MOBI-DBS-B: Datenbanksysteme Architektur und Anfrageverarbeitung

Vorlesung Sommersemester 2019

Tanya Braun, Universität zu Lübeck

Lehrauftrag SoSe 19, Universität Bamberg

Architektur, Indexierung, Anfrageverarbeitung

Inhalte

- Architektur
  - Speicher
  - Puffer
  - Seiten
- Anfrageverarbeitung
  - Anfragepläne
  - Indexierung
    - ISAM-Index
    - B^+-Bäume (B*-Bäume)
    - Hash-basierte Indexe
  - Join-Verarbeitung
  - Optimierungspotentiale

Kompetenzen

- Verstehen, wie Anfragen im Hintergrund bearbeitet werden und dadurch Probleme erkennen
- Verstehen, wann und warum Indices in Datenbanken eingesetzt werden
- Relevante Indexstrukturen für Datenbanksysteme verstehen
- Anfragepläne verstehen und Möglichkeiten zur Optimierung erkennen

Bezug zu Phasen des DB-Entwurfs

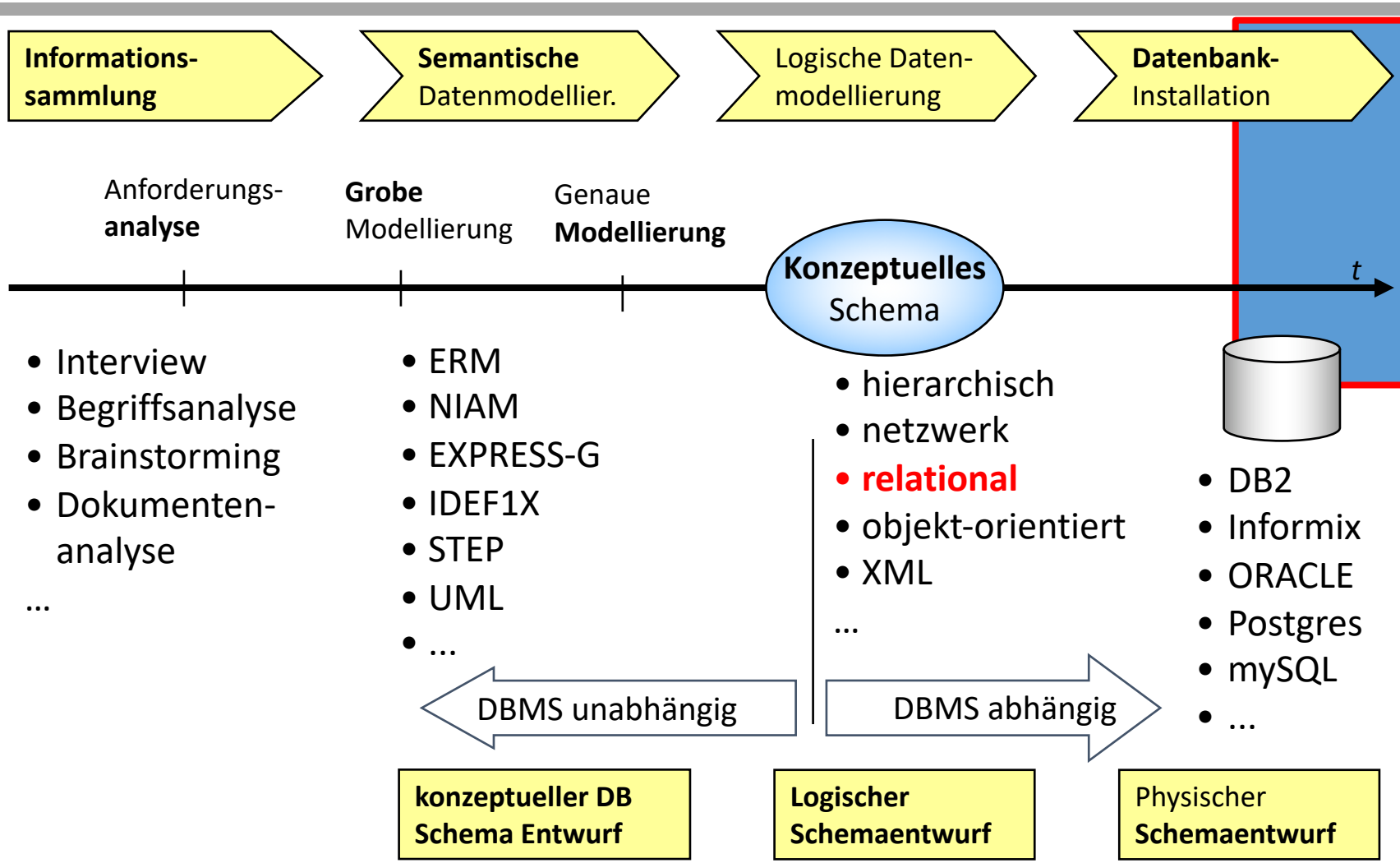
Danksagung

- Folien basieren ursprünglich auf dem Kurs

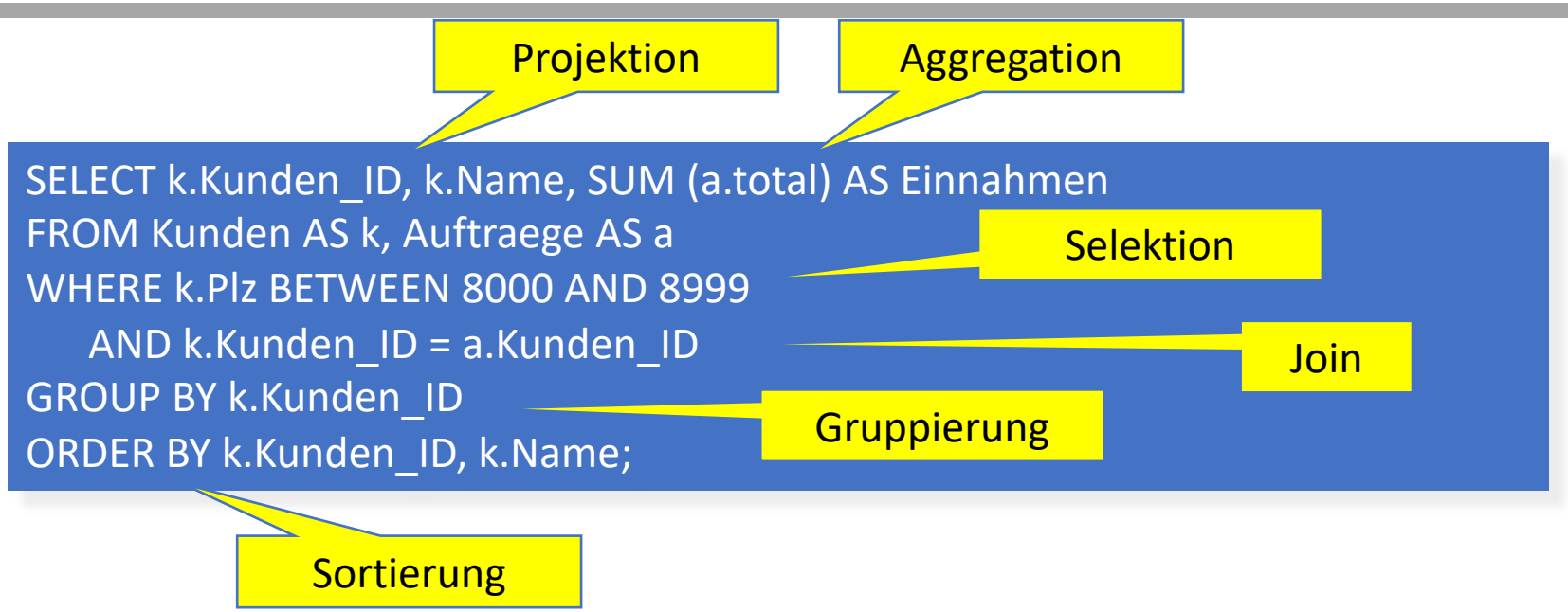
  „Architecture and Implementation of Database Systems“
  von Jens Teubner an der ETH Zürich

- Graphiken wurden mit Zustimmung des Autors aus diesem Kurs übernommen

Die Phasen des DB-Entwurfs



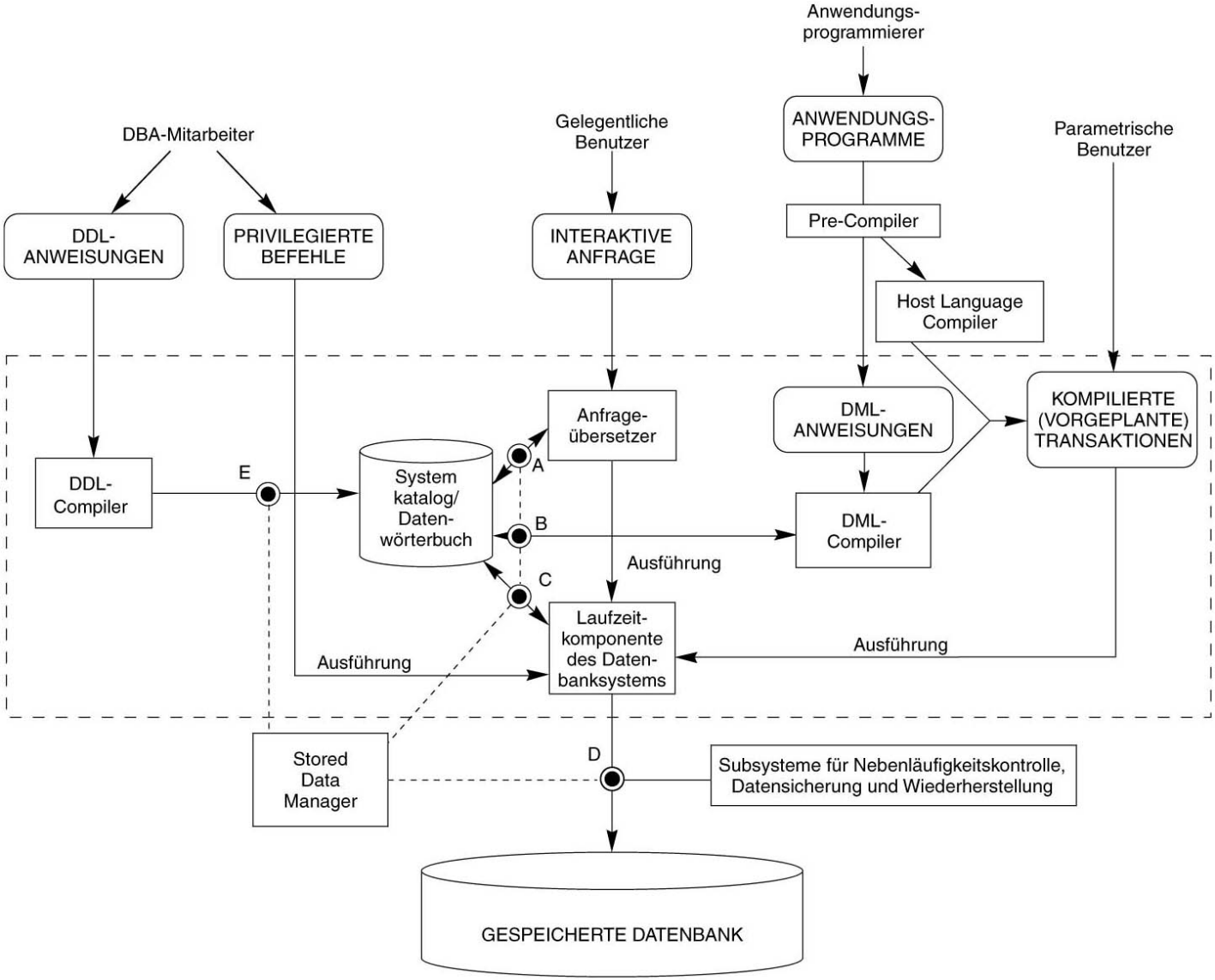
Anfragebeantwortung



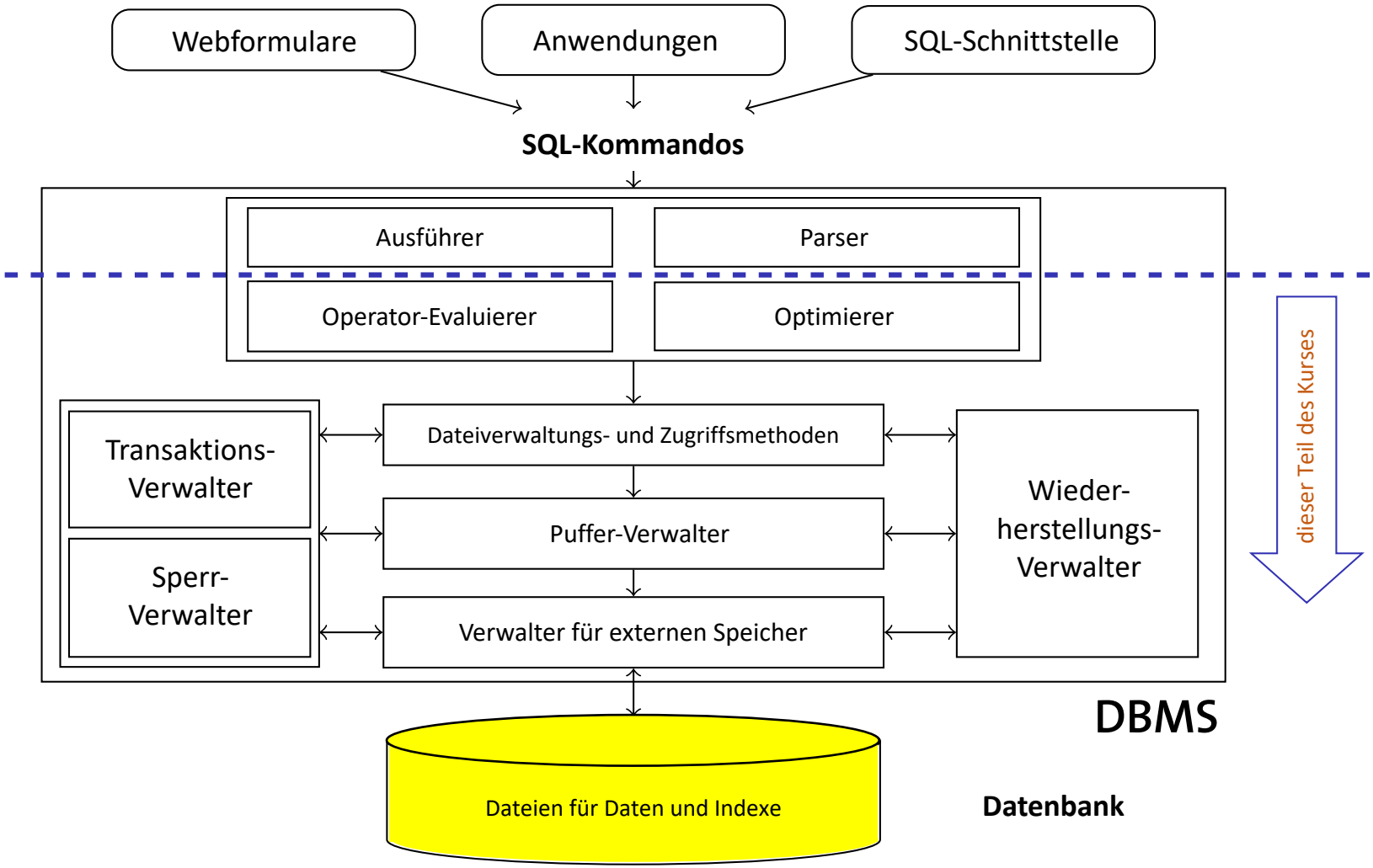
Ein DBMS muss eine Menge von Aufgaben erledigen:

- mit minimalen Ressourcen
- über großen Datenmengen
- und auch noch so schnell wie möglich!

DBMS-Systemumgebung



Architektur eines DBMS



Speicherung

- Daten in Datenbanken sind i.d.R.
  - Persistent
  - Zu groß um in den Hauptspeicher zu passen

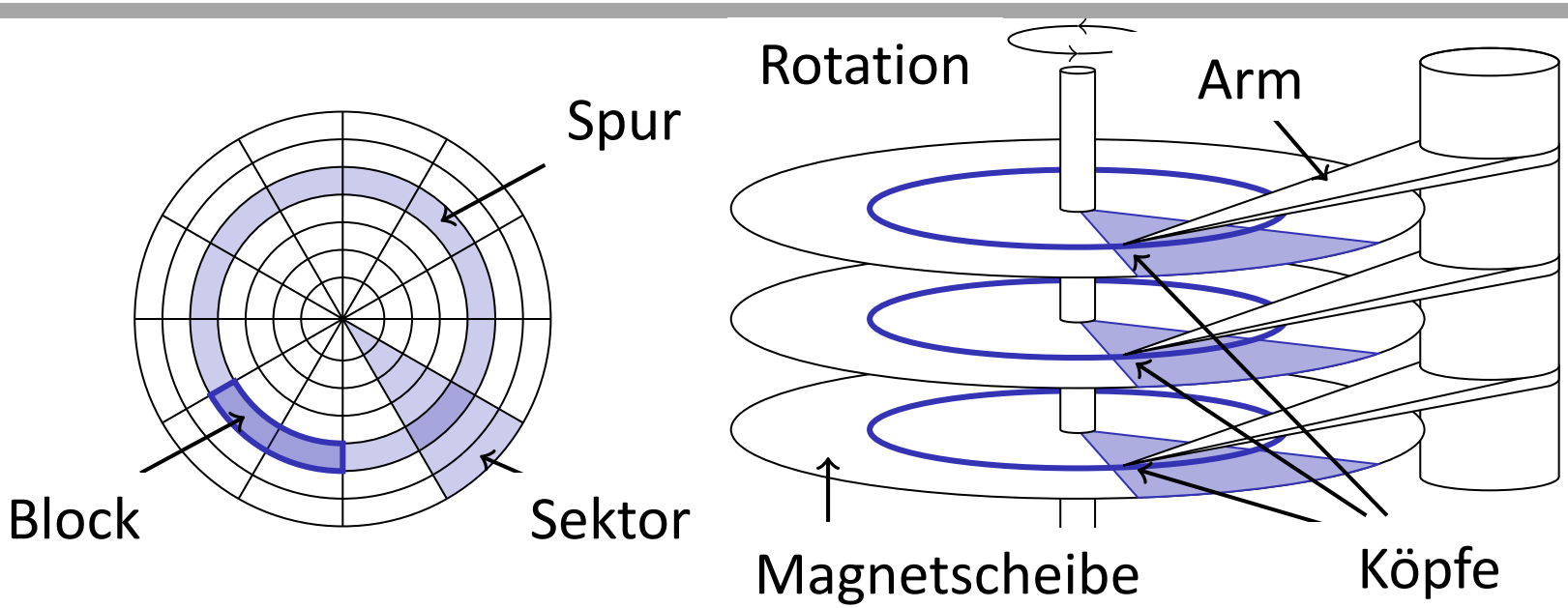
- Anforderung an Speicherung
  - Entsprechend große Menge an Speicherplatz
  - Schneller Zugriff vs. vertretbare Kosten
  - Sicherung der Daten gegeben (Totalverlust inakzeptabel)

Speicherhierarchie

	Kapazität	Latenz
• CPU (mit Registern)	Bytes	< 1 ns
• Cache-Speicher	Kilo-/Mega-Bytes	< 10 ns
• Hauptspeicher	Giga-Bytes	20-100 ns
• Flash-Speicher / SSD	Giga/Tera/Peta-Bytes	30-250 µs
• Festplatte	Tera/Peta-Bytes	3-10 ms
• Bandautomat	Peta-Bytes	variierend

- Zur CPU: Schnell aber klein
- Zur Peripherie: Langsam aber groß
- Cache-Speicher zur Verringerung der Latenz
- Blockweises Lesen/Schreiben ab Flash/SSD (Block etwa 4K)

Magnetische Platten / Festplatten



- Schrittmotor positioniert Arme auf bestimmte Spur
- Magnetscheiben rotieren ständig

Photo: http://www.metallurgy.utah.edu/

Zugriffszeit bei Festplatten

Konstruktion der Platten hat Einflüsse auf Zugriffszeit (lesend und schreibend) auf einen Block

1. Bewegung der Arme auf die gewünschte Spur (Suchzeit t_s)
2. Wartezeit auf gewünschten Block bis er sich unter dem Arm befindet (Rotationsverzögerung t_r)
3. Lesezeit bzw. Schreibezeit (Transferzeit t_{tr})

Zugriffszeit: $t = t_s + t_r + t_{tr}$

- Beispiel: Hitachi Travelstar 7K200 (für Laptops)
  - 4 Köpfe, 2 Magnetplatten, 512 Bytes/Sektor, Kapazität: 200 GB
    → 2 Köpfe pro Platte, max. halbe Runde für Blockanfang
  - Rotationsgeschwindigkeit: 7200 rpm → = 120 rps → 1r = 8,33 ms
  - Mittlere Suchzeit: 10 ms → $t_s = 10$ ms
  - Transferrate: ca. 50 MB/s → $t_{tr} = 8KB/50MB\ s = 0,16$ ms
  - Wie groß ist die Zugriffszeit auf einen Block von 8 KB?
    → $t = 10\ ms + 4,17\ ms + 0,16ms = 14,33\ ms$

$t_r = 8,33/2\ ms = 4,17\ ms$

Sequentieller vs. Wahlfreier Zugriff

Beispiel: Lese 1000 Blöcke von je 8 KB (8MB)

- Wahlfreier Zugriff:
  - $t_{rnd} = 1000 \cdot 14{,}33\ ms = 14\,330\ ms$
- Sequentieller Zugriff:
  - Travelstar 7k200 hat 63 Sektoren pro Spur, mit einer Track-to-Track-Suchzeit $t_{s,track\text{-}to\text{-}track}$ von 1 ms (von einer Spur zur nä. Spur)
  - Ein Block mit 8 KB benötigt 16 Sektoren (8KB/512 B/Sektor): 16.000 Sektoren lesen

  → Bei 63 Sektoren pro Spur macht das 16000/63 ≈ 254 Spuren
  - $T_{seq} = t_s + t_r + 1000 \cdot t_{tr} + 254 \cdot t_{s,track\text{-}to\text{-}track}$

    $\approx 10\ ms + 4{,}17\ ms + 1000 \cdot 0{,}16\ ms + 254 \cdot 1\ ms \approx 428\ ms$

- Einsicht: Sequentieller Zugriff **viel** schneller als wahlfreier Zugriff: Vermeide wahlfreie I/O wenn möglich
  - Wenn 428 ms / 14330 ms ≈ 3% einer 8MB Datei wahlfrei benötigt wird, kann man gleich die ganze Datei lesen, sofern Blöcke hintereinander stehen.

Verbesserung der Festplattentechnologie

- Latenz der Platten über die letzten 10 Jahre nur marginal verbessert (≈ 10% pro Jahr)
  - Latenzproblem kaum zu vermeiden
  - Durchsatz kann recht leicht gesteigert werden durch Ausnutzung von Parallelität
- Aber:
  - Durchsatz (Transferraten) um ≈ 50% pro Jahr verbessert
  - Kapazität der Festplatten um ≈ 50% pro Jahr verbessert
- Daher:
  - Kosten für wahlfreien Zugriff über die Zeit hinweg relativ gesehen immer bedeutsamer
- Werden Festplatten durch SSDs ersetzt?

Netzwerk-Speicher ist kein Flaschenhals

- Durchsatz SSD: >500 MB/s
- SDRAM: 50 Gbit/s (Latenz: ~ ns)

- Ethernet
  - 100-Gbit/s (Latenz: ~ ms)
  - 400 Gbit/s
  - Terabit Ethernet (TbE) um 2020

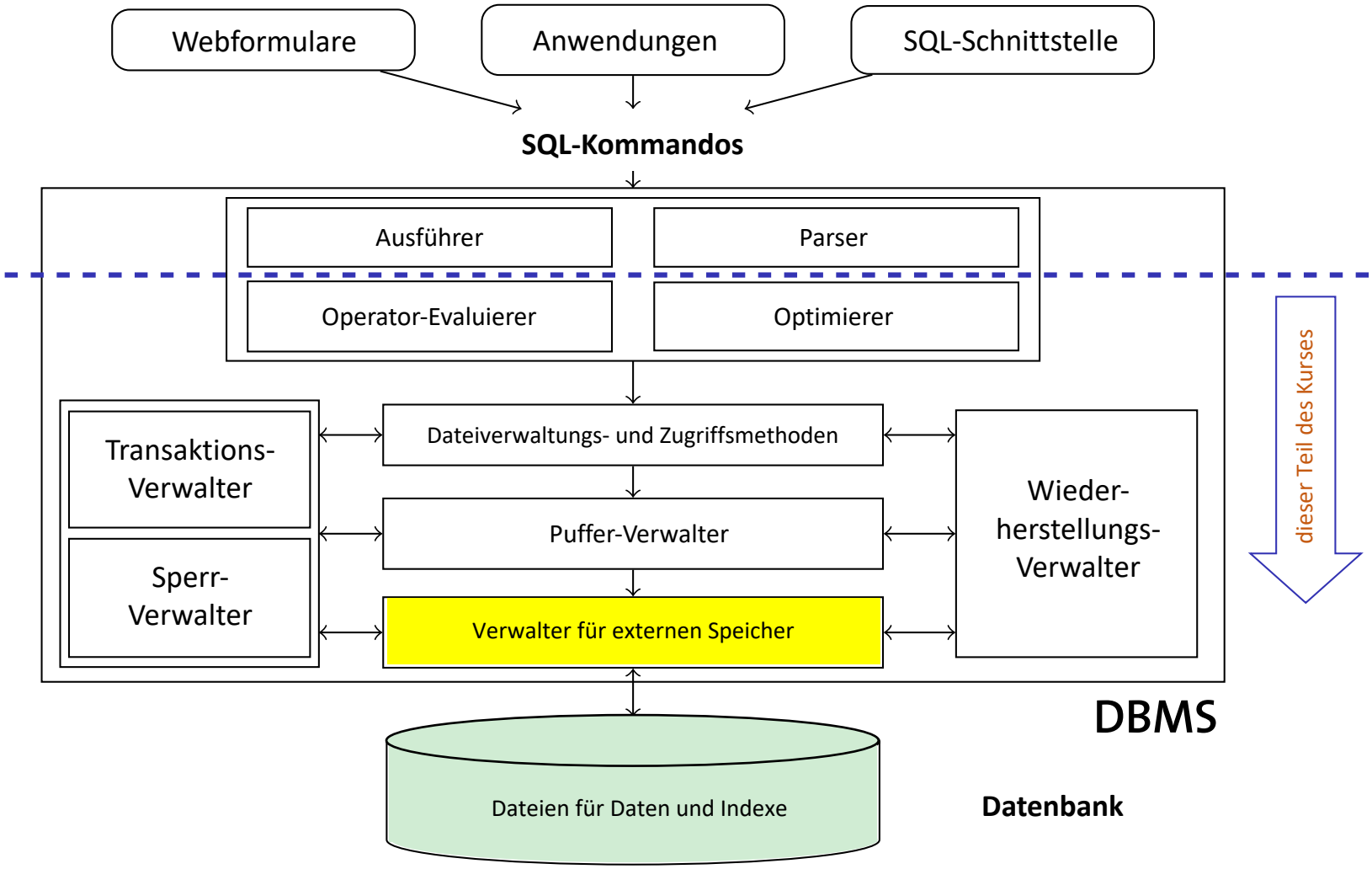
- Warum also nicht Datenbank-Speicher über das Netzwerk referenzieren?

