



**Hasso
Plattner
Institut**

IT Systems Engineering | Universität Potsdam

Datenbanksysteme II
Multidimensionale Indizes
(Kapitel 14)

26.11.2009

Felix Naumann

Motivation

2

- Annahme bisher: Eine Dimension
 - Ein einziger Suchschlüssel
- Suchschlüssel kann auch Kombination von Attributen sein
 - Konkatenation verschiedener Werte
 - Immer noch 1-dimensional
- Sequentielle Dateien und B-Bäume nehmen an, dass alle Schlüssel in einer einheitlichen Sortierung vorliegen.
- Hashtabellen nehmen an, dass der gesamte Schlüsselwert vorhanden ist.
- Es gibt Anwendungen, in denen diese Annahmen nicht stimmen.
 - Insbesondere: Besondere Anfragearten
 - Oft zweidimensional

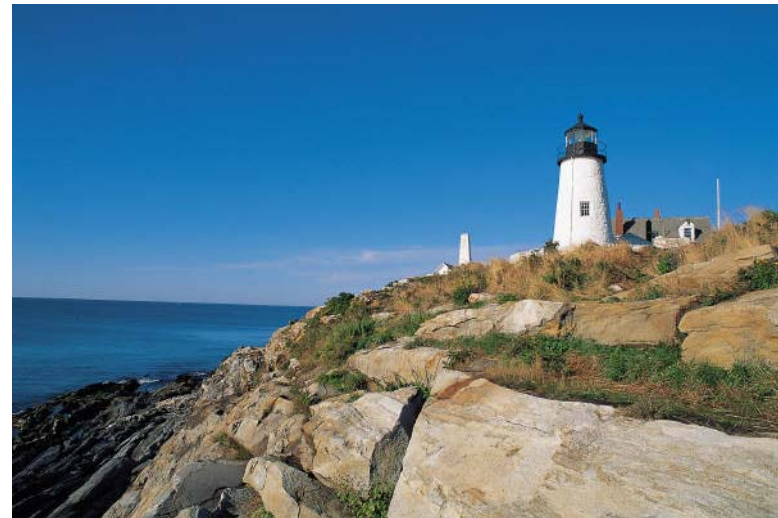
Neue Indexstrukturen

3

- Grid files
 - Erweiterung von Hashtabellen
 - Partitionierte Hashfunktionen
 - Indizes über mehrere Schlüssel
 - kd-Bäume
 - Verallgemeinerung von B-Bäumen auf Punktmengen
 - Quadrantenbaum (quad trees)
 - Mehrwege Baum
 - Jeder Knoten repräsentiert einen Quadranten
 - R-Bäume
 - Verallgemeinerung von B-Bäumen für Regionen
- } Hash-basiert
- } Baum-basiert

4

- Anwendungen für Multidimensionale Indizes
- Hashstrukturen für Multidimensionale Daten
- Baumstrukturen für Multidimensionale Daten



Geographische Informationssysteme (GIS)

5

- Üblicherweise 2 Dimensionen
 - Punkte oder Flächen auf einer Ebene
 - Häuser, Straßen, Brücken, Rohre, ...
- Auch: Schaltkreisentwurf, Windows, ...
- Typische Anfragen
 - Partielle Anfrage: Nur einige der Suchschlüssel werden gebunden
 - Bereichsanfragen: Anfrage spezifiziert Bereich und sucht Punkte darin
 - ◇ Teilmengen, Berührung, Überlappung
 - Nächster-Nachbar-Anfragen (nearest neighbor): Nächster Punkt zu einem gegebenen Punkt
 - ◇ Z.B.: Nächste große Stadt
 - Wo-bin-ich-Anfragen: Gegeben ein Punkt, auf welcher Fläche liegt er
 - ◇ Z.B: Windows-clicks, mouse-over

