Grundlagen der Datenbanken

Physische Datenorganisation

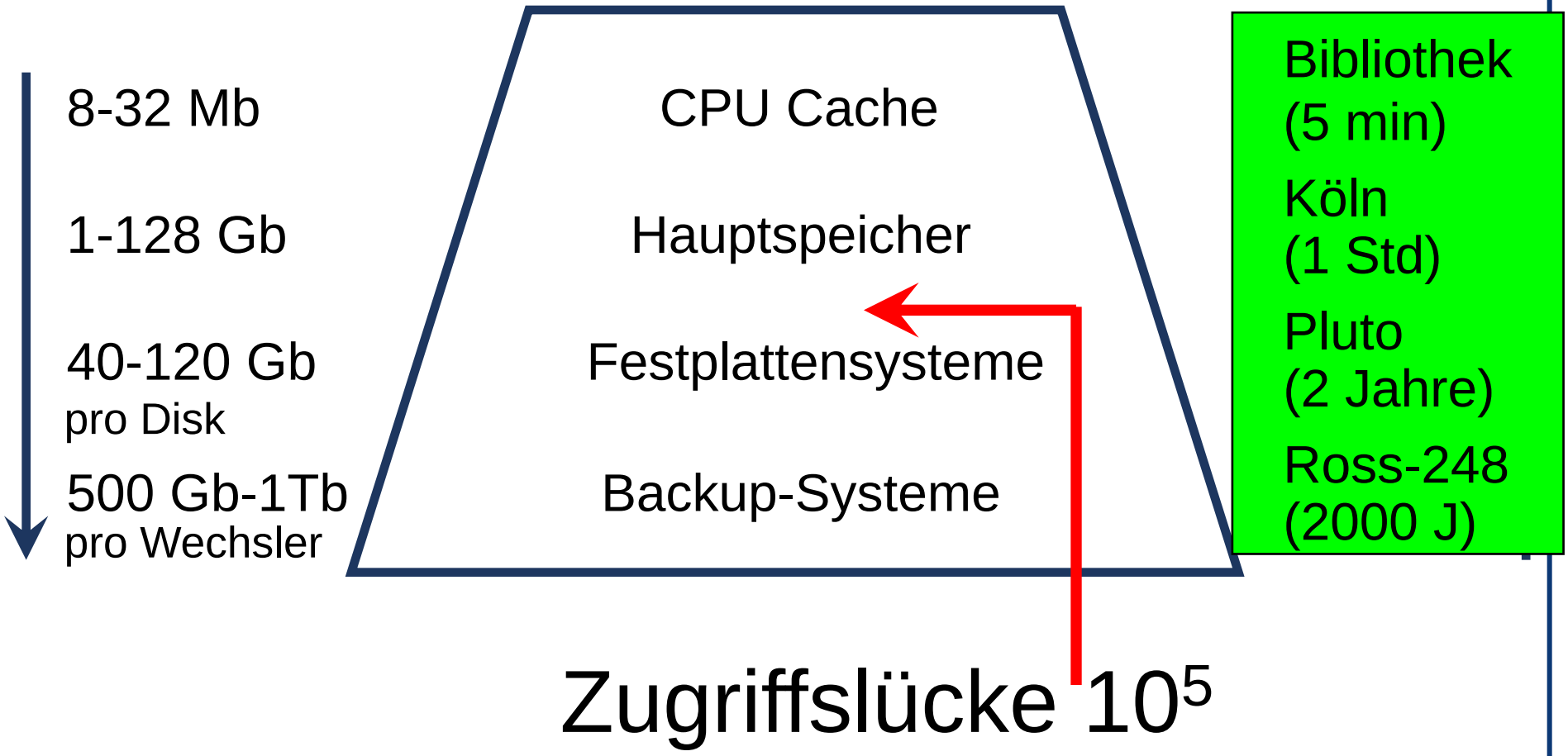
Dr. Thomas Gottron

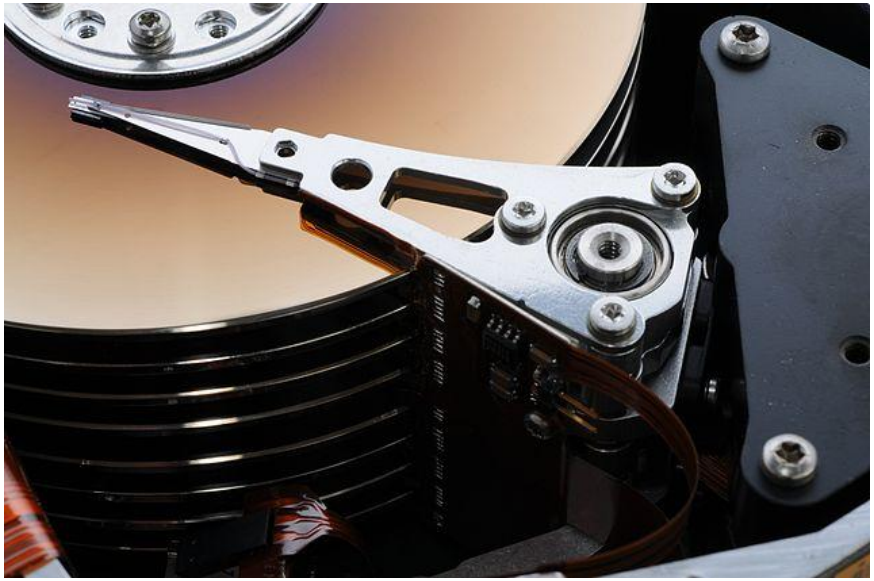
Wintersemester 2012/13

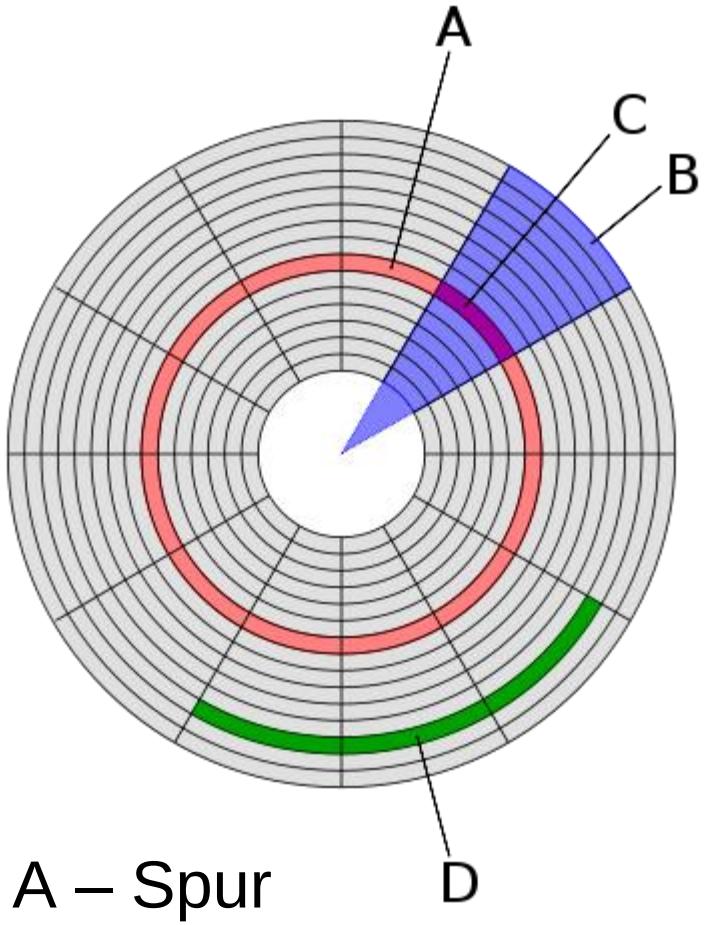
Lernziele

- Speicherhierarchie
- ISAM
- B-Bäume, B^+-Bäume, ...
- Hashing
- Mehrdimensionalen Datenstrukturen (R-Baum)

Zugriffshierarchie

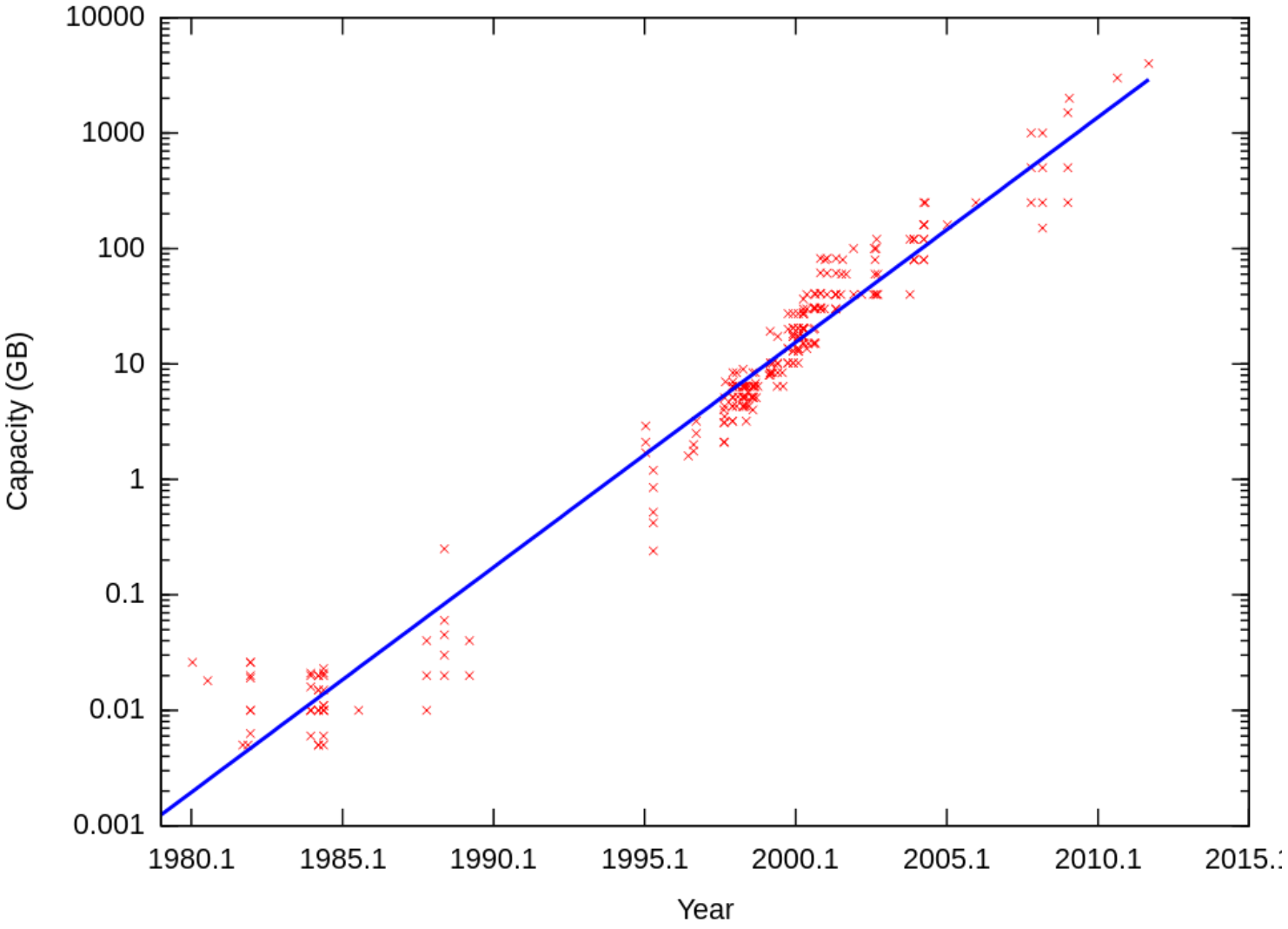

8-32 Mb
CPU Cache
1-128 Gb
Hauptspeicher
40-120 Gb
pro Disk
Festplattensysteme
500 Gb-1Tb
pro Wechsler
Backup-Systeme
Bibliothek
(5 min)
Köln
(1 Std)
Pluto
(2 Jahre)
Ross-248
(2000 J)
Zugriffslücke 10^5





