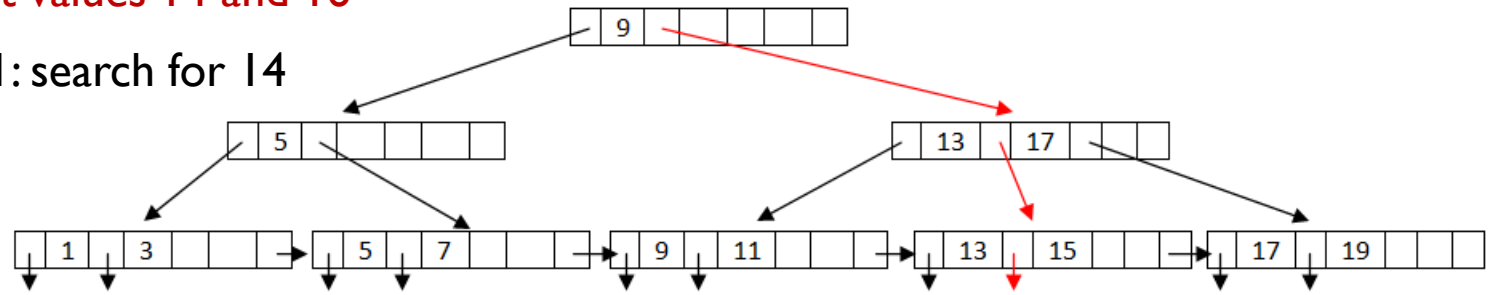
Updates in a B$^+$-tree
Insertion: node division

- Division of the leaf node
  1. Take the *n* ordered pairs (including the one to be inserted). Keep the first $\lceil n/2 \rceil$ pairs in the original leaf node and create a new leaf node containing the rest of pairs
  2. Let *P* be the new node and *k* the smallest value of the search key in *P*. Insert pair (*k*,*p*) in the parent node of the dividing leaf – where p is the pointer to P
  3. If the parent is full, this must be divided, propagating upward the division until a non-fully occupied node is found. In the worst case, the root node is divided, case which increases the depth of the tree.
- Dividing an internal node N when inserting a pair (k,p)
  1. Create a temporary node M to store n+1 pointers and n values; insert the pairs in N and the pair (k,p)
  2. Copy $P_1,K_1, \ldots, K_{\lceil n/2 \rceil -1},P_{\lceil n/2 \rceil}$ from M back in N
  3. Copy $P_{\lceil n/2 \rceil +1},K_{\lceil n/2 \rceil +1},\ldots,K_n,P_{n+1}$ from M in a new node N'
  4. Insert $(K_{\lceil n/2 \rceil},N')$ in the parent of N

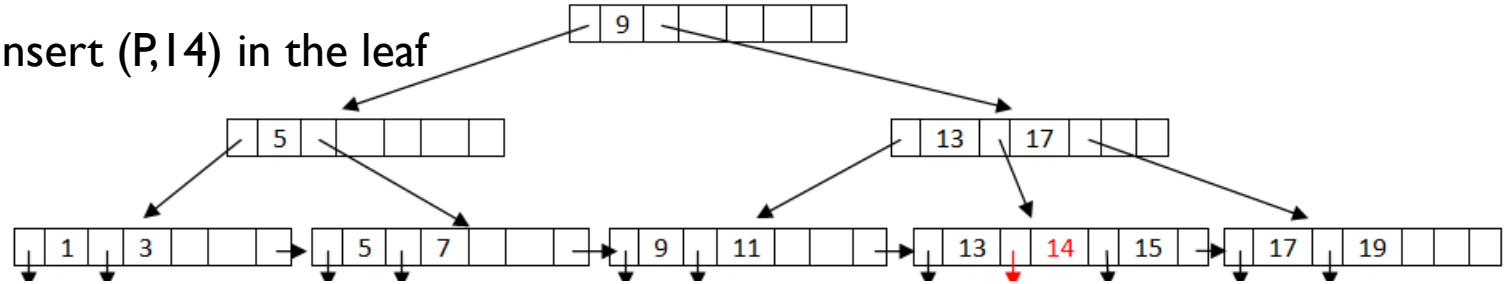
Updates in a B$^+$-tree
Insertion: Example (1)

