

Lecture

Database Implementation Techniques / Databases II

OvGU Magdeburg, WinSem 2009/2010

Sandro Schulze, Gunter Saake

{sanschul,saake}@iti.cs.uni-magdeburg.de.

Bitmap Indices

- Idea: *Bit array* for encoding the assignment of tuple to attribute value
- Comparison with tree-based index structures:
 - ◆ Avoidance of degenerated B-Trees
 - ◆ More insensbile against higher number of attributes
 - ◆ Easier support of queries, where only some (of the indexed) attributes are confined
 - ◆ In return, usually higher effort for updates
 - For example, in Data Warehouses this is quite unproblematic because of the predominant read access

Bitmap Index: Realization

- Principle: Replacement of TIDs (rowid) by a bit list for a key value in B^+ -Tree
- Structure of nodes:

			order status
F : 010010 ... 01	O : 0111010 ... 00	P : 100000 ... 10	

- Advantage: Less memory requirements
 - ◆ Example: 150.000 tuples, 3 different key values, 4 byte for TID
 - B^+ -Tree: 600 KB
 - Bitmap: $3 \cdot 18750$ Byte = 56KB
- Disadvantage: Effort for update

Bitmap Index: Realization /2

- Definition in Oracle

```
CREATE BITMAP INDEX order_status_idx  
ON order(status);
```

- Storage in compressed form

Standard Bitmap Index

- Every attribute is stored separately
- For each value of an attribute a bitmap vector is created:
 - ◆ For each tuple one bit exists, which is set to 1, if the indexed attribute in the tuple contains the reference value of this bitmap vector
 - ◆ The number of resulting bitmap vectors per dimension corresponds to the number of different values, which exist for the attribute

Standard Bitmap Index /2

- Example: Attribut gender
 - ◆ 2 values (m/f)
 - ◆ 2 bitmap vectors

PersId	Name	gender	Bitmap-f	Bitmap-m
007	James Bond	M	0	1
008	Amelie Lux	F	1	0
010	Harald Schmidt	M	0	1
011	Heike Drechsler	F	1	0

Standard Bitmap Index /3

- Selection of tuples by concatenation of respective bitmap vectors
- Example: Bitmap index on attributes gender and month of birth
 - ◆ (i.e., 2 bitmap vectors B-f und B-m for gender 12 bitmap vectors B-1, ..., B-12 for the month, if all months occur)
- Query: „all women, born in march“
 - ◆ Computation: $B-f \wedge B-3$ (concatenated bitwise conjunctive)
 - ◆ Result: all tuples, where at the position in the bitmap vector (in the result) a 1 occurs

Multicomponent Bitmap Index

- With bitmap standard index \Rightarrow attributes with many values result in a great many of bitmap vectors
- $\langle n, m \rangle$ -Multicomponent bitmap indices enable that $n \cdot m$ possible values can be indexed by $n + m$ bitmap vectors
- Every value $x (0 \leq x \leq n \cdot m - 1)$ can be represented by two values y and z :

$$x = n \cdot y + z \text{ with } 0 \leq y \leq m - 1 \text{ and } 0 \leq z \leq n - 1$$

- ◆ As a result, at most m bitmap vectors for y and n bitmap vectors for z are needed
- ◆ Memory requirements reduced from $n \cdot m$ to $n + m$
- ◆ In exchange, for a exact match query two bitmap vectors have to be read

Multicomponent Bitmap Index (II)

- Example: Two-component bitmap index
- for $M = 0..11$ about $x = 4 \cdot y + z$
- y-values: B-2-1, B-1-1, B-0-1
- z-values: B-3-0, B-2-0, B-1-0, B-0-0

x	y			z			
M	B-2-1	B-1-1	B-0-1	B-3-0	B-2-0	B-1-0	B-0-0
5	0	1	0	0	0	1	0
3	0	0	1	1	0	0	0
0	0	0	1	0	0	0	1
11	1	0	0	1	0	0	0

Example: Postal Code

- Encoding of postal code
- Values from 00000 to 99999
- Direct realization: 100.000 columns
- Two-component bitmap index (first 2 digits + 3 last digits): 1.100 columns
- Five components: *50 columns*
 - ◆ Suited for range queries „PLZ 39***“
- Binary coded (up to 2^{17}): 34 columns
 - ◆ only for exact match queries!
- *Note: encoding with base 3 results in only 33 columns....*

Multidimensional Storage Techniques

- So far: onedimensional (no partial-match queries, only linear order)
- Now: multidimensional (even partial-match queries, positioning in multidimensional data space)
- k dimensions = k attributes can be supported equal
- This section
 - ◆ multidimensional B-Tree
 - ◆ multidimensional hash technique
 - ◆ Grid files
- Further multidimensional techniques for multimedia and geo(graphical) data in next chapter

Multidimensional Tree Techniques

KdB-Tree is a B^+ -Tree, where the index pages are realized as binary trees with access attributes, access attribute values and pointer

Variants of k -dimensional index trees:

- *kd-Tree* by Bentley and Friedman: Multidimensional basic structure (binary tree) developed for main memory algorithms
- *KDB-Tree* by Robinson: Combination of kd-Tree and B-Tree (k -dimensional index tree with higher branching factor)
- *KdB-Tree* by Kuchen: Improvement of Robinsons variant, covered in this lecture

Multidimensional Tree Techniques (II)

