5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Regelbasierte Anfrageoptimierung
   - Kostenbasierte Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

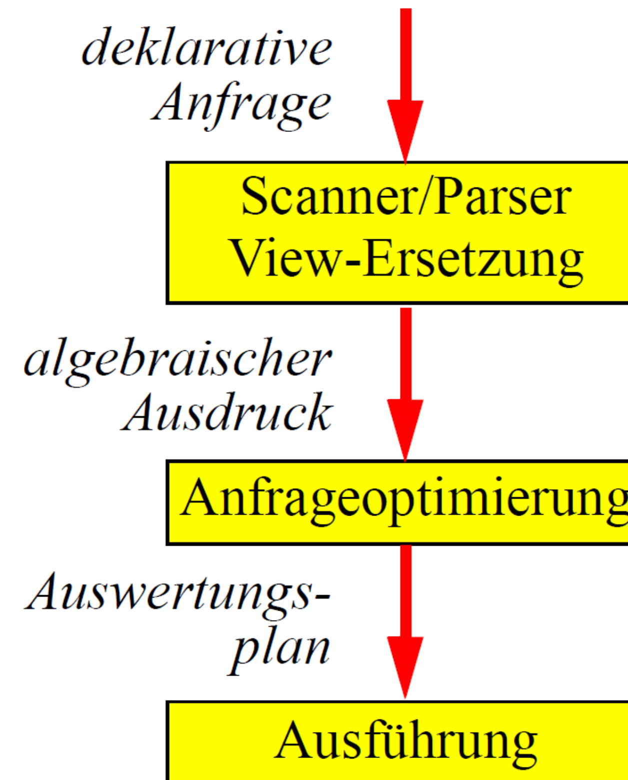
Informatik 5 -
Informationssysteme
und Datenbanken
RWTH AACHEN UNIVERSITY

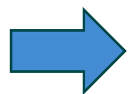
5. Relationale Anfragebearbeitung

1. **Anfragebearbeitung und -optimierung**
    - **Einführung**
    - Regelbasierte Anfrageoptimierung
    - Kostenbasierte Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
    - Indexstrukturen für eindimensionale Daten
    - Indexstrukturen für mehrdimensionale Daten

Anfragebearbeitung und -optimierung: Einführung

- Datenbankanfragesprachen wie SQL sind *deklarativ*
- Ein Ausdruck der Sprache beschreibt, welche Eigenschaften die Tupel der Ergebnisrelation haben müssen, ohne dafür eine Berechnungsprozedur anzugeben (**WAS**).
- Um diese Ausdrücke auswerten zu können, ist es erforderlich, die Operationen zu bestimmen, mit deren Hilfe das Ergebnis der Anfrage berechnet werden kann (**WIE**).
- Hierzu eignet sich eine *prozedurale* Sprache wie z.B. die *relationale Algebra*.





Zentrale Aufgabe der Anfragebearbeitung ist die Übersetzung der *deklarativen Anfrage* in einen *effizienten, prozeduralen Auswertungsplan.*

Übersetzen von Anfragen

- Anfragesprachen definiert durch *kontextfreie Grammatiken*, *Syntax-Diagramme* o.ä.
- Lexikalische und syntaktische Analyse wie bei Programmiersprachen

  Scanner und Parser können mit Compilerbau-Tools (z.B. LEX/YACC) erzeugt werden.
- Zusätzlich werden Views durch den Rumpf ihrer Definition ersetzt.
- Das Ergebnis der ersten Übersetzungsphase ist ein *Auswertungsplan* (engl. *query evaluation plan, QEP*) der relationalen Algebra in einer kanonischen Form.

Kanonischer Auswertungsplan zu einer SQL-Anfrage

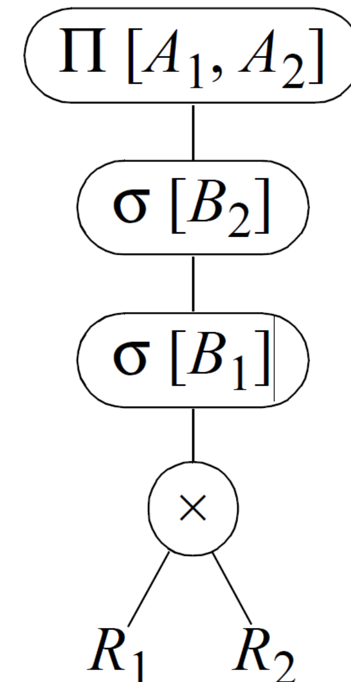
Allgemeine Form einer einfachen SQL-Anfrage (ohne Subqueries usw.):

select *A*1, *A*2, ...

from *R*1, *R*2, ...

where *B*1 **and** *B*2, ...