- A – Spur
- B – Sektor
- C – Blöcke eines Sektors
- D – Dateisystem Block

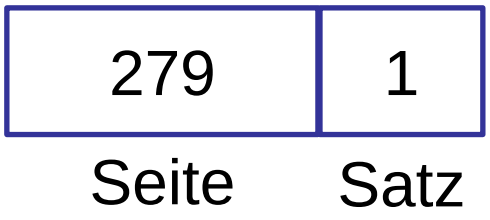
- Kapazität seit den 80er Jahren stetig gestiegen

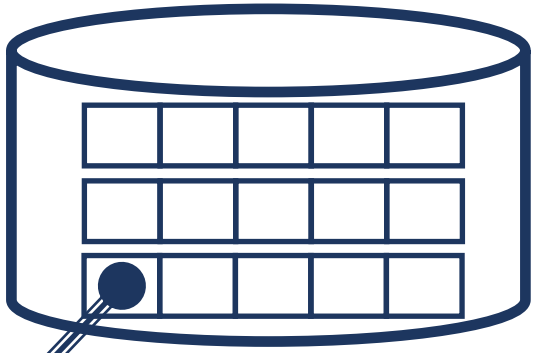


Plattenspeicher: Beobachtungen

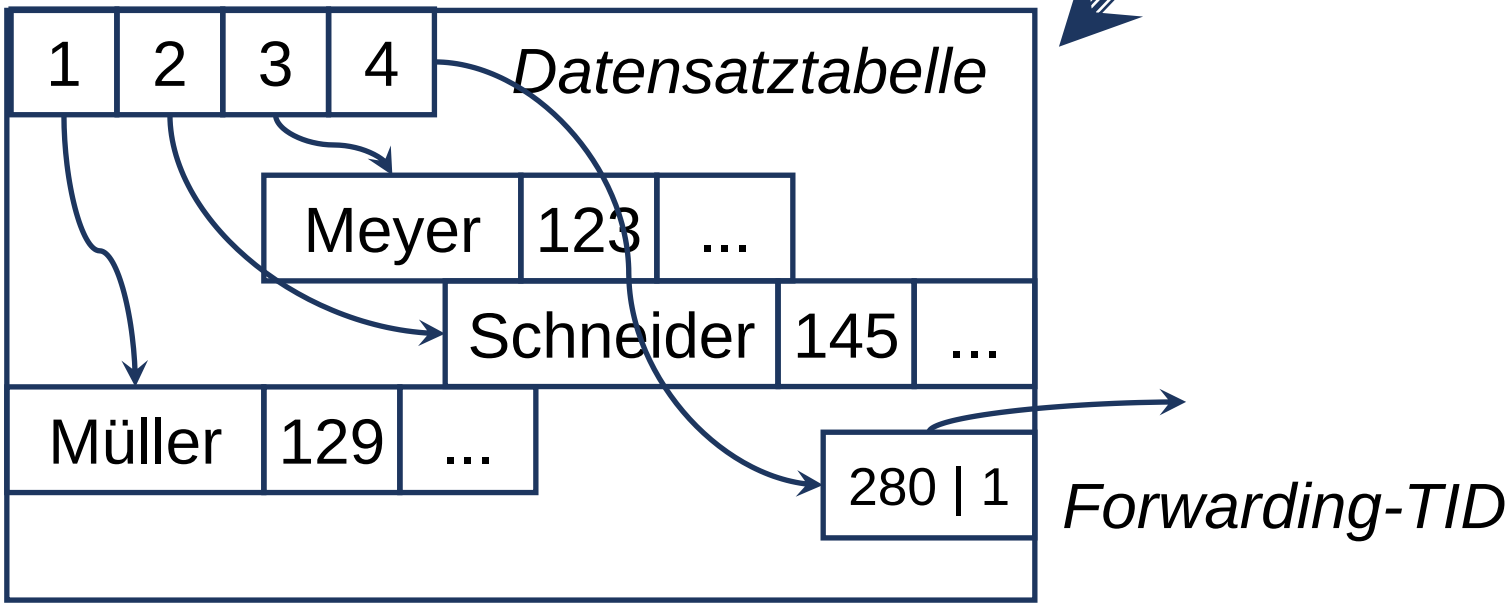
- Grenzen der mechanischen Leistungssteigerung nahezu erreicht (Drehzahlen, Festplattenmechanik, etc.)
- Hybride Modelle (mit Flash-Cache) als Alternativlösung
  - Betriebssystem oder Datenbank muss den Puffer steuern
- Solid-State Disks als Alternativlösung
  - hohe Geschwindigkeit bei Random-IOs,
  - langsames Schreiben von vielen kleinen Datenblöcken wg. größerer Speicherblöcke
- In vielen Fällen bereits heute 'reine' Hauptspeicher-DB einsetzbar (Beispiel: Oracle TimesTen, SAP HANA)
  - persistente Speicherung & Updates bleiben notwendig

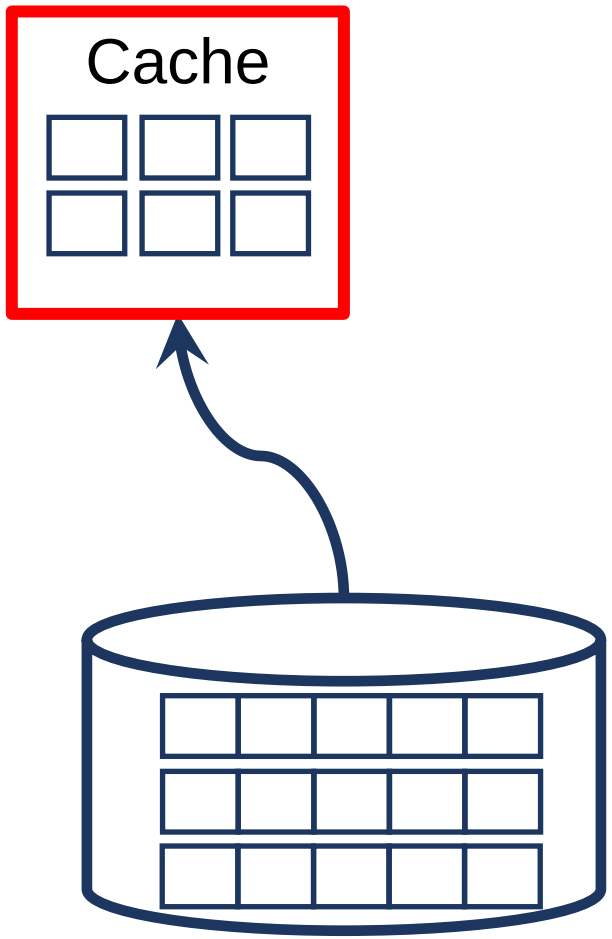
TID : Tupel Identifikator





Datenbank-Seite (32-64 Kb)





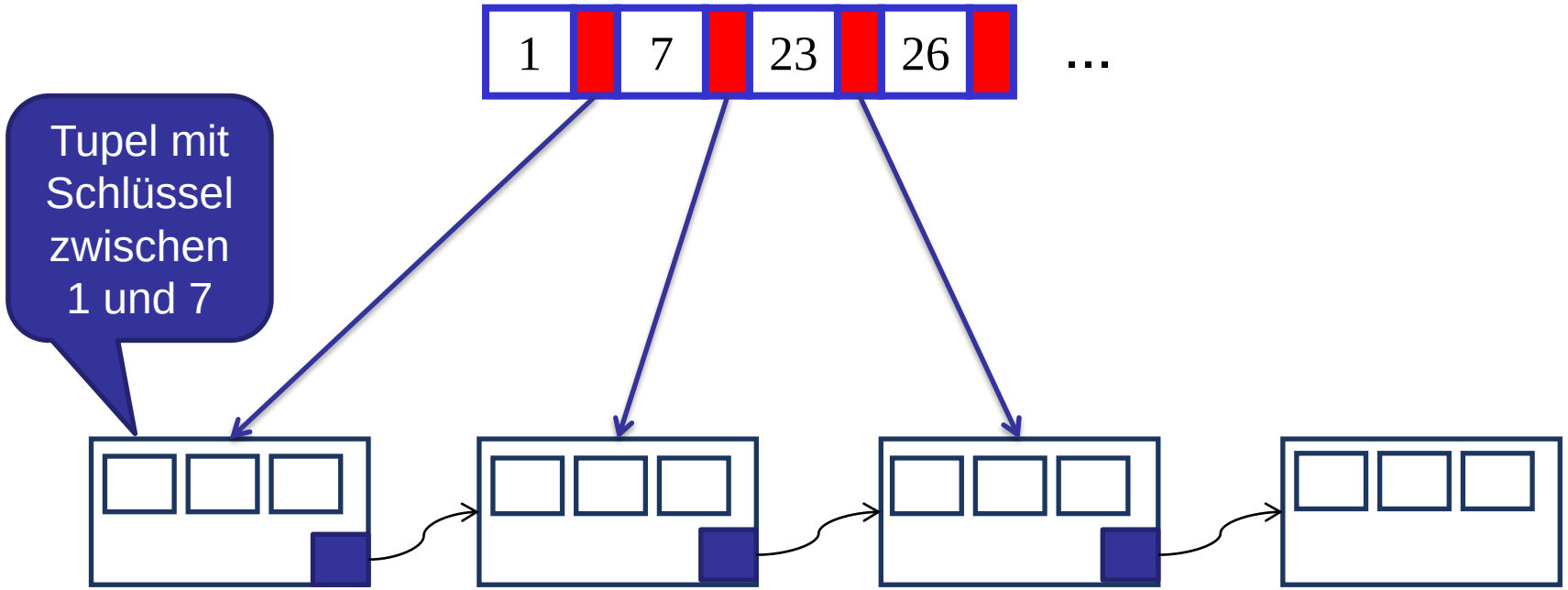
- Laden von Seiten aus dem Hintergrundspeicher zur Verarbeitung
- Daten länger behalten als zwingend notwendig
  - Ersetzungsstrategie
  - Clevere Strategien erreichen 95% Trefferquote
- Arbeiten im Cache und Synchronisation mit Hintergrundspeicher