Insert values 14 and 16

Step1: search for 14

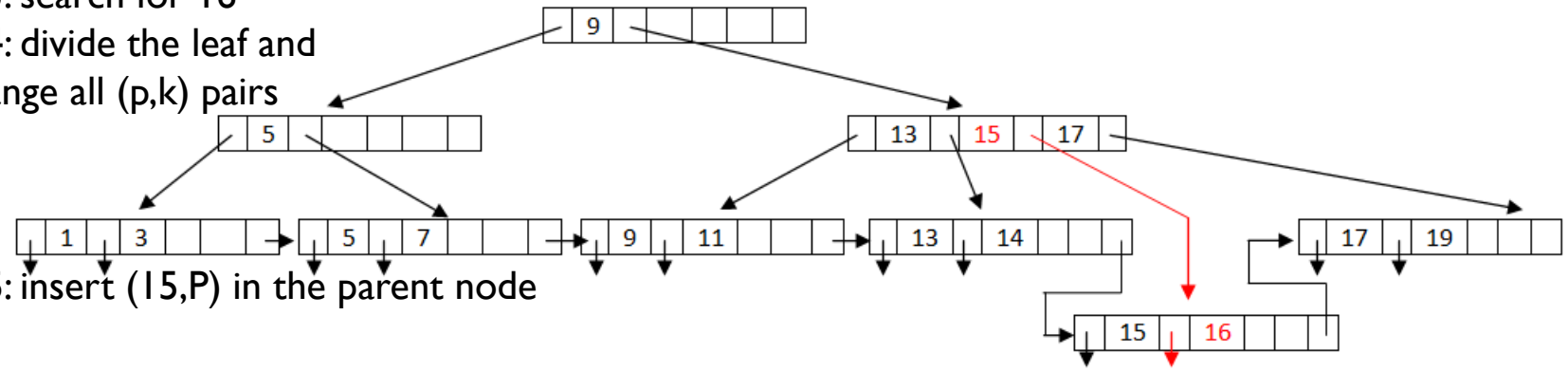


Step2: insert (P,14) in the leaf



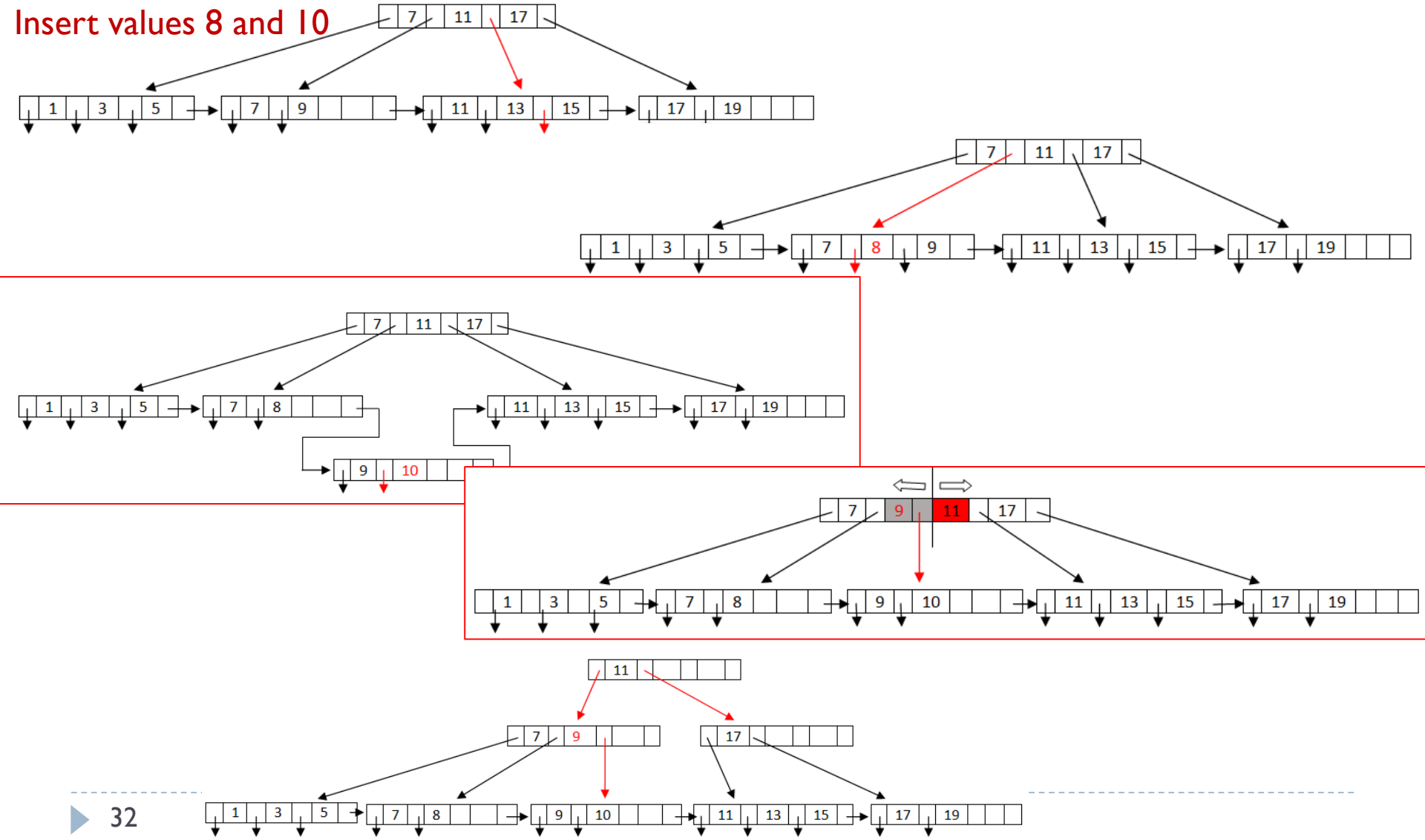
Step 3: search for 16

Step 4: divide the leaf and rearrange all (p,k) pairs



Step 5: insert (15,P) in the parent node

Updates in a B$^+$-tree
Insertion: Example (2)

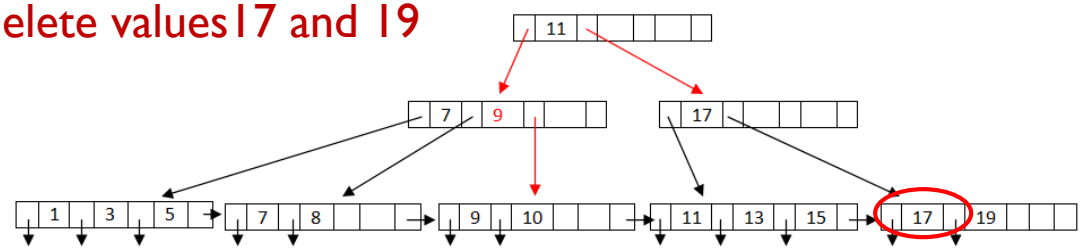


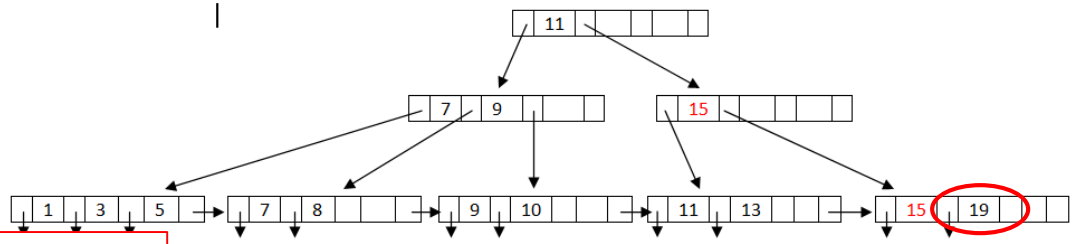
Updates in a B$^+$-tree Deletions

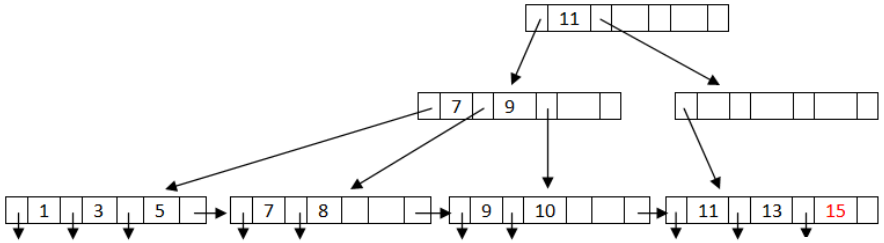
1. Delete the record from the data file
2. If the corresponding pointer is part of a bucket, the entry in the bucket is deleted. Otherwise (also if the bucket becomes empty) delete from the leaf node the pair (pointer, key_value)
3. If the leaf node remains with too few entries, and if there is room for these in a neighboring leaf, one leaf node is deleted:
    1. Insert all the entries in the left leaf node and delete the right one
    2. Delete pair (K_{i-1}, P_i), where P_i is the pointer to the deleted leaf node in the parent. If necessary, propagate upword the deletion. If the rrot node has only one pointer, it will be deleted.
4. Otherwise, if there is not enough room in neighboring leaves, the (pointer, key_value) pairs are redistributed among tow leaves:
    1. Such that the minimum is still satisfied
    2. Update of the corresponding (key_value, pointer) pair in the parent node may be necessary

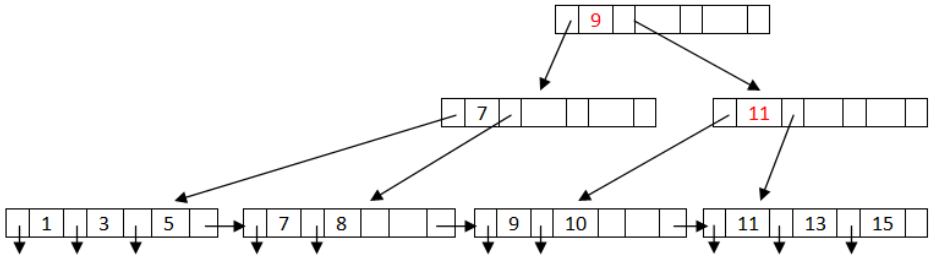
Updates in a B$^+$-tree
Deletions: Example (1)

Delete values17 and 19



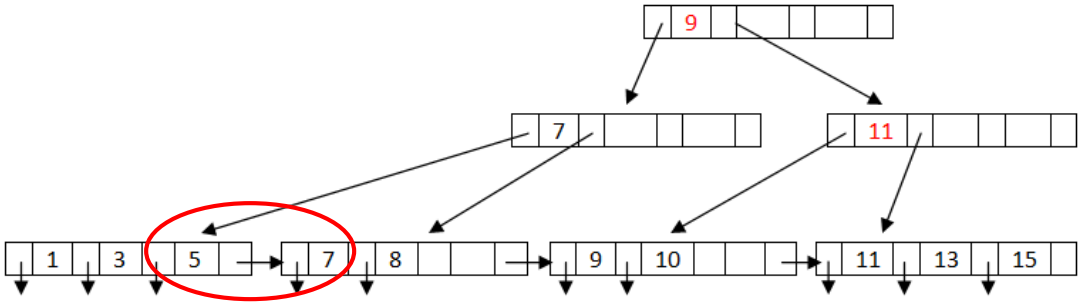


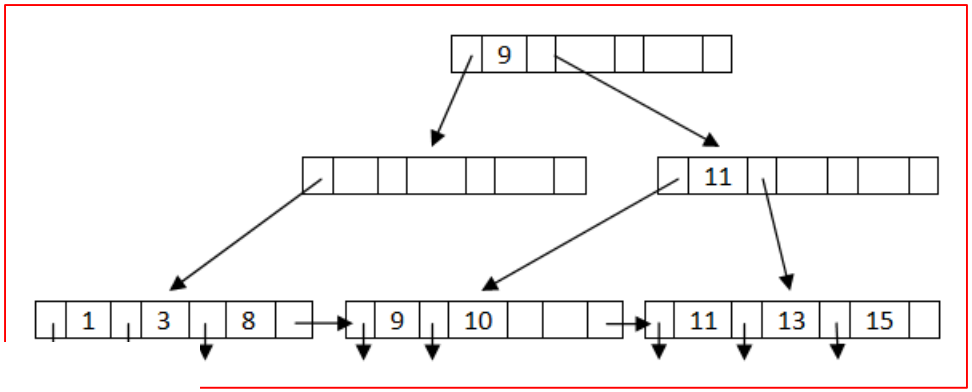


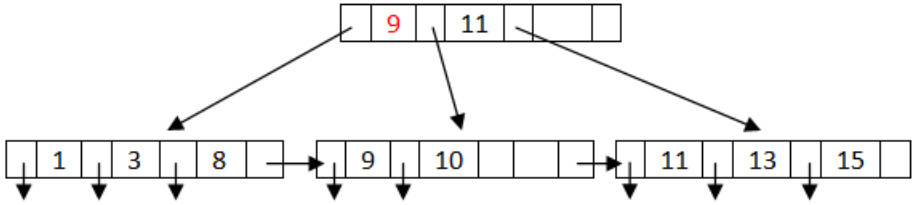


Updates in a B$^+$-tree
Deletions: Example (2)