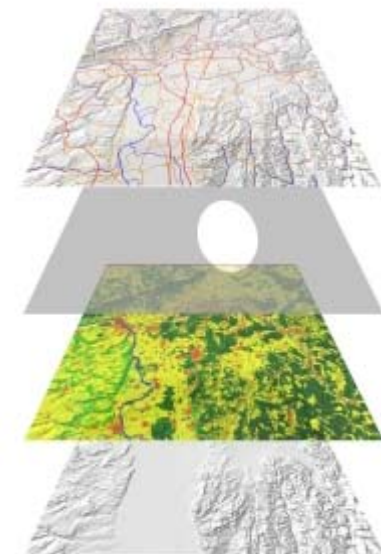


Abb.: WWW Prof. Döllner

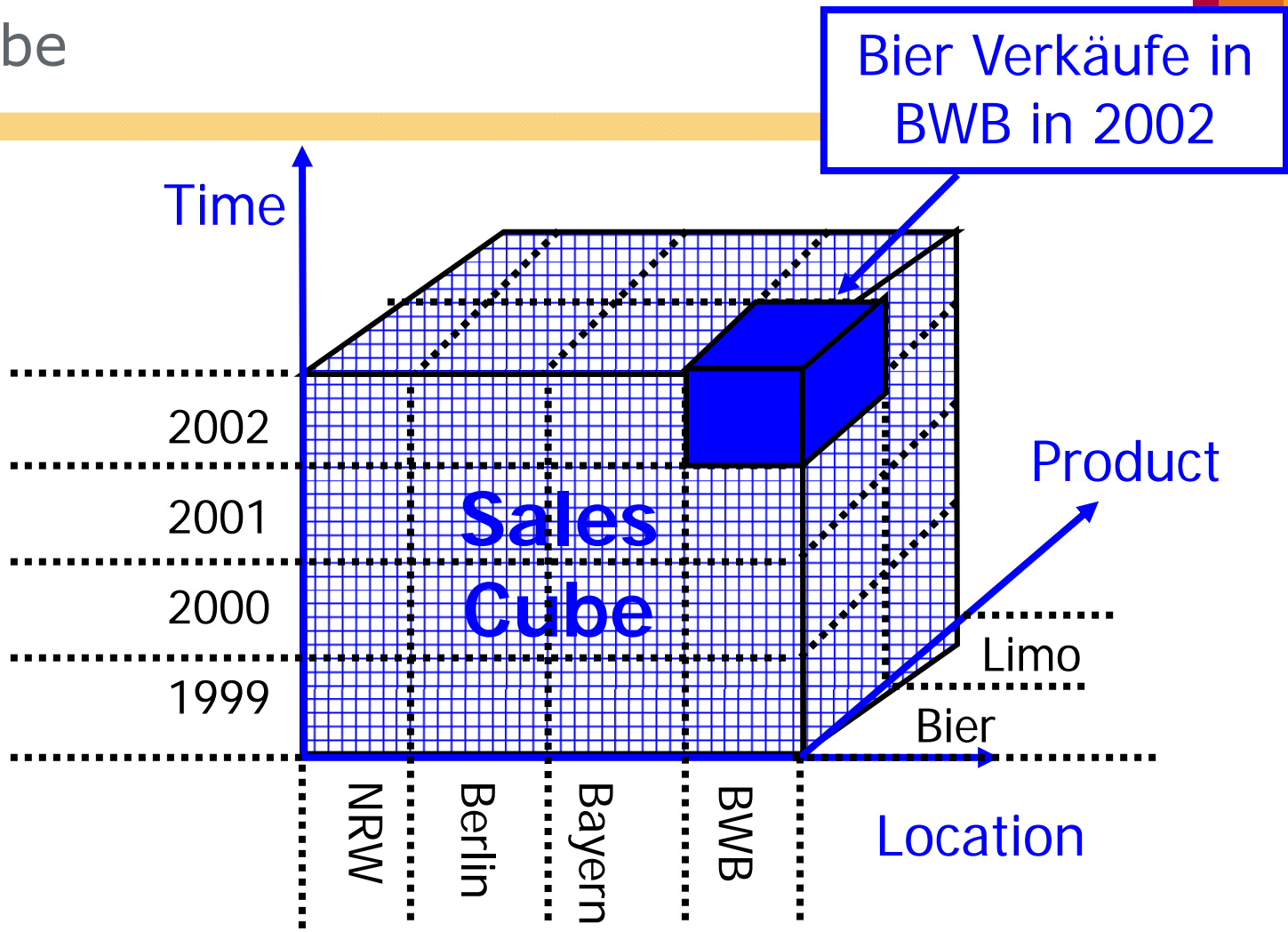
Data cubes (Datenwürfel)

6

- Herkömmliche (Verkaufs-)Daten werden als Daten höherer Dimension betrachtet.
 - Dimensionen einer Transaktion
 - ◇ Zeitlich: Tag und Uhrzeit
 - ◇ Räumlich: Ladengeschäft
 - ◇ Produkt: Kaufgegenstand, Produktgruppe
 - ◇ Farbe
 - ◇ Größe
 - ◇ Kunde: Name, Kundengruppe
- Jedes Attribut entspricht einer Dimension.
- Typische Anfragen schränken einige Dimensionen ein.
 - U.a. durch Aggregation
 - Anzahl Verkäufe roter Hemden für jeden Laden und jeden Monat in 1998.
 - *Decision support*
- Auch: Unterstützung von Data Mining

Cube

7



Cube -> Hypercube: Bon / Lieferant / Kunde / ...

Cube

8

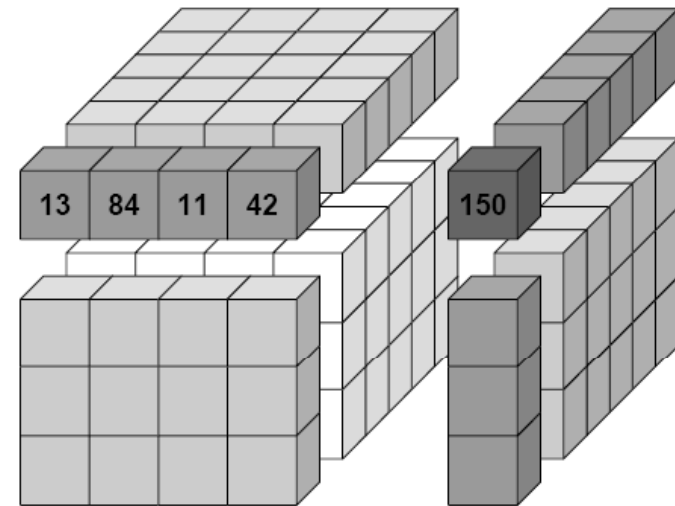
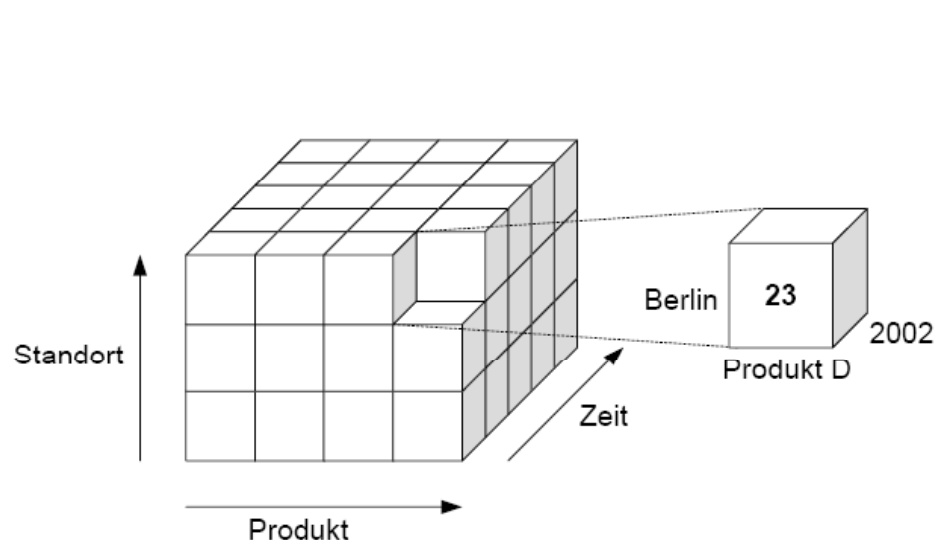


