Data-Warehouse-Technologien

Exercise sheet 9

Assignment 1: **Classify the index structures discussed in the lecture!**

Assignment 2: **Define the properties of an ideal index for data warehouses.**

Assignment 3: **Compare the resource consumption of B+- and standard bitmap indizes for the following data:**

1. **number of tuples: 5000000; number of key values: 100; resource consumption for 1 TID in Byte: 1;**
2. **number of tuples: 5000000; number of key values: 3; resource consumption for 1 TID in Byte: 1;**
3. **number of tuples: 5000000; number of key values: 3; resource consumption for 1 TID in Byte: 4;**

Assignment 4: **Transform the postal code 39106 into a bitmap index using the Standard-Bitmap-Index and the Multi-Component-Bitmap-Index. How can Bitmap-Indexes be used to support range queries for postal codes?**

Assignment 5: **Construct a R-Tree for the following data. Insert the data in the order of the given list. A node has at most 3 entries, thus a MBR is described. Outline the insertion process. Assume an approach to minimize the number of boxes.**

(10 Z), (20 K), (30 A), (40 L), (50 M), (60 N), (70 D), (80 X)

Assignment 6: **What is a Gridfile? How is it created? what are its advantages compared to tree based approaches? Use the information of exercise 5 to create a grid file.**

 Good Luck!

Assignment 1

Classify index structures discussed in the lecture

Index structure:
- organisator. Mechanismen, die genutzt werden, um die Leistung von Abfragen auf großen Datenmengen zu verbessern
- → Konzepte: wie Indizes organisiert werden!

Kriterien

Clustering: (Gruppierung von Daten)
- data that are likely to often processed together will be stored physically close to each other
  - → **Tuple clustering:** storing tuples on same physical page (z.B. alle Daten eines Kunden liegen auf 1 Seite)
  - → **Page clustering:** storing related pages close together in secondary storage (z.B. alle Daten eines Monats nacheinander gespeichert)

introduces ordering based on similarity

if I want to add sth. insertion overhead

Dimensionality: (Anzahl genutzter Spalten in Index)
- specify how many attributes (dimensions) of the underlying relation for calculation of the index key can be used → how many dimensions can be indexed
  - → one-dimensional index
  - → multi-dimensional index

↳ does order of indexed values make a query performance difference

Symmetry: (Reihenfolge der Spalten in Index)
- performance is independent of the order of the index attributes (order does not matter)
- symmetrical index structures or asymmetrical → yes
  - ↓ no
  - border values difference on performance

Tuple references: (wie Index auf Daten zeigt)
- type of tuple references within the index structure (direkt: Index zeigt auf Daten | indirekt: Index zeigt auf z.B. speicher, wo Daten drinne sind)

↳ reference via TID or store values directly

dynamic behavior: (Was passiert bei Änderungen)
- effort to update index structure for insert, update delete (Anpassung Index, wenn Daten geändert, gelöscht, eingefügt werden)

↳ how flexible Insert, Update, Delete is supported

→ dynamic (scales flexibly by shrinking and increasing) → static (↳ needs frequent rebuilds)

Classification

- B-Trees
- Bitmap Indexes
- multi-dimensional Index structure

1) B-Trees

=> one dimensional tree structure

clustering: clustered (tuple clustering)

dimensionality: one dimensional

symmetry: asymmetric -> order important

tuple references: references are stored

dynamic behaviour: dynamic (efficient update algorithms)

exam question

what is UB tree

2.) UB-Tree

- uses space-filling curve (z-curve)
  - to generate 1-dim. key
  - key is indexed via B-Tree

clustering: clustered (curve puts data into order)

dimensionality: multidimensional

symmetry: symmetric (no preference of single dimension)

tuple references: references are stored

dynamic behaviour: dynamic (eg. grow dynamically)

2.) Bit-Map Index

inkl. Standard Bitmap, Multicomponent BM, Range Encoded BM, Interval Encoded BM

clustering: not clustered
(scan whole bitmap → similar values are not stored/clustered closed together)

dimensionality: one-dimensional

symmetry: symmetric

tuple references: tuple references

dynamic behaviour: high update effort
(sth. between static and dynamic)

Assignment 2

Ideal Index for data warehouse

clustering:
- page clustering
- enables faster reading of large data blocks through prefetching

dimensionality:
- multidimensional
- eg. Bitmap excels in handling multiple dimensions efficiently and are highly compressible

symmetry:
- symmetric index structures
- ensures that order of columns in query does not impact performance

tuple references:
- direct references
- reduce number of intermediate steps, when accessing data

dynamic behaviour:
- resilient to updates
- BitMap or B-Tree
  - ↳ for frequent updates
- Should dynamically grow

Assignment 3 (not important for exam) !!!

