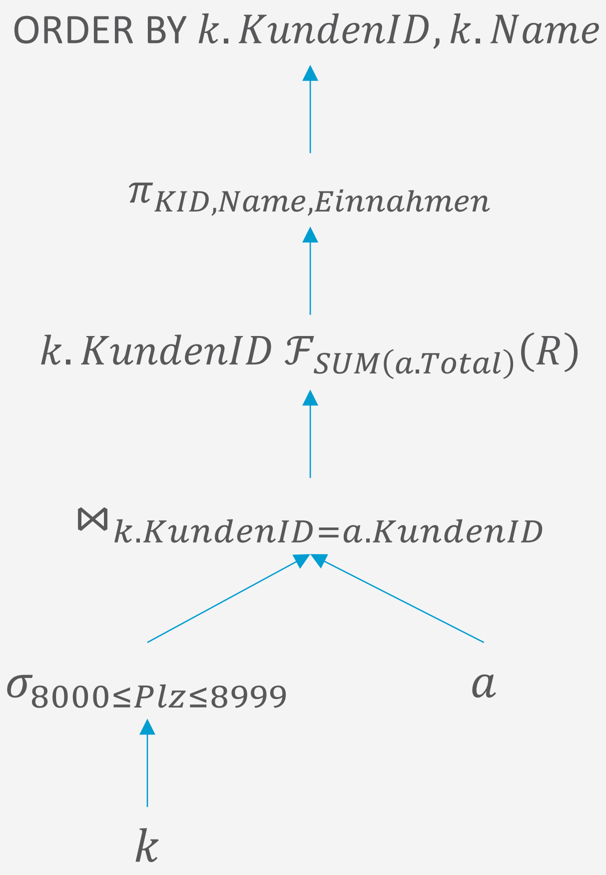
WWU MÜNSTER

Anfrageverarbeitung

Datenbanken



Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Danksagung

- Folien basieren ursprünglich auf dem Kurs

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

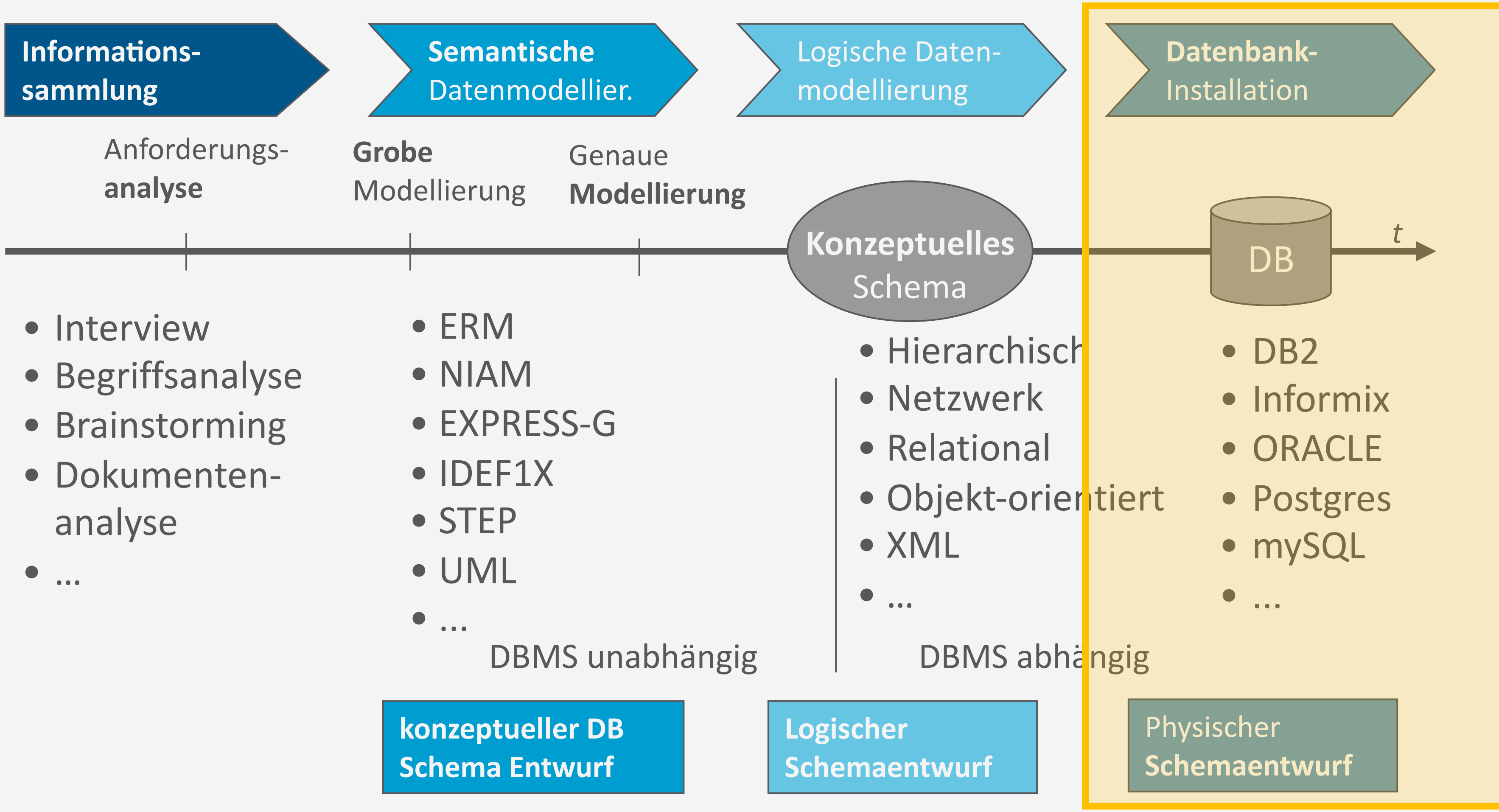
- Graphiken wurden mit Zustimmung des Autors aus diesem Kurs übernommen

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



Anfragebeantwortung

- DBMS muss eine Menge von Aufgaben erledigen
  - Mit minimalen Ressourcen
  - Über großen Datenmengen
  - So schnell wie möglich

Selektion

Projektion

Aggregation

```
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;
```

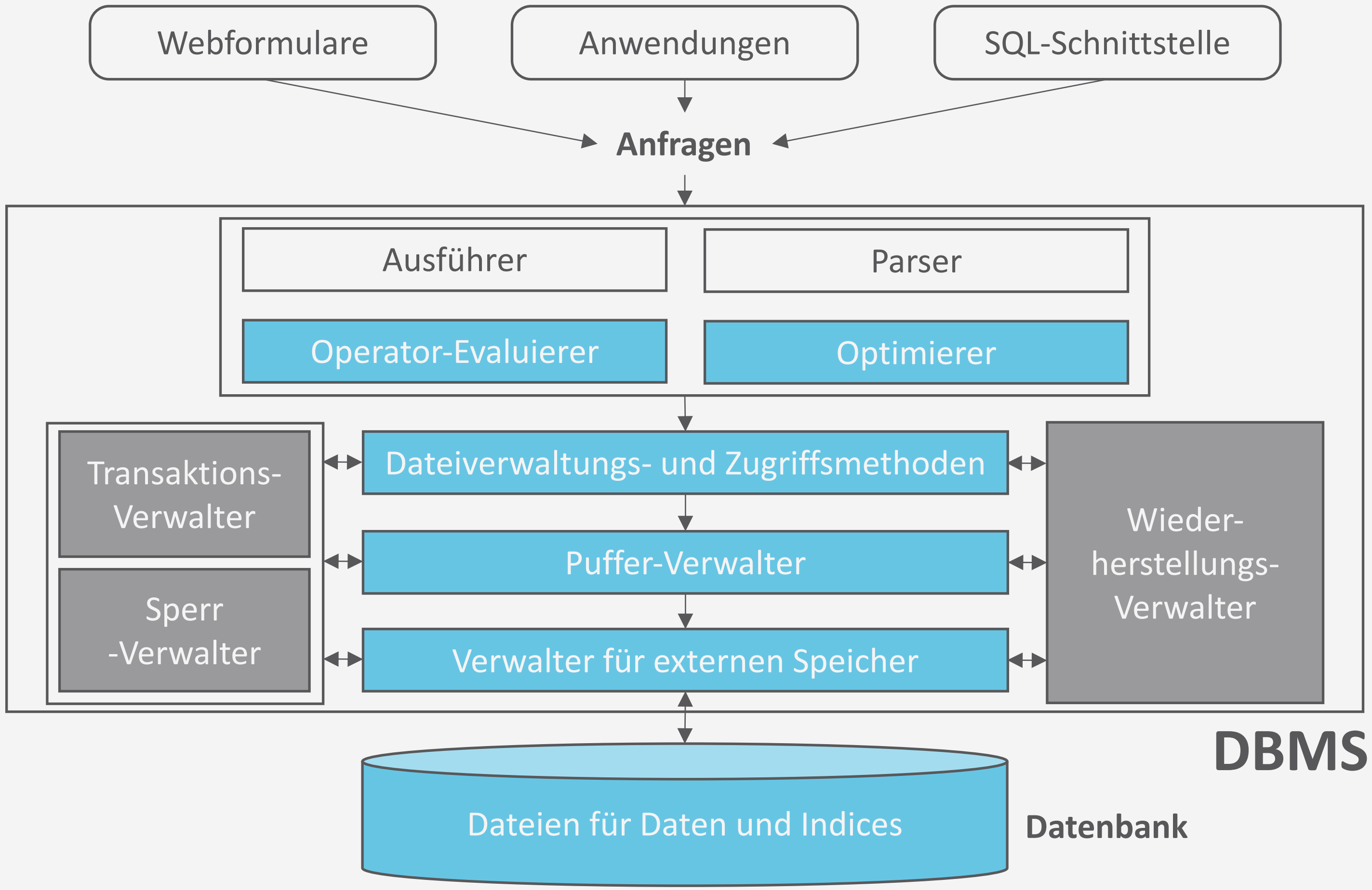
Gruppierung

Join

Sortierung

Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
  - Puffer
  - Zugriff
- Anfrageverarbeitung
  - Operator-Evaluierer
  - Optimierer
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



Überblick: 6. Anfrageverarbeitung

A. Speicherung
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

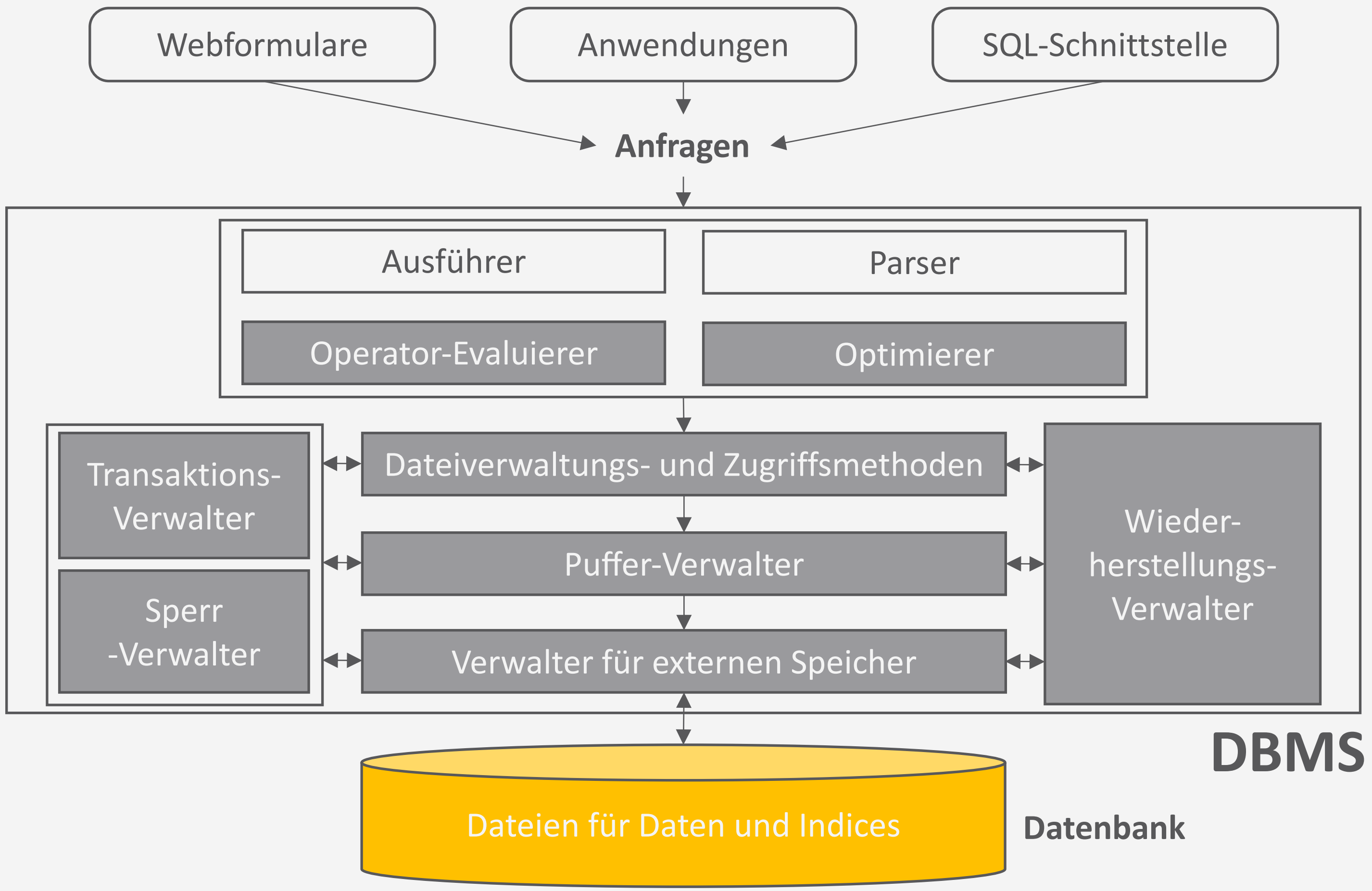
B. Indexierung
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. Anfragebeantwortung
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. Anfrageoptimierung
- Rewriting
- Datenabhängige Optimierung

Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
  - Puffer
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



Speicherung

- Daten in Datenbanken sind in der Regel
  - 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)

- Wir schauen uns Festplattenspeicher als Beispiel an
  - Ob Festplattenspeicher, Netzwerkspeicher oder anderes → Bottlenecks / Anforderungen kennen

Speicherhierarchie

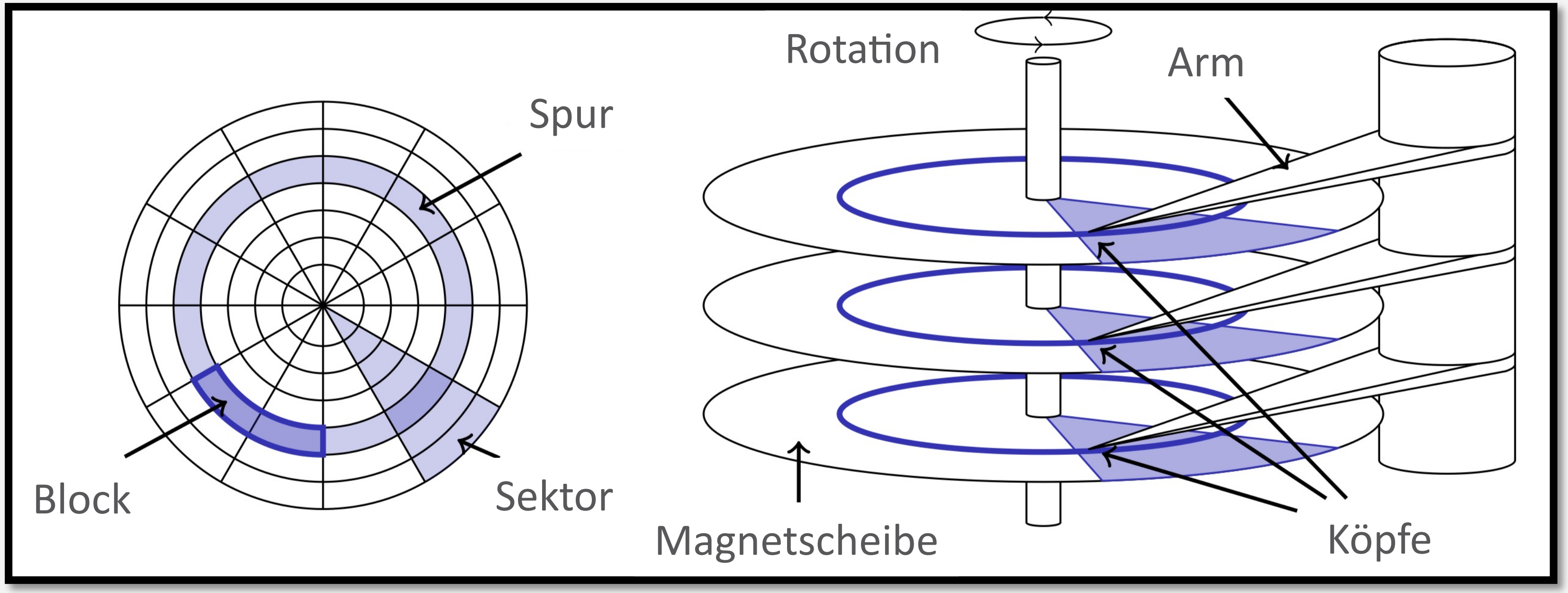
	Kapazität	Latenz
• CPU (mit Registern)	Bytes	< 1 ns
• Cache-Speicher	Kilo-/Mega-Bytes	< 10 ns
• Hauptspeicher	Giga-Bytes	20-100 ns
• Flash-Speicher / SSD	Giga/Tera/Peta-Bytes	30-250 µs
• Festplatte	Tera/Peta-Bytes	3-10 ms
• Bandautomat	Peta-Bytes	variierend

- 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. Lese- bzw. Schreibezeit (Transferzeit t_{tr})
- Zugriffszeit: $t = t_s + t_r + t_{tr}$

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

Sequentieller vs. Wahlfreier Zugriff

- Fortsetzung Travelstar Beispiel: Lese 1000 Blöcke von je 8 KB (8MB)
  - Wahlfreier Zugriff:
    - $t_{rand} = 1000 \cdot 14{,}33\ ms = 14.330\ ms$
  - Sequentieller Zugriff:
    - 63 Sektoren / Spur, Track-to-Track-Suchzeit $t_{s,t2t}$ (von einer Spur zur nächsten Spur): $1\ ms$
    - 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
    - $t_{seq} = t_s + t_r + 1000 \cdot t_{tr} + 254 \cdot t_{s,t2t} \approx 10ms + 4{,}17ms + 1000 \cdot 0{,}16ms + 254 \cdot 1\ ms \approx 428ms$
- Einsicht: Sequentieller Zugriff **viel** schneller als wahlfreier → Vermeide wahlfreie I/O, wenn möglich
  - Wenn $428\ ms/14330\ ms \approx 3\%$ einer 8MB Datei wahlfrei benötigt wird, kann man gleich die ganze Datei lesen, sofern Blöcke hintereinander stehen

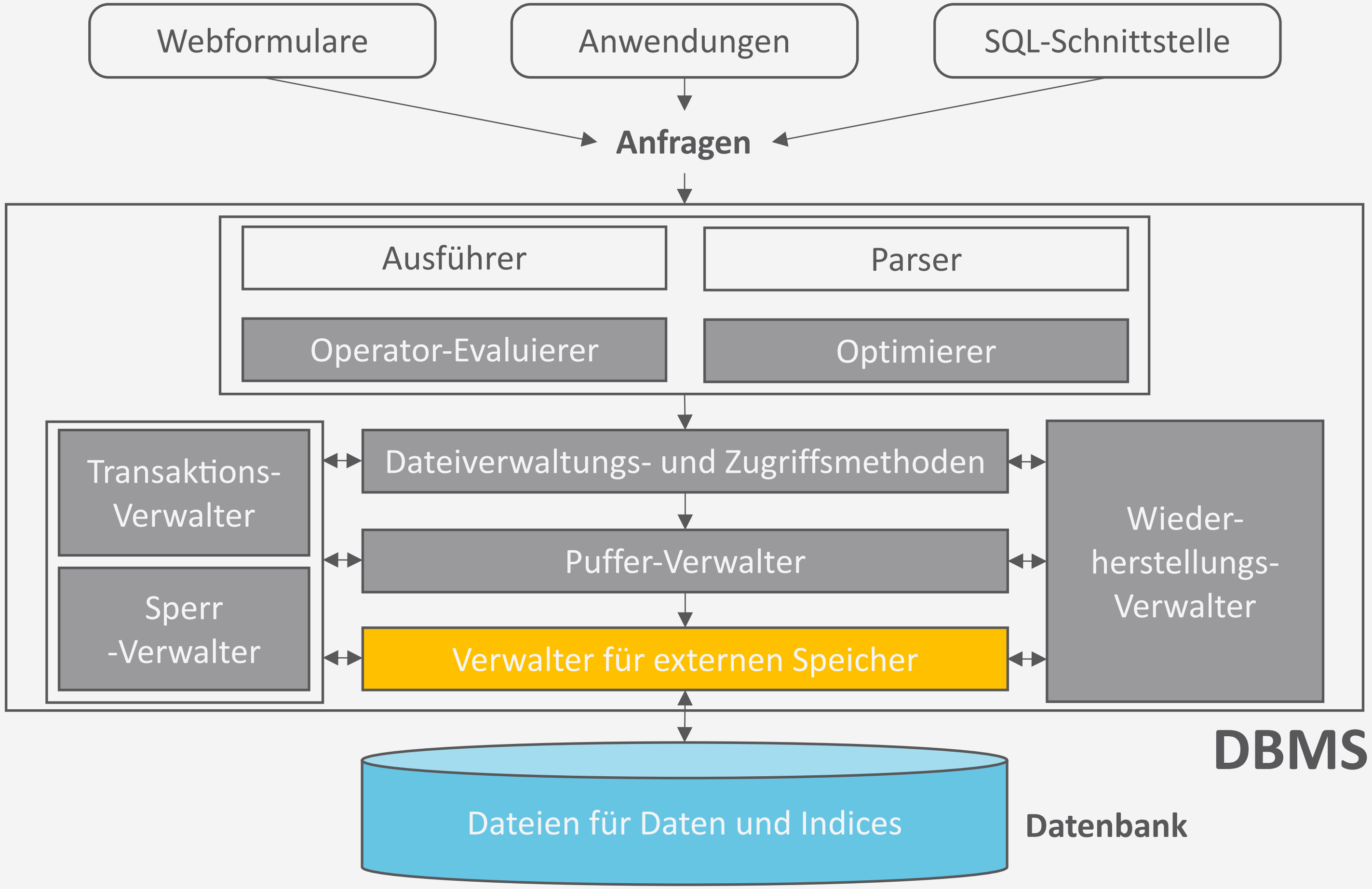
Speichernetzwerk (Storage Area Network, SAN)

- 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, 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

- Speicherung
  - Speichermedien
    - Datenbanken in der Regel nicht im Hauptspeicher vorhaltbar
    - Wahlfreier Zugriff teuer im Vergleich zu sequentiellem Zugriff
  - Verwaltung
  - Puffer
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



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

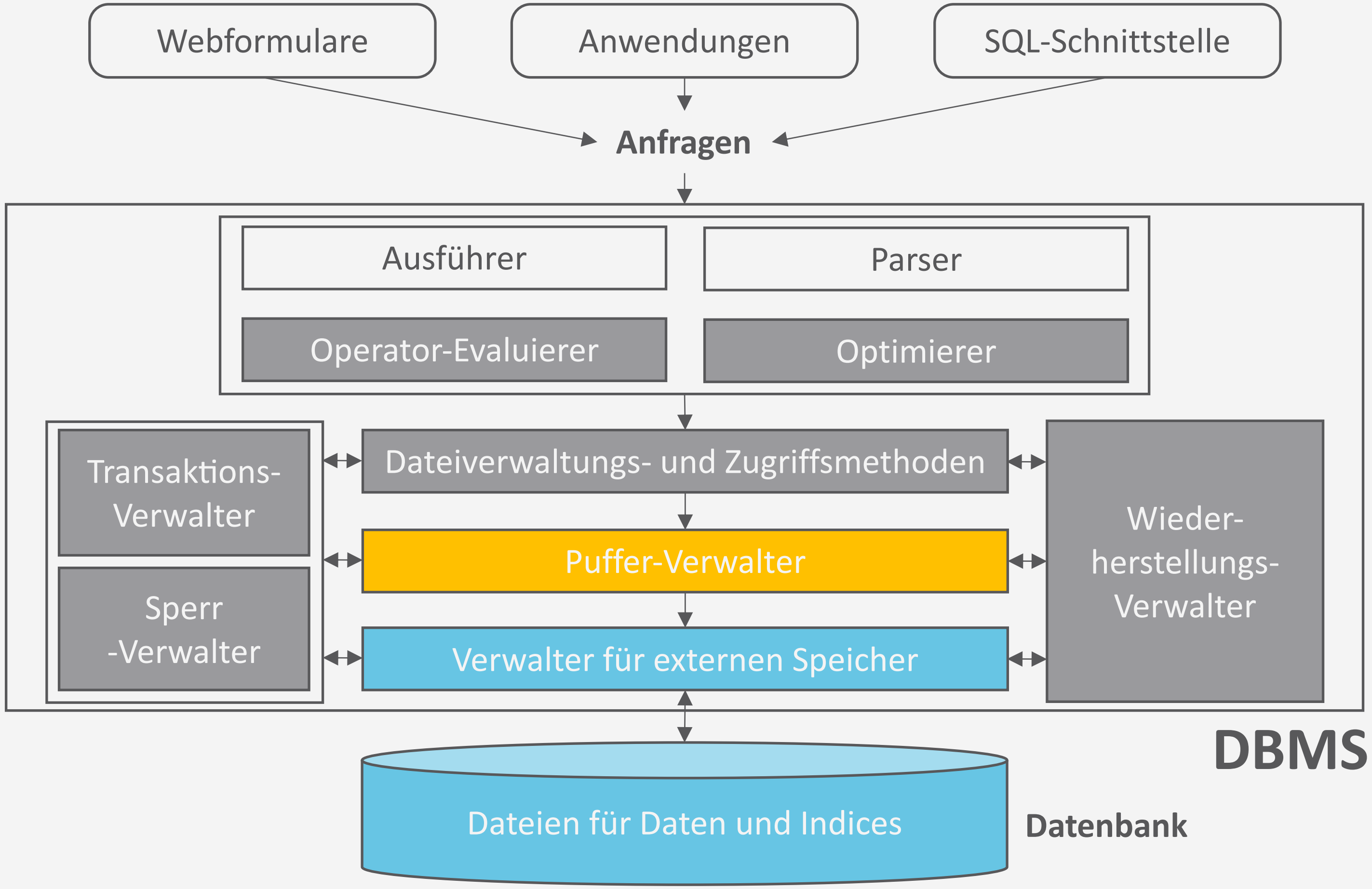
  sein kann

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
     - Hinzufügen, 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

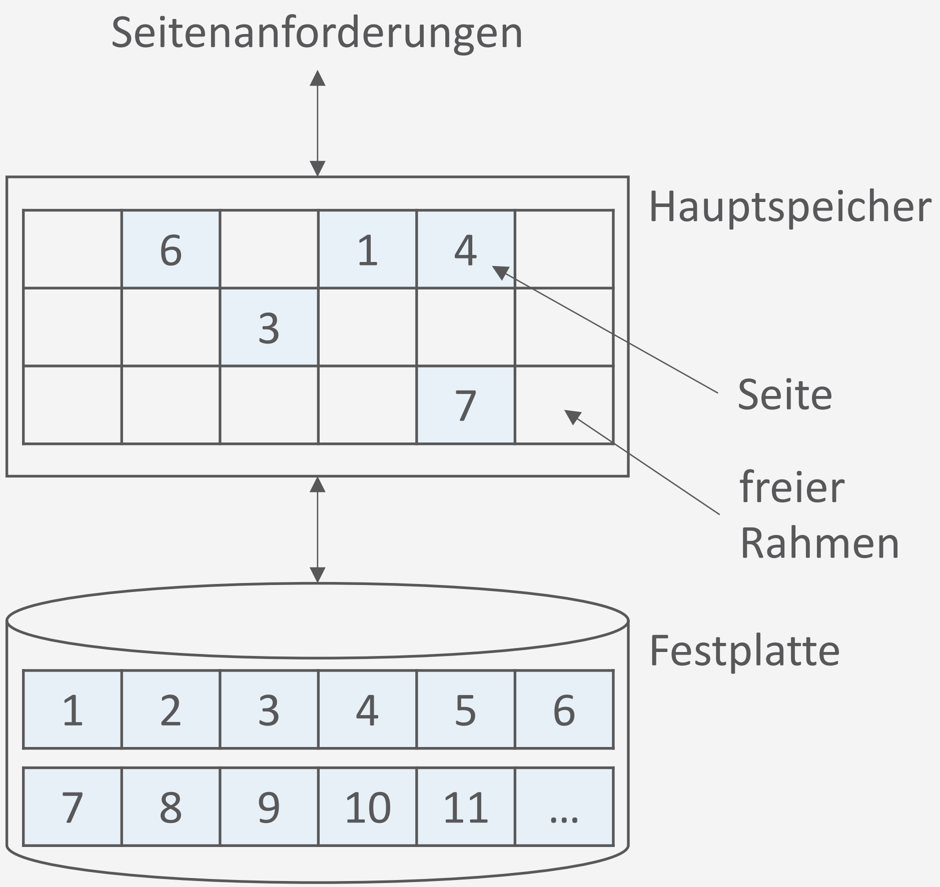
Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
    - Seiten zur Referenz von Speichereinheiten
  - Puffer
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



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
  - $\mathrm{pin}(pageno)$
    - Anfrage nach Seitennummer $pageno$
    - Lade Seite in Hauptspeicher falls nötig
    - Rückgabe einer Referenz auf $pageno$
- Funktion unpin für Freistellungen von Seiten nach Verwendung
  - $\mathrm{unpin}(pageno, dirty)$
    - Freistellung einer Seite $pageno$ zur möglichen Auslagerung
    - $dirty = \mathrm{true}$ bei Modifikationen der Seite

Wofür nötig?

Implementation von $\mathrm{pin}()$

```
function pin(pageno)
    if buffer pool already contains pageno then
        pinCount(pageno) ← pinCount(pageno) + 1
        return address of frame holding pageno
    else
        Select a victim frame v using the replacement policy
        if dirty(v) then
            Write v to disk
        Read page pageno from disk into frame v
        pinCount(pageno) ← 1
        dirty(pageno) ← false
        return address of frame v
```

Wofür nötig?

Implementation von unpin()

function $\mathrm{unpin}(pageno, dirty)$
    $pinCount(pageno) \leftarrow pinCount(pageno) - 1$
    if $dirty$ **then**
        $dirty(pageno) \leftarrow dirty$

Warum werden Seiten nicht gleich beim unpin zurückgeschrieben?