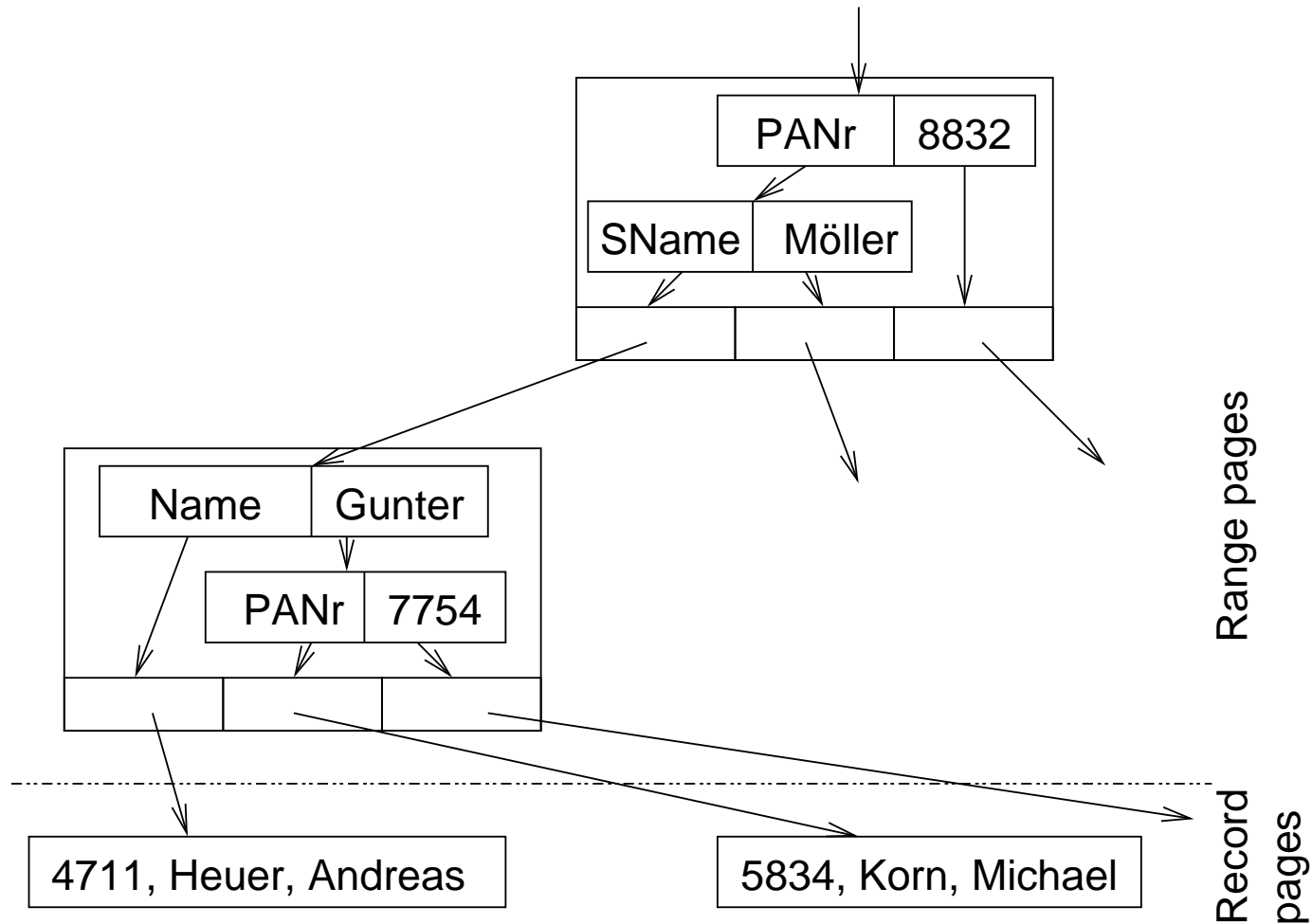
- KdB-Tree can support primary key and several secondary keys in parallel
- Usage as file organisation form \Rightarrow additional secondary indices are unnecessary

Definition KdB-Tree

Idea: on each index page a partial tree is represented, which branches after several consecutive attributes

- *KdB-Tree of Type (b, t)* consists of
 - ◆ *Inner nodes (range pages)* which contain a *kd-Tree* with at most b internal nodes
 - ◆ *Leaves (record pages)* which can contain up to t tuples of the stored relation
- Range pages: *kd-Tree* contained with *slice elements* and two pointer
 - ◆ Slice element contains *access attribute* and *access attribute value*; left pointer: smaller access attribute values; right pointer: bigger access attribute values

Example



KdB-Tree: Structure

- Range pages
 - ◆ Number of slice and address elements of the page
 - ◆ Pointer to root of contained kd-Tree
 - ◆ *Slice and address elements.*
- Slice element
 - ◆ Access attribute
 - ◆ Access attribute value
 - ◆ Two pointer to child node of the kd-Tree of this page
(can be slice as well as address elements)
- Address elements: Address of a successor of the range page in KdB-Tree (range or record page)

