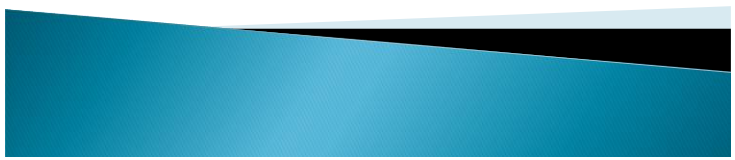
Advanced databases

Storage, indexing

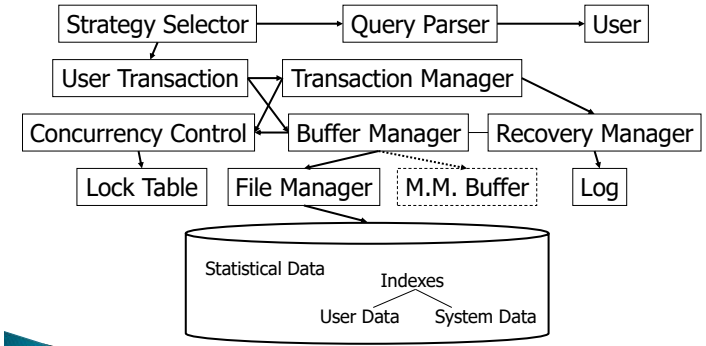
(based on materials from authors of:
Database System Implementation, Hector Garcia-Molina, Jeffrey D. Ullman , Jennifer D. Widom, Prentice-Hall, 2000)

dr inż. Artur Wilczek



1

DBMS



2

2

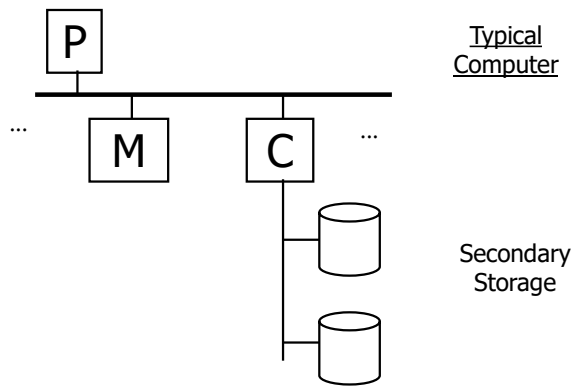
Hardware

DBMS

Data Storage

3

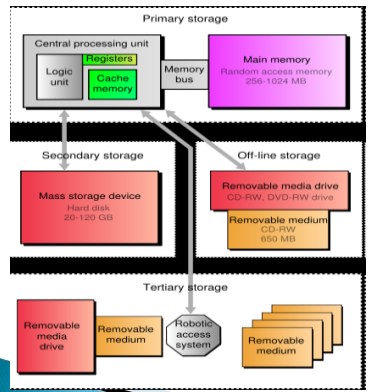
3



4

4

Memory hierarchy



- Cache memory,
- Main memory (RAM),
- Secondary storage
- (on-line storage)
- Tertiary storage
- (off-line storage)

5

5

Storage

Data Items
↓
Records
↓
Blocks
↓
Files
⋮
Memory

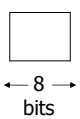
6

6

What are the data items we want to store?

- a salary
- a name
- a date
- a picture

⇨ What we have available: Bytes



7

7

To represent:

- Integer (short): 2 bytes
  e.g., 35 is

00000000 00100011

- Real, floating point
  n bits for mantissa, *m* for exponent….

8

8

To represent:

- Characters
  → various coding schemes suggested,
  most popular is ascii

Example:
A: 1000001
a: 1100001
5: 0110101
LF: 0001010

Record – Collection of related data items (called FIELDS)

E.g.: Employee record:
name field,
salary field,
date-of-hire field, ...