Ersetzungsstrategien

- Least Recently Used (LRU)
  - Verdrängung der Seite mit am längsten zurückliegendem $\mathrm{unpin}()$
- LRU-k
  - Wie LRU, aber k-letztes $\mathrm{unpin}()$, nicht letztes
- Most Recently Used (MRU)
  - Verdrängung der Seite mit jüngstem $\mathrm{unpin}()$
- Random
  - Verdrängung einer beliebigen Seite
- Viele mehr...
- Seite muss pincount = 0 haben, um für Ersetzung zur Verfügung zu stehen

Was, wenn es keine Seite mit pincount = 0 gibt?

Pufferverwaltung in der Praxis

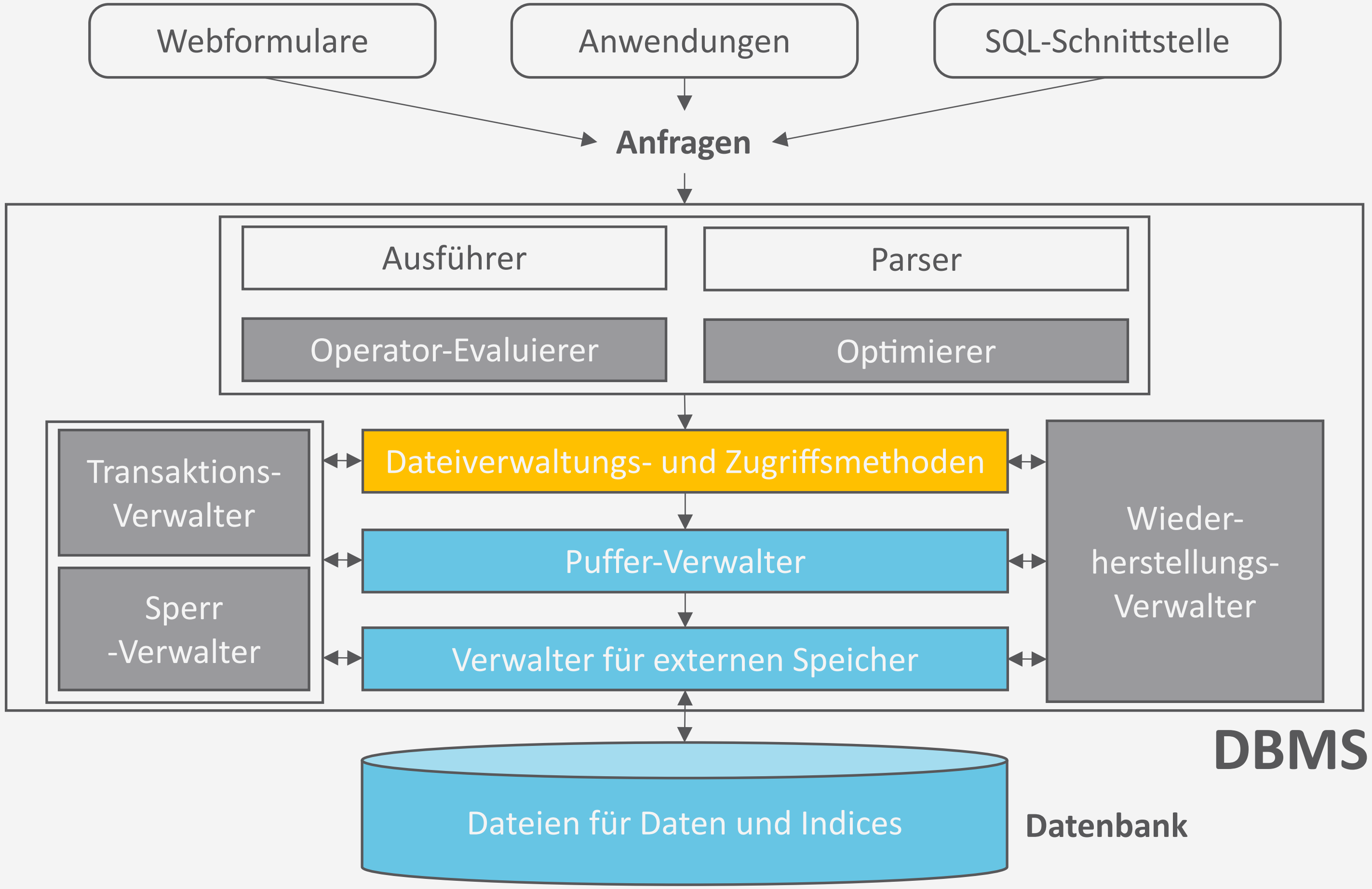
- Prefetching
  - Antizipation von Anfragen, um CPU- und I/O-Aktivität zu überlappen
    - Spekulatives Prefetching: Nehme sequentiellen Seitenzugriff an und lese im Vorwege
    - Prefetch-Listen mit Instruktionen für den Pufferverwalter für Prefetch-Seiten
- Fixierungs- oder Verdrängungsempfehlung
  - Höherer Code kann Fixierung (z.B. für Indexseiten) oder schnelle Verdrängung (bei sequentiellen 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
- Daher: DBMS schaltet oft Betriebssystemdienste aus
  - Direkter Zugriff auf Festplatten
  - Eigene Prozessverwaltung

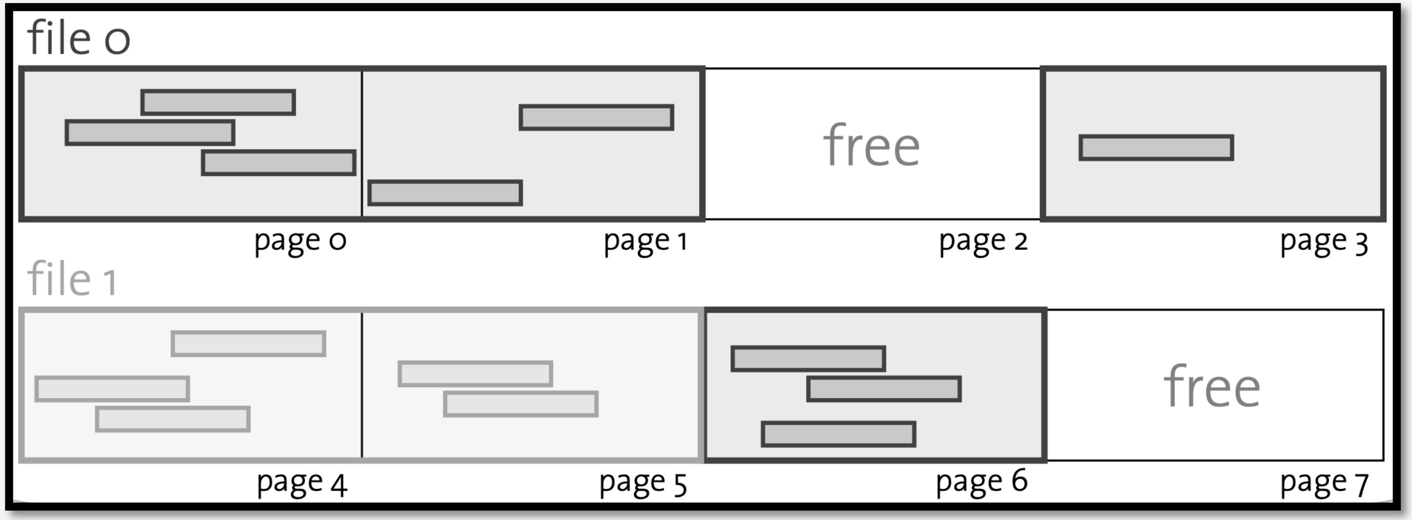
Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
  - Puffer
    - Rahmen im Pufferbereich um Seiten aus externem Speicher in den Hauptspeicher zu laden
    - Verdrängungsstrategien
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



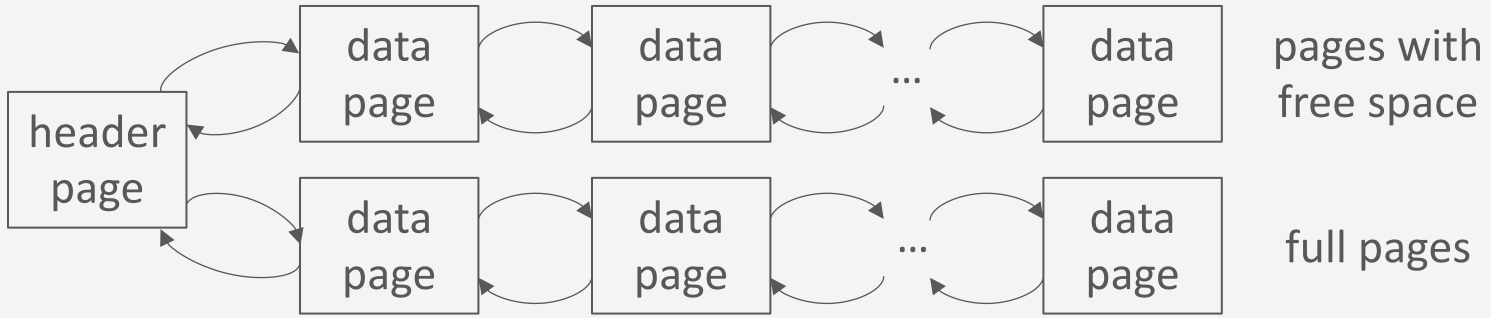
Datenbank-Dateien

- 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 einen 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



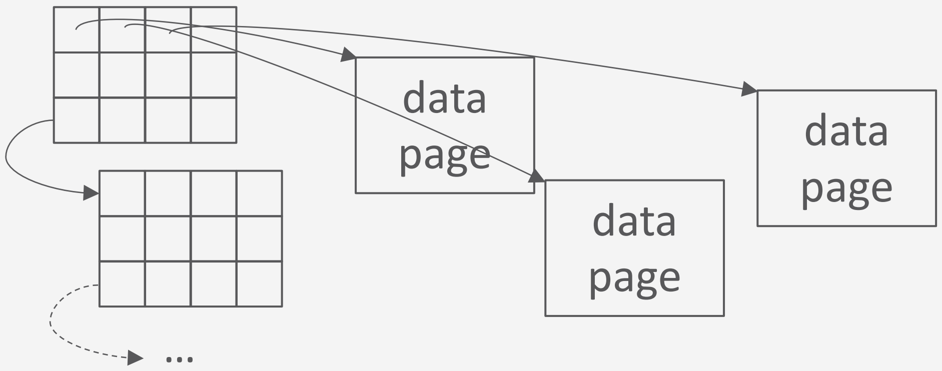
✓ Einfach zu implementieren

✗ 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 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

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

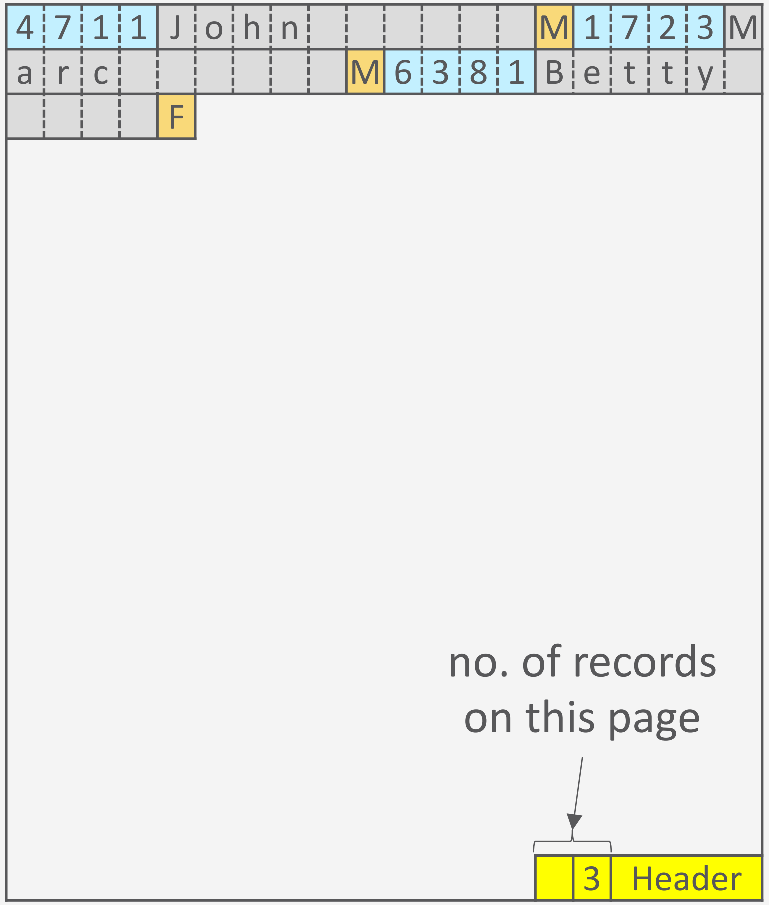
$$rid = \langle pageno, slotno \rangle$$

  - Slot: Speicherplatz für einen Datensatz
  - Datensatz-Position (Versatz auf der Seite):

$$slotno \cdot \text{Bytes pro Slot}$$

    - Beginnend bei 0 (sowohl erstes Bit auf der Seite, als auch Slotnummer)
    - Funktioniert so nur bei konstanter Slotgröße
- Beispiel: Angenommen Seite sei 42
  - Referenz Datensatz mit ID 6381:
    - Kennung ist $\langle 42{,}2 \rangle$

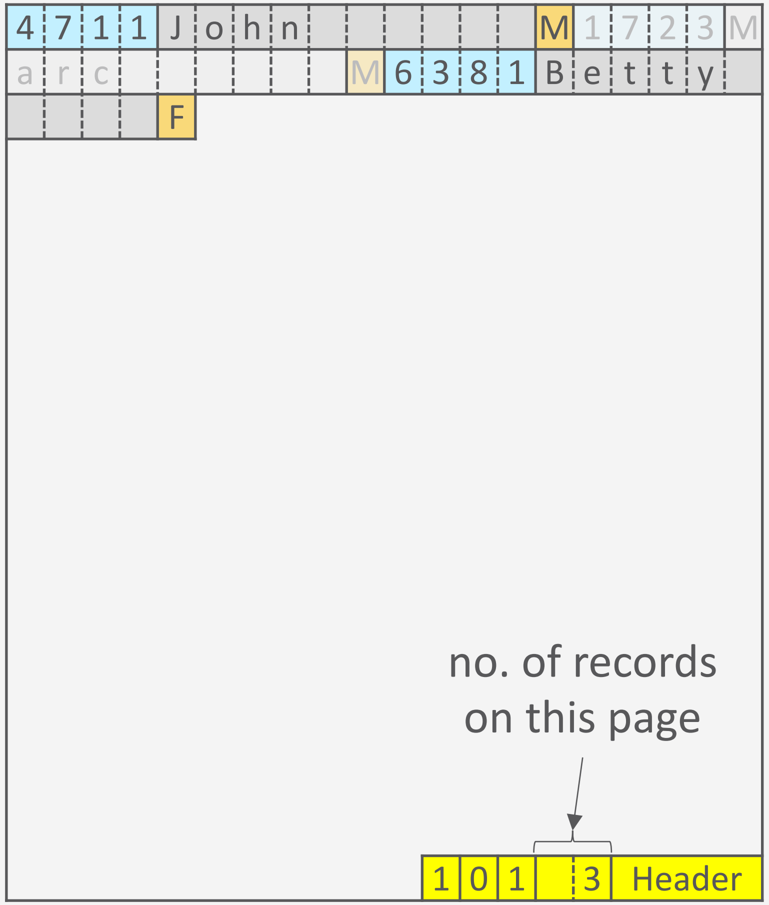
ID	Name	Sex
4711	John	M
1723	Marc	M
6381	Betty	F



Inhalte einer Seite

- Datensatz gelöscht → rid sollte sich nicht ändern
  - Slot-Verzeichnis (Bitmap) markiert durch 1/0, ob Datensatz noch gültig
- Beispiel Fortsetzung:
  - Nach Löschung von Datensatz mit ID 1723
  - Referenz Datensatz mit ID 6381
    - Kennung ist immer noch ⟨42,2⟩

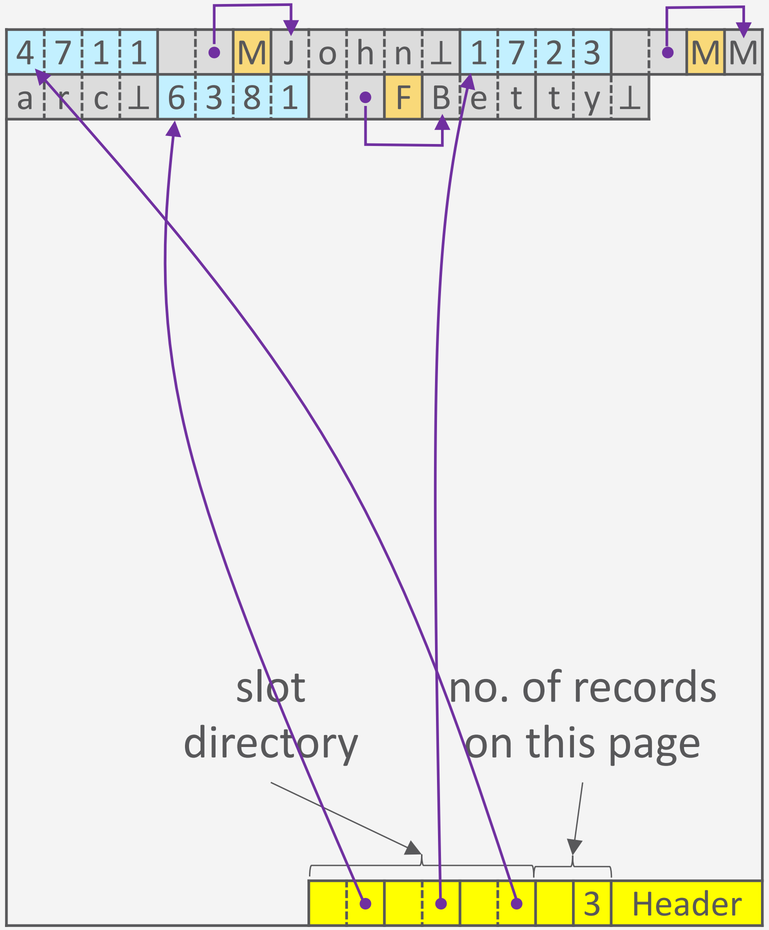
ID	Name	Sex
4711	John	M
1723	Marc	M
6381	Betty	F



Inhalte einer Seite: Felder variabler Länge

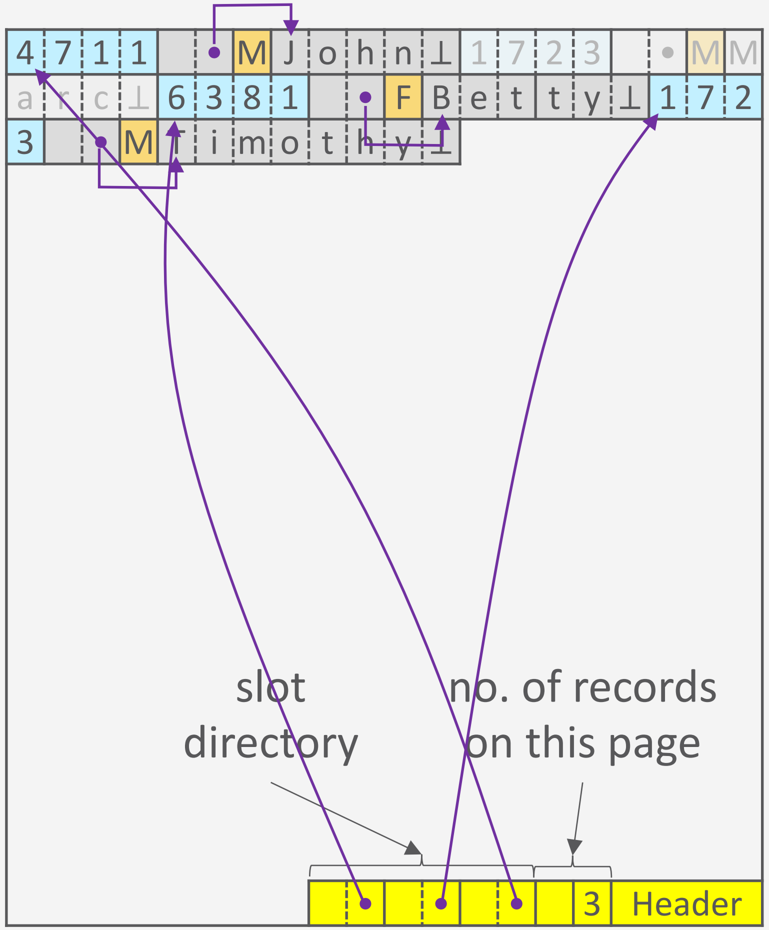
- Felder variabler Länge zum Ende verschoben
  - Platzhalter zeigt auf Position
- Dann brauchen wir ein Slot-Verzeichnis
  - Zeigt auf Start eines Feldes
  - Datensatz-Kennung bleibt bei $rid = \langle pageno, slotno \rangle$, verweist jetzt aber auf Position im Slot-Verzeichnis
  - Damit können auch Datensätze unterschiedlicher Tabellen mit unterschiedlicher Länge pro Datensatz auf einer Seite gespeichert werden

Warum?



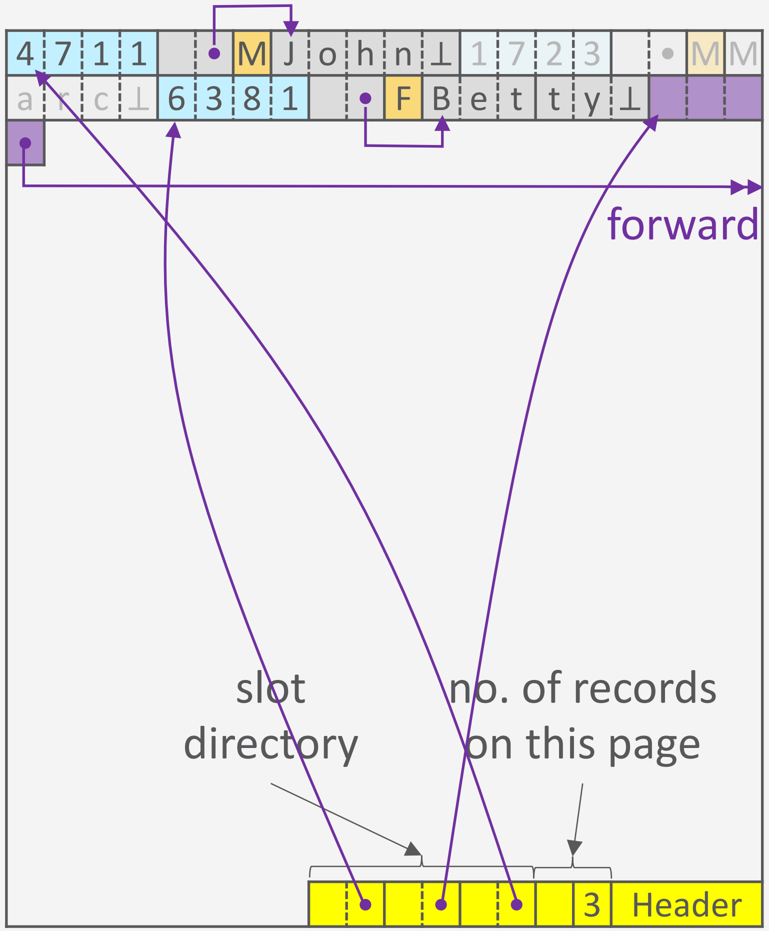
Inhalte einer Seite: Felder variabler Länge

- Felder können auf Seite verschoben werden (z.B. wenn sich Feldgröße ändert)
- Beispiel:
  - Datensatz mit ID 1723 hat Kennung ⟨42,1⟩
  - Name aktualisieren zu Timothy, was nicht in den Platz passt
  - Umkopieren an neue Stelle
    - Referenz in Slot-Verzeichnis aktualisieren
    - Kennung bleibt ⟨42,1⟩



Inhalte einer Seite: Felder variabler Länge

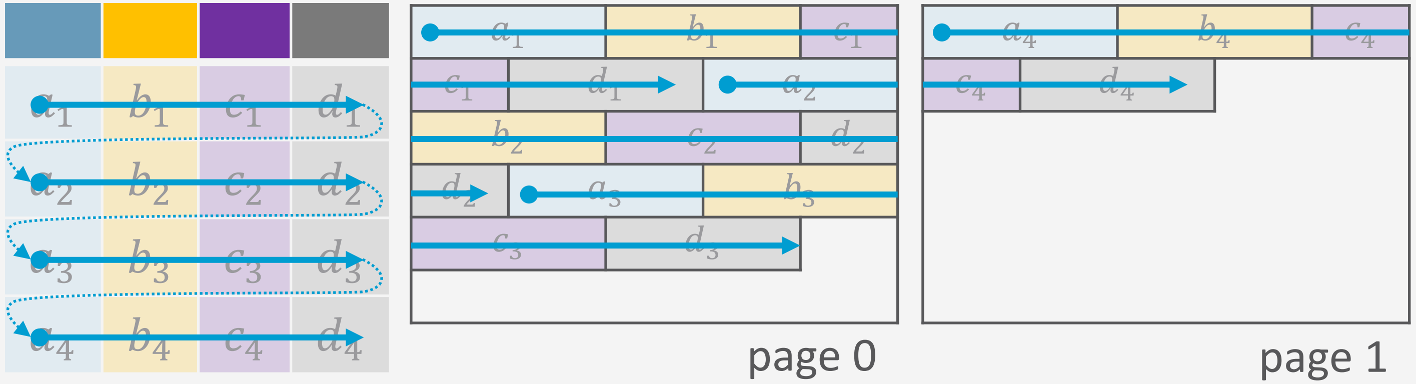
- Felder können auf Seite verschoben werden (z.B. wenn sich Feldgröße ändert)
- Beispiel:
  - Datensatz mit ID 1723 hat Kennung ⟨42,1⟩
  - Name aktualisieren zu Timothy, was nicht in den Platz passt
  - Umkopieren an neue Stelle
    - Referenz in Slot-Verzeichnis aktualisieren
    - Kennung bleibt ⟨42,1⟩
- Einführung einer Vorwärtsreferenz, wenn Feld nicht auf Seite passt



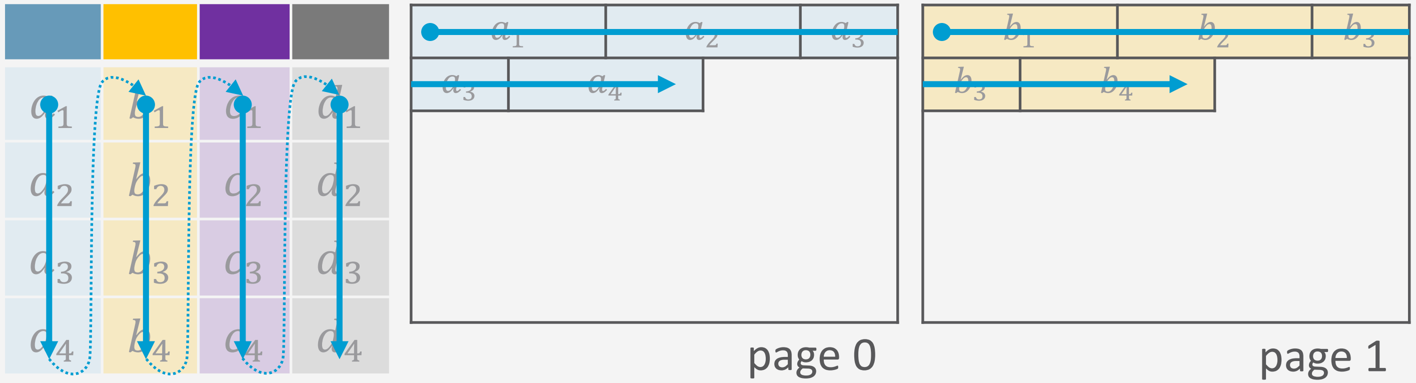
Alternative Seiteneinteilungen

Wann lohnt sich welche Einteilung?