B^+ Tree Index Resource Consumption

- requires storing a key value and corresponding TIDs (tuple IDs)
- Cost formula:

B^+-Tree-cost = (Number of key values x Average TID list size per key) x Resource consumption per TID

Average TID list size: $\frac{\text{Number of tuples}}{\text{Number of key values}}$

Bitmap Index structure

- each key value in a bitmap index has a bit vector representing the presence of each tuple in the dataset
- Cost formula:

Bitmap Cost = (Number of key values x Number of tuples) x Resource consumption per bit

(1/8 byte (as 1 byte = 8 bits))

A) 5.000.000 tuples, 100 key values,
resource consumption for TID: 1 Byte

B^+ Tree Cost

Average TID list size = $\frac{5000.000}{100} = 50.000$

B^+ Tree Cost = $100 \times 50.000 \times 1 = 5,000,000$ bytes

Bitmap Cost

Bitmap Cost = $100 \times 5.000.000 \times 1/8 = 62,500,000$ bytes

B) 5.000.000 tuples, 3 key values, Resource consumption per TID: 1 byte

B^+ Tree Cost

$$\text{Cost} = 3 \times \frac{5000000}{3} \times 1 = 5,000,001 \text{ bytes}$$

Bit Map Cost

$$\text{Cost} = 3 \times 5.000.000 \times 1/8 = 18,750,000 \text{ bytes}$$

C) 5000.000 tuples, 3 keyvalues, Resource consumption per TID: 4 bytes

B^+ Tree Cost

$$\text{Cost} = 3 \times \frac{5000.000}{3} \times 4 = 20,000,004 \text{ bytes}$$

Bitmap Cost

$$\text{Cost} = 3 \times 5.000.000 \times 1/8 = 18,750,000 \text{ bytes}$$

Assignment 4

Transform postcode 39106 to Bitmap index structure

Standard Bitmap Index → one for each value → 100.000 Bitmap Indexes (00000-99999)

ID	city	PLZ	Bitmap-39104	Bitmap-39106
0	c1	39106	0	1
1	c2	39106	0	1
2	c3	39104	1	0
3	c4	39104	1	0
4	c5	39106	0	1

→ for exact match: one Bitmap
→ Range query: [n,m]: m-n Bitmaps

Multi-Component-Bitmap-Index $x = y + z$

↓ each Digit is one component

x	y						z				
M	B-5-1	B-4-1	B-3-1	B-2-1	B-1-1	B-0-1	B-4-0	B-3-0	B-2-0	B-1-0	B-0-0
3	0	0	1	0	0	0	0	0	0	0	1
9	1	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0
6	0	0	1	0	0	0	0	1	0	0	0

→ 50 Bitmaps 50 · 10 (0-9 values)
→ exact match: 5 Bitmaps
→ Range: 24 Bitmaps

↓ eg. 39[106] - 39[100] → two ranges on the digits
→ all numbers are in the range
1+1+2+10+10

3 and 9 are always in range, 391 - 392 → two digits

better storage consumption but complexer than standard BMI

How can Bitmap Indexes be used to support range queries for postal codes?