Delete values 5 and 7







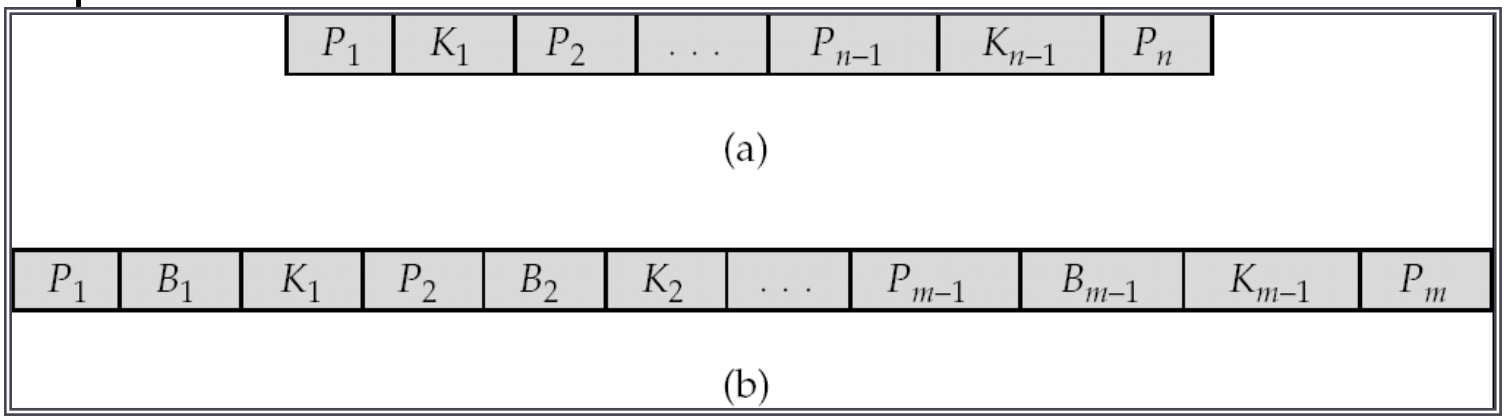
B^+-trees
Efficiency

- Search: at most $\lceil \log_{\lceil (m+1)/2 \rceil}(K) \rceil$ blocks transferred
  - Because the leaf nodes are connected, interval queries are also efficiently solved

- Insertion, deletion: at most $2 \lceil \log_{\lceil (m+1)/2 \rceil}(K) \rceil$ transferred blocks

B-trees

- Similar to B^+-trees but allow only one occurrence of each key value
- The key values do not necessarily appear on the leaf level, necessitating one extra pointer

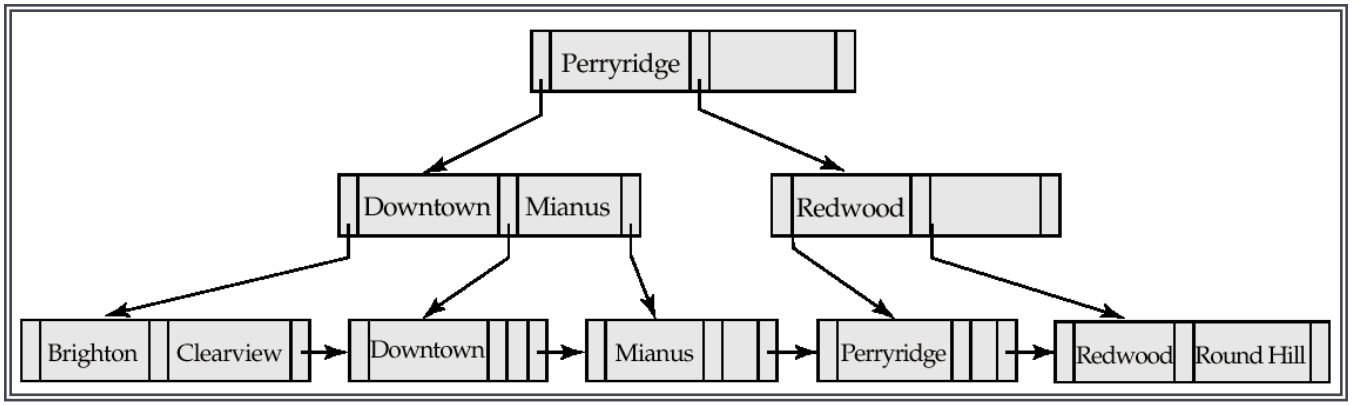


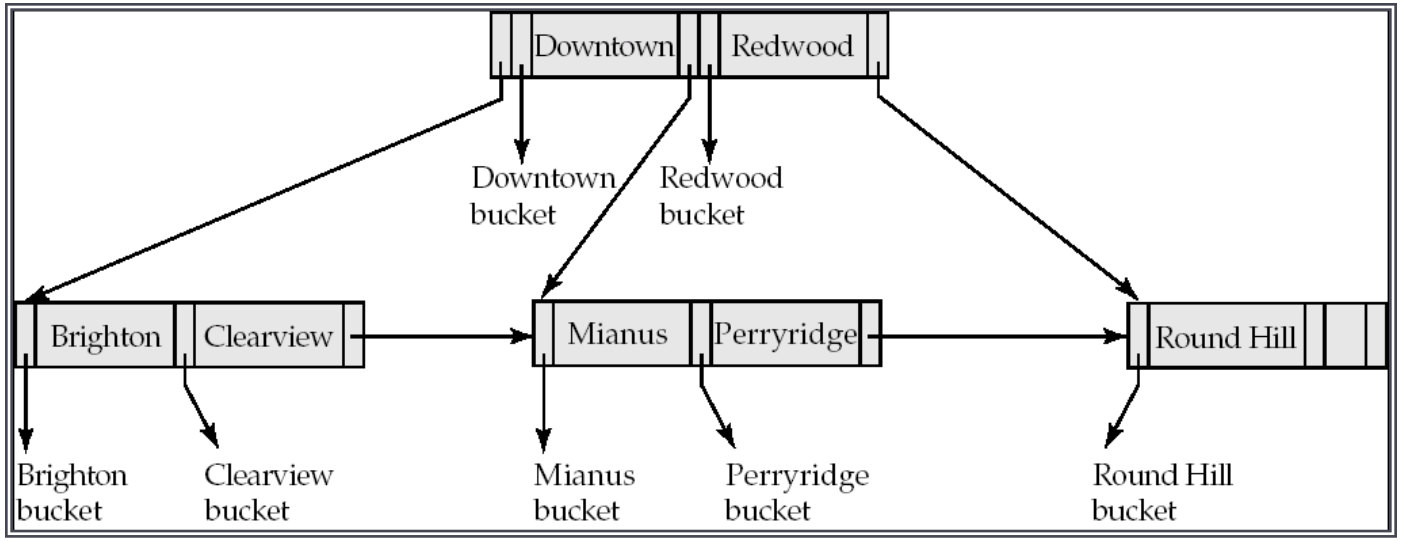
(a) B^+-tree node

(b) B-tree node

- Pointers B_i lead to records in the data file or buckets of pointers to the records

B-tree index Example*





* *Avi Silberschatz Henry F. Korth S. Sudarshan. "Database System Concepts".* McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)

B-tree indexes
Observations

- Advantages
  - May require less nodes than the corresponding B^+-tree
  - It is possible to locate the sought record before reaching the leaf level
- Disadvantages
  - Internal nodes are larger, resulting in trees with higher depth
  - Insertions and deletions are more complicated -> implementation is more difficult
  - It is not possible to scan a table based on the leaves

- Advantages do not weight more than advantages: B^+-trees are preferred over B-trees by DBMSs

Ordered indexes: Multi-key indexes

Multi-key access

```
SELECT *
FROM students
WHERE county= 'Bihor' AND year > 2010;
```

- Several search strategies involving single-attribute indexes are possible:
  - Using the index associated to the *county* search key
  - Using the index associated to the *year* search key
  - Using both indexes above followed by intersection
    - What if only one of the conditions is satisfied by many records?