Speichernetzwerk (Storage Area Network, SAN)

- Block-basierter Netzwerkzugriff auf Speicher
  - Als logische Platten betrachtet (Suche Block 4711 von Disk 42)
- SAN-Speichergeräte abstrahieren von RAID oder physikalischen Platten und zeigen sich dem DBMS als logische Platten
  - Hardwarebeschleunigung und einfachere Verwaltung
- Üblicherweise lokale Netzwerke mit multiplen Servern und Speicherressourcen
  - Bessere Fehlertoleranz und erhöhte Flexibilität

- Alternative: Cloud-Speicher
  - Cluster von vielen Standard-PCs (z.B. Google, Amazon)
    - Systemkosten vs. Zuverlässigkeit und Performanz
    - Verwendung massiver Replikation von Datenspeichern
  - CPU-Zyklen und Disk-Kapazität als Service
    - Amazons „Simple Storage System (S3)“
      - Latenz: 100 ms bis 1s!
      - Datenbank auf Basis von S3 entwickelt in 2008

Architektur eines DBMS



Verwaltung des externen Speichers

- Abstraktion von technischen Details der Speichermedien
- Konzepte der Seite (page) mit typischerweise 4-64KB als Speichereinheiten für die restlichen Komponenten
- Verzeichnis für Abbildung

  Seitennummer → Physikalischer Speicherort

  wobei der physikalische Speicherort
  - eine Betriebssystemdatei inkl. Versatz,
  - eine Angabe Kopf-Sektor-Spur einer Festplatte oder
  - eine Angabe für Bandgerät und -nummer inkl. Versatz

  sein kann

Leere Seiten

- Leere Seiten
  - Insert: Finde leere Seite, die das Datenobjekt speichern kann
  - Delete: Seite wird frei

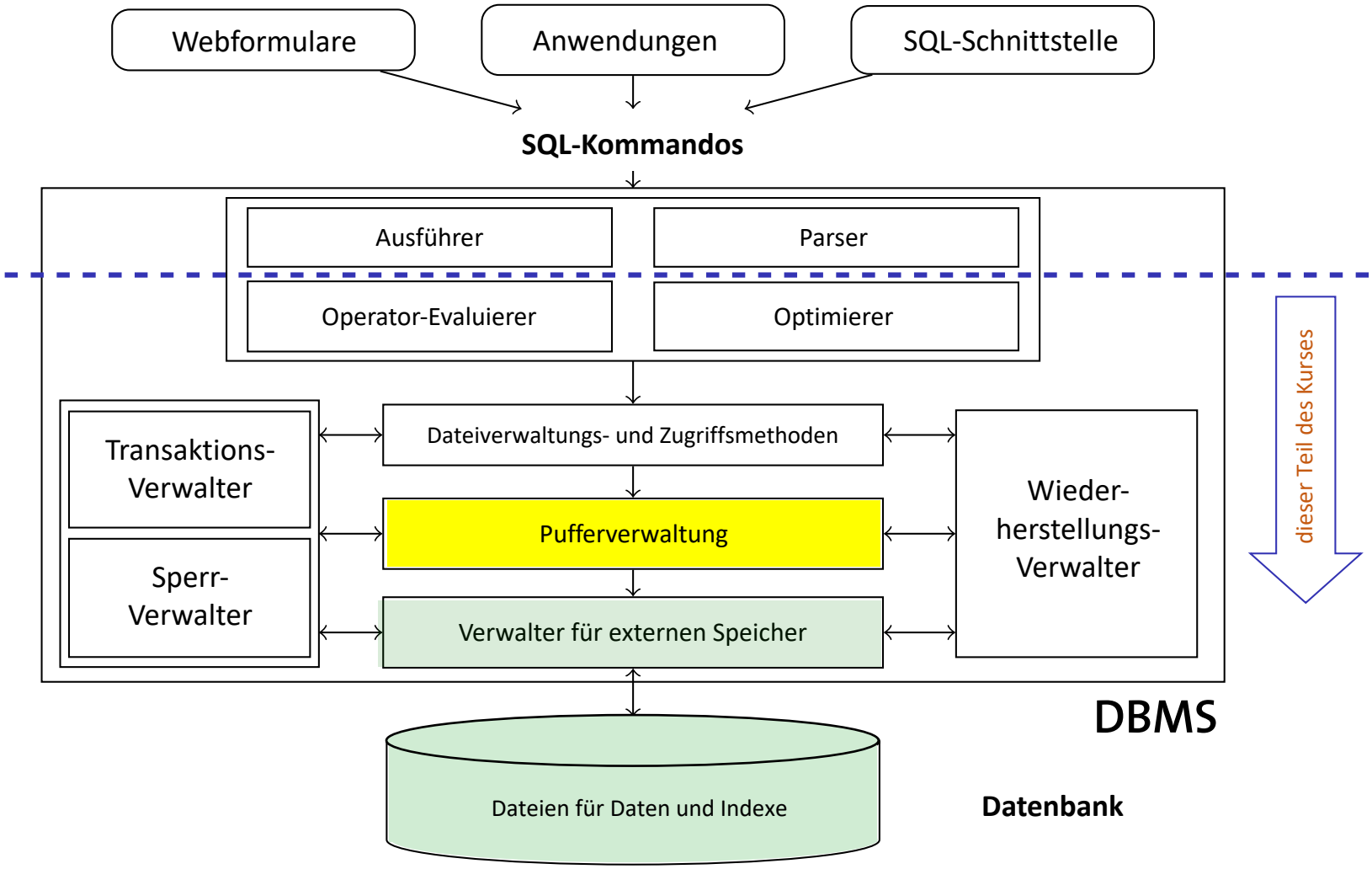
- Verwaltung leerer Seiten:

  1. Liste der freien Seiten
     - Hinzufügung falls Seite nicht mehr verwendet

  2. Bitmap mit einem Bit für jede Seite
     - Umklappen des Bits p, wenn Seite p (de-)alloziert wird
     - Finden von hintereinanderliegenden Seiten einfacher

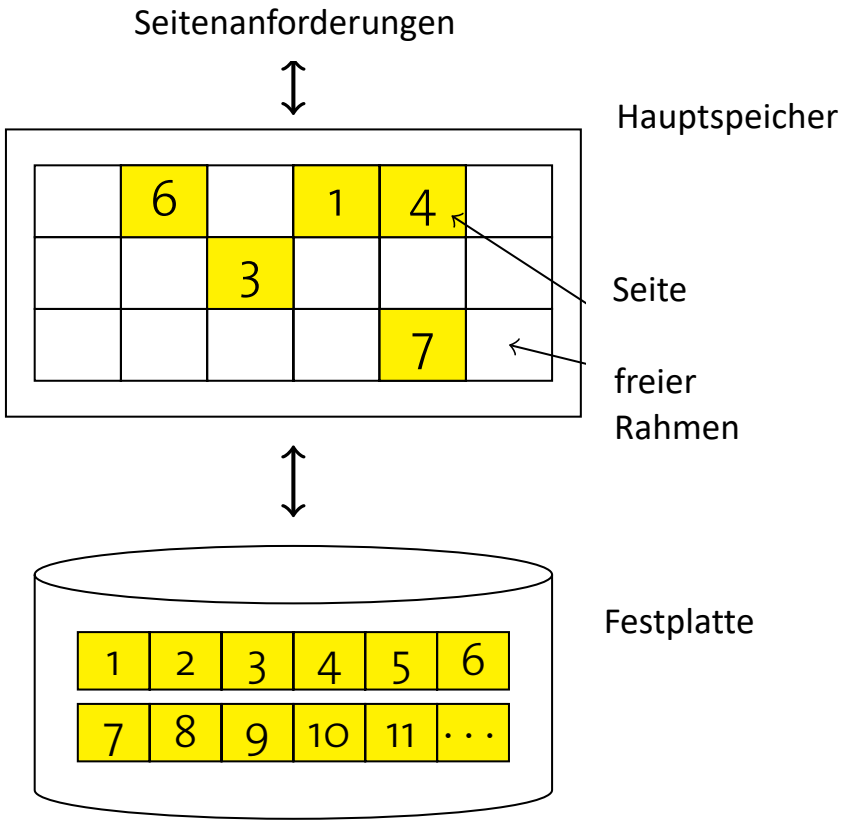
- Persistent als Verwaltungsinformationen zu speichern

Architektur eines DBMS



Puffer-Verwalter

- Vermittelt zwischen externem und internem Speicher (Hauptspeicher)
- Verwaltet hierzu einen besonderen Bereich im Hauptspeicher, den Pufferbereich (buffer pool)
- Externe Seiten in Rahmen des Pufferbereichs laden
- Ersetzungsstrategien für den Fall, dass der Pufferbereich voll ist



Schnittstelle zum Puffer-Verwalter

- Funktion pin für Anfragen nach Seiten
- Funktion unpin für Freistellungen von Seiten nach Verwendung

- pin(pageno)
  - Anfrage nach Seitennummer pageno
  - Lade Seite in Hauptspeicher falls nötig
  - Rückgabe einer Referenz auf pageno

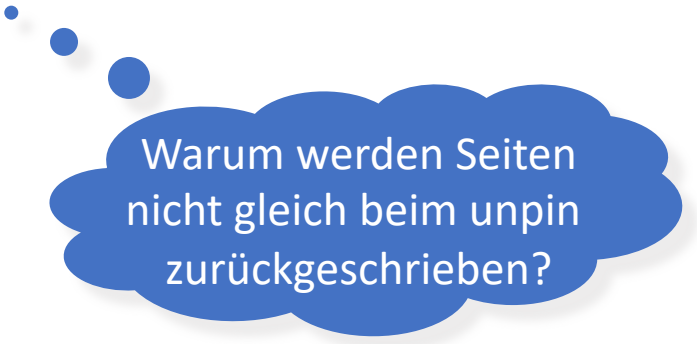
- unpin(pageno, dirty)
  - Freistellung einer Seite pageno zur möglichen Auslagerung
  - dirty = true bei Modifikationen der Seite

Implementation von pin()

```
1  Function: pin(pageno)
2  if buffer pool already contains pageno then
3  |   pinCount (pageno) ← pinCount (pageno) + 1 ;
4  |   return address of frame holding pageno ;
5  else
6  |   select a victim frame v using the replacement policy ;
7  |   if dirty (v) then
8  |   |   write v to disk ;
9  |   read page pageno from disk into frame v ;
10 |   pinCount (pageno) ← 1 ;
11 |   dirty (pageno) ← false ;
12 |   return address of frame v ;
```

Implementation von unpin()

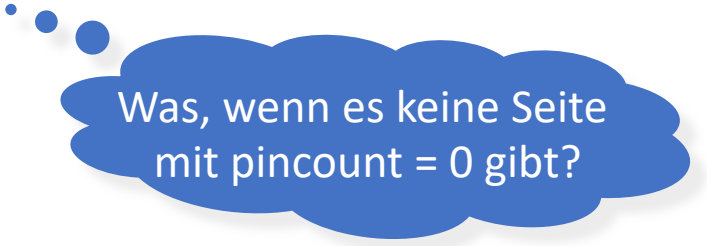
```
1 Function: unpin(pageno, dirty)
2 pinCount (pageno) ← pinCount (pageno) − 1;
3 if dirty then
4     dirty (pageno) ← dirty;
```



Ersetzungsstrategien

- Least Recently Used (LRU)
  - Verdrängung der Seite mit am längsten zurückliegendem unpin()
- LRU-k
  - Wie LRU, aber k-letztes unpin(), nicht letztes
- Most Recently Used (MRU)
  - Verdrängung der Seite mit jüngstem unpin()
- Random
  - Verdrängung einer beliebigen Seite
- Viele mehr...

- Seite muss pincount = 0 haben, um für Ersetzung zur Verfügung zu stehen!



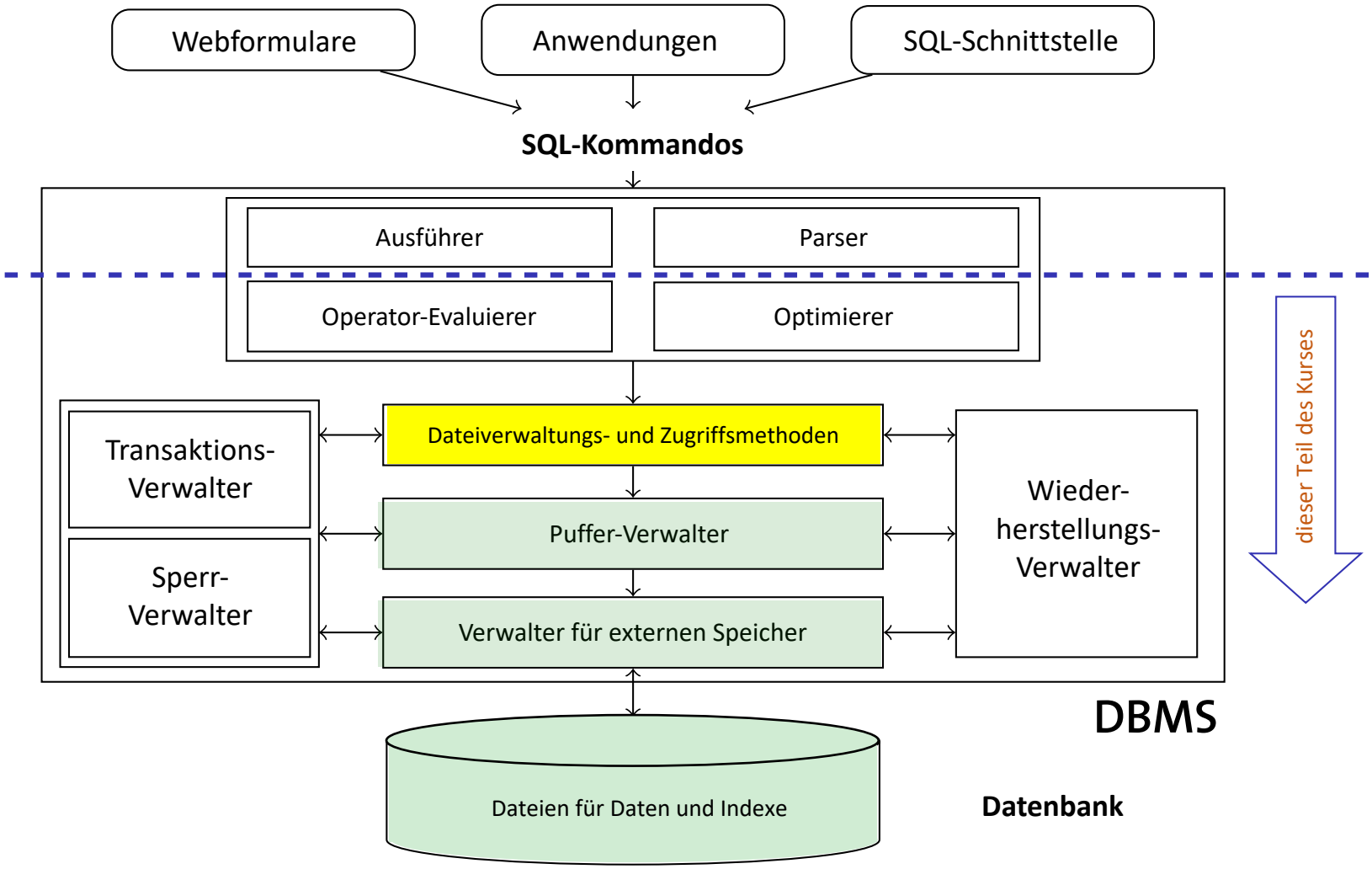
Pufferverwaltung in der Praxis

- Prefetching
  - Antizipation von Anfragen, um CPU- und I/O-Aktivität zu überlappen
    - Spekulatives Prefetching: Nehme sequentiellen Seitenzugriff an und lese im Vorwege
    - Prefetch-Listen mit Instruktionen für den Pufferverwalter für Prefetch-Seiten
- Fixierungs- oder Verdrängungsempfehlung
  - Höherer Code kann Fixierung (z.B. für Indexseiten) oder schnelle Verdrängung (bei seq. Scans) empfehlen
- Partitionierte Pufferbereiche
  - Z.B. separate Bereiche für Index und Tabellen

Datenbanken vs. Betriebssysteme

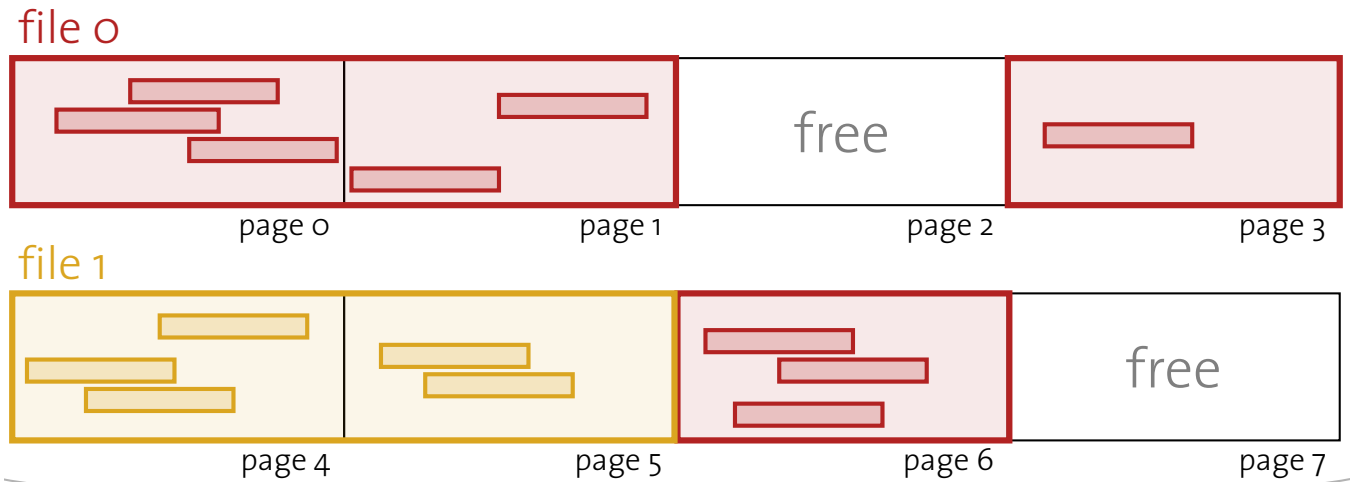
- Haben wir nicht gerade ein Betriebssystem entworfen?
- Ja
  - Verwaltung für externen Speicher und Pufferverwaltung ähnlich
- Aber
  - DBMS weiß mehr über Zugriffsmuster (z.B. für Prefetching)
  - Limitationen von Betriebssystemen häufig zu stark für DBMS (Obergrenzen für Dateigrößen, Plattformunabhängigkeit nicht gegeben)
- Gegenseitige Störung möglich
  - Doppelte Seitenverwaltung
  - DMBS-Transaktionen vs. Transaktionen auf Dateien organisiert vom Betriebssystem
  - DBMS Pufferbereiche durch Betriebssystem ausgelagert
- DBMS schaltet oft Betriebssystemdienste aus
  - Direkter Zugriff auf Festplatten
  - Eigene Prozessverwaltung

Architektur eines DBMS



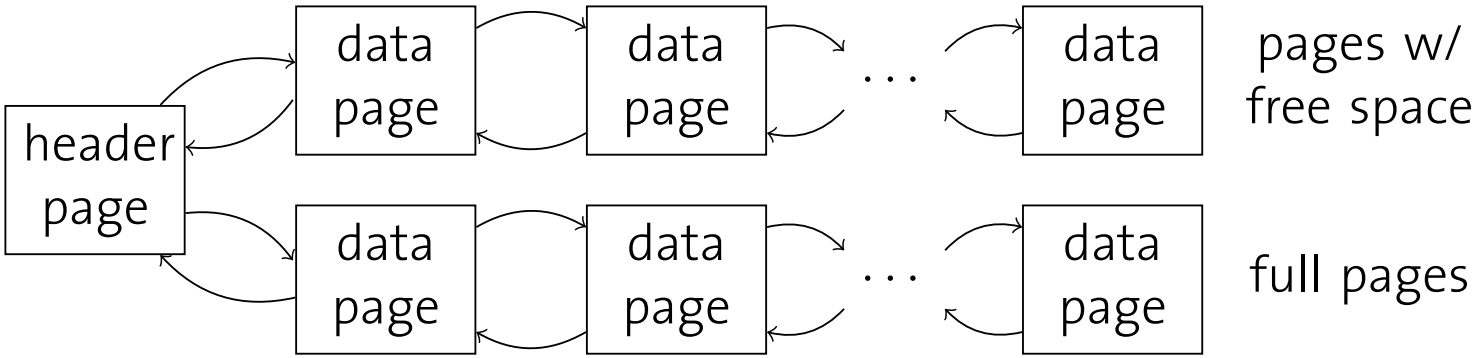
Datenbank-Dateien

- Seitenverwaltung unbeeinflusst vom Inhalt
- DBMS verwaltet Tabellen von Tupeln, Indexstrukturen, ...
- Tabellen sind Dateien von Datensätzen (records)
  - Datei besteht aus einer oder mehrerer Seiten
  - Jede Seite speichert eine oder mehrere Datensätze
  - Jeder Datensatz korrespondiert zu einem Tupel



Heap-Dateien

- Wichtigster Dateityp: Speicherung von Datensätzen mit willkürlicher Ordnung (konform mit SQL)
- Umsetzung
  - Verkettete Liste von Seiten



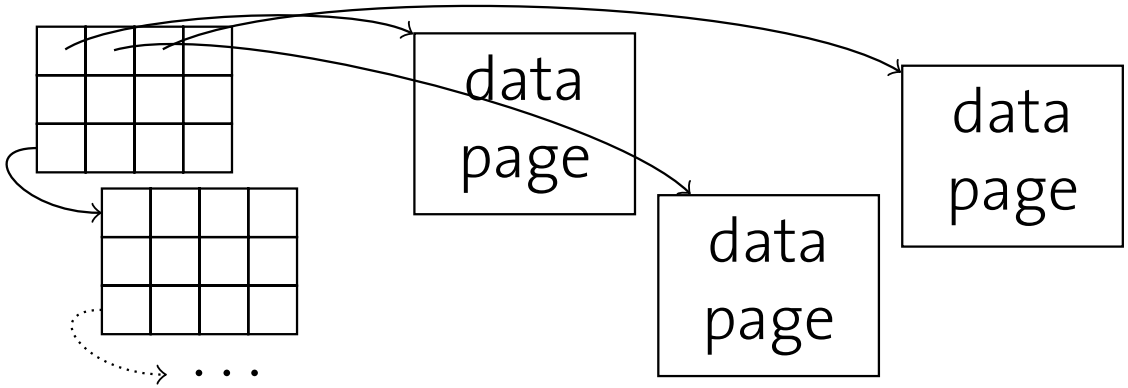
✓Einfach zu implementieren

✗Viele Seiten auf der Liste der freie Seiten (haben also noch Kapazität)

✗Viele Seiten anzufassen bis passende Seite gefunden

Heap-Dateien

- Wichtigster Dateityp: Speicherung von Datensätzen mit willkürlicher Ordnung (konform mit SQL)
- Umsetzung
  - Verzeichnis von Seiten



  - Verwendung als Abbildung mit Informationen über freie Plätze (Granularität ist Abwägungssache)

✓ Suche nach freien Plätzen effizient

✗ Zusatzaufwand für Verzeichnisspeicher

Freispeicher-Verzeichnis

Welche Seite soll für neuen Datensatz gewählt werden?

- Append Only
  - Immer in letzte Seite einfügen, sonst neue Seite anfordern
- Best-Fit
  - Alle Seiten müssen betrachtet werden, Reduzierung der Fragmentierung
- First-Fit
  - Suche vom Anfang, nehme erste Seite mit genug Platz
  - Erste Seiten füllen sich schnell, werden immer wieder betrachtet
- Next-Fit
  - Verwalte Zeiger und führe Suche fort, wo Suche beim vorigen Male endete

Inhalte einer Seite

ID	NAME	SEX
4711	John	M
1723	Marc	M
6381	Betty	F

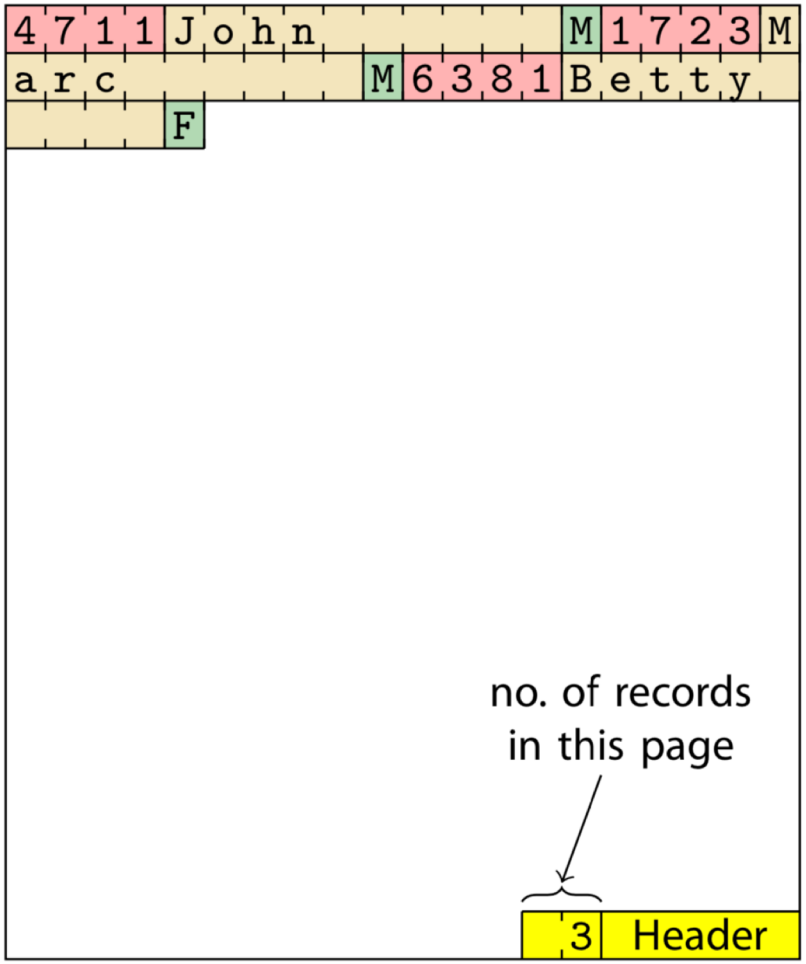
- Für jeden Datensatz ergibt sich eine Datensatz-Kennung (record identifier, rid), typisch:

  <pageno, slotno>

- Datensatz-Position (Versatz auf der Seite):

  slotno x Bytes pro Slot

- Datensatz gelöscht?
  - rid sollte sich nicht ändern



Inhalte einer Seite

ID	NAME	SEX
4711	John	M
~~1723~~	~~Marc~~	~~M~~
6381	Betty	F

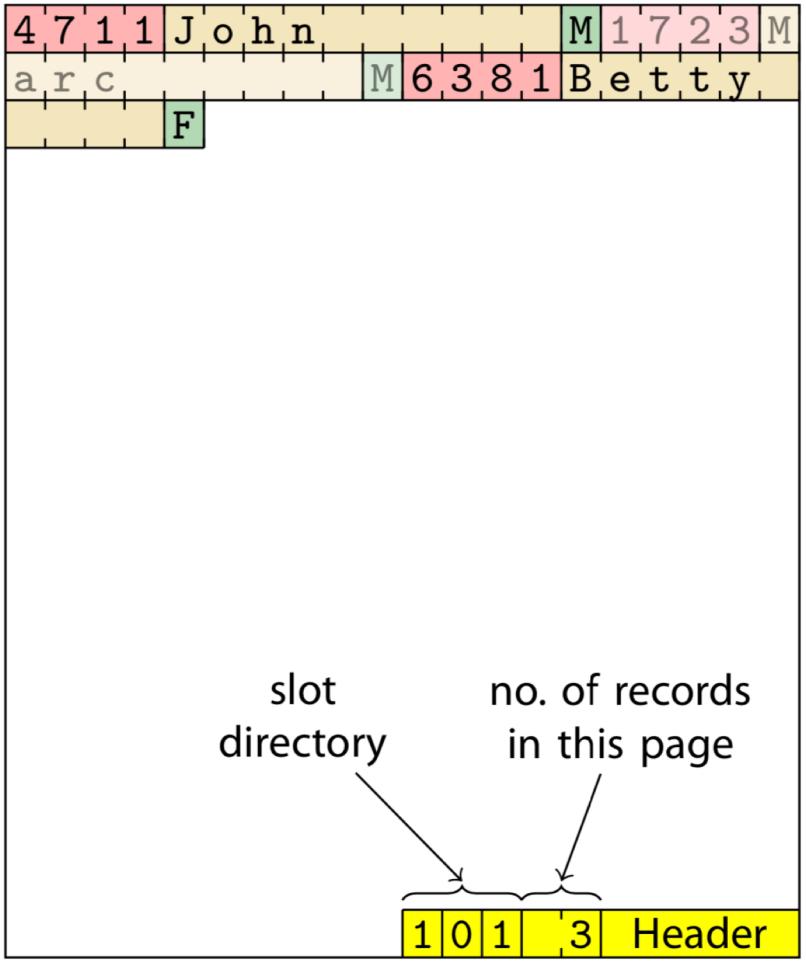
- Für jeden Datensatz ergibt sich eine Datensatz-Kennung (record identifier, rid), typisch:

  <pageno, slotno>

- Datensatz-Position (Versatz auf der Seite):

  slotno x Bytes pro Slot

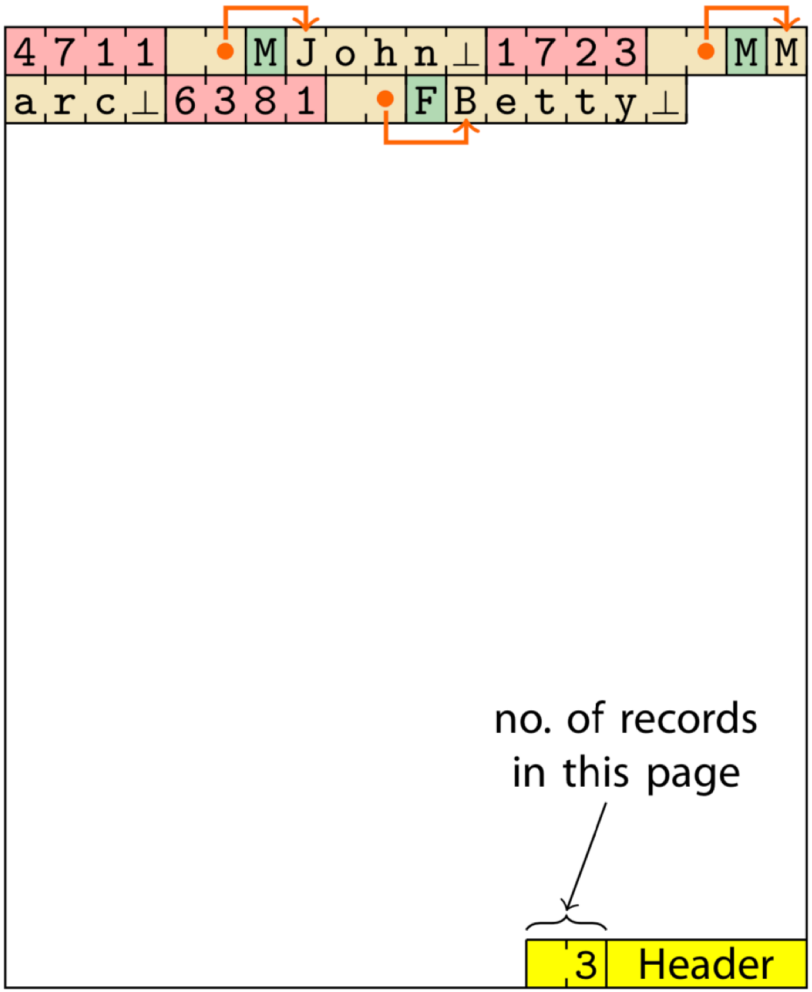
- Datensatz gelöscht?
  - rid sollte sich nicht ändern
  - Slot-Verzeichnis (Bitmap) markiert durch 1/0, ob Datensatz noch gültig



Inhalte einer Seite: Felder variabler Länge

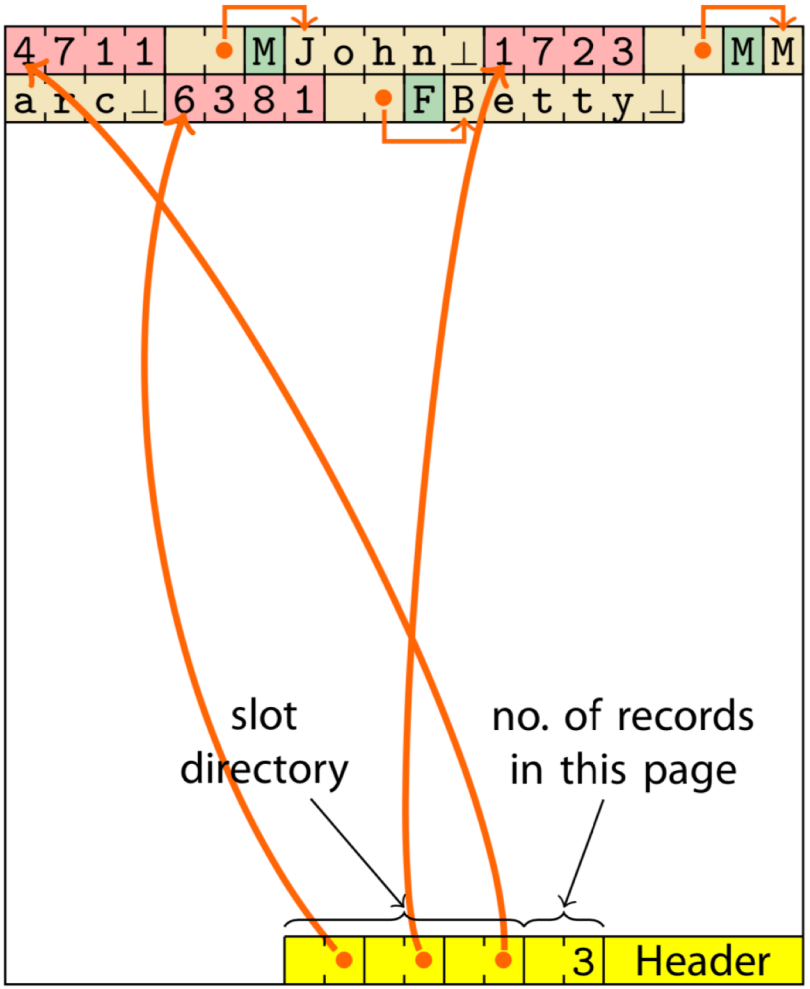
- Felder variabler Länge zum Ende verschoben
  - Platzhalter zeigt auf Position

Warum?