- Im Beispiel wurden Datensätzen zeilenweise angeordnet:



- Spaltenweise Anordnung genauso möglich:

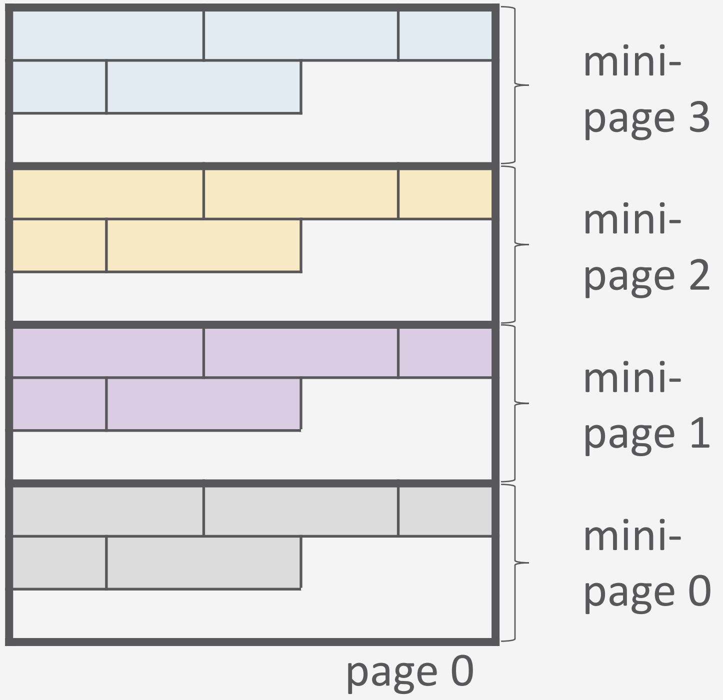


Alternative Seitenanordnungen

Vorgestellte Schemata heißen auch:

- Row-Store
- Column-Store

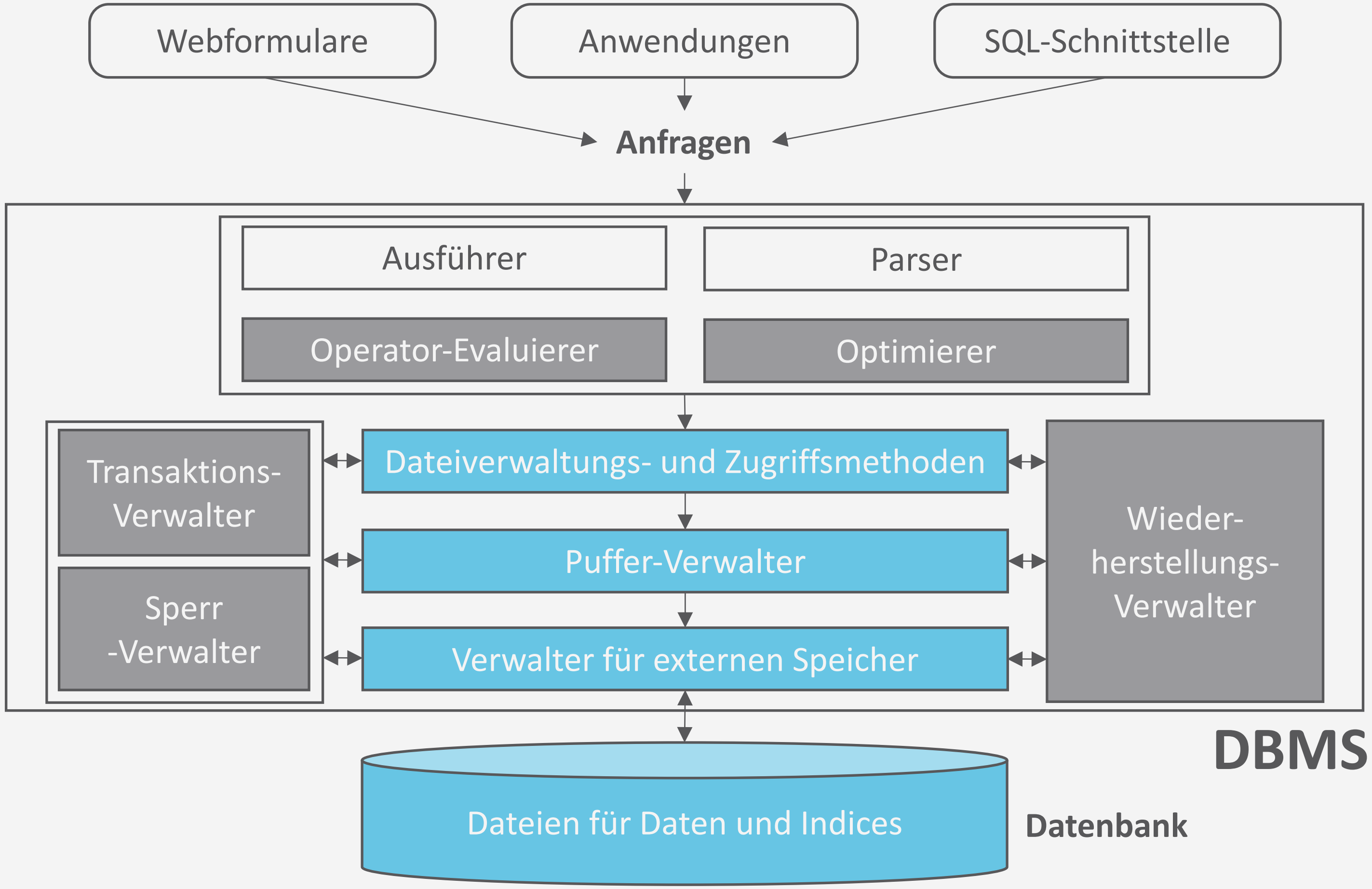
- Anwendungen für verschiedene Lasttypen und Anwendungskontexte
- Unterschiedliche Kompressionsmöglichkeiten
- Kombination möglich:
  - Unterteilung einer Seite in Miniseiten
  - Mit entsprechender Aufteilung



Ailamaki et al. Weaving Relations for Cache Performance. VLDB **2001**

Zwischenzusammenfassung

- Speichermedien
  - DBs idR zu groß für Hauptspeicher
  - Wahlfreier Zugriff teuer
- Verwaltung
  - Seiten als Referenz auf Speicher
- Puffer
  - Seiten aus externem Speicher in Rahmen des Puffer zu laden
  - Verdrängungsstrategien
- Zugriff
  - Stabile Datensatz-Kennung rid zur Identifikation der Speicherposition



Überblick: 6. Anfrageverarbeitung

A. *Speicherung*
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

B. ***Indexierung***
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. *Anfragebeantwortung*
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

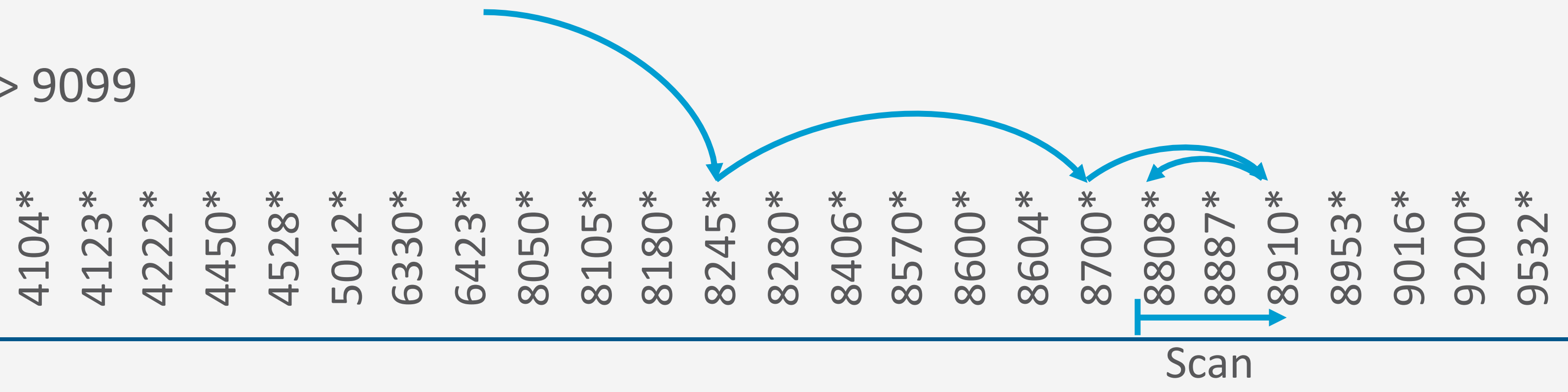
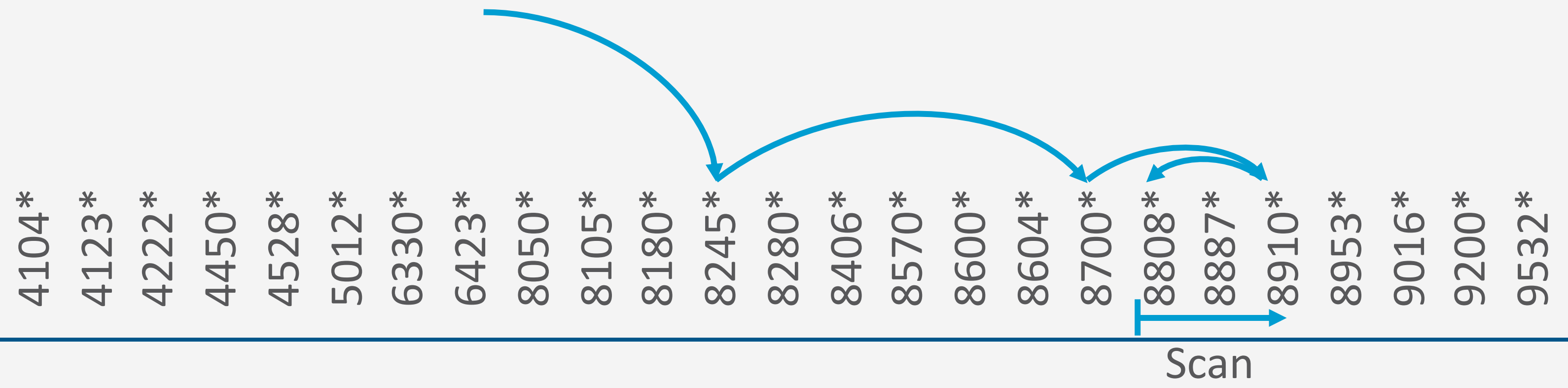
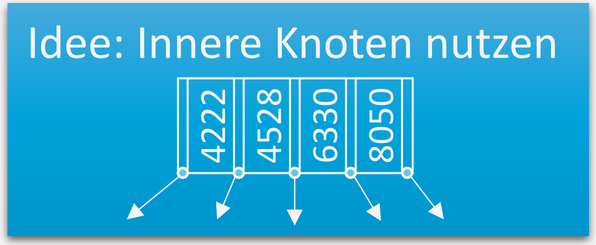
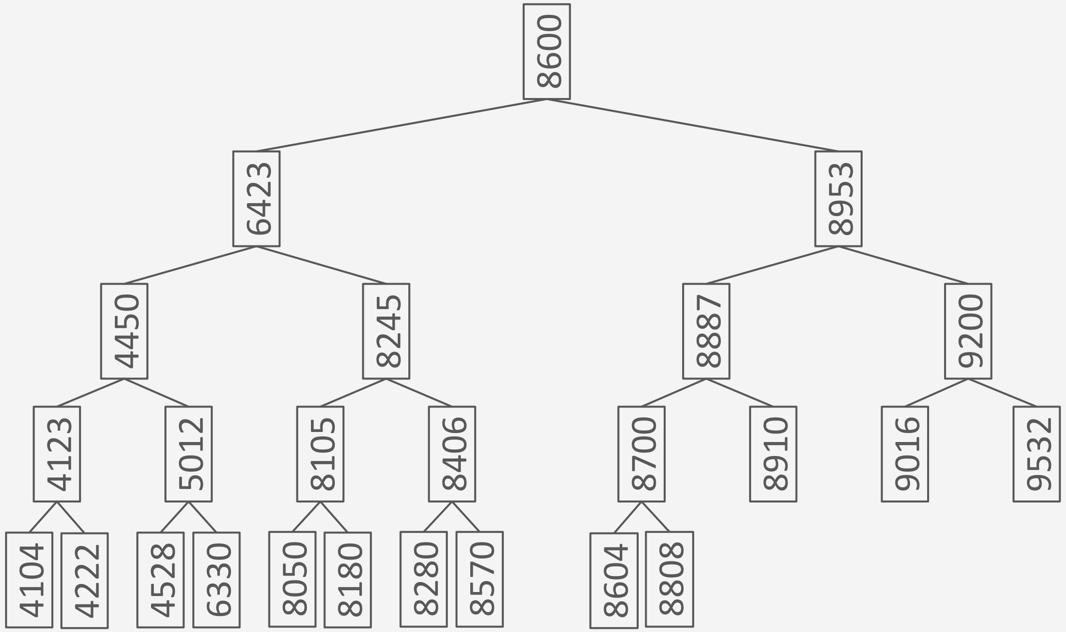
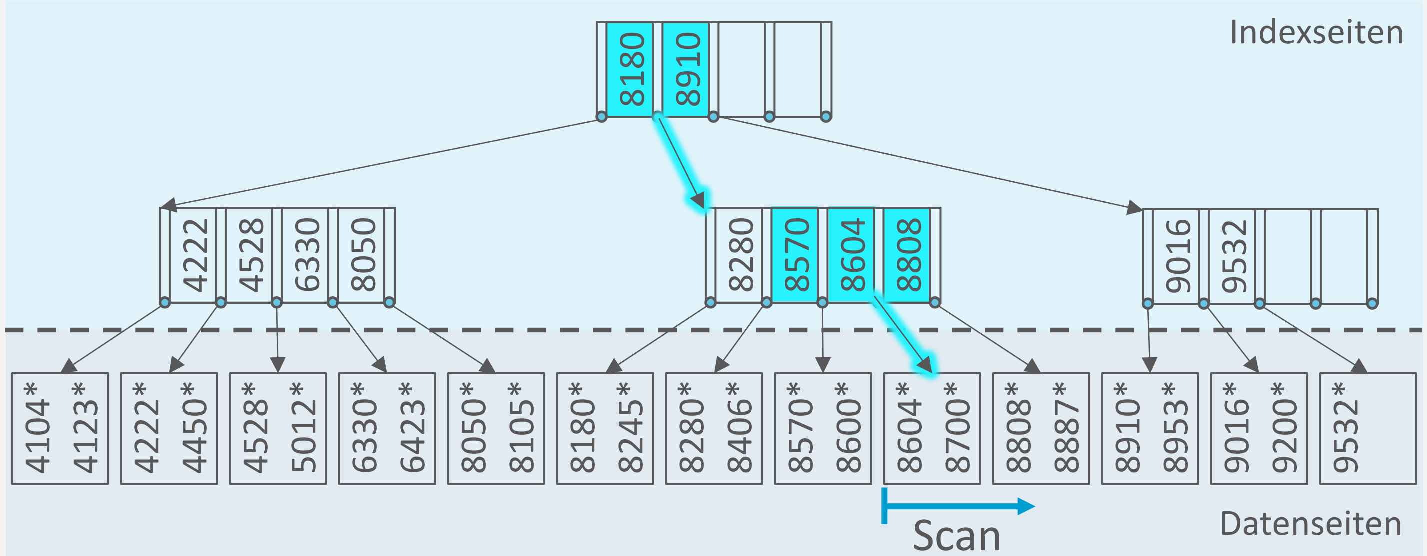
D. *Anfrageoptimierung*
- Rewriting
- Datenabhängige Optimierung

Effiziente Evaluierung einer Anfrage

- Beispiel
  - ```
    select *
    from Kunden
    where Plz between 8800 and 9099;
    ```
  - Auswertung
    1. Sortierung der Tabelle Kunden auf der Platte (nach Plz)
       - Effizienter Sortieralgorithmus benötigt
    2. Suche Startpunkt, an dem PLZ ≥8800
       - Binärsuche für Effizienz
    3. Scanne Datensätze, bis Plz > 9099

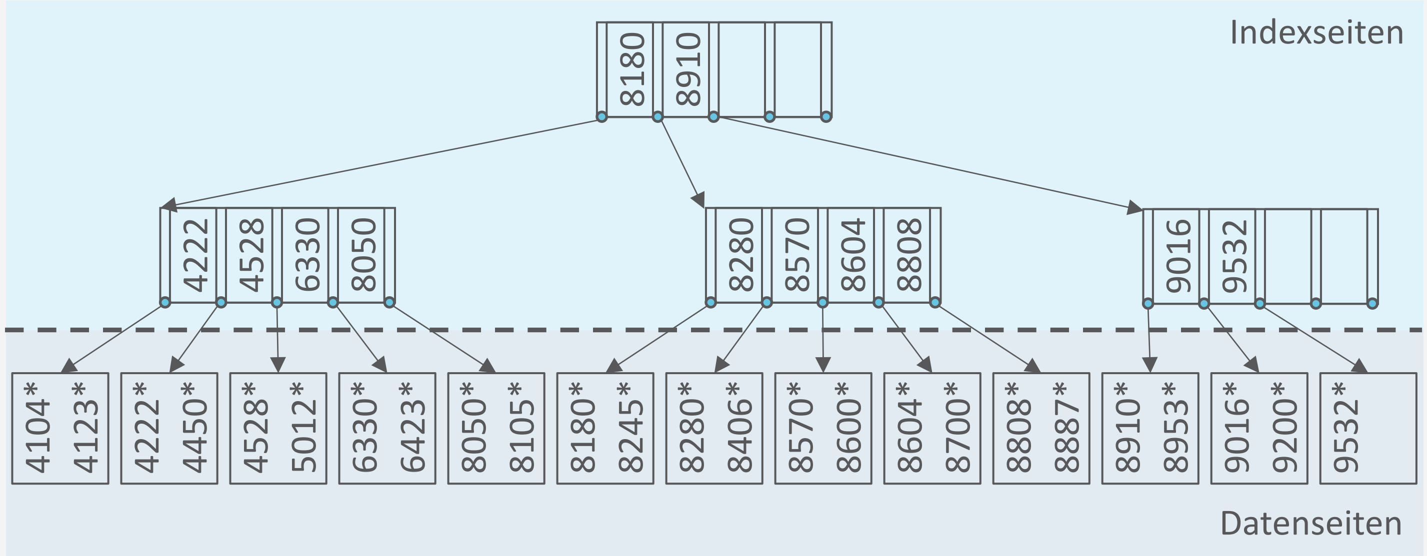
function binarySearch$(x, list)$
  $mid = list.length/2$
  if $x = list[mid]$ **then**
    return reference to $list[mid]$
  else if $x > list[mid]$ **then**
    binarySearch$(x, list[mid+1: end])$
  else
    binarySearch$(x, list[start: mid-1])$

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


    ```

Geordnete Dateien und binäre Suche

✓ Sequentieller Zugriff während der Scan-Phase

✓ $\log_2(\#Tupel)$ während der Such-Phase + entsprechende Tupel während der Scan-Phase lesen (statt insgesamt alle bei unsortierten Datensätzen; plus natürlich Sortieraufwand)

✗ 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



Helfen Bäume?

- Suchen innerhalb einer Seite im Hauptspeicher effizient machbar
  - Ziel: möglichst wenige Seiten aus Sekundärspeicher laden
- Bäume als Navigationsstruktur
  - Beispiel Binärbaum
    - Innere Knoten: Schlüsselwerte
    - Linker Unterbaum: Alle Schlüsselwerte kleiner
    - Rechter Unterbaum: Alle Schlüsselwerte größer
    - Tiefe bei Ausgeglichenheit: $\log_2(\#Tupel)$





Index-basierte Lösung

- Zurück zum Beispiel
  - ```
    select *
    from Kunden
    where Plz between 8800 and 9099;
    ```
  - Auswertung:
    1. Suche Startpunkt in Index, an dem PLZ ≥ 8800
    2. Finde Startpunkt auf Datenseite; scanne Datensätze, bis Plz > 9099

Indexed Sequential Access Method (ISAM)
- Von IBM Ende der 1960er Jahre entwickelt


    ```

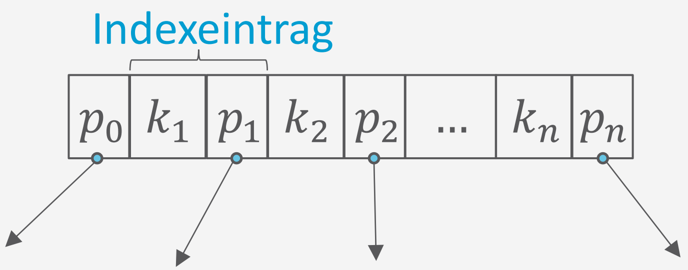
Indexed Sequential Access Method (ISAM)

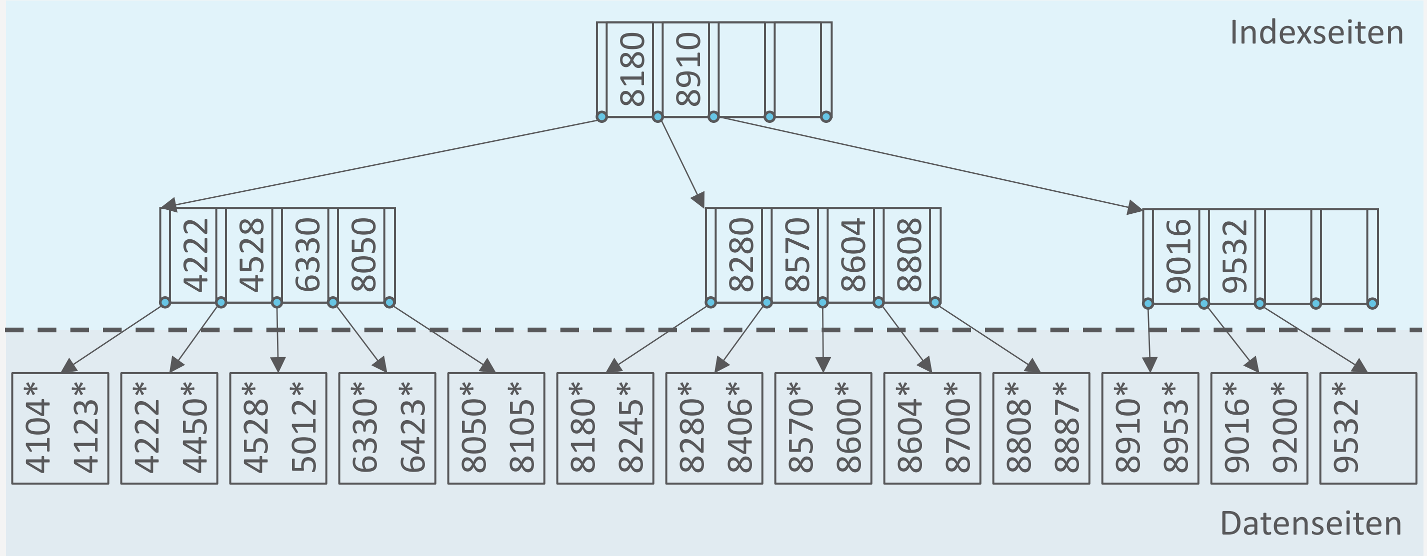
- Datenseiten im Speicher, sortiert gemäß Suchschlüssel (wie für die Binärsuche)
- Indexseiten mit Schlüsselwerten im Baum
  - Hunderte Einträge pro Seite
    - Hohe Verzweigung
    - Kleine Tiefe
  - Felder fester Länge
    - Navigation mittels Versatz
    - Binärsuche möglich



ISAM: Indexeinträge

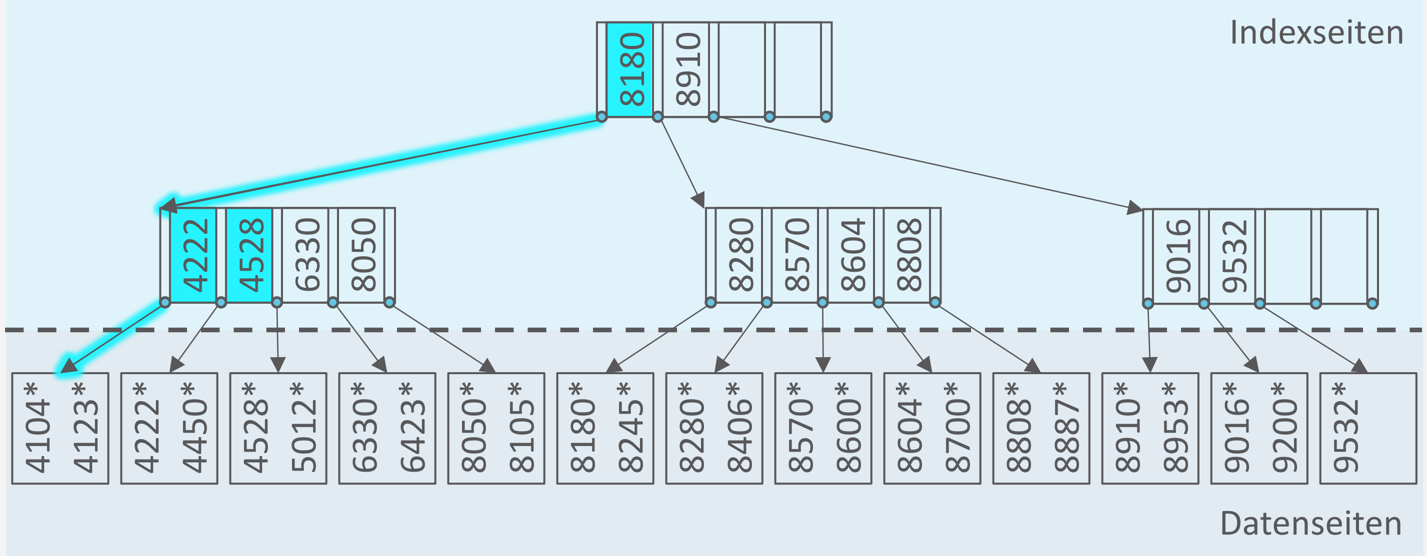
- Indexeintrag $\langle k, p \rangle$
  - Suchschlüssel k
  - Separator p
    - Referenz auf Index- oder Datenseite
    - Schlüsselwerte in Seite referenziert durch p_i alle $\geq k_i$ und $< k_{i+1}$





ISAM: Anfragetypen

- Anfragetypen
  - Bereichsanfragen
    - Beispiel: `Plz between 8800 and 9099`
  - Punktanfragen
    - Beispiel: `Plz=4123`
- Indexierung möglich, solange eine totale Ordnung definiert ist
  - Zahlenbasierte Datentypen
  - Lexikographisch: String, Char
    - $(A \prec B \prec \ldots)$
    - Manche Systeme erlauben keinen Index über sehr lange Strings



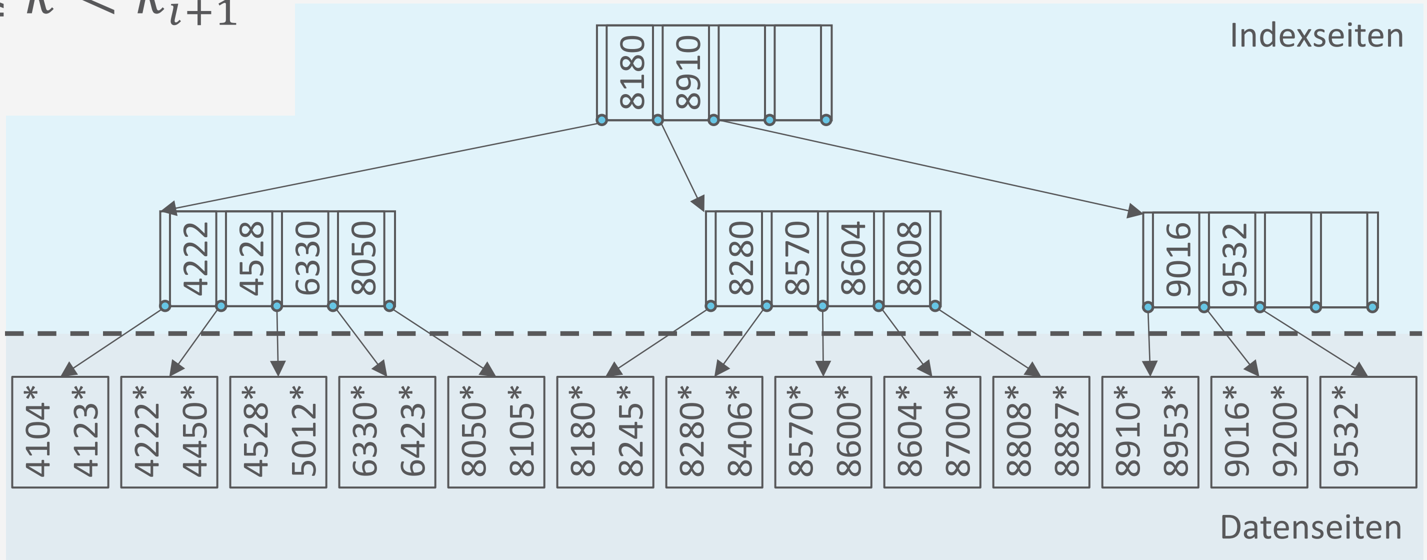
ISAM: Suche

Was haben wir gewonnen?

Was haben wir bezahlt?

- Suche von Einträgen mit Schlüsselwert k in Indexseite n
  - n ist eine Referenz auf eine Indexseite
    - Suche startet bei Wurzelknoten $root$
      - Implementierung: Funktion $\text{search}(k)$ mit Aufruf $\text{search}(k, root)$
  - Suche Separator p_i in n, so dass $k_i \leq k < k_{i+1}$
    - Wenn $k < k_1$, dann $p_i = p_0$
    - Wenn $k_n < k$, dann $p_i = p_n$
    - Mittels Binärsuche umsetzen
  - Wenn p_i Referenz auf Datenseite, suche nach k auf Datenseite (und bei Bereichsanfrage scanne ab da)
  - Sonst: Rekursiver Aufruf der Suche mit Indexseite n', auf die p_i verweist

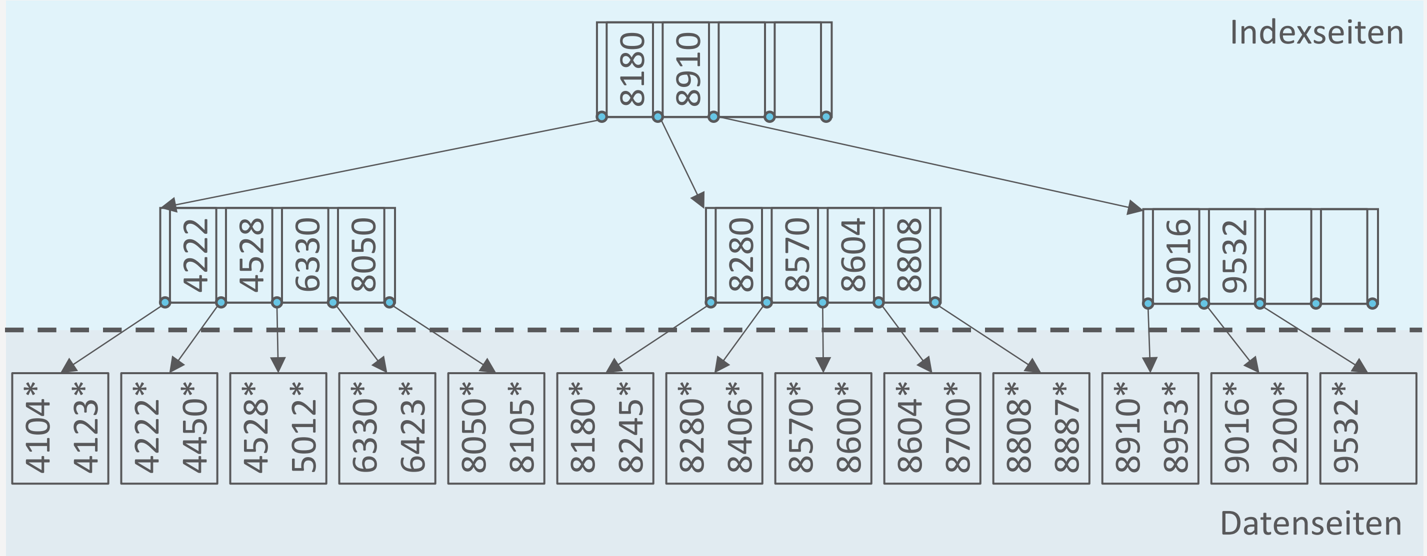
function $\text{search}(k, n)$
　$p \leftarrow \text{binarySearch}^*(k, n)$
　if p refers to data page **then**
　　return $\text{binarySearch}(k, p)$
　$\text{search}(k, p)$　▹ p index page



binarySearch^* setzt eine Binärsuche nach k um, wie in binarySearch, berücksichtigt aber die Separatoren bei der Berechnung der Positionen und gibt den passenden Separator zurück