Multi-key indexes

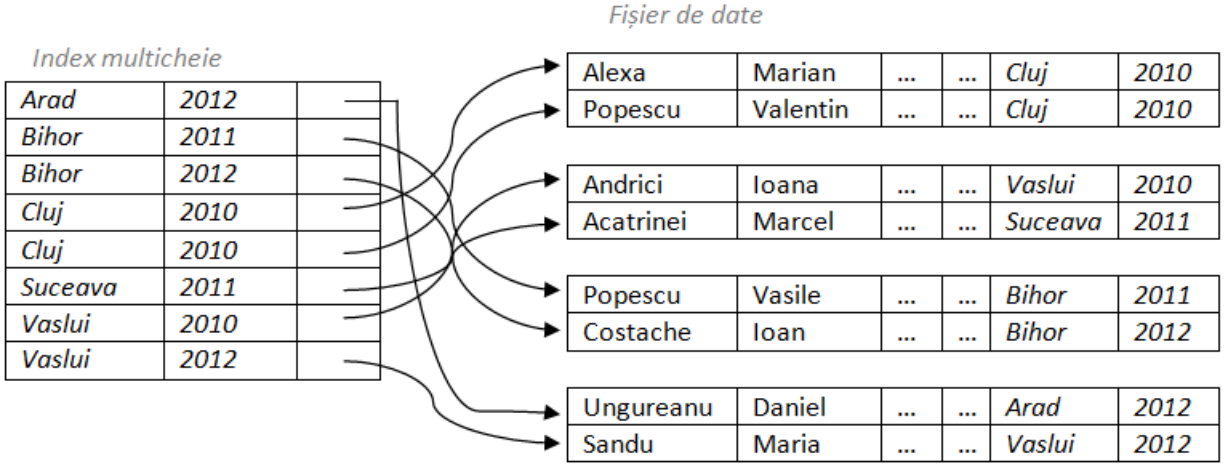
- The search key is composed by more than one attribute
- Lexicographical order is used: $(a_1, a_2) < (b_1, b_2)$ if
  - $a_1 < b_1$ or
  - $a_1 = b_1$ and $a_2 < b_2$

Ex. Consider the multi-key index *(county, year)*

Is it possible to efficiently solve both queries below?

a) where county= 'Bihor' AND year > 2010

b) where county> 'Bihor' AND year = 2010



Efficiency

- The order of the attributes in a multi-key index DOES matter!
- The efficiency depends on the selectivity of the attributes

k-d trees

- Generalization of the binary search tree:
  - k= number of attributes of the search key
  - Every level corresponds to one of the k attributes
  - The sequence of levels represents iterations over the attribute set
  - To obtain balanced trees, usually the median is used at the nodes

Hash file organization
Hash indexes

Hashing

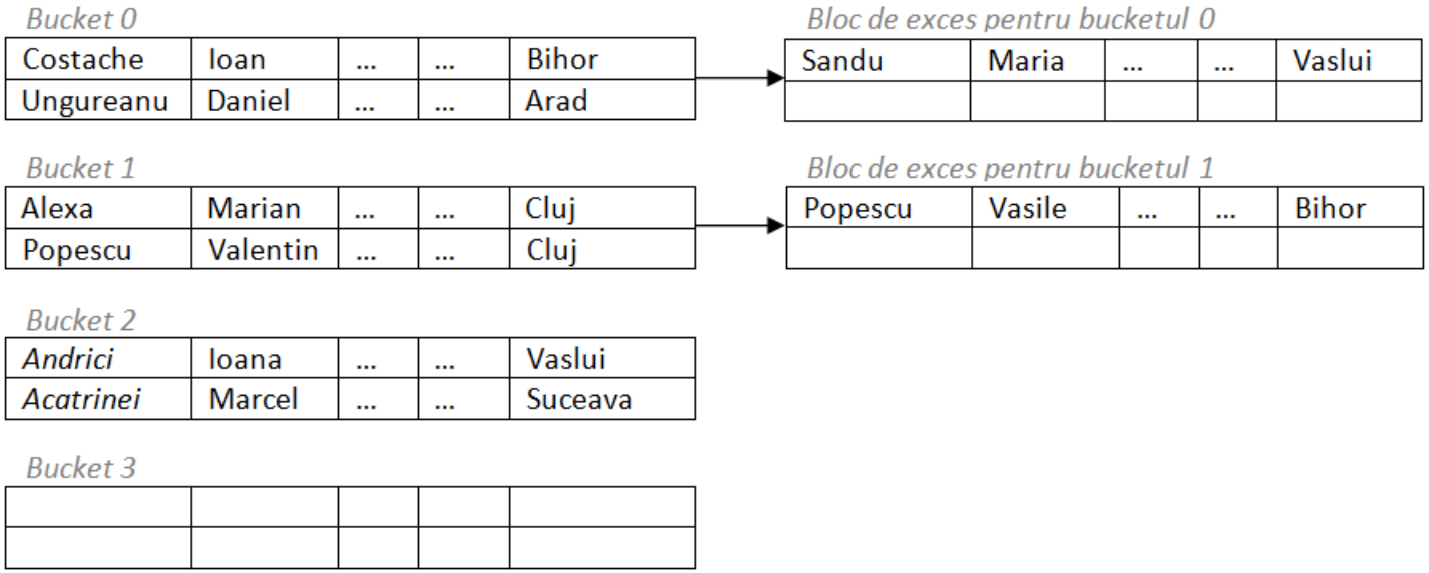
- In a hash file organization of the data file, the records are grouped in buckets that are retrieved based on the values of the search key using a hash function (the size of a bucket usually corresponds to a memory block)
- A hash function *h:K->B* is a function from the set of values of the search key to the set of addresses of the buckets

- Records with different values of the search key may be mapped to the same bucket
  - The search involves looking sequentially in the bucket

Hash functions

- Requirements
  - Uniform distribution
  - Random assignment
- Typical hash functions use computations on the binary (internal) representation of the search key
- There may appear situations when the size of the bucket is too small to store all the mapped records -> excess buckets are used