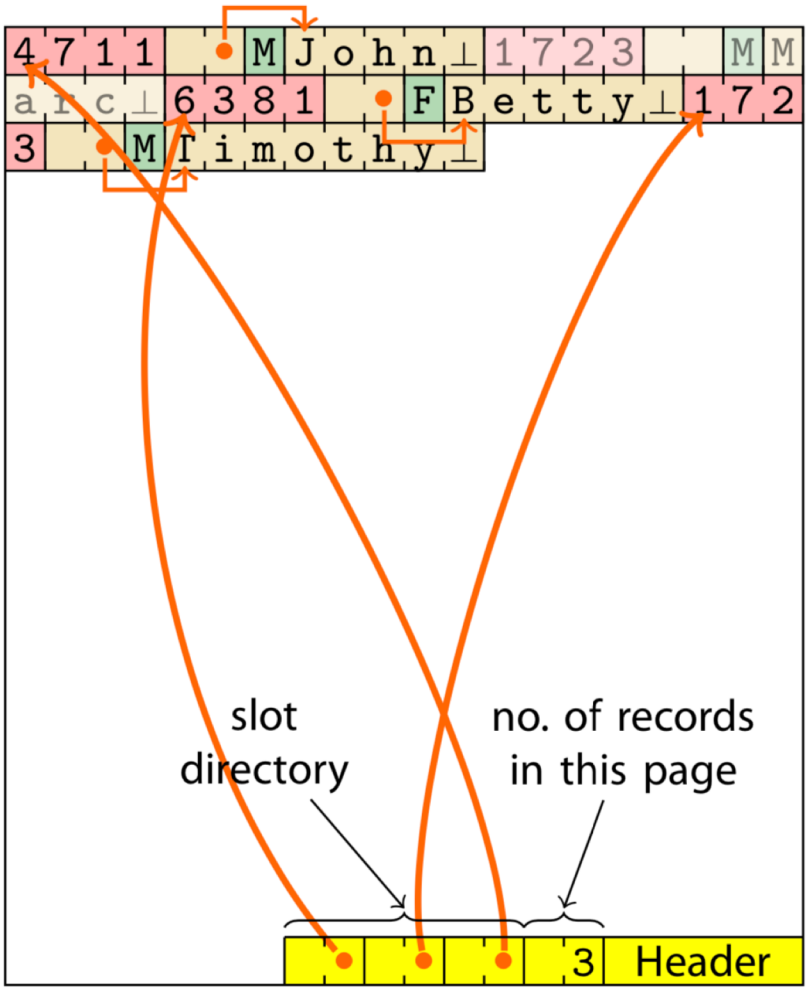
Inhalte einer Seite: Felder variabler Länge

- Felder variabler Länge zum Ende verschoben
  - Platzhalter zeigt auf Position
- Slot-Verzeichnis zeigt auf Start eines Feldes



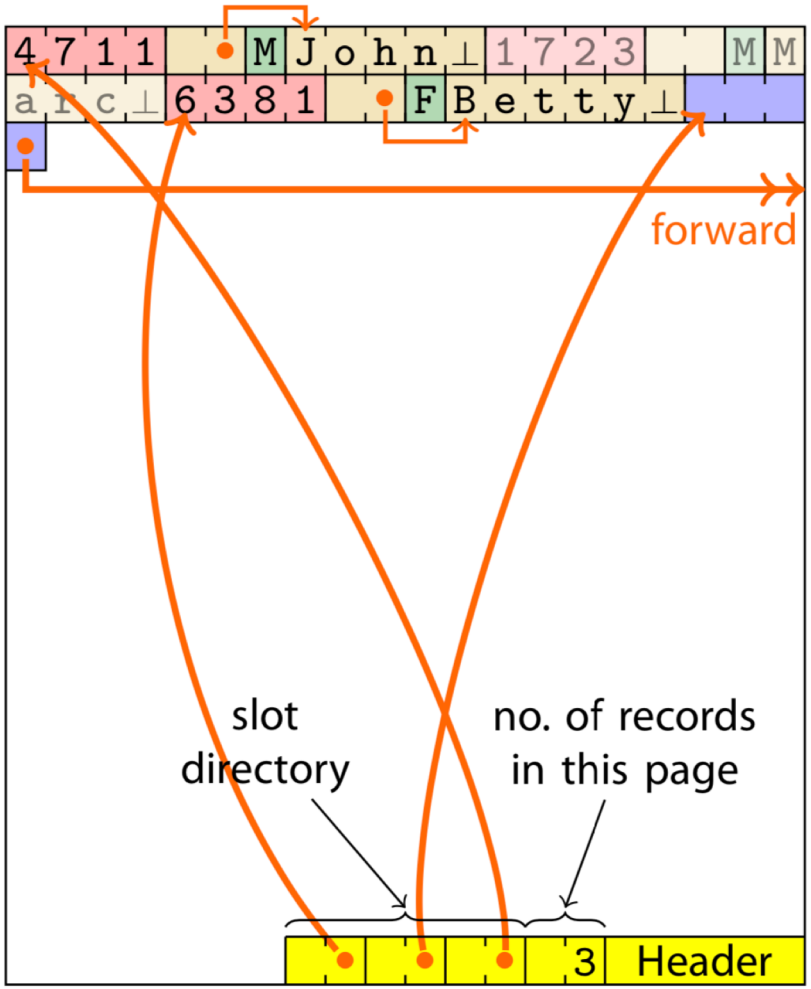
Inhalte einer Seite: Felder variabler Länge

- Felder variabler Länge zum Ende verschoben
  - Platzhalter zeigt auf Position
- Slot-Verzeichnis zeigt auf Start eines Feldes
- Felder können auf Seite verschoben werden (z.B. wenn sich Feldgröße ändert)



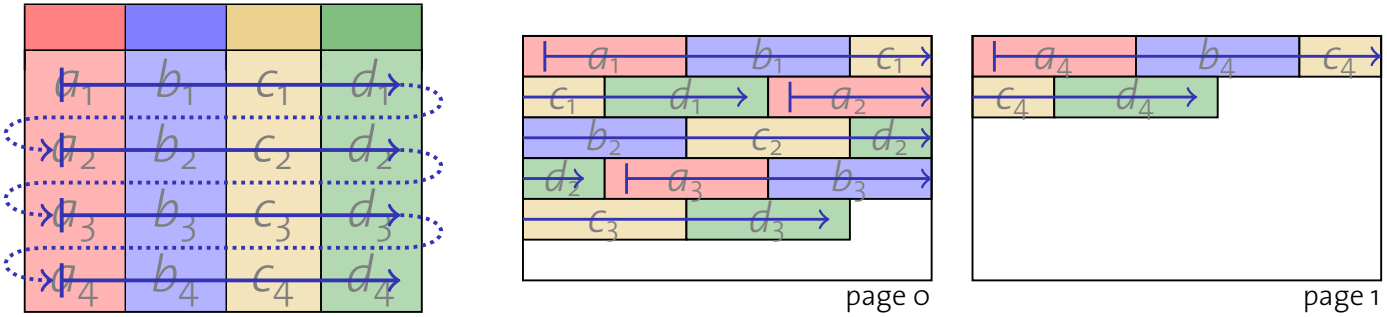
Inhalte einer Seite: Felder variabler Länge

- Felder variabler Länge zum Ende verschoben
  - Platzhalter zeigt auf Position
- Slot-Verzeichnis zeigt auf Start eines Feldes
- Felder können auf Seite verschoben werden (z.B. wenn sich Feldgröße ändert)
- Einführung einer Vorwärtsreferenz, wenn Feld nicht auf Seite passt

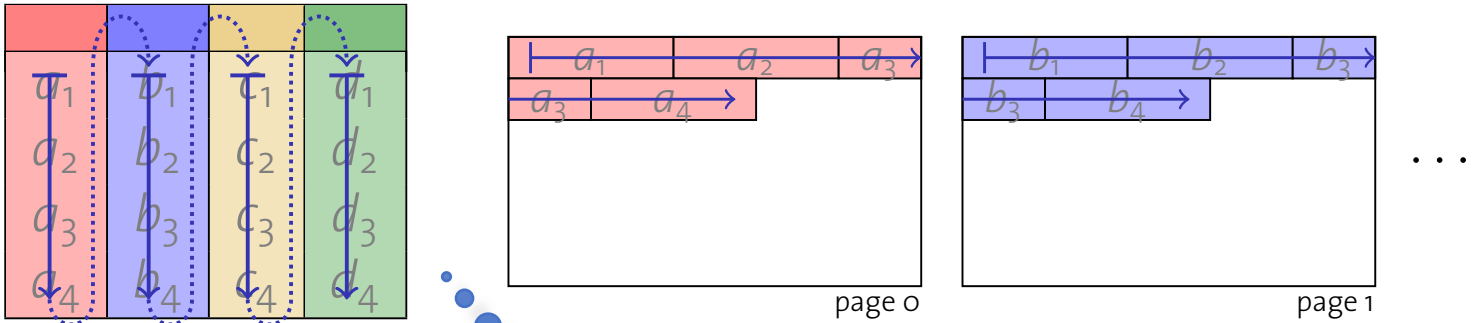


Alternative Seiteneinteilungen

- Im Beispiel wurden Datensätzen zeilenweise angeordnet:



- Spaltenweise Anordnung genauso möglich:



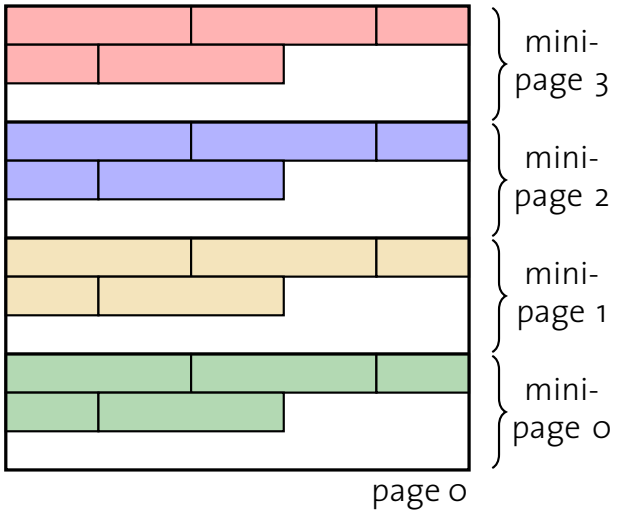
Wann lohnt sich welche Einteilung?

Alternative Seitenanordnungen

Vorgestellte Schemata heißen auch:

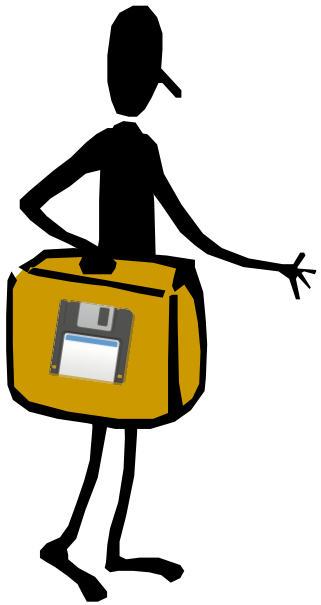
- Row-Store
- Column-Store

- Anwendungen für verschiedene Lasttypen und Anwendungs-kontexte
- Unterschiedliche Kompressionsmöglichkeiten
- Kombination möglich:
  - Unterteilung einer Seite in Miniseiten
  - Mit entsprechender Aufteilung

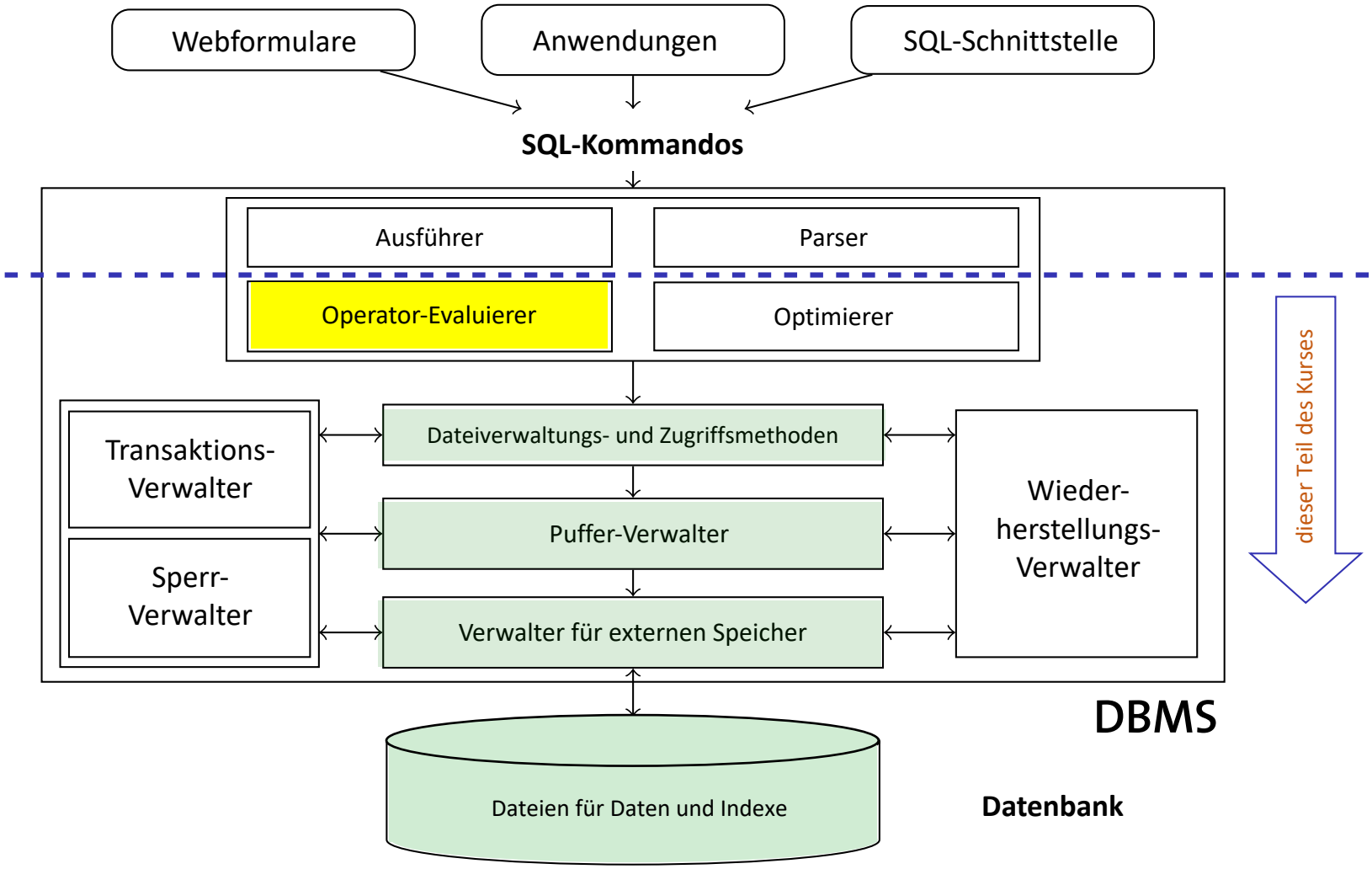


Zwischen-Rückblick

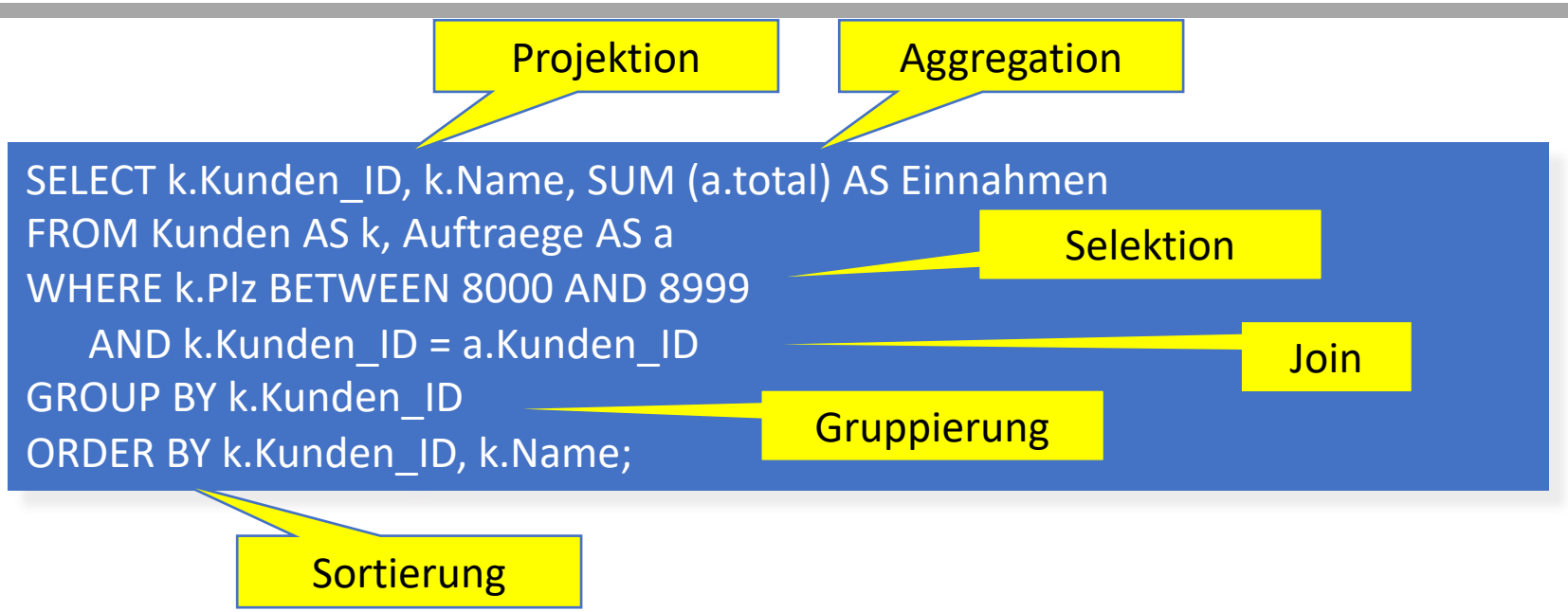
- Kennzeichen von Speichermedien
  - Wahlfreier Zugriff langsam (I/O-Komplexität)
- Verwalter für externen Speicher
  - Abstraktion von Hardware-Details
  - Seitennummer → Physikalischer Speicherort
- Puffer-Verwalter
  - Seiten-Caching im Hauptspeicher
  - Verdrängungsstrategie
- Dateiorganisation
  - Stabile Datensatz-Bezeichner (rids)
  - Verwaltung statischer und dynamischer Felder



Architektur eines DBMS



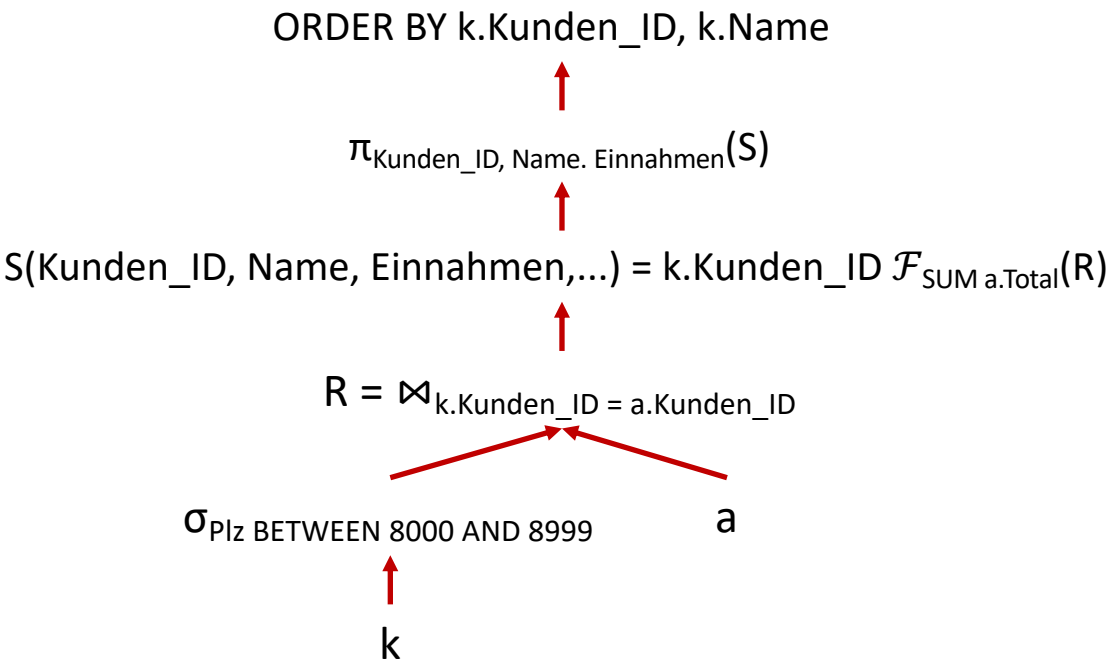
Ausführungspläne in der Anfragebeantwortung

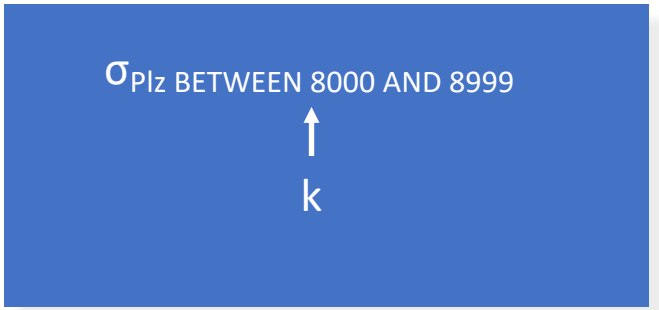


- Operatoren setzen die Operationen, die hinter Anfragen stecken, um
- Operatoren können zu Ausführungsplänen zusammengesetzt werden
- Jeder Planoperator führt zur Verarbeitung einer vollständigen Anfrage eine Unteraufgabe aus
- Nicht immer eindeutige Ausführungspläne

Ausführungspläne

```
SELECT k.Kunden_ID, k.Name, SUM (a.Total) AS Einnahmen
FROM Kunden AS k, Auftraege AS a
WHERE k.Plz BETWEEN 8000 AND 8999
    AND k.Kunden_ID = a.Kunden_ID
GROUP BY k.Kunden_ID
ORDER BY k.Kunden_ID, k.Name;
```



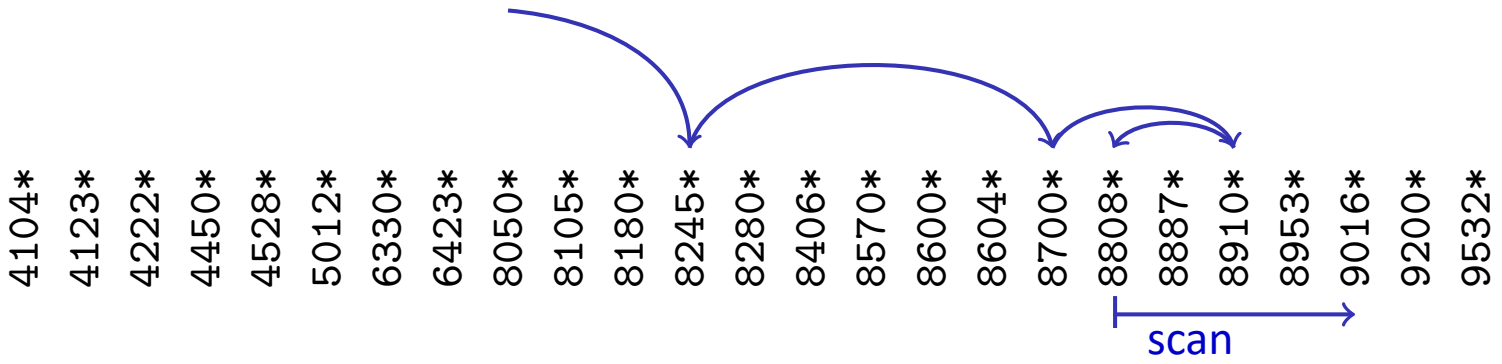


Indexierung

Operator-Evaluierer

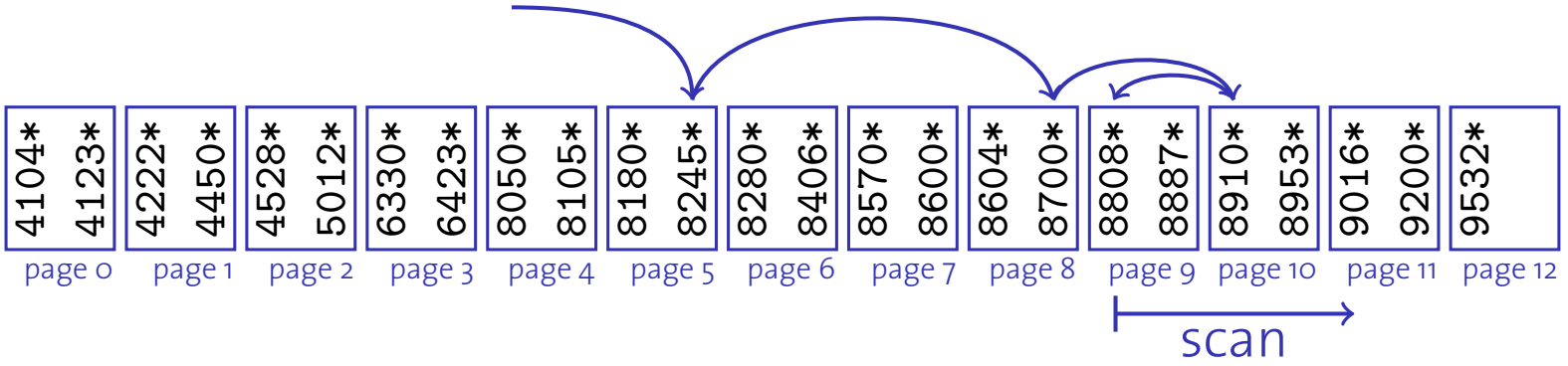
Effiziente Evaluierung einer Anfrage

- Kleineres Beispiel
  - SELECT *
    FROM Kunden
    WHERE Plz BETWEEN 8800 AND 9099;
- Auswertung
  - Sortierung der Tabelle Kunden auf der Platte (nach Plz)
    - Effizienter Sortieralgorithmus benötigt!
  - Zur Evaluierung von Anfragen Verwendung von binärer Suche, um erstes Tupel zu finden, dann Scan solange Plz < 9100



k* denotiert einen Datensatz mit Suchschlüssel k (hier Plz)

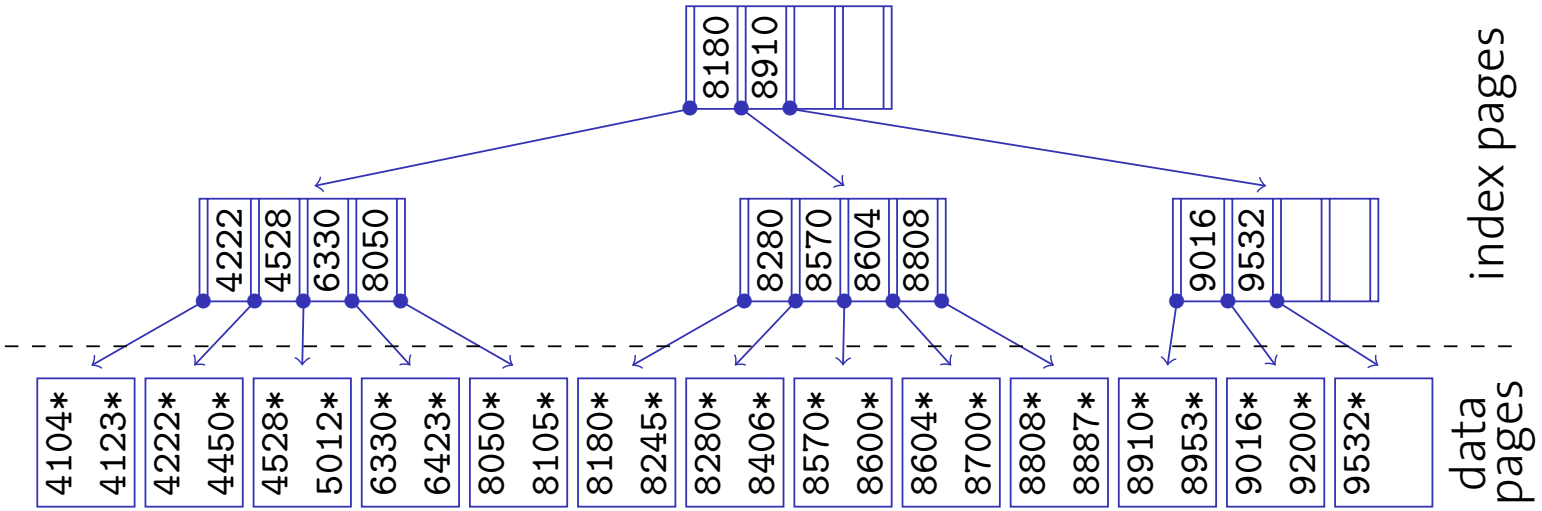
Geordnete Dateien und binäre Suche



- ✓ Sequentieller Zugriff während der Scan-Phase
- ✓ Es müssen $\log_2$(#Tupel) während der Such-Phase gelesen werden und dann die entsprechenden Tupel während der Scan-Phase (statt insgesamt alle bei unsortierten Datensätzen)
- ✗ Für jeden Zugriff während der Such-Phase eine Seite!
  - Weite Sprünge sind die Idee der binären Suche, daher Datensätze während der Suche vermutlich nicht auf einer Seite

ISAM: Indexed Sequential Access Method

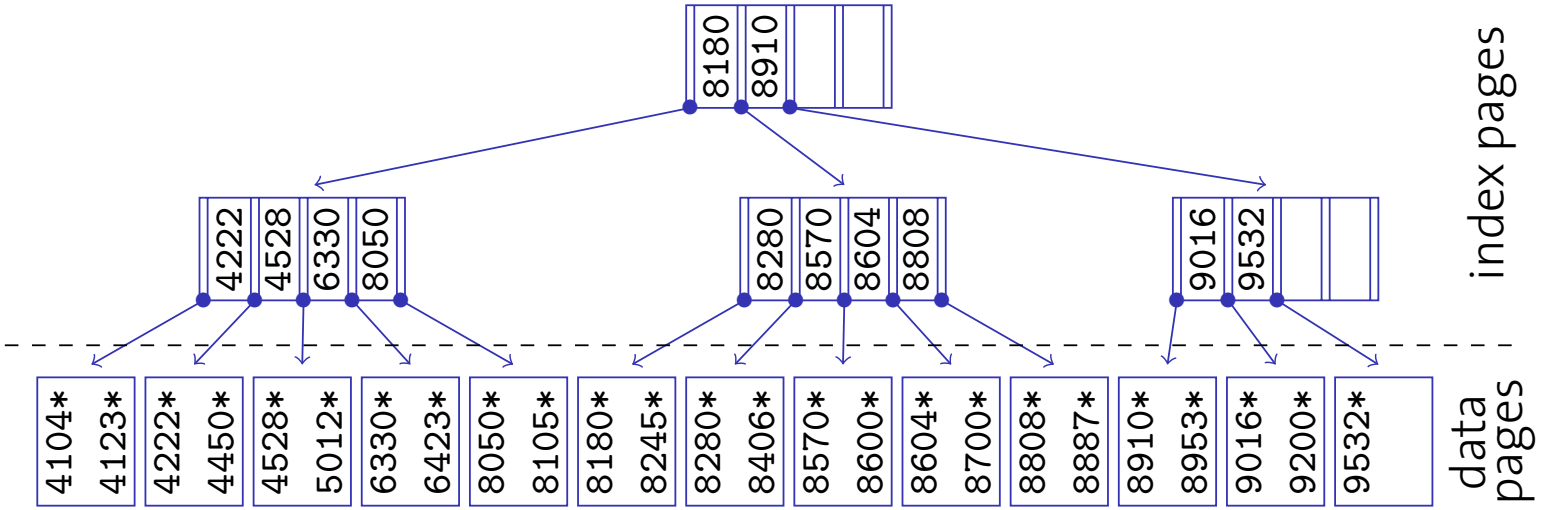
- Idee: Beschleunige die Suchphase durch sog. Index
  - Datenseiten sind, wie für die Binärsuche, sortiert gemäß Suchschlüssel