KdB-Tree: Operations

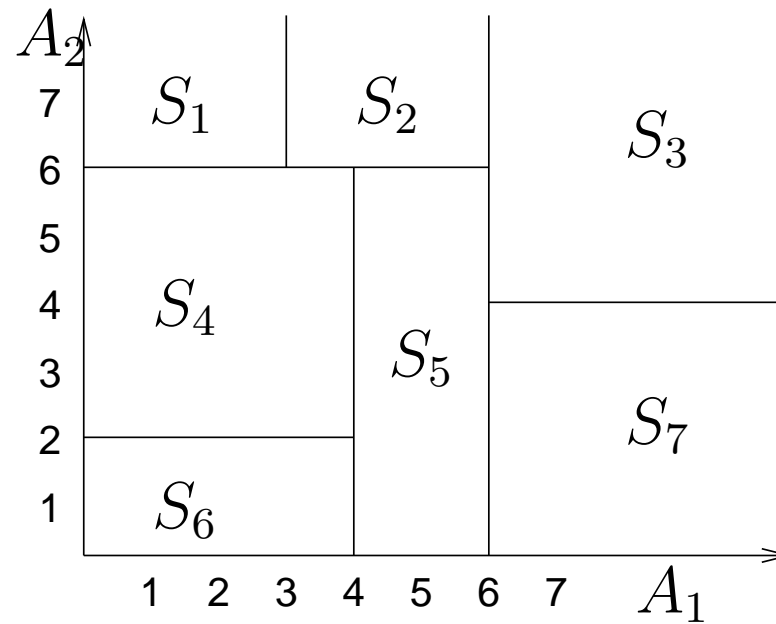
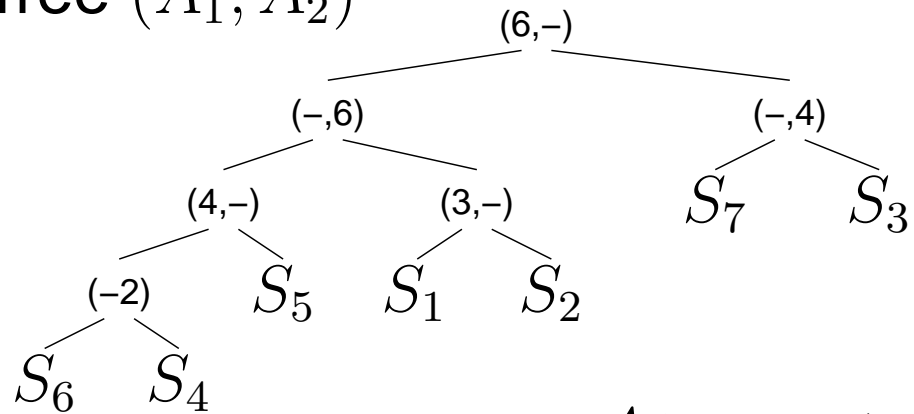
- Complexity **lookup, insert** and **delete** for *exact-match* $O(\log n)$
- For *partial-match* better than $O(n)$
- If t of k attributes are specified in the query: Access complexity of $O(n^{1-t/k})$

KdB-Tree: Slice attributes

- Order of slice attributes
 - ◆ Determined cyclic
 - ◆ or selectivities are included: Access attribute with high selectivity should be used earlier and more frequently as slice element
- slice attribute value: Because of information on distribution of attribute values a suitable „center“ of a attribute value range to be sliced should be determined

KdB-Tree: Brickwall

2d-Tree (A_1, A_2)



Multidimensional Hashing

- Idea: Bit interleaving
- The bits of the address of different access attribute values are computed alternately
- Example: Two dimensions

	*0*0	*0*1	*1*0	*1*1
0*0*	0000	0001	0100	0101
0*1*	0010	0011	0110	0111
1*0*	1000	1001	1100	1101
1*1*	1010	1011	1110	1111

MDH by Kuchen

Idee

- MDH is based on linear hashing
- Hash values are bit sequences, where the beginning sequence is used as current hash value respectively
- One bit string per participating attribute is computed respectively
- Beginning sequences are now processed cyclical by the principle of bit interleaving
- Hash value is composed in turns from the bits of the single values

MDH formal (I)

- Multidimensional value x

$$x := (x_1, \dots, x_k) \in D = D_1 \times \dots \times D_k$$

- Sequence of hash functions, indexed with i , is constructed
- i -th hash function $h_i(x)$ is composed by a composition function \bar{h}_i from the respective i -th beginning sequences of the local hash values $h_{i_j}(x_j)$:

$$h_i(x) = \bar{h}_i(h_{i_1}(x_1), \dots, h_{i_k}(x_k))$$

- Local hash functions h_{i_j} result into bit vector of length z_{i_j} :

$$h_{i_j} : D_j \rightarrow \{0, \dots, z_{i_j}\}, j \in \{1, \dots, k\}$$

MDH formal (II)

- z_{i_j} should be equal, so that the dimensions are considered equally
- Composition function \bar{h}_i composes local bit vectors to bit vector of length i :

$$\bar{h}_i : \{0, \dots, z_{i_1}\} \times \dots \times \{0, \dots, z_{i_k}\} \rightarrow \{0, \dots, 2^{i+1} - 1\}$$

MDH formal (III)