Index Strukturen

- Schnelles Finden von Tupeln (TID)
- Zusätzlicher Speicherplatz!
- Arten
  - Primärindex
    - Legt physische Anordnung der Daten fest
    - Indiziert meist den Primärschlüssel
  - Sekundärindex
    - Indiziert weitere Attribute

- Nicht indizierte Daten:
  - Sequenzielle Suche über alle (!) Tupel

- Sortierte Liste der Schlüssel (als Seiten)
- Verweise auf die Datenseiten

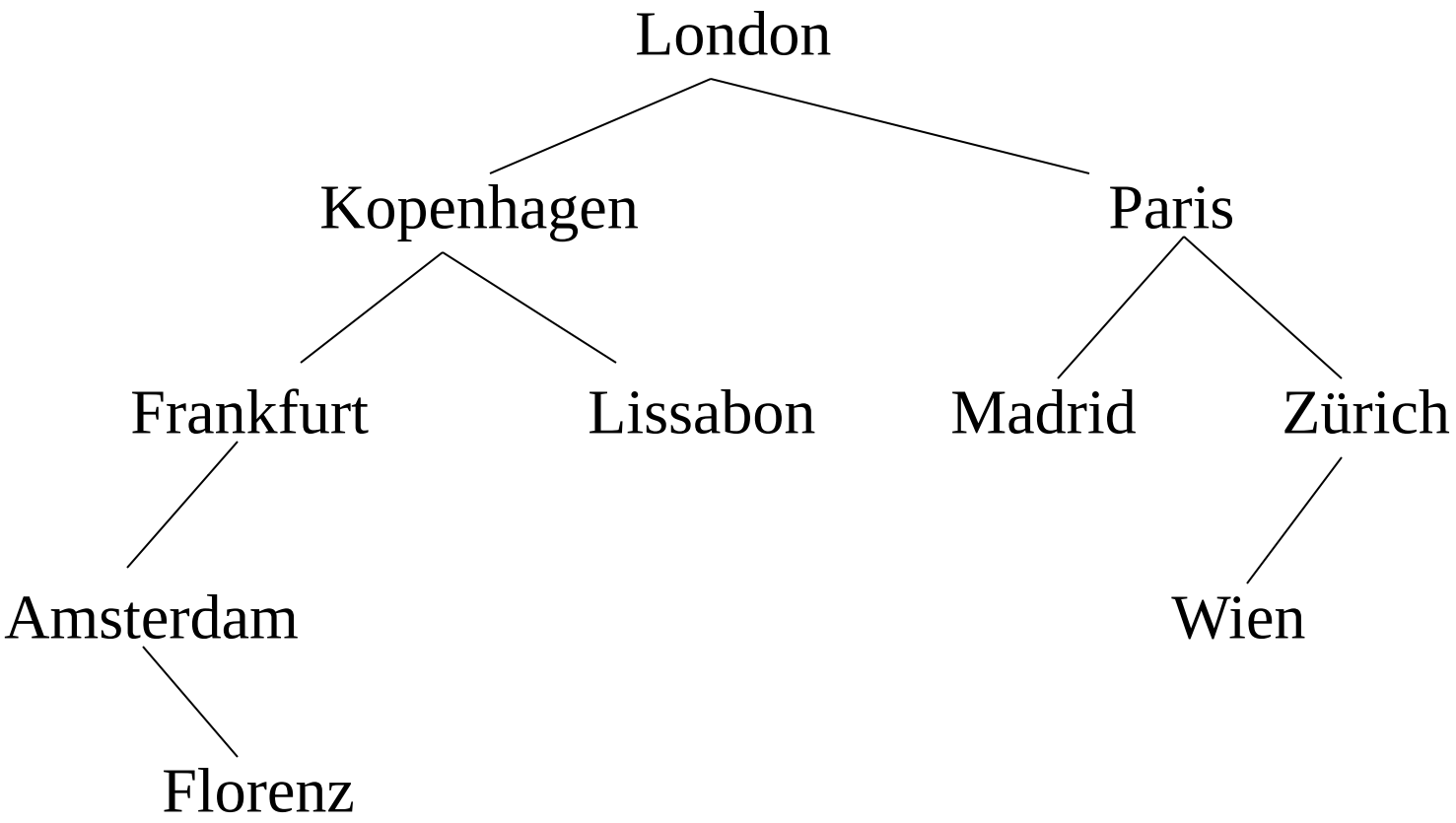


- Suchen
  - Binäre Suche im Index ➔ Datenseite
  - Sequentielle Suche in Datenseite
- Einfügen
  - Datenseite voll
    - Überlauf in Nachbarn und Anpassen der Schlüssel
    - Erweitern der Indexseite (verschieben)
- Löschen
  - Datenseite leer
    - Ausgleich mit Nachbarn (Anpassen der Schlüssel)
    - Löschen im Index (verschieben)

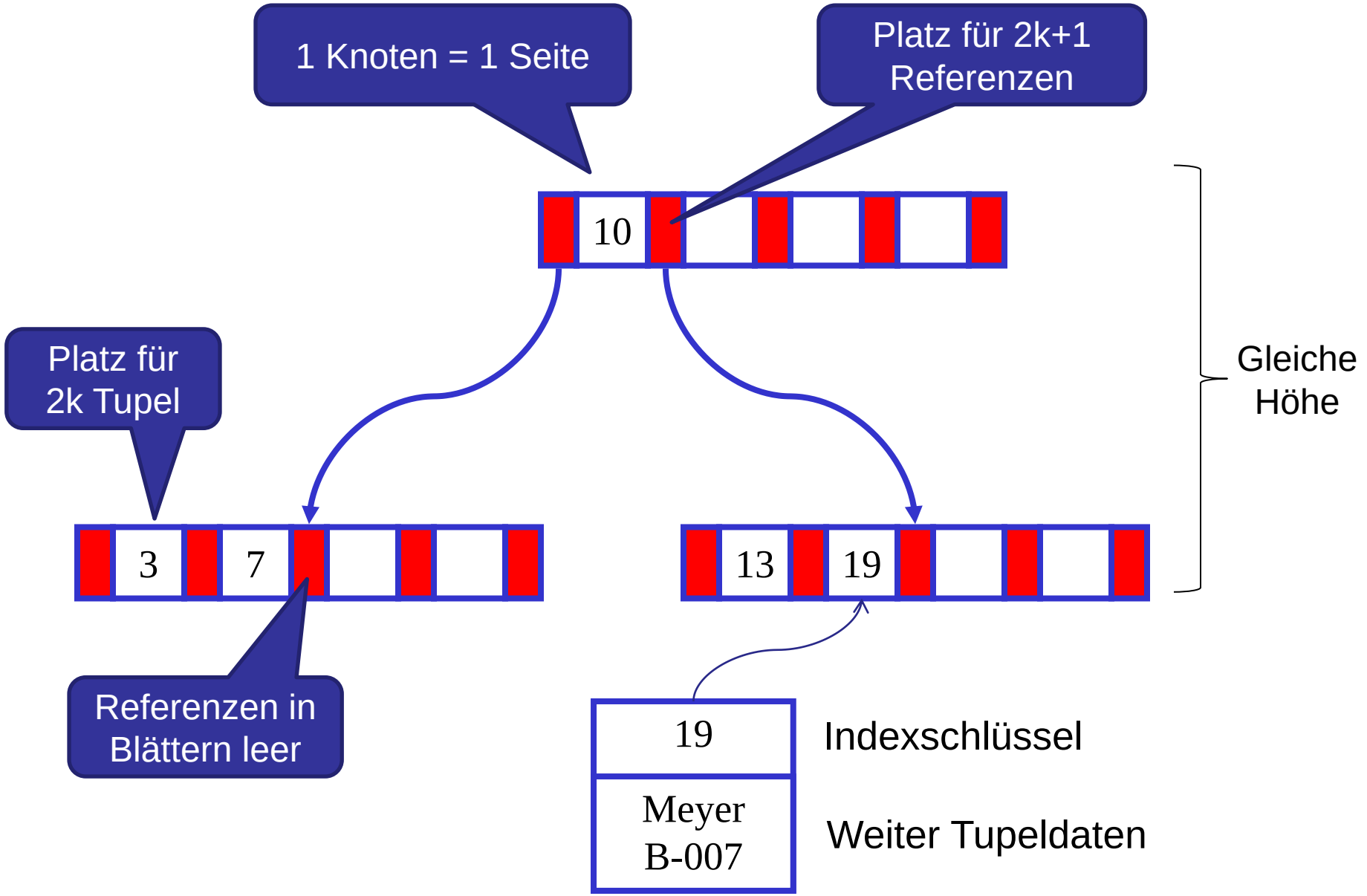
- Anforderungen an Index
  - Garantien für
    - Suchen
    - Einfügen
    - Löschen
  - Ausnutzung der Seitenstruktur
  - Minimierung der Seitenzugriffe
- Suchbaum

B-Baum ≠ binärer Suchbaum

London, Paris, Madrid, Kopenhagen, Lissabon,
Zürich, Frankfurt, Wien, Amsterdam, Florenz



Überblick: B-Baum mit Grad k



Eigenschaften: B-Baum mit Grad k

- Alle Wege von Wurzel zu Blättern gleich lang
- Jeder Knoten (außer Wurzel) mit mindestens k Tupeln belegt (Aulastung ≥ 50%)
- Tupel in einem Knoten sortiert
- Knoten (außer Blätter) mit n Tupeleinträgen hat n+1 Kinder (und somit n+1 Referenzen)
- Sind S_1 bis S_n die Schlüsselinizes in einem Knoten und V_0 bis V_n die Referenzen, dann
  - V_0 verweist auf Knoten mit Einträgen $x < S_1$
  - V_i verweist auf Knoten mit Einträgen $S_i < x < S_{i+1}$
  - V_n verweist auf Knoten mit Einträgen $x > S_n$