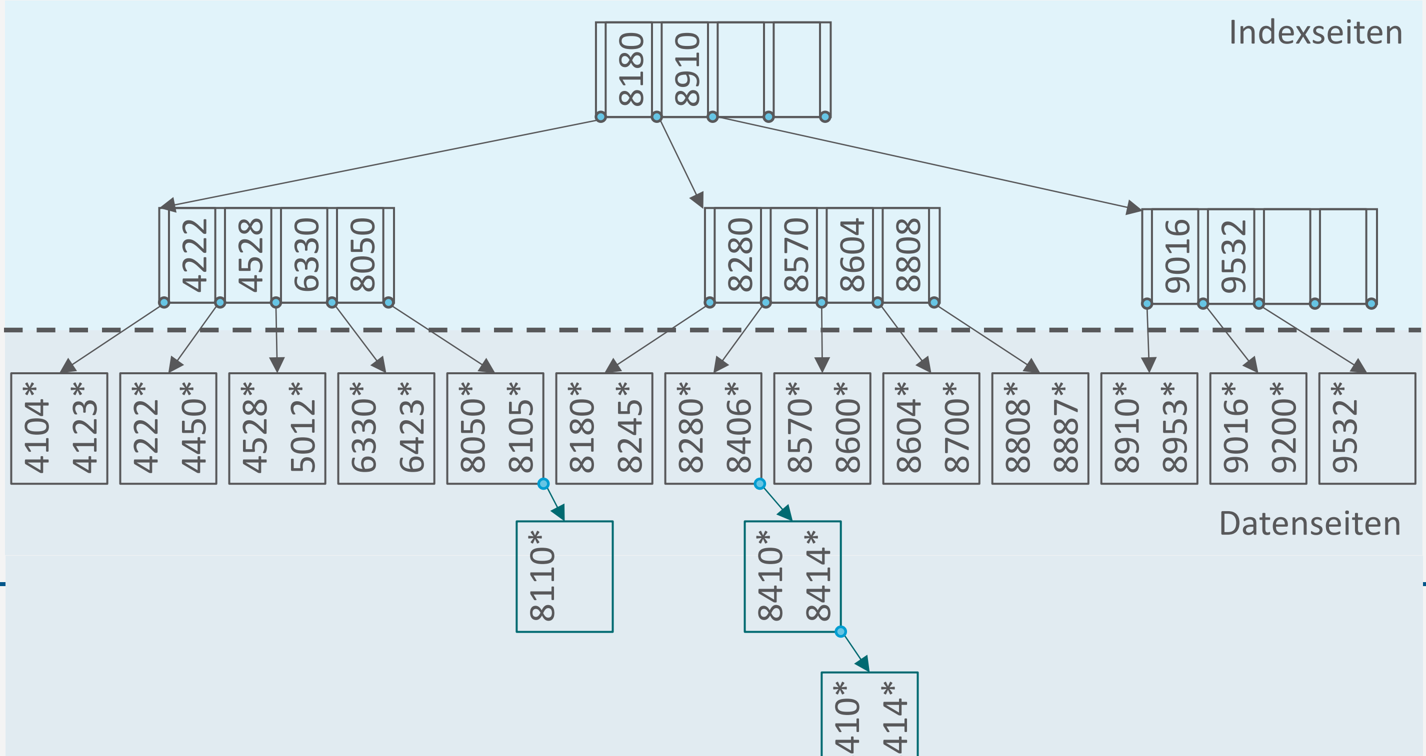
ISAM: Diskussion

- Vorteile
  - Nicht für jeden Zugriff während der Suche eine eigene Seite laden
  - Binärsuche innerhalb einer Indexseite
  - Einstiegspunkt nahe Startpunkt gefunden, ab dem sequentiell gelesen werden kann
- Nachteil
  - Zusätzlicher Speicher für Indexseiten
    - Analyse von Anfragen zum Finden häufiger Suchschlüssel für Indices



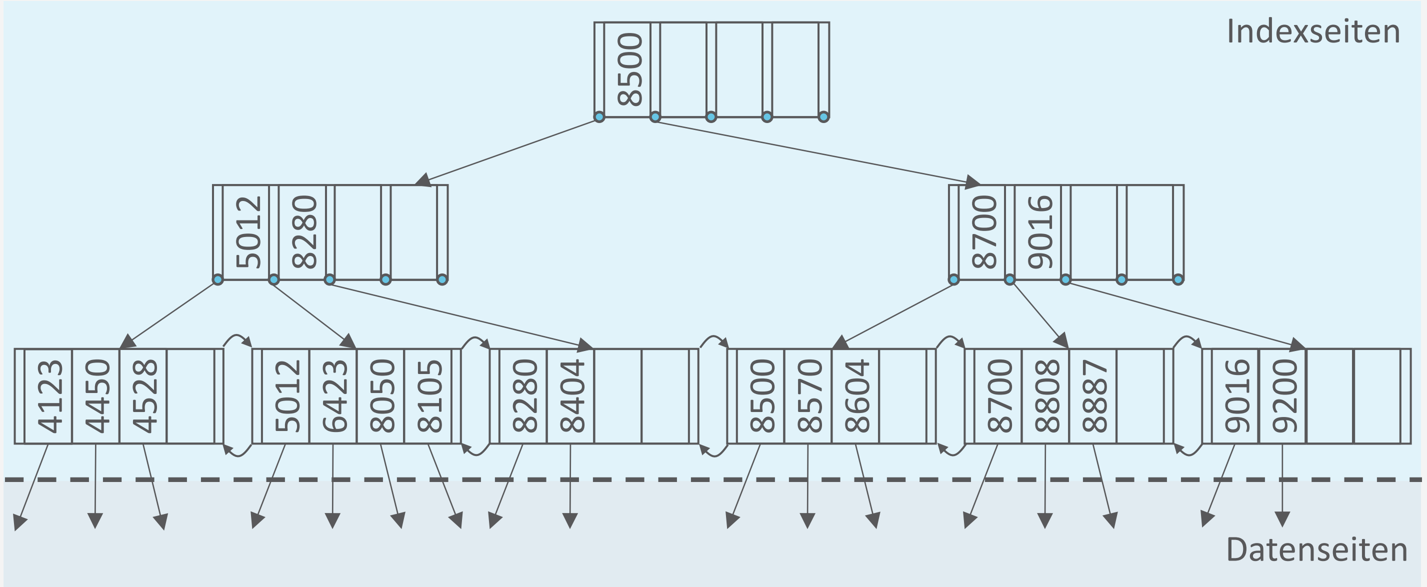
ISAM: Aktualisierungsoperationen

- Löschen: Datensatz über Index suchen und anschließend aus Datenseite entfernen
- Einfügen: Auf passende Datenseite navigieren und Datensatz einfügen
  - Problem, wenn Seite voll
    - Im Vorhinein Platz lassen
    - Überlauf-Seite einfügen
- Vorteil: Index statisch, i.e., bleibt gültig
  - Kein Aufwand bei Änderungen
- Nachteil: Index degeneriert
  - Sequentielle Ordnung durch Überlauf-Seiten zerstört



ISAM: Aktualisierungsoperationen

- 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

Indexierung

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

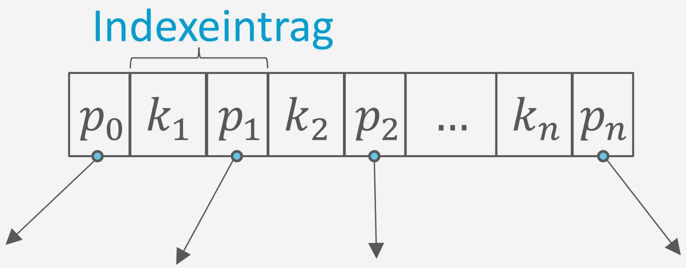
  vs.
  - Indexknotensuche nicht zu linear

R. Bayer and E. M. McCreight, Organization and Maintenance of Large Ordered Indexes, Acta Informatica, vol. 1, no. 3, **1972**

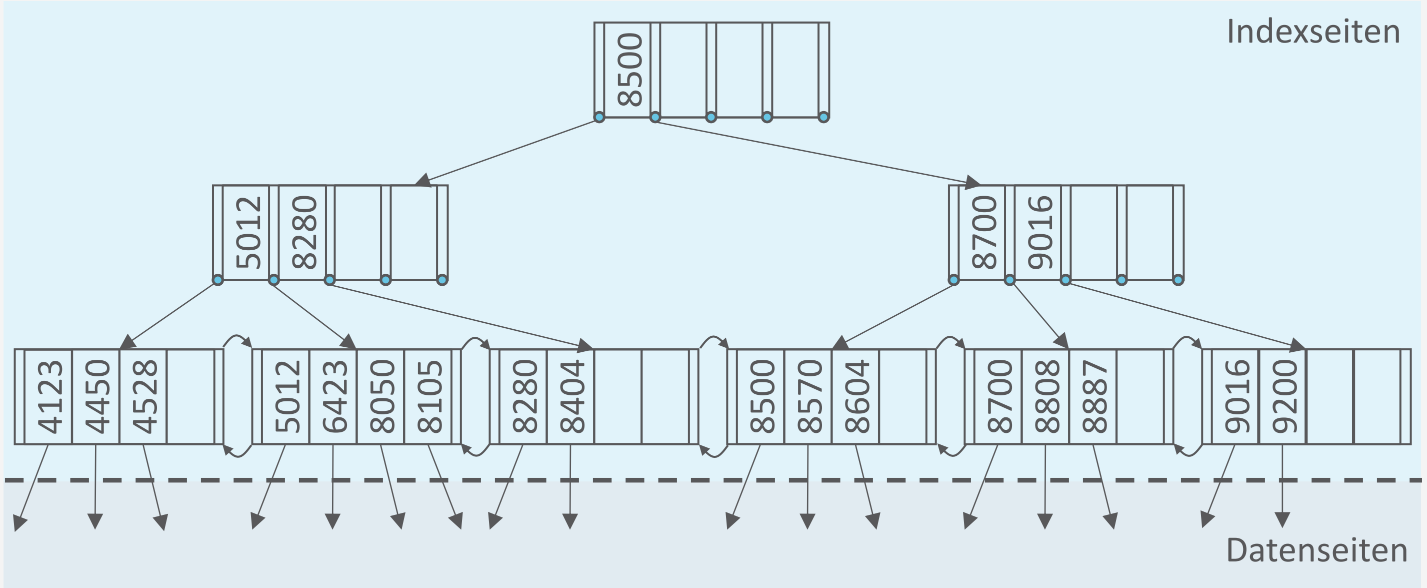
B+-Bäume: Grundlagen

- B+-Bäume ähnlich zu ISAM-Index, wobei
  - Referenzierte Datenseiten i.d.R. nicht in sequentieller Ordnung
  - Index-Blätter zu doppelt verketteter Liste verbunden
    - Schlüssel dort noch einmal wiederholt, sortiert

Warum?



- Jeder Knoten enthält zwischen d und $2d$ Einträge
  - d heißt Ordnung des Baumes, Wurzel ist Ausnahme
- Es gilt weiterhin für die Indexknoten (Nicht-Blätter): Eintrag $\langle k, p \rangle$



B⁺-Bäume: Was wird in den Blättern gespeichert?

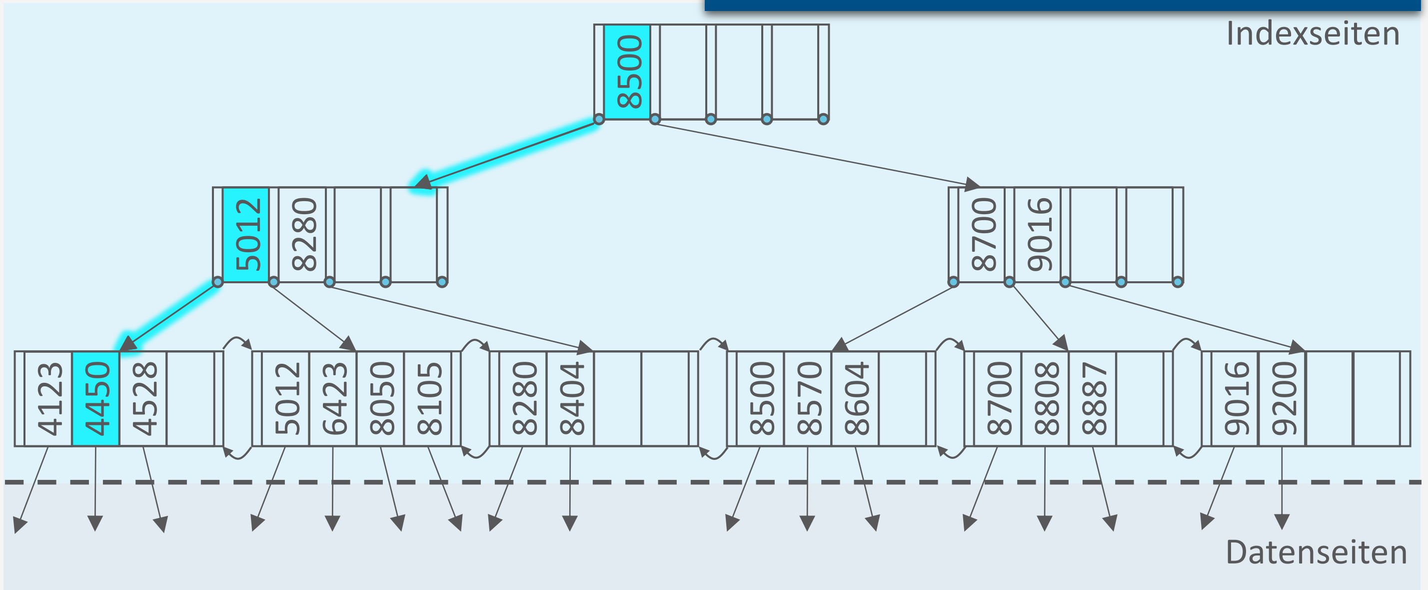
Alternative 2 scheint am meisten verwendet zu werden. Wir nehmen im Folgenden Variante 2 an.

- Drei Alternativen
  1. Vollständiger Datensatz k^*:
     - Blatt ist Datenseite (wie bei ISAM Index)
  2. Ein Paar $\langle k, rid \rangle$, wobei rid (record ID) Zeiger auf Datensatz:
     - Blatt enthält Liste von $\langle k, rid \rangle$ Paaren, sortiert nach k
     - Suchschlüssel k kann auf Blatt häufiger auftreten (mehrere rid‘s mit Suchschlüssel k)
     - Bei $rid = \langle pageno, slotno \rangle$ lässt sich der genaue Speicherort des Datensatzes bestimmen
  3. Ein Paar $\langle k, \{rid_1, rid_2, \dots\} \rangle$, wobei alle rid‘s den Suchschlüssel k haben
     - Suchschlüssel k tritt nur einmal auf (weniger Redundanz)
     - Aber: kein regelmäßiger Abstand von einem $\langle k, \{rid_1, rid_2, \dots\} \rangle$ zum nächsten $\langle k', \{rid'_1, rid'_2, \dots\} \rangle$
- Varianten 2. und 3. bedingen, dass rid‘s stabil sein müssen, also nicht (einfach) verschoben werden können

B⁺-Bäume: Suche

- Suche von Einträgen mit Schlüssel k in B⁺-Baum wie beim ISAM-Index, wobei aber die Abbruchbedingung nicht auf Datenseite prüft, sondern auf Index-Blattseite:
  - Suche in Index nach k bis zum Index-Blattknoten e
  - Suche in e nach k
  - Folge der Referenz
- Beispiel:
  - Punktanfrage: Schlüssel 4450
  - Bereichsanfrage:
    Schlüssel zwischen 4450 und 6500

function $\text{search}(k, n)$
  $p \leftarrow \text{binarySearch}^*(k, n)$
  if p refers to index leaf **then**
    return $\text{binarySearch}(k, p)$
  $\text{search}(k, p)$



binarySearch* setzt eine Binärsuche nach k um, wie in binarySearch, berücksichtigt aber die Separatoren bei der Berechnung der Positionen und gibt den passenden Separator zurück

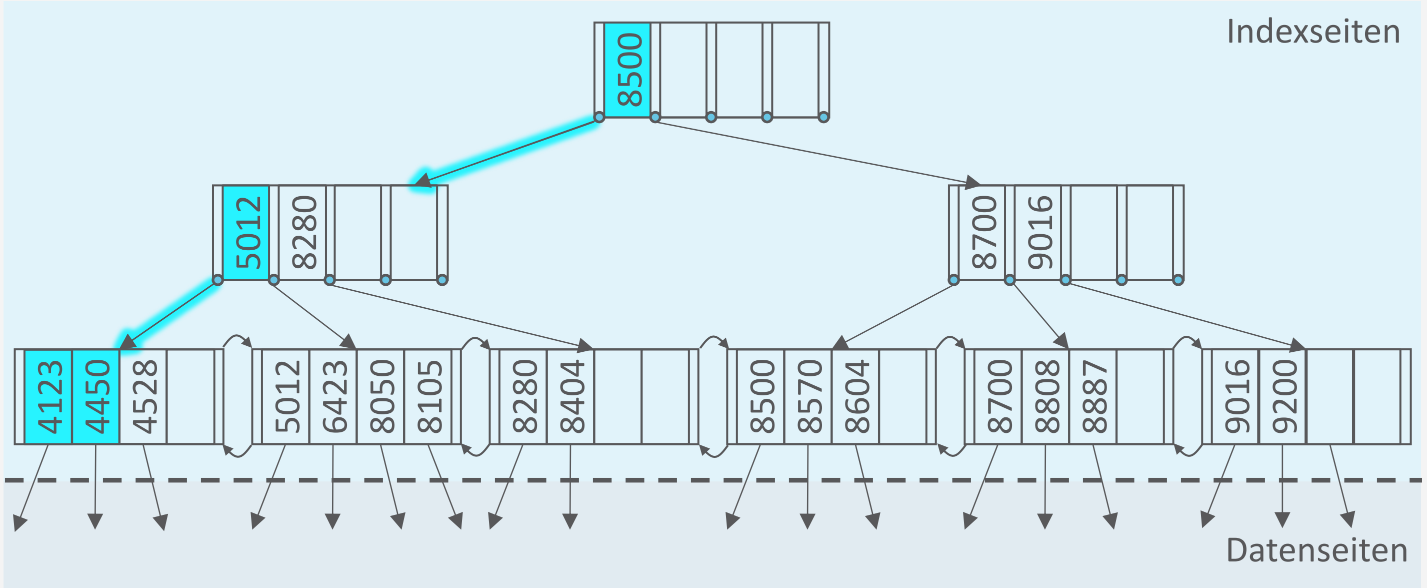
B+-Bäume: Einfügen

- B+-Baum soll nach Einfügung balanciert bleiben, i.e., keine Überlauf-Seiten
- Überblick über Vorgehen zum Einfügen mit Eingaben k und rid ($\text{insert}(k, rid)$):
  1. Suche Blattseite n, in der Eintrag für k sein kann
  2. Falls n genug Platz hat (höchstens $2d - 1$ Einträge): Füge Eintrag $\langle k, rid \rangle$ in n ein
     - Suche nach passender Position mittels Binärsuche
  3. Sonst: Teile n auf in n und n' inklusive $\langle k, rid \rangle$ und füge neuen Eintrag $\langle k', n' \rangle$ in Elternknoten e von n ein, wobei n' als Separator für die Referenz auf n' steht und k' der kleinste Schlüssel in n' ist
     - Wenn e keinen Platz mehr hat ($2d$ Einträge), 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

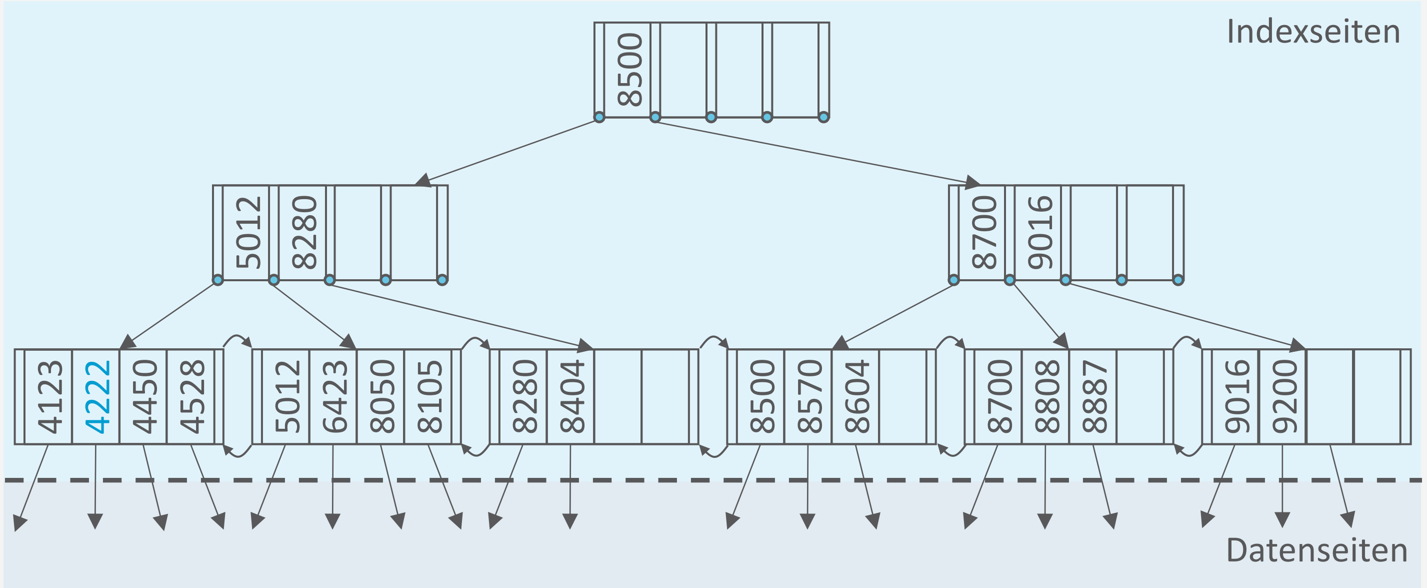
B⁺-Bäume: Einfügen – Beispiel ohne Aufspaltung

- Einfügen eines Eintrags mit Schlüssel 4222
  1. Suche führt zu erstem Blattknoten in der verketteten Liste $(3 < 2d - 1, d = 2)$
  2. Blattknoten hat genug Platz $(3 < 2d - 1, d = 2)$, von daher einfach einfügen
     - Erhalte Sortierung innerhalb der Knoten

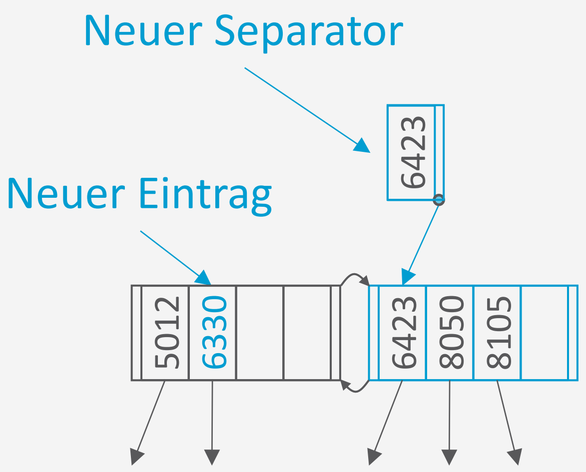


B+-Bäume: Einfügen – Beispiel ohne Aufspaltung

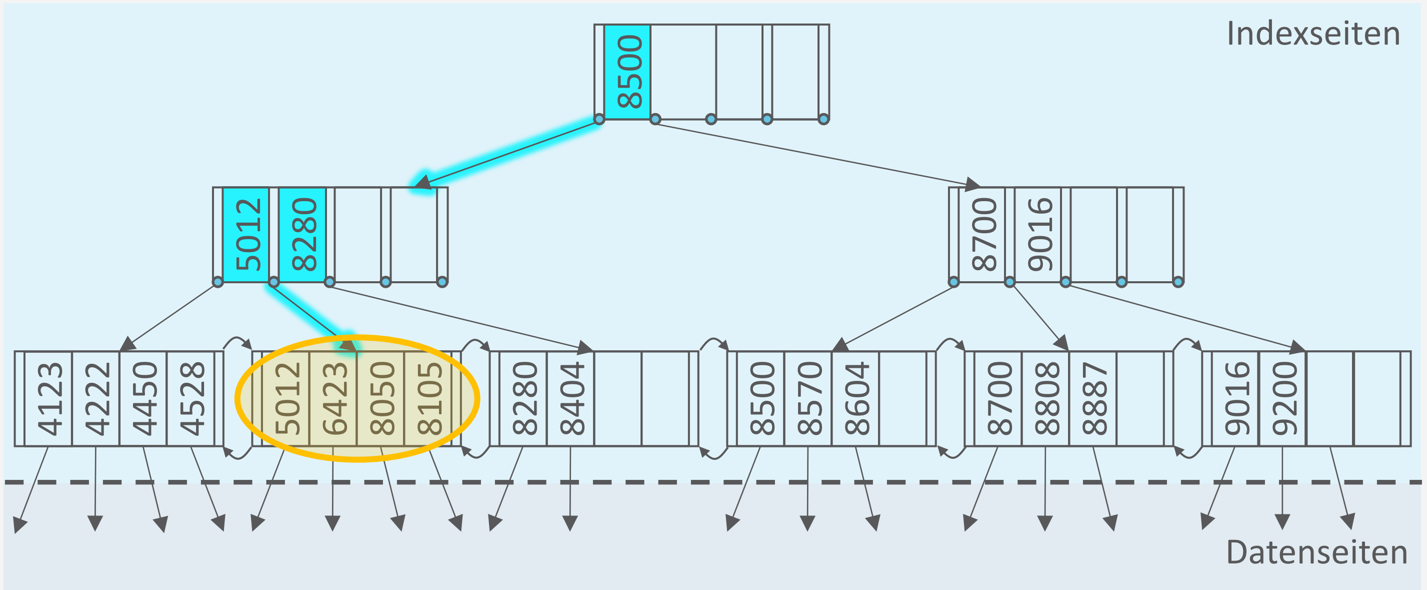
- Einfügen eines Eintrags mit Schlüssel 4222
  1. Suche führt zu erstem Blattknoten in der verketteten Liste $(3 < 2d - 1, d = 2)$
  2. Blattknoten hat genug Platz $(3 < 2d - 1, d = 2)$, von daher einfach einfügen
     - Erhalte Sortierung innerhalb der Knoten



B⁺-Bäume: Einfügen – Beispiel mit Aufspaltung

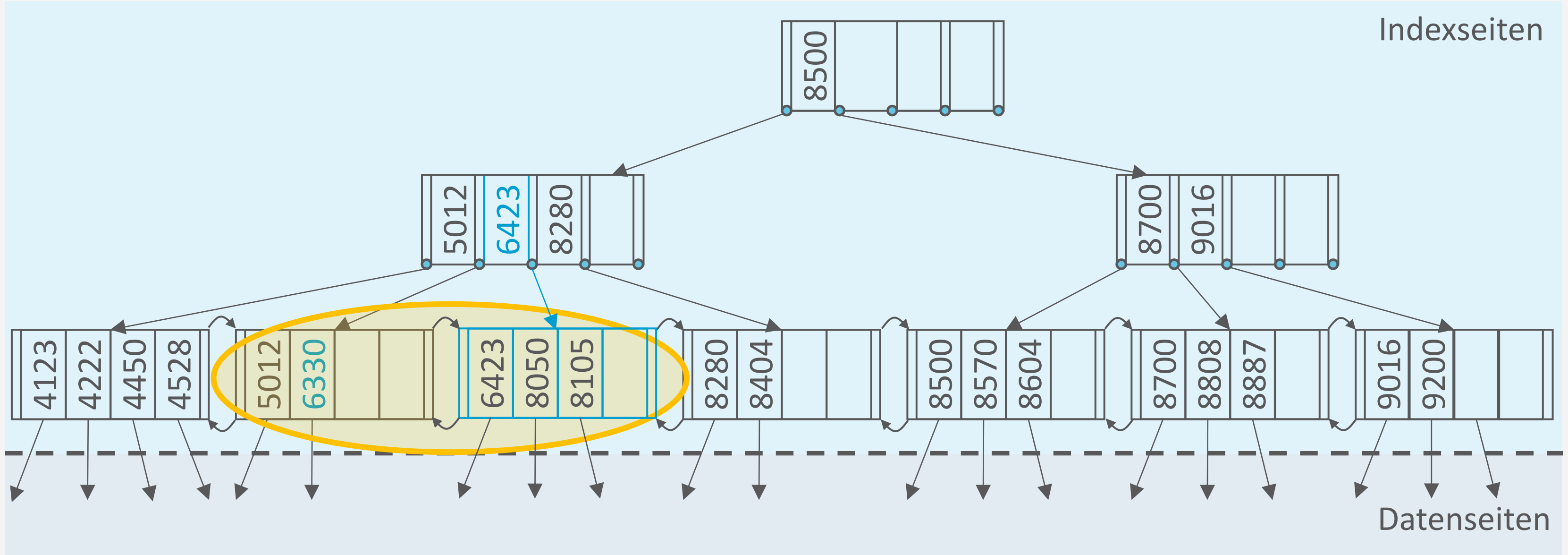


- Einfügen eines Eintrags mit Schlüssel 6330
  1. Suche führt zu zweitem Blattknoten in verketteter Liste
  2. Blattknoten hat nicht mehr genug Platz
  3. Index anpassen
     - Erste Hälfte (d Einträge) bleibt in Knoten
     - Zweite Hälfte ($d + 1$ Einträg) geht in neuen Knoten
     - Neuer Eintrag in Elternknoten
       - Kleinster Eintrag im neuen Knoten: 6423
       - Platz in Elternknoten, daher direkt einfügen



B⁺-Bäume: Einfügen – Beispiel mit Aufspaltung

- Einfügen eines Eintrags mit Schlüssel 6330
  1. Suche führt zu zweitem Blattknoten in verketteter Liste
  2. Blattknoten hat nicht mehr genug Platz
  3. Index anpassen
     - Angepasster Index:



B⁺-Bäume: Insert

- $\text{insert}(k, rid)$ wird von außen aufgerufen
- Blattknoten enthalten rids, innere Knoten enthalten Zeiger auf andere B⁺-Baum-Knoten

function $\text{insert}(k, rid)$
    $\langle key, ptr \rangle \leftarrow \text{treeInsert}(k, rid, root)$ ▹ root contains the root of the index tree
    if key is not **null then**
        Allocate new root page r
        Populate r with $p_0 \leftarrow root, k_1 \leftarrow key, p_1 \leftarrow ptr$
        $root \leftarrow r$

B+-Bäume: Tree-Insert für Insert

- Einträge pro Seite maximal: $2d = n$

function treeInsert$(k, rid, node)$
    if $node$ is leaf **then**
        return leafInsert$(k, rid, node)$
    if $k < k_0$ **then**
        $i \leftarrow 0$
    else if $k_n < k$ **then**
        $i \leftarrow n$
    else
        Find i such that $k_i \leq k < k_{i+1}$ ▹ Use binary search
    $\langle sep, ptr \rangle \leftarrow$ treeInsert(k, rid, p_i) ▹ p_i contains the reference to next node
    if sep is **null then**
        return $\langle null, null \rangle$
    else
        return split$(sep, ptr, node)$

B+-Bäume: Leaf-Insert für Insert

- Einträge pro Seite maximal: $2d$
- Einträge auf Seite: $\{\langle k_1, rid_1\rangle, \dots, \langle k_{2d}, rid_{2d}\rangle\}$

function $\mathrm{leafInsert}(k, rid, leaf)$
    if another entry fits into $leaf$ **then**
        Insert $\langle k, rid\rangle$ into $leaf$
        return $\langle null, null\rangle$
    Allocate new leaf node l
    Let $\{\langle k'_1, rid'_1\rangle, \dots, \langle k'_{2d+1}, rid'_{2d+1}\rangle\} \leftarrow \{\langle k_1, rid_1\rangle, \dots, \langle k_{2d}, rid_{2d}\rangle\} \cup \{\langle k, rid\rangle\}$
    Store entries $\langle k'_1, rid'_1\rangle, \dots, \langle k'_d, rid'_d\rangle$ in $leaf$
    Store entries $\langle k'_{d+1}, rid'_{d+1}\rangle, \dots, \langle k'_{2d+1}, rid'_{2d+1}\rangle$ in l
    return $\langle k'_{d+1}, l\rangle$