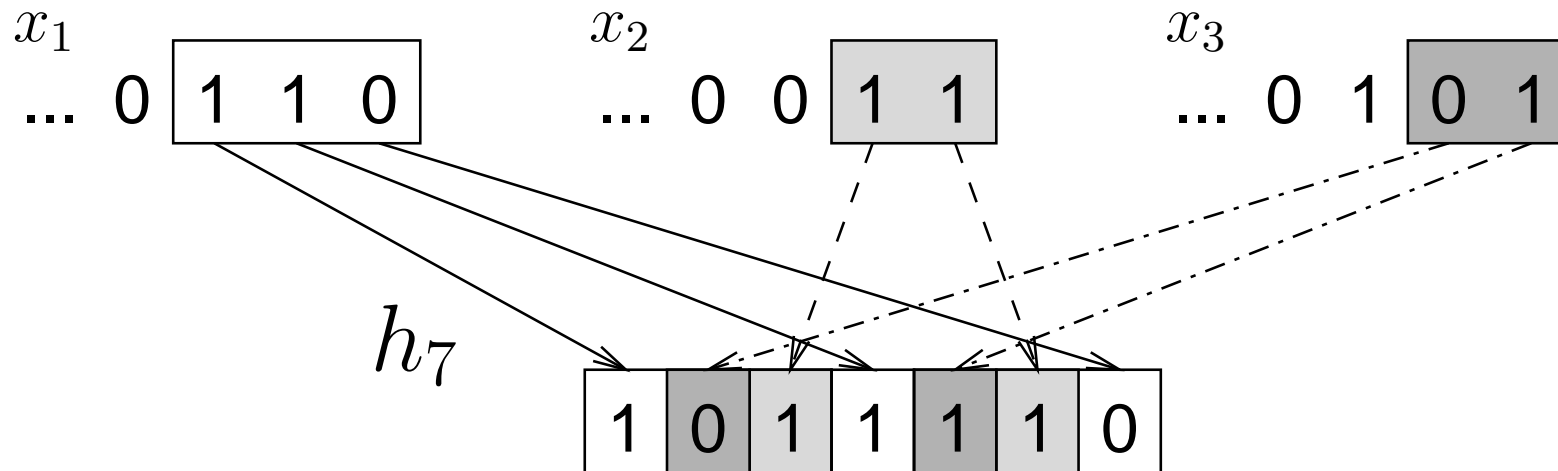
- Balanced length of z_{ij} is determined by following definition, which increases the lengths cyclical for each enhancement step (for one position):

$$z_{ij} = \begin{cases} 2^{\lfloor \frac{i}{k} \rfloor + 1} - 1 & \text{für } j - 1 \leq (i \bmod k) \\ 2^{\lfloor \frac{i}{k} \rfloor} - 1 & \text{für } j - 1 > (i \bmod k) \end{cases}$$

- Composition function:

$$\bar{h}_i(x) = \sum_{r=0}^i \left(\frac{(x_{(r \bmod k)+1} \bmod 2^{\lfloor \frac{r}{k} \rfloor + 1}) - (x_{(r \bmod k)+1} \bmod 2^{\lfloor \frac{r}{k} \rfloor})}{2^{\lfloor \frac{r}{k} \rfloor}} \right) 2^r$$

MDH Illustration



- Clarifies composition of hash function h_i for three dimensions and the value $i = 7$
- Highlighted parts of bit string correspond to the values $h_{7_1}(x_1)$, $h_{7_2}(x_2)$ und $h_{7_3}(x_3)$
- During the step to $i = 8$ a further bit of x_2 (precise: of $h_{8_2}(x_2)$) would be used

MDH Complexity

- Exact-Match queries: $O(1)$
- Partial-Match queries, with t of k attributes determined, complexity $O(n^{1-\frac{t}{k}})$
- Results from the number of pages, if certain bits are „unknown“
- Special cases: $O(1)$ for $t = k$, $O(n)$ for $t = 0$

Grid Files

- Most common and attractive multidimensional file organisation form (regarding the underlying technique)
- Separate category: Elements are combinations of key transformation (like hash techniques) and index files (like tree techniques)
- Multidimensional space is partitioned equally (in contrast to Brickwall)

Grid File: Objectives

Hinrichs and Nievergelt

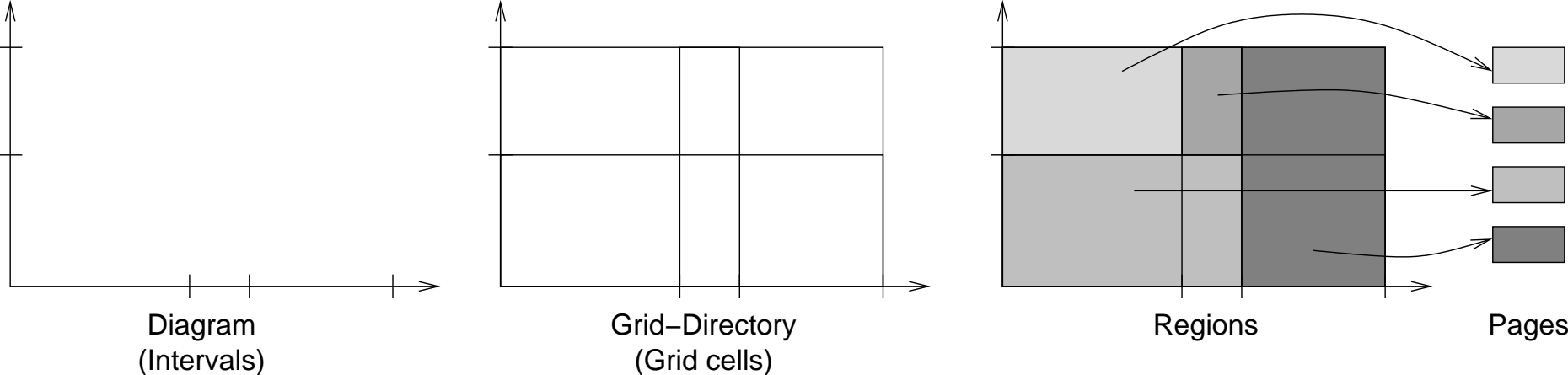
- Principle of two disk accesses: Every record should be achievable in two accesses, regarding an *exact-match* query
- Decomposition of data space in cuboids: n -dimensional cuboids represent the search regions of the grid file
- Principle of adjacency preservation: Similar objects should be stored on the same page
- Symmetric treatment of all space dimensions: *partial-match* queries possible
- Dynamical adaptation of grid structure in the case of delete and insert operations