1. Bilde das kartesische Produkt der Relationen *R*1, *R*2, ...
2. Führe Selektionen mit den einzelnen Bedingungen *B*1, *B*2, ... durch.
3. Projiziere die Ergebnistupel auf die erforderlichen Attribute *A*1, *A*2, ...



Beispieldatenbank

Kunden

KNr	Name	Adresse	Region	Saldo
201	Klein	Lilienthal	Bremen	200.000
337	Horn	Dieburg	Rhein-Main	100.000
444	Berger	München	München	300.000
108	Weiss	Würzburg	Unterfranken	500.000

Bestellt

BNr	Datum	KNr	PNr
221	10.5.2002	201	12
312	11.5.2002	201	4
401	20.5.2002	337	330
456	13.5.2002	444	330
458	14.5.2002	444	98

PNr	Bezeichnung	Anzahl	Preis
12	BMW 318i	10	40.000
4	Golf3-90	40	25.000
330	Fiat Uno	5	18.000
98	Ferrari 380	1	180.000
14	Opel Corsa	14	17.000

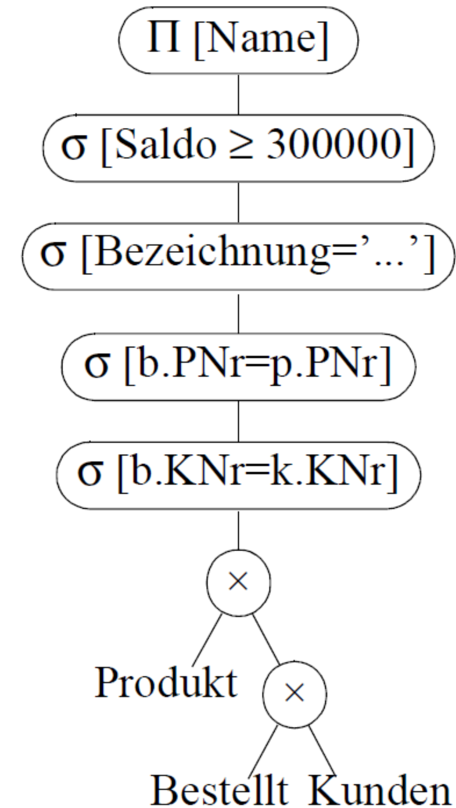
Beispiel: SQL-Anfrage

- Anfrage: *Welche Kunden (Name) haben einen Saldo von mindestens 300.000 und einen Fiat Uno bestellt?*

- SQL:

  select Name **from** Kunden k, Bestellt b, Produkt p
  where b.KNr = k.KNr
  and b.PNr = p.PNr
  and Bezeichnung = 'Fiat Uno'
  and Saldo ≥ 300000