Hash file organization
Example



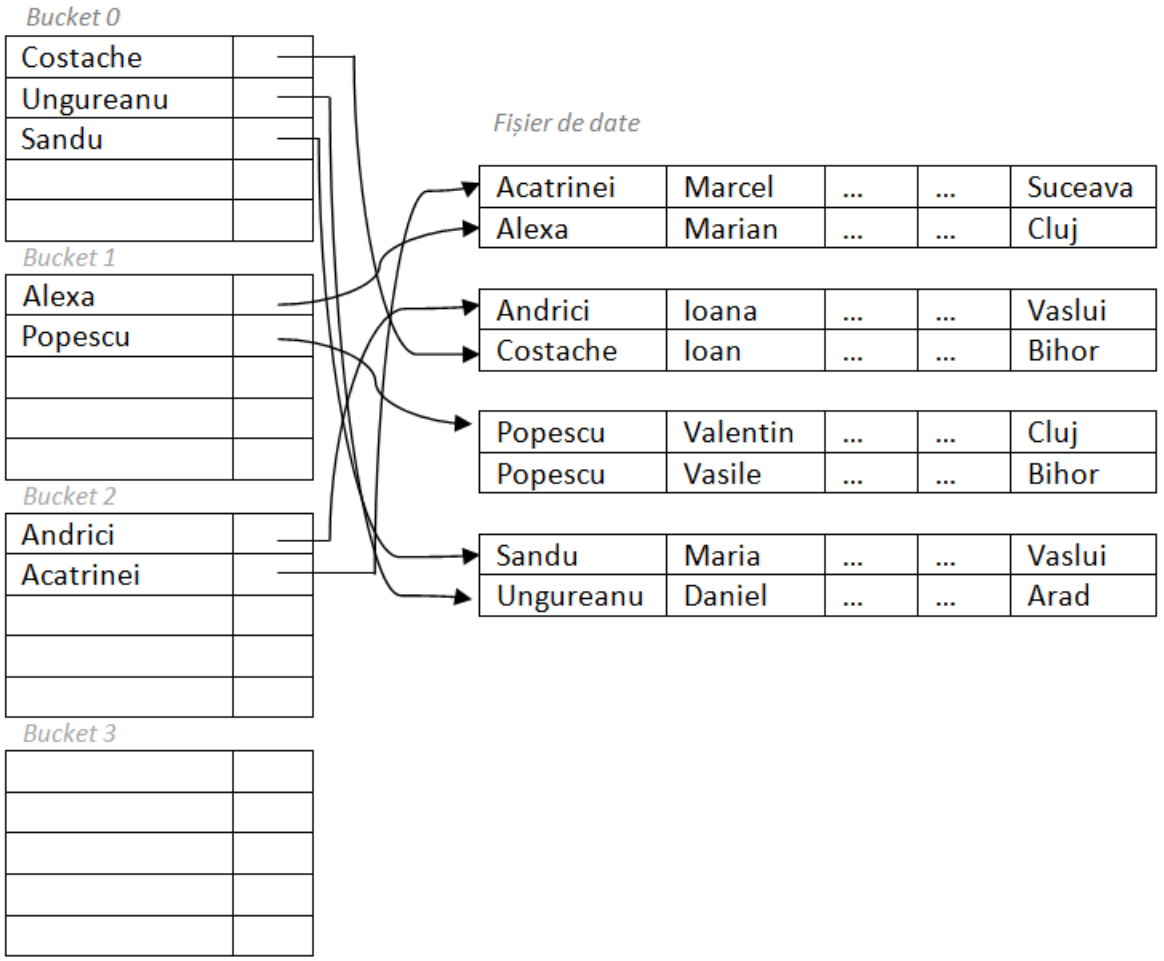
Use *last_name* as key:

Hash function: h:Dom(last_name)->{*0,1,2,3*} – *computes the sum on the binary representation modulo* 4

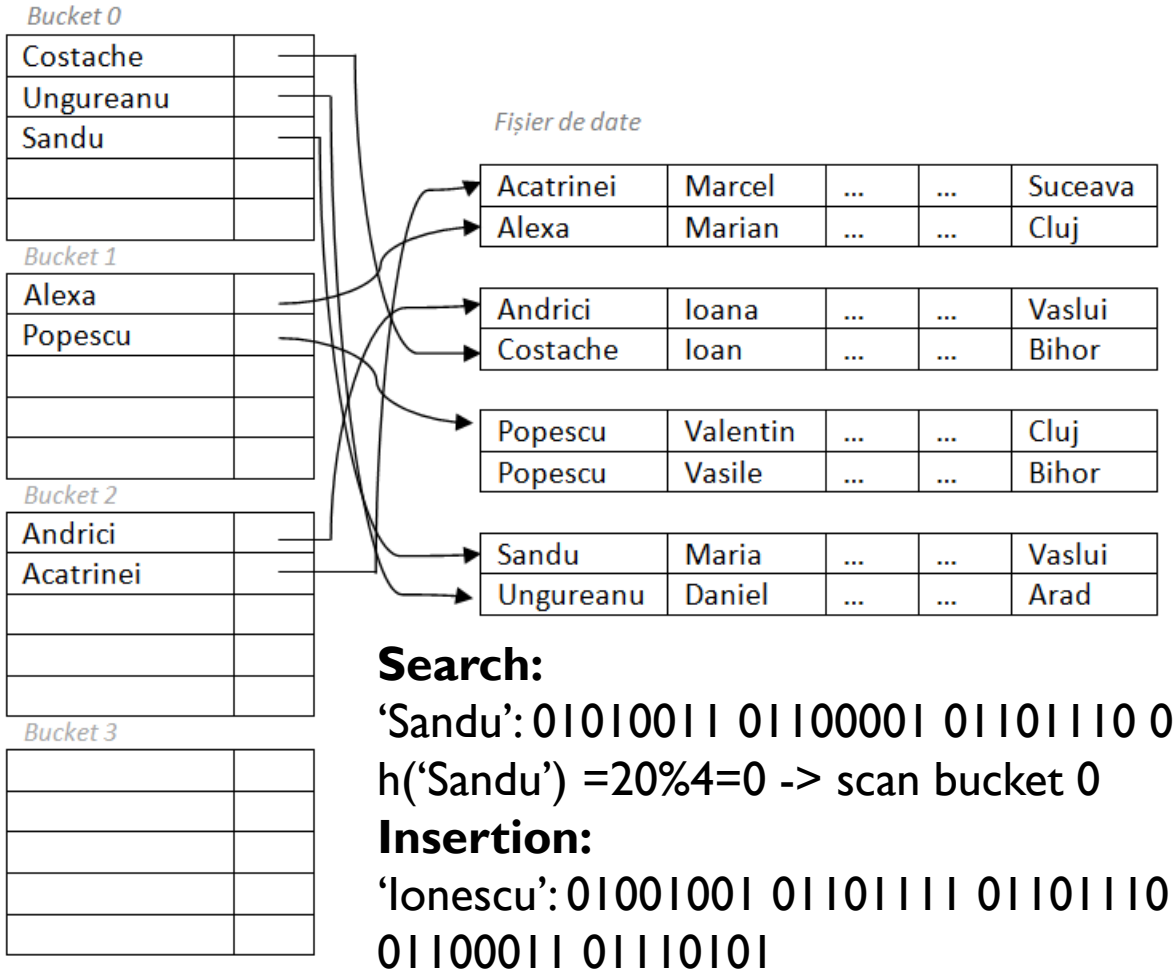
'Acatrinei':
'1000001 1100011 1100001 1110100 1110010 1101001 1101110 1100101 1101001'
h('Acatrinei') = 34%4 = 2.

Hash indexes

- Organizes the search keys with the associated pointers into a hash structure



Hash indexes
Operations



Search:

'Sandu': 01010011 01100001 01101110 01100100 01110101

h('Sandu') =20%4=0 -> scan bucket 0

Insertion:

'Ionescu': 01001001 01101111 01101110 01100101 01110011 01100011 01110101

H(Ionescu)=32%4=0 -> insert first the record in the data file and than the index entry in bucket 0