Principle of Two Disk Accesses

For exact-match

1. sought-after k -tuple is mapped to the intervals of the *scales*; index values are computed as combination of determined intervals; scales in main memory \Rightarrow no disk access
2. Access on *Grid-Directory* via computed index values; here, addresses of record pages are stored; first *disk access*.
3. Record access: second *disk access*.

Structure of a Grid File (I)



Structure of a Grid File (II)

- *Grid*: k onedimensional fields (scales), every scale represents an attribute
- *Scales* consist of partition of the corresponding range of values in *intervals*
- *Grid directory* consist of grid cells, which partition the data space in cuboids
- *Grid cells* represent grid region; to each region exactly one record page is assigned
- *Grid region*: k -dimensional, convex structure (regions are pairwise disjunctive)

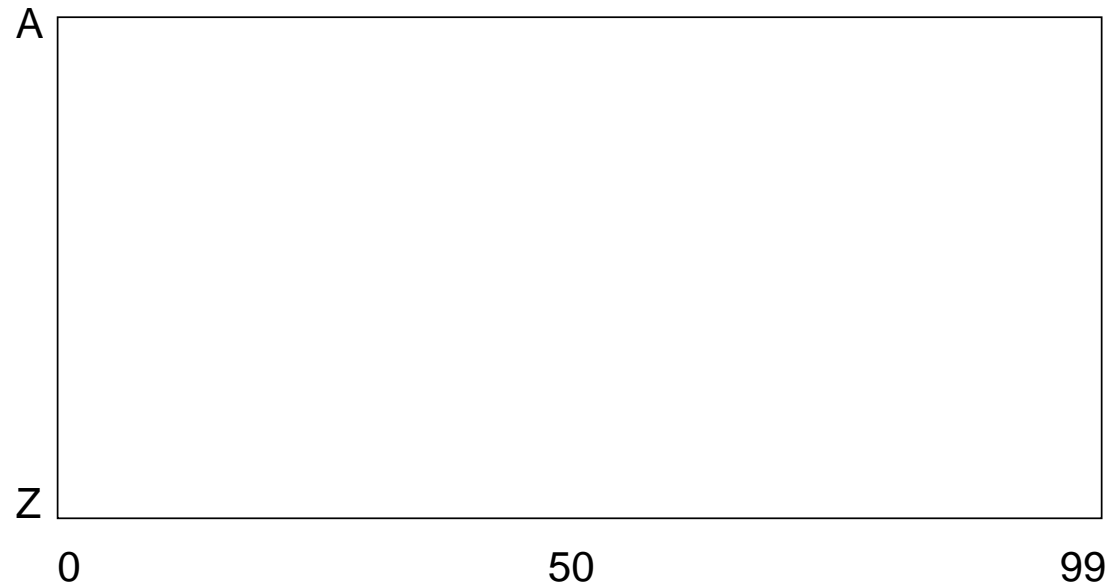
Operations

At the beginning: Cell = Region = one record page

- *Page overflow*: Page is split. If the grid region, corresponding to the page, consists of only one grid cell, an interval on the scale must be partitioned into two intervals. If a region consists of several cells, these cells have to be separated in several (single) regions.
- *Page underflow*: Two regions are merged to one region, if the result yields to a new, convex region.

Example

- Initial grid file



- Insert records: (45, D), (2, A), (87, S), (75, M), (55, K), (3, Z), (15, D), (25, K), (48, F)
- Every page of the grid file can contain up to three records

Buddy System

- Described technique: *Buddy system* (twin system)
- Cells, resulting from the same step can be merged to regions; no other aggregation of cells is allowed within the buddy system
- Inflexible deletion: Only merging of regions is allowed, which have been evolved as twins
- Example: Delete (15,D): Merge pages 1 and 4; delete (87,S) , page 2 is understaffed, but can not be merged with any other page

Cluster Creation

- Jointly storage of records on pages
- Important special cases:
 - ◆ *Cluster on key attributes.*
Support of range queries and grouping: Records are stored coherent in sorting order on pages \Rightarrow *index-organized tables* or clustered, densely populated primary indices
 - ◆ *Cluster on foreign key attributes.*
Groups of records, sharing one attribute value, are clustered on pages (support of join queries)
- Component relations instead of join attributes in OODBS

Index-Organized Tables

- Tuples are directly contained in index
- However, in case of frequent splitting the TID concept is meaningless
- Further secondary index can not be created due to missing TID
- E.g., **unique** not possible

Cluster for Join Queries

Join attribute: Cluster key

OrderNo	Order date	Customer	Delivery date
100	15.04.98	Orion Enterprises	01.01.2001

Position	Part	Amount	Price
1	Aluminiumtorso	2	3145,67
2	Antenna	2	32,50
3	Overkill	1	1313,45
4	Rivets	1000	-.50