Abb: Seminararbeit Mark Liebetrau

Dimensionen in SQL

9

Mehrdimensionale Anwendungen können mittels SQL vom DBMS unterstützt werden.

- Nearest Neighbor Anfrage

- Relation: Punkte(X, Y)

- Weitere Attribute für die Eigenschaften eines Punkts

- Anfrage: Suche nächsten Punkt zu (10, 20)

- `SELECT * FROM Punkte p`

- `WHERE NOT EXISTS (`

- `SELECT * FROM Punkte q`

- `WHERE (q.X-10)*(q.X-10) + (q.Y-20)*(q.Y-20) <`

- `(p.X-10)*(p.X-10) + (p.Y-20)*(p.Y-20)`

Dimensionen in SQL

10

- Welche Rechtecke umschließen einen gegebenen Punkt?
 - Relation: Rechtecke(id, xll, yll, xur, yur)
 - Suche alle Rechtecke, die den Punkt (10, 20) umschließen.
 - ◇

```
SELECT ID FROM Rechtecke
WHERE xll <= 10 AND yll <= 20
AND xur >= 10 AND yur >=20
```
- Data Cube Anfrage
 - Relation: Verkaeufe(Tag, Laden, Item, Farbe, Groesse)
 - Anzahl verkaufter roter Hemden nach Tag und Laden
 - ◇

```
SELECT Tag, Laden, COUNT(*) AS AnzahlRoteHemden
FROM Verkaeufe
WHERE Item = ,Hemd` AND Farbe = ,rot`
GROUP BY Tag, Laden
```

Bereichsanfragen mit herkömmlichen Indizes

11

- Annahme
 - Zwei Dimensionen: X und Y
- Zwei B-Bäume als Sekundärindizes auf jede Dimension
- Gegeben ein Rechteck
 - = Bereichsanfrage in beiden Dimensionen
- Suche mittels B-Baum alle Pointer auf Datensätze in dem X-Bereich
- Suche mittels B-Baum alle Pointer auf Datensätze in dem Y-Bereich
- Bilde Schnittmenge der Pointer

Bereichsanfragen mit herkömmlichen Indizes – Beispiel

12

- Gegeben
 - X-Y-Region jeweils (0, 1000)
 - 1.000.000 Punkte zufällig darin verteilt
 - 100 Punkte pro Block
 - 200 Schlüssel-Pointer-Paare pro B-Baum Knoten
 - B-Bäume auf beide Dimensionen
- Anfrage: Punkt in einem 100×100 Quadrat um den Punkt (500,500) herum
 - $450 \leq X \leq 550$ und $450 \leq Y \leq 550$
- Vorgehensweise
 - B-Baum findet alle Punkte in dem X-Bereich (ca. 100.000)
 - B-Baum findet alle Punkte in dem Y-Bereich (ca. 100.000)
 - Schnittmenge ergibt ca. 10.000 Pointer.
- Jetzt: Kostenberechnung

Bereichsanfragen mit herkömmlichen Indizes – Beispiel

13

- Annahme: B-Baum-Wurzeln im Hauptspeicher
- Kosten für Index
 - Jeweils 1 innerer Knoten
 - Jeweils alle relevanten Blattknoten für die 100.000 Pointer
 - ◇ Liegen hintereinander
 - ◇ $100.000 = 500 \text{ Blätter} * 200 \text{ Pointer}$
 - = 1002 I/Os
- Kosten für Lesen der 10.000 Datensätze
 - Insgesamt 1.000.000 Datensätze auf 10.000 Blöcken
 - => Praktisch jeder Block muss gelesen werden.
 - => Indizes haben eh kaum genutzt.

Auch allgemein ein Problem bei mehr-dimensionalem Zugriff: Daten können auf der Festplatte nur nach einer Dimension sortiert sein.