Deletion: compute the hash, scan the bucket, delete

Hash indexes
Efficiency

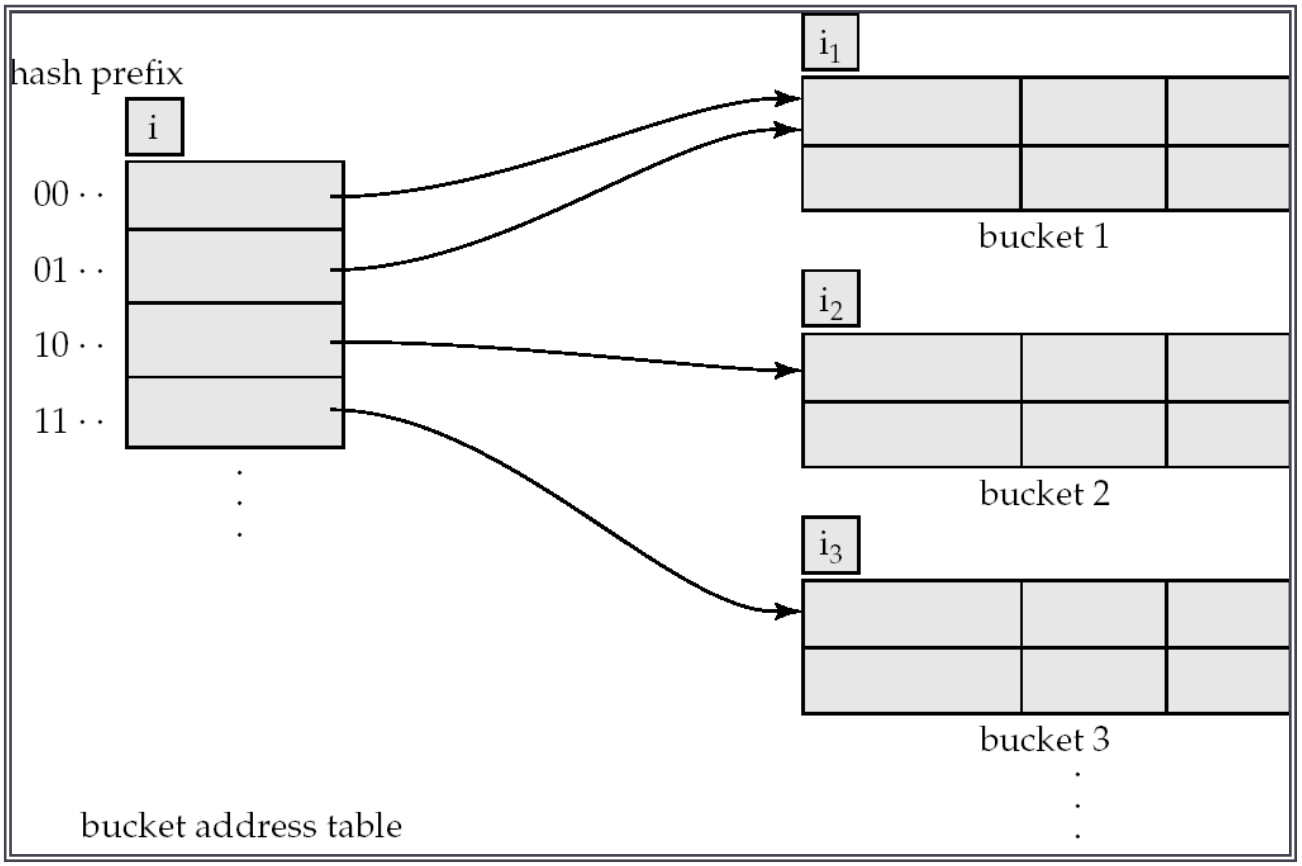
- When searching for a single value, in the absence of collisions, retrieving a value requires a single block read
- For range search, the hash indexes are not efficient. WHY?

- In practice:
  - Postgres and SQLServer implement hash indexes
  - Oracle implements hash data file organization but not hash indexes

Dynamic Hash

- Function *h* maps values of the search key to a fix set of bucket addresses.
  - If the data file increases in size, excess buckets are generated
  - If the file decreases in size, unnecessary space is allocated
- Solutions:
  - Periodic reorganization with a new hash functions (costly, necessitates interrupting the operations)
  - The number of buckets is dynamically modified
- Extensible Hash: the hash function is dynamically modified
  - Generates values in a large set, typically integers on 32 bytes
  - At a certain moment only a prefix of the hash function is used (only the first i bits) and its length increases or decreases as needed

Extensible Hash organization General structure



$i=2, i_2 = i_3 = i, i_1 = i - 1$

Extensible Hash Use

- Every bucket j stores a value i_j
  - All the entries pointing to bucket j will have the same value on the first i_j bits
- To search for a value with the search key K_j:
  - compute $h(K_j) = X$
  - Use only the first i bits of X, scan the address table and follow the pointer to the bucket
- To insert a record with search key K_j:
  - Find bucket j as above
  - If there is room in the bucket insert the record
  - Otherwise divide the bucket and retry insertion

Extensible Hash
Dividing the bucket at insertion

To divide bucket j when inserting a new value K_j:

- If $i > i_j$
  1. Allocate a new bucket z and set $i_j = i_z = (i_j + 1)$
  2. Update the second half of the address table to point to bucket z
  3. Remove records from j and reinsert them in j or z
  4. Recompute the address of the bucket for K_j and proceed with insertion

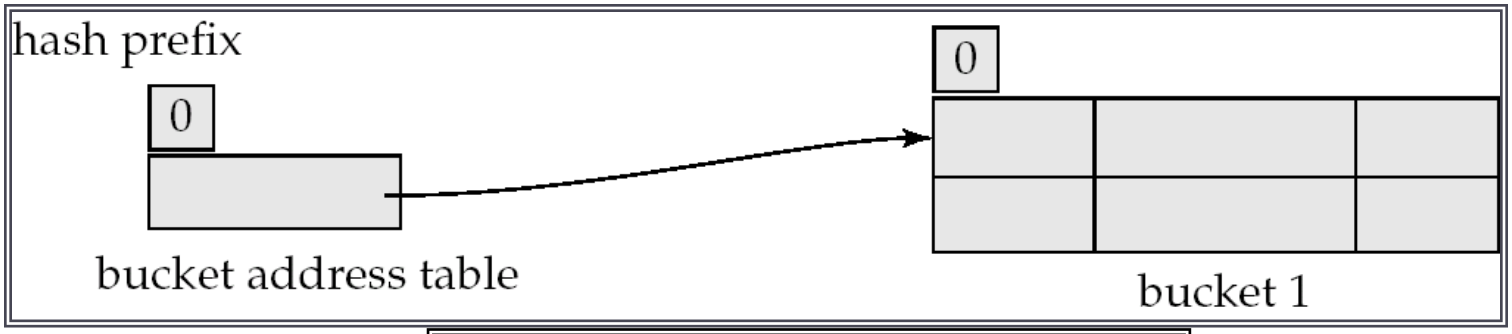
1. If $i = i_j$
   1. If for some reason a limit for i is set and this is reached, use excess buckets
   2. Otherwise
      1. Increment i and double the size of the address table
      2. Replace every entry in the table with two entries, both pointing to the same bucket
      3. Recompute bucket address for K_j and proceed with insertion (now $i > i_j$)

Extensible Hash Deletion

- To delete a record
  - Retrieve the bucket and delete the record from it
  - If the bucket gets empty perform the necessary changes in the address table
  - Buckets having the same value for i_j and the same prefix $i_j - 1$ are merged
  - Decrease the size of the address table (i) if possible

Extensible hash Example*

branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

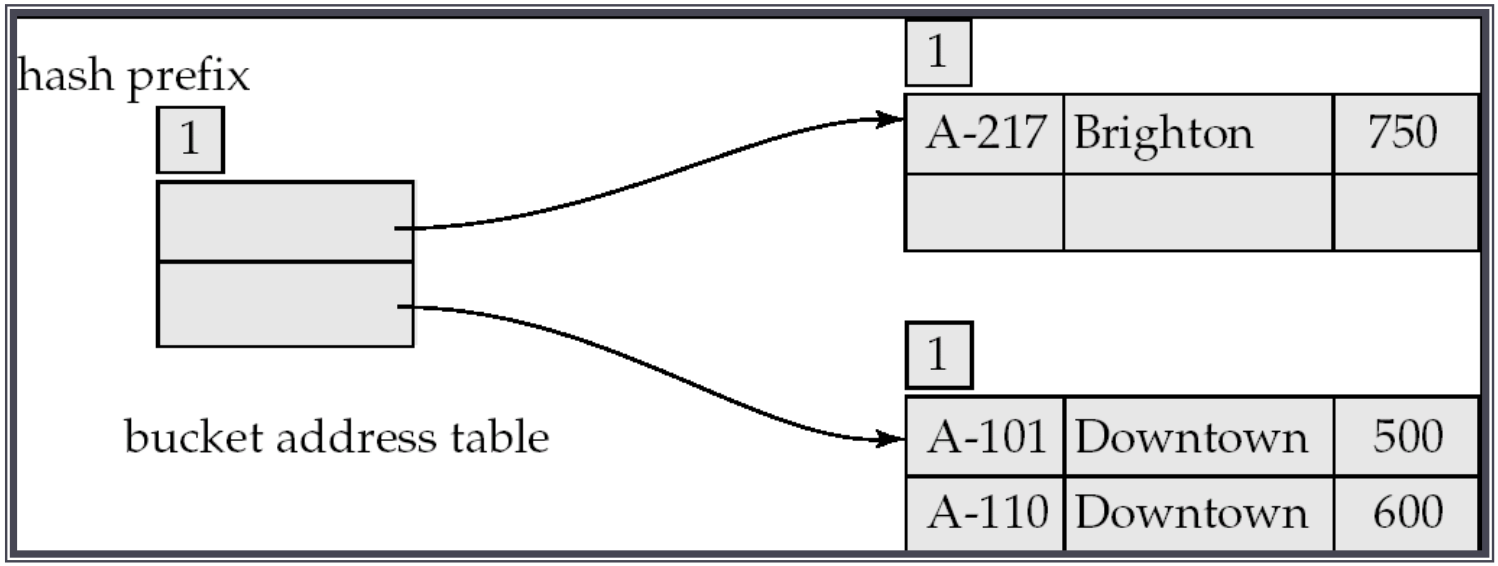


Initial hash structure, bucket size = 2

* *Avi Silberschatz Henry F. Korth S. Sudarshan. "Database System Concepts".* McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)