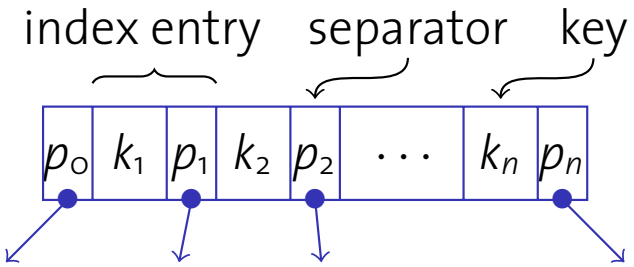
- Knoten von der Größe einer Seite
  - Hunderte Einträge pro Seite
  - Hohe Verzweigung, kleine Tiefe

ISAM: Indexeintrage

- Idee: Beschleunige die Suchphase durch sog. Index
  - Datenseiten sind, wie für die Binärsuche, sortiert gemäß Suchschlüssel

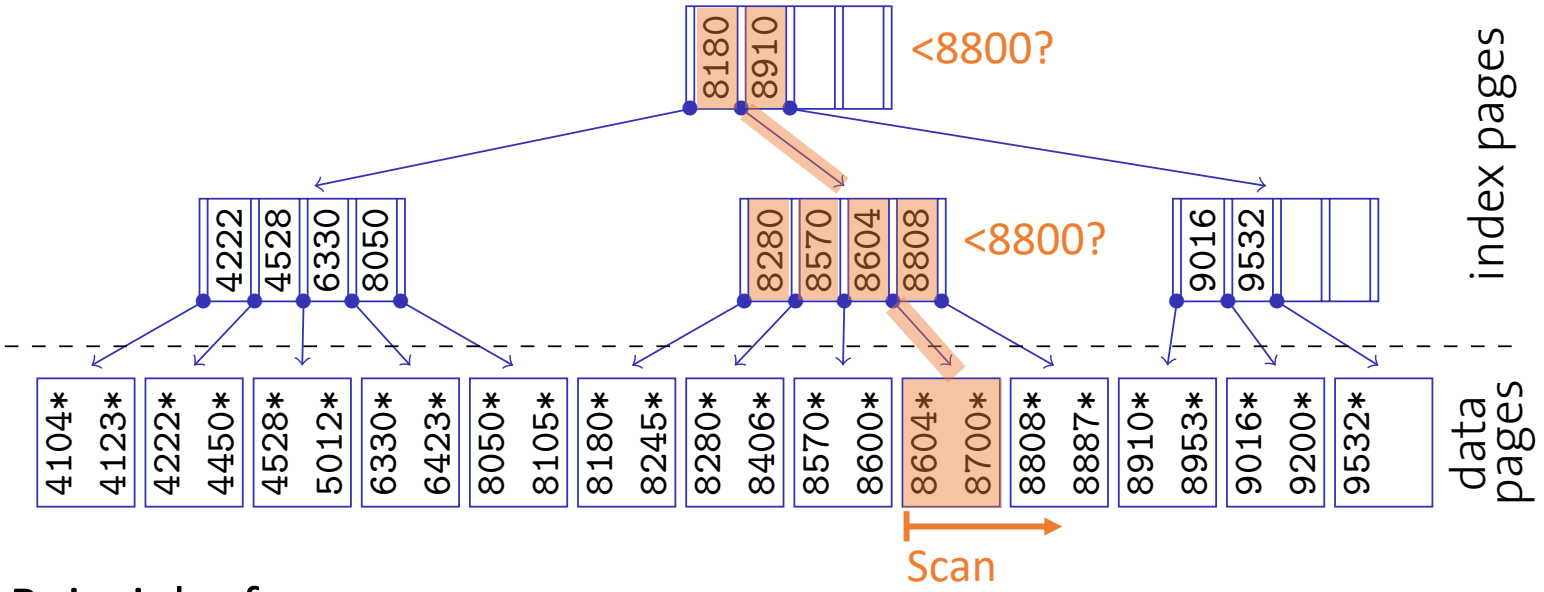


- Indexeintrag <k, p>
  - Suchschlüssel k
  - Separator p: Referenz auf Seite mit Einträgen
  - Felder feste Länge,
    Navigation mittels Versatz möglich



ISAM: Suche

- Idee: Beschleunige die Suchphase durch sog. Index
  - Datenseiten sind, wie für die Binärsuche, sortiert gemäß Suchschlüssel



- Beispielanfrage

```
SELECT *
FROM Kunden
WHERE Plz BETWEEN 8800 AND 9099;
```

  - Suche 8800 in Index, scanne ab da Seiten
  - Scanne solange, bis k > 9099

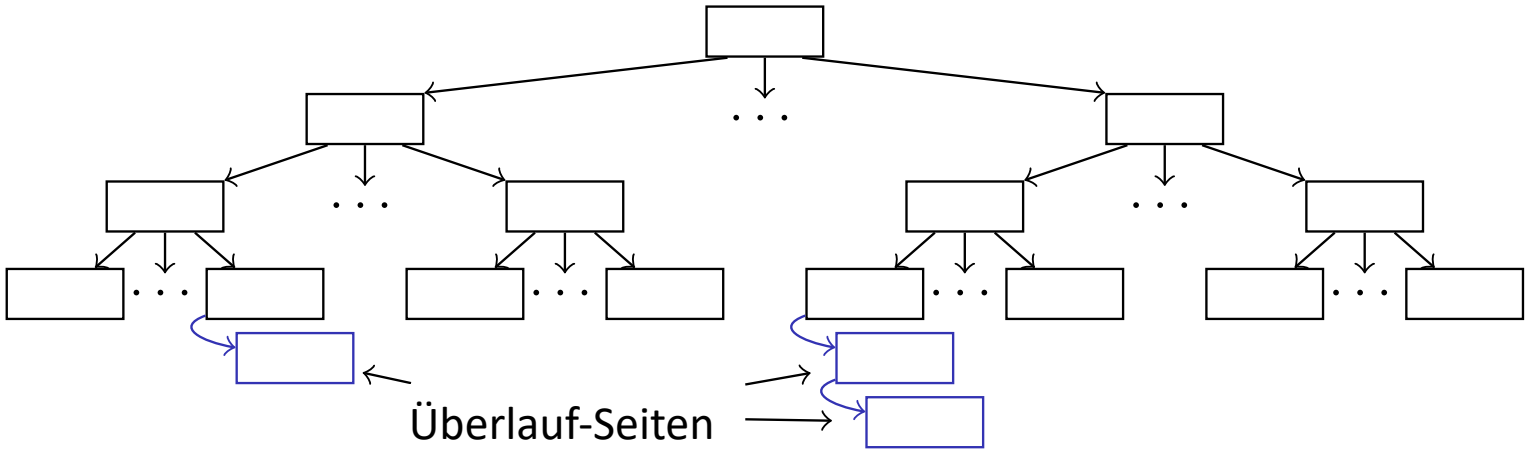
Kann ein Index bei der folgenden Anfrage helfen?

```
SELECT *
FROM Kunden k, Auftraege a
WHERE k.Name = 'IBM Corp.' AND
k.Kunden_ID=a.Kunden_ID;
```

ISAM-Index: Aktualisierungsoperationen

ISAM-Indexe sind statisch

- Löschen einfach: Lösche Datensatz von Datenseite
  - Indexstruktur bleibt gültig
- Einfügen von Daten aufwendig
  - Falls noch Platz auf Blattseite, füge Datensatz ein (z.B. nach einer vorherigen Löschung)
  - Sonst füge Überlauf-Seite ein (zerstört sequentielle Ordnung)
  - ISAM-Index degeneriert



Anmerkungen

- Freiraum bei der Indexerzeugung vorsehen
  - Reduziert das Einfügeproblem
  - Typisch sind 20% Freiraum
- Da Indexseiten statisch, keine Zugriffskoordination (bei Mehrbenutzerbetrieb) nötig
  - Zugriffskoordination (Sperren) würde gleichzeitigen Zugriff (besonders nahe der Wurzel) für andere Anfragen vermindern
- ISAM ist nützlich für (relativ) statische Daten

B^+-Bäume: Eine dynamische Indexstruktur

- B^+-Bäume vom ISAM-Index abgeleitet, sind aber dynamisch
  - Keine Überlauf-Ketten
  - Balancierung wird aufrechterhalten
  - Behandelt insert und delete angemessen
  - Indexseiten nicht statisch!

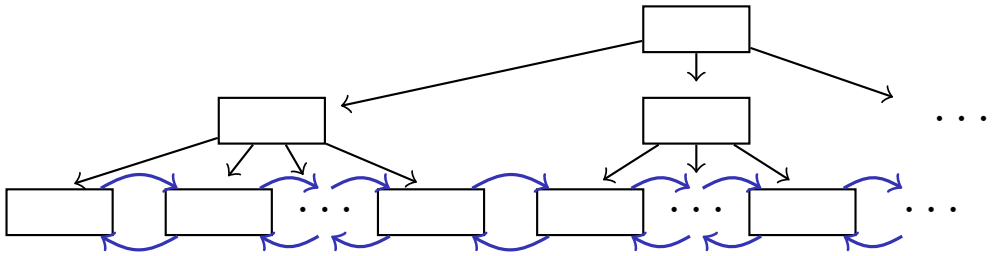
- Minimale Besetzungsregel für B^+-Baum-Knoten (außer der Wurzel): 50% (typisch sind 67%, wird dann manchmal auch B*-Baum genannt)
  - Verzweigung nicht zu klein

  vs.
  - Indexknotensuche nicht zu linear

B⁺-Bäume: Grundlagen

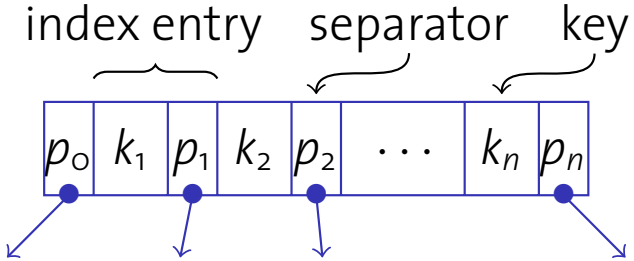
B⁺-Bäume ähnlich zu ISAM-Index, wobei

- Referenzierte Datenseiten i.d.R. nicht in sequentieller Ordnung
- Blätter zu doppelt verketteter Liste verbunden



- Jeder Knoten enthält zwischen d und 2d Einträge (d heißt Ordnung des Baumes, Wurzel ist Ausnahme)

- Es gilt weiterhin für die Indexblätter (Nicht-Blätter): Eintrag <k, p>



Was wird in den Blättern gespeichert?

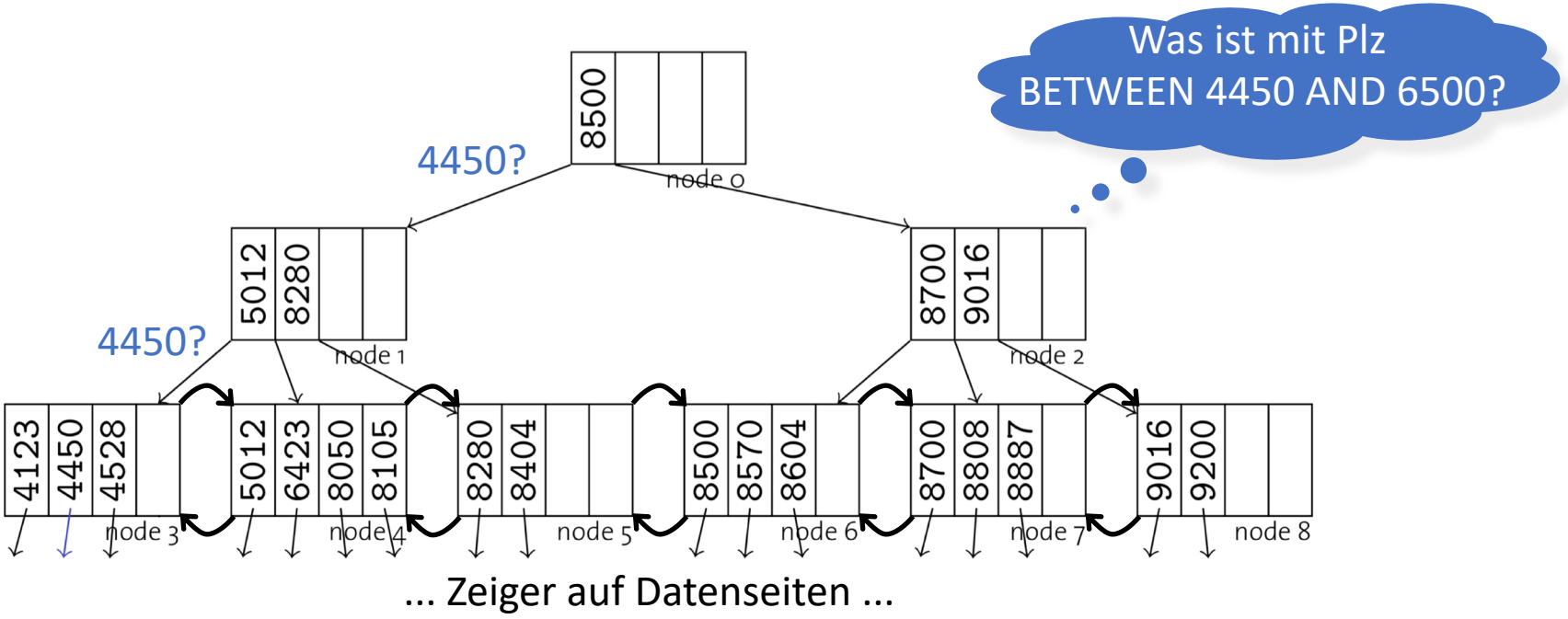
- Drei Alternativen
  1. Vollständiger Datensatz k*:
     - Blatt ist Datenseite (wie bei ISAM Index)

  2. Ein Paar <k, rid>, wobei rid (record ID) Zeiger auf Datensatz:
     - Blatt enthält Liste von <k, rid> Paaren, sortiert nach k
     - Suchschlüssel k kann auf Blatt häufiger auftreten (mehrere rid's mit Suchschlüssel k)
     - Bei rid=<pageno, slotno> lässt sich der genaue Speicherort des Datensatzes bestimmen

  3. Ein Paar <k, {rid_1, rid_2, ... }>, wobei alle rid's den Suchschlüssel k haben
     - Suchschlüssel k tritt nur einmal auf (weniger Redundanz)
     - Aber: kein regelmäßiger Abstand von einem <k, rid> zum nächsten

- Varianten 2. und 3. bedingen, dass rid's stabil sein müssen, also nicht (einfach) verschoben werden können
- Alternative 2 scheint am meisten verwendet zu werden
- Wir nehmen im Folgenden Variante 2 an

Suche: Beispiel



- Suche von Einträgen mit Schlüssel 4450
  - Schlüssel führt zu Blattknoten 3
  - Anschließend: Scan des Blattknotens nach Suchschlüssel 4450
    - Erinnerung: Liste von <k, rid> Paaren
    - Suche nach <4450, rid> in Liste → von rid zur Datenseite

Wie?

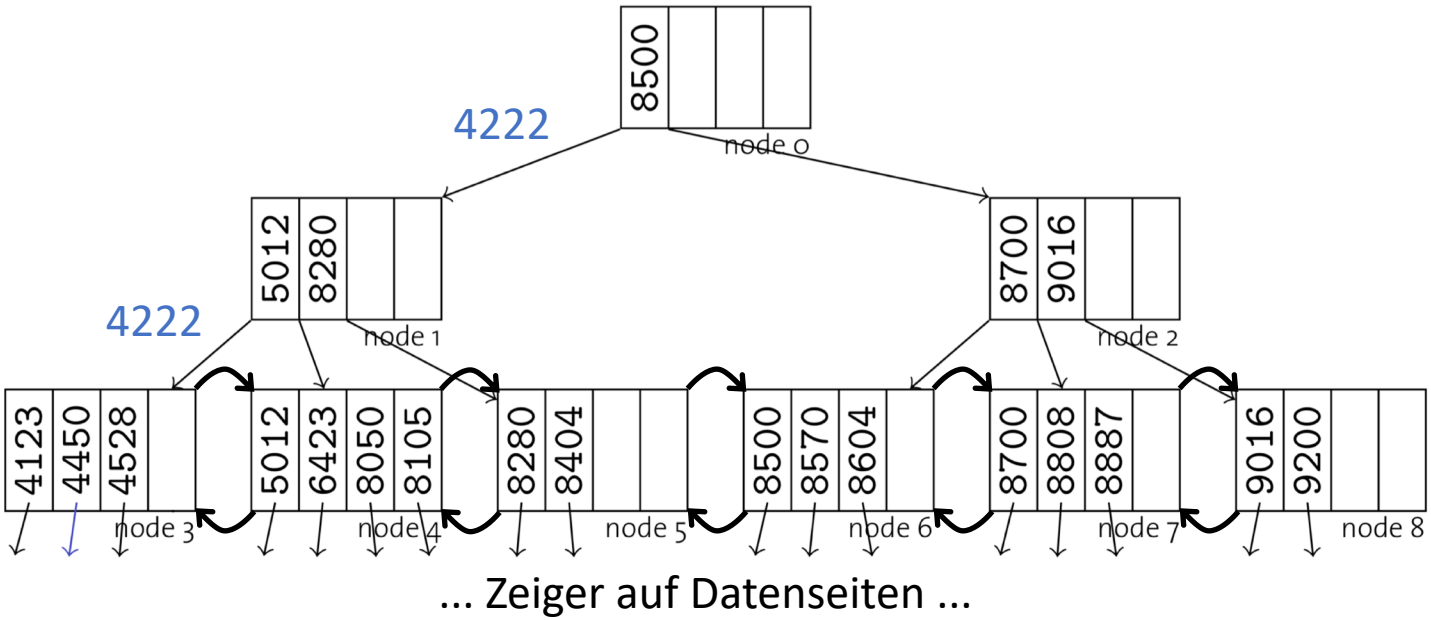
Insert: Überblick

- B^+-Baum soll nach Einfügung balanciert bleiben
  - Keine Überlauf-Seiten
- Algorithmus für insert(k, rid) mit Eingaben k und rid:
  1. Finde Blattseite n, in der Eintrag für k sein kann
  2. Falls n genug Platz hat (höchstens 2d-1 Einträge), füge Eintrag <k, rid> in n ein
  3. Sonst muss n aufgeteilt werden in n und n‘ – des weiteren muss ein Separator in den Elternknoten e von n eingefügt werden
     - Wenn e keinen Platz mehr hat (2d Einträge), dann muss e ebenfalls aufgeteilt werden

→ Aufspaltung kann sich rekursiv nach oben fortsetzen, eventuell bis zur Wurzel

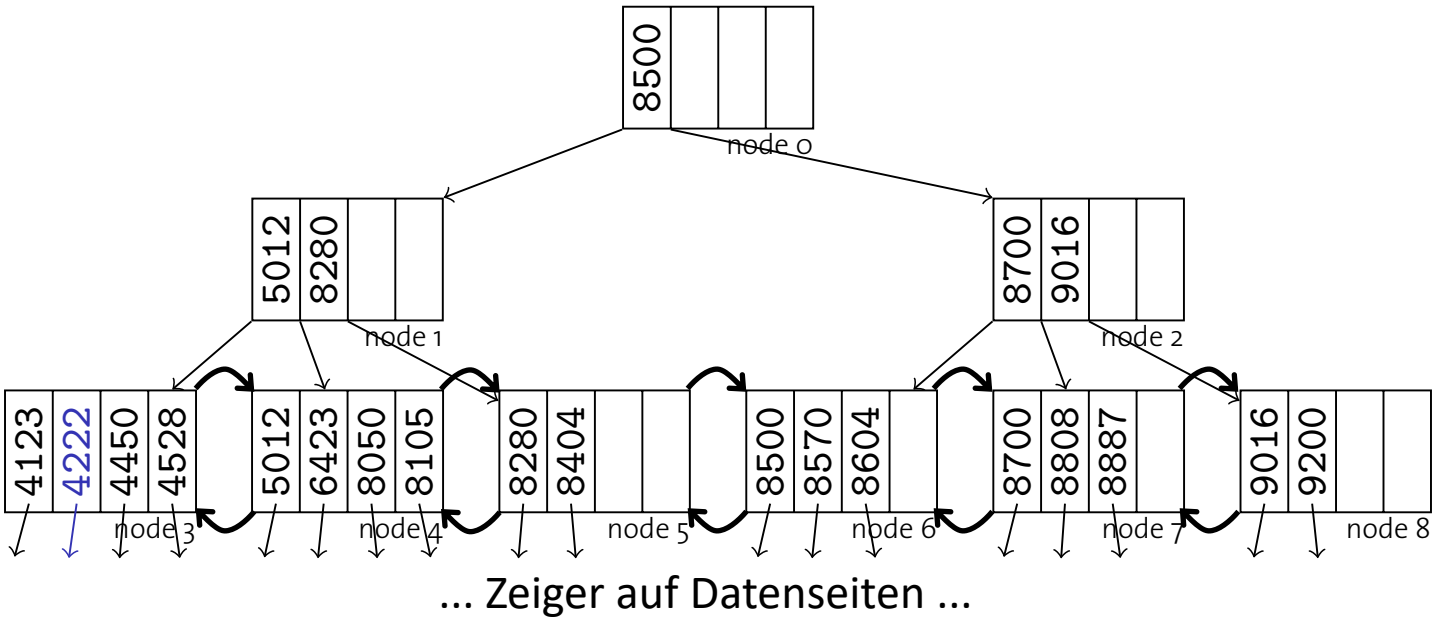
- Wenn die Wurzel aufgeteilt wird, wird ein neuer Wurzelknoten als Elternknoten eingeführt (Baum erhöht sich)
- Wurzel kann unter 50% gefüllt sein

Insert: Beispiel ohne Aufspaltung



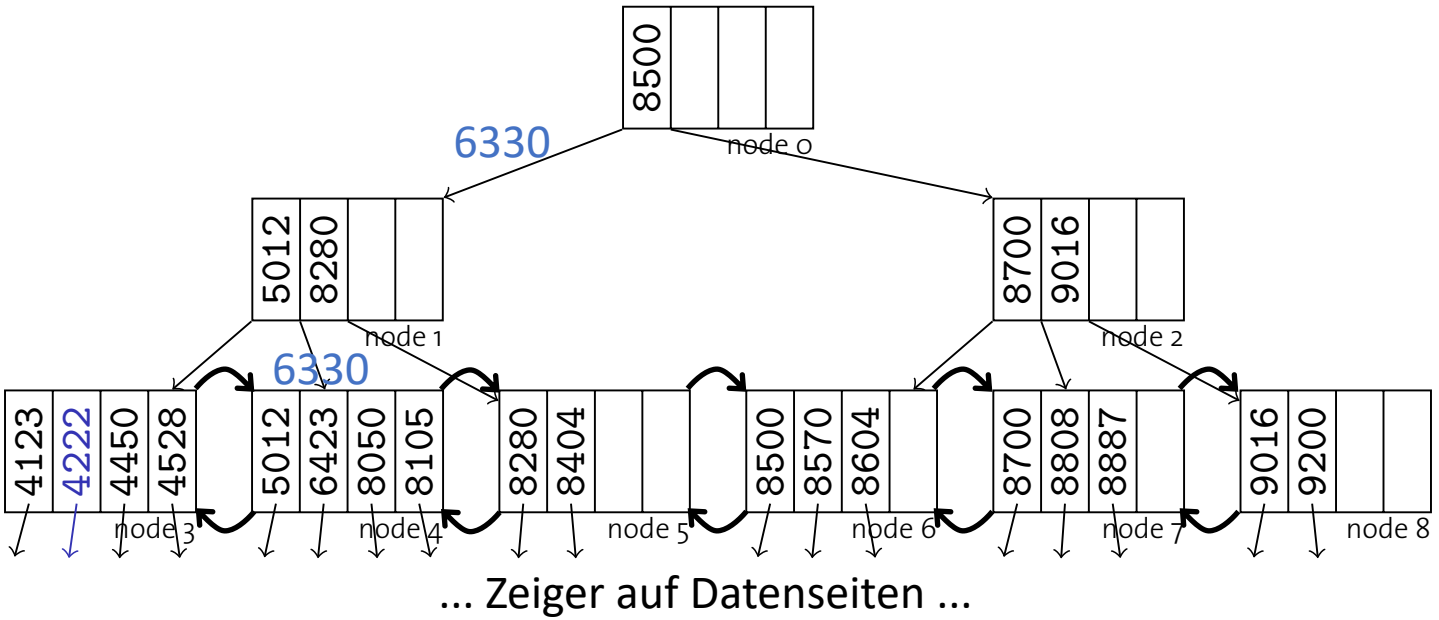
- Einfügung eines Eintrags mit Schlüssel 4222
  - Es ist genug Platz in Knoten 3 (3 < 2d-1, d = 2), einfach einfügen
  - Erhalte Sortierung innerhalb der Knoten

Insert: Beispiel ohne Aufspaltung

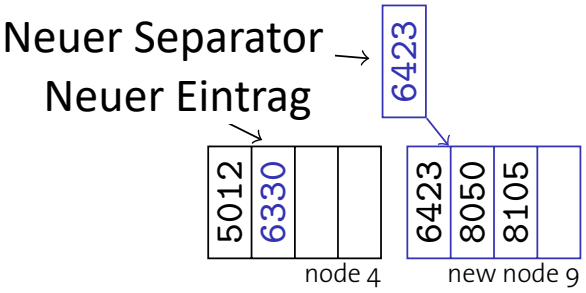


- Einfügung eines Eintrags mit Schlüssel 4222
  - Es ist genug Platz in Knoten 3 (3 < 2d-1, d = 2), einfach einfügen
  - Erhalte Sortierung innerhalb der Knoten

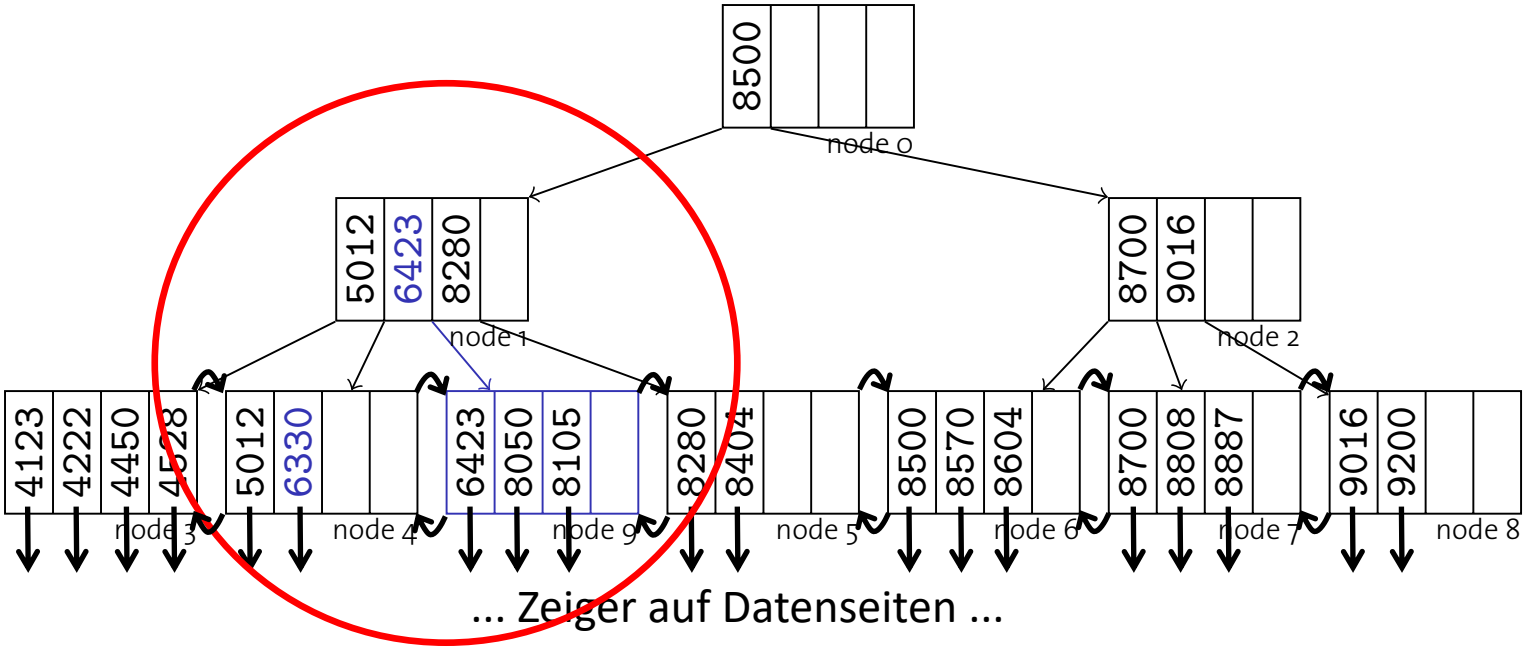
Insert: Beispiel mit Aufspaltung



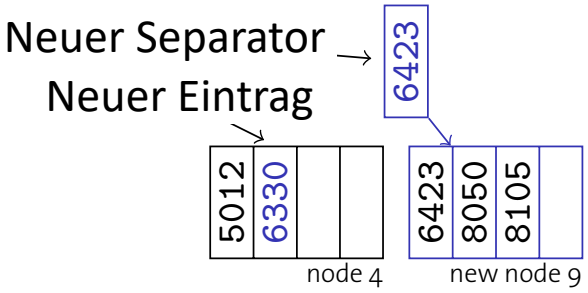
- Einfügung eines Eintrags mit Schlüssel 6330
  - Knoten 4 aufspalten
    - Erste Hälfte - 1 bleibt in Knoten 4
    - (Zweite Hälfte + 1 geht in neuen Knoten 9)
  - Neuer Separator in Knoten 1