Types of records:

- Main choices:
  - FIXED vs VARIABLE FORMAT
  - FIXED vs VARIABLE LENGTH

Fixed format

A SCHEMA (not record) contains following information
- # fields
- type of each field
- order in record
- meaning of each field

Example: fixed format and length

Employee record
(1) E#, 2 byte integer
(2) E.name, 10 char.
(3) Dept, 2 byte code

Schema

55	smith	02
83	jones	01

Records

13

13

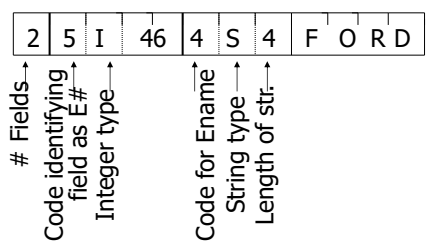
Variable format

- Record itself contains format
  "Self Describing"

14

14

Example: variable format and length



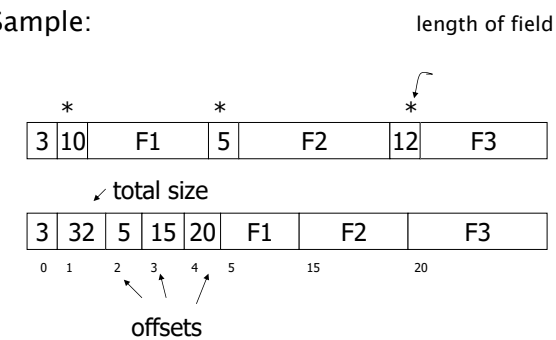
Field name codes could also be strings, i.e. TAGS

15

15

Many variations in internal organization of record

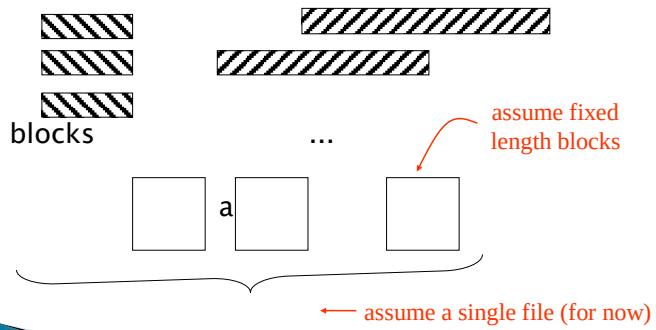
Sample:



16

16

Placing records into blocks



17

17

Options for storing records in blocks:

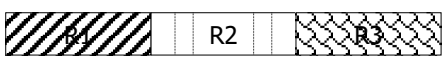
(1) separating records
(2) spanned vs. unspanned
(3) mixed record types – clustering
(4) split records
(5) sequencing
(6) indirection

18

18

(1) Separating records

Block



(a) no need to separate – fixed size recs.
(b) special marker
(c) give record lengths (or offsets)
  - within each record
  - in block header

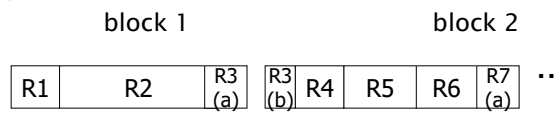
19

19

(2) Spanned vs. Unspanned

- Unspanned: records must be within one block

- Spanned



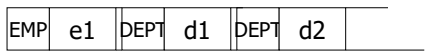
20

20

(3) Mixed record types

- Mixed – records of different types
  (e.g. EMPLOYEE, DEPT)
  allowed in same block

e.g., a block



(4) Split records

Typically for hybrid format

- Fixed part in one block
- Variable part in another block

(5) Sequencing

- Ordering records in file (and block) by some key value

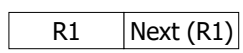
Sequential file ($\Rightarrow$ sequenced)

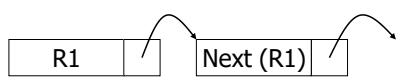
Sequencing Options

(a) Next record physically contiguous

...