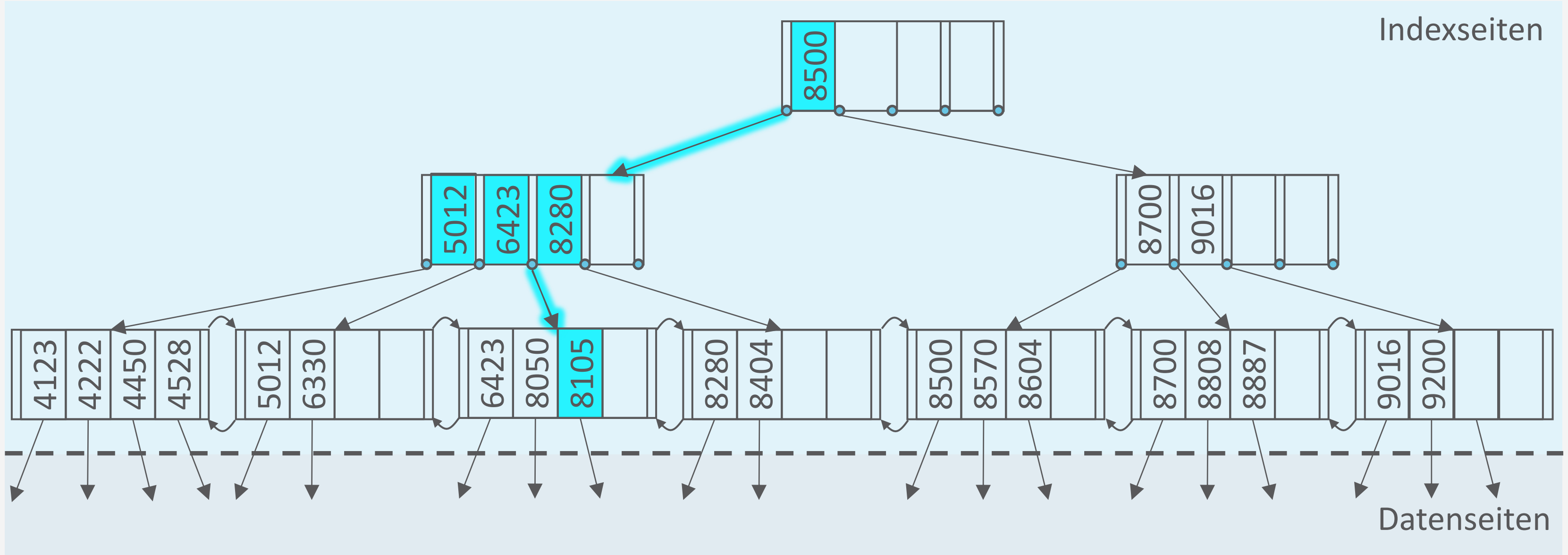
B⁺-Bäume: Split für Insert

- Einträge pro Seite maximal: $2d$
- Einträge auf Seite: $\{\langle k_1, p_1\rangle, \dots, \langle k_{2d}, p_{2d}\rangle\}$, dazu noch p_0

function $\text{split}(k, ptr, node)$
    if another entry fits into $node$ **then**
        Insert $\langle k, ptr\rangle$ into $node$
        return $\langle null, null\rangle$
    Allocate new node p
    Let $\{\langle k'_1, p'_1\rangle, \dots, \langle k'_{2d+1}, p'_{2d+1}\rangle\} \leftarrow \{\langle k_1, p_1\rangle, \dots, \langle k_{2d}, p_{2d}\rangle\} \cup \{\langle k, ptr\rangle\}$
    Store entries $\langle k'_1, p'_1\rangle, \dots, \langle k'_d, p'_d\rangle$ in $node$, keeping p_0
    Store entries $\langle k'_{d+2}, p'_{d+2}\rangle, \dots, \langle k'_{2d+1}, p'_{2d+1}\rangle$ in p
    $p_0 \leftarrow p'_{d+1}$ in p
    return $\langle k'_{d+1}, p\rangle$

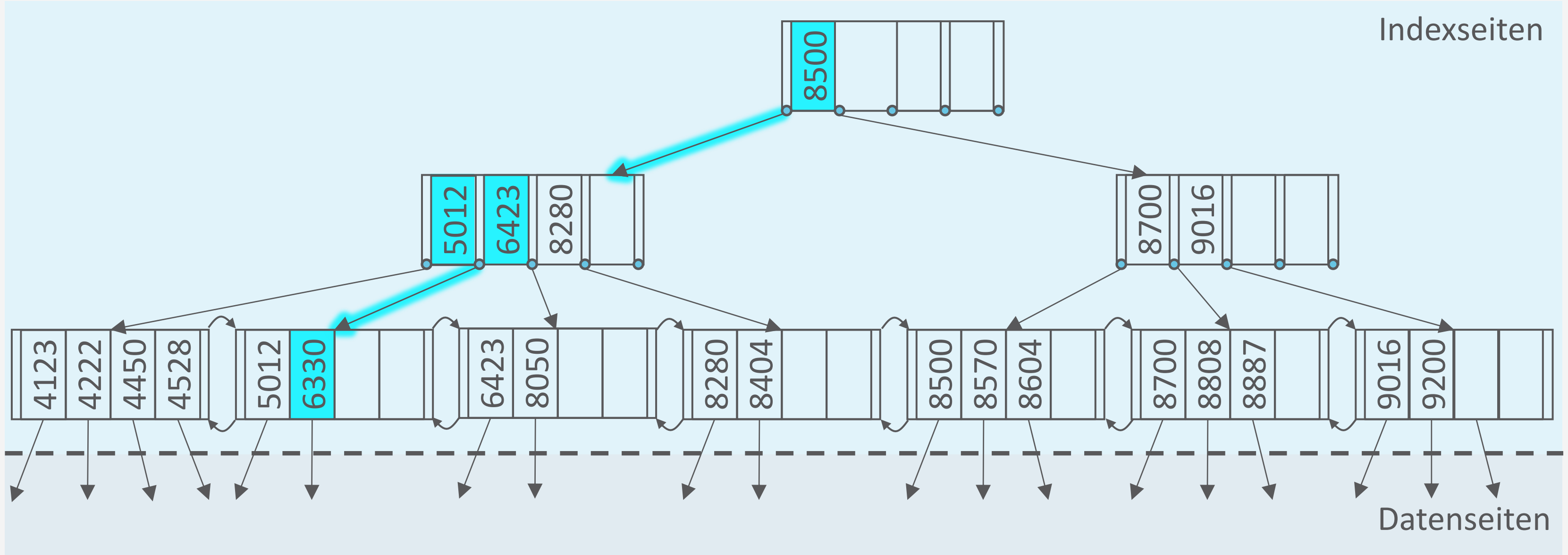
B+-Baum: Löschen

- B+-Baum soll nach Löschung balanciert bleiben
  - Wenn Löschen nicht für einen Unterlauf sorgt (min. $d + 1$ Einträge), einfach löschen
    - Innere Knoten können dann alte Schlüssel enthalten → ist ok, da die Ordnung erhalten bleibt
  - Sonst, d.h., wenn nach löschen nur $d - 1$ Einträge übrig bleiben: Index wieder balancieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8105



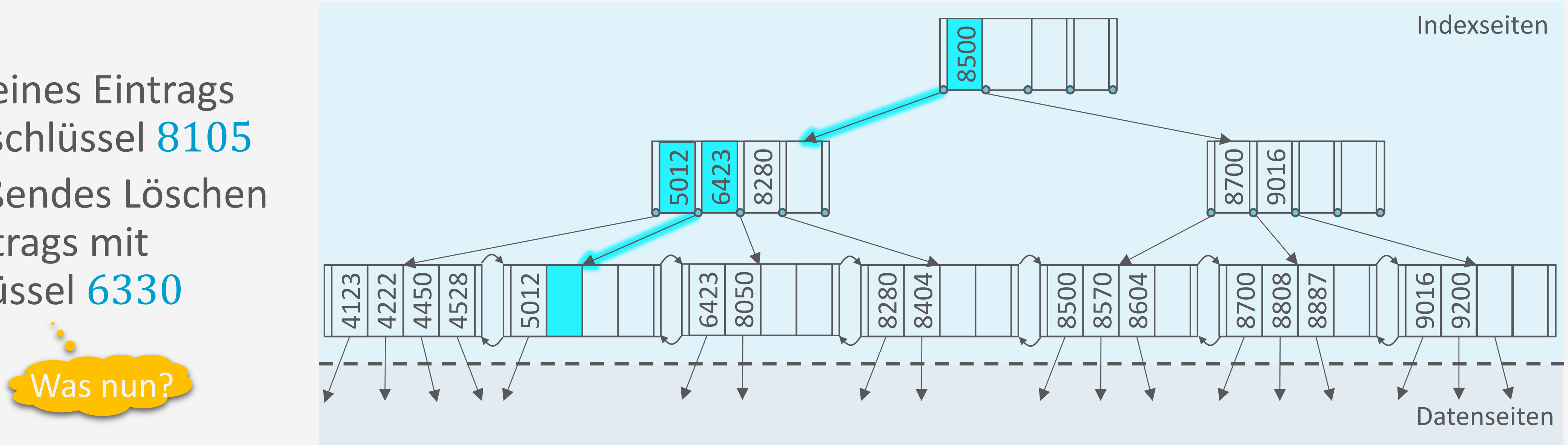
B⁺-Baum: Löschen

- B⁺-Baum soll nach Löschung balanciert bleiben
  - Wenn Löschen nicht für einen Unterlauf sorgt (min. $d + 1$ Einträge), einfach löschen
    - Innere Knoten können dann alte Schlüssel enthalten → ist ok, da die Ordnung erhalten bleibt
  - Sonst, d.h., wenn nach löschen nur $d - 1$ Einträge übrig bleiben: Index wieder balancieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8105
  - Anschließendes Löschen eines Eintrags mit Suchschlüssel 6330



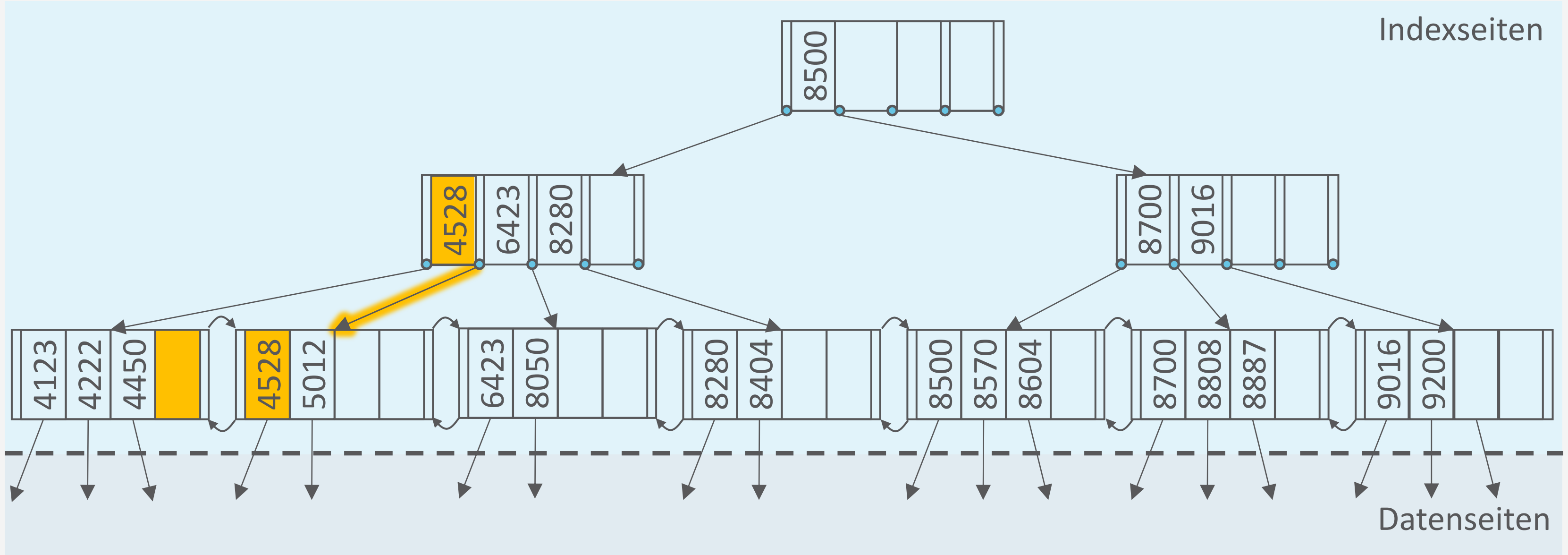
B⁺-Baum: Löschen

- B⁺-Baum soll nach Löschung balanciert bleiben
  - Wenn Löschen nicht für einen Unterlauf sorgt (min. $d+1$ Einträge), einfach löschen
    - Innere Knoten können dann alte Schlüssel enthalten → ist ok, da die Ordnung erhalten bleibt
  - Sonst, d.h., wenn nach löschen nur $d-1$ Einträge übrig bleiben: Index wieder balancieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8105
  - Anschließendes Löschen eines Eintrags mit Suchschlüssel 6330



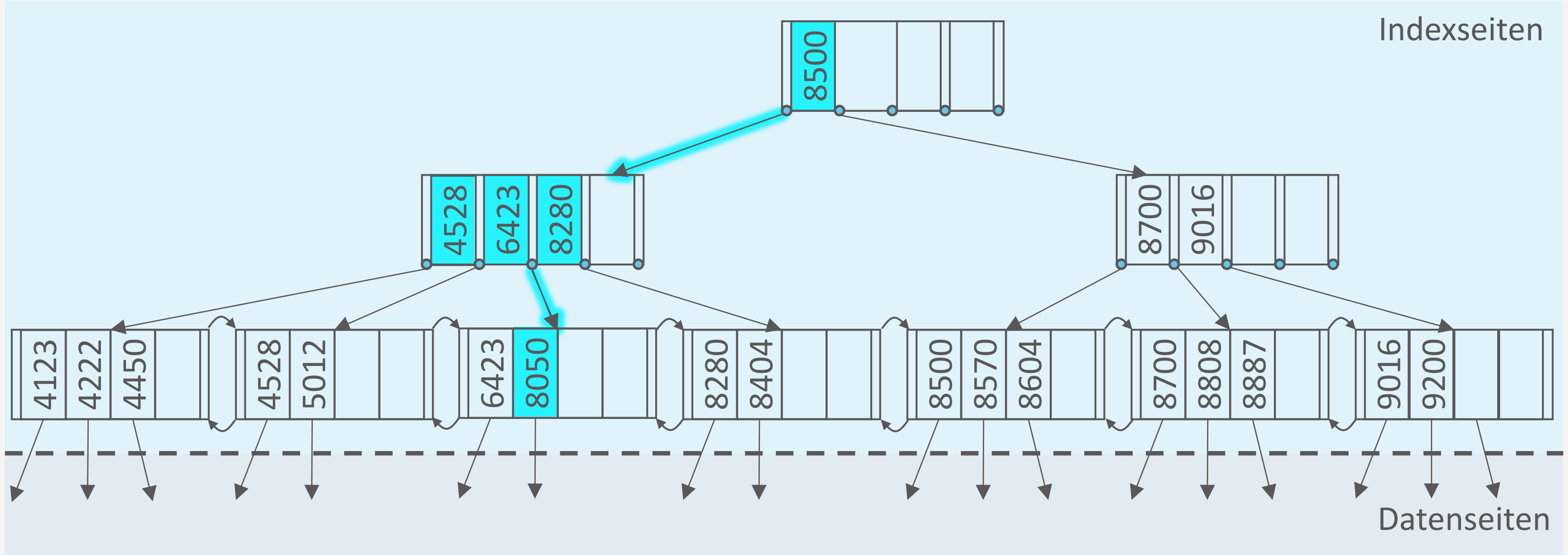
B+-Baum: Löschen – Unterlauf

- Vorgehen, wenn eines der Nachbarblätter genug Einträge ($> d$) hat
  - Blatt mit Eintrag aus Nachbarknoten auffüllen und Indexeintrag in Vorfahrknoten aktualisieren
    - Wenn Nachbarblatt gleichen Elternknoten wie unterlaufener Knoten hat, dann Elternknoten anpassen, sonst Großelternknoten anpassen
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 6330 führt zu Unterlauf, welcher mit Nachbar-Eintrag aufgefüllt werden kann



B⁺-Baum: Löschen – Unterlauf

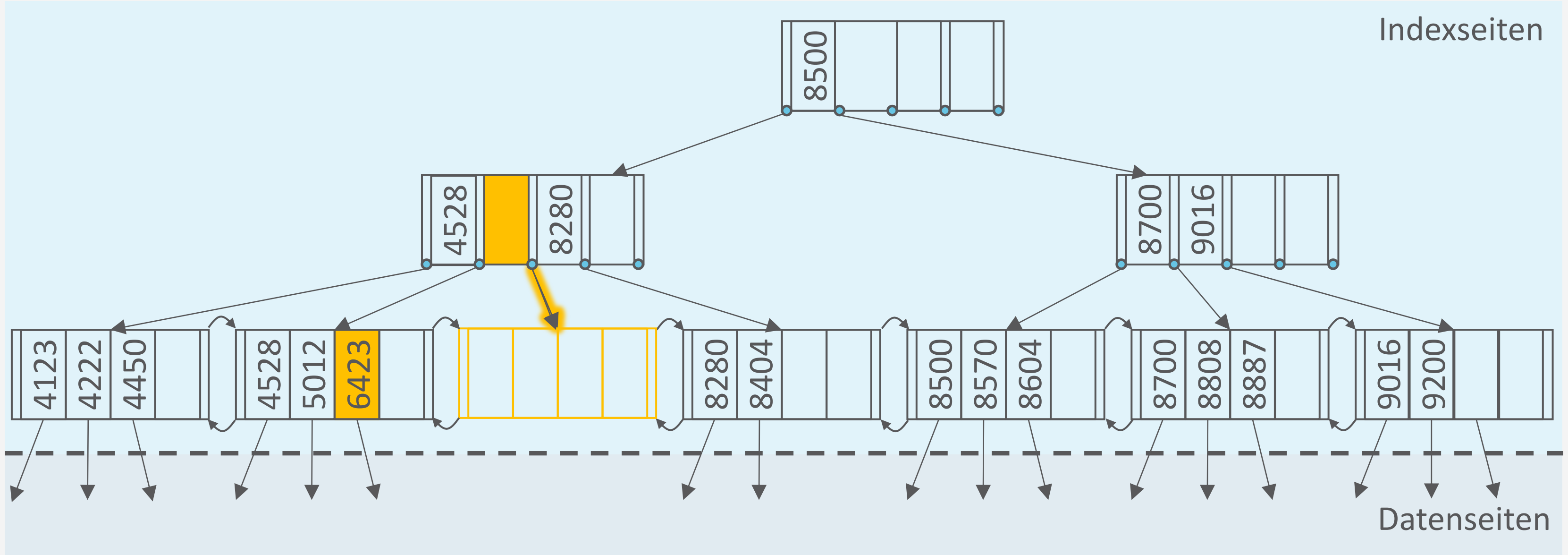
- Vorgehen, wenn eines der Nachbarblätter genug Einträge ($> d$) hat
  - Blatt mit Eintrag aus Nachbarknoten auffüllen und Indexeintrag in Vorfahrknoten aktualisieren
    - Wenn Nachbarblatt gleichen Elternknoten wie unterlaufener Knoten hat, dann Elternknoten anpassen, sonst Großelternknoten anpassen
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 6330 führt zu Unterlauf, welcher mit Nachbar-Eintrag aufgefüllt werden kann
  - Anschließendes Löschen eines Eintrags mit Suchschlüssel 8050



Und nun?

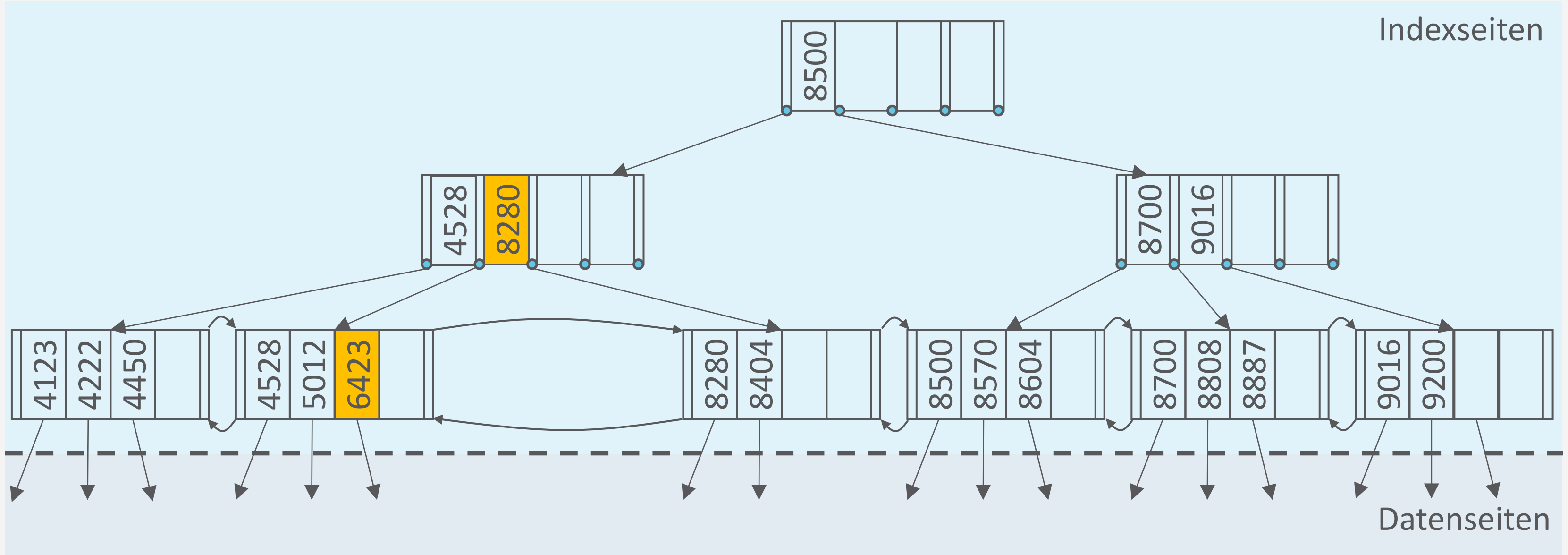
B+-Baum: Löschen – Unterlauf

- Vorgehen, wenn kein Nachbarblatt genug Einträge hat (beide nur d Einträge)
  - Blattseite mit einem der Nachbarblätter verschmelzen und verweisenden Indexeintrag in Elternknoten löschen
    - Verschmelzen bei unterschiedlichen Elternknoten: Zusätzlich Großeltern-Indexeintrag aktualisieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8050 führt zu Unterlauf
    - Einträge in vorheriges Blatt überführen
    - Dritte Blattseite und Indexeintrag $\langle 6423, leaf_3 \rangle$ löschen



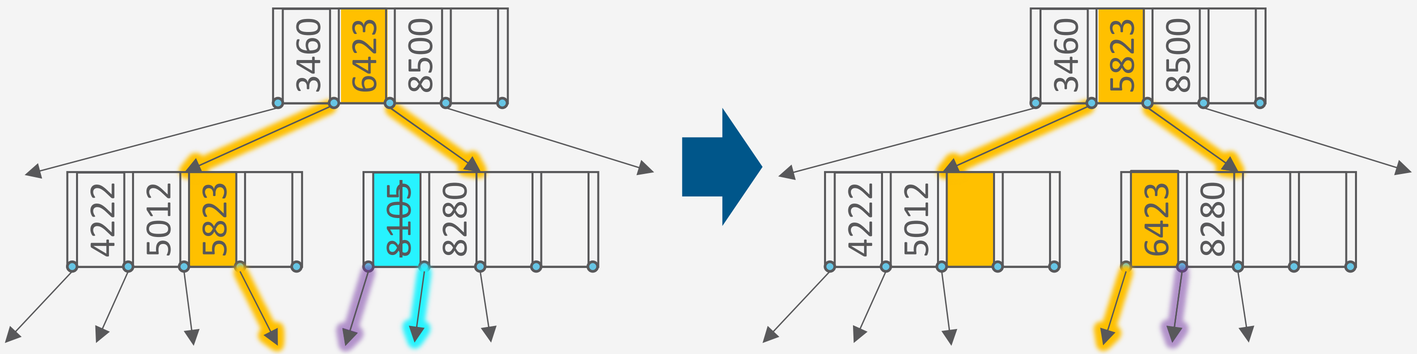
B+-Baum: Löschen – Unterlauf

- Vorgehen, wenn kein Nachbarblatt genug Einträge hat (beide nur d Einträge)
  - Blattseite mit einem der Nachbarblätter verschmelzen und verweisenden Indexeintrag in Elternknoten löschen
    - Verschmelzen bei unterschiedlichen Elternknoten: Zusätzlich Großeltern-Indexeintrag aktualisieren
- Beispiel:
  - Ergebnis
- Verbleibende Aufgabe: Was machen, wenn durch Löschen des Indexeintrags im Elternknoten der Elternknoten unterläuft?



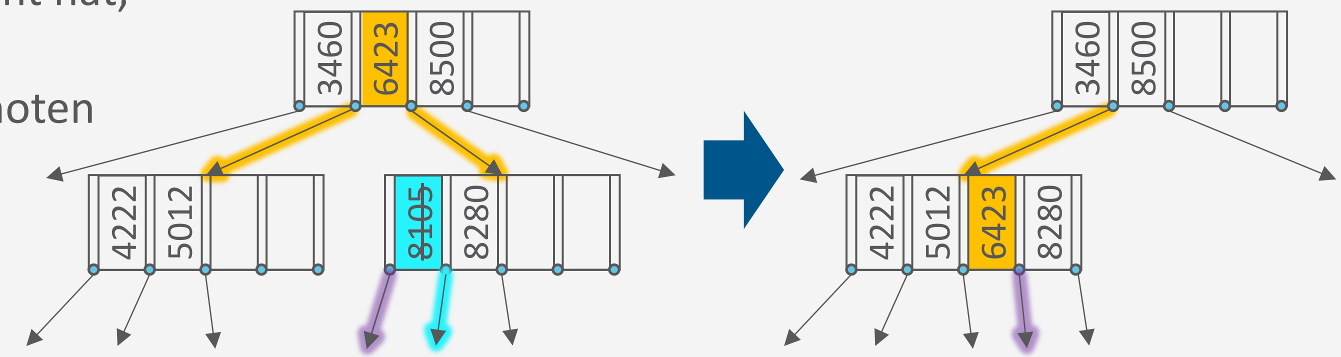
B⁺-Baum: Löschen – Indexeintrag

- Löschen von einem Indexeintrag $\langle k, p \rangle$ in Knoten n (ähnlich zu vorher):
  1. Wenn n genügend Einträge ($> d$) hat, dann $\langle k, p \rangle$ einfach löschen
  2. Sonst: (Umverteilen oder verschmelzen)
     a. Wenn Nachbarknoten genügend Einträge ($> d$) hat, umverteilen:
        - Rotiere Eintrag $\langle k', p' \rangle$ aus Nachbarknoten über den Elternknoten in den unterlaufenen Knoten
        i. Suchschlüssel k' ersetzt k'' im Elternknoten
        ii. Eintrag $\langle k'', p' \rangle$ wird im unterlaufenen Knoten eingefügt
     - Beispiel
        - Indexeintrag $\langle 8105, p \rangle$ löschen
        - Rotieren des Schlüssels 5823 in den Elternknoten mit Weiterrotation des Eintrags 6423 im Elternknoten in den unterlaufenen Kindknoten



B+-Baum: Löschen – Indexeintrag

- Löschen von einem Indexeintrag mit Schlüssel k in Knoten n ähnlich zu vorher):
  1. Wenn n genügend Einträge ($> d$) hat, dann $\langle k, p\rangle$ 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 n und Nachbarknoten n'
        - Ziehe Schlüssel k', der n und n' getrennt hat, in den verschmolzenen Knoten
        - Lösche Indexeintrag $\langle k', p'\rangle$ in Elternknoten (rekursiver Aufruf)



B+-Baum: Delete

Bei der Umsetzung zu beachten:
- Wo liegt der Nachbarknoten gemäß der Ordnung?
- Hat der Nachbarknoten den gleichen Elternknoten?

Versuchen Sie sich gern selbst mal am Pseudocode dafür.