Nächste-Nachbarn-Anfragen mit herkömmlichen Indizes

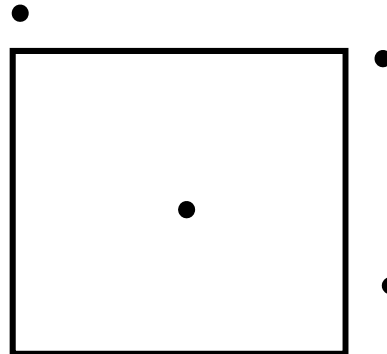
14

- Allgemeine Idee: Verwende Indizes in jeder Dimension zur Wahl eines Bereich um Punkt herum. Wähle nächsten Punkt
- Anfrage: Suche nächsten Punkt zu (10, 20)
 - `SELECT * FROM Punkte p`
`WHERE NOT EXISTS (`
`SELECT * FROM Punkte q`
`WHERE (q.X-10)*(q.X-10) + (q.Y-20)*(q.Y-20) <`
`(p.X-10)*(p.X-10) + (p.Y-20)*(p.Y-20)`
 - Zwei B-Bäume auf X und Y
 - Einschränkung auf $10-d$ und $10+d$ und $20-d$ und $20+d$

Nächste-Nachbarn-Anfragen mit herkömmlichen Indizes

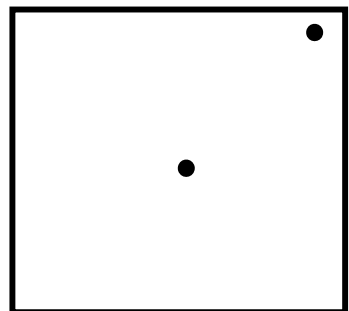
15

- Problem 1: Bereich enthält nicht unbedingt einen Punkt.



- Wiederhole mit höherem d .

- Problem 2: Nächster Punkt im Bereich ist nicht unbedingt insgesamt nächster.



- Falls d' Abstand nächsten Punktes und $d' > d$:
Wiederhole mit d' .

Nächste-Nachbarn-Anfragen mit herkömmlichen Indizes – Kosten

16

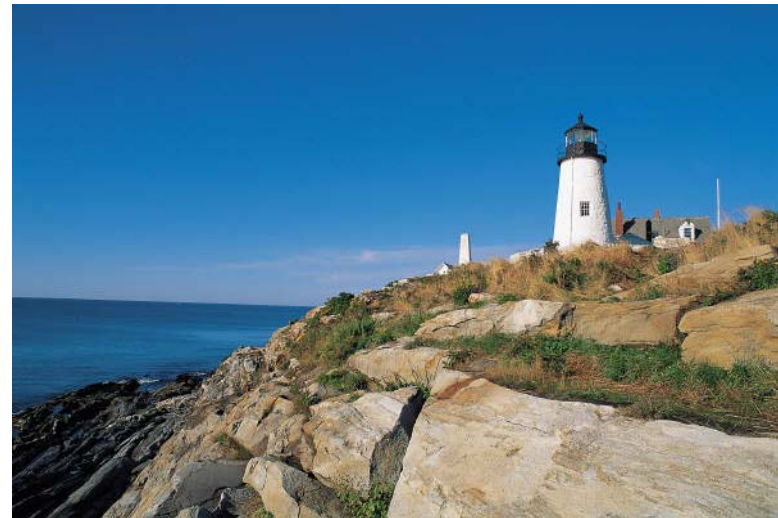
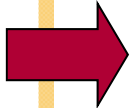
- $P = (10, 20)$; wähle $d = 1$
- Erwartete Punktmenge im 2×2 Quadrat um P : 4
- B-Baum: $9 \leq X \leq 11$ ergibt 2.000 Punkte
 - \Rightarrow Mindestens 10 Blätter, vermutlich 11
 - Zusammen mit innerem Knoten: 12 I/Os
- Analog für Y-Dimension
- +1 I/O zum Lesen des nächsten Punktes
- Zusammen: 25 I/Os
- Falls kein Punkt gefunden oder nächster Punkt weiter als d weg ist: Wiederholung
- Conclusio: Nicht schlecht, aber deutlich langsamer als Punktanfragen oder 1-dimensionale Bereichsanfragen

Überblick über Lösungen

17

- Hashstrukturen – geben auf
 - Antwort zu einer Anfrage ist nicht mehr nur in einem Bucket
- Baumstrukturen – geben mindestens eine Eigenschaft auf
 - Balance des Baumes geht verloren
 - ◇ Tiefer dort wo viele Punkte liegen
 - Korrespondenz zwischen Baumknoten und Diskblöcken geht verloren
 - Updategeschwindigkeit sinkt

- Anwendungen für Multidimensionale Indizes
- Hashstrukturen für Multidimensionale Daten
- Baumstrukturen für Multidimensionale Daten