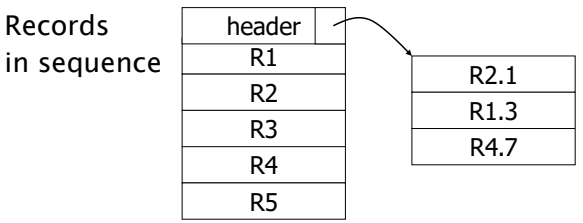
(b) Linked



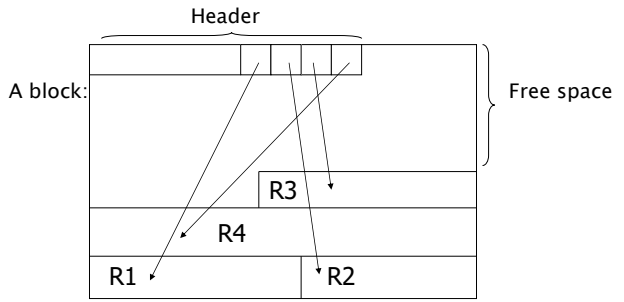


Sequencing Options

(c) Overflow area



Block



Double Buffering

Problem: Have a File

- Sequence of Blocks B1, B2

Have a Program

- Process B1
- Process B2
- Process B3

⋮

Single Buffer Solution

(1) Read B1 → Buffer
(2) Process Data in Buffer
(3) Read B2 → Buffer
(4) Process Data in Buffer ...

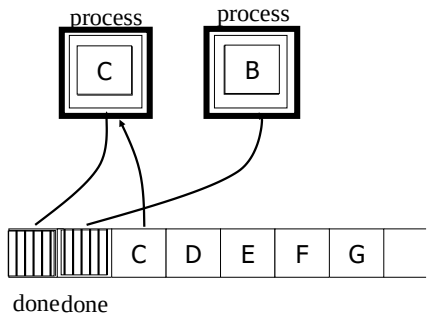
Say P = time to process/block
R = time to read in 1 block
n = # blocks

Single buffer time = n(P+R)

Double Buffering

Memory:

Disk:



Say P ≥ R

P = Processing time/block
R = IO time/block
n = # blocks

What is processing time?

- Double buffering time = R + nP
- Single buffering time = n(R+P)

☆

File – Set of blocks containing records

- Insertion, deletion, update of records
- Retrieval of a record
- Search of records with given condition

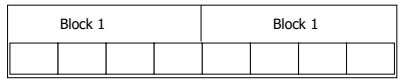
File allocation

Standard methods for allocation of blocks in files in external memory:

- Continues
- Linked
- Index based

Continues allocation

Block after block



- Efficient scan over complete file
- Costly insertion

Linked allocation

Each block have a pointer to the next block

- Costly scan of complete file
- Efficient insertion

Continues-linked

Area of continues allocated blocks (segment, extent) point to other area

- Less costly scan of complete file
- Efficient insertion

Index based allocation

Access based on indexing

- Efficient search for given record
- Efficient insertion

37

37

☆ File header – contains information organizing the file access

- Address of first block
- Schema definition
- Special separator, types dictionary definitions

38

38

☆ File organization - method of representing data, organization of records, blocks and access structures (e.g. bindings)

- Heap files
- Sorted files
- Hashed files

39

39

Heap files

Heap file - the easiest way where new records are saved in blocks in the order in which they arrive (unordered files).
The new record is saved at the end of the file.

40

40

Heap file properties

- The simplest file structure with disordered records
- As the file grows or shrinks, additional memory areas are allocated or freed
- To enable record operations, you must know:
  - number of blocks in the file
  - free area in blocks
  - number of records in blocks
- There are many ways to maintain this informationn

41

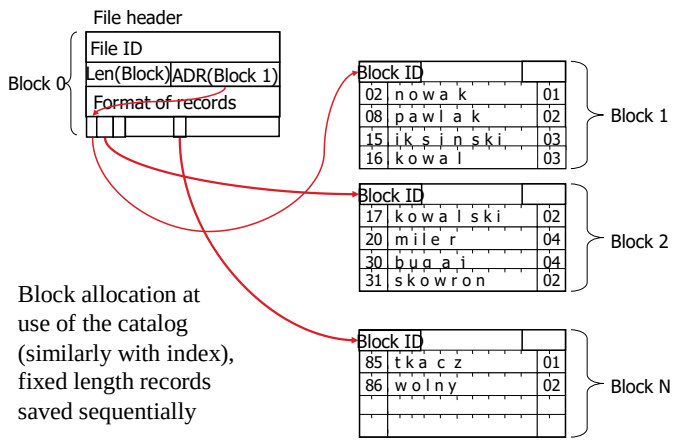
41

Sorted files

Sorted file – records stored in external memory can be ordered based on the value of the selected record field (ordering field), usually it is the record key (primary key), which provides a unique value for each record.