Zustand nach jeder Operation garantieren

Einfügen eines Tupels

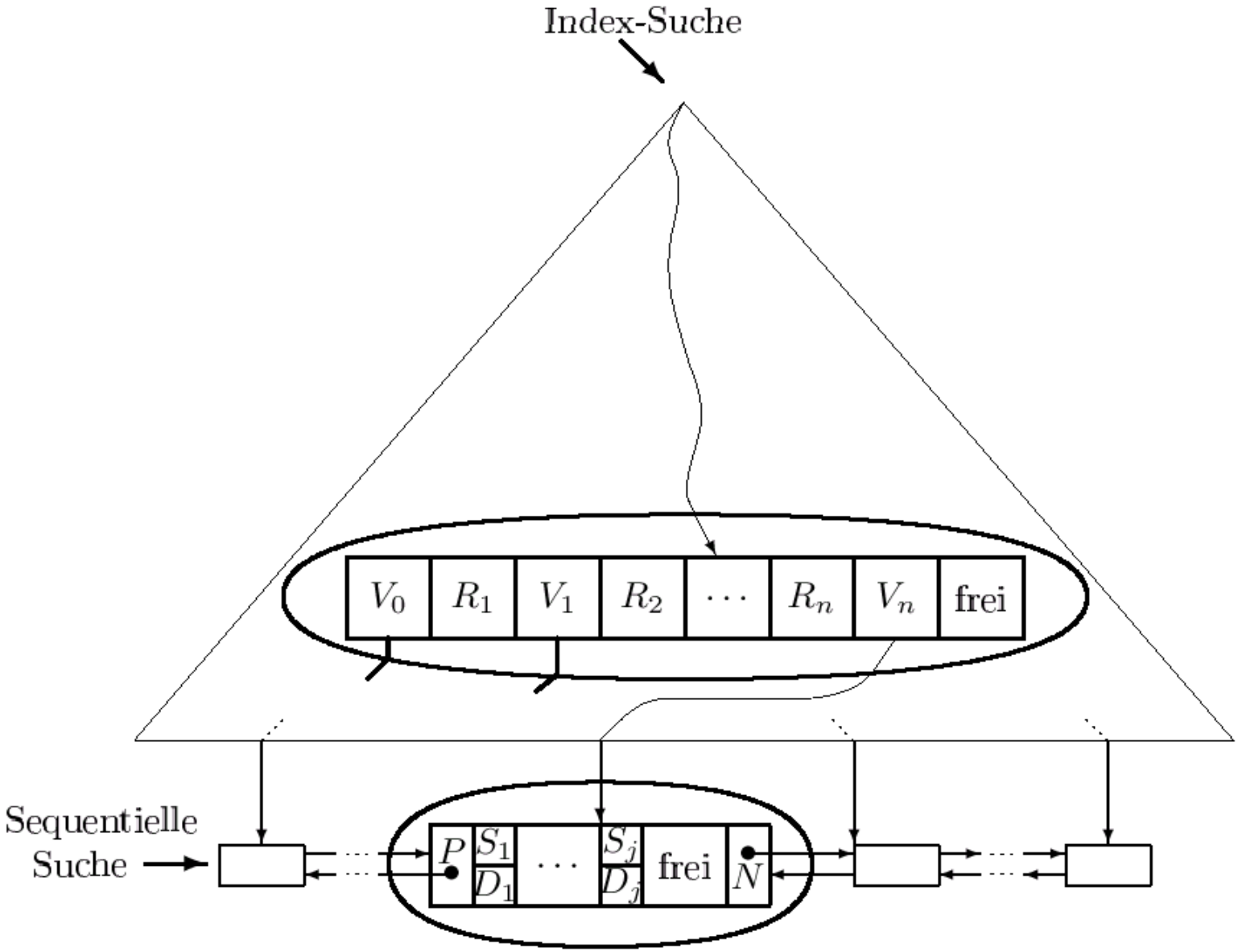
- Relevant ist der zu Wert des Indexattributes: X
- Suche X im B-Baum
  - Endet erfolglos an Stelle an der eingfügt werden muss.
  - Füge Tupel dort ein
  - Wenn Knoten überfüllt (>2k Einträge)
    - Spalte Knoten am mittleren Eintrag auf, schiebe Elemente rechts davon in neuen Knoten
    - Ziehe mittleren Eintrag in Vaterknoten (in der Wurzel: erstelle Vater)
    - Setze Referenz rechts von neuem Eintrag im Vaterknoten auf den neuen Knoten
  - Prüfe rekursiv ob Vaterknoten überfüllt

- Suche im B-Baum
  - Lösche Tupel
    - Falls innerer Knoten: Fülle Lücke mit Tupel as rechtem oder linkem Kind („hochziehen“)
  - Wenn Blattknoten unterfüllt (<k Einträge)
    - Versuche Ausgleich mit Nachbarn mit mindestens k+1 Einträgen
    - Verschmelze mit Nachbarn mit k Einträgen und ziehe trennenden Eintrag aus Vater nach unten

(externer Foliensatz)

- Nachteil B-Baum:
  - Parameter k hängte von Tupelgröße ab
    - ➔ Baumtiefe hängt von Tupelgröße ab
- B^+-Baum:
  - Keine Daten in den inneren Knoten
  - nur Referenzschlüssel
  - Verkettung der Blattknoten durch Referenzen V_0 und V_n
- Ergebnis: flacherer Baum, der immer bis zu den Blättern durchlaufen werden muss

Aufbau eines B$^+$-Baum



- Die Referenzschlüssel müssen keine „echten“ Werte sein

- Künstliche Werte genügen, sofern Bedingung erfüllt ist:
  - V_i verweist auf Knoten mit Einträgen $S_i < x < S_{i+1}$
  - Anders formuliert:
    - $V_{i-1} < S_i < V_i$

- Lösungsansatz:
  - Verwende Präfix von V_i der größer als V_{i-1} ist

Präfixbaum

- Bäume: $\log_k(n)$ viele Seitenzugriffe
- Hashing:
  - Fast eindeutige Zuordnung von Datum zu Bucket (Behälter)
  - h: S → B
    - S Indexschlüssel
    - B: Nummerierung von n Behältern
    - Zugriff innerhalb von 1-2 Schritten
- Charakteristiken der gesuchten Hash-Funktion
  - Fester vs. flexibler Wertebereich
  - Gute Verteilung über den Wertebereich, auch bei schlechter Verteilung der Datencharakteristiken über den Eingabebereich

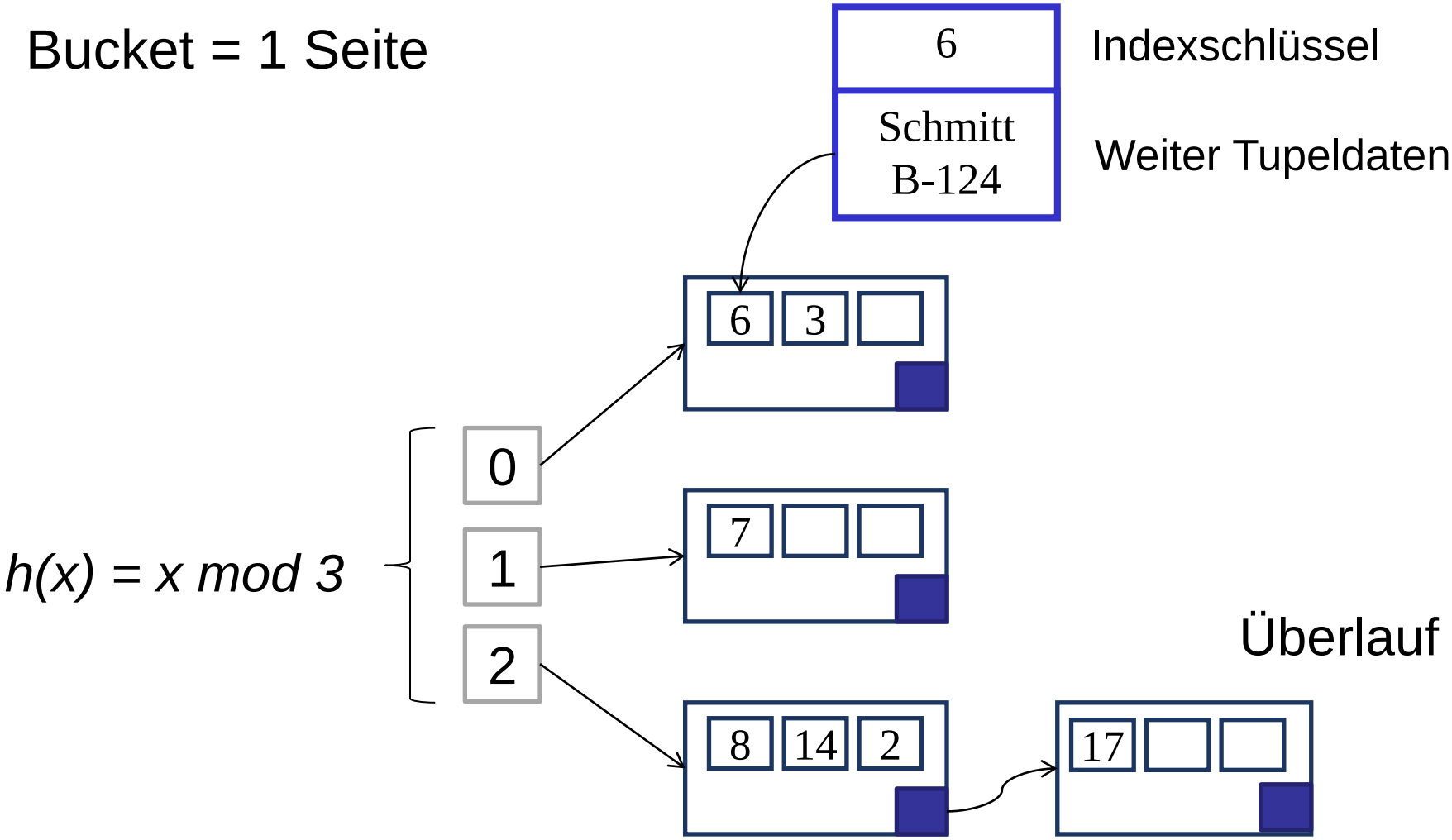
- Abbildung h: D → [0..m-1], genannt **Hash-Funktion**, von Schlüsseln x_1, ..., x_n aus Domain D (z.B. Strings) auf Positionen $h(x_1)$, ..., $h(x_n)$ in Array a[0..m-1], genannt **Hash-Tabelle** (mit n >> m)

  → Speicherung von Schlüssel x_i in $a[h(x_i)]$

- **Anforderungen** an h:
  - sehr effiziente Berechenbarkeit
  - zufällige „Streuung“ (Randomisierung) von x_1, ..., x_n auf [0..m-1]
  - Urbilder von j_1, j_2 in [0..m-1] annähernd gleich groß für alle j_1, j_2 und alle möglichen x_1, ..., x_n
  - für geordnete Schlüssel $x_1 < x_2 < ... < x_n$ sollte die Ordnung von $h(x_1)$, $h(x_2)$, ..., $h(x_n)$ eine zufällige Permutation sein

- **Beispiele** für brauchbare Hash-Funktionen
  - $h(x) = (ax + b) \bmod m$ — für Integers x mit Konst. a, b
  - $h(x) = (\text{mittlere k Ziffern von } x^2) \bmod m$ — für k-stellige Integers x
  - $h(x) = (ord(c_1)+...+ord(c_k)) \bmod m$ — für Strings $c_1c_2...c_k$ in $alph^k$