Insert: Beispiel mit Aufspaltung

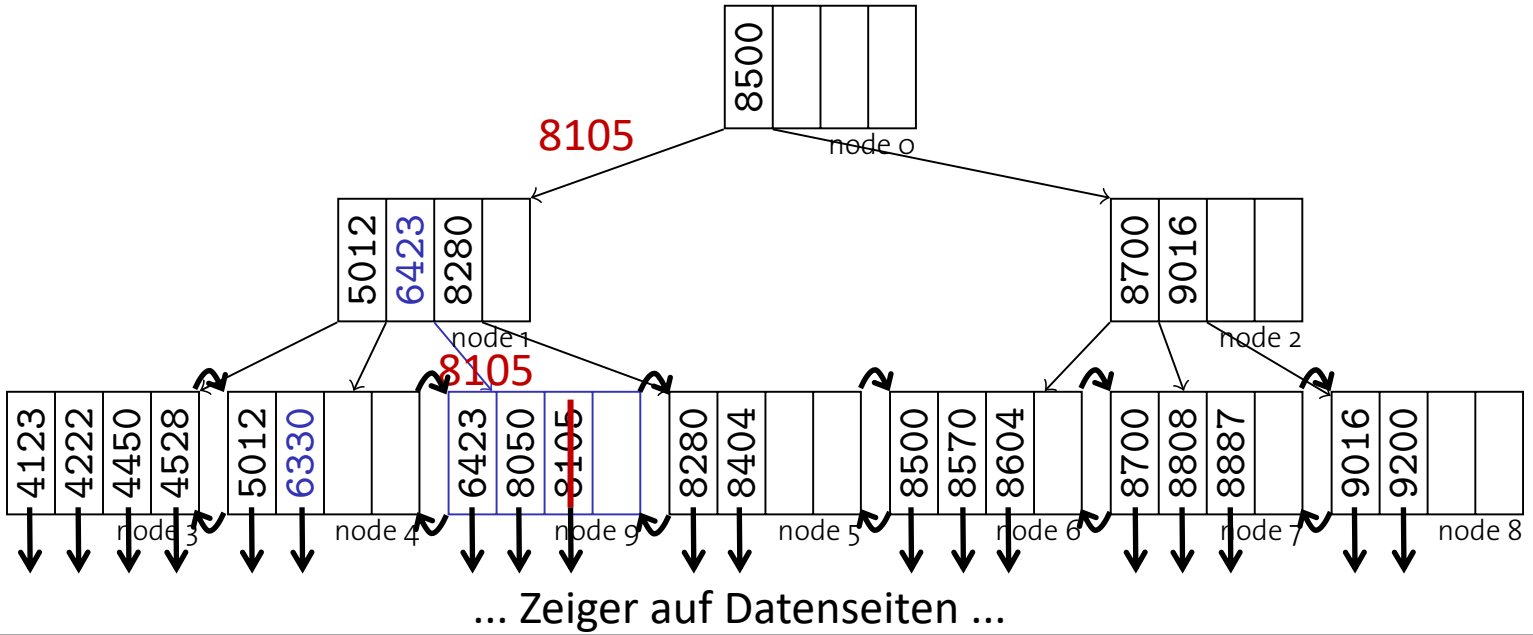


- Einfügung eines Eintrags mit Schlüssel 6330
  - Knoten 4 aufspalten
    - Erste Hälfe - 1 bleibt in Knoten 4
    - (Zweite Hälfte + 1 geht in neuen Knoten 9)
  - Neuer Separator in Knoten 1



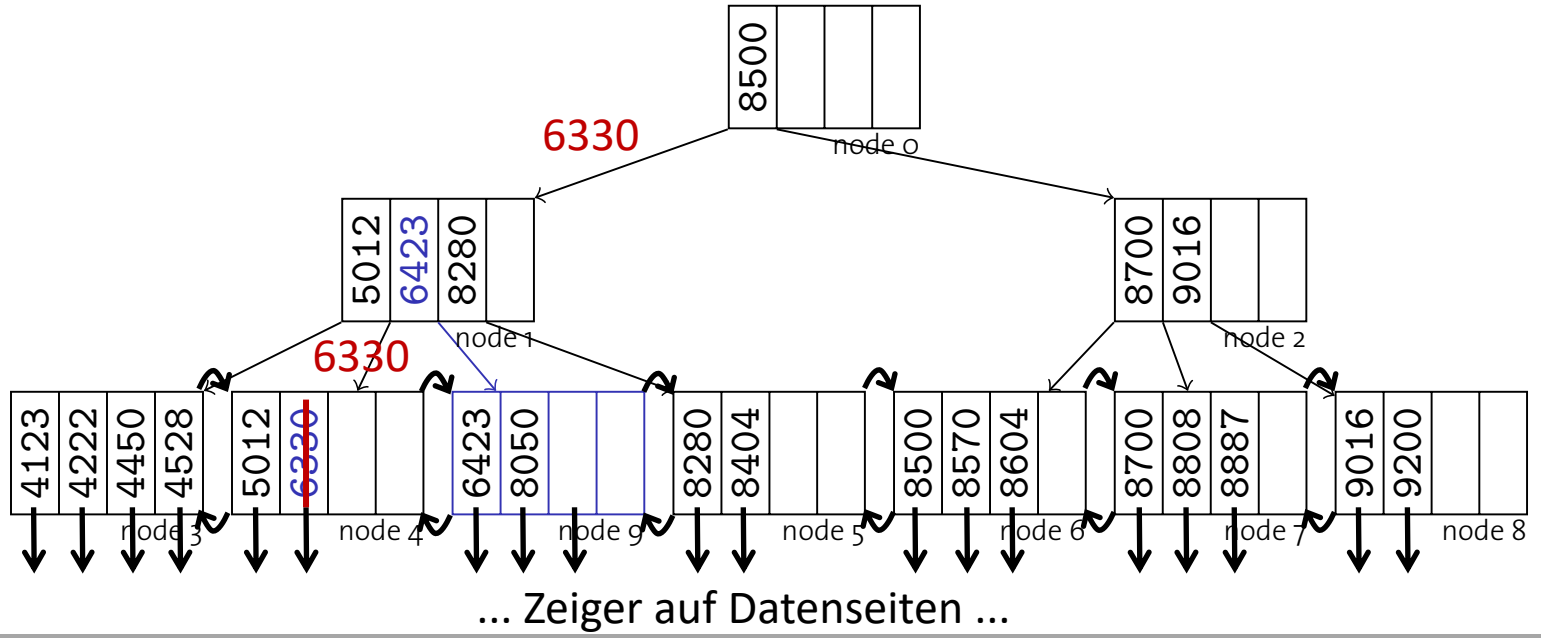
Löschung in Blattknoten

- B$^+$-Baum soll nach Löschung balanciert bleiben
- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  1. Finde Blattseite l, in der Eintrag für <k, rid> steht
  2. Falls l genügend gefüllt (mindestens d+1 Einträge), Eintrag einfach löschen
     - Hinterher können innere Knoten Schlüssel enthalten, die zu Einträgen gehören, die nicht mehr existieren. Das ist OK.



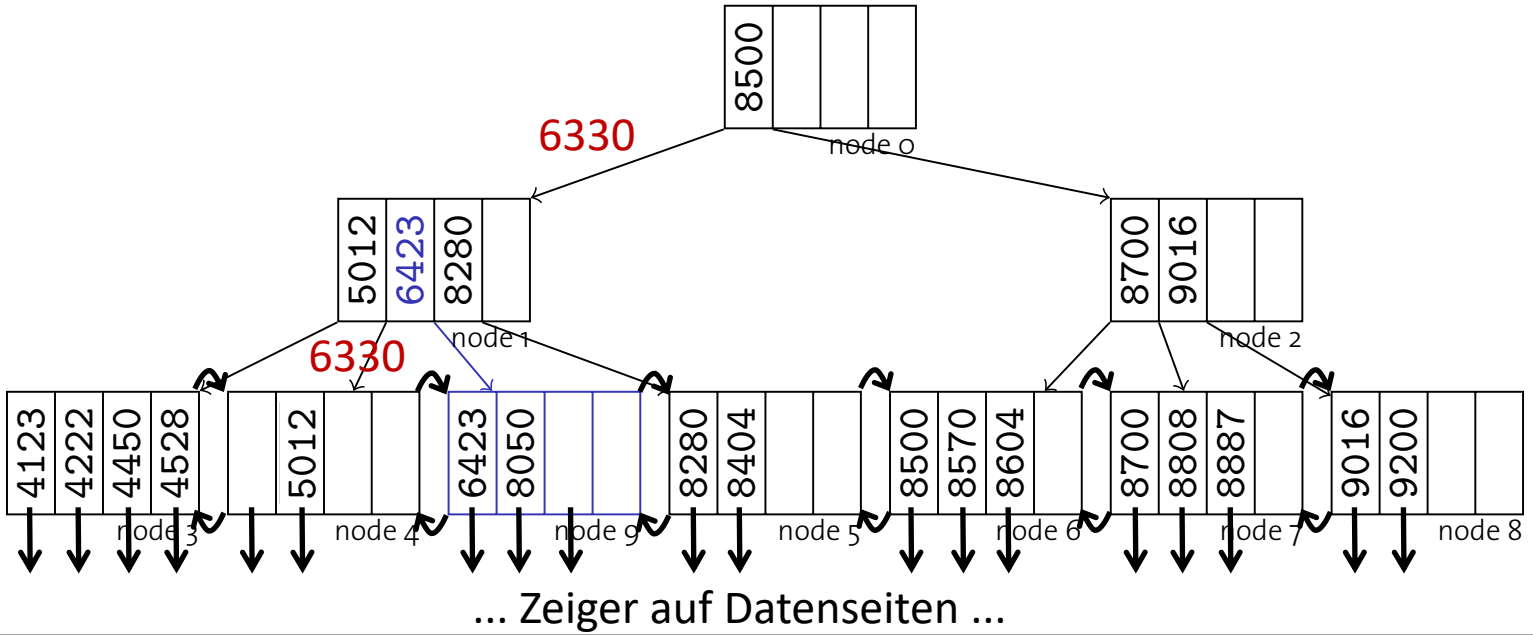
Löschung in Blattknoten

- B^+-Baum soll nach Löschung balanciert bleiben
- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  1. Finde Blattseite l, in der Eintrag für <k, rid> steht
  2. Falls l genügend gefüllt (mindestens d+1 Einträge), Eintrag einfach löschen
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
        Mit Eintrag aus n auffüllen und Separator im Elternknoten aktualisieren



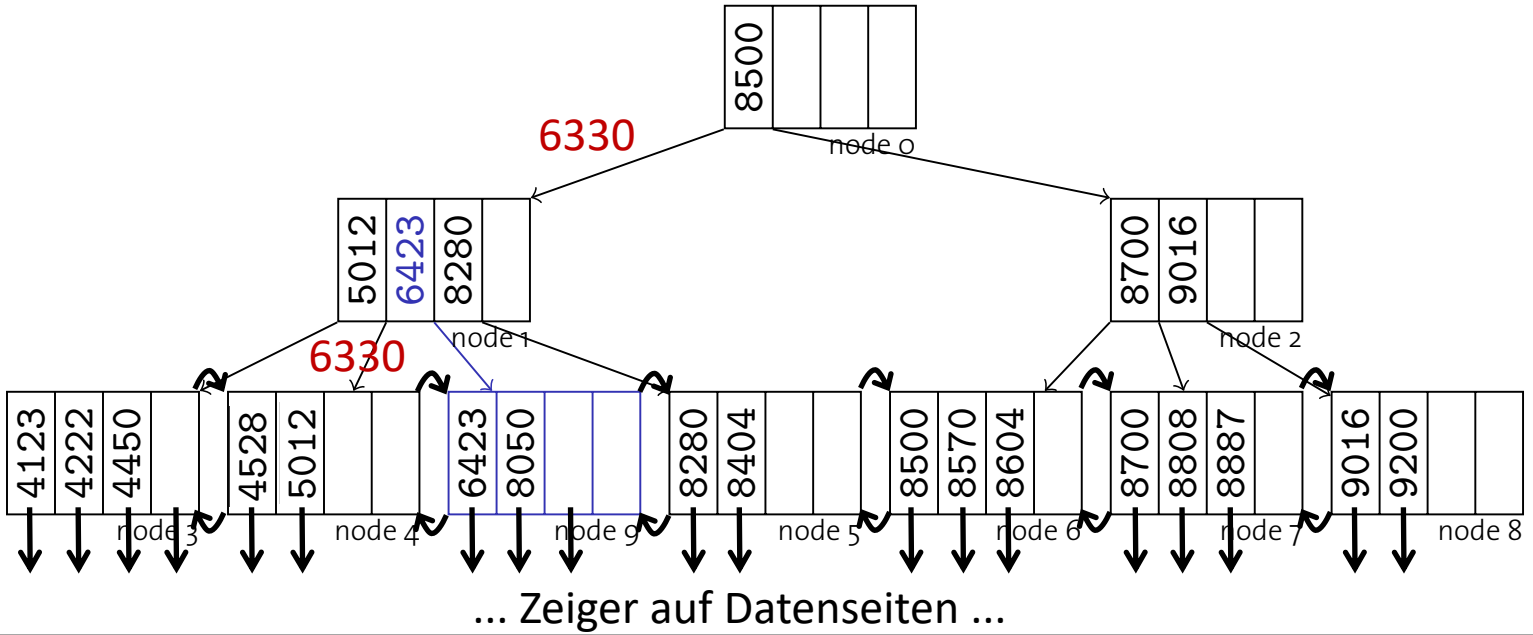
Löschung in Blattknoten

- B^+-Baum soll nach Löschung balanciert bleiben
- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  1. Finde Blattseite l, in der Eintrag für <k, rid> steht
  2. Falls l genügend gefüllt (mindestens d+1 Einträge), Eintrag einfach löschen
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
     Mit Eintrag aus n auffüllen und Separator im Elternknoten aktualisieren



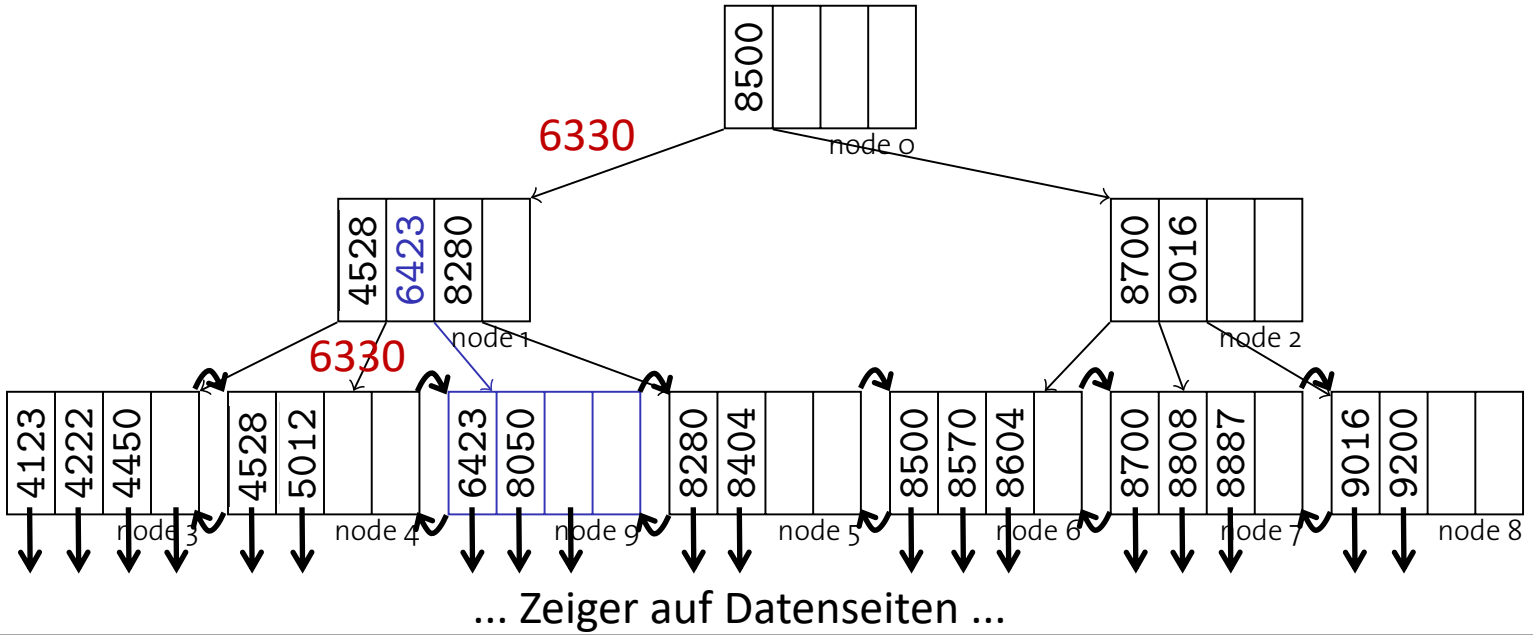
Löschung in Blattknoten

- B^+-Baum soll nach Löschung balanciert bleiben
- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  1. Finde Blattseite l, in der Eintrag für <k, rid> steht
  2. Falls l genügend gefüllt (mindestens d+1 Einträge), Eintrag einfach löschen
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
     Mit Eintrag aus n auffüllen und Separator im Elternknoten aktualisieren



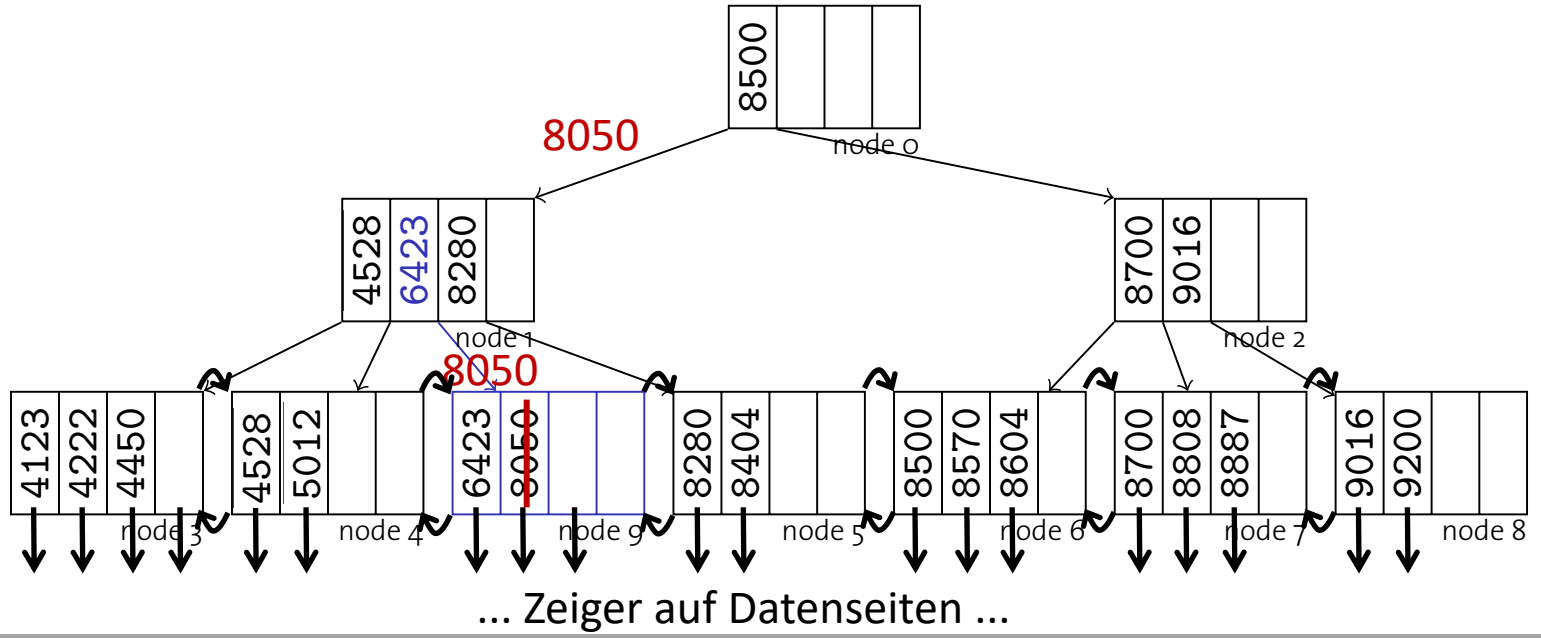
Löschung: Datensatz

- B^+-Baum soll nach Löschung balanciert bleiben
- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  1. Finde Blattseite l, in der Eintrag für <k, rid> steht
  2. Falls l genügend gefüllt (mindestens d+1 Einträge), Eintrag einfach löschen
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
        Mit Eintrag aus n auffüllen und Separator im Elternknoten aktualisieren



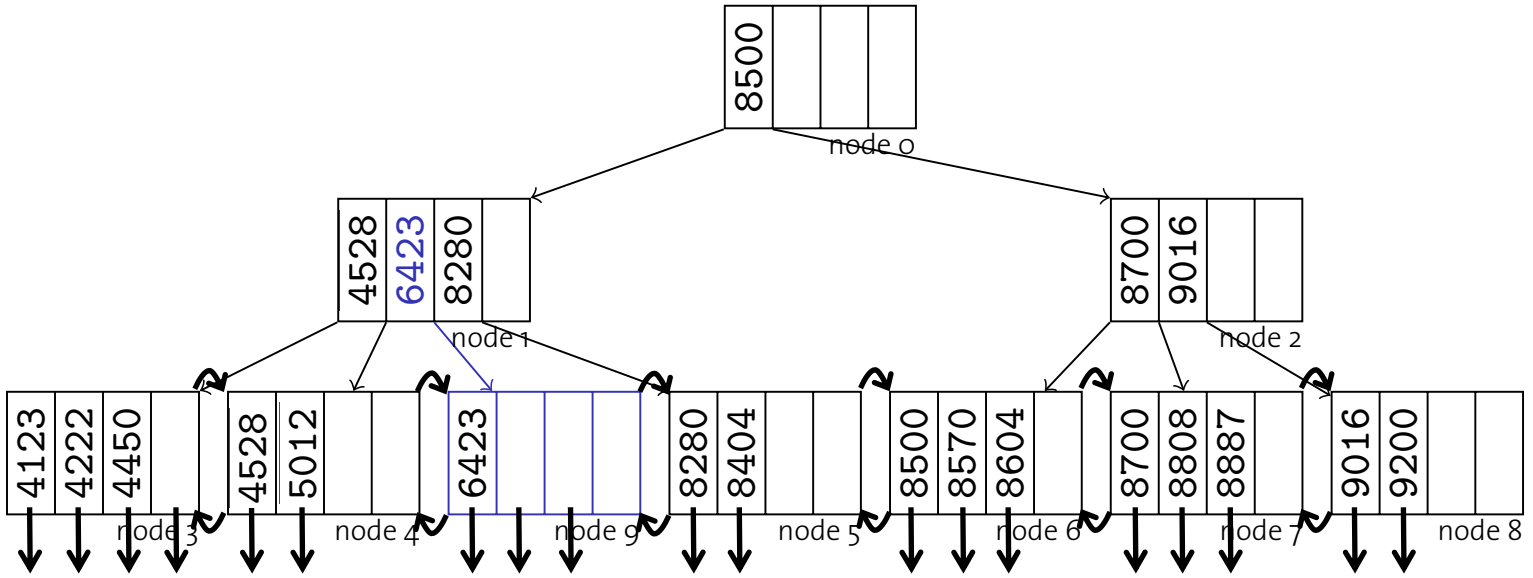
Löschung: Datensatz

- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
        Mit Eintrag aus n auffüllen und Separator aktualisieren
     b) Sonst (Nachbarblätter haben auch nur d Einträge):



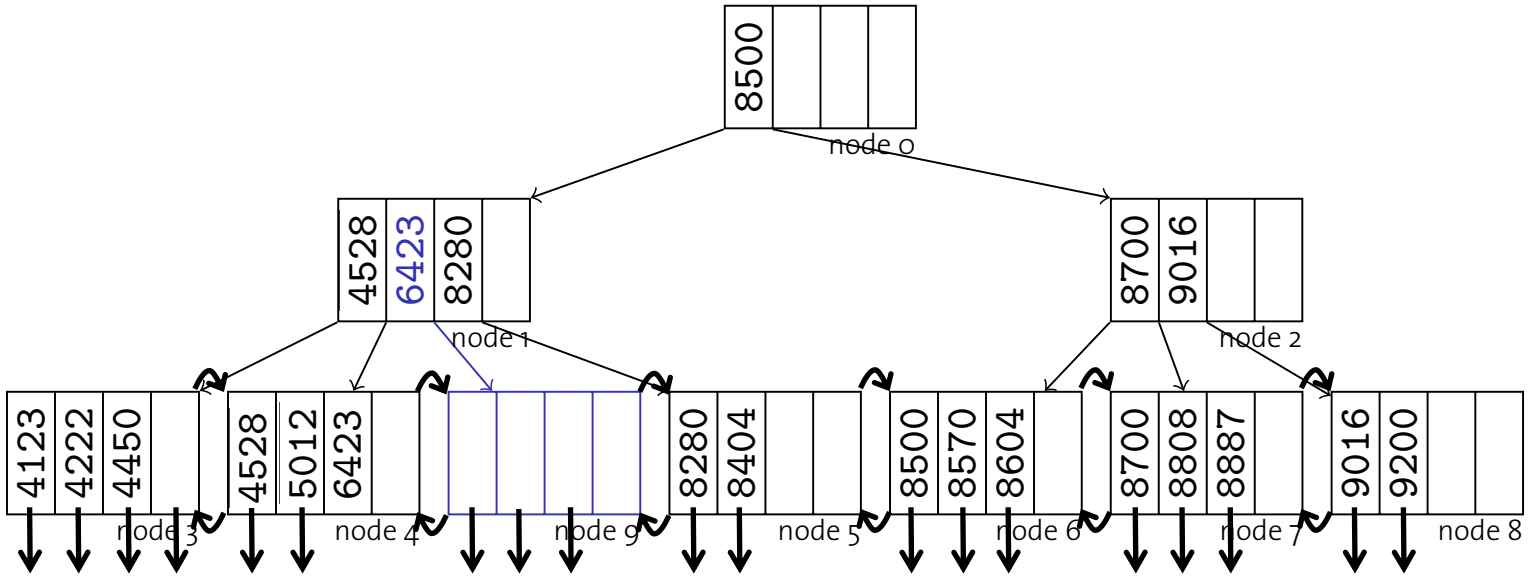
Löschung: Datensatz

- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
        Mit Eintrag aus n auffüllen und Separator aktualisieren
     b) Sonst (Nachbarblätter haben auch nur d Einträge):
        Nachbarblatt n und l verschmelzen und Separator löschen



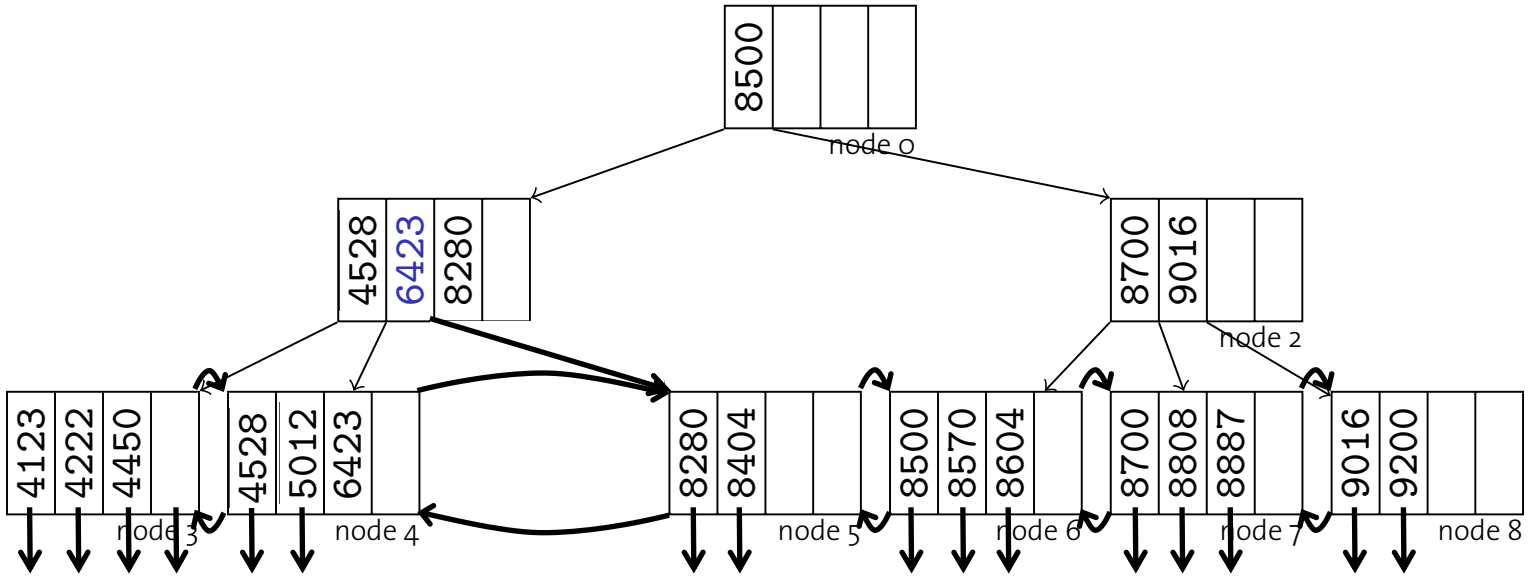
Löschung: Datensatz

- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
        Mit Eintrag aus n auffüllen und Separator aktualisieren
     b) Sonst (Nachbarblätter haben auch nur d Einträge):
        Nachbarblatt n und l verschmelzen und Separator löschen



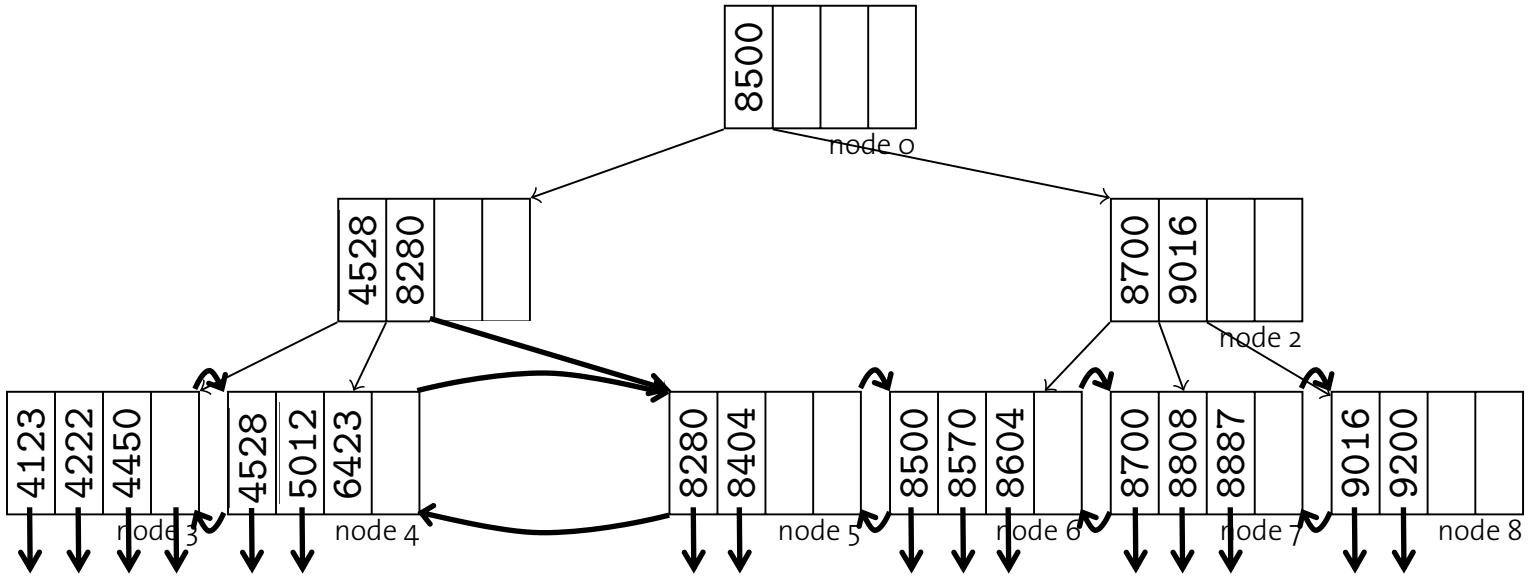
Löschung: Datensatz

- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
        Mit Eintrag aus n auffüllen und Separator aktualisieren
     b) Sonst (Nachbarblätter haben auch nur d Einträge):
        Nachbarblatt n und l verschmelzen und Separator löschen