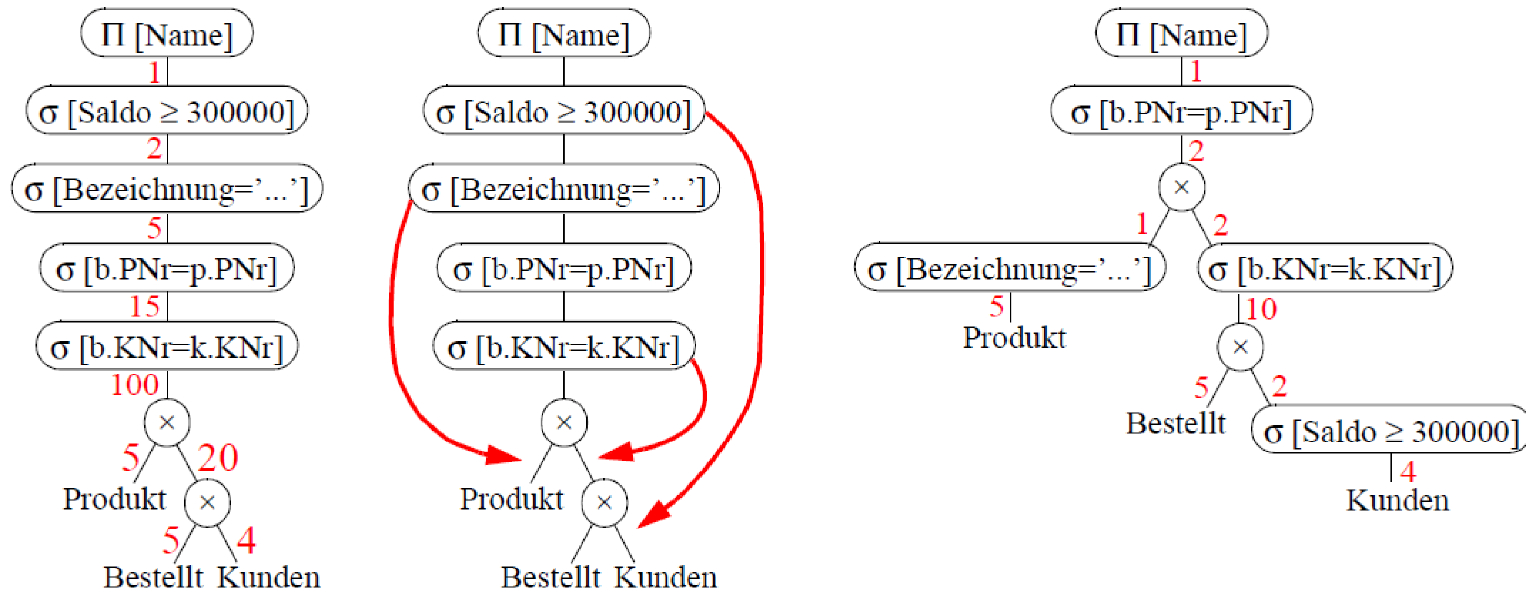
- Übersetzung in die rel. Algebra (kanonisch):



Beobachtungen

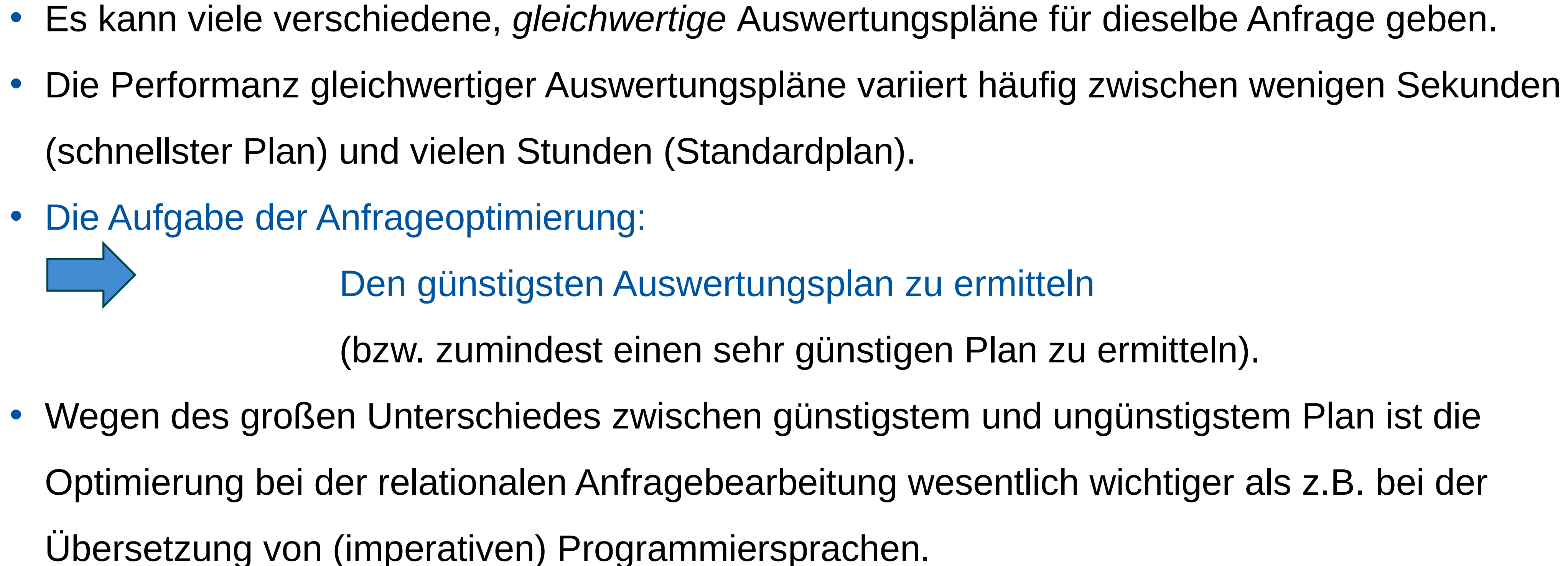
- Der kanonische Auswertungsplan erzeugt das kartesische Produkt der 3 Relationen
- Die Kardinalität des kartesischen Produktes ist |Kunden|·|Bestellt|·|Produkt| = 100 Tupel
- Für jedes der 100 Tupel muss z.B. die Bedingung b.KNr = k.KNr ausgewertet werden