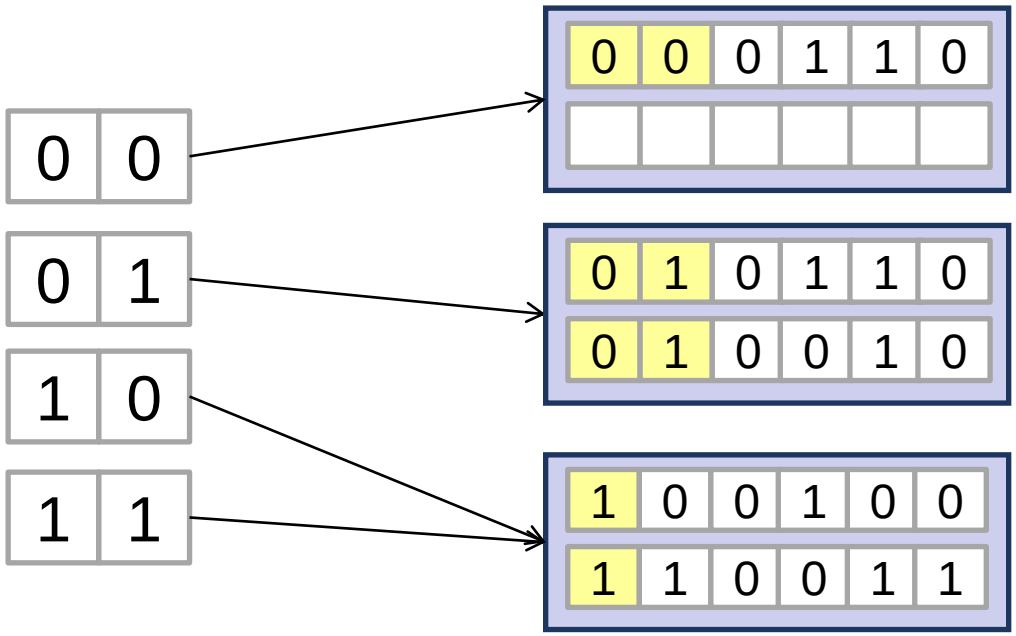
- Bucket = 1 Seite



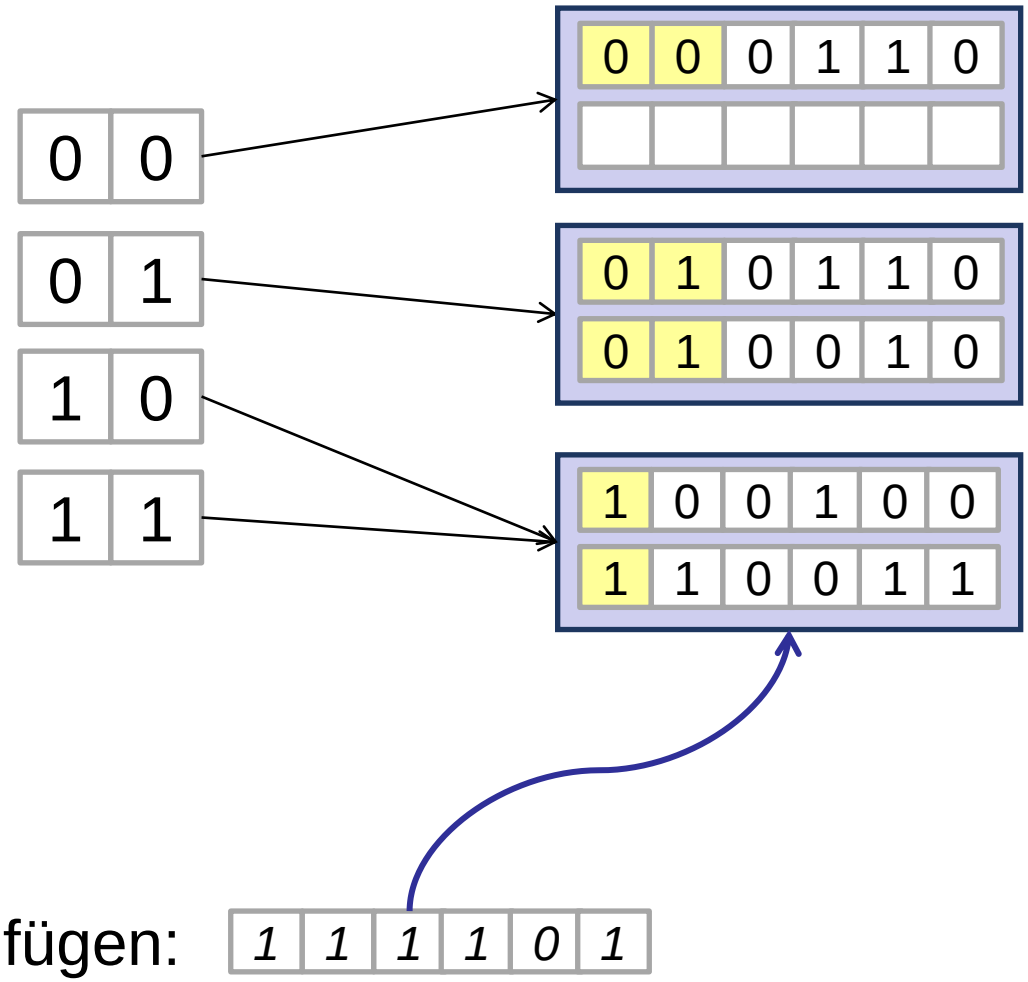
- Statisches Hashing: eines der beiden Probleme
  - Zu viele leere Einträge
  - Zu viel Überlauf

- Erweiterbares Hashing
  - Verwende nur einen Präfix des Hashwertes
  - Aufteilen des Hashwertes: h(x) = d p
    - d : Directory gibt Bucket an (globale Tiefe)
    - p : ungenutzter Teil
  - Sofern möglich deckt eine Seite mehrere Buckets ab (lokale Tiefe)