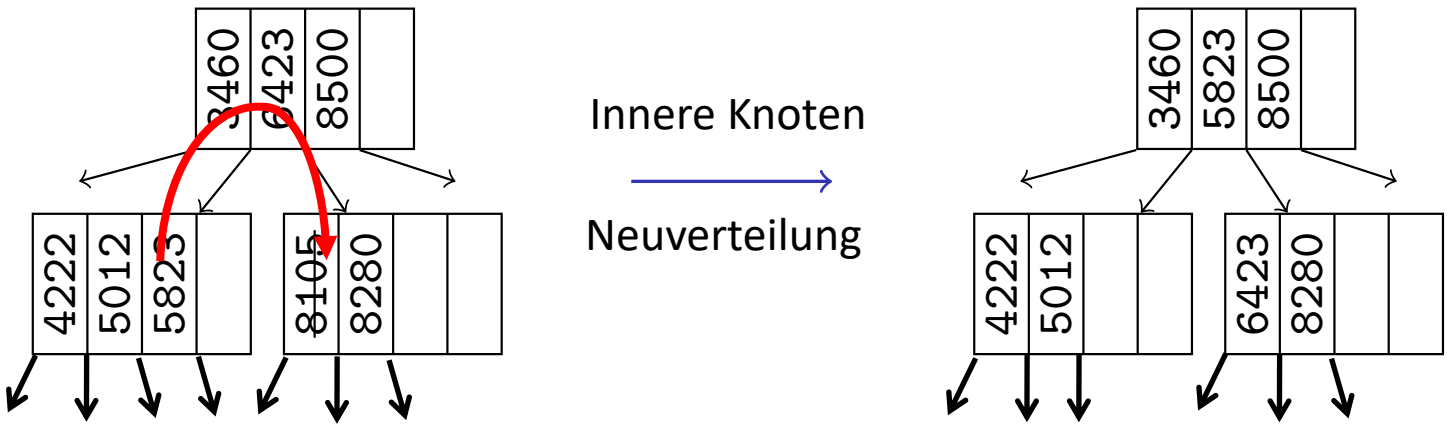
Löschung: Datensatz

- Algorithmus für delete(k, rid) mit Eingaben k und rid:
  3. Sonst (d Einträge):
     a) Wenn Nachbarblatt n genügend Einträge (>d) hat:
        Mit Eintrag aus n auffüllen und Separator aktualisieren
     b) Sonst (Nachbarblätter haben auch nur d Einträge):
        Nachbarblatt n und l verschmelzen und Separator löschen
     - Separator löschen kann zu unterbesetzten inneren Knoten führen, dann sind weitere Umstrukturierungen nötig



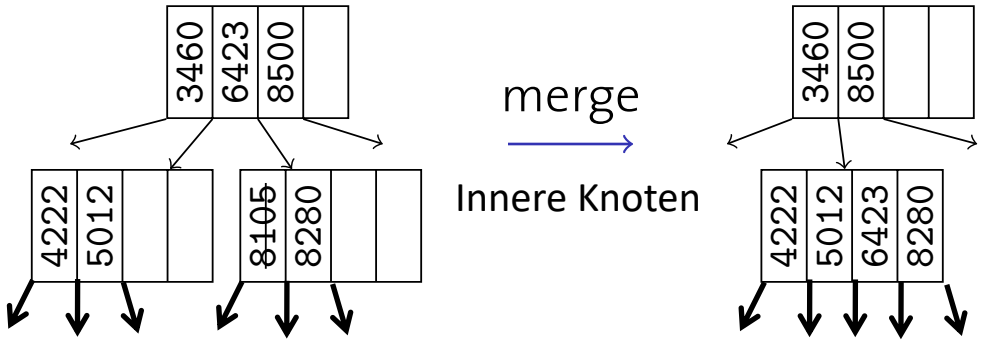
Löschung: Separator

- Löschung von einem Separator k in Knoten n (ähnlich zu vorher):
  1. Wenn n genügend Einträge (>d) hat, dann k einfach löschen
  2. Sonst: (Umverteilen oder verschmelzen)
     a) Wenn Nachbarknoten genügend Einträge (>d) hat, umverteilen:
        - Rotiere Eintrag über den Elternknoten:
          i. Entsprechender Eintrag aus Nachbarknoten geht als neuer Separator in den Elternknoten
          ii. Separator aus Elternknoten geht als neuer Eintrag in den unterbesetzten Knoten



Löschung: Separator

- Löschung von einem Separator k in Knoten n (ähnlich zu vorher):
  1. Wenn n genügend Einträge (>d) hat, dann k einfach löschen
  2. Sonst: (Umverteilen oder verschmelzen)
     a) Wenn Nachbarknoten genügend Einträge (>d) hat, umverteilen:
        - Rotiere Eintrag über den Elternknoten
     b) Sonst (d Einträge der Nachbarknoten):
        - Verschmelze Knoten
        - Ziehe Separator in den verschmolzenen Knoten



B^+-Bäume in realen Systemen

- Implementierungen verzichten auf die Kosten der Verschmelzung und der Neuverteilung und weichen die Regel der Minimumbelegung auf
- Beispiel: IBM DB2 UDB
  - MIN**PCT**USED als Parameter zur Steuerung der Blattknotenverschmelzung (Online-Indexreorganisation)
  - Innere Knoten werden niemals verschmolzen (nur bei Reorganisation der gesamten Tabelle)
- Zur Verbesserung der Nebenläufigkeit evtl. nur Markierung von Knoten als gelöscht (keine aufwendige Neuverzeigerung)

PCT = Partition Change Tracking

Erzeugung von Indexstrukturen in SQL

- Implizite Indexe
  - Indexe automatisch erzeugt für Primärschlüssel und Unique-Integritätsbedingungen
    - Können z.B. bei Einfügeoperationen genutzt werden für effiziente Prüfung, ob Wert schon existiert
    - Schnelle Findung von Einträgen anhand des Primärschlüssels
- Explizite Indexe (Wissen über häufige Anfragen nutzen!)
  - Einfache Indexe:

```
CREATE INDEX name
ON table_name(attr) ;
```

Beispiel

```
CREATE INDEX PlzIndex
ON Kunden(Plz) ;
```

```
SELECT *
FROM Kunden
WHERE Plz BETWEEN 8800 AND 8999 ;
```

  - Zusammengesetzte Indexe (Verbundindexe):

```
CREATE INDEX name
ON table_name(attr1, attr2, ..., attrn) ;
```

Indexe mit zusammengesetzten Schlüsseln

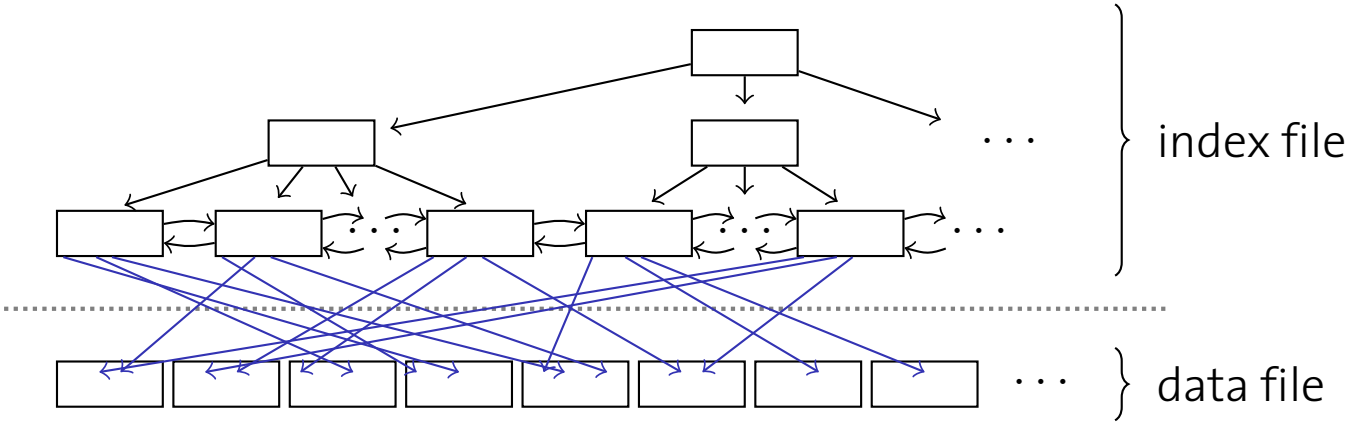
- B^+-Bäume können verwendet werden, um Dinge mit einer definierten **totalen Ordnung** zu indizieren (im Prinzip[1])
  - Integer, Zeichenketten, Datumsangaben, ...
  - und auch eine Hintereinandersetzung davon (basierend auf einer lexikographischen Ordnung)
- Beispiel:

```
CREATE INDEX idx_nachname_vorname
ON Kunden(Nachname, Vorname);
```

[1] In einigen Implementierungen können lange Zeichenketten nicht als Index verwendet werden

B+-Bäume und Sortierung

- Eine typische Situation mit <k, rid> Paaren in Blättern sieht so aus:

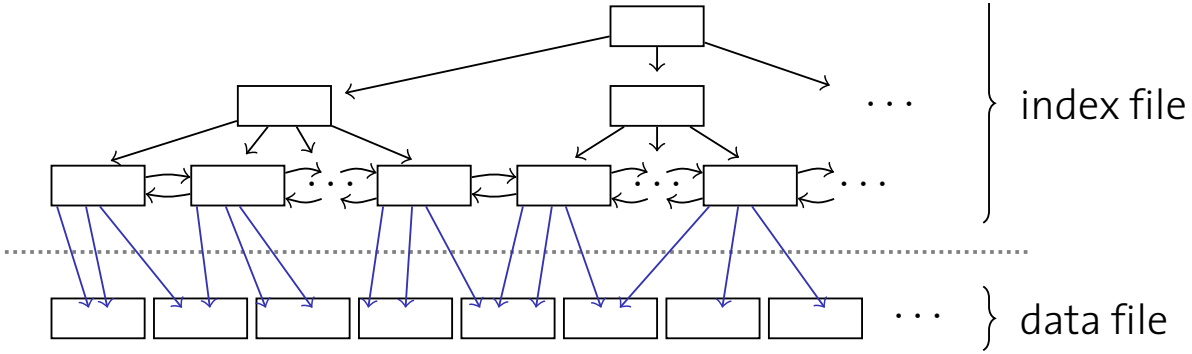


- Was passiert, wenn man Folgendes ausführt?

```
SELECT *
FROM Kunden
WHERE Plz BETWEEN 8800 AND 9099;
ORDER BY Plz;
```

Geclusterte B$^+$-Bäume

- Wenn die Datei mit den Datensätzen sortiert und sequentiell gespeichert ist, erfolgt der Zugriff schneller



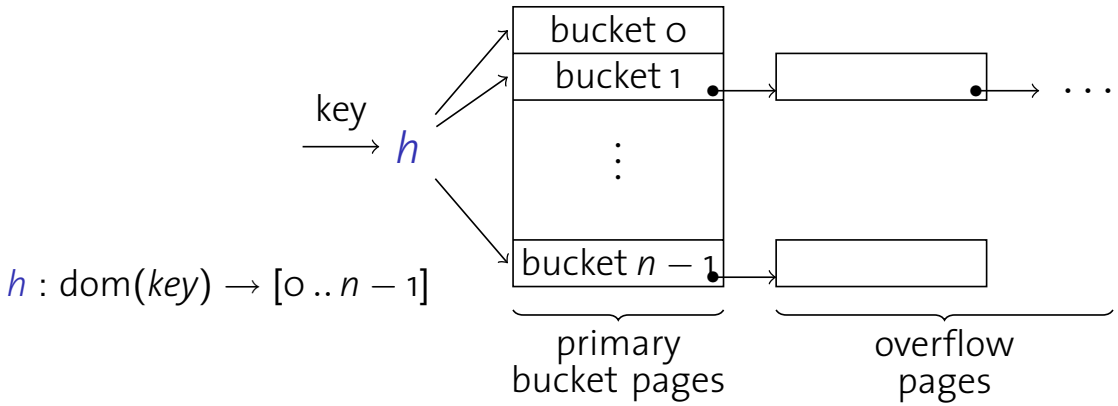

```
SELECT *
FROM Kunden
WHERE Plz BETWEEN 8800 AND 9099;
ORDER BY Plz;
```

- Ein so organisierter Index heißt geclusterter Index
  - Sequentieller Zugriff während der Scan-Phase
  - Besonders für Bereichsanfragen geeignet

Warum macht man Indexe nicht immer geclustert?

Hash-basierte Indexierung

- B⁺-Bäume dominieren in Datenbanken
- Alternative: hash-basierte Indexierung



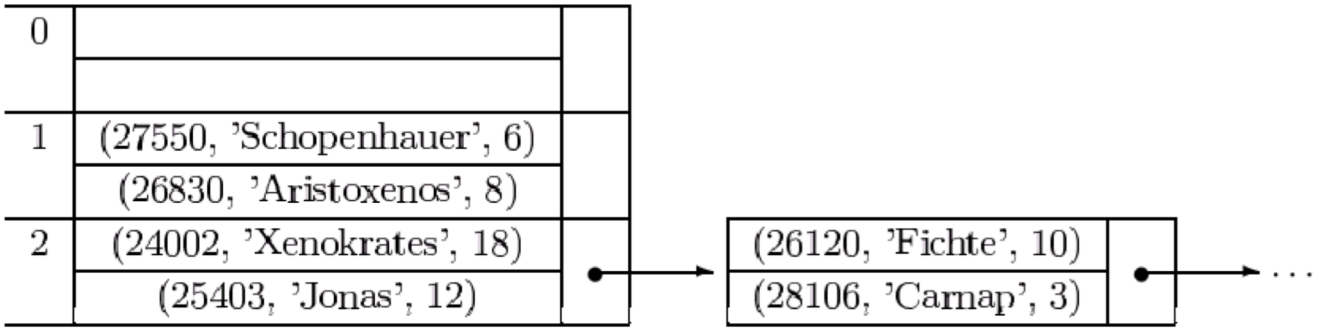
- Anzahl an Buckets n: vorher festzulegen
  - n << Anzahl an Eingabewerten (Werte eines Attributs oder einer Kombination von Attributen)
- Inhalt Bucket-Seite: Datensätze oder rid Liste
  - Wird gefüllt durch Anwendung der Hash-Funktion auf Datensatz und anschließende Einsortierung in Bucket
  - Wenn Seite voll: Kollisionslisten (overflow pages), lineares Sondieren o.ä.

Statisches Hashing

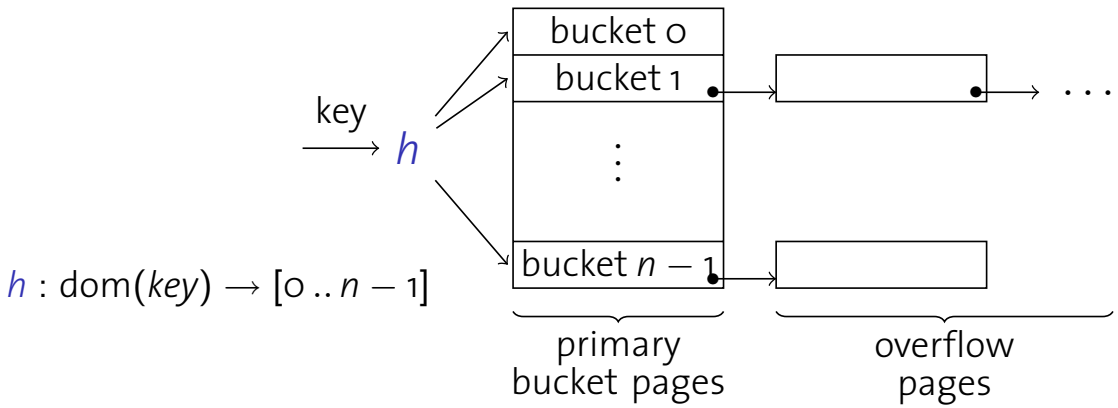
- Relation Student(Matrikelnummer, Name, Semester)
- Hashfunktion h(x) = x mod 3, x = Matrikelnummer
  - Anzahl an Buckets n = 3, daher mod 3

0	
1	(27550, 'Schopenhauer', 6)
2	(24002, 'Xenokrates', 18)
	(25403, 'Jonas', 12)

- Kollisionsbehandlung mit Kollisionslisten



Hash-basierte Indexierung



- Hash-Indexe eignen sich nur für Gleichheitsprädikate
  - Insbesondere für (lange) Zeichenketten

Warum?

Warum und wie?

- Beispielanfragen, für die sich ein Hash-Index lohnen könnte
  - SELECT *
    FROM AbtStandort
    WHERE Standort='Stafford‘;

```
SELECT *
FROM Kunden k, Auftraege a
WHERE k.Name='IBM Corp.' AND
      k.Kunden_ID=a.Kunden_ID;
```

Dynamisches Hashing

- Statisches Hashing ineffizient bei unvorhersehbaren Daten und langen Kollisionslisten

Problem: Wie groß soll die Anzahl n der Buckets sein?

- n zu groß → schlechte Platznutzung und –Lokalität
- n zu klein → viele Überlaufseiten, lange Listen

Datenbanken verwenden daher **dynamisches Hashen** (dynamisch wachsende und schrumpfende Bildbereiche)

- **Erweiterbares** Hashen
  (Vermeidung des Umkopierens)

Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R., Extendible Hashing - A Fast Access Method for Dynamic Files, *ACM Transactions on Database Systems* **4** (3): 315–344, **1979**

Indexe: Zusammenfassung

- Zugriff auf Daten von O(n) ungefähr auf O(log n)
- Kosten der Indexierung aber nicht zu vernachlässigen
  - Nicht bei „kleinen“ Tabellen
  - Nicht bei häufigen Update- oder Insert-Anweisungen
  - Nicht bei Spalten mit vielen Null-Werten
- Standardisierung nicht gegeben
  - Beispiel: MySQL (Ausschnitt)

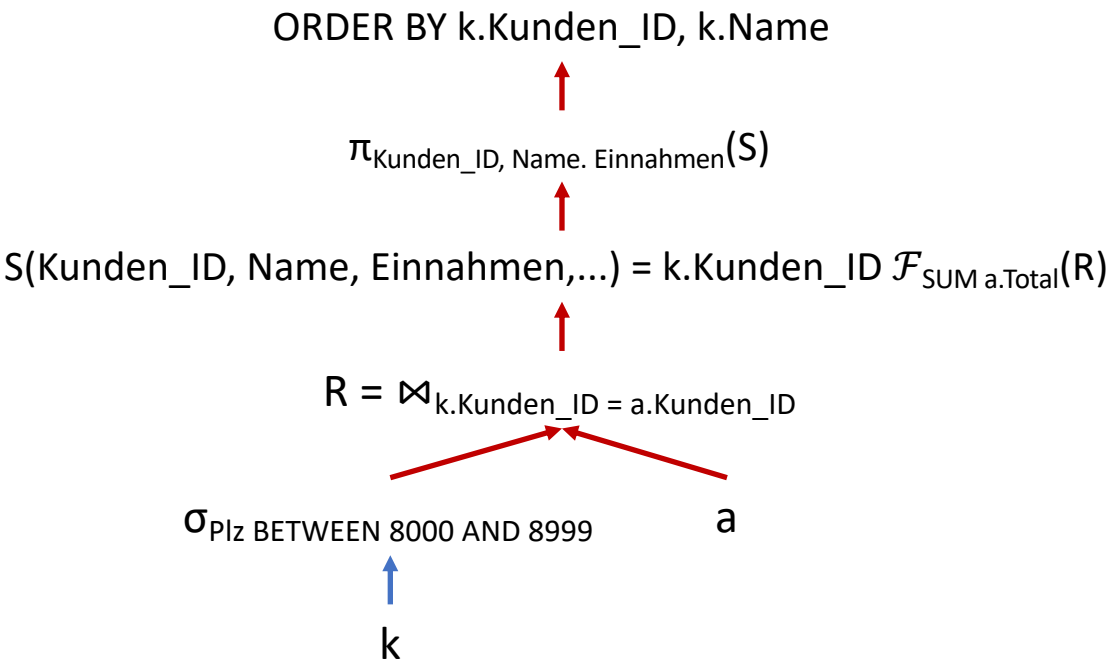
```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name [index_type]
   ON tbl_name (index_col_name,...) [index_type]

index_col_name:
   col_name [(length)] [ASC | DESC]

index_type:
   USING {BTREE | HASH}
```

Zwischen-Rückblick

- Index-Sequentielle Zugriffsmethode (ISAM-Index)
  - Statisch, baum-basierte Indexstruktur
- B+-Bäume
  - Die Datenbank-Indexstruktur
  - Auf linearer Ordnung basierend
  - Dynamisch
  - Kleine Baumhöhe für fokussierten Zugriff auf Bereiche
- Geclusterte vs. ungeclusterte Indexe
  - Sequentieller Zugriff vs. Verwaltungsaufwand
- Hash-basierte Indexe
  - Gleichheitsprädikate

Ausführungspläne

```
SELECT k.Kunden_ID, k.Name, SUM (a.Total) AS Einnahmen
FROM Kunden AS k, Auftraege AS a
WHERE k.Plz BETWEEN 8000 AND 8999
    AND k.Kunden_ID = a.Kunden_ID
GROUP BY k.Kunden_ID
ORDER BY k.Kunden_ID, k.Name;
```



ORDER BY k.Kunden_ID, k.Name

↑

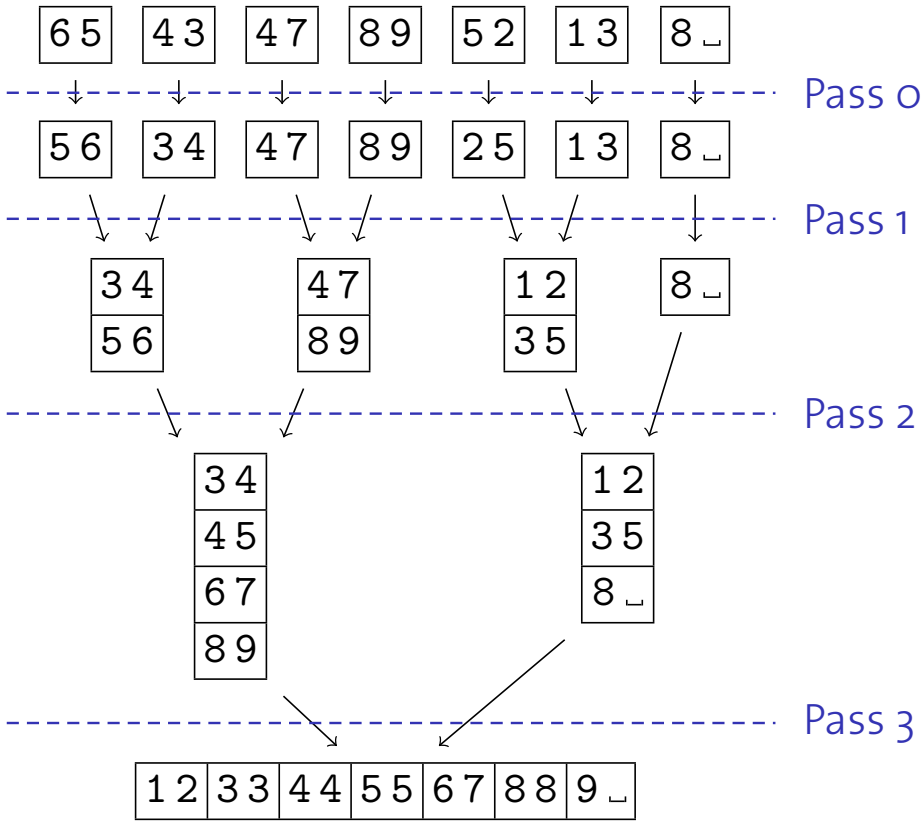
$\pi_{\text{Kunden\_ID, Name. Einnahmen}}(S)$

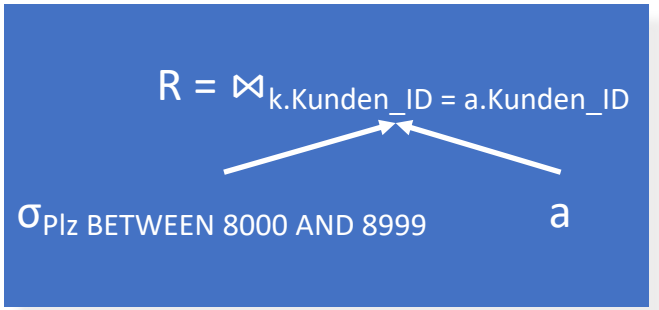
Sortieren

Operator-Evaluierer

Effizientes Sortieren

- Häufiges Vorkommen von Sortieroperationen
  - SQL-Anweisung ACS, DESC
  - B^+-Baum bauen: einfach bei sortierter Eingabe
  - Duplikate-Eliminierung einfach
  - Andere Operatoren erfordern manchmal sortierte Eingaben (mehr dazu später)
- Merge-Sort (Misch-Sortierung)
  - Implementierung mit nur drei Pufferseiten möglich (Aufwand innerhalb von N log N bei N Datensätzen; Schneller bei mehr Pufferseiten, Parallelverarbeitung, ...; weitere Tricks anwendbar)
- Gewählte Implementierung abhängig von verfügbaren Ressourcen und Größe der Daten



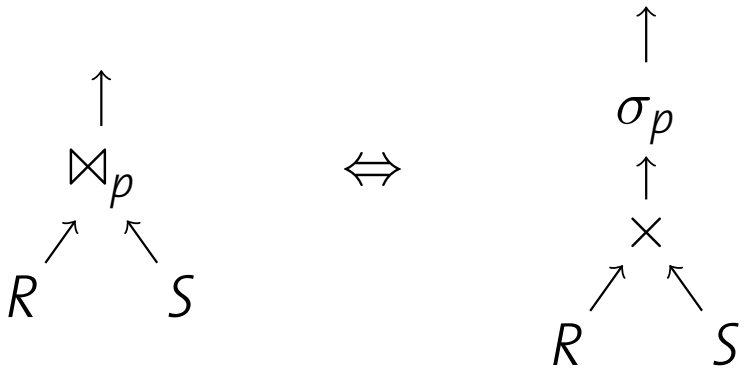


Join-Implementierung

Operator-Evaluierer

Verbundoperator (Join) $R \bowtie S$

- Ein Join $\bowtie_p$ ist eine Abkürzung für die Zusammensetzung von Kreuzprodukt $\times$ und Selektion σ_p



- Daraus ergibt sich eine einfache Implementierung von $\bowtie_p$
  1. Enumeriere alle Datensätze aus $R \times S$
  2. Wähle die Datensätze, die p erfüllen

- Ineffizienz aus Schritt 1 kann überwunden werden (Größe des Zwischenresultats: $|R| \times |S|$)

Join-als-geschachtelte-Schleifen

- Einfache Implementierung des Joins (ohne Enumerierung):
  - nl = nested loop

```
1 Function: nljoin(R, S, p)
2 foreach record r ∈ R do
3     foreach record s ∈ S do
4         if ⟨r, s⟩ satisfies p then
5             append ⟨r, s⟩ to result
```

- Sei N_R und N_S die Seitenzahl in R und S, sei p_R und p_S die Anzahl der Datensätze pro Seite in R und S
  - Anzahl der Plattenzugriffe:

$$N_R + \underbrace{p_r \cdot N_R}_{\text{\#Tupel in R}} \cdot N_S$$

Join-als-geschachtelte-Schleifen

- Nur 3 Seiten nötig (zwei Seiten fürs Lesen von R und S und eine um das Ergebnis zu schreiben)
- I/O-Verhalten: Leider sehr viele wahlfreie Zugriffe
  - Annahme $p_R = p_S = 100$, $N_R = 1000$, $N_s = 500$:

    $$N_R + p_r \cdot N_R \cdot N_S = 1000 + 5 \cdot 10^7 \text{ Seiten zu lesen}$$

  - Mit einer Zugriffszeit von 10ms für jede Seite dauert der Vorgang 140 Stunden
  - Wenn $|S|<|R|$:
    Vertauschen von R und S verbessert die Situation nur marginal
  - Seitenweises Lesen bedingt volle Plattenlatenz, obwohl beide Relationen in sequenzieller Ordnung verarbeitet werden.

Blockweiser Join mit Schleifen

- Einsparung von Kosten durch wahlfreien Zugriff durch blockweises Lesen von R und S mit b_R und b_S vielen Seiten
  - Braucht mehr Seiten als nljoin(R, S p)

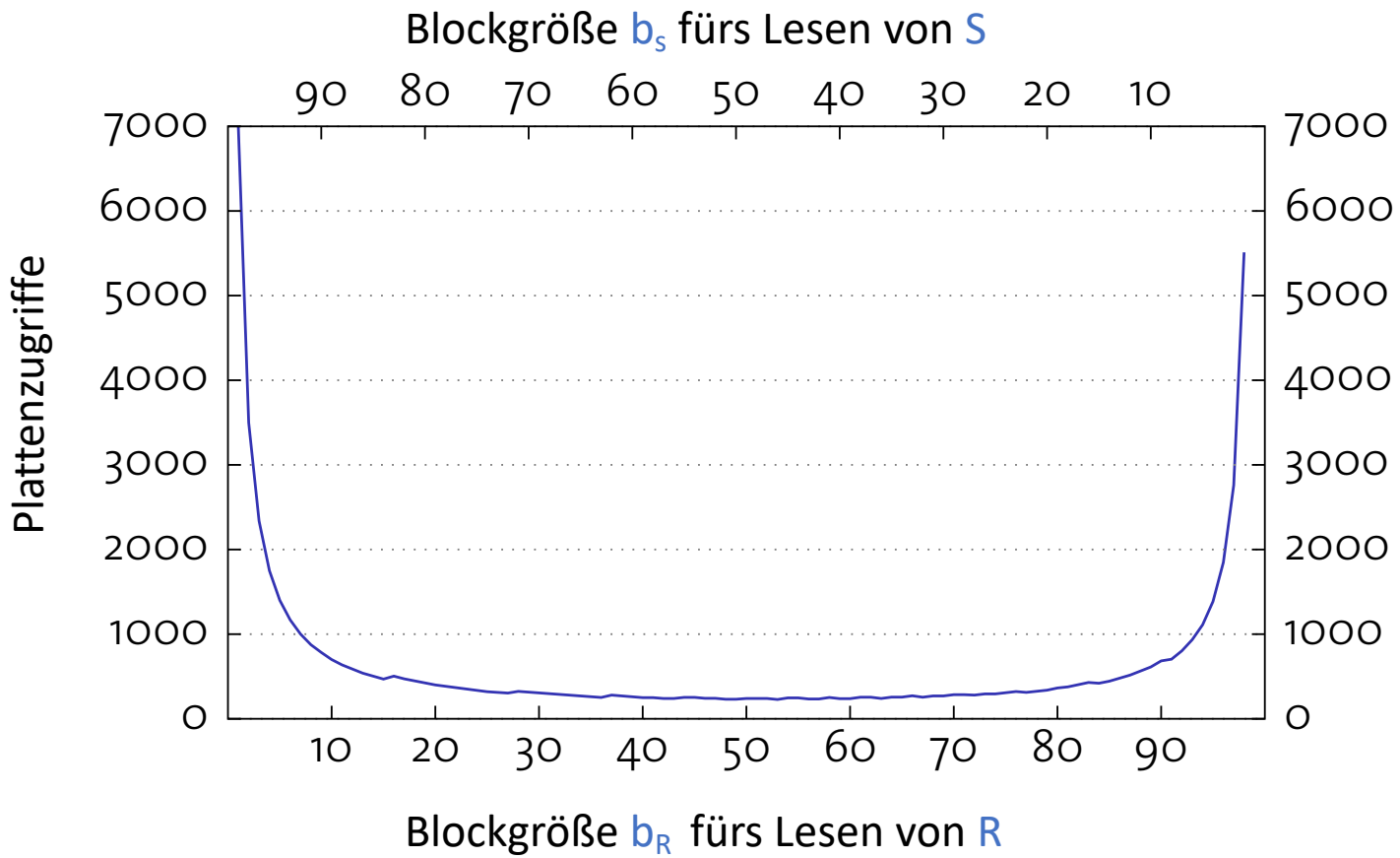
```
1 Function: block_nljoin (R, S, p)
2 foreach b_R-sized block in R do
3     foreach b_S-sized block in S do
4         find matches in current R- and S-blocks and
          append them to the result ;
```

Join im Hauptspeicher ausführbar

- R wird (einmal) vollständig gelesen, aber mit nur $\lceil N_R/b_R \rceil$ Lesezugriffen
- S nur $\lceil N_R/b_R \rceil$ mal gelesen, mit $\lceil N_R/b_R \rceil \cdot \lceil N_S/b_S \rceil$ Plattenzugriffen

Wahl von b_R und b_S

Pufferbereich mit $B = 100$ Rahmen, $N_R = 1000$, $N_S = 500$:



Performanz des Hauptspeicher-Joins

- Zeile 4 in block_nljoin(R, S, p) bedingt einen Hauptspeicherverbund zwischen Blöcken aus R und S
- Aufbau einer Hashtabelle kann den Verbund erheblich beschleunigen

```
1 Function: block_nljoin'(R, S, p)
2 foreach b_R-sized block in R do
3     build an in-memory hash table H for the current R-block ;
4     foreach b_S-sized block in S do
5         foreach record s in current S-block do
6             probe H and append matching ⟨r, s⟩ tuples to result ;
```

Warum Hashtabelle für R-Block und nicht S-Block?