Extensible hash Example

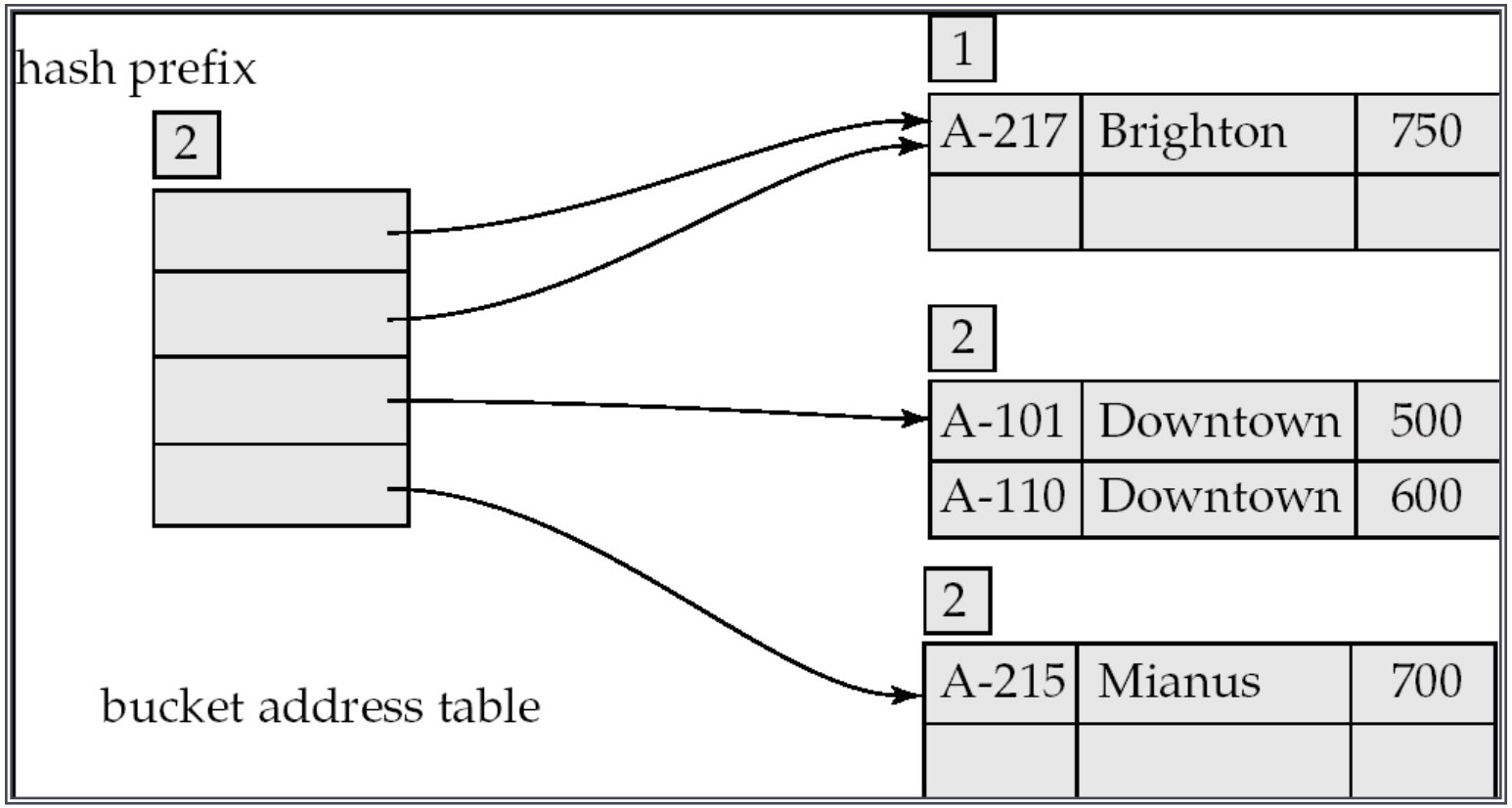
- After inserting one record Brighton and two records Downtown



* *Avi Silberschatz Henry F. Korth S. Sudarshan. "Database System Concepts".* McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)

Extensible hash Example

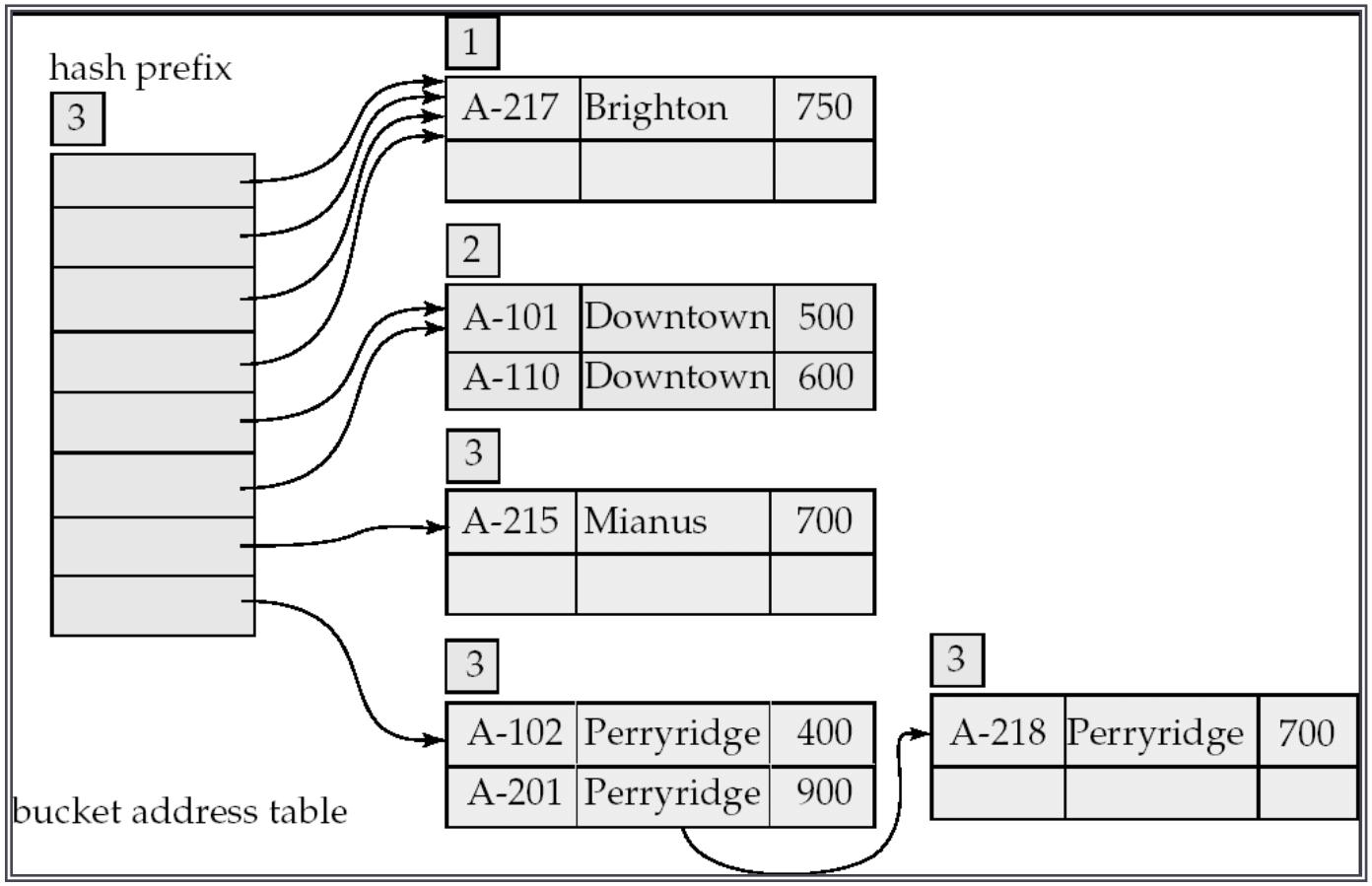
- After inserting one record Mianus



* *Avi Silberschatz Henry F. Korth S. Sudarshan. "Database System Concepts".* McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)