42

42



Block allocation at use of the catalog (similarly with index), fixed length records saved sequentially

43

43

Properties of ordered files

- Quick access to records if searched by ordering field conditions
- Quick access to records when listing file contents in order
- Finding the next record does not usually require additional block reading (the next record is often in the same block)

44

44

Search Techniques

Find - search for a record that meets the specified condition.

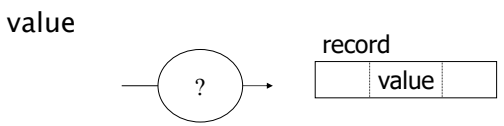
For sorted file:

- linear search
- binary search
- interpolation search
- block search

Hash files

Hash file – records stored in external memory can be searched based on the value of the selected record field (hash field) usually it is a record key (primary key) that provides a unique value for each record. Using the hash function, the address of the block that stores the record with the given key value is determined.

Indexing & Hashing



Hashing techniques

Techniques using hashing functions are used in a number of applications:

- Internal Hashing - use of hashing techniques in internal memory
- External Hashing - the use of hashing techniques in external memory while distinguishing methods
- Static hashing
- Dynamic hashing

Static allocation - properties

Static allocation of buckets - fixed number of buckets.

Let's say we have M buckets and a maximum of m records we can save in a given bucket.

If the number of records in << M * m

then we have significant "waste of space".

If the number of records in >> M * m

is the lack of space in the buckets and the need to search lists of redundant records.

Dynamic allocation

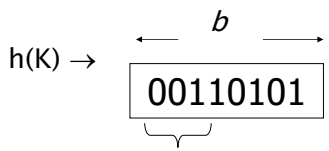
The answer to the shortcomings of static allocation due to the need to periodically reorganize the file and reduce the search efficiency due to maintaining an excess area of records is dynamic allocation of buckets, that is, an allocation that assumes a variable number of buckets in the file.

Techniques used:

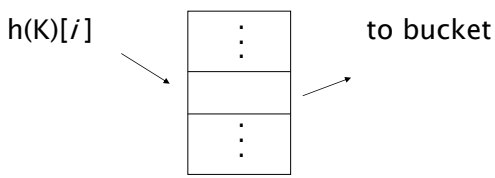
- Extensible
- Linear

Extensible hashing: two ideas

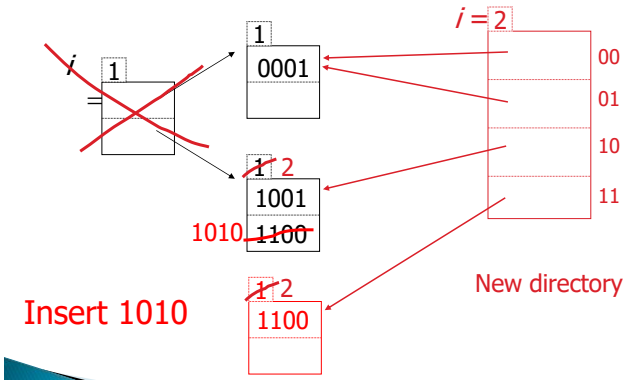
(a) Use *i* of *b* bits output by hash function



(b) Use directory



Example: h(k) is 4 bits; 2 keys/bucket



53

53

☆ Index - an additional structure and an accompanying algorithm, taking information on the input regarding certain attribute values and providing information at the output that allows the quick location of records with the specified attribute values in the data file. Used to increase the search efficiency of records that meet certain conditions.

54

54

Index properties

- The index usually does not affect the physical position of records on disk, although certain types of indexes require some specific organization of external memory
- Each of the external memory organizations discussed so far (heap, sorted, hash) can be assisted by applying an index. The index can also be built using them.
- The index in the DBMS can be built and decommissioned dynamically according to your needs.

55

55

Index Properties c.d.

- Indexes allow you to efficiently search for records that meet the search conditions based on the indexing fields used to build the index
- Each record field can be an index field. You can create indexes for different fields in the same data file.
- Indexes are most often files that contain records of a specific structure. Searching for a record requires access to an index that points to a block or blocks that store records

56

56

Index classification

Depending on the selection of the index field and the techniques of the organization, indexes can be divided into:

- single-level vs. multi-level
- primary vs. secondary
- grouping vs. non-grouping
- dense vs. rare

57

☆

B-trees – balanced trees used to build dynamic multi-level indexes.
B-trees introduce additional restrictions to maintain tree sustainability and ensure that the expected fill of tree nodes is maintained.