- Funktioniert nur für Gleichheitsprädikate im Join
  - Vgl. Hash-Index

Indexbasierte Verbunde R ⋈ S

- Verwendung eines vorhandenen Index für die innere Relation S
  - Ggf. innere und äußere Relation vertauschen

1 **Function:** `index_nljoin` (R, S, p)
2 **foreach** record $r \in R$ **do**
3 probe index using r and append all matching tuples to result ;

Suchschlüssel für Suche im Index

- Index muss verträglich mit der Join-Bedingung sein
  - Join-Attribute heißen dann „sargable“ (SARG = Search ARGument)
  - Hash-Index (nur für Gleichheitsprädikate) oder auch B^+-Baum
  - Häufiger Join → Passenden Index aufbauen
  - Manchmal auch nur partielle Abdeckung nützlich (wenn z.B. jeweils ein Konjunkt mit einem jeweils anderen Index verträglich)

Sortier-Merge-Join

- Join-Berechnung wird einfach, wenn Eingaberelationen bzgl. Join-Attribut(en) sortiert
- Merge-Join kombiniert Eingabetabellen ähnlich wie beim Merge-Sort (Misch-Sortieren)
- Es gibt aber i.d.R mehrfache Korrespondenzen in der anderen Relation

A	*B*
"foo"	1
"foo"	2
"bar"	2
"baz"	2
"baf"	4

$\bowtie_{B=C}$

C	*D*
1	false
2	true
2	false
3	true

- Merge-Join nur für Gleichheitsprädikate verwendbar

Merge-Join

```
1  Function: merge_join (R, S, α = β)      // α, β: join columns in R, S
2  r ← position of first tuple in R ;      // r, s, s′: cursors over R, S, S
3  s ← position of first tuple in S ;
4  while r ≠ eof and s ≠ eof do            // eof: end of file marker
5      while r.α < s.β do
6          advance r ;
7      while r.α > s.β do
8          advance s ;
9      s′ ← s ;                            // Remember current position in S
10     while r.α = s′.β do                 // All R-tuples with same α value
11         s ← s′ ;                        // Rewind s to s′
12         while r.α = s.β do              // All S-tuples with same β value
13             append ⟨r, s⟩ to result ;
14             advance s ;
15         advance r ;
```

Merge-Join: I/O-Verhalten

- Wenn beide Eingaben sortiert und keine außergewöhnlich langen Sequenzen mit identischen Schlüsselwerten vorhanden, dann ist der I/O-Aufwand

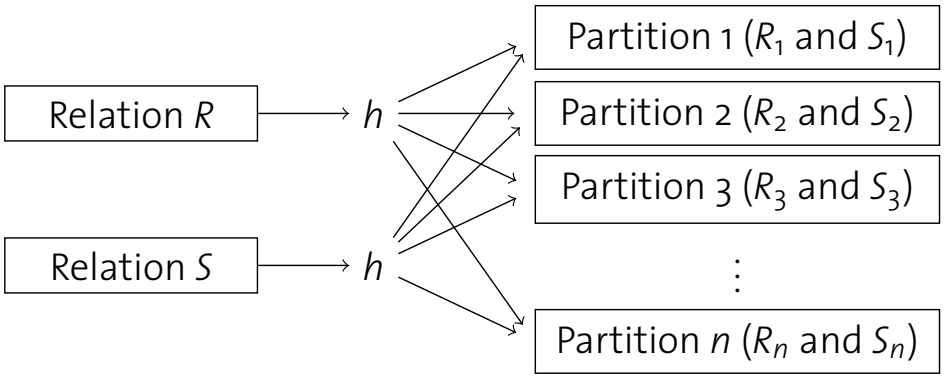
$$N_R + N_S \text{ (das ist dann optimal)}$$

  - Durch blockweises I/O treten fast immer sequenzielle Lesevorgänge auf
- Vorher sortieren kann sich also für Join-Berechnung auszahlen
- Ausgabe weiterhin sortiert
  - Wenn später eine Sortierung der Ausgabe gefordert wird, lohnt sich die vorherige Sortierung noch mehr
  - Zudem weniger Festplattentransfers mit Merge-Join im Vergleich zu erst eine Art von Join ausführen und dann sortieren

Hash-Join

- Sortierung bringt korrespondierende Tupel in eine „räumliche Nähe“, so dass eine effiziente Verarbeitung möglich ist
- Ähnlicher Effekt erreichbar mit Hash-Verfahren

- Zerlege R und S in Teilrelationen R_1, ..., R_n und S_1,...,S_n mit der gleichen Hashfunktion (angewendet auf die Join-Attribute)



- $R_i \bowtie S_j = \emptyset$ für alle $i \neq j$

Hash-Join

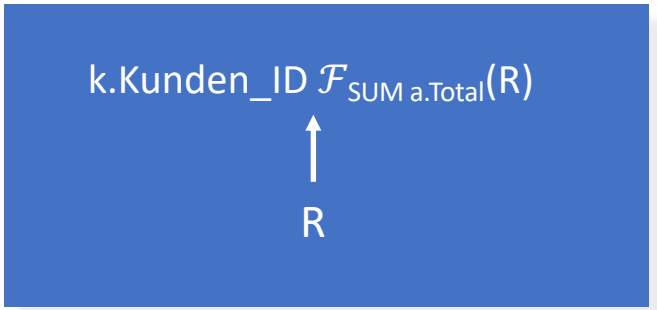
- Mittels Hashfunktion werden die Tupel aus R und S partitioniert
  - Durch Partitionierung werden kleine Relationen R_i und S_i geschaffen
  - Korrespondierende Datensätze kommen garantiert in korrespondierende Partitionen der Relationen
- Es muss $R_i \bowtie S_i$ (für alle i) berechnet werden (einfacher)

- Die Anzahl der Partitionen n (d.h. die Hashfunktion) sollte mit Bedacht gewählt werden, so dass $R_i \bowtie S_i$ als Hauptspeicher-Join berechnet werden kann
  - Solange mit Hashfunktionen partitionieren bis die Partitionen der kleineren Relation in den Hauptspeicher passen
  - Partitionen der größeren Relation müssen nicht in den Hauptspeicher passen, werden dann block-weise geladen, wenn $R_i \bowtie S_i$ berechnet wird

Hash-Join-Algorithmus

```
1  Function: hash_join(R, S, α = β)
2  foreach record r ∈ R do
3      append r to partition R_{h(r.α)}
4  foreach record s ∈ S do
5      append s to partition S_{h(s.β)}
6  foreach partition i ∈ 1, ..., n do
7      build hash table H for R_i, using hash function h′;
8      foreach block in S_i do
9          foreach record s in current S_i-block do
10             probe H and append matching tuples to result ;
```

- I/O-Aufwand wenn $|R \bowtie S|$ „klein“ $3 \cdot (N_R + N_S)$
  - Lesen und Schreiben beider Relationen für Partitionierung + Lesen beider Relationen für Join



Gruppierung + UNIQUE-Behandlung

Anfrageverarbeitung

Gruppierung und Duplikate-Elimination

- Herausforderung: Finde „identische“ Datensätze in einer Datei
  - Identische Datensätze = Duplikate-Elimination
  - Identisch bzgl. Gruppierungsattribut(e) = Gruppierung
- Ähnlichkeiten zum Eigenverbund (self-join)
  - Duplikate: basierend auf allen Spalten der Relation
  - Gruppierung: basierend auf Gruppierungsattribut(en)

- Umsetzung der Duplikate-Elimination oder Gruppierung mit Hash-Join oder Sortierung
  - Siehe vorherige Folien

$\pi_{\text{Kunden\_ID, Name. Einnahmen}}(S)$

$\sigma_{\text{Plz BETWEEN 8000 AND 8999}}$

Selektion und Projektion

Anfrageverarbeitung

Andere Anfrage-Operatoren

Projektion π

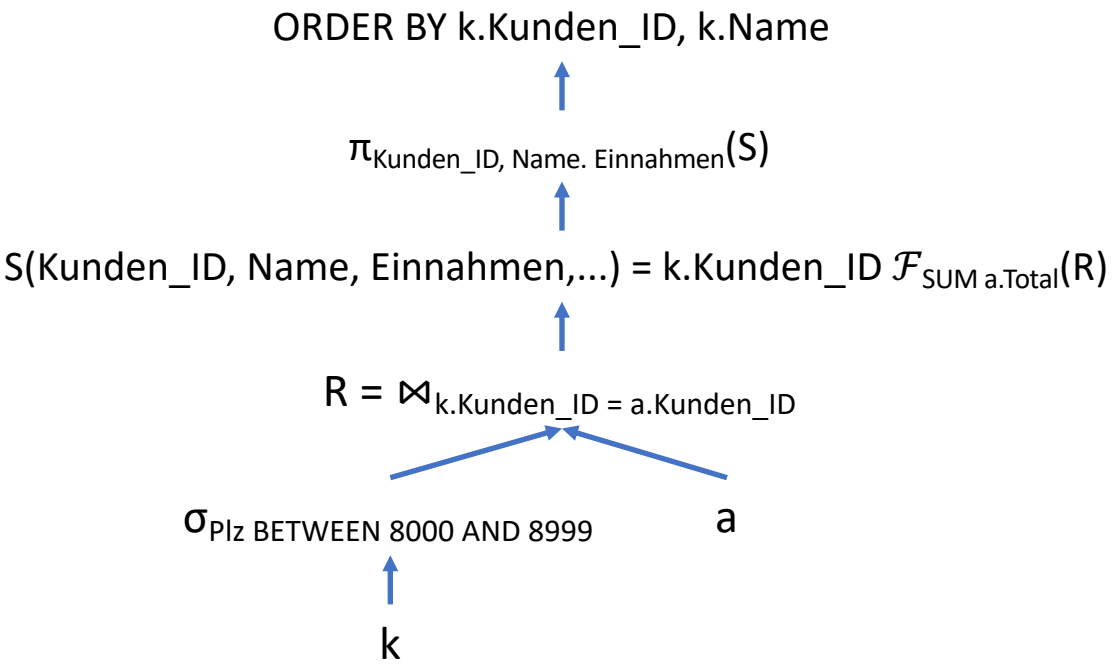
- Implementierung durch
  a) Entfernen nicht benötigter Spalten
  b) Eliminierung von Duplikaten
- Die Implementierung von
  - a) bedingt das Ablaufen (scan) aller Datensätze in der Datei,
  - b) siehe vorherigen Abschnitt
- Systeme vermeiden b) sofern möglich

Selektion σ

- Ablaufen (scan) aller Datensätze
- Eventuell Sortierung ausnutzen oder Index verwenden

Ausführungspläne

```
SELECT k.Kunden_ID, k.Name, SUM (a.Total) AS Einnahmen
FROM Kunden AS k, Auftraege AS a
WHERE k.Plz BETWEEN 8000 AND 8999
    AND k.Kunden_ID = a.Kunden_ID
GROUP BY k.Kunden_ID
ORDER BY k.Kunden_ID, k.Name;
```

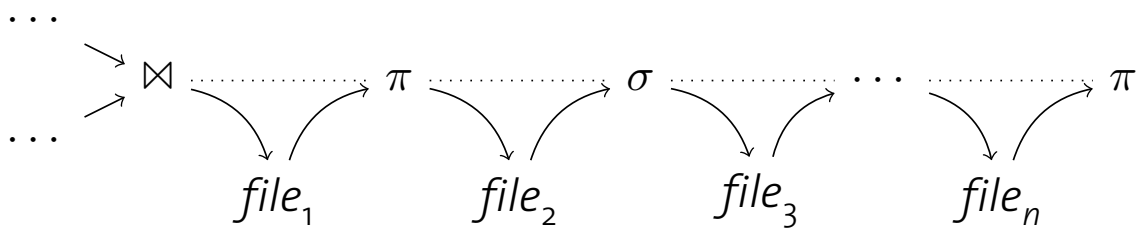


Durchführung der Anfrage

Anfrageverarbeitung

Organisation der Operator-Evaluierung

- Bisher gehen wir davon aus, dass Operatoren ganze Dateien verarbeiten



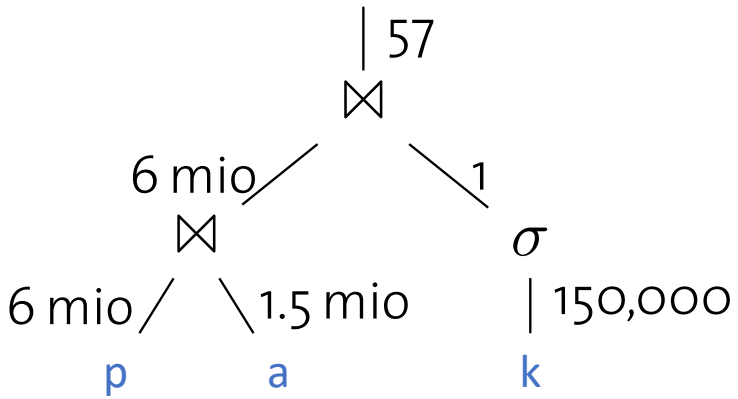
- Das erzeugt offensichtlich viel I/O
- Außerdem: lange Antwortzeiten
  - Ein Operator kann nicht anfangen, solange nicht seine Eingaben vollständig bestimmt sind (materialisiert sind)
  - Operatoren werden nacheinander ausgeführt

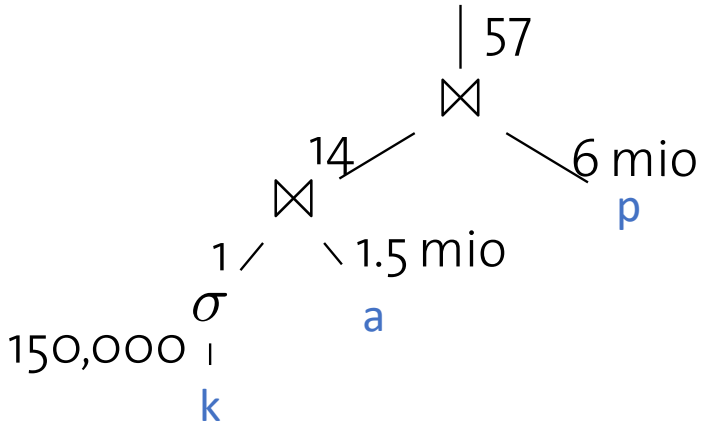
Pipeline-orientierte Verarbeitung

- Alternativ könnte jeder Operator seine Ergebnisse direkt an den nachfolgenden senden, ohne die Ergebnisse erst auf die Platte zu schreiben
- Ergebnisse werden so früh wie möglich weitergereicht und verarbeitet (Pipeline-Prinzip)
- Granularität ist bedeutsam:
  - Kleinere Brocken reduzieren Antwortzeit des Systems
  - Größere Brocken erhöhen Effektivität von Instruktions-Cachespeichern
  - In der Praxis meist tupelweises Verarbeiten verwendet
- Siehe auch Gebiet der Stromverarbeitung

Auswirkungen auf die Performanz

```
SELECT p.Teile_ID, p.Anzahl, p.Preis
FROM Auftragsposten p, Auftraege a, Kunden k
WHERE p.Auftrag_ID = a.Auftrag_ID
    AND a.Kunden_ID = k.Kunden_ID
    AND k.Name = 'IBM Corp.';
```





- Rechter Ausführungsplan
  - Tupelweises Verarbeiten der Relation k
  - Sofortiges Weiterleiten nach der Selektion
  - Kombiniert mit tupelweisem Verarbeiten der Relationen a und p

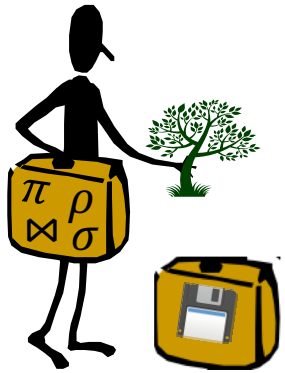
Blockierende Operatoren

- Pipelining reduziert Speicheranforderungen und Antwortzeiten, da jeder Datensatz gleich weitergeleitet
- Funktioniert so nicht für alle Operatoren
  - Welche?
    - Misch-Sortierung
    - Gruppierung, Duplikate-Elimination, Max/Min über einer unsortierten Eingabe

- Solche Operatoren nennt man blockierend
- Blockierende Operatoren konsumieren die gesamte Eingabe in einem Rutsch, bevor die Ausgabe erzeugt werden kann
  (Daten auf Festplatte zwischengespeichert)

Zusammenfassung

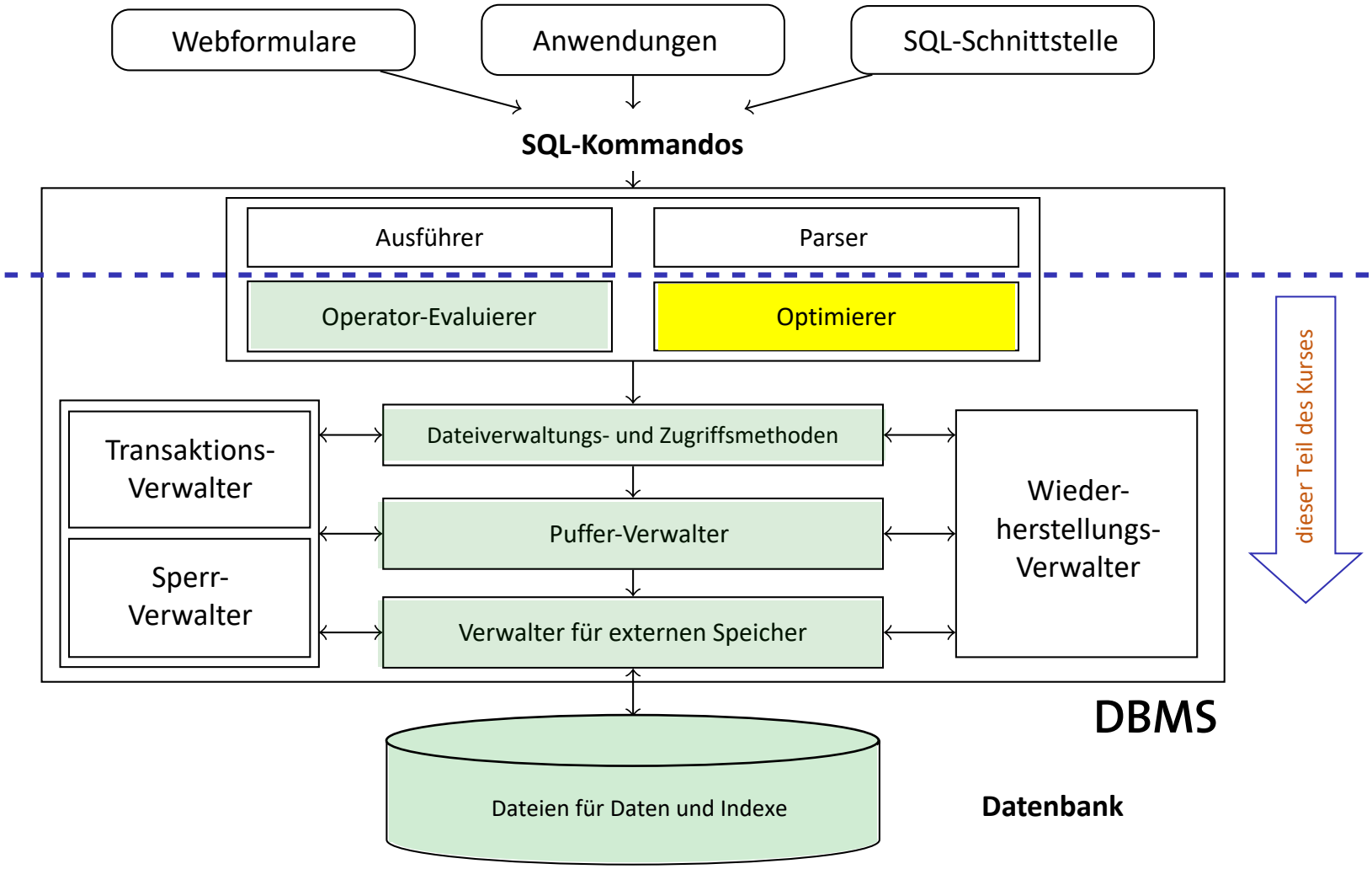
- Teile-und-Herrsche
  - Zerlegung einer großen Anfrage in kleine Teile
  - Beispiel:
    - Merge-Sort
    - Partitionierung mit Hashfunktion (Hash-Join)



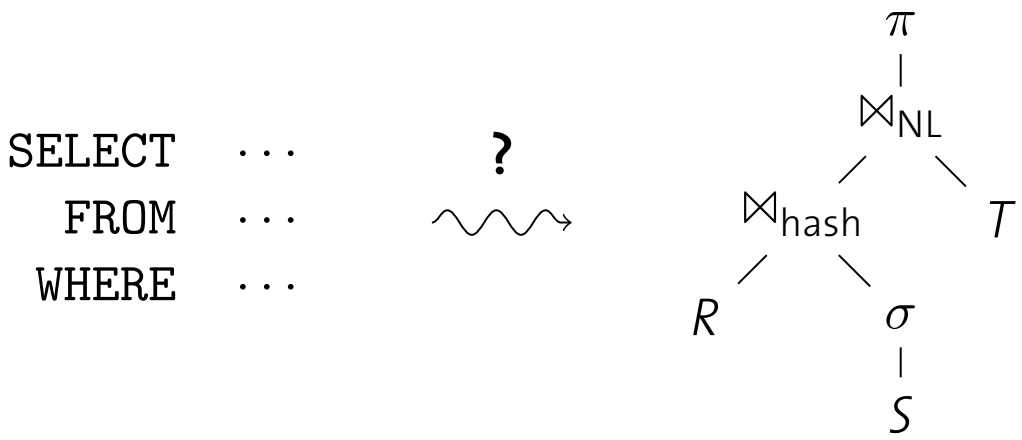
- Blockweises Durchführen von I/O
  - Lesen und Schreiben von größeren Einheiten kann die Verarbeitungszeit deutlich reduzieren, da wahlfreier Zugriff auf Daten vermieden wird
  - Beispiel:
    - Join-Implementierungen

- Pipeline-orientierte Verarbeitung
  - Speicherreduktion und Laufzeitverbesserung durch Vermeidung der vollständigen Materialisierung von Zwischenresultaten

Architektur eines DBMS



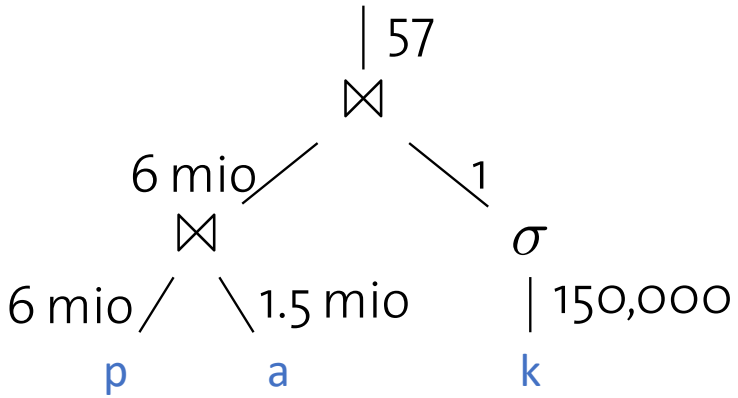
Anfrageoptimierung

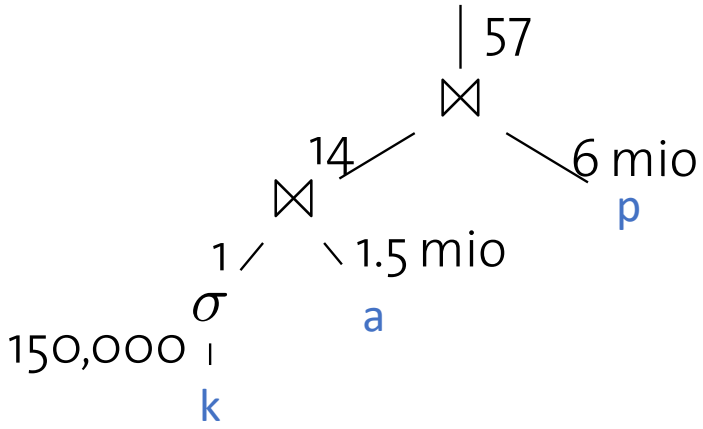


- Es gibt mehr als eine Art, eine Anfrage zu beantworten
  - Welche Implementation eines Join-Operators?
  - Welche Parameter für Blockgrößen, Pufferallokation, ...
  - Automatisch einen Index aufsetzen?
- Die Aufgabe, den besten Ausführungsplan zu finden, ist der heilige Gral der Datenbankimplementierung

Auswirkungen auf die Performanz

```
SELECT p.Teile_ID, p.Anzahl, p.Preis
FROM Auftragsposten p, Auftraege a, Kunden k
WHERE p.Auftrag_ID = a.Auftrag_ID
    AND a.Kunden_ID = k.Kunden_ID
    AND k.Name = 'IBM Corp.';
```

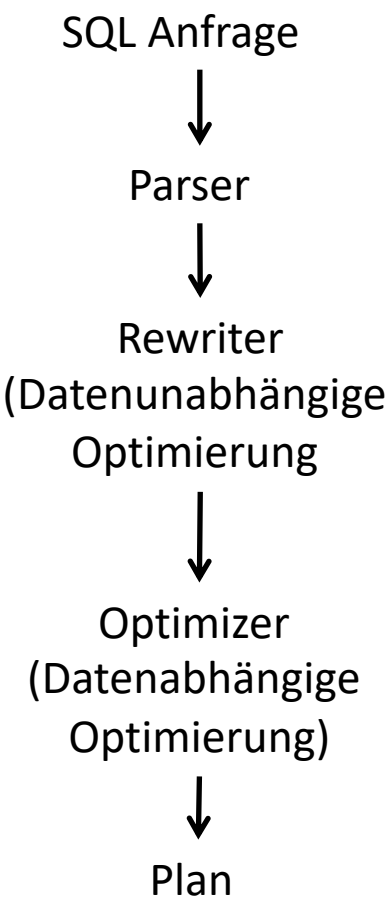




- Bezogen auf die Ausführungszeit können die Unterschiede „Sekunden vs. Tage“ bedeuten

Optimierung

- Optimierungen können unabhängig von den Daten erfolgen; man spricht dann auch vom Umschreiben (Rewriting)
  - Selektionsprädikate früh anwenden
  - Vermeide Duplikatenelimination, wenn möglich
  - ...
- Datenabhängige Optimierung (Optimizer)
  - Kostenbasiert auf Basis der Daten in der DB bzw. statistisch relevanter Größen der DB
- Hier nicht näher besprochen
  - Minimierung einer Anfrage durch Elimination einer Unteranfrage
  - Elimination eines teuren Operators
  - Bestimmung relevanter Tabellen



M. Benedikt: How Can Reasoners Simplify Database Querying (And Why Haven't They Done It Yet)? (Invited Talk at PODS 2018.)

Prädikatsvereinfachung (Rewriting)

- Beispiel: Schreibe

Non-Sargable

```
SELECT *
FROM Einzelposten p
WHERE p.Steuern * 100 < 5;
```

um in

Sargable

```
SELECT *
FROM Einzelposten p
WHERE p.Steuern < 0.05;
```

- Prädikatsvereinfachung ermöglicht Verwendung von Indexen und vereinfacht die Erkennung von effizienten Join-Implementierungen

Zusätzliche Verbundprädikate

- Implizite Verbundprädikate wie in

```
SELECT *
FROM A, B, C
WHERE A.a = B.b AND B.b = C.c;
```

können explizit gemacht werden

```
SELECT *
FROM A, B, C
WHERE A.a = B.b AND B.b = C.c AND A.a = C.c;
```

- Hierdurch werden Pläne möglich wie $(A \bowtie C) \bowtie B$

Geschachtelte Anfragen

- SQL bietet viele Wege, geschachtelte Anfrage zu schreiben
  - Unkorrelierte Unteranfragen

```
SELECT *
FROM Auftraege
WHERE Kunden_ID IN (SELECT Kunden_ID
                    FROM Kunden
                    WHERE Name = 'IBM Corp.');
```

  - Korrelierte Unteranfragen

```
SELECT *
FROM Auftraege a
WHERE Kunden_ID IN (SELECT k.Kunden_ID
                    FROM Kunden k
                    WHERE k.Kontostand = a.Gesamtpreis);
```

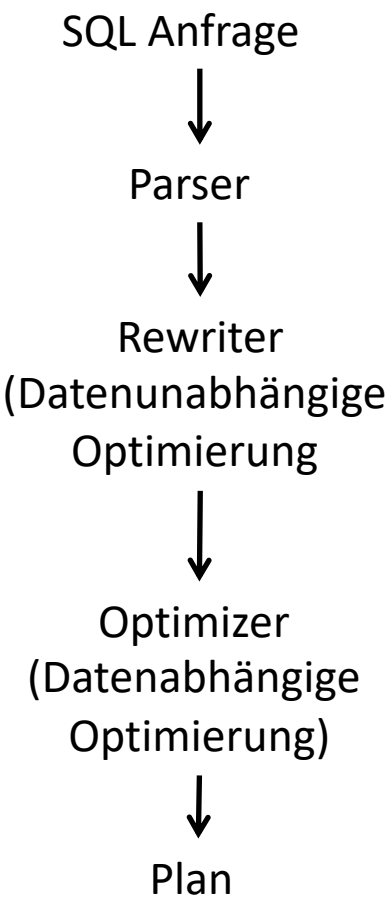
- Unkorreliert: Unteranfrage nur einmal auswerten
- Meist sind Unteranfragen nur syntaktische Varianten von Joins
  - Rewriting: Joins explizit machen für Join-Order-Optimierung

Optimierung

- Optimierungen können unabhängig von den Daten erfolgen; man spricht dann auch vom Umschreiben (Rewriting)
  - Selektionsprädikate früh anwenden
  - Vermeide Duplikatenelimination, wenn möglich
  - ...