Grundidee

19

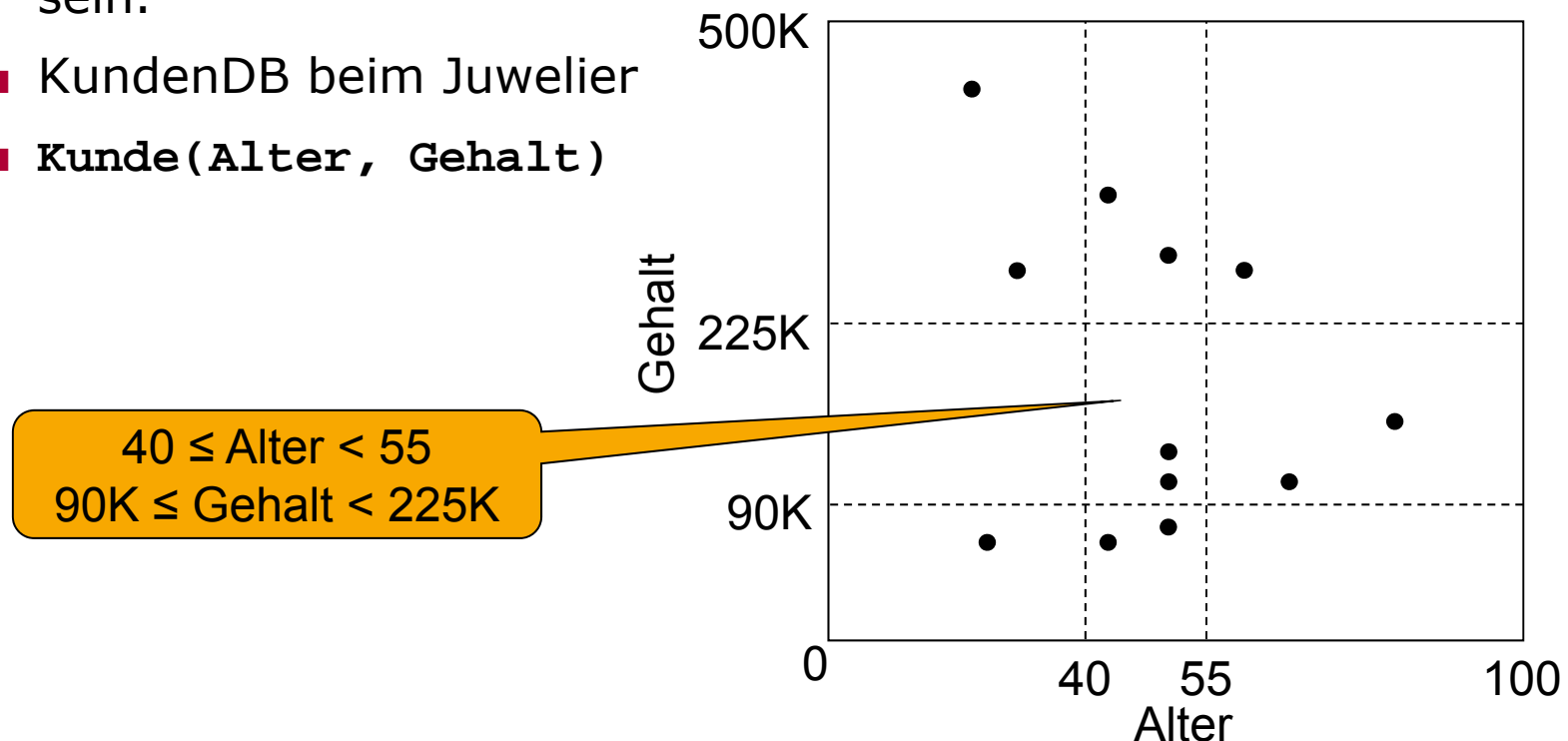
- Verallgemeinerung einer Hashtabelle
- Hashfunktion über Attributwerte aller Dimensionen
- Variante 1: Grid file
 - Sortierung (streng genommen kein Hashing)
 - Partitionierung der Dimensionen
- Variante 2: Partitioniertes Hashing
 - Hash über alle Attributwerte
 - Jede Dimension bestimmt den Bucket mit

Grid files

20

Idee: Raster (grid) über den Raum legen

- Grid teilt jede Dimension in Streifen (stripes).
- Anzahl der Streifen kann für jede Dimension unterschiedlich sein.
- Breite der Streifen kann innerhalb einer Dimension unterschiedlich sein.
- KundenDB beim Juwelier
- `Kunde(Alter, Gehalt)`