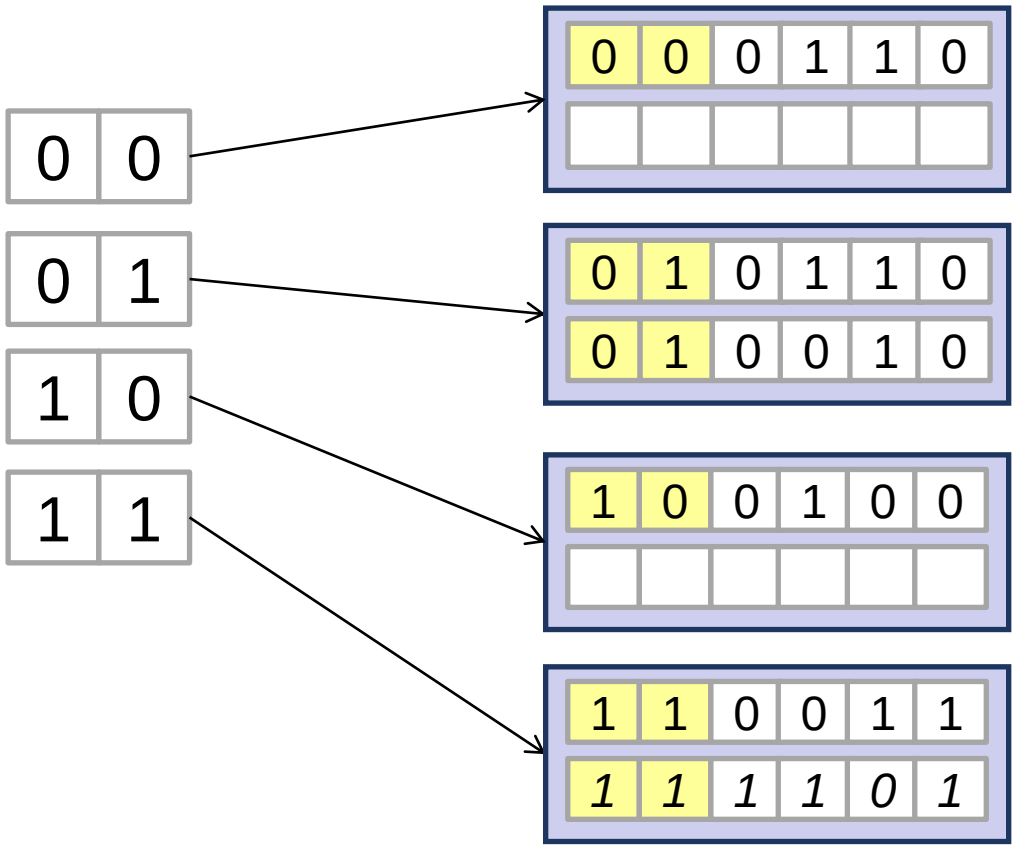
Globale Tiefe: 2



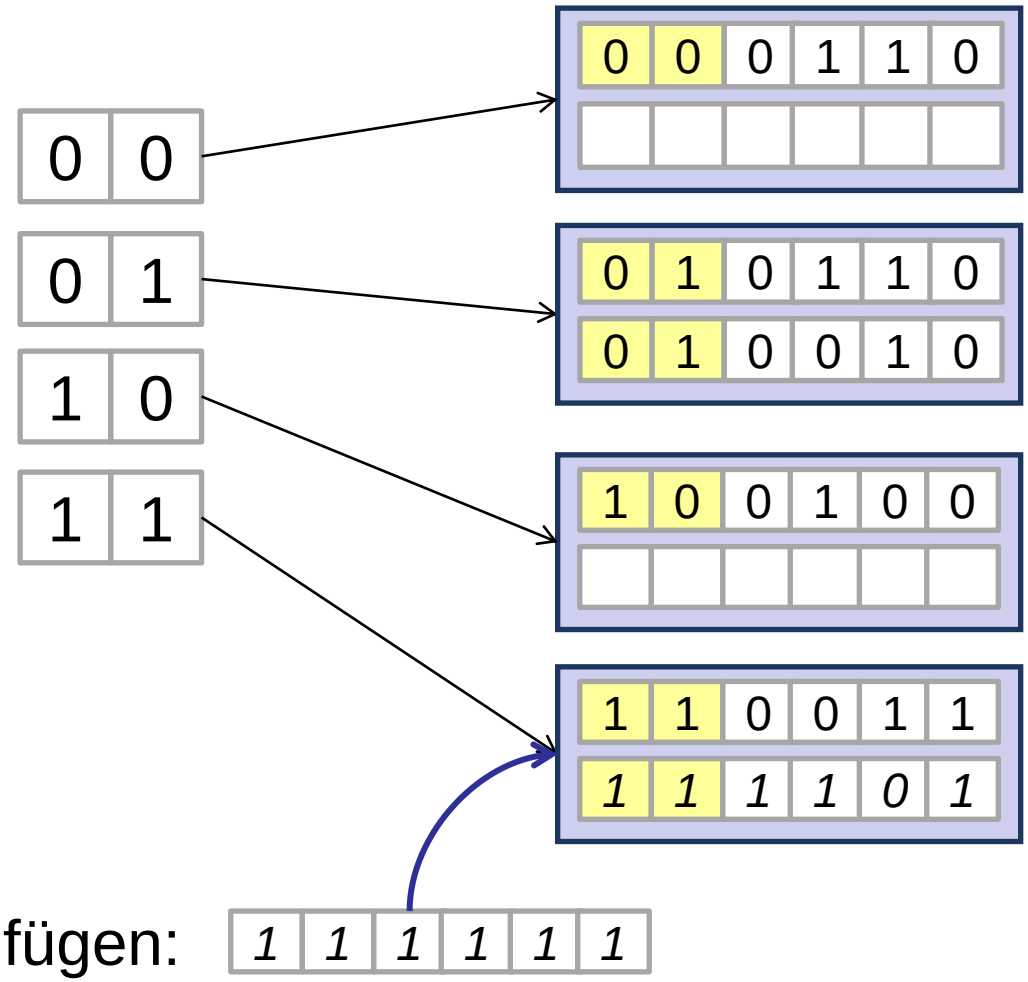
Globale Tiefe: 2



Globale Tiefe: 2



Globale Tiefe: 2



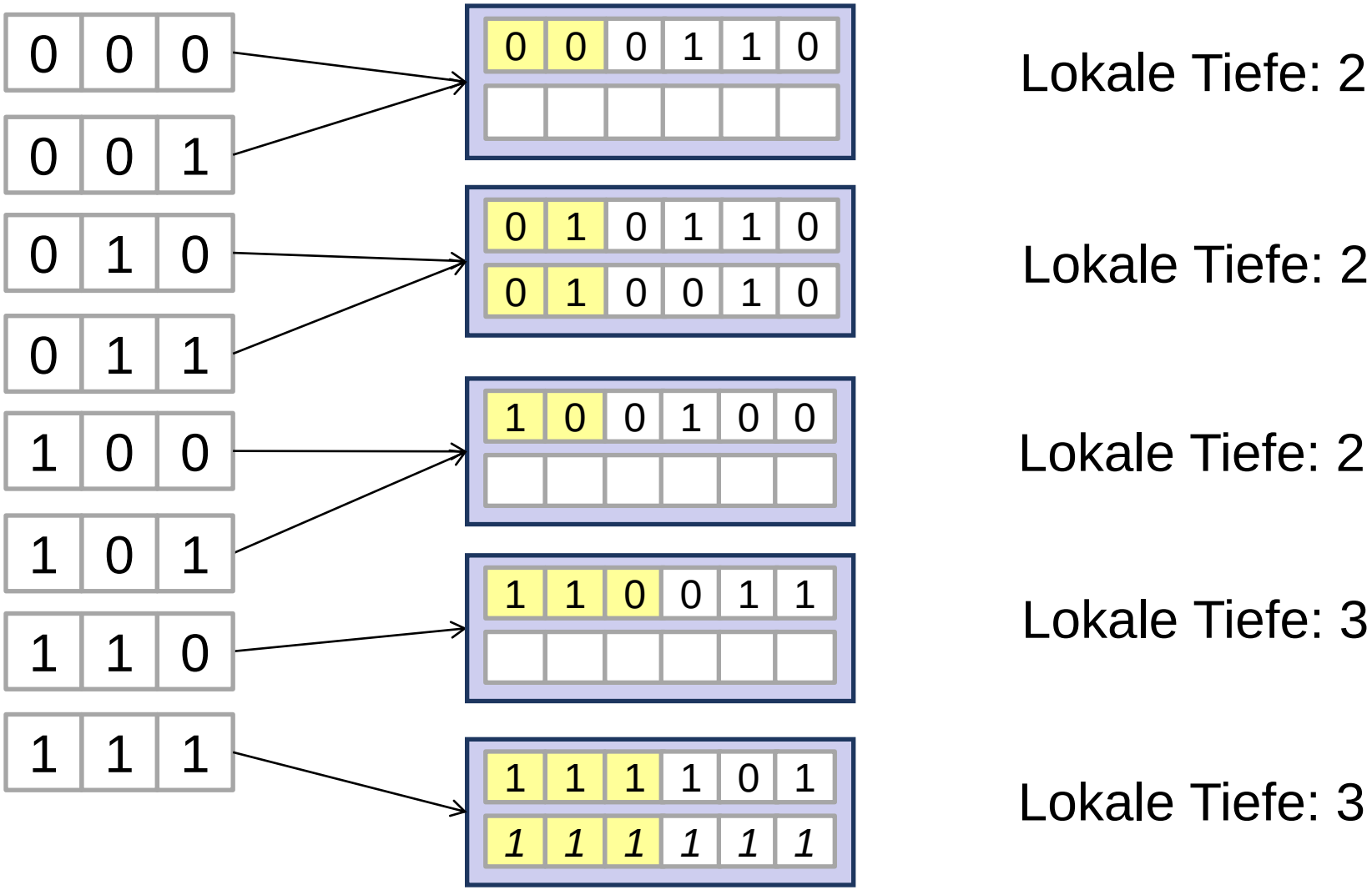
Lokale Tiefe: 2

Lokale Tiefe: 2

Lokale Tiefe: 2

Lokale Tiefe: 2

Globale Tiefe: 3



Ergänzung: Erweitertes Hashing

- Doppelte Hashwerte:
  - Überlauf abfangen (wie bei statischer Hashtabelle)

- Löschen von Werten:
  - Verschmelzen von Seiten
    - Reduktion der lokalen Tiefe
  - Evtl. Reduktion der globalen Tiefe möglich

- Hashing
  - Punktuelle Abfragen:

```
SELECT Name
FROM Student
WHERE MatrNr=2444525;
```

- Baum Indizes
  - „für den allgemeinen Fall“
  - Bereichsabfragen:

```
SELECT MatrNr
FROM Pruefung
WHERE Note>1.0 AND Note<2.0;
```

SQL: Create Index

- Grobsyntax:

```
CREATE [UNIQUE] INDEX Indexname
  ON Tabellenname (Attribut1, Attribut2 ..)

DROP INDEX Indexname
```

Primary Key hat immer einen Index (muss nicht explizit indexiert werden)

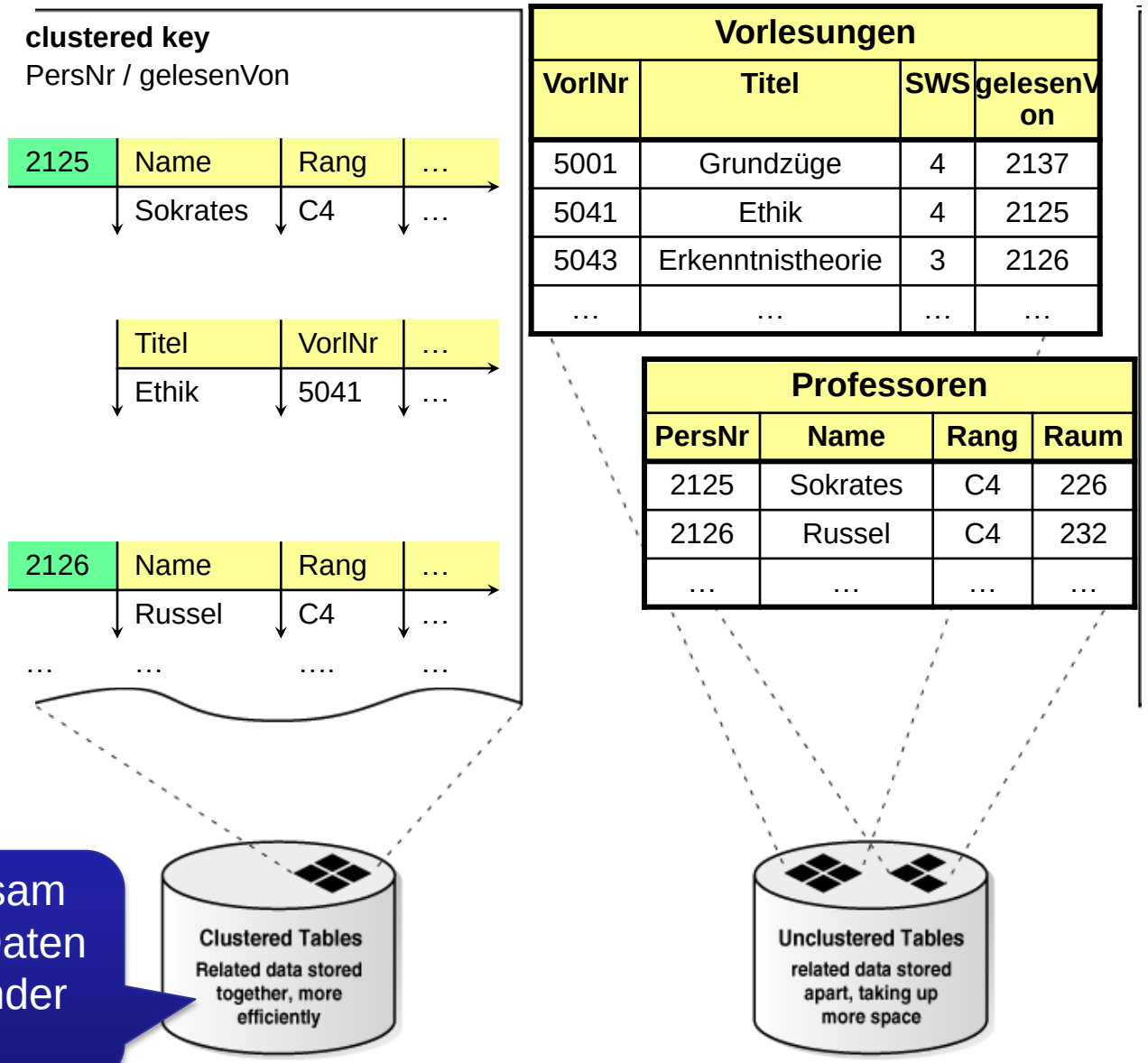
.. Oracle: default-Indextyp ist ein B^+-Baum

- Beispiele:

```
CREATE INDEX Studenten_idx1 ON Studenten(Semester)

DROP INDEX Studenten_idx1
```

Objektballung / Clustering



Oft gemeinsam angefragte Daten dicht beinander ablegen

- Mit einem B+ Baum:

```
CREATE CLUSTER cluster_name ( attribute type, ... );
CREATE INDEX index_name ON cluster_name;
CREATE TABLE table_name ( usual parameters)
CLUSTER cluster_name ( attribute_name );
```

- Mit Hashing:

```
CREATE CLUSTER cluster_name ( attribute type, ... )
[HASH IS hashfunktion] HASHKEYS anzahl;
CREATE TABLE table_name ( usual parameters)
CLUSTER cluster_name ( attribute_name );
```

Mehrdimensionale Indexstrukturen

Abfragen auf mehreren Attribute

- Wertbasierter Zugriff auf der Grundlage mehrerer Attribute,
  - einzeln oder
  - in beliebigen Kombinationen.

- Typische Anforderungen aus CAD, VLSI-Entwurf, Kartographie,...