Günstiger wäre es z.B., wenn man sich gleich von Anfang an auf das Produkt ’Fiat Uno’ und die Kunden mit hohem Saldo beschränken würde:



Beobachtungen

- Es kann viele verschiedene, *gleichwertige* Auswertungspläne für dieselbe Anfrage geben.
- Die Performanz gleichwertiger Auswertungspläne variiert häufig zwischen wenigen Sekunden (schnellster Plan) und vielen Stunden (Standardplan).
- Die Aufgabe der Anfrageoptimierung:



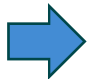
Den günstigsten Auswertungsplan zu ermitteln

(bzw. zumindest einen sehr günstigen Plan zu ermitteln).

- Wegen des großen Unterschiedes zwischen günstigstem und ungünstigstem Plan ist die Optimierung bei der relationalen Anfragebearbeitung wesentlich wichtiger als z.B. bei der Übersetzung von (imperativen) Programmiersprachen.

Logische und physische Anfrageoptimierung

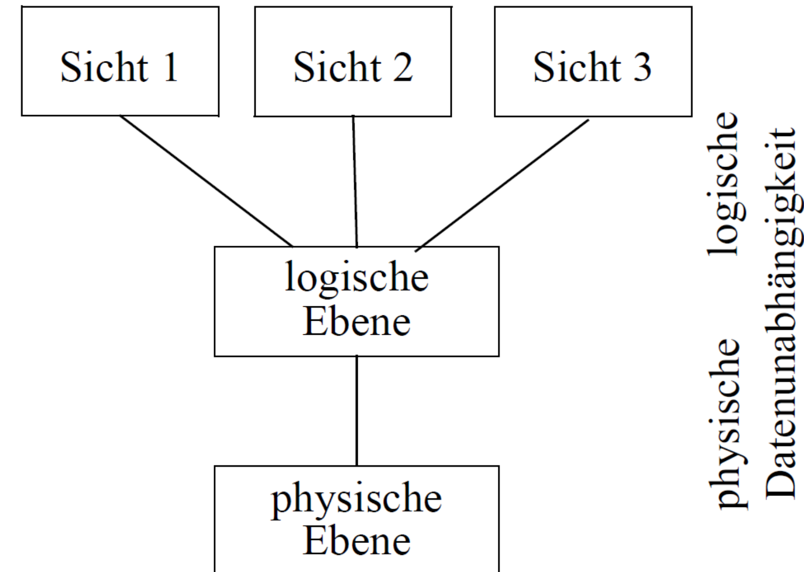
- Die in dem Beispiel angewandte Heuristik, Selektionen möglichst frühzeitig durchzuführen, wird als *push selection* bezeichnet. Weitere wichtige Optimierungen betreffen
  - die Erkennung der Join-Operation aus kartesischem Produkt und Selektion sowie deren Zusammenfassung
  - die Reihenfolge von Join-Operationen bzw. kartesischen Produkten
  - das Erkennen von widersprüchlichen (d.h. leeren) oder redundanten Teilen (gleichen Teilbäumen) in Auswertungsplänen (die nur einmal ausgewertet werden müssen)
- *Logische (algebraische) Anfrageoptimierung :* Optimierungstechniken, die den Auswertungs-plan betrachten und "umbauen" (d.h. die Reihenfolge der Operatoren verändern).
- *Physische Anfrageoptimierung*: Die Auswahl einer geeigneten Auswertungsstrategie für die Join-Operation oder die Entscheidung, ob für eine Selektionsoperation ein Index verwendet wird oder nicht und wenn ja, welcher (bei unterschiedlichen Alternativen).