- Eintrag $\langle k, rid \rangle$ löschen
  1. Finde Blattseite l, in der Eintrag für $\langle k, rid \rangle$ steht
  2. Falls l genügend gefüllt (min. $d + 1$ Einträge), Eintrag löschen
  3. Sonst (d Einträge):
     a. Wenn Nachbarblatt n genügend Einträge ($> d$) hat: Mit Eintrag aus n auffüllen und Indexeintrag im Vorfahren aktualisieren
     b. Sonst (Nachbarblätter haben auch nur d Einträge): Nachbarblatt n und l verschmelzen und Indexeintrag in Vorfahren löschen

- Indexeintrag $\langle k, p \rangle$ in Knoten n löschen
  1. Wenn n genügend Einträge ($> d$) hat, dann $\langle k, p \rangle$ einfach löschen
  2. Sonst: (Umverteilen oder verschmelzen)
     a. Wenn Nachbarknoten genügend Einträge ($> d$) hat, umverteilen:
        - Rotiere Eintrag über den Elternknoten
        - Eventuell Großelternknoten aktualisieren
     b. Sonst (d Einträge der Nachbarknoten):
        - Verschmelze Knoten n und Nachbarknoten n'
        - Ziehe Schlüssel k', der n und n' getrennt hat, in den verschmolzenen Knoten
        - Lösche Indexeintrag $\langle k', p' \rangle$ in Elternknoten (rekursiver Aufruf)

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

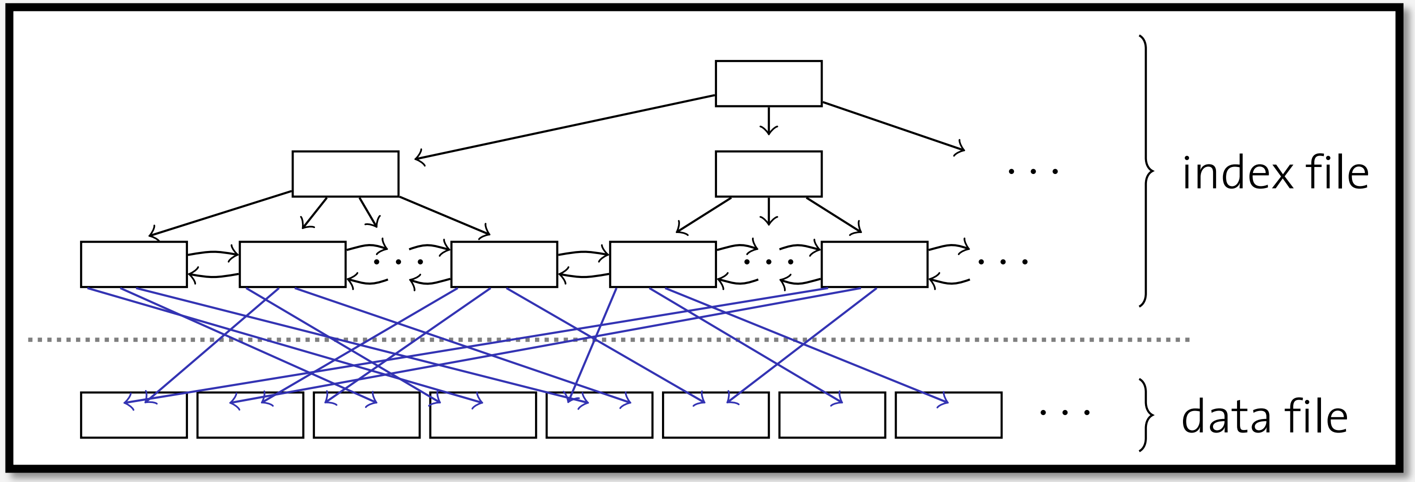
Erzeugung von Indexstrukturen in SQL

- Implizite Indexe
  - Indexe automatisch erzeugt für Primärschlüssel und Unique-Integritätsbedingungen
    - Werden z.B. bei Einfügeoperationen genutzt für effiziente Prüfung, ob Wert schon existiert
      - Kein Zugriff auf Datensatz selbst nötig
    - Schnelle Findung von Einträgen anhand des Primärschlüssels
- Explizite Indexe: CREATE INDEX $Name$ ON $R(A_1, \ldots, A_n)$
  - Wissen über häufige Anfragen nutzen
  - Einfache Indexe über ein Attribut, zusammengesetzte Indexe (Verbundindexe) über mehrere Attribute einer Tabelle
  - Beispiel
    - Bei häufigen Anfragen mit Selektion der PLZ
    - `create index PlzIndex on Kunden(Plz)`

```
create index IndexName
on TableName(Attr);
```

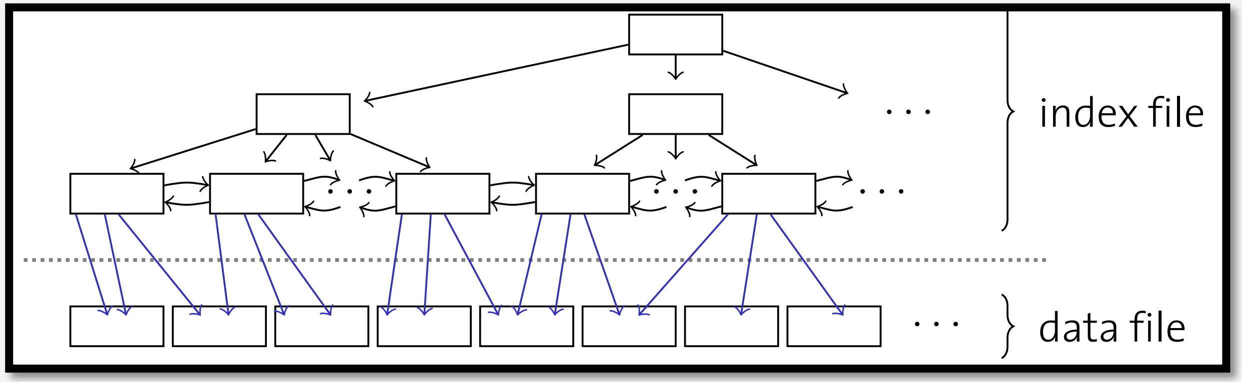
```
create index IndexName
on TableName(Attr_1, Attr_2, ..., Attr_n);
```

Indexe mit zusammengesetzten Schlüsseln

- Voraussetzung: Dinge haben eine definierte **totale Ordnung**
  - Integer, Zeichenketten, Datumsangaben, ...
    - In einigen Implementierungen können lange Zeichenketten nicht als Index verwendet werden
  - Auch Hintereinandersetzung möglich
    - Z.B. basierend auf einer lexikographischen Ordnung
- Beispiel:
  - Bei häufigen Anfragen nach Kundennach- und –vornamen (in der Reihenfolge)
  - ```
    create index idx_nname_vname
    on Kunden(Nachname,Vorname)
    ```
    ```

B+-Bäume und Sortierung

- Eine typische Situation mit $\langle k, rid \rangle$ 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



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

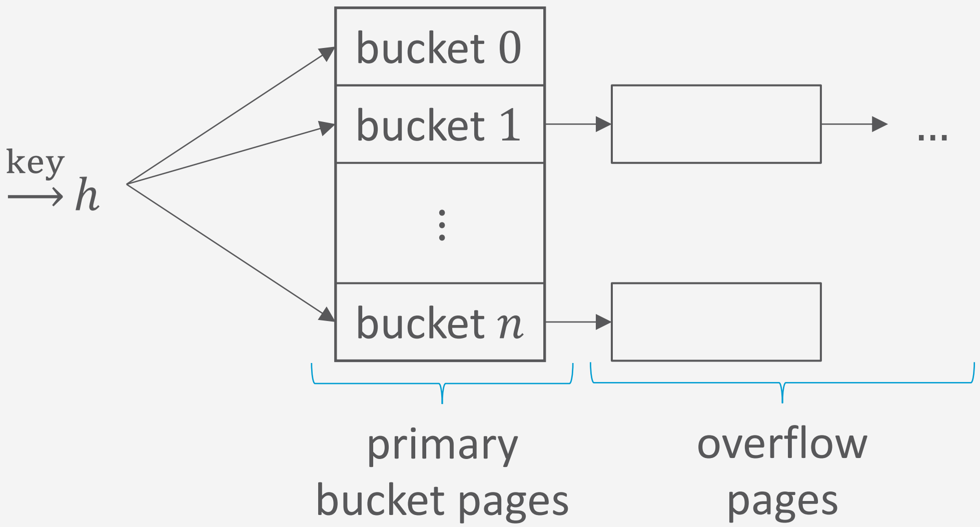
Warum macht man Indexe nicht immer geclustert?

Hash-basierte Indexierung

Effiziente Indexierung bei *Gleichheitsprädikaten*

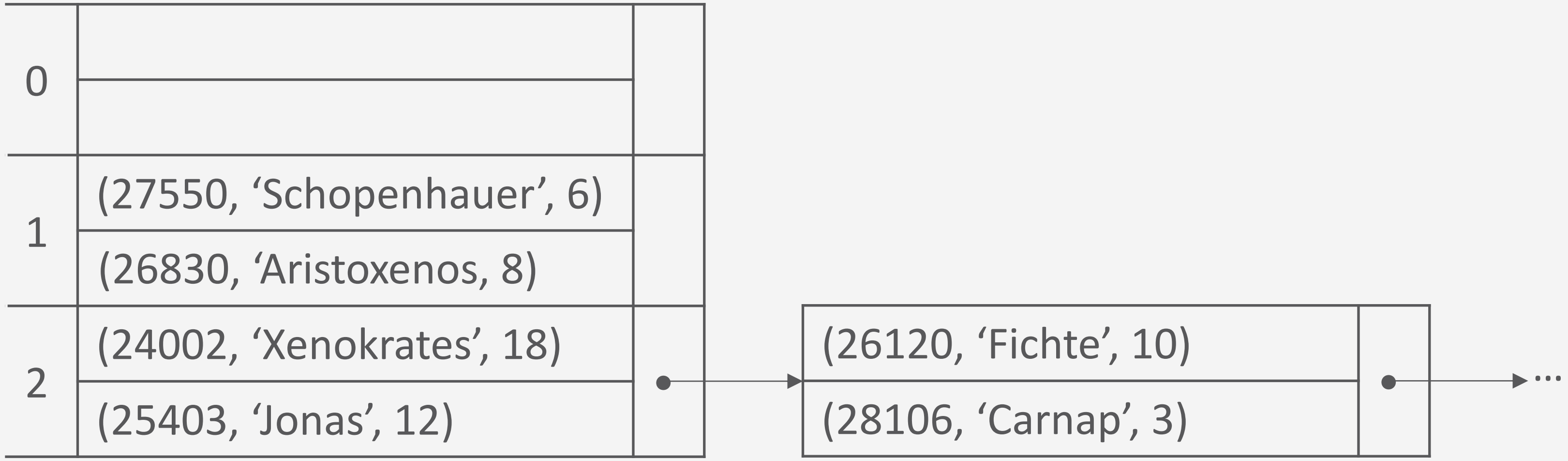
Hash-basierte Indexierung

- B⁺-Bäume dominieren in Datenbanken
- Alternative: hash-basierte Indexierung für Gleichheitsprädikate
  - Aufteilung von möglichen Eingabewerten auf n Buckets mittels Hash-Funktion h
    - $h : \mathrm{dom}(key) \rightarrow [0, \dots, n-1]$
    - Anzahl an Buckets n: vorher festzulegen
      - $n \ll$ 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.
  - Suche nach k: Suche nach k in Bucket $h(k)$



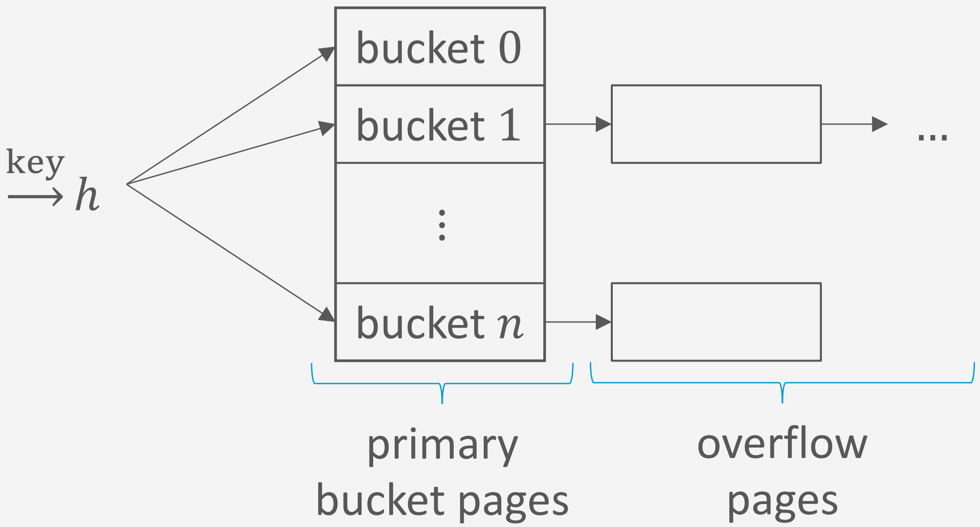
Statisches Hashing

- Relation $Student(Matrikelnummer, Name, Semester)$
- Hashfunktion $h(x) = x \bmod 3$, x Matrikelnummer
  - Anzahl an Buckets $n = 3$, daher $\bmod 3$
  - Kollisionsbehandlung mit Kollisionslisten
- Beispiel: (26830, ‘Aristoxenos, 8), (26120, ‘Fichte’, 10), (28106, ‘Carnap’, 3), … einfügen



Hash-basierte Indexierung

- Hash-Indexe eignen sich nur für Gleichheitsprädikate
  - Insbesondere für (lange) Zeichenketten
  - 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.KundenID=a.KundenID;
      ```



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**

Vor- und Nachteile von Indices

- 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
  - Die meisten Implementierungen legen Indices für Schlüssel und Unique-Eigenschaften zur schnellen Überprüfung automatisch an

```
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}
```

Zwischenzusammenfassung

- 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
- Bei Einsatz von Indices Kosten vs. Nutzen abwegen

Überblick: 6. Anfrageverarbeitung

A. *Speicherung*
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

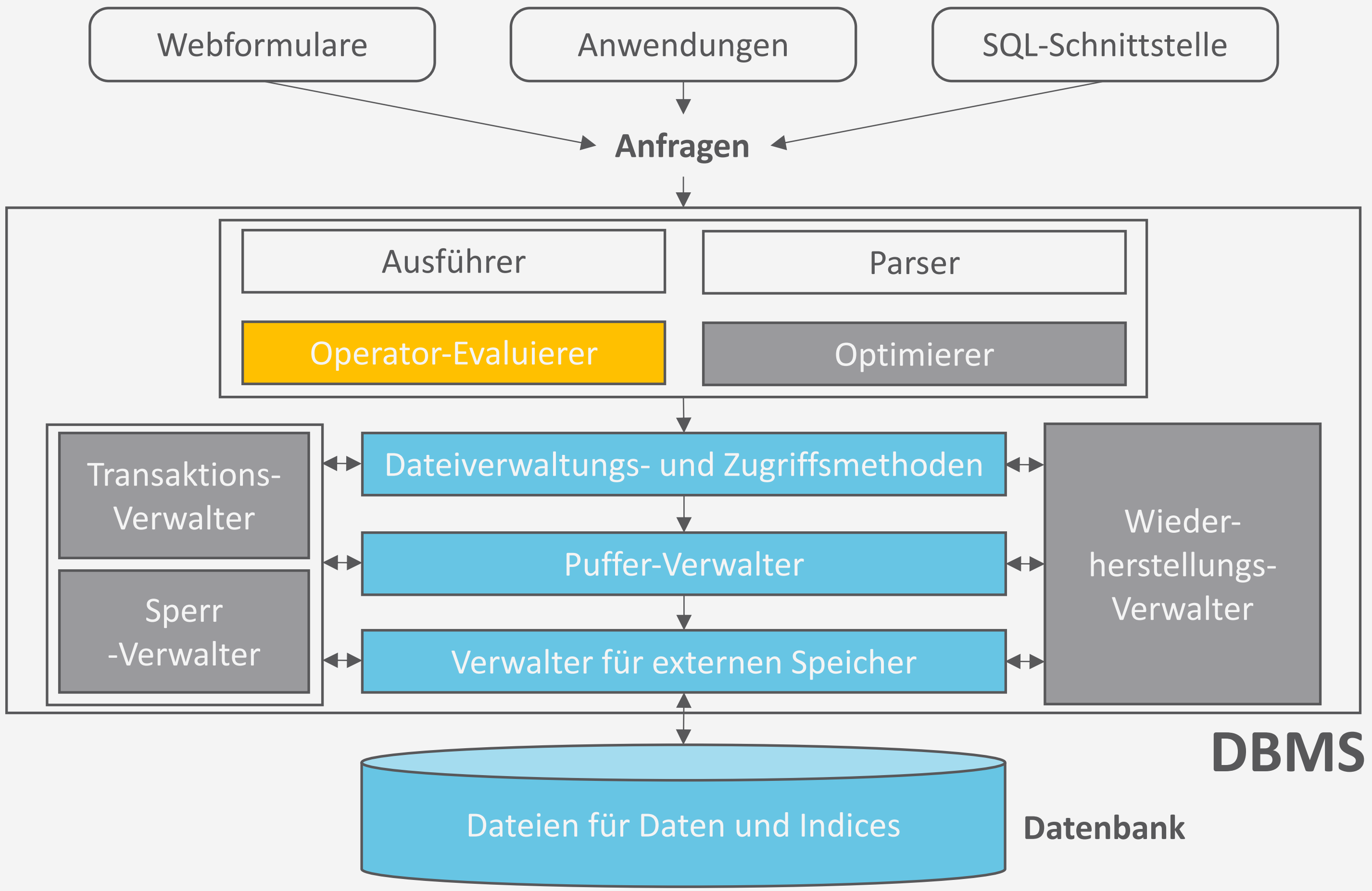
B. *Indexierung*
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. ***Anfragebeantwortung***
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. *Anfrageoptimierung*
- Rewriting
- Datenabhängige Optimierung

Architektur eines DBMS

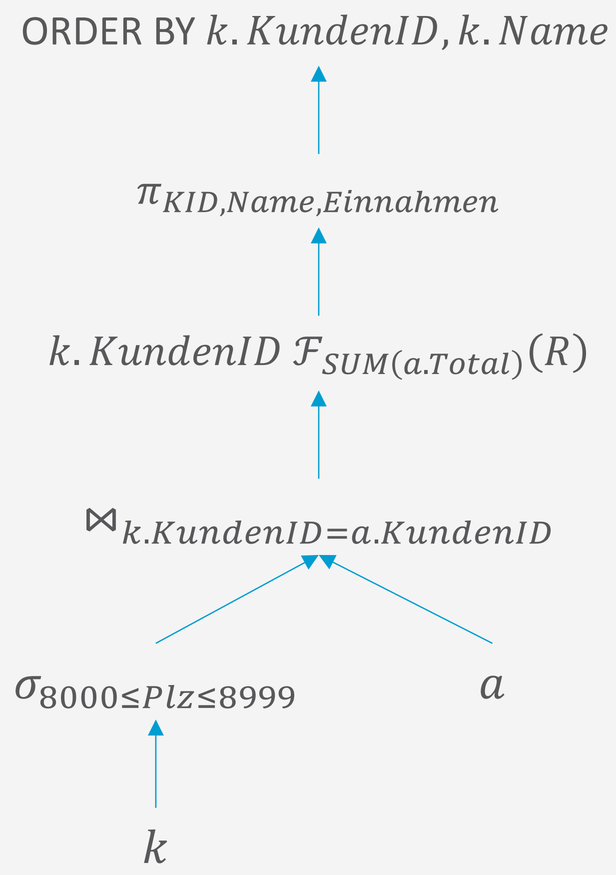
- Speicherung
- Anfragebeantwortung
  - Operator-Evaluierer
  - Optimierer
- Transaktionsmanagement



Ausführungspläne

- Operator zur Umsetzung einer Operation in einer Anfrage
  - Jeder Planoperator führt zur Verarbeitung einer vollständigen Anfrage eine Unteraufgabe aus
  - Zu Ausführungsplänen zusammensetzen
  - Nicht immer eindeutige Ausführungspläne

```sql
select k.KundenID, k.Name,
       sum (a.total) as Einnahmen
from Kunden k, Auftraege a
where k.Plz between 8000 and 8999
      and k.KundenID = a.KundenID
group by k.KundenID
order by k.KundenID, k.Name;
```



ORDER BY $k.KundenID, k.Name$

$\uparrow$

$\pi_{KundenID,Name,Einnahmen}$

Sortieren

Operator-Evaluierer

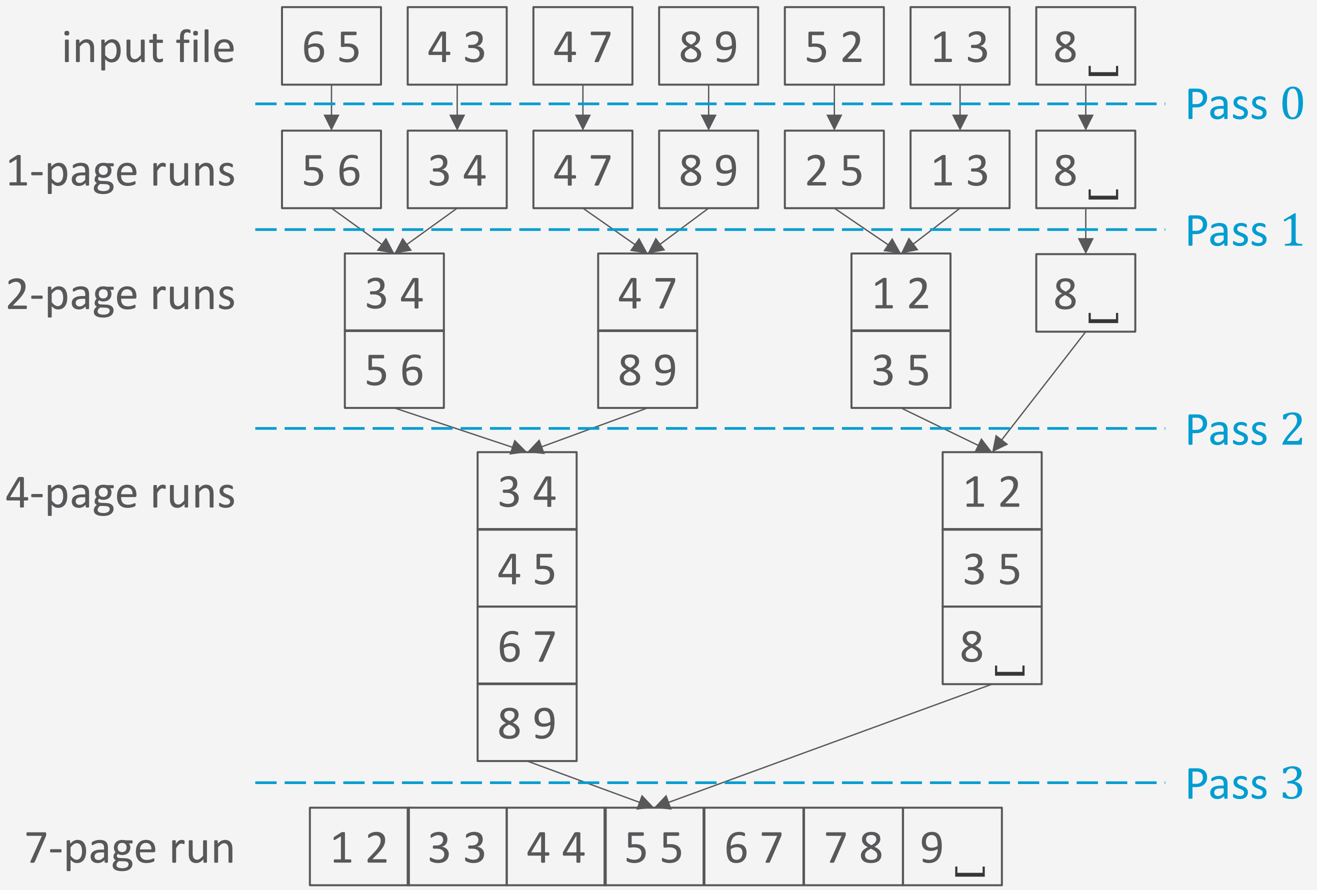
Effizientes Sortieren

- 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)

Wie können wir eine Datei sortieren, die nicht in den Hauptspeicher passt?

Zwei-Wege-Mischsortieren (*Two-way Merge Sort*)

- Implementierung mit nur drei Pufferseiten möglich
  → Zwei-Wege-Mischsortieren
  - Sortierung von N Datensätzen in mehreren Durchgängen
  - 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



$N \in \{2^{k-1}+1, \dots, 2^k\}$

Zwei-Wege-Mischsortieren

Pass 0 (Input: $N = 2^k$ unsorted pages, output: 2^k sorted runs)

1. Read N pages, one page at a time
2. Sort page records in main memory
3. Write sorted pages to disk (each page results in a *run*)

▹ This pass requires one page of buffer space.

Pass 1 (Input: $N = 2^k$ sorted runs, output: 2^{k-1} sorted runs)

1. Open two runs r_1 and r_2 from Pass 0 at a time for reading
2. Merge records from r_1 and r_2, reading input page-by-page
3. Write new two-page run to disk (page-by-page)

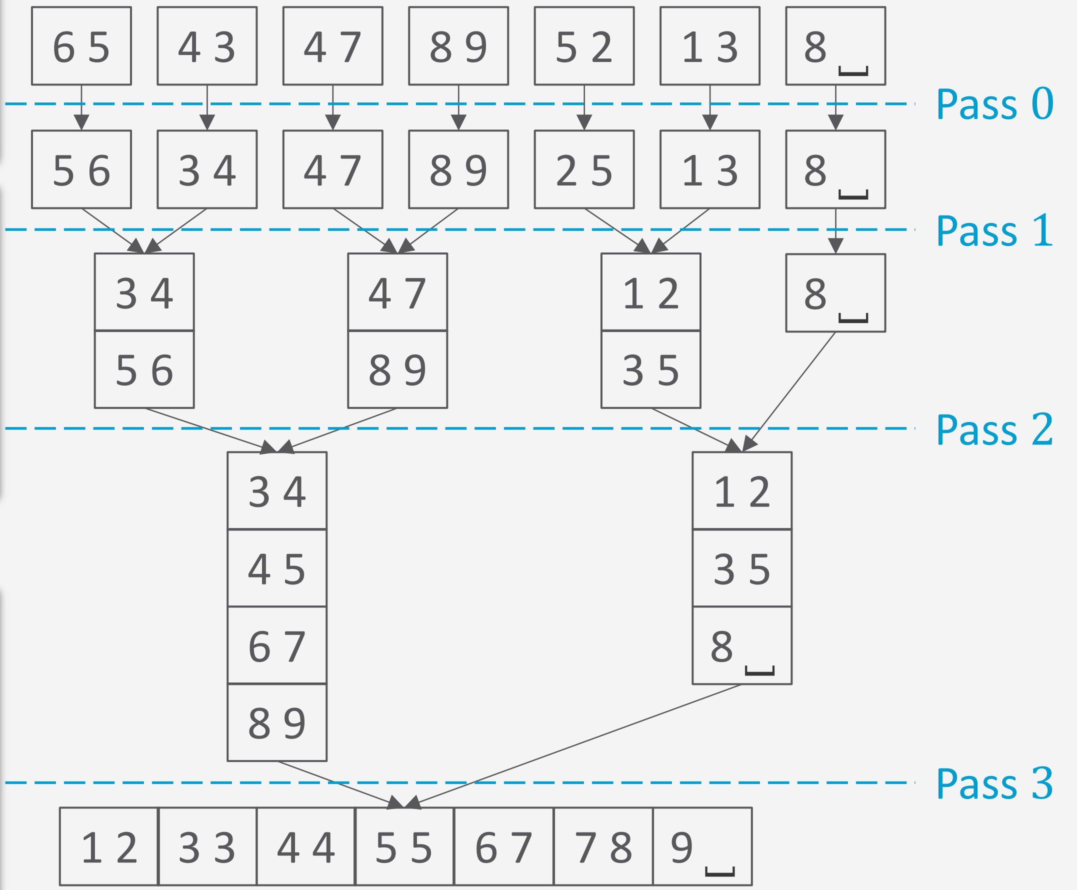
▹ This pass requires *three pages* of buffer space.

⋮

Pass n (Input: $N = 2^{k-n+1}$ sorted runs, output: 2^{k-n} sorted runs)

1. Open two runs r_1 and r_2 from Pass $n-1$ for reading
2. Merge records from r_1 and r_2, reading input page-by-page
3. Write new two-page run to disk (page-by-page)

▹ This pass requires *three pages* of buffer space.



Zwei-Wege-Mischsortieren: I/O-Verhalten

- Um eine Datei mit N Seiten zu sortieren, werden in jedem Durchgang N Seiten gelesen und geschrieben $\rightarrow 2 \cdot N$ I/O-Operationen pro Durchgang
- Anzahl der Durchgänge: $1 + \lceil \log_2 N \rceil$

  Durchgang 0 Durchgänge $1 \ldots k$

- Anzahl der I/O-Operationen: $2 \cdot N \cdot (1 + \lceil \log_2 N \rceil)$
  - Beispiel: Sortierung einer 8GB Datei bei einer Travelstar?
    - 8KB pro Seite, wahlfreier Zugriff mit Zugriffszeit $t = 14{,}33ms$ (siehe Folie 12)
    - Anzahl an Seiten $N = \frac{8GB}{8KB} = 10^6$
    - I/O-Operationen: $2 \cdot 10^6 \cdot (1 + \lceil 6 \cdot \log_2 10 \rceil) = 42 \cdot 10^6$
    - Zeit: $42 \cdot 10^6 \cdot 14{,}33ms \approx 7d$

Externes Mischsortieren

- Bisher freiwillig nur drei Seiten verwendet
- Wie kann ein großer Pufferbereich genutzt werden:
  B Seiten auf einmal bearbeiten
  - Mit B Seiten im Puffer können B Seiten eingelesen und im Hauptspeicher sortiert werden
  - Zwei wesentliche Stellgrößen
    - Reduktion der initialen Runs durch Verwendung des Pufferspeichers beim Sortieren im Hauptspeicher
    - Reduktion der Anzahl der Durchgänge durch Mischen von mehr als zwei Seiten

Externes Mischsortieren: Reduktion der initialen Runs

- Mit B Seiten im Puffer können B Seiten eingelesen und im Hauptspeicher sortiert werden
- Anzahl der I/O-Operationen: $2 \cdot N \cdot \left(1 + \left\lceil \log_2 \frac{N}{B} \right\rceil\right)$
  - Zugriffsmuster auf diese Ein-Ausgaben: Chunks von B Seiten sequenziell gelesen
  - Beispiel (Fortsetzung)
    - $B = 1000$
    - Durchgänge: $1 + \lceil \log_2 10^6/10^3 \rceil = 1 + \lceil \log_2 10^3 \rceil = 11$
    - $t_s + t_r$ bei 10 Merge-Durchgängen
      - Im ersten Durchgang t_s, t_r vernachlässigbar
      - Rest: $2 \cdot 10^6 \cdot 10 \cdot 14{,}33ms \approx 3d$
    - Transferzeit $t_{tr} = 2 \cdot 10^6 \cdot 11 \cdot 0{,}16ms \approx 1h$

Pass 0 (Input: N unsorted pages, output: $\lceil N/B \rceil$ sorted runs)
1. Read N pages, B pages at a time
2. Sort records in main memory
3. Write sorted pages to disk (resulting in $\lceil N/B \rceil$ *runs*)

▹ This pass uses B pages of buffer space.

Externes Mischsortieren: Reduktion der Anzahl der Durchgänge

- Mit B Seiten im Puffer können auch $B-1$ Seiten gemischt werden (eine Seite dient als Schreibpuffer)
- Anzahl der I/O-Operationen: $2 \cdot N \cdot \left(1 + \left\lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil \right)$
  - Zugriffsmuster auf diese Ein-Ausgaben: Random access
    - In den Mergephasen muss entschieden werden, welcher der minimalen Elemente auf Outputbuffer gehen.
  - Beispiel (Fortsetzung)
    - Durchgänge: $1 + \lceil \log_{999} 10^3 \rceil = 3$
    - 2 Merge-Durchgänge: $t_s + t_r$ $= 2 \cdot 10^6 \cdot 2 \cdot 14{,}33ms \approx 16h$
    - Transferzeit t_{tr} $= 2 \cdot 10^6 \cdot 3 \cdot 0{,}16ms \approx 1min$

Pass n (Input: $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$ sorted runs, output: $\frac{\lceil N/B \rceil}{(B-1)^{n}}$ sorted runs)

1. Open $B-1$ runs $r_1, \ldots, r_{B-1}$ from Pass $n-1$ for reading
2. Merge records from $r_1, \ldots, r_{B-1}$, reading input page-by-page
3. Write new $B \cdot (B-1)^n$-page run to disk (page-by-page)

▹ This pass uses B pages of buffer space.

Externes Mischsortieren: Blockweise Ein-Ausgabe

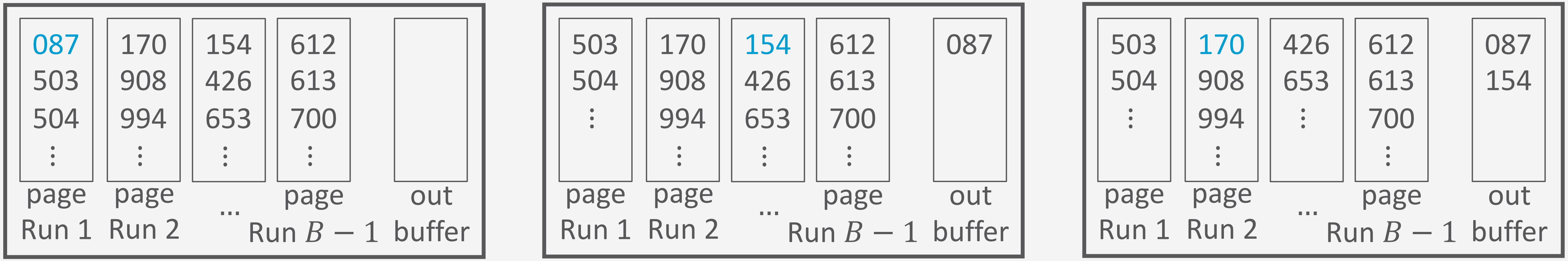
- Man kann das I/O-Muster verbessern, in dem man Blöcke von b Seiten in den Mischphasen verarbeitet
  - Alloziere b Seiten für jede Eingabe (statt nur eine)
  - Reduktion der Ein-Ausgabe um Faktor b pro Seite
    - Anzahl der I/O-Operationen: $2 \cdot \left\lceil \frac{N}{b} \right\rceil \cdot \left(1 + \left\lceil \log_{\left\lceil \frac{B}{b} \right\rceil - 1} \left\lceil \frac{N}{\left\lceil \frac{B}{b} \right\rceil} \right\rceil \right\rceil \right)$
  - Preis: Reduzierte Einfächerung (was in mehr Durchgängen und damit in mehr I/O-Operationen resultiert)
  - Beispiel (Fortsetzung)
    - 63 Sektoren pro Spur, Track-to-Track-Suchzeit $t_{s,t2t} = 1ms$
    - Blockweises I/O mit $b = 32$
    - 1 Block mit 8KB benötigt 16 Sektoren
    - Ca. $10 \cdot 31.250$ Plattenzugriffe mit je $27.42ms \rightarrow 2{,}38h$

Externes Mischsortieren: Hauptspeicher als Ressource

- In der Praxis meist genügend Hauptspeicher vorhanden, so dass Dateien in einem Mischdurchgang sortiert werden kann (mit blockweisem I/O)
  - Beispiel (Fortsetzung)
    - Benötigter Speicher um mit einem Mischvorgang auszukommen: 256MB

Bisher nur IO-Zeit betrachtet

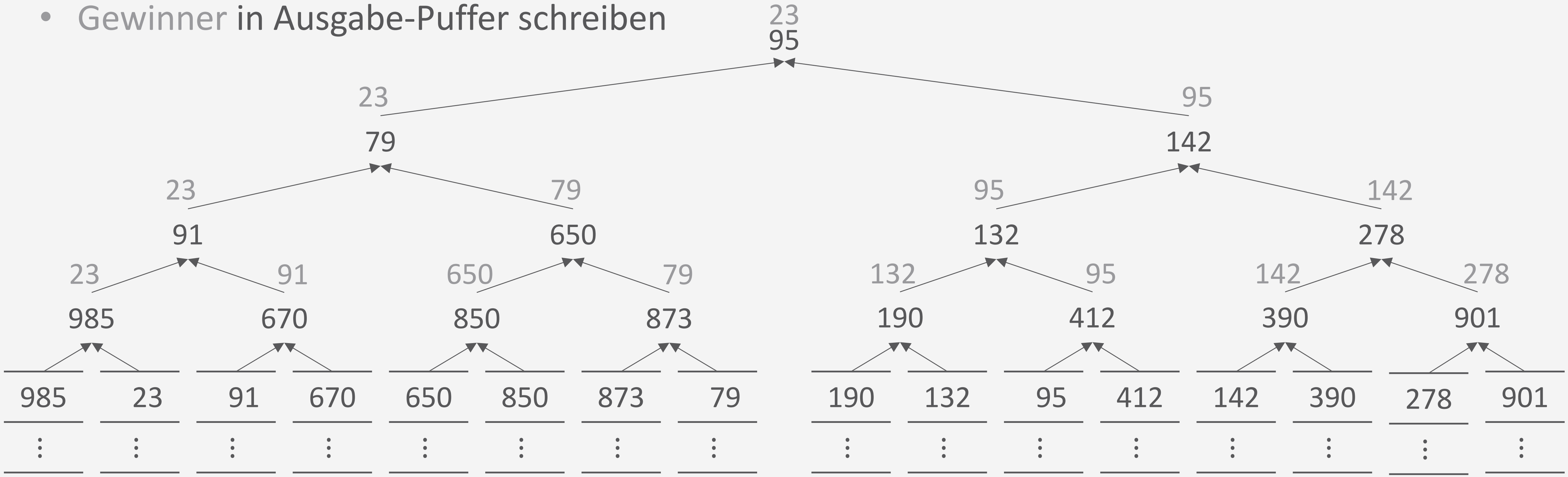
- Auswahl des nächsten Datensatzes für den Output unter $B - 1$ (oder $^{B}/_{b} - 1$) Eingabeläufen kann CPU-intensiv sein ($B - 2$ Vergleiche)
- Beispiel: $B - 1 = 4$, Ordnung: $<$



D. Knuth, The Art of Computer Programming,vol.3, **1973**

Tree of Losers (and Hidden Winners)

- Verwende Auswahlbaum zur Kostenreduktion
  - Reduktion der Vergleiche auf $\log_2(B-1)$ bzw. $\log_2({}^{B}/_{b}-1)$
  - Gewinner in Ausgabe-Puffer schreiben



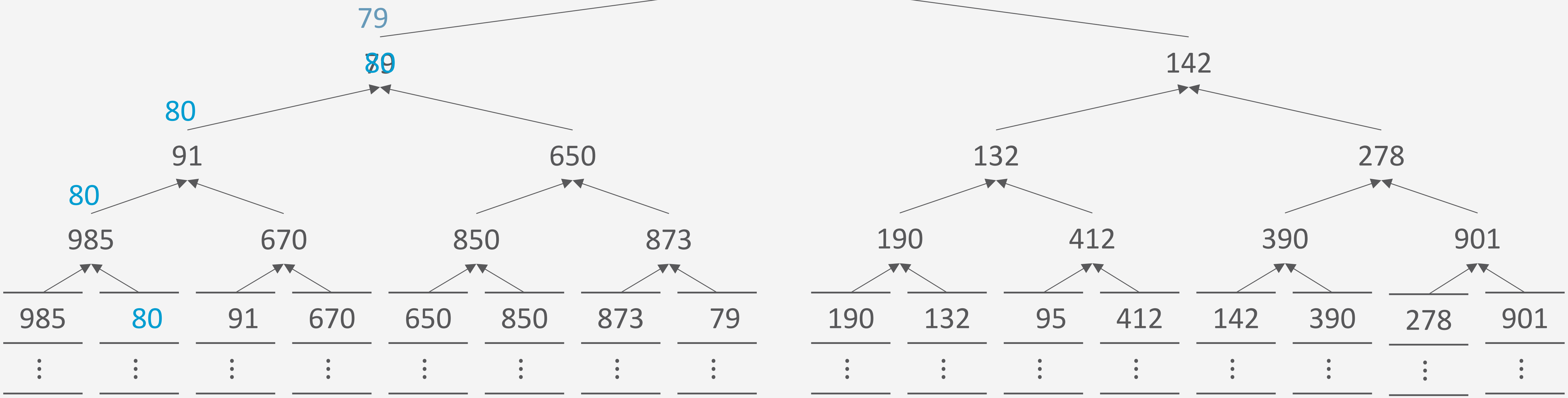
$B-1=16$ Runs

D. Knuth, The Art of Computer Programming,vol.3, **1973**

Tree of Losers (and Hidden Winners): Nächstes Element

Aktualisierungsaufwand nur im Gewinner-Pfad

- Im Gewinner-Run: Nächste Zahl nach oben propagieren
  - Solange bis andere Zahl v kleiner ist, dann Zahl ersetzen und v nach oben propagieren
  - Bis an Wurzel angekommen → Kleinste Zahl schreiben und wieder von vorn



D. Knuth, The Art of Computer Programming,vol.3, **1973**

$$R = \bowtie_{k.KundenID=a.KundenID}$$

$$\sigma_{8000 \leq Plz \leq 8999} \qquad a$$

Join-Implementierung

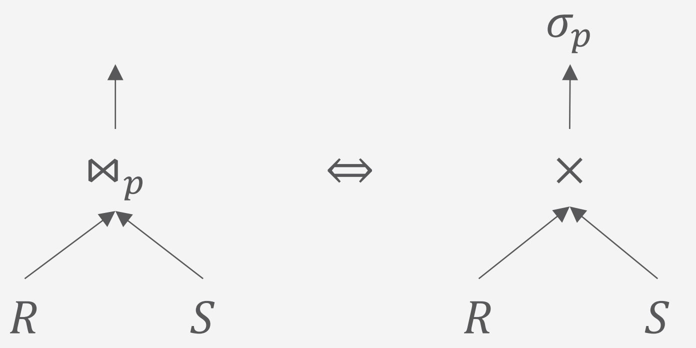
Operator-Evaluierer

Verbundoperator (Join) $R \bowtie S$

- Join $R \bowtie_p S$ ist Abkürzung für Kreuzprodukt mit anschließender Selektion $\sigma_p(R \times S)$

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

- Aber:
  Größe des Zwischenresultats aus Schritt 1: $|R| \cdot |S|$
  - Ineffizienz kann überwunden werden



Join-als-geschachtelte-Schleifen

- Einfache Implementierung des Joins
  - nl = nested loop
- Sei N_R und N_S die Seitenzahl von R und S
- Sei p_R und p_S die Anzahl der Datensätze pro Seite in R und S
- Anzahl an Plattenzugriffen

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

  - Jede der N_R Seiten muss geladen werden
  - Für jedes Tupel auf den N_R muss jede der N_S Seiten geladen werden

function $\mathrm{nljoin}(R, S, p)$
    $result \leftarrow \emptyset$
    for each record $r \in R$ **do**
        for each record $s \in S$ **do**
            if $\langle r, s \rangle$ satisfies p **then**
                Append $\langle r, s \rangle$ to $result$
    return $result$

Join-als-geschachtelte-Schleifen

- Geringer Speicherbedarf
  - Nur drei Seiten nötig
    - Zwei Seiten fürs Lesen von R und S
    - Eine Seite um das Ergebnis zu schreiben