OrderNo	Order date	Customer	Order.date
123	05.10.98	Kirk Enterpr.	31.12.1999

Position	Part	Amount	Price
1	Beamer	1	13145,67
2	Energiekristall	2	32,99
3	Phaser	5	1313,45
4	Rivets	2000	-.50

Definition of Cluster

```
create cluster OrderCluster
  (OrderNo number(3))
  pctused 80 pctfree 5;

create table T_Order (
  OrderNo number(3) primary key, ...)
  cluster OrderCluster (OrderNo);

create table T_Orderposition (
  Position number(3),
  OrderNo number(3) references T_Order,
  ...
  constraint OrderPosKey
    primary key (Position, OrderNo)
  )
  cluster OrderCluster (OrderNo);
```

Organisation of Cluster

- *Indexed cluster* use an sorted index (e.g. B⁺-Tree) on the cluster key for access on the cluster
- *Hash cluster* determine the suitable cluster with a hash function
- Indices for cluster correspond to normal indices for the cluster key
- Cluster identifiers or direct storage addresses are used instead of TIDs (for hash techniques)

Indexed Cluster

```
create index OrderClusterIndex  
on cluster OrderCluster
```

Hash Cluster

```
create cluster OrderCluster (  
    OrderNo number(5,0))  
    pctused 80  
    pctfree 5  
    size 2k  
    hash is OrderNo  
    hashkeys 100000;
```

Physical Data Definition in SQL

	Ingres	Oracle	DB2	Informix
sequential, Heap	+	+	+	+
index-seq. (sparsely)	+	-	-	-
indexed non-seq. (densely)	+	-	-	-
multilevel	+	-	-	-
B-Tree	-	-	-	-
B ⁺ -Tree (densely)	+	+	+	+
KdB-Tree	-	-	-	-
Hash	+	(+)	-	-
MDH	-	-	-	-
Cluster	-	+	+	-

Ingres

```
create [ unique ] index indexname
      on relname (
attrname [ asc | desc ],
...
) [ with-clause ];
```

Information of physical storage in **with-clause**

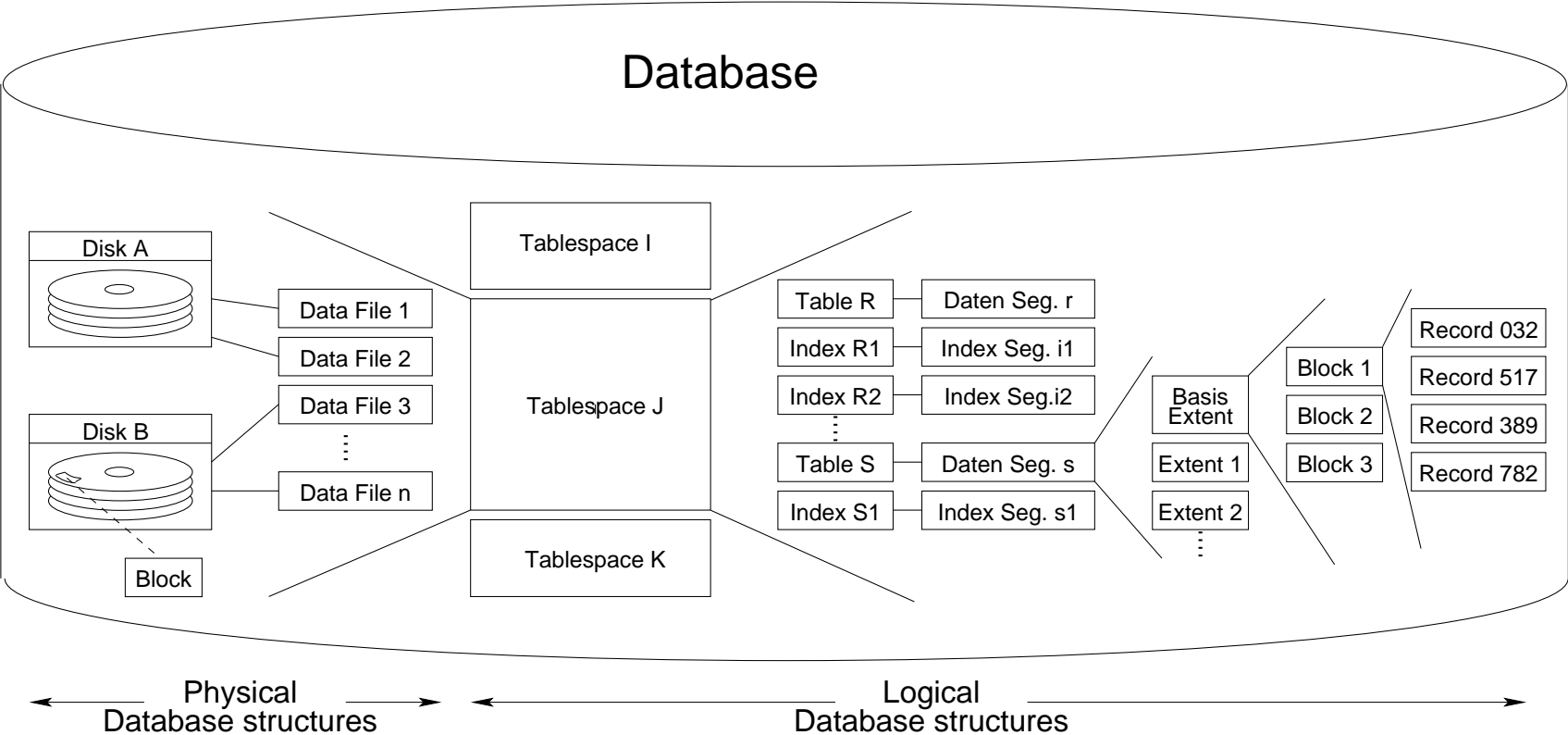
```
structure = cbtree | btree | cisam | isam
          | chash | hash,
key = ( attrname, ... ),
minpages = n, maxpages = n, fillfactor = n,
leaffill = n, noleaffill = n,
location (location, ...)
```