- selection of relevant range for each position
- less storage consumption
- efficient support of range queries

Range Encoded Bitmap Index

≤ 1	< 2	≤ 3
1	1	1
0	1	1
1	1	1
0	0	1
1	1	1

Bitmap of 2 and not BM-1

BM-2 ⊓ ¬BM-1

99.999 Bitmap Indexes
exact match: 2
Range: 2

Interval Encoded BMI

00000 - 00100 (one Bitmap)
00001 - 00101
00002 - 00102

BMaps: 99.999
Exact Match: 2
Range: 1 → if query window fits range window

Know: How many Bitmap you need for specific query

Assignment 5

Construct a R-Tree

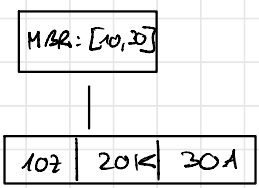
- Insert Data in order of given list
- node has at most 3 entries
- outline insertion process
- Ziel: MBR (Minimum Bounding Rectangles) sollten minimal sein, in Größe u. Überschneidung

R-Tree
- MBR
  - bendoses indexed points taking minimal space
  - → similar to B-Tree
    - balanced tree
    - balancing algorithm
  - → rectangels can overlap ↗ defined by min & max of the MBR diagonal

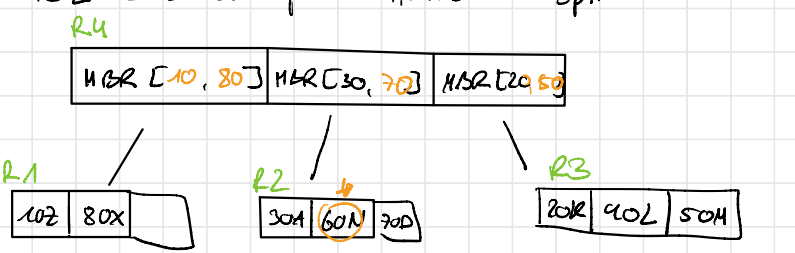
exam relevant

B-Tree

1.) Insert 10Z, 20K u. 30A

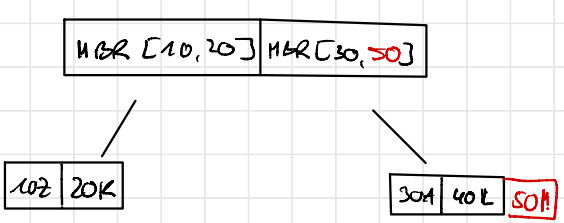


2.) Insert 40L exceed space limit → split

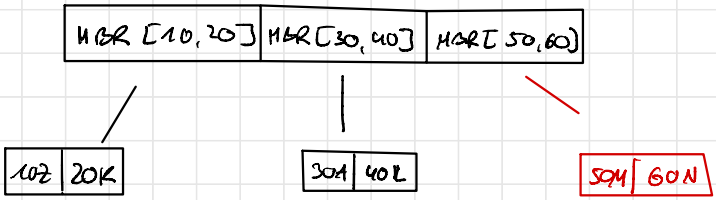


→ don't want to have another 50X thats why we add point to irgendeinem node

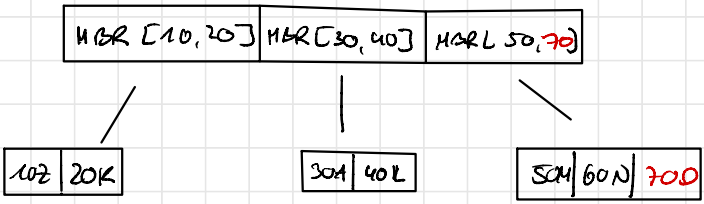
3.) Insert 50M



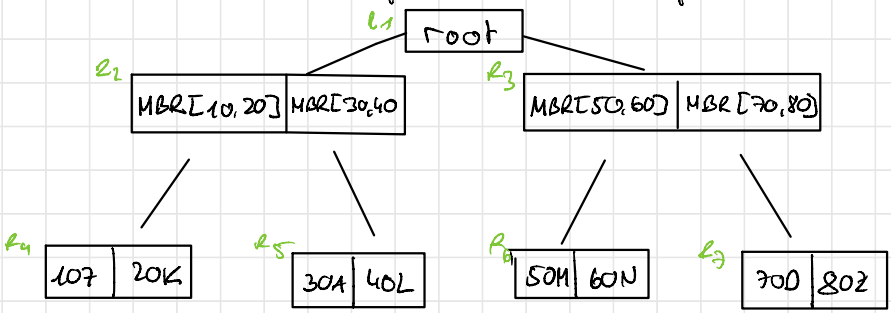
4) Insert 60N exceeds space limit → split