```
function nljoin(R, S, p)
    result ← ∅
    for each record r ∈ R do
        for each record s ∈ S do
            if ⟨r, s⟩ satisfies p then
                Append ⟨r, s⟩ to result
    return result
```

- I/O-Verhalten: Sehr viele wahlfreie Zugriffe
  - Annahme $p_R = p_S = 100, N_R = 1000, N_S = 500$:
    $$N_R + p_R \cdot N_R \cdot N_S = 1000 + 5 \cdot 10^7$$
  - 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
    - Abwechselndes 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 $\text{nljoin}(R, S, p)$

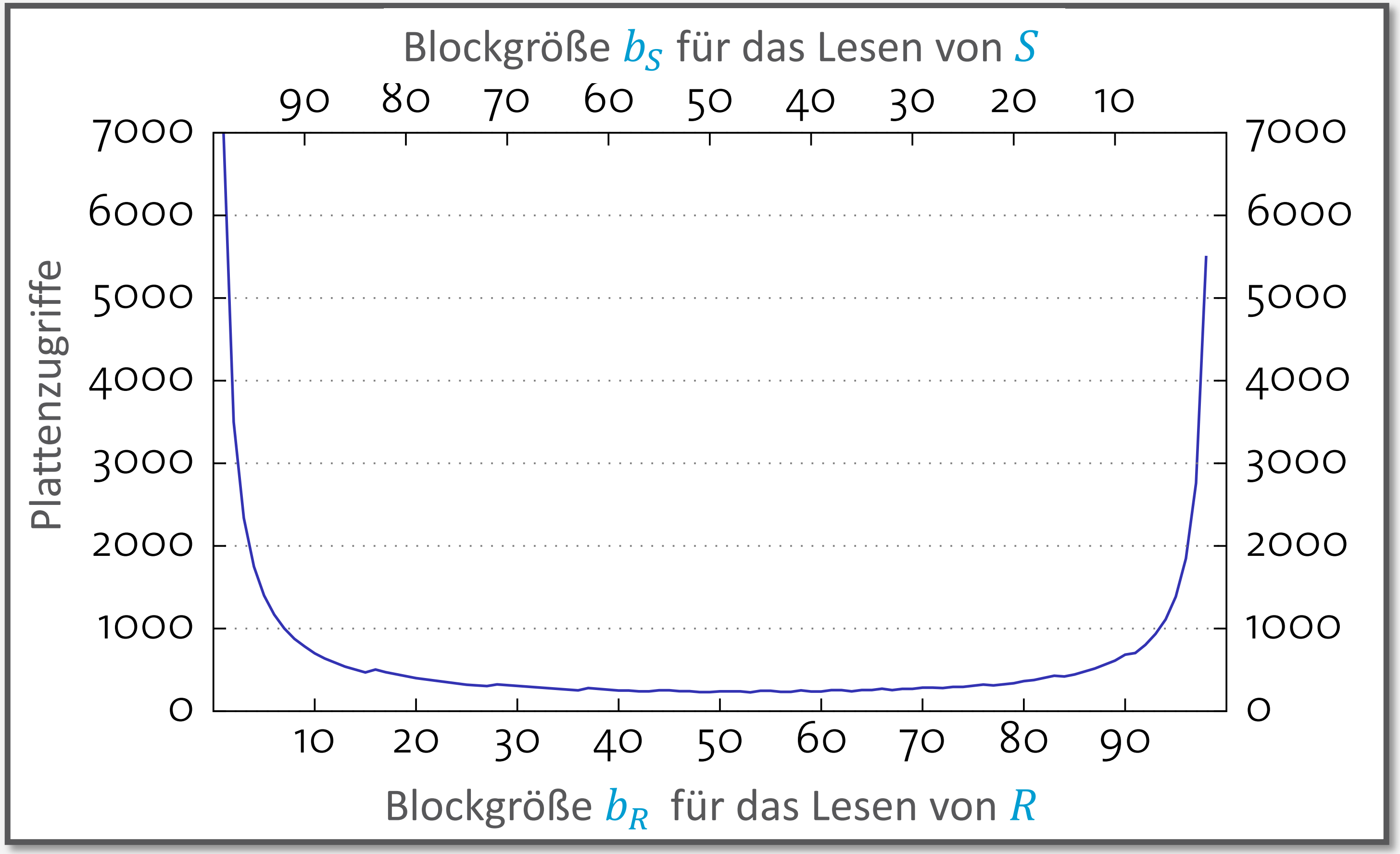
```
function block_nljoin(R, S, p)
    result ← ∅
    for each b_R-sized block in R do
        for each b_S-sized block in S do
            Find matches in current R and S block
                and append them to result
    return result
```

Join im Hauptspeicher ausführbar

- Plattenzugriffe
  - R wird (einmal) vollständig gelesen, aber mit nur $\left\lceil\frac{N_R}{b_R}\right\rceil$ Plattenzugriffen
  - S wird nur $\left\lceil\frac{N_R}{b_R}\right\rceil$ mal gelesen, mit $\left\lceil\frac{N_R}{b_R}\right\rceil \cdot \left\lceil\frac{N_S}{b_S}\right\rceil$ Plattenzugriffen

Wahl von b_R und b_S

- Pufferbereich mit
  - $B = 100$ Rahmen
  - $N_R = 1000$
  - $N_S = 500$



Performanz des Hauptspeicher-Joins

- Anweisung im Inneren bedingt einen Hauptspeicherverbund zwischen Blöcken aus R und S
- Aufbau einer Hashtabelle kann den Verbund erheblich beschleunigen
  - Funktioniert nur für Gleichheitsprädikate im Join

function $\text{block\_nljoin}(R, S, p)$
  $result \leftarrow \emptyset$
  for each b_R-sized block in R **do**
    for each b_S-sized block in S **do**
      Find matches in current R and S block
        and append them to $result$
  return $result$

function $\text{block\_nljoin}'(R, S, p)$
  $result \leftarrow \emptyset$
  for each b_R-sized block in R **do**
    Build an in-memory hash table H for current R block
    for each b_S-sized block in S **do**
      for each record s in current S block **do**
        Probe H and append matching $\langle r, s \rangle$ tuples to $result$
  return $result$

Warum Hashtabelle für R Block und nicht S Block?

Indexbasierte Verbunde $R \bowtie S$

- Verwendung eines vorhandenen Index für die innere Relation S
  - Ggf. innere und äußere Relation vertauschen (nur Index für R vorhanden)
- 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

Suchschlüssel
für Suche im Index

function index_nljoin(R, S, p)
  $result \leftarrow \emptyset$
  for each record $r \in R$ **do**
    Probe index using r and
      append all matching tuples to $result$
  return $result$

$R = \bowtie_{k.KundenID=a.KundenID}$

$\sigma_{8000 \leq Plz \leq 8999}$ a

Sortier-Merge-Join

- Join-Berechnung einfach, wenn Relationen bzgl. Join-Attribut(en) sortiert
  - Kombiniere Eingabetabellen ähnlich wie beim Merge-Sort
    - Nur für Gleichheitsprädikate

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

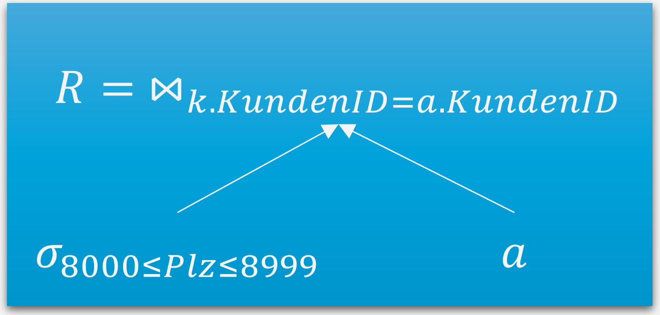
function $\text{merge\_join}(R, S, \alpha = \beta)$ ▹ α, β join columns in R, S
  $result \leftarrow \emptyset$
  $r \leftarrow$ position of first tuple in R
  $s \leftarrow$ position of first tuple in S
  while $r \neq$ **eof** and $s \neq$ **eof do**
    while $r.\alpha < s.\beta$ **do**
      Advance r
    while $r.\alpha > s.\beta$ **do**
      Advance s
    $s' \leftarrow s$ ▹ Remember current position in s
    while $r.\alpha = s'.\beta$ **do** ▹ All R tuples with same α value
      $s \leftarrow s'$ ▹ Rewind s to s'
      while $r.\alpha = s.\beta$ **do** ▹ All S tuples with same β values
        Append $\langle r, s\rangle$ to $result$
        Advance s
      Advance r
  return $result$

Merge-Join: I/O-Verhalten

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

  $N_R + N_S$ (das ist dann optimal)

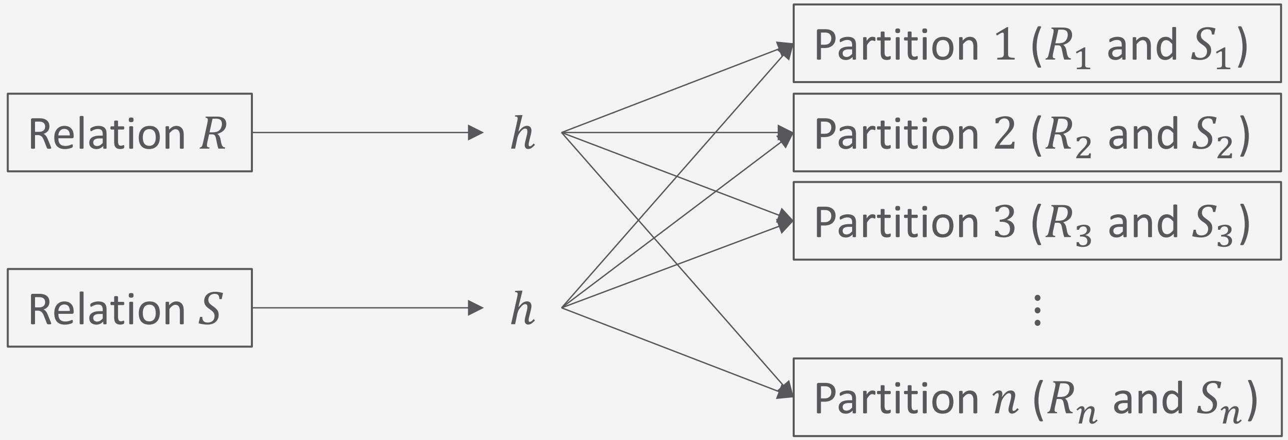
  - Durch blockweises I/O treten fast immer sequenzielle Lesevorgänge auf
- Vorher sortieren kann sich also für Join-Berechnung auszahlen
- Ausgabe weiterhin sortiert
  - Wenn später eine Sortierung der Ausgabe gefordert wird, lohnt sich die vorherige Sortierung noch mehr
  - Zudem weniger Festplattentransfers 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 $R_1, \dots, R_n$ und $S_1, \dots, S_n$ 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

- 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

function hash_join$(R, S, \alpha = \beta)$
    $result \leftarrow \emptyset$
    for each record $r \in R$ **do**
        Append r to partition $R_{h(r.\alpha)}$
    for each record $s \in S$ **do**
        Append s to partition $S_{h(s.\beta)}$
    for each partition $i \in 1, \ldots, n$ do
        Build hash table H for R_i using hash function h'
        for each block in S_i **do**
            for each record s in current S_i block **do**
                Probe H and append matching tuples to $result$
    return $result$

$$k.KundenID\ \mathcal{F}_{SUM(a.Total)}(R)$$
$$\uparrow$$
$$R$$

Gruppierung + UNIQUE-Behandlung

Operator-Evaluierer

Gruppierung und Duplikate-Elimination

- Herausforderung: Finde *identische* Datensätze in einer Datei
  - Duplikate-Elimination: Identische Datensätze bzgl. aller Attribute zur Eliminierung
  - Gruppierung: Identisch bzgl. Gruppierungsattribut(e) zur Gruppierung

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

$\pi_{KundenID,Name,Einnahmen}(S)$

$\sigma_{8000 \leq Plz \leq 8999}$

Selektion und Projektion

Anfrageverarbeitung

Andere Anfrage-Operatoren

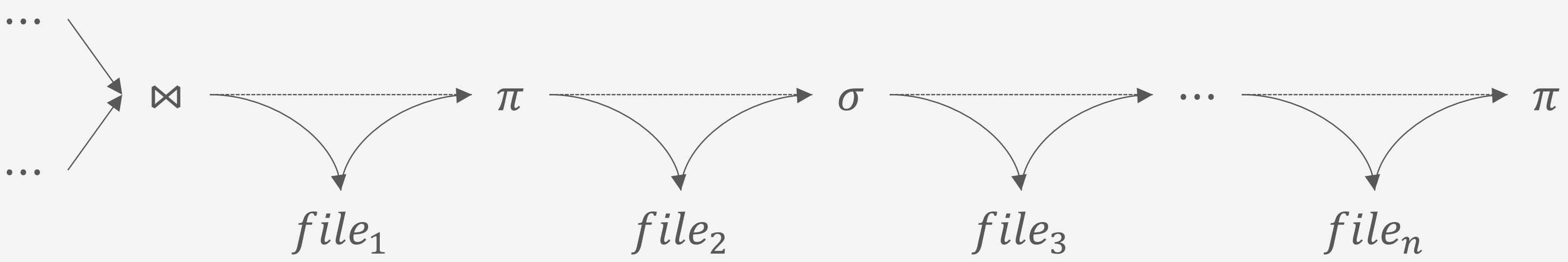
- Projektion π
  - Implementierung durch
    a. Entfernen nicht benötigter Spalten
    b. Eliminierung von Duplikaten
  - Die Implementierung von
    - a. Bedingt das Ablaufen (scan) aller Datensätze in der Datei
    - b. Siehe vorherigen Abschnitt zu Duplikate-Eliminierung
  - Systeme vermeiden b. sofern möglich
- Selektion σ
  - Ablaufen (scan) aller Datensätze
  - Eventuell Sortierung ausnutzen oder Index verwenden

Pipelining

Anfrageverarbeitung

Organisation der Operator-Evaluierung

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



- 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

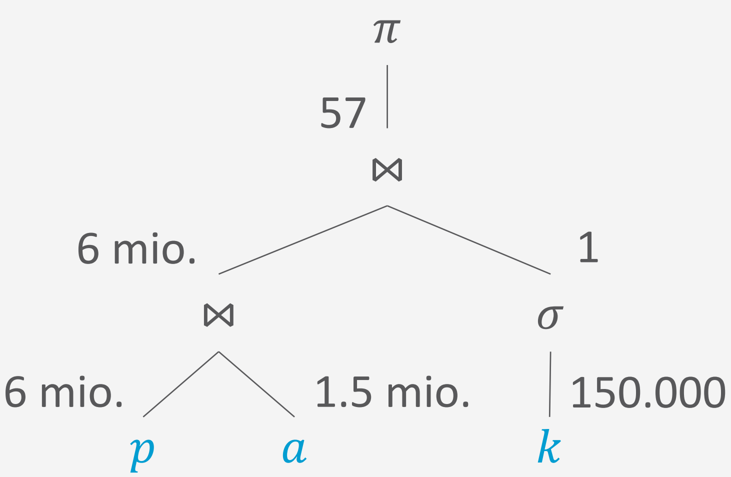
- Alternativ könnte jeder Operator seine Ergebnisse direkt an den nachfolgenden senden, ohne die Ergebnisse erst auf die Platte zu schreiben
- Ergebnisse werden so früh wie möglich weitergereicht und verarbeitet (Pipeline-Prinzip)
- Granularität ist bedeutsam:
  - Kleinere Brocken reduzieren Antwortzeit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

- Siehe auch Gebiet der Stromverarbeitung

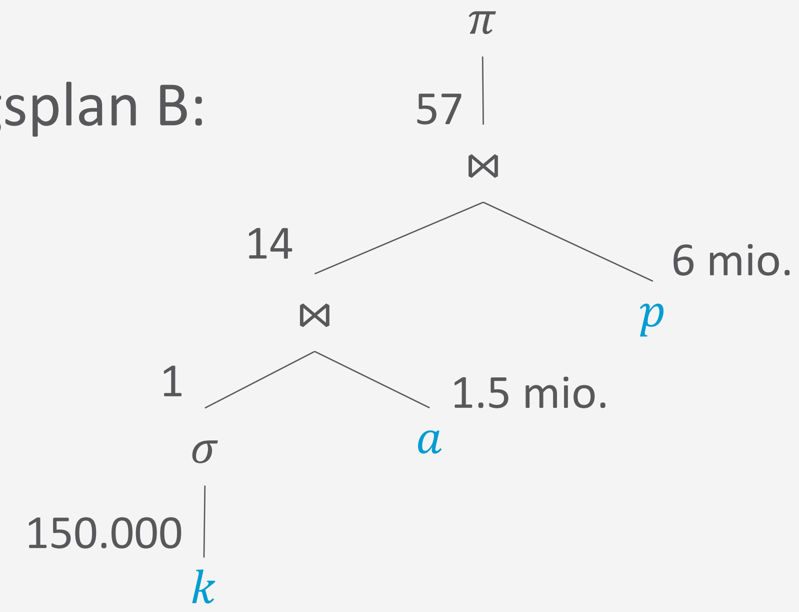
Auswirkungen auf die Performanz

```sql
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.';
```

- Ausführungsplan A:



- Ausführungsplan B:



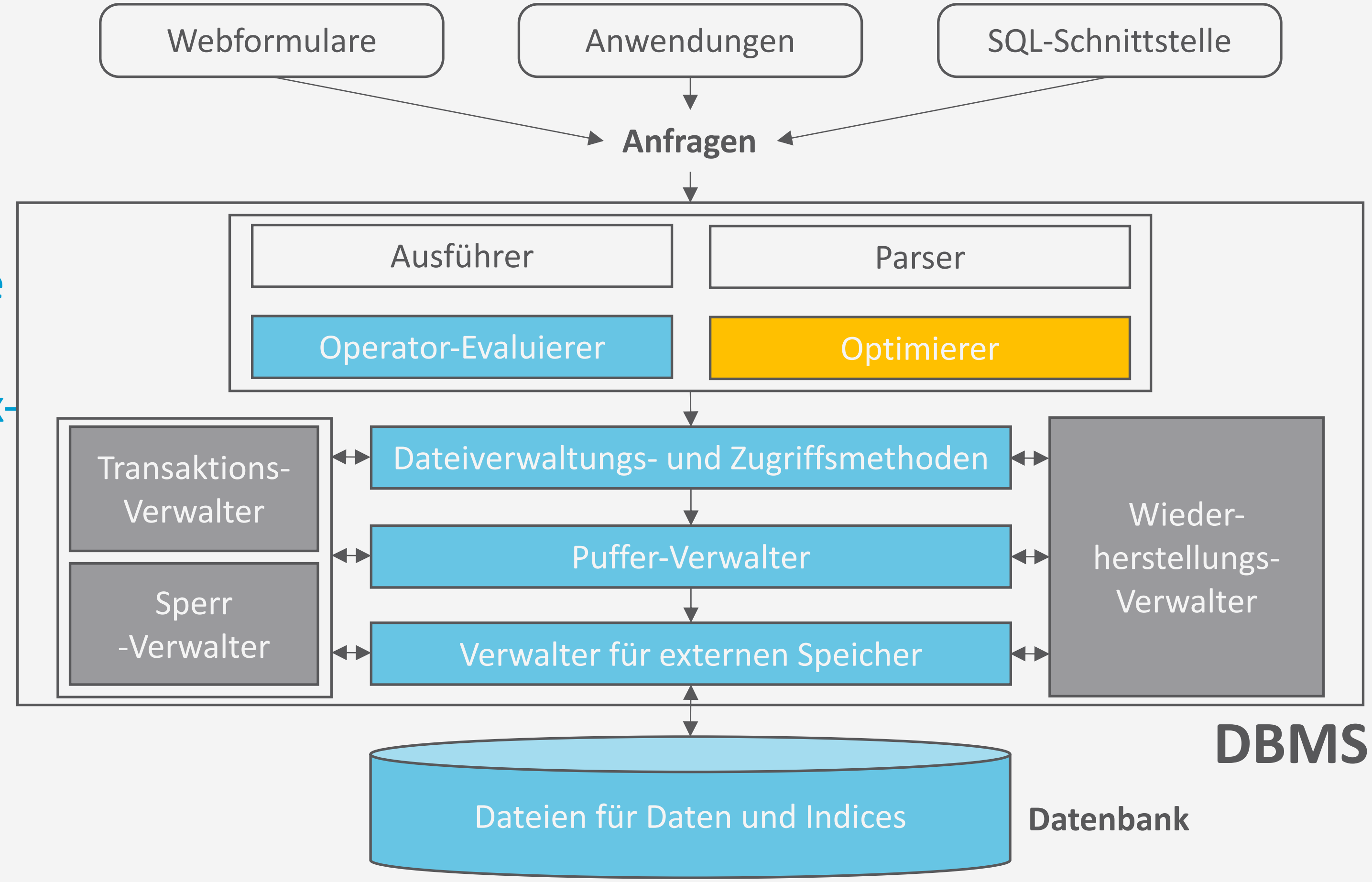
- Ermöglicht
  - Tupelweises Verarbeiten der Relation k
  - Sofortiges Weiterleiten nach der Selektion
  - Kombiniert mit tupelweisem Verarbeiten der Relationen a und p

Blockierende Operatoren

- Pipelining reduziert Speicheranforderungen und Antwortzeiten, da jeder Datensatz gleich weitergeleitet
- Funktioniert so nicht für alle Operatoren Warum?
  - 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

Architektur eines DBMS

- Speicherung
- Anfragebeantwortung
  - Operator-Evaluierer
    - Sortieren: externes Merge-Sort, Tree of Losers
    - Join-Verarbeitung: blockweise, index-basiert, Sortier-Merge, hash-basiert
    - Gruppierung, Duplikate-Eliminierung, Selektion, Projektion möglicherweise unterstützt durch Indices / Sortierung
    - Pipelining
  - Optimierer



Überblick: 6. Anfrageverarbeitung

A. *Speicherung*
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

B. *Indexierung*
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

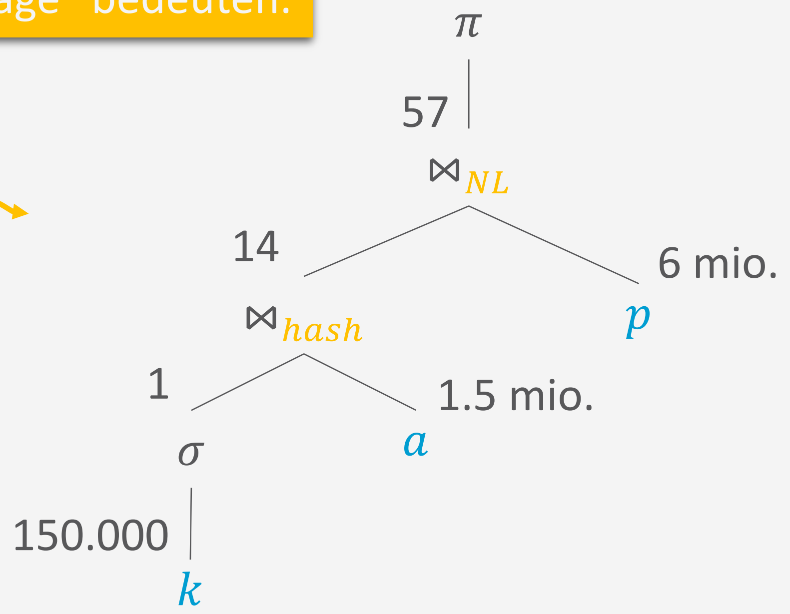
C. *Anfragebeantwortung*
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. ***Anfrageoptimierung***
- Rewriting
- Datenabhängige Optimierung

Anfrageoptimierung

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