58

B-trees

B-trees are constructed on the basis of the following assumptions:

(1) A b-tree of the order p is a tree with nodes containing at most p-1 search box values (K_n) and pointers to records (Pr_n) And *p* pointers to descendants (P_n)
<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2> , ..., P_{q-1}, <K_{q-1}, Pr_{q-1}>, P_q>,
Where q <=p

(2) For each node: $K_1 < K_2 < ... < K_{q-1}$

(3) For each X value of the sub-tree indicated by the P_i :
- $K_{i-1} < X < K_i$, for $1 < i < q$
- $X < K_i$, for $i = 1$
- $X > K_{i-1}$, for $i = q$

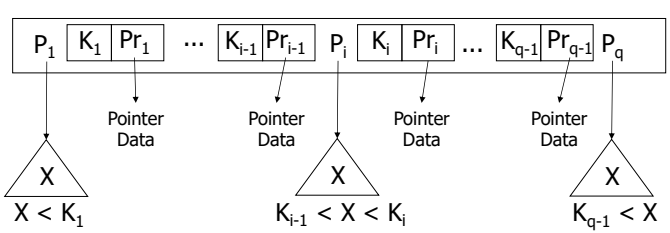
(4) Each node, except the root and leaves, contains at least $\lceil (p/2) \rceil$ pointers to descendants (root at least 2)

(5) All leaves are at the same level

59

B-trees c.d.

- Child pointers on leaves are blank
- All index field values on the B-tree are unique



60

Example B-tree (capacity)

(1) The nodes are filled on average in 70%

Then:

each node will be filled with 70% * 23 = 16 child pointers and 15 key values and record pointers

	#nodes	<key, Pr>	P
Root:	1	15	16
Level1:	16	240	256
Level2:	256	3840	4096
Level3:	4096	61440	
		Total: 65535	

61

61

☆

B+trees – most applications are based on a certain variety of B-trees called B+trees.

In B-trees, each key value appears once.

In B+trees, data pointers are found only in leaves. Therefore, the structure of the leaves differs from the other nodes.

62

62

B+trees

B+trees are constructed on the following assumptions:

(1) B+tree of the order p is a tree with internal nodes containing at most *p-1* search key values (K_n) and *p* pointers to descendants (P_n)
$<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$, Where q <=p

(2) For each internal node: $K_1 < K_2 < ... < K_{q-1}$

(3) For each X value of the sub-tube indicated by the P_i :
- $X < K_i$, for i = 1
- $K_{i-1} \le X < K_i$, for 1 < i < q
- $X \ge K_{i-1}$, for i = q

(4) Each inner node, except the root, contains at least $\lceil (p/2) \rceil$ pointers to descendants (root at least 2)

63

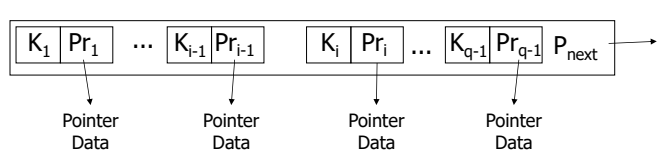
63

B+trees c.d.

(5) Each terminal node (Leaf) B+trees order *p* contains at most *p-1* search key values (K_n) and pointers to records (Pr_n) and a pointer to the next leaf (P_{next})
$<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}>$, Where q <=p

(6) For each leaf: $K_1 < K_2 < ... < K_{q-1}$

(7) Each leaf contains at least $\lfloor (p/2) \rfloor$ key values

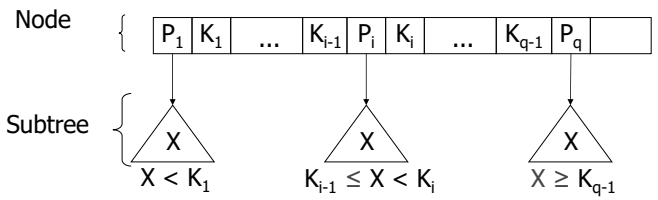


64

64

B+trees c.d.