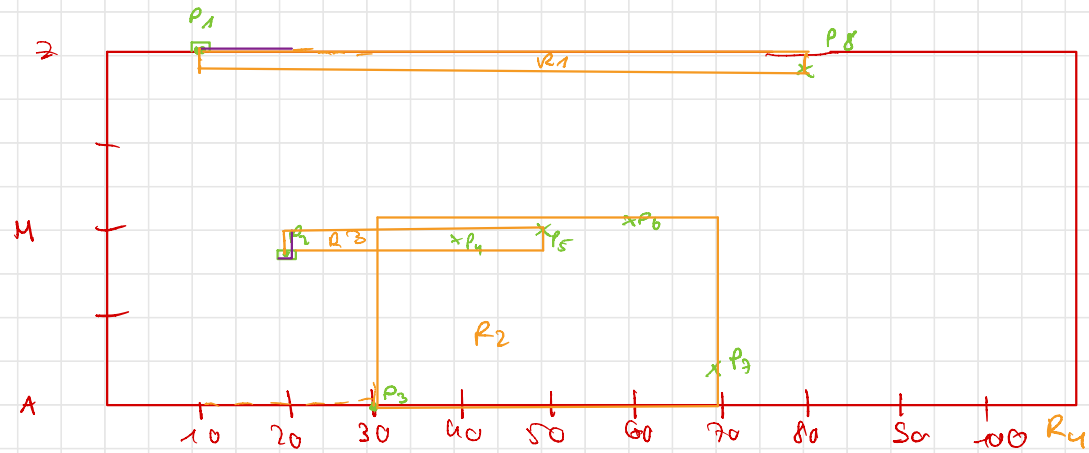


5.) Insert 70D



6) Add 80X exceeds space limit → split




P1
P8
R1
P2
P4
P5
P6
P7
P3
R2
Z
M
A
10
20
30
40
50
60
70
80
90
100
R4

Assignment 6

Gridfile

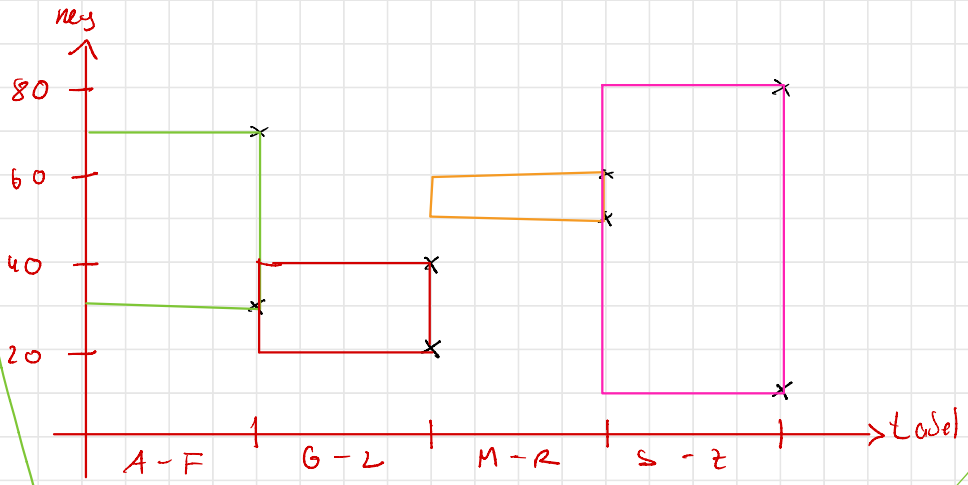
- scales: 1 per dimension
- grid directory: formed by partition of scales
- cells: 1 or more cells form regions
- 1 region = 1 record page

- multidimensional form of data organization (combination of key transformation elements and Index Files)
  - Hash
  - Tree

- data space is divided into a grid of cells
- Each cell corresponds to a range of values
- effektiv for multidimensional data (20, 30)