- Anfragen decken den Bereich ab zwischen
  - mehrdimensionalem Punktzugriff (EMQ) und
  - mehrdimensionalen Bereichsanfragen (RQ)

- Problemstellung:
  - Mehrdimensionale Nachbarschaftsverhältnisse

- Bereichsabfragen:

```
SELECT MatrNr
FROM Pruefung
WHERE Note>1.0 AND Note<2.0
      AND Jahr>2005 AND Jahr<2010;
```

- Mit B-Bäumen:
  - Auswahl der TIDs nach Note
  - Auswahl der TIDs nach Jahr
  - Bildung des Schnitts

    ➔ evtl. deutlich zu viele Daten angefasst

1. Exact Match Query
   spezifiziert Suchwert für jede Dimension D_i
2. Partial Match Query
   spezifiziert Suchwert für einen Teil der Dimensionen
3. Range Query
   spezifiziert ein Suchintervall $[ug_i, og_i]$ für alle Dimensionen
4. Partial Range Query
   spezifiziert ein Suchintervall für einen Teil der Dimensionen

- Wertebereiche D_0,..., D_{k-1}:

  alle D_i sind endlich, linear geordnet und besitzen kleinstes ($-\infty_i$) und größtes (∞_i) Element

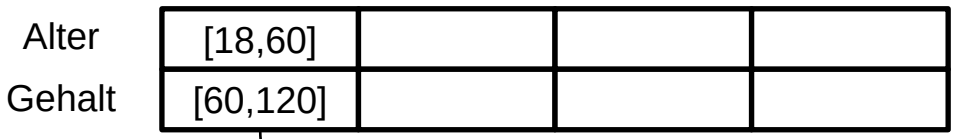
- Datenraum $\boldsymbol{D} = D_0$ x... x D_{k-1}

- k-dimensionaler Schlüssel entspricht Punkt im Datenraum $p \in \boldsymbol{D}$

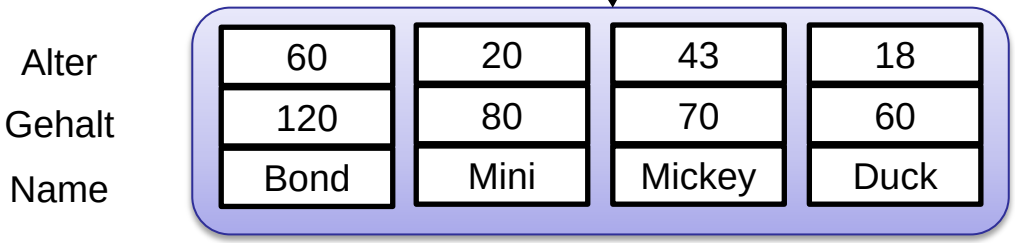
- Klassiker: **R-Baum** (Guttman, 1984)

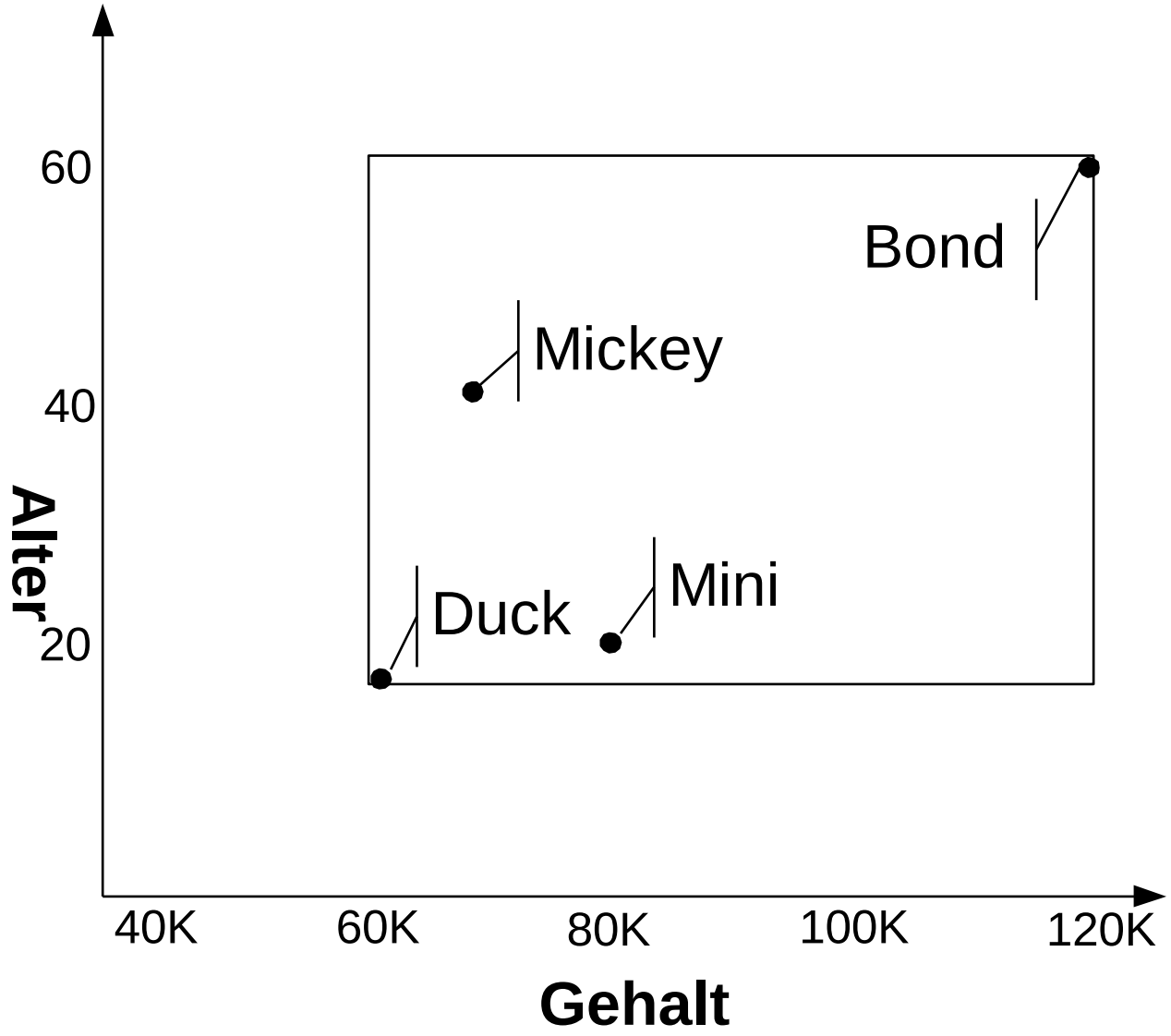
- Innere Knoten und Blätter sind unterschiedlich
- Einträge in innere Knoten:
  - Angabe einer k-dimensionalen Region
    - Interval in jeder Dimension
  - Verweise auf Kindknoten

Regionen können überlappen



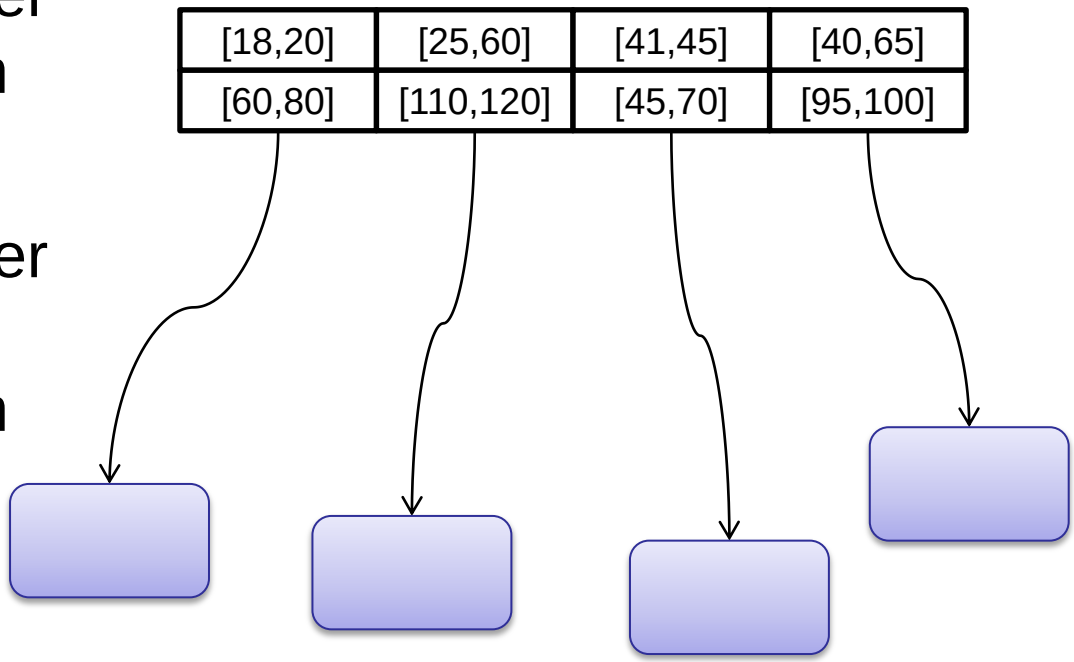
- Einträge Blattknoten:
  - Tupeldaten

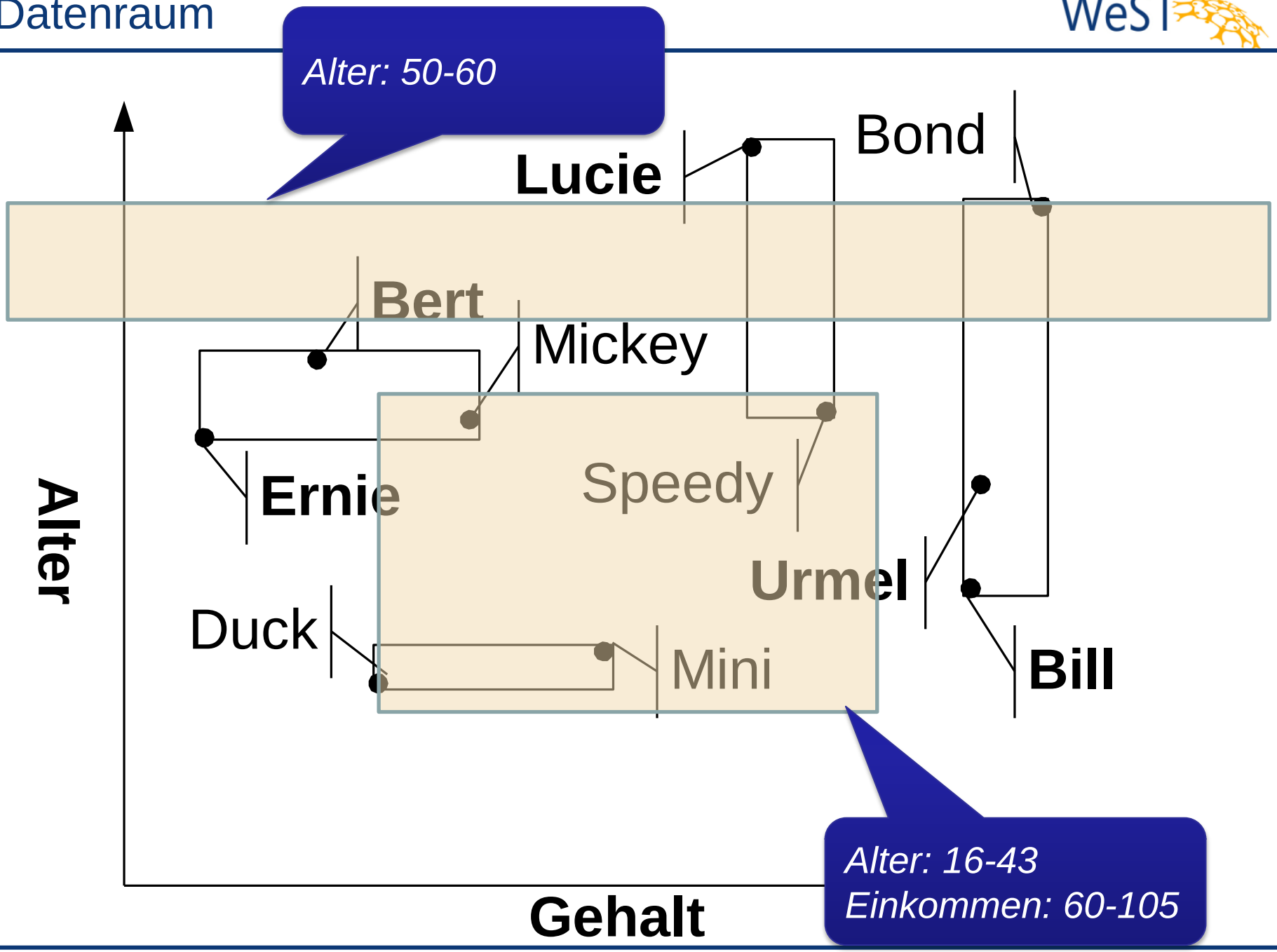



60
40
20
Alter
Bond
Mickey
Duck
Mini
40K
60K
80K
100K
120K
Gehalt

- Suchen
  - Identifizieren *aller* betroffenen Region(en)
  - Auswahl der tatsächlich betroffenen Tupel