Grid file – Suche

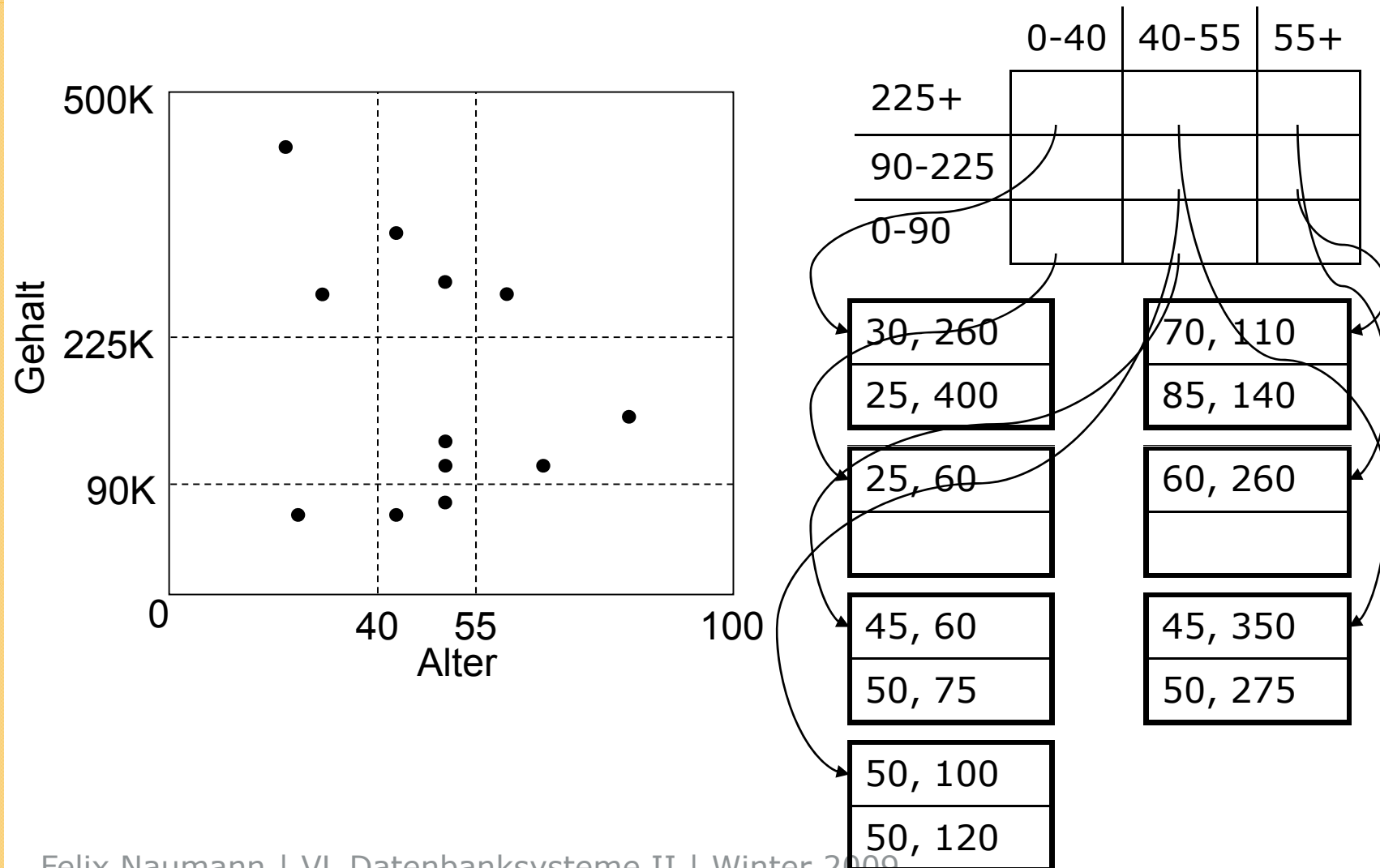
21

Idee: Jede Region entspricht einem Bucket der Hashtabelle.

- Jeder Punkt wird in dem Bucket gespeichert in dessen Region er liegt.
 - Overflowblocks sind erlaubt
- Bucketarray hat mehrere Dimensionen
 - Man speichert die Positionen der Grid-Linien
- Gegeben ein Punkt
 - Betrachte jede Dimension des Punktes separat
 - Bestimme Position im Grid für jede Dimension
 - ◇ Zusammengenommen bestimmen sie den Bucket

Grid file – Suche

22



Zugriff auf Buckets

23

Problem: Viele Buckets; Suche des richtigen Buckets aufwändig

- Binäre Suche
- Indizes für jede Dimension
 - Jeder Index bestimmt einen Streifen

Grid file – Einfügen

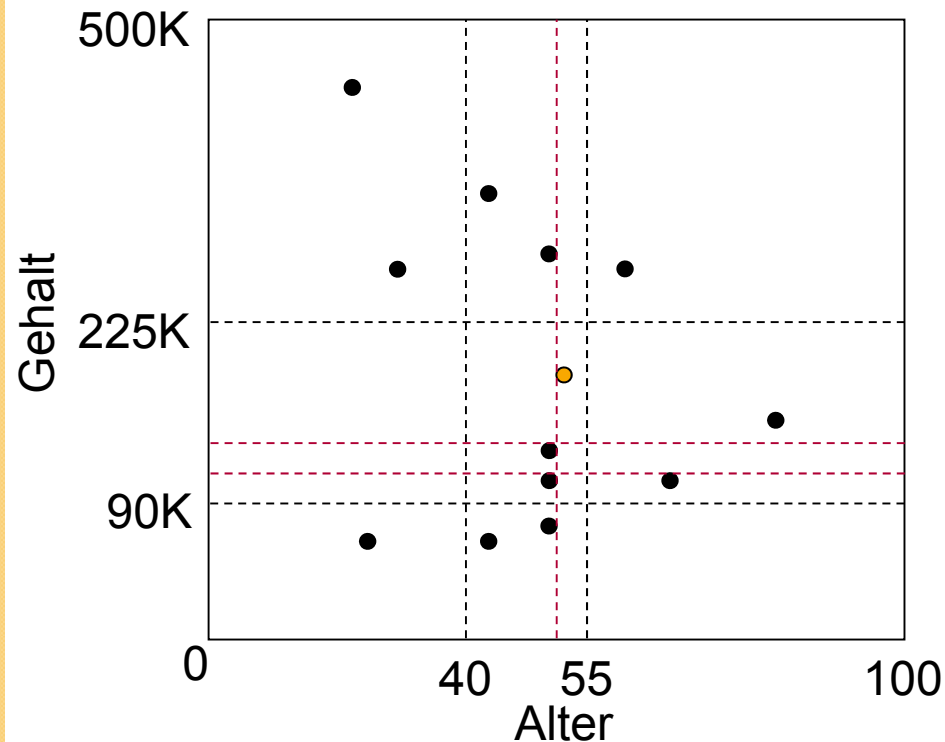
24

- Suche entsprechenden Bucket
- Falls Platz: Einfügen
- Falls kein Platz
 - Variante 1: Overflow Block
 - ◇ Ketten von Overflow Blocks sollten nicht zu lang werden.
 - Variante 2: Grid ändern (ähnlich wie dynamisches Hashing)
 - ◇ Grid-Linien hinzufügen
 - ◇ Grid-Linien bewegen
 - ◇ Problem: Bewegen oder Hinzufügen einer Linie hat Auswirkungen auf alle Buckets entlang der Linie
 - ◇ Problem: Optimale Wahl nicht immer möglich
 - Viele leere Buckets entstehen
 - Buckets bleiben voll

Grid file – Einfügen

25

Füge ein: (52, 200)