- Datenabhängige Optimierung (Optimizer)
  - Kostenbasiert auf Basis der Daten in der DB bzw. statistisch relevanter Größen der DB

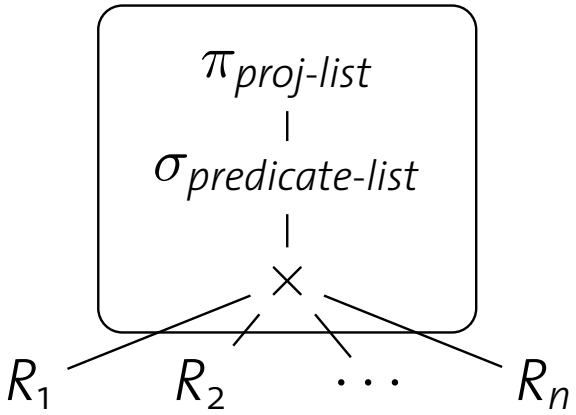
- Hier nicht näher besprochen
  - Minimierung einer Anfrage durch Elimination einer Unteranfrage
  - Elimination eines teuren Operators
  - Bestimmung relevanter Tabellen



M. Benedikt: How Can Reasoners Simplify Database Querying (And Why Haven't They Done It Yet)? (Invited Talk at PODS 2018.)

Abschätzung der Ergebnisgröße

Betrachte Anfrageblock für Select-From-Where-Anfrage Q



Abschätzung der Ergebnisgröße von Q durch

- die Größe der Eingabetabellen $|R_1|, |R_2|, \ldots, |R_n|$ und
- die **Selektivität** sel(predicate-list)

$$|Q| = |R_1| \cdot |R_2| \cdot \ldots \cdot |R_n| \cdot \text{sel(predicate-list)}$$

Tabellenkardinalitäten

- Die Größe einer Tabelle ist über den Systemkatalog verfügbar (hier IBM DB2)
- Vor Ausführung der Anfrage verfügbar: offline (bei DB-Änderungen wird Tabelle upgedatet)

```
db2 => SELECT TABNAME, CARD, NPAGES
db2 (cont.) => FROM SYSCAT.TABLES
db2 (cont.) => WHERE TABSCHEMA = 'TPCH';
TABNAME         CARD                 NPAGES
--------------- -------------------- --------------------
ORDERS          1500000              44331
CUSTOMER        150000               6747
NATION          25                   2
REGION          5                    1
PART            200000               7578
SUPPLIER        10000                406
PARTSUPP        800000               31679
LINEITEM        6001215              207888
8 record(s) selected.
```

Grobe Abschätzung der Selektivität

- ... durch Induktion über die Struktur des Anfrageblocks
  - V(A,R) = Anzahl verschiedener Werte von Attribut (Spalte) A in Relation R
  - R.A=value: $sel(\cdot) =$
    - $1/V(A,R)$ falls V(A,R) bestimmbar
    - $1/10$ sonst
  - R.A=S.B: $sel(\cdot) =$
    - $1/\max\{V(A,R), V(B,S)\}$ falls V(A,R), V(B,S) bestimmbar
    - $1/V(Rel,Attr)$ falls nur eins bestimmbar
    - $1/10$ sonst
  - p_1 AND p_2: $sel(\cdot) = sel(p_1) \cdot sel(p_2)$
  - p1 OR p2: $sel(\cdot) = sel(p_1) + sel(p_2) - sel(p_1) \cdot sel(p_2)$

Woher?

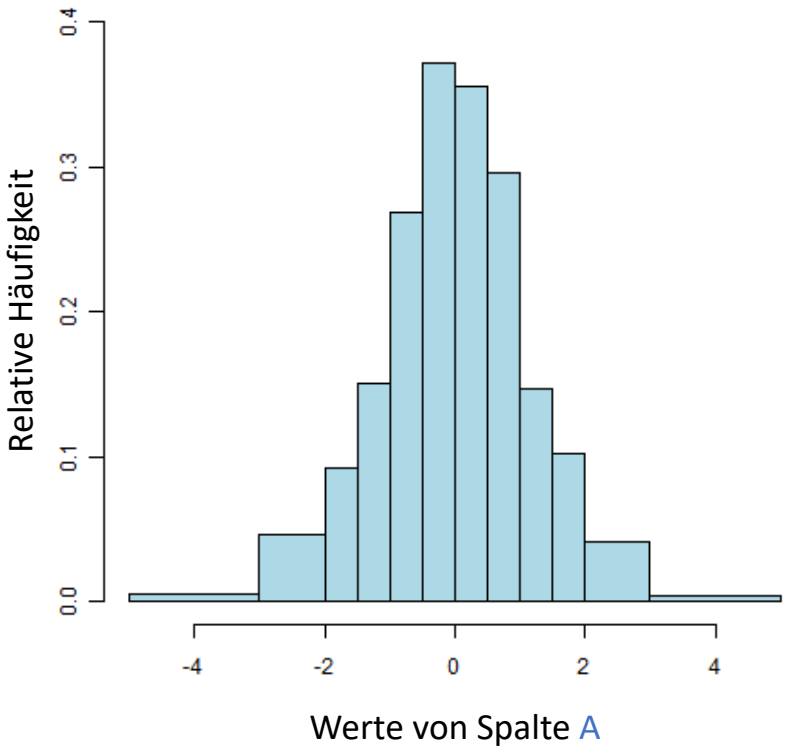
Warum 1/10?

Verbesserung der Selektivitätsabschätzung

- Annahmen
  - Gleichverteilung der Datenwerte in einer Spalte
  - Unabhängigkeit zwischen einzelnen Prädikaten
- Annahmen nicht immer gerechtfertigt
- Sammlung von Datenstatistiken (offline)
  - Speicherung im Systemkatalog
    - IBM DB2: RUNSTATS ON TABLE
  - Meistverwendet: Histogramme

Histogramme

- Mit Histogrammen können echte Verteilungen von Werten einer Spalte A approximiert werden
  - Wenn Domäne von A endlich: Aufteilung nach möglichen Werten x mit eingeschränkter Beziehung zwischen den Werten
  - Wenn Domäne von A gegeben durch Zahlen: Aufteilung in angrenzende Intervalle mit Grenzwerten x_i
- Sammle statistische Parameter für jedes Intervall, z.B.
  1. Anzahl Zeilen z mit $x_i\text{-}1 < z.A <= x_i$ bzw. mit $z.A = x$
  2. Anzahl verschiedener Werte von A im Intervall $(x_i\text{-}1,x_i]$, absolut oder relativ



Histogramme

- DB2: SYSCAT.COLDIST enthält Informationen wie
  - n-häufigste Werte (und deren Anzahl)
  - Auch Anzahl der verschiedenen Werte pro Histogramm-Rasterplatz anfragbar
- Tatsächlich können Histogramme auch absichtlich gesetzt werden, um den Optimierer zu beeinflussen

```
SELECT SEQNO, COLVALUE, VALCOUNT
FROM SYSCAT.COLDIST
WHERE TABNAME = 'LINEITEM'
AND COLNAME = 'L_EXTENDEDPRICE'
AND TYPE = 'Q';
SEQNO COLVALUE           VALCOUNT
----- ----------------- --------
1     +0000000000996.01 3001
2     +0000000004513.26 315064
3     +0000000007367.60 633128
4     +0000000011861.82 948192
5     +0000000015921.28 1263256
6     +0000000019922.76 1578320
7     +0000000024103.20 1896384
8     +0000000027733.58 2211448
9     +0000000031961.80 2526512
10    +0000000035584.72 2841576
11    +0000000039772.92 3159640
12    +0000000043395.75 3474704
13    +0000000047013.98 3789768
```

DB2: TYPE='Q' Quantile (cumulative distribution), TYPE='F' Frequency

Bessere Abschätzung der Selektivität

- MCV(A,R) = n-häufigsten (top-n) Werte in Spalte A einer Tabelle R
- MCF(A,R) = Häufigkeiten dieser Werte
  - MCF(A,R)[value]: Zugriff auf Häufigkeit von value, falls value ∈ MCV(A,R)
- Verbesserte Abschätzung für R.A=value

- sel() =
  - 1/MCF(A,R)[value] falls value ∈ MCV(A,R)
  - 1/V(A,R) falls value ∉ MCV(A,R), aber V(A,R) vorhanden
  - 1/10 sonst

Kardinalitätsabschätzung für Projektion

- Anfrage $Q : \pi_L(R)$ mit $L = (A_1, ..., A_K)$ Liste von Spalten

- $|Q| =$

$V(A,R)$	falls $L = (A)$ (mit Duplikateneliminierung)
$\|R\|$	falls Schlüsselattribut(e) von R in L
$\|R\|$	ohne Duplikateneliminierung
$\text{Min}\,(\|R\|, \Pi_{Ai \in L}\, V(A_i,R))$	sonst

Kardinalitätsabschätzung für Join

- Im Allgemeinen nicht-trivial
- Wenn Fremdschlüsselbeziehungen vorliegen, wie folgt abschätzbar:
  - Fremdschlüssel auf Primärschlüssel:

```
CREATE TABLE R (
        A INTEGER NOT NULL,
        ...,
        PRIMARY KEY (A)
);
```

```
CREATE TABLE S (
        ...,
        A INTEGER NOT NULL,
        ...,
        FOREIGN KEY (A) REFERENCES R
);
```

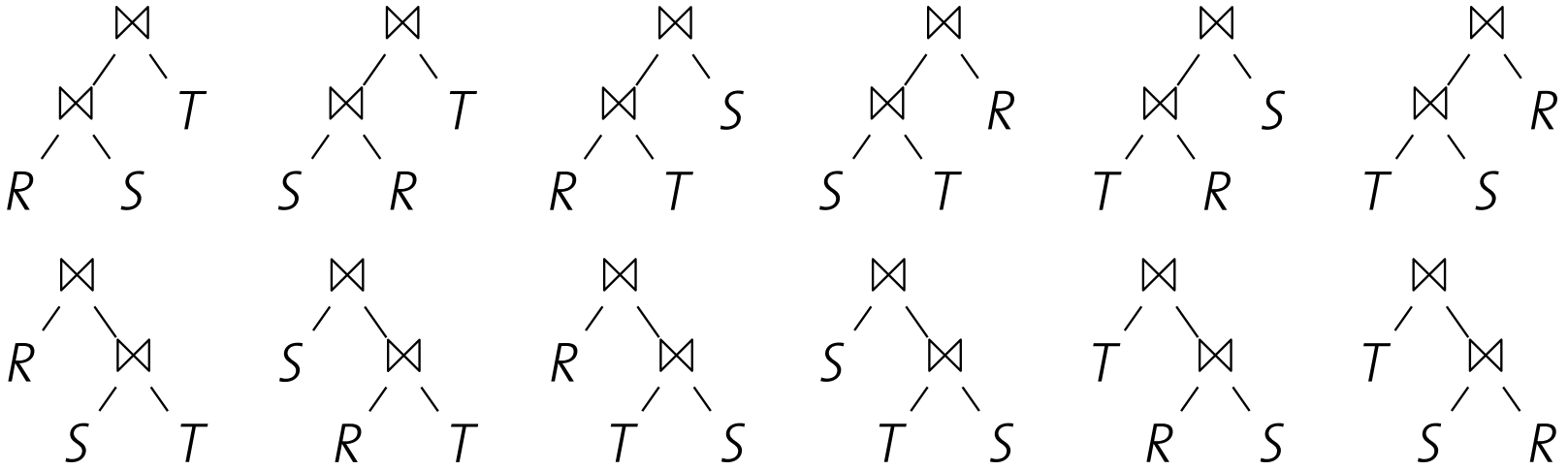
$|\ R \bowtie_{R.A = S.A} S\ | = |S|$ falls S.A Fremdschlüssel auf R.A, R.A Primärschlüssel

  - Fremdschlüssel auf sonstiges Attribut

$$|\ R \bowtie_{R.A = S.B} S\ | = \begin{cases} |R| \cdot |S| \,/\, V(A,R) & \text{falls S.B Fremdschlüssel auf R.A} \\ |R| \cdot |S| \,/\, V(B,S) & \text{falls R.A Fremdschlüssel auf S.B} \end{cases}$$

Join-Optimierung: Es ist noch nicht alles gesagt...

- Problem: Welche Reihenfolge beim Join?
- Beispiel
  - Auflistung der möglichen Ausführungspläne, d.h.
    alle 3-Wege-Join-Kombinationen bei Join über Relationen R, S und T



Suchraum

- Der sich ergebende Suchraum ist enorm groß:
  Schon bei 4 Relationen ergeben sich 120 Möglichkeiten

number of relations *n*	join trees
2	2
3	12
4	120
5	1,680
6	30,240
7	665,280
8	17,297,280
10	17,643,225,600

Anzahl der Bäume für n = k + 1 Inputrelationen:
$C_k \cdot (k+1)! = (2k)!/k!$

C_k = k-te Catalanzahl = $(2k)!/(k+1)!k!$

Dynamische Programmierung

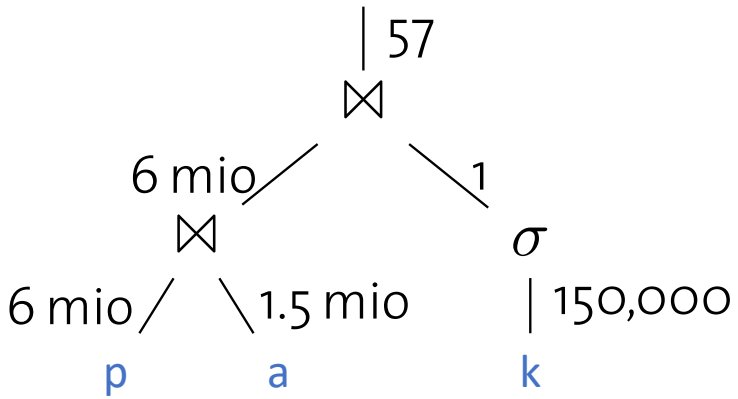
- Sammle gute Zugriffspläne für Einzelrelation (z.B. auch mit Indexscan und mit Ausnutzung von Ordnungen)
- Beschränke dich bei der Betrachtung der nächsten Kombination auf die guten Pläne der vorherigen Kombination

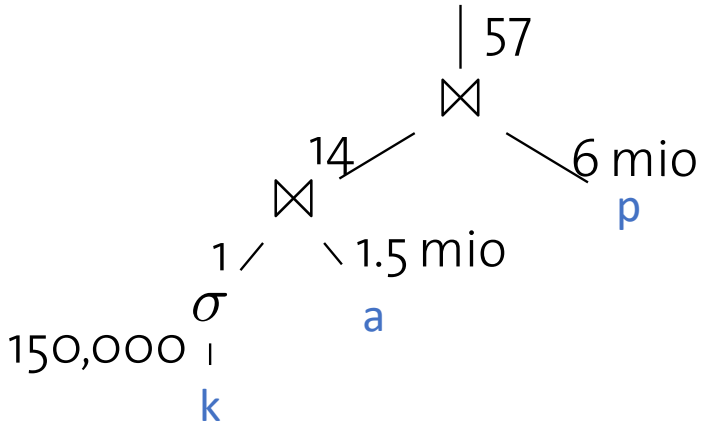
- Grundannahme: Optimalitätsprinzip

  Um den global optimalen Plan zu finden, reicht es aus, die optimalen Pläne bzgl. der Unteranfragen zu betrachten.

Kardinalitätsabschätzung: Beispiel

```
SELECT p.Teile_ID, p.Anzahl, p.Preis
FROM Auftragsposten p, Auftraege a, Kunden k
WHERE p.Auftrag_ID = a.Auftrag_ID
    AND a.Kunden_ID = k.Kunden_ID
    AND k.Name = 'IBM Corp.';
```





- |p| = 6 Mio
- |a| = 1,5 Mio
- |k| = 150.000

Kardinalitätsabschätzung: Beispiel

```
SELECT p.Teile_ID, p.Anzahl, p.Preis
FROM Auftragsposten p, Auftraege a, Kunden k
WHERE p.Auftrag_ID = a.Auftrag_ID
    AND a.Kunden_ID = k.Kunden_ID
    AND k.Name = 'IBM Corp.';
```

- $|p| = 6$ Mio
- $|a| = 1{,}5$ Mio
- $|k| = 150.000$
- $p \bowtie_{p.Auftrag\_ID = a.Auftrag\_ID} a$
  - p.Auftrag_ID Fremdschlüssel auf a.Auftrag_ID → $|p| = 6$ Mio
- $a \bowtie_{a.Kunden\_ID = k.Kunden\_ID} k$
  - a.Kunden_ID Fremdschlüssel auf k.Kunden_ID → $|a| = 1.5$ Mio
- $s = \sigma_{k.Name='IBM\ Corp.'}(k)$
  - Annahme, dass der Name eindeutig ist → Kardinalität: $|s| = 1$

Kardinalitätsabschätzung: Beispiel

```
SELECT p.Teile_ID, p.Anzahl, p.Preis
FROM Auftragsposten p, Auftraege a, Kunden k
WHERE p.Auftrag_ID = a.Auftrag_ID
    AND a.Kunden_ID = k.Kunden_ID
    AND k.Name = 'IBM Corp.';
```

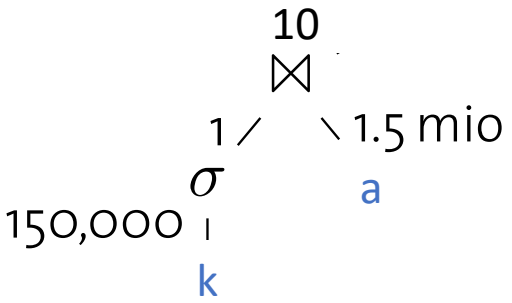
- |p| = 6 Mio
- |a| = 1,5 Mio
- |k| = 150.000
- $p \bowtie_{p.Auftrag\_ID = a.Auftrag\_ID} a$
  - p.Auftrag_ID Fremdschlüssel auf a.Auftrag_ID → |p| = 6 Mio
- $j = a \bowtie_{a.Kunden\_ID = S.Kunden\_ID} s$
  - |a|·|s| / V(Kunden_ID, s) = 1,5 Mio ·1/1 = 1,5 Mio
    bzw. immer noch Fremdschlüssel auf Primärschlüssel
  - Als Selektion: |a|·sel(Kunden_ID=*id*) = |a|·V(Kunden_ID, a)
  - Annahme V(Kunden_ID, a) = 150.000, da 150.000 Kunden insgesamt in DB, dann |j| = 1,5 Mio·1/150.000 = 10

$$\sigma_{\text{150,000}}^{1}\ k$$

Kardinalitätsabschätzung: Beispiel

```
SELECT p.Teile_ID, p.Anzahl, p.Preis
FROM Auftragsposten p, Auftraege a, Kunden k
WHERE p.Auftrag_ID = a.Auftrag_ID
    AND a.Kunden_ID = k.Kunden_ID
    AND k.Name = 'IBM Corp.';
```

- |p| = 6 Mio
- |a| = 1,5 Mio
- |k| = 150.000
- p $\bowtie_{p.Auftrag\_ID = J.Auftrag\_ID}$ j
  - |p| · |j| / V(Auftrag_ID,j) = 6 Mio · 10/10 = 6 Mio
    bzw. immer noch Fremdschlüssel auf Primärschlüssel
  - Alternativ als 10 mal Selektion mit Auftrag_ID=*value* auffassen:
    - Bei 6 Mio Posten und 1,5 Mio Aufträgen könnte man bei Annahme Gleichverteilung abschätzen, dass es 6 Mio/ 1,5 Mio = 4 Posten pro Auftrag gibt
    - Das heißt bei 10 Selektionen auf einen Auftrag: 4 · 10 = 40
  - (Es ist die letzte mögliche Operation, von daher ist die Kardinalität weniger relevant.)

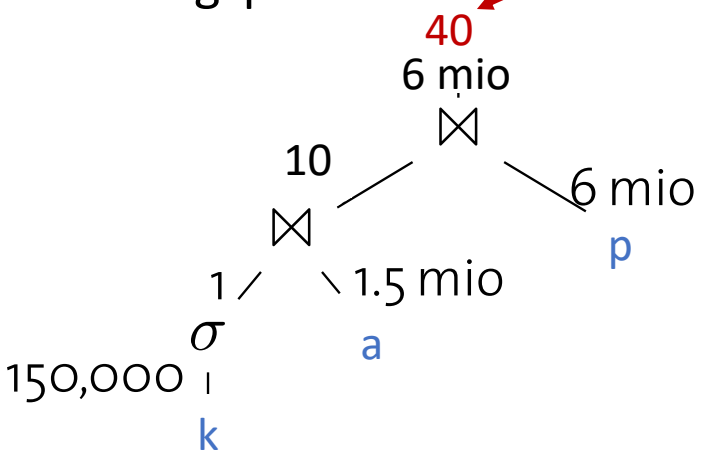


Kardinalitätsabschätzung: Beispiel

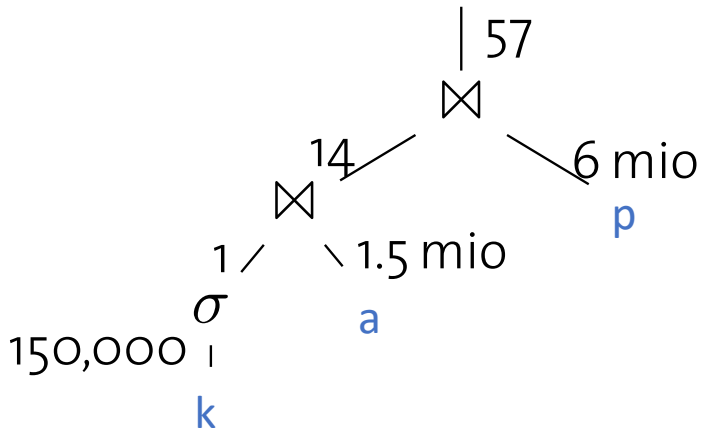
```
SELECT p.Teile_ID, p.Anzahl, p.Preis
FROM Auftragsposten p, Auftraege a, Kunden k
WHERE p.Auftrag_ID = a.Auftrag_ID
    AND a.Kunden_ID = k.Kunden_ID
    AND k.Name = 'IBM Corp.';
```

- |p| = 6 Mio
- |a| = 1,5 Mio
- |k| = 150.000
- Ausführungsplan:

Mit Zusatzüberlegung bzgl. Join als Selektion von der Vorfolie (rot)

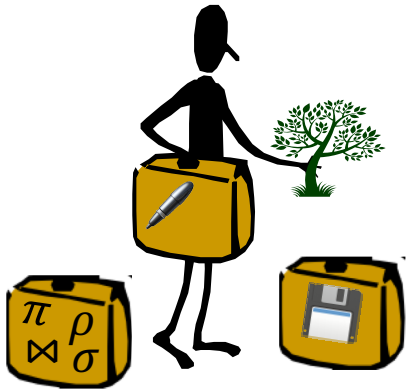


Tatsächliche Kardinalitäten:



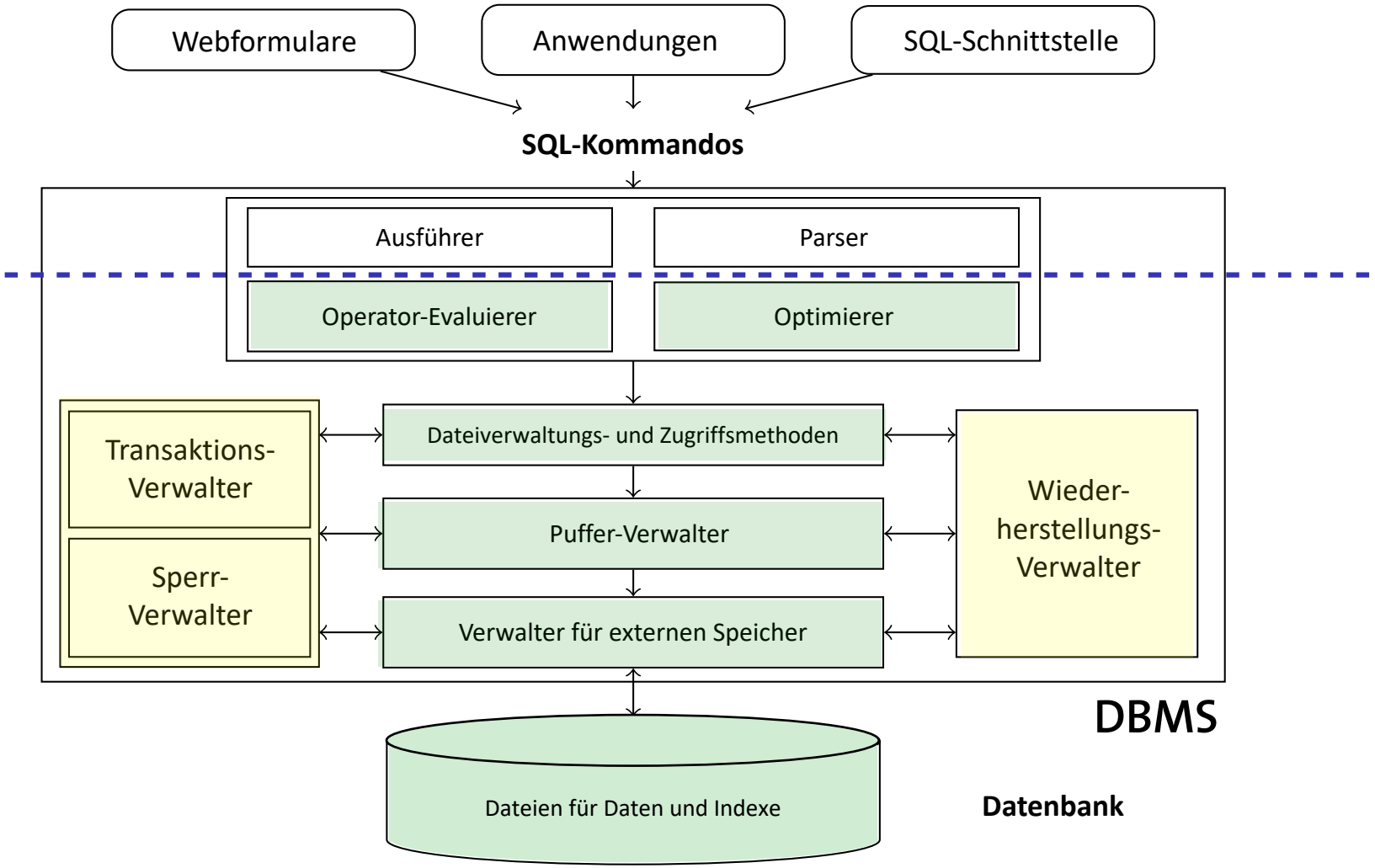
Zusammenfassung

- Umschreiber (Rewriter)
  - Logische Optimierung (unabhängig vom DB-Inhalt)
  - Prädikatsvereinfachung
  - Anfrageentschachtelung



- Datenabhängige Optimierung (Optimizer)
  - Bestimmung des „günstigsten“ Plan auf Basis
    - eines Kostenmodells (I/O-Kosten, CPU-Kosten) und
    - Statistiken (Histogramme) sowie
    - Physikalischen Planeigenschaften (interessante Ordnungen)

Noch zu diskutieren... Keine Angst - nächstes Mal!



Anhang

Pseudocode für die Search- und Insert-Operation im B^+-Baum

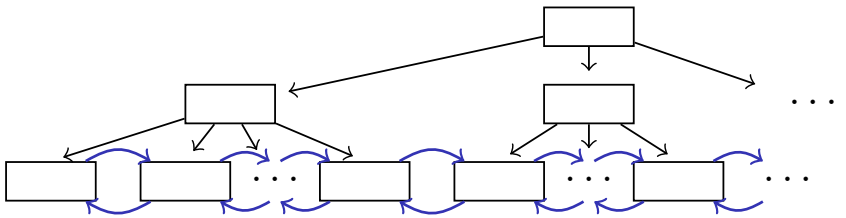
Algorithmus search

```
1  Function: search (k)
2  return tree_search (k, root);
```

```
1  Function: tree_search (k, node)
2  if node is a leaf then
3      return node;
4  switch k do
5      case k < k_1
6          return tree_search (k, p_0);
7      case k_i ≤ k < k_{i+1}
8          return tree_search (k, p_i);
9      case k_last < k
10         return tree_search (k, p_last);
```

$i < last \leq 2d$

- Funktionsaufruf search(k) bestimmt Blatt, das potentielle Treffer für eine Suche nach Elementen mit Suchschlüssel k enthält
- Entsprechend viele Fälle wie Suchschlüssel in node



Algorithmus insert

```
1 Function: insert (k, rid)
2 ⟨key, ptr⟩ ← tree_insert (k, rid, root); // root contains the root of the index tree
3 if key is not null then
4     allocate new root page r;
5     populate r with
6         p0 ← root;
7         k1 ← key;
8         p1 ← ptr;
9     root ← r ;
```

- insert(k, rid) wird von außen aufgerufen
- Blattknoten enthalten Rids, innere Knoten enthalten Zeiger auf andere B^+-Baum-Knoten

Algorithmus tree_insert

```
1  Function: tree_insert (k, rid, node)
2  if node is a leaf then
3      return leaf_insert (k, rid, node);
4  else
5      switch k do
6          case k < k_1
7              ⟨sep, ptr⟩ ← tree_insert (k, rid, p_o);
8          case k_i ≤ k < k_{i+1}
9              ⟨sep, ptr⟩ ← tree_insert (k, rid, p_i);
10         case k_last < k
11             ⟨sep, ptr⟩ ← tree_insert (k, rid, p_last);
12     if sep is null then
13         return ⟨null, null⟩;
14     else
15         return split (sep, ptr, node);
```

see `tree_search ()`

$i < last \leq 2d$

Algorithmus leaf_insert

1 **Function:** `leaf_insert` $(k, rid, node)$
2 **if** another entry fits into *node* **then**
3 insert $\langle k, rid \rangle$ into *node* ;
4 **return** $\langle$**null**, **null**$\rangle$;
5 **else**
6 allocate new leaf page p ;
7 **take** $\{\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle\}$:= entries from $node \cup \{\langle k, ptr \rangle\}$
8 leave entries $\langle k_1^+, p_1^+ \rangle, \ldots, \langle k_d^+, p_d^+ \rangle$ in *node* ;
9 move entries $\langle k_{d+1}^+, p_{d+1}^+ \rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+ \rangle$ to p ;
10 **return** $\langle k_{d+1}^+, p \rangle$;

Algorithmus split

1 **Function:** `split` $(k, ptr, node)$

2 **if** another entry fits into *node* **then**

3 insert $\langle k, ptr \rangle$ into *node* ;

4 **return** $\langle$**null**, **null**$\rangle$;

5 **else**

6 allocate new node p;

7 **take** $\{\langle k_1^+, p_1^+\rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+\rangle\}$:= entries from *node* $\cup$ $\{\langle k, ptr\rangle\}$

8 leave entries $\langle k_1^+, p_1^+\rangle, \ldots, \langle k_d^+, p_d^+\rangle$ in *node* ;

9 move entries $\langle k_{d+2}^+, p_{d+2}^+\rangle, \ldots, \langle k_{2d+1}^+, p_{2d+1}^+\rangle$ to p ;

10 set $p_0 \leftarrow p_{d+1}^+$ in *node*;

11 **return** $\langle k_{d+1}^+, p\rangle$;

Der erste Zeiger p_0 in der neuen Seite p wird auf p_{d+1} gesetzt. Dieser Zeiger bildet die Trennstelle, kommt demnach nicht mehr auf der alten Seite *node* vor.