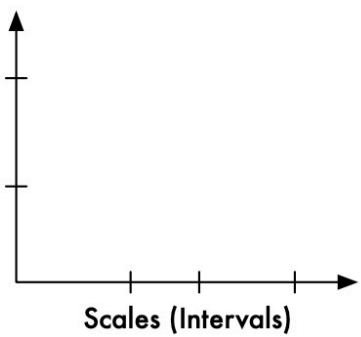
Creation

1.) Define Dimensions
2.) Construct Grid Directory
3.) Allocate Buckets

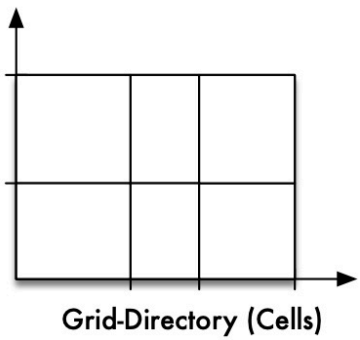


Advantages

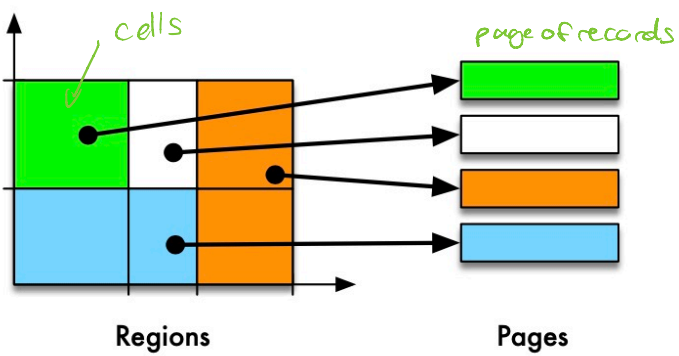
- direct access to data buckets
- efficient for range queries
- dynamic and flexible
  ↳ easily add more grid lines or split buckets
- simple and compact (in comparison to a tree structure)



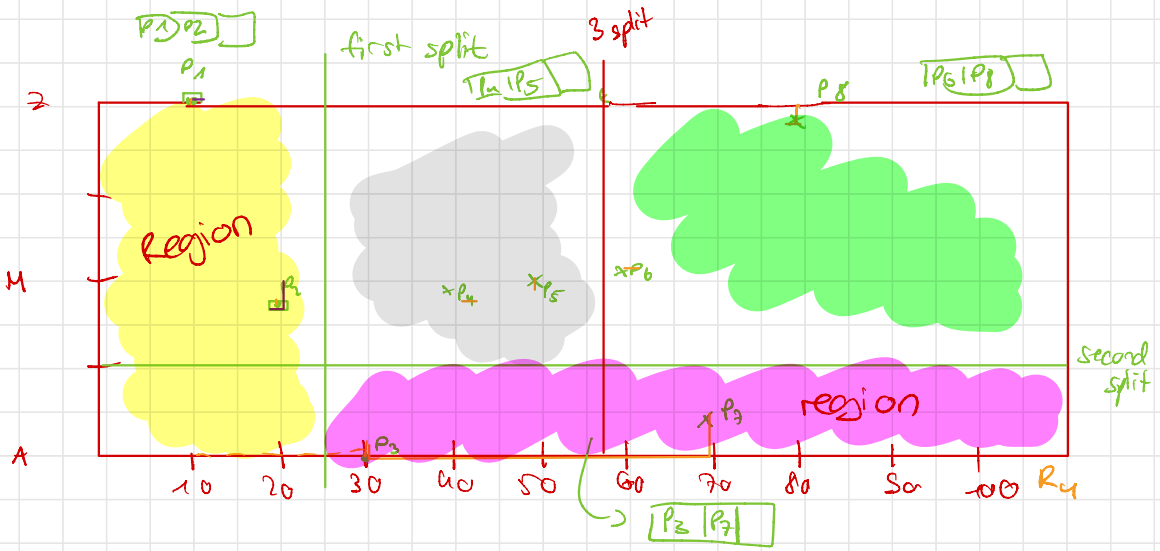
Scales (Intervals)



Grid-Directory (Cells)



Regions Pages



⇒ 3 entries per page

↳ split: start split on y-axis, next split on x-axis → than again → eg. equal number of all pages

* 2. split at y-axis : (~~gl~~ally split setwen P_3, P_4, P_5, P_6)

3. split x-axis (again)

⇒ 6 cells with 4 regions

→ can dynamically grow a tree

→ split depends on heuristic I choose myself (eg. Median value setween overloaded node)