- Mittlerer Bucket ist schon voll
- Overflow bucket – langweilig
- 3 Varianten für Grid-Linien
 - Neue Linie: Alter 51
 - Neue Linie: Gehalt 130
 - Neue Linie: Gehalt 100
- Optimum?
 - Variante 1 ungünstig
 - Variante 2 und 3 gleichwertig
 - Variante 2 „mittlerer“

Grid file – Effizienz

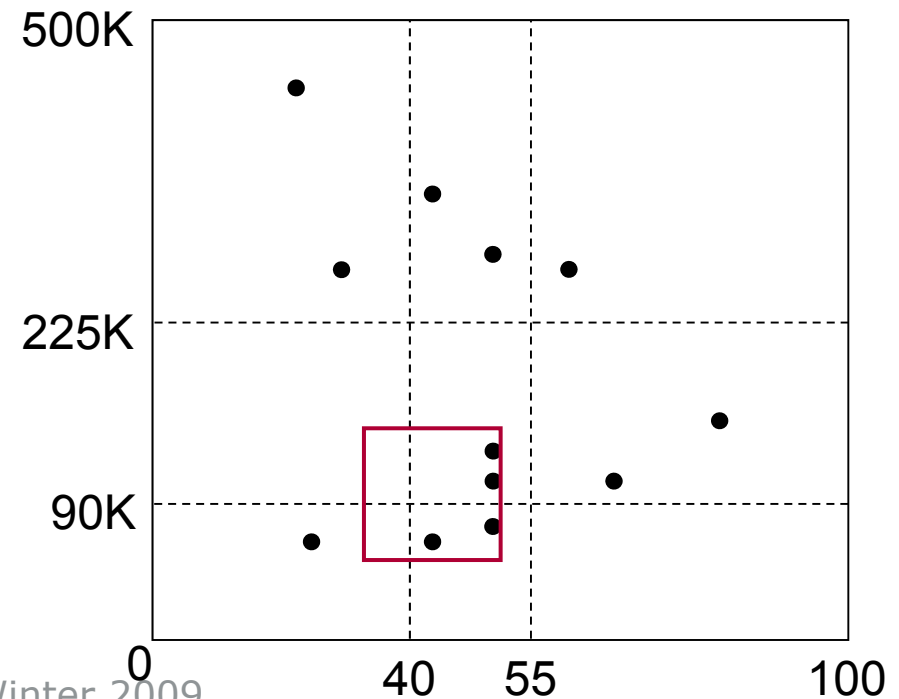
26

- Hohe Dimensionalität: Exponentiell viele Buckets – viele leer.
 - Auch schon bei zwei Dimensionen zu erwarten. Warum z.B.?
 - ◇ Falls Alter und Gehalt korrelieren
- Wahl der Grid-Linien, so dass
 - Bucket Matrix passt in Hauptspeicher
 - Index für jede Dimension passt in Hauptspeicher
 - Nicht zu viele Overflow Blocks
- Punktanfrage (Alter und Gehalt spezifiziert)
 - 1 I/O um Bucket zu lesen
 - Einfügen/Löschen: 1 zusätzlicher I/O
 - Overflow Bucket: Noch 1 zusätzlicher I/O

Grid file – Effizienz

27

- Partielle Anfrage (nur Alter oder nur Gehalt spezifiziert)
 - I/O: Alle Buckets einer Zeile oder einer Spalte der Bucket-Matrix
- Bereichsanfragen: Spezifiziert mehrdimensionalen Ausschnitt
 - Alle Buckets, die mit diesem Bereich überlappen sind relevant
 - $35 \leq \text{Alter} \leq 45$
 - $50 \leq \text{Gehalt} \leq 100$
 - Bei vielen Buckets
 - ◇ Nur wenige am Rand
 - Anzahl I/Os nicht viel mehr als Minimum

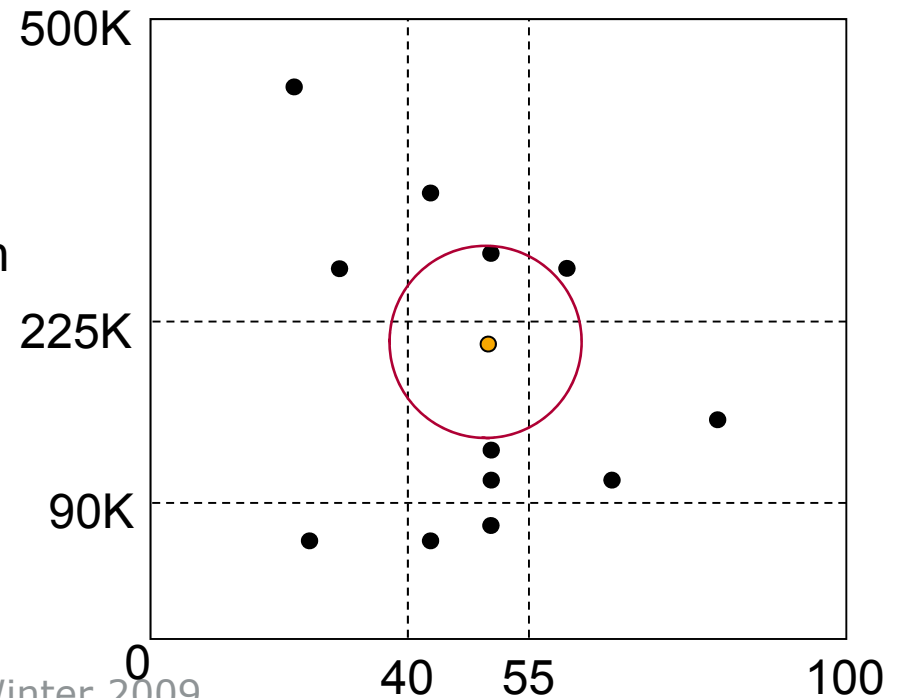


Grid file – Effizienz

28

- Nächste-Nachbar-Anfragen (Gegeben Punkt P)
 - Beginne in Bucket für P
 - Falls $d(P, P') < \text{Ränder des Buckets}$
 - ◇ Suche in allen solchen benachbarten Buckets