Ingres: modify

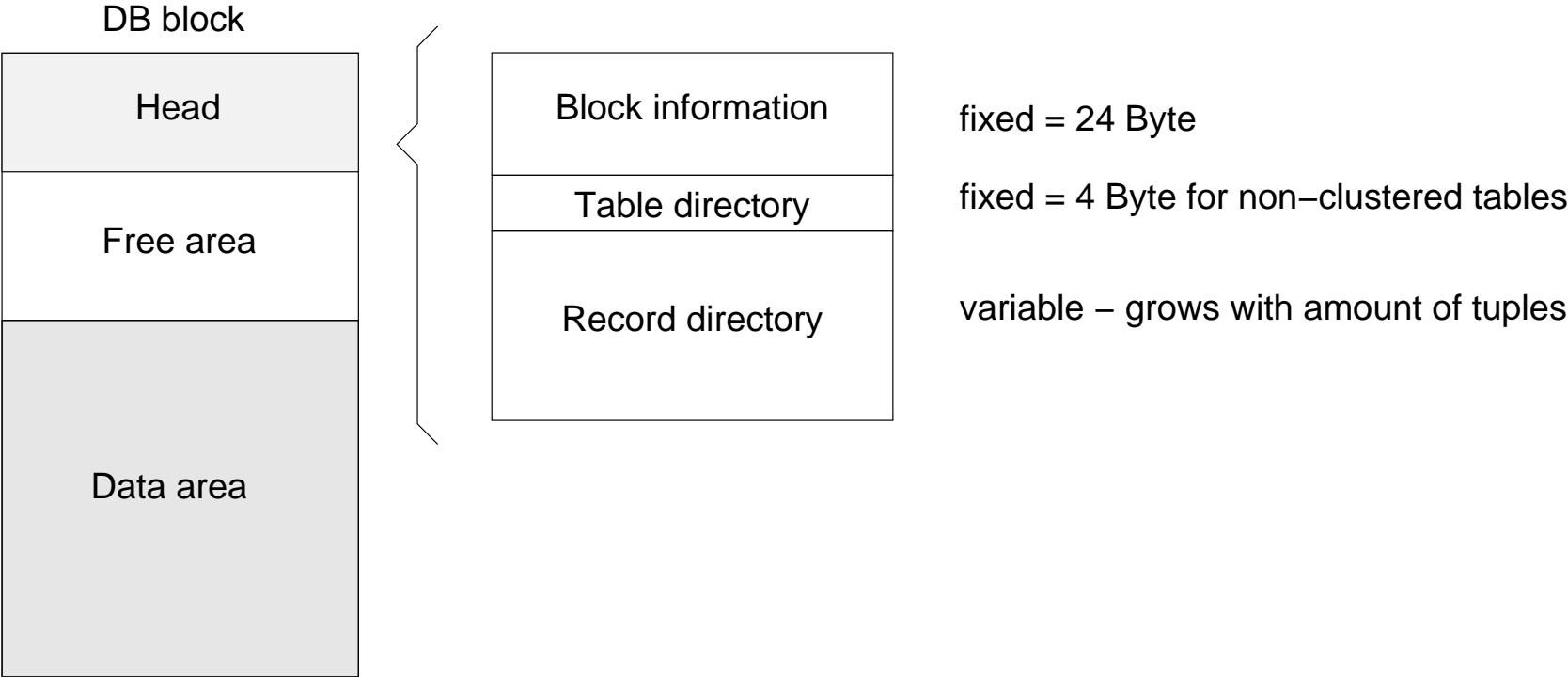
```
modify rel-or-index-name  
to storage-structure  
[ on attr-name [asc|desc]  
{ , attr-name [asc|desc] } ]  
with-clause
```

- Furthermore compressed versions for all organisation forms
- Especially useful for strings to be indexed: (CHeap, CHash, CBtree, ...)
- Addressing of records takes place using the TID concept