Extensible hash Example

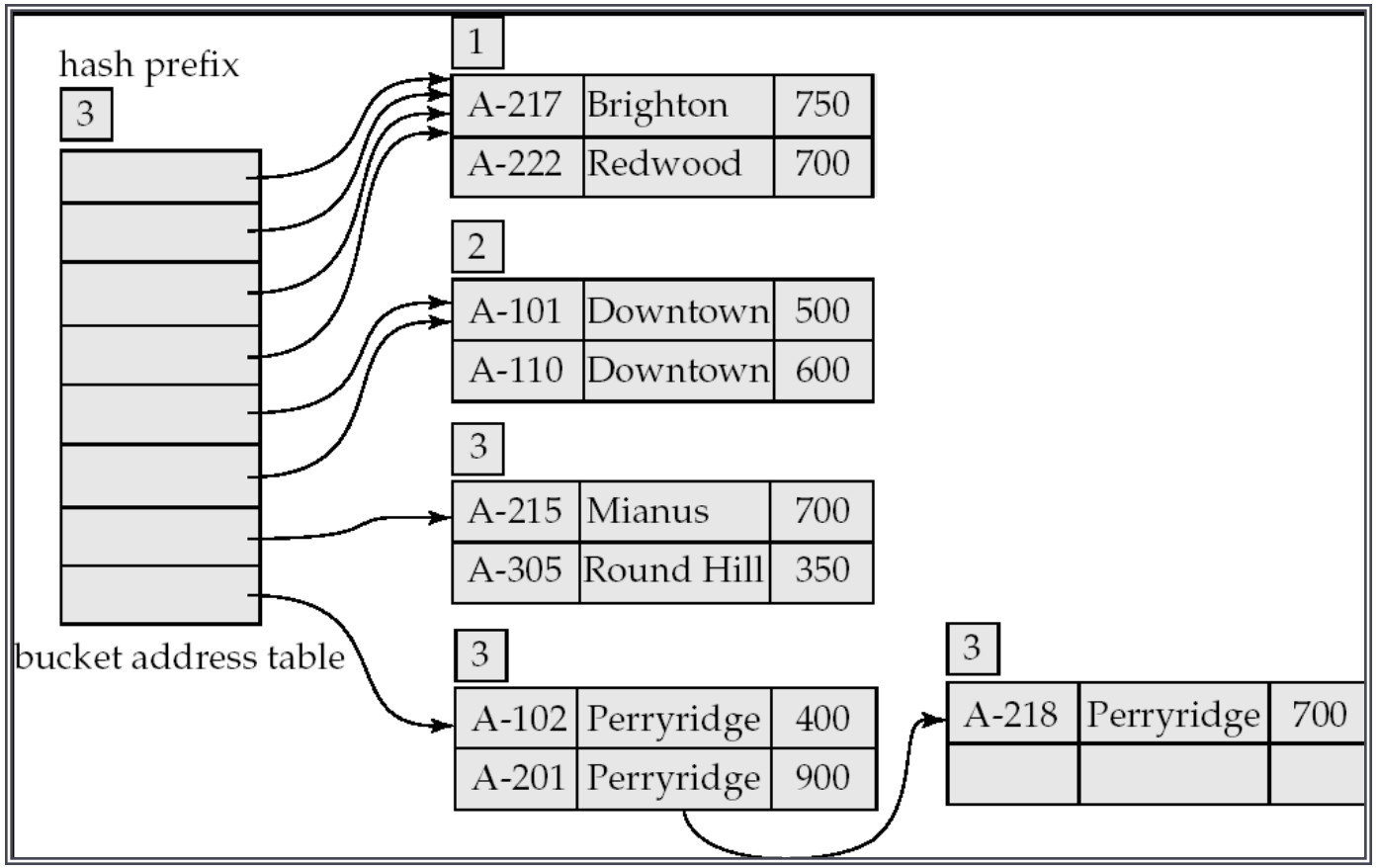
- After inserting three records Perryridge



*Avi Silberschatz Henry F. Korth S. Sudarshan. "Database System Concepts". McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)

Extensible hash Example

- After inserting records Redwood and Round Hill



* *Avi Silberschatz Henry F. Korth S. Sudarshan. "Database System Concepts".* McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)

Extensible hash Efficiency

- Advantages
  - Performance does not degrade with increasing file size
  - Minimise space allocation
- Drawbacks
  - The address table may become very large
    - Solution: use a B^+-tree to efficiently retrieve the necessary entry in the address table
  - Modifying the size of the address table is costly
- Depending on the query type:
  - Like static hashing, it is efficient when specifying a certain value, not for range queries

Bitmap indexes

Bitmap indexes

- Designed to efficiently deal with queries with several search keys
- Applicable for attributes taking a small number of distinct values
- The tuples of the relation are considered to be numbered
- Structure:
  - For each value of the attribute a bit string of length equal to the number of records
  - Value 1 means the record takes the value to which the bitstring is attached

nume	prenume	str	loc	judet
Alexa	Marian	Strada Florilor	Cluj Napoca	Cluj
Popescu	Valentin	Strada Unirii	Dej	Cluj
Andrici	Ioana	Bulevardul Republicii	Vaslui	Vaslui
Acatrinei	Marcel		Putna	Suceava
Popescu	Vasile	Bulevardul Independentei	Oradea	Bihor
Costache	Ioan	Strada Teiului	Nucet	Bihor
Ungureanu	Daniel	Aleea Amara	Arad	Arad
Sandu	Maria	Strada Victoriei	Barlad	Vaslui

Arad	0 0 0 0 0 0 1 0
Bihor	0 0 0 0 1 1 0 0
Cluj	1 1 0 0 0 0 0 0
Suceava	0 0 0 1 0 0 0 0
Vaslui	0 0 1 0 0 0 0 1

Bitmap indexes
Observations

- Queries may be solved using logical operators:
  - Intersection – AND
  - Union – OR
  - Complementariation – NOT

SELECT *
FROM students
WHERE county IN ('Arad', 'Cluj') AND year <> 2010,

  - (Arad OR Cluj) AND NOT(2010)

  - It is not necessary to access the data file
  - Also useful when the query necessitates counting
- Efficient implementation:
  - At deletion an *existence* bit string is used
  - Bitmaps are packed into words (the *word* type) on 32 or 64 bits (the AND operator on a word – one CPU instruction)

Defining indexes in SQL

Declaring indexes in SQL

- The standard does not regulate, but in practice DBMSs agreed on the same syntax
- Create:

create index <index-name> **on** <relation-name>
(<attribute-list>)

E.g.: **create index** *c-index* **on** *students(county)*

- Delete:

drop index <index-name>

- Most DBMSs allow specifying the index type
- Most DBMSs create by default indexes when specifying the *unique* constraint
- Sometimes indexes are also generates when specifying referential integrity constraints

Indexing in Oracle

- Indexes can be created on:
  - Attributes and lists of attributes
  - The result of a function over attributes
- Oracle offers support for B^+-trees by default
- Bitmap indexes are created with the syntax

  create bitmap index <index-name> **on** <relation-name> (<attribute-list>)
- Oracle does not offer support for hash indexes but implements hash file organization

Indexing in Oracle
When and how?

- It is recommended that index creation is performed after loading/inserting the data in the table (but is possible to create an index at any moment)
- Choose the right columns:
  - The columns have (mostly) unique values
  - Selection filters a small number of tuples in a large table (high selectivity ~ 15%)
  - The columns are used in a join
- Choose the appropriate index type:
  - Columns have a few distinct values -> bitmap
  - Range query -> B+trees
  - Single-value queries are frequents: -> hash file organization
  - Selection with functions -> indexes defined over functions

References

- Chapter 11 in *Avi Silberschatz Henry F. Korth S. Sudarshan.* "*Database System Concepts*". McGraw-Hill Science/Engineering/Math; 6 edition (January 27, 2010)

- Practice: execute script *12.1create* and then the statements in *12.2indexes* inspecting the execution plans