- Pointers in internal nodes point to tree blocks, and leaf pointers point to blocks of the root file.
- Leaf blocks are linked and key values are ordered in -> dense index



65

65

Pros and Cons:

☺ In B+trees, internal nodes do not contain a pointer to the data, so more entries can fit in them => fewer levels => speed up access

☹ In B+trees, leaves, and nodes have different contents

⍰ B+trees preferred!

66

66

Example B+tree (capacity)

(1) The nodes are filled on average in 70%

Then:

each node will be filled 70% * 31 = 21 pointers to descendants i 20 key values

each leaf will be filled 70% * 31 = 21 key values

	#node	<key[, Pr]>	P
Root:	1	20	21
Level1:	21	420 (21*20)	441 (21*21)
Level2:	441	8820(441*20)	9261(441*21)
Leafs:	9261	194481(9261*21)	

Total: 194481

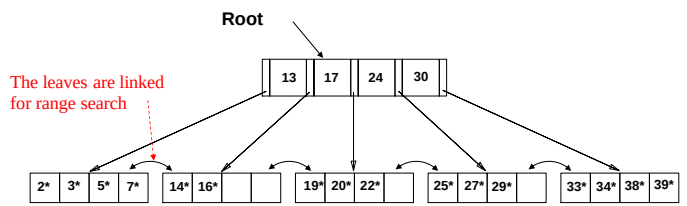
(b-tree: 65535, diff = 128946 !!!)

67

67

B+tree – Search

- The search starts at the root, and comparing the keys leads to the leaf (if equal to right).
- Search 5*, 15*, >= 24* ...



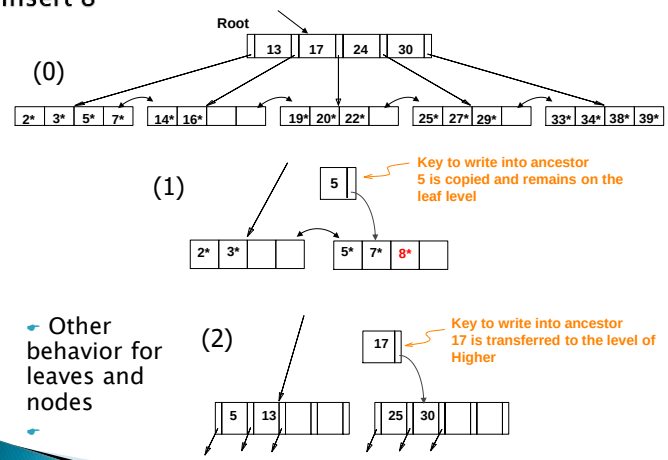
68

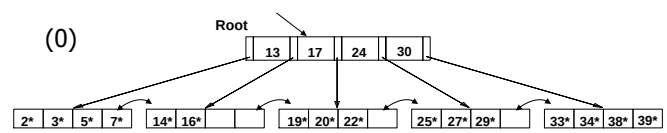
68

B+tree – Insert

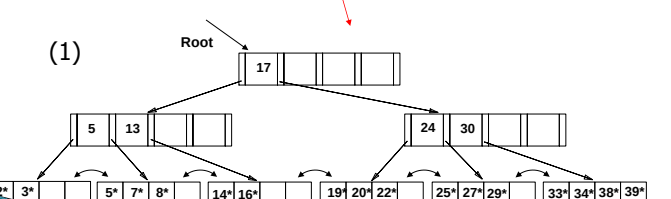
- Find the right leaf *L.*
- Insert data into*L.*
  - If there is a free space in the *L*, Ready!
  - otherwise, divide the *L (L and a new node L2)*
    - Move the contents when split relative to the middle key that is copied to the ancestor.
    - Insert the pointer to *L2* in the ancestor *L.*
- There may be recursion
  - Splitting a node moves its contents and moves the middle key to the ancestor.
  - Divisions cause the tree to grow, the number of levels may increase

Insert 8*



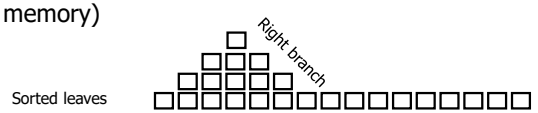


B+ tree before and after insertion 8*



B+tree – Bulk loading

- Sort records by key value
  - All operations are best performed in internal memory
  - Sequential external memory read is more efficient
  - (!) Once sorted, we already have all the tree leaves
- Insert each leaf node sequentially
  - No need to search for the right leaf
  - Only the most right branch is modified (always in memory)