- Beispiel: (50, 200)
 - Kandidat (50,120) hat Abstand 80
 - Unterer Rand: Kein Problem
 - Oberer, rechter und linker Rand sind dichter.
 - Suche in allen fünf Buckets



Partitioniertes Hashing

29

- Idee 1: Kombiniere Werte verschiedener Dimensionen und berechne Hashwert

- Person(Name, Alter)

$$h(\text{Name}, \text{Alter}) = \text{MOD} \left(\frac{\text{ASCII}(\text{Name}) + \text{Alter}}{B} \right)$$

- Problem?

- ◇ Nur für Anfragen, die Name und Alter spezifizieren.

- Idee 2: Hashfunktion ist Liste aus einzelnen Hashfunktionen

- Hashwert ist Konkatenation einzelner Hashbitwerte

- Beispiel: 10 Bit (= 1024 Buckets)

- 4 Bits für Name, 6 Bits für Alter

- $h_a(\text{Name}) = 0101$

- $h_b(\text{Alter}) = 111000$

- Hashwert 0101111000

- Gegeben nur Name: Nur 64 Buckets kommen infrage

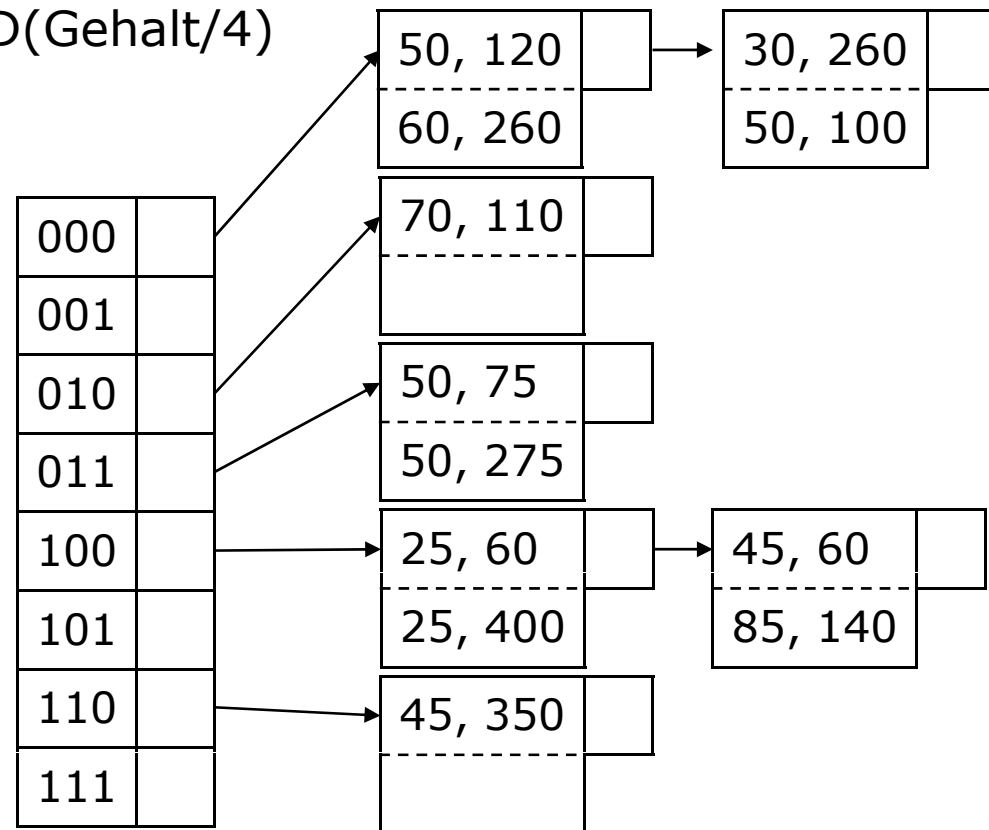
- Gegen nur Alter: Nur 16 Buckets kommen infrage

Partitioniertes Hashing

30

- Beispiel von eben; 3 Bits
 - 1 Bit für Alter: $\text{MOD}(\text{Alter}/2)$
 - 2 Bits für Gehalt: $\text{MOD}(\text{Gehalt}/4)$

- Verteilung nicht toll
- Suche nach Alter=25
 - $12 \times 2 = 24$
 - $1xy$
- Suche nach Gehalt=110
 - $27 \times 4 = 108$
 - $x10$
- Suche nach Alter=85 und Gehalt = 140



Diskussion

31

- Partitioniertes Hashing
 - Nächste-Nachbar-Anfrage?
 - ◇ Physische Nähe wird nicht dargestellt
 - ◇ Lösung: Geeignete Hashfunktion, die kleine Werte auf kleine Hashwerte abbildet.
 - ◇ Hmm...
 - Verteilung besser als in Grid files
 - ◇ Korrelation zwischen Dimensionen egal
 - ◇ Weniger Overflow Blocks