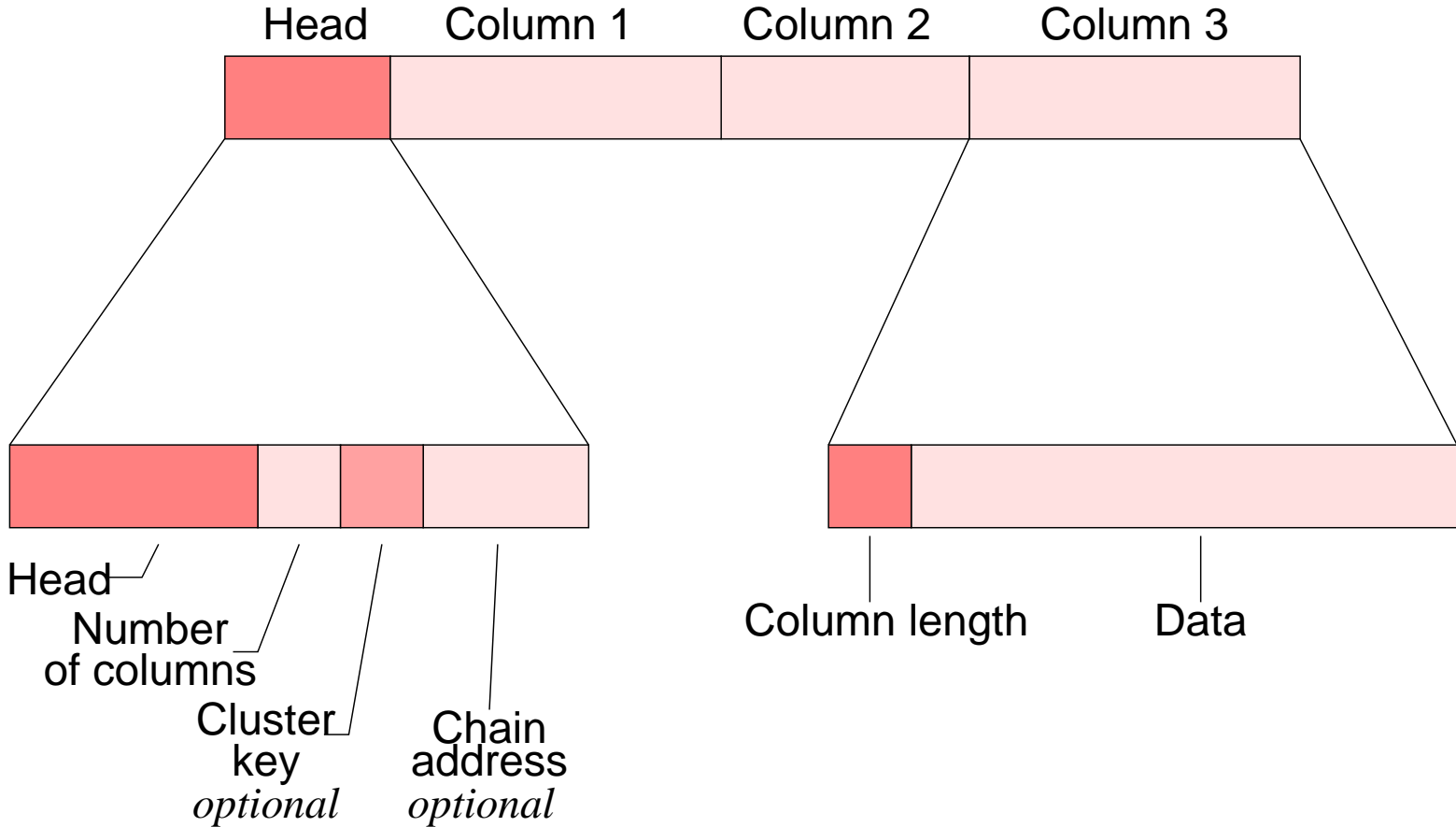
Oracle



Oracle: Blocks



Oracle: Records



Oracle: Data Organisation

- Standard index ist structured as B^+ -Tree
- Index-organized tables store tuples directly in the leaves of B^+ -Tree
- Clustering of several relations possible; cluster indices can be organized as B^+ -Tree or hash index
- Bitmap indices store bit matrices for enumeration attributes (see Data Warehouses)
- Reverse indices interpret bytes of primary key in reverse order \Rightarrow take off of storage in sorting order

Informix

- **dbspace** (several Chunks – raw file) und **blobspace**
- Chunks - Extents - Page (Timestamp, TID field)
- Pages within an extent (guaranteed clustered)
 - ◆ *Bitmap Page*: Directory of all pages in extent
 - ◆ *Data Page*: Real data from tables
 - ◆ *Remainder Page*: Spanned records with overflow pages
 - ◆ *Index Page*: Index data
 - ◆ *Blob Page*: BLOBs with buffer and log mechanisms (in original record reference to start page of list of BLOB pages)
 - ◆ *Free Page*: Free pages in this extent
- **tablespace**: Several Extents

Informix: Data Definition

- Index structure: B⁺-Tree

```
create [ unique | distinct ] [ cluster ]
      index indexname on relname (
  attrname [ asc | desc ],
  attrname [ asc | desc ], ... )
      [ using indexart, ]
      [ fillfactor = percent ];
```

- **cluster** option: clustered index (only for new creation this characteristic is ensured)
- **using**: „B-tree“ for B⁺-Tree, „R-tree“
- **fillfactor**: initial 90%
- Methods can be indexed instead of attributes

Secondary index: B⁺-Tree

- Clustered and densely populated **clustered**
- Non-clustered and densely populated **non-clustered**

DB2 (II)

tablespace

- Set of *container*
- Container: OS directory, OS file, secondary storage medium
- **tablespace**: For several tables (physical clustering)
- Several **tablespaces** per table: (index files, BLOB files, ...)
- Granularity for allocation: *Extents*
- OS-managed (SMS, System Managed Space)
- DB2-managed (DMS, Database Managed Space)

DB2: bufferpool

tablespace in main memory is per Default a **bufferpool**

- Changes of size of **bufferpools**
- Creation of several **bufferpools** for one **tablespace**
- **prefetchsize**: Number of pages to be loaded in case of *page fault*

DB2: reorg

- Records of a table are stored in interrelated parts of the storage again
- Redetermine sorting of a file
- Eliminate overflow pages after increase of records (two-level TID concept)