Bitmap index

- Effective search on compound keys
- Records are numbered e.g. 0,1,2,…
  - Assumption:
    The read of a record number *n* must be efficient
- Works well for attributes with small cardinality
  - e.g. gender, country, month, …
  - e.g. level of income (ranges 0-9999, 10000-19999, 20000-50000, 50000- ∞)
- Bitmap = table (vector) *n* bits

Value-list bitmap index

- Value-List bitmap index:
  - Vector for every values of attribute A (B^v)
  - Length of the vector = number of records
  - Records has attribute values A ($\pi_A(R)$)
  - In vector B^v :
    - n-th bit = 1 iff n-th record has value V in attribute A
    - 0 in other cases.

	$\pi_A(R)$	B^8	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
1	3	0	0	0	0	0	1	0	0	0
2	2	0	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	0	0	1	0
4	2	0	0	0	0	0	0	1	0	0
5	8	1	0	0	0	0	0	0	0	0
6	2	0	0	0	0	0	0	1	0	0
7	2	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	1
9	7	0	1	0	0	0	0	0	0	0
10	5	0	0	0	1	0	0	0	0	0
11	6	0	0	1	0	0	0	0	0	0
12	4	0	0	0	0	1	0	0	0	0
	(a)					(b)				

Operations on data

- Logical operations on index supported
  - and
  - or
  - not

e.g. 100110 AND 110011 = 100010
100110 OR 110011 = 110111
NOT 100110 = 011001

  - Search condition: A = 7 or 2 : **Q** = B^7 OR B^2 = 0101 0110 1000
    - Easy to recognize how many records in result

Data Partitioning

- Distributes data over a number of processing elements
- Each processing element is then executed simultaneously with other processing elements, thereby creating parallelism
- Can be physical or logical data partitioning
- In a shared-nothing architecture, data is placed permanently over several disks
- In a shared-everything (shared-memory and shared-disk) architecture, data is assigned logically to each processor

Basic Data Partitioning

- Vertical vs. Horizontal data partitioning
- Vertical partitioning partitions the data vertically across all processors. Each processor has a full number of records of a particular table. This model is more common in distributed database systems
- Horizontal partitioning is a model in which each processor holds a partial number of complete records of a particular table. It is more common in parallel relational database systems

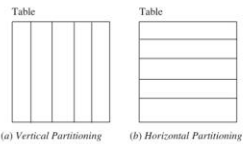


Figure 3.2 Vertical and horizontal data partitioning

D. Taniar, C.H.C. Leung, W. Rahayu, S. Goel: High-Performance Parallel Database Processing and Grid Databases, John Wiley & Sons, 2008

77

Hash data partitioning

- A hash function is used to partition the data
- Hence, data is grouped semantically, that is data on the same group shared the same hash value
- Selected processors may be identified when processing a search operation (exact-match search), but for range search (especially continuous range), all processors must be used
- Initial data allocation is not balanced either

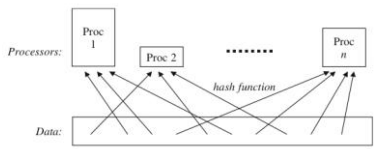


Figure 3.4 *Hash* data partitioning

D. Taniar, C.H.C. Leung, W. Rahayu, S. Goel: High-Performance Parallel Database Processing and Grid Databases, John Wiley & Sons, 2008

78

Range data partitioning

- Spreads the records based on a given range of the partitioning attribute
- Processing records on a specific range can be directed to certain processors only
- Initial data allocation is skewed too

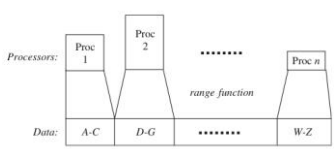


Figure 3.5 *Range* data partitioning

D. Taniar, C.H.C. Leung, W. Rahayu, S. Goel: High-Performance Parallel Database Processing and Grid Databases, John Wiley & Sons, 2008

79