```
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.';
```



- Neben der Anordnung in den Ausführungsplänen gibt es weitere Entscheidungen, die auf die Anfragebeantwortung Auswirkung haben
  - Welche Implementation eines Join-Operators?
  - Welche Parameter für Blockgrößen, Pufferallokation, ...
  - Automatisch einen Index aufsetzen?
- Aufgabe, den besten Ausführungsplan zu finden: Heilige Gral der DB-Implementierung

Optimierung

- Optimierungen können unabhängig von den Daten erfolgen: Rewriting
  - Selektionsprädikate vereinfachen / früh anwenden
  - Geschachtelte Anfragen entschachteln bzw. Joins explizit machen
  - Vermeide Duplikate-Elimination, wenn möglich
- Datenabhängige Optimierung: Optimiser
  - Kostenbasiert auf Basis der Daten bzw. statistisch relevanter Größen der DB

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

SQL Anfrage
↓
Parser
↓
Rewriter (Datenunabhängige Optimierung
↓
Optimizer (Datenabhängige Optimierung)
↓
Plan

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
  - ```
    select *
    from Einzelposten p
    where p.Steuern * 100 < 5;
    ```
    Non-Sargable

  um in

  - ```
    select *
    from Einzelposten p
    where p.Steuern < 0.05;
    ```
    Sargable

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

Geschachtelte Anfragen

- SQL bietet viele Wege, geschachtelte Anfrage zu schreiben
  - Unkorrelierte Unteranfragen
    → nur einmal auswerten

```
select *
from Auftraege
where Kunden_ID in
   (select Kunden_ID
    from Kunden
    where Name = 'IBM Corp.‘);
```

  - Korrelierte Unteranfragen
    → für jedes Tupel auswerten

```
select *
from Auftraege a
where Kunden_ID in
   (select Kunden_ID
    from Kunden k
    where k.Kontostand = a.Gesamtpreis);
```

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

Won Kim. On Optimizing an SQL-like Nested Query. ACM TODS, vol. 7, no. 3, **1982**

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
    ```

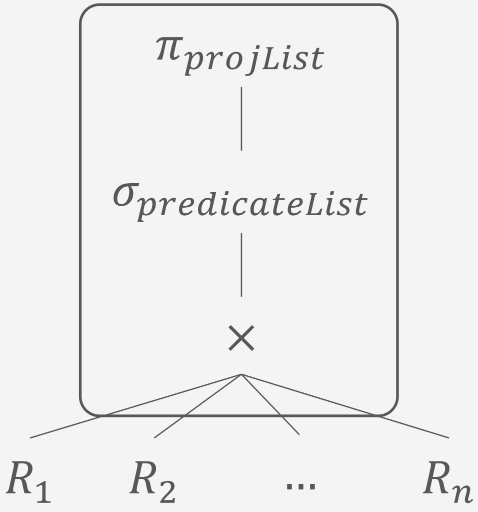
- Hierdurch werden Pläne möglich wie $(A \bowtie C) \bowtie B$

Optimiser: Kostenbasierte Optimierung

Anfragenoptimierer

Abschätzung der Ergebnisgröße

- Betrachte Anfrageblock für SELECT-FROM-WHERE-Anfrage Q
  - $R_1, \dots, R_n$ Eingabetabellen im FROM
  - Mit einer Projektion $\pi_{projList}$ im SELECT
  - Mit einer Selektion $\sigma_{predicateList}$ im WHERE



- Abschätzung der Ergebnisgröße von Q durch

$$|Q| = |R_1| \cdot \dots \cdot |R_n| \cdot \mathrm{sel}(predicateList)$$

  - wobei
    - $|R_1|, \dots, |R_n|$ Größe der Eingabetabellen
    - $\mathrm{sel}(predicateList)$ **Selektivität** von σ

Tabellengrößen

- Größe einer Tabelle über den Systemkatalog verfügbar
  - Hier IBM DB2, Anzeige von
    - Tabellenname
    - Kardinalität
    - Anzahl der Seiten im Speicher
  - Vor Ausführung der Anfrage verfügbar
    - Bei DB-Änderungen wird Tabelle aktualisiert

```
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.
```

Selektivität

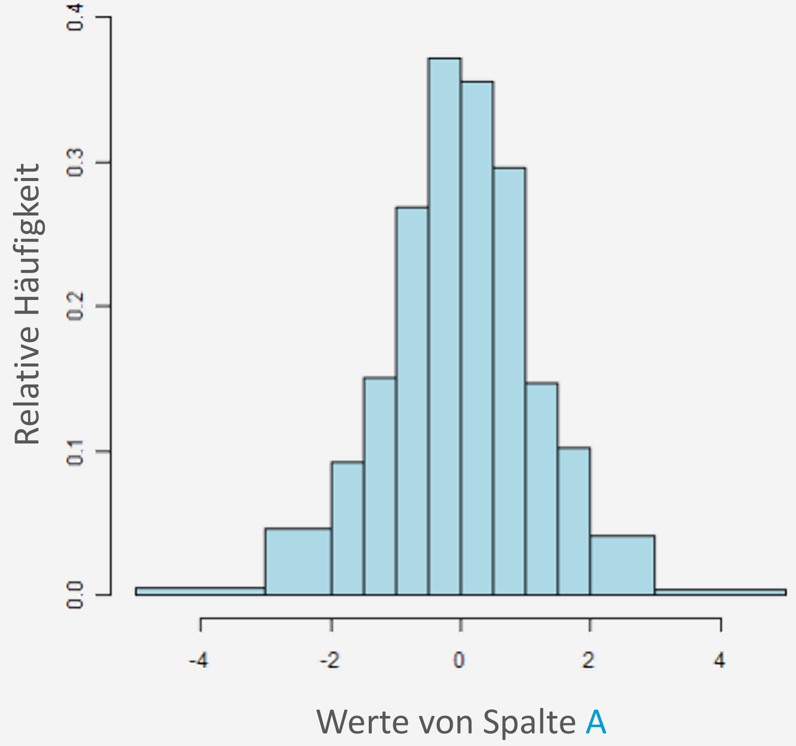
- Grobe Abschätzung durch Induktion über die Struktur des Anfrageblocks
  - $V(A,R)$ = Anzahl verschiedener Werte von Attribut (Spalte) A in Relation R Woher?
- $R.A = value$: $\quad \text{sel}(\cdot) = \begin{cases} \frac{1}{V(A,R)} & \text{falls } \exists V(A,R) \\ \frac{1}{10} & \text{sonst} \end{cases}$ Warum $\frac{1}{10}$?
- $R.A = S.B$: $\quad \text{sel}(\cdot) = \begin{cases} \frac{1}{\max\{V(A,R),V(B,S)\}} & \text{falls } \exists V(A,R) \land \exists V(B,S) \\ \frac{1}{V(Attr,Rel)} & \text{falls entweder } \exists V(A,R) \text{ oder } \exists V(B,S) \\ \frac{1}{10} & \text{sonst} \end{cases}$
- $p_1 \land p_2$: $\quad \text{sel}(\cdot) = \text{sel}(p_1) \cdot \text{sel}(p_2)$
- $p_1 \lor p_2$: $\quad \text{sel}(\cdot) = \text{sel}(p_1) + \text{sel}(p_2) - \text{sel}(p_1) \cdot \text{sel}(p_2)$

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 t mit $x_i - 1 < t.A \leq x_i$ bzw. mit $t.A = x$
  2. Anzahl verschiedener Werte von A im Intervall $(x_i - 1, x_i]$, absolut oder relativ



WWU MÜNSTER

Histogramme

- DB2: SYSCAT.COLDIST enthält Informationen wie
  - DB2: TYPE='Q' Quantile (cumulative), TYPE='F' Frequency
  - k-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_EXTPRICE'
       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
```

Bessere Abschätzung der Selektivität

- $MCV(A, R) = k$-häufigsten (top-k) Werte in Spalte A einer Tabelle R
- $MCF(A, R) =$ Häufigkeiten dieser Werte
  - $MCF(A, R)[value]$: Zugriff auf Häufigkeit von $value$, falls $value \in MCV(A, R)$
- Verbesserte Abschätzung für $R.A = value$
- $R.A = value$: $$\mathrm{sel}(\cdot) = \begin{cases} \frac{1}{MCF(A,R)[value]} & \text{falls } value \in MCV(A,R) \\ \frac{1}{V(A,R)} & \text{falls } value \notin MCV(A,R)\text{, aber } \exists V(A,R) \\ \frac{1}{10} & \text{sonst} \end{cases}$$

Kardinalitätsabschätzung für Projektion

- Anfrage $Q = \pi_L(R)$ mit $L = (A_1, \ldots, A_k)$ Liste von Spalten
- $|Q| = \begin{cases} V(A,R) & \text{falls } L = (A) \text{ (mit Duplikatseliminierung)} \\ |R| & \text{falls Schlüsselattribut(e) von } R \text{ in } L \\ |R| & \text{ohne Duplikateneliminierung} \\ \min\{|R|, \prod_{A \in L} V(A,R)\} & \text{sonst} \end{cases}$

Kardinalitätsabschätzungen für Mengenoperationen

- Abschätzung der Kardinalitäten für
  - Vereinigung (∪):
    $$|R \cup S| \leq |R| + |S|$$
  - Differenz (−):
    $$\max\{0, |R| - |S|\} \leq |R - S| \leq |R|$$
  - Kartesisches Produkt (×):
    $$|R \times S| = |R| \cdot |S|$$

Kardinalitätsabschätzung für Join

- Im Allgemeinen nicht-trivial
- Bei Fremdschlüsselbeziehungen wie folgt abschätzbar:
  - Fremdschlüssel $S.A$ auf Primärschlüssel $R.A$
    - $|R \bowtie_{R.A=S.A} S| = |S|$
  - Bei Fremdschlüssel auf sonstiges Attribut
    - $|R \bowtie_{R.A=S.B} S| =$

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

```
create table R (
    A int not null primary key, ...);
create table S (
    A int not null references R(A), ...);
```

```
create table R (
    A int, ...);
create table S (
    B int references R(A), ...);
```

```
create table R (
    A int references S(B), ...);
create table S (
    B int, ...);
```

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

$(R \bowtie S) \bowtie T$ $\quad (S \bowtie R) \bowtie T$ $\quad (R \bowtie T) \bowtie S$ $\quad (S \bowtie T) \bowtie R$ $\quad (T \bowtie R) \bowtie S$ $\quad (T \bowtie S) \bowtie R$

$R \bowtie (S \bowtie T)$ $\quad S \bowtie (R \bowtie T)$ $\quad R \bowtie (T \bowtie S)$ $\quad S \bowtie (T \bowtie R)$ $\quad T \bowtie (R \bowtie S)$ $\quad T \bowtie (S \bowtie R)$

Join-Optimierung: Suchraum

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

Anzahl an Relationen	C_{n-1}	Anzahl an Bäumen
2	1	2
3	5	12
4	14	120
5	42	1.680
6	132	30.240
7	429	665.280
8	1.430	17.297.280
10	16.796	17.643.225.600

Anzahl der Bäume für n Eingaberelationen:

$$n! \cdot C_{n-1} = \frac{\big(2(n-1)\big)!}{(n-1)!}$$

$C_{n-1} = \frac{(2(n-1))!}{n!(n-1)!}$ (($n-1$)-te Catalanzahl)

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

- Annahme: Optimalitätsprinzip

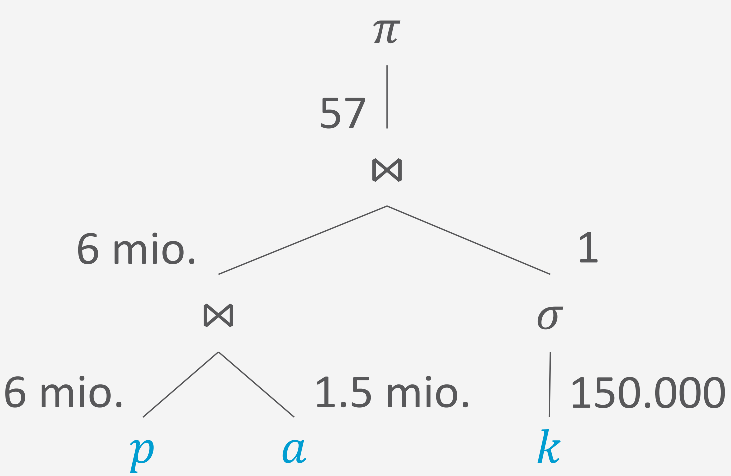
  Um den global optimalen Plan zu finden, reicht es aus, die optimalen Pläne bzgl. der Unteranfragen zu betrachten
  - Muss nicht gelten!

P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price: Access Path Selection in a Relational Database Management System. *In* Proc. ACM SIGMOD Conference on the Management of Data, pages 23-34, **1979**.

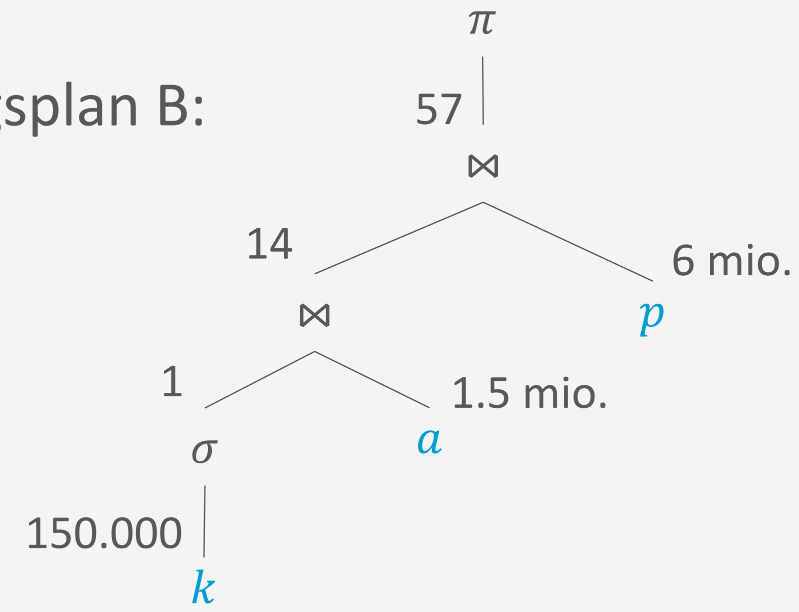
Kardinalitätsabschätzung: Beispiel

```
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.';
```

- Ausführungsplan A:



- Ausführungsplan B:



- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$

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.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$

- Abschätzung der Kardinalitäten der drei möglichen Operationen im ersten Schritt
  - $p \bowtie_{p.AuftragID = a.AuftragID} a$
    - $p.AuftragID$ Fremdschlüssel auf $a.AuftragID$
      $\rightarrow |p \bowtie_{...} a| = |p| = 6.000.000$
  - $a \bowtie_{a.KundenID = k.KundenID} k$
    - $a.KundenID$ Fremdschlüssel auf $k.KundenID$
      $\rightarrow |a \bowtie_{...} k| = |a| = 1.500.000$
  - $s = \sigma_{k.Name = "IBM\ Corp."}(k)$
    - Annahme, dass der Name eindeutig ist
      $\rightarrow$ 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.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$

1

σ

150.000 | k

- Abschätzung der Kardinalitäten der zwei möglichen nächsten Operationen
  - $p \bowtie_{p.AuftragID=a.AuftragID} a$
    - $|p \bowtie_{...} a| = |p| = 6.000.000$ (wie vorher)
  - $j = a \bowtie_{a.KundenID=s.KundenID} s$
    - $|j| = |a \bowtie_{...} s| = \frac{|a| \cdot |s|}{V(KundenID, s)} = \frac{1.500.000 \cdot 1}{1}$
      - Bzw. immer noch Fremd- auf Primärschlüssel
    - Als Selektion auffassen, da $|s| = 1$:

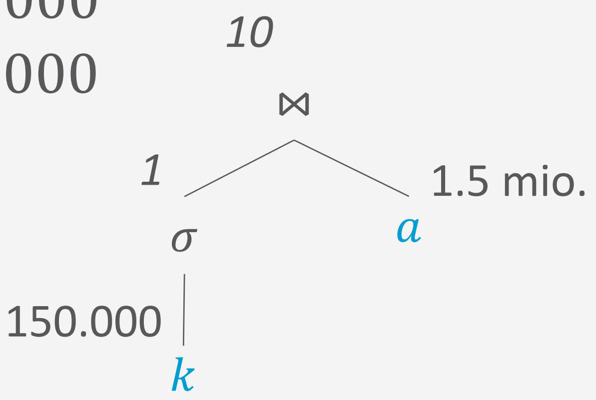
$$|a| \cdot \mathrm{sel}(KundenID = id) = |a| \cdot \frac{1}{V(KundenID, a)}$$

    - Annahme $V(KundenID, a) = 150.000$, da $|k| = 150.000$, dann $|j| = \frac{1.500.000 \cdot 1}{150.000} = 10$

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.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$



- Abschätzung der Kardinalitäten der einen möglichen nächsten Operation
  - Nicht so wichtig, weil letzte Operation
  - $p \bowtie_{p.AuftragID=j.AuftragID} j$
  - $|p \bowtie_{...} j| = \frac{|p| \cdot |j|}{V(AuftragID, j)} = \frac{6.000.000 \cdot 10}{10}$
    - Bzw. immer noch Fremd- auf Primärschlüssel
  - Als zehnfache Selektion auffassbar ($|j| = 10$):

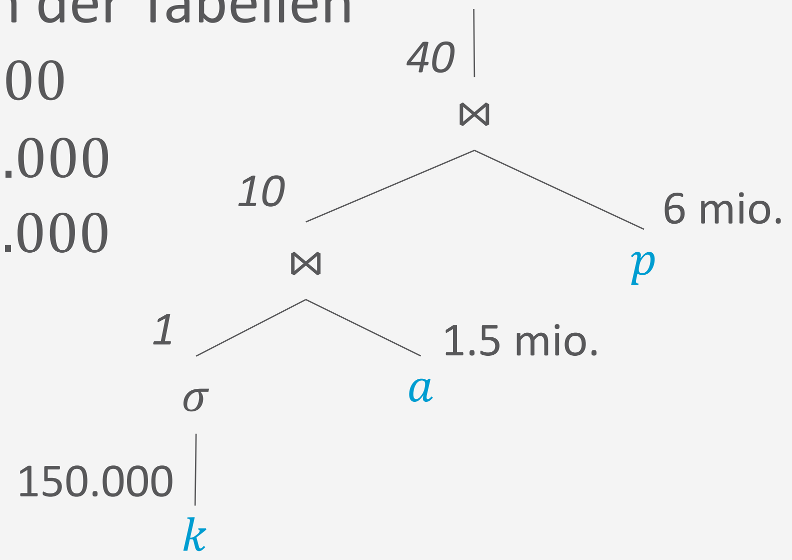
$$|j| \cdot \mathrm{sel}(AuftragID = id) = |j| \cdot \frac{|p|}{V(AuftragID, p)}$$

  - Unter Annahme der Gleichverteilung:
    - $\frac{|p|}{V(AuftragID,p)} = \frac{6.000.000}{1.500.000} = 4$ Posten / Auftrag
    - Dann $|p \bowtie_{...} j| = 10 \cdot 4 = 40$

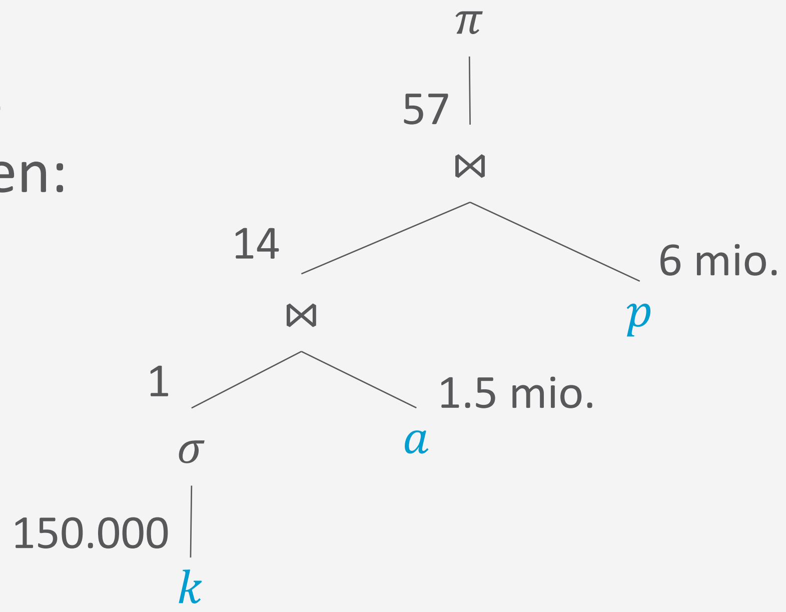
Kardinalitätsabschätzung: Beispiel

```
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.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$

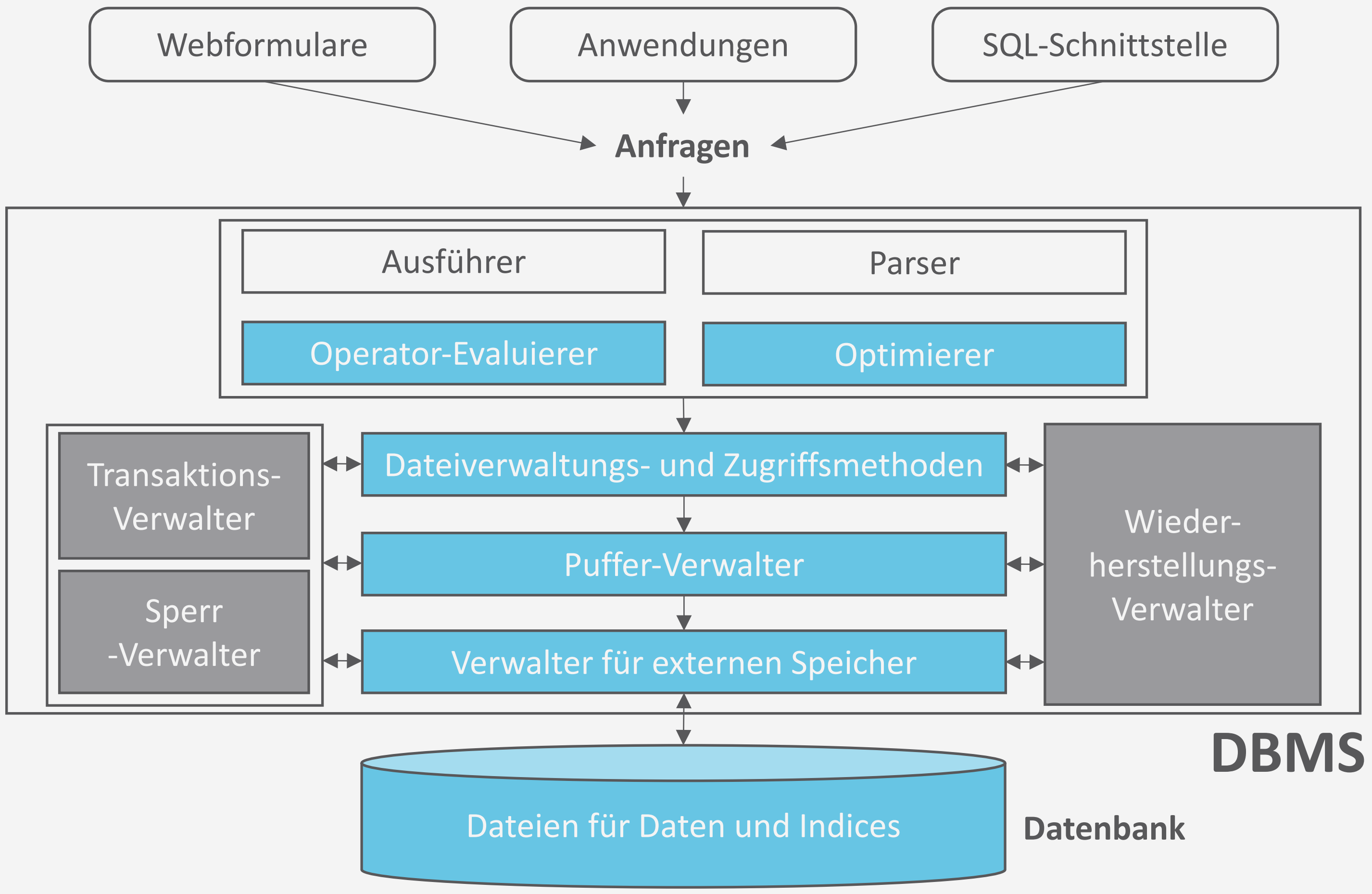


- Tatsächliche Kardinalitäten:



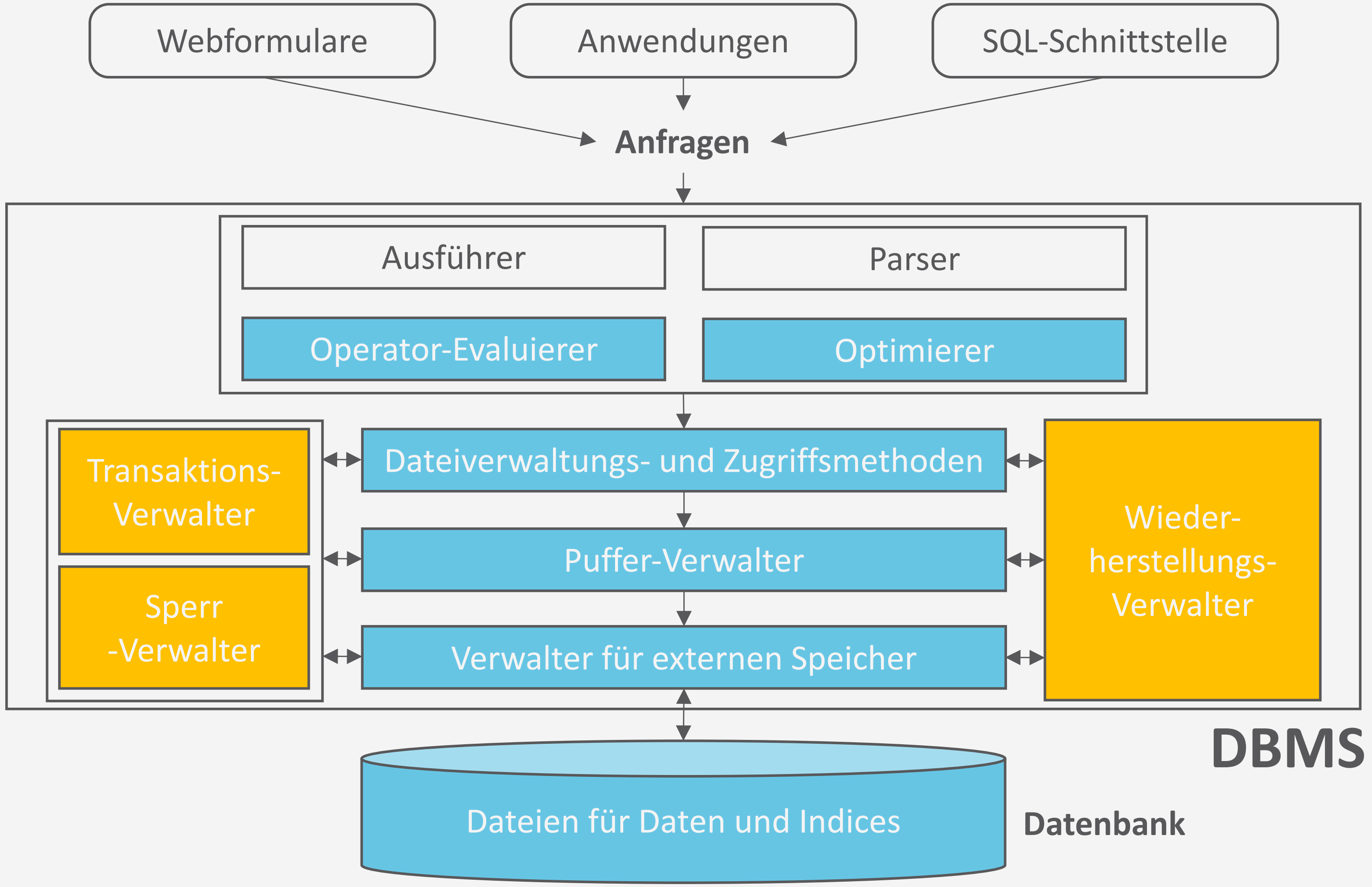
Architektur eines DBMS

- Speicherung
- Anfragebeantwortung
  - Operator-Evaluierer
  - Optimierer
    - Datenunabhängige Optimierung
      - Prädikatsvereinfachung
      - Anfrageentschachtelung
    - Datenabhängige Optimierung
      - Kardinalitätsabschätzung
      - Join-Reihenfolgen
- Transaktionsmanagement



Architektur eines DBMS

- Speicherung
- Anfragebeantwortung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



Überblick: 6. Anfrageverarbeitung

A. *Speicherung*
   - Speichermedien
   - Verwaltung
   - Puffer
   - Zugriff

B. *Indexierung*
   - ISAM-Index
   - B^+-Bäume (B*-Bäume)
   - Hash-basierte Indexe

C. *Anfragebeantwortung*
   - Sortieren
   - Join-Verarbeitung
   - Weitere Operationen
   - Pipelining

D. *Anfrageoptimierung*
   - Rewriting
   - Datenabhängige Optimierung

→ Transaktionen