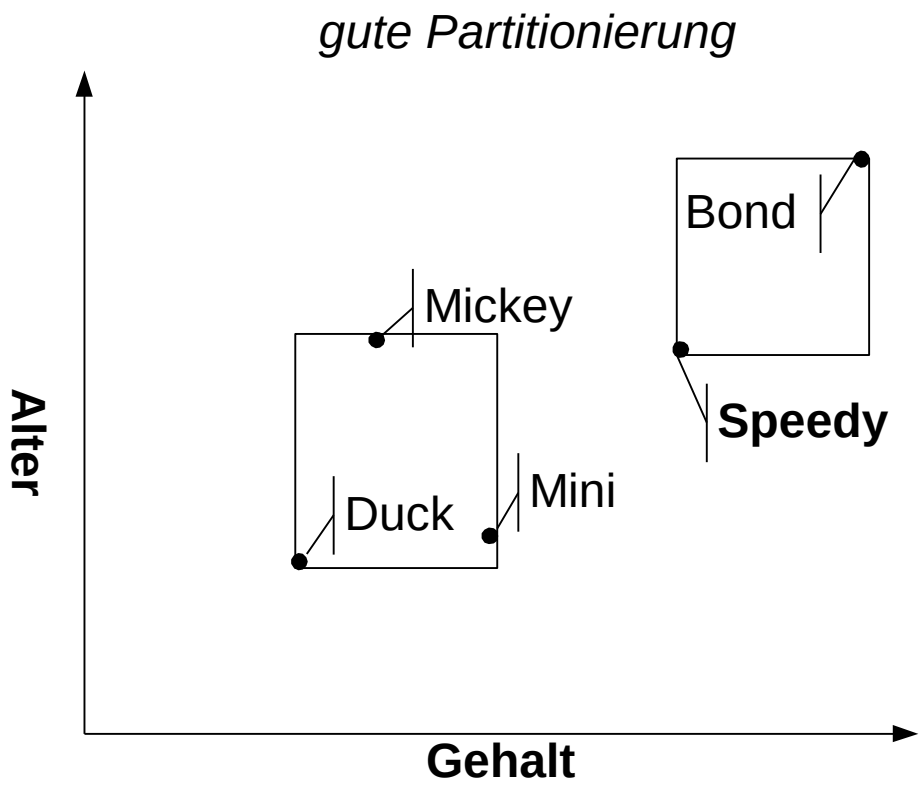
- Beispiel:
  - Alle Personen im Alter von 50 bis 60 Jahren
  - Alle Personen im Alter von 16 bis 43 Jaren und Gehalt zwischen 60 und 105

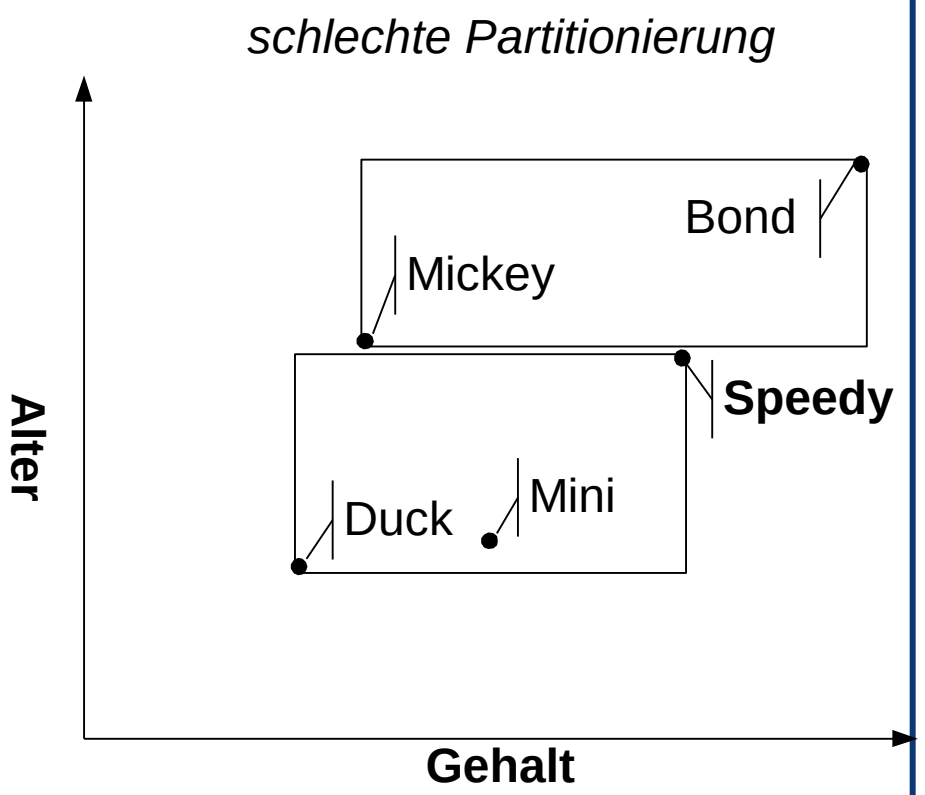



Alter: 50-60
Lucie
Bond
Bert
Mickey
Ernie
Speedy
Urmel
Duck
Mini
Bill
Alter
Gehalt
Alter: 16-43
Einkommen: 60-105

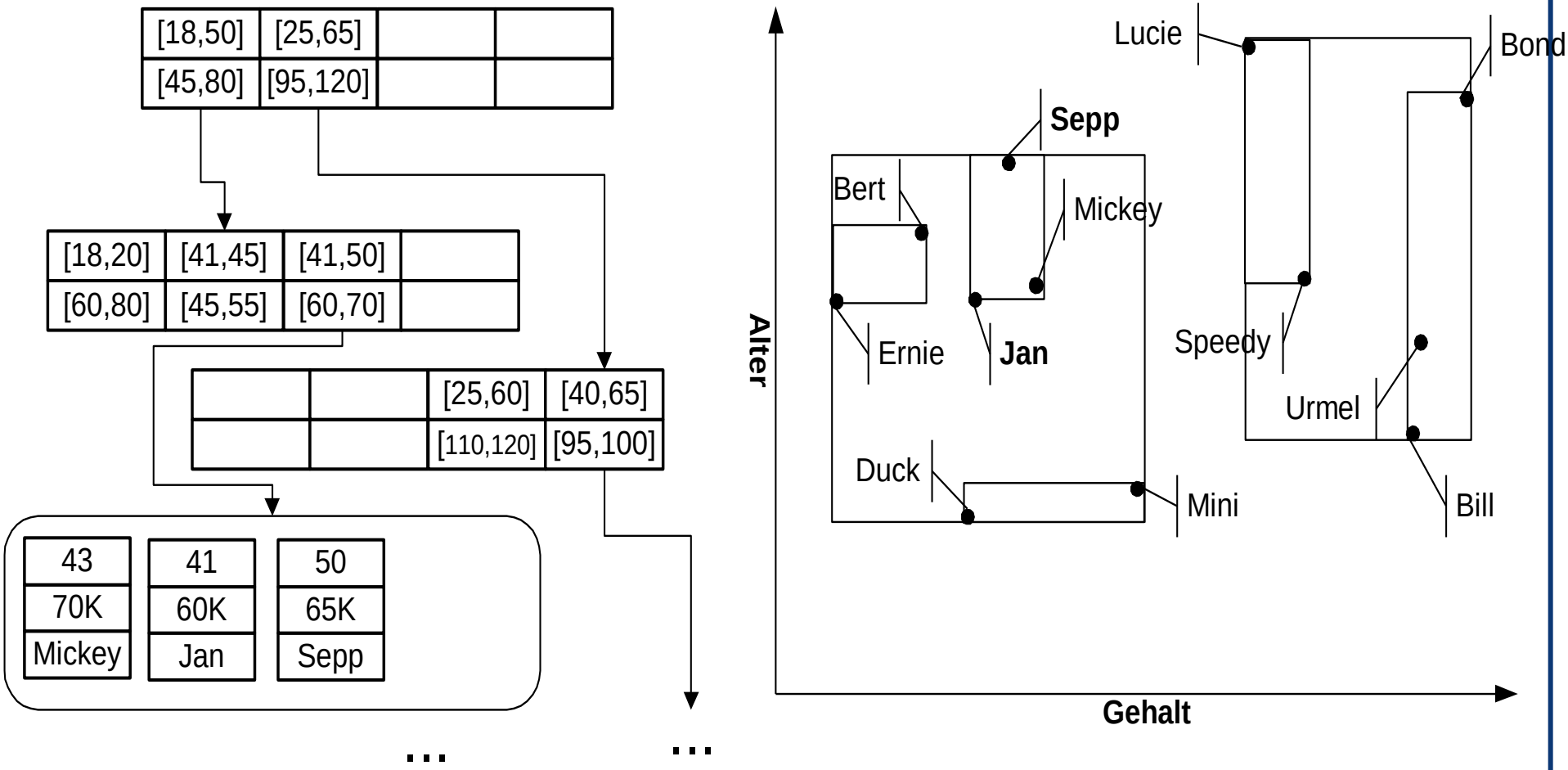
- Suche passende Region(en) und zugehöriges Blatt:
  - Keine Region: passe die Region an, die am wenigsten vergößert werden muss
  - Eine Region: erweitere diese
  - Mehrere Regionen: wähle *eine* aus
- Einfügen
  - In das ausgewählte Blatt
  - Bei Überlauf:
    - Region aufteilen (Partionierung):
      - Gleichmäßige Verteilung der Einträge
      - Möglichst kleine Regionen
    - Optimal Aufteilung zu aufwändig – Heuristiken
  - Evtl. Partitionierung der inneren Knoten nötig (R-Baum wächst nach oben)

Gute vs. schlechte Partitionierung





Nach oben gewachsener R-Baum



Zusammenfassung

- Zugriffslücke erfodert effizienten Zugriff auf Hintergrundspeicher
- Indexstrukturen optimieren die Anzahl der Zugriffe
  - ISAM
  - B-Bäume
  - B+-Bäume
  - Erweiterbares Hashing
- Clusterung gruppiert oft genutzte Daten zusammen
- Mehrdimensionale Indexstrukturen für kombinierte Anfragen

51

Fragen ?


gottron@uni-koblenz.de


http://west.uni-koblenz.de/teaching/ws1213/datenbanken