die Auswahl eines geeigneten Algorithmus für jede Operation im Auswertungsplan

Weitere Gründe für die Optimierung

- Das Prinzip der physischen Datenunabhängigkeit besagt, dass die Benutzer des Datenbanksystems von der physischen Organisation der Daten abgeschirmt sein sollen.
- Der Benutzer braucht eine Anfrage nicht selbst zu optimieren.
- Indexe können zur Leistungssteigerung vom DB-Administrator angelegt werden.
- Die Änderungen sind für Anwendungsprogramme und adhoc-Anfragen transparent.



Zwei Arten von Optimierungen

- Regelbasierte Optimierung
  - Ausnutzung von Gleichungen der relationalen Algebra
  - Heuristiken auf Basis der Schemainformation
  - Keine Betrachtung der betroffenen Daten

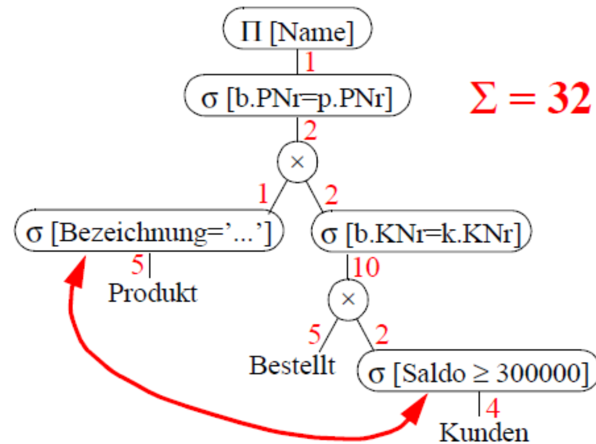
- Kostenbasierte Optimierung
  - Berücksichtigung der Daten
  - Verwendung von Statistiken (z.B. Histogramme)
  - Schätzung der Kosten von Auswertungsplänen

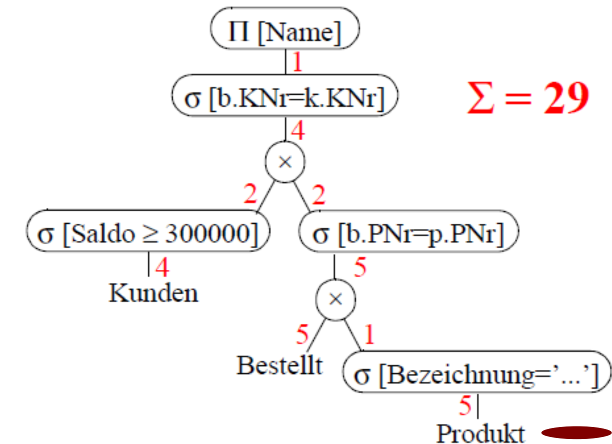
5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - **Regelbasierte Anfrageoptimierung**
   - Kostenbasierte Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Regelbasierte Anfrageoptimierung

- Es gibt zahlreiche Regeln (Heuristiken), um die Reihenfolge der Operatoren im Auswertungsplan zu modifizieren und eine Performanz-Verbesserung zu erreichen, z.B.:
  - Push Selection: Führe Selektionen möglichst frühzeitig (vor Joins) aus
  - Elimination leerer Teilbäume
  - Erkennen gemeinsamer Teilbäume
- Optimierer, die sich ausschließlich nach diesen starren Regeln richten, nennt man *regelbasierte* oder auch *algebraische Optimierer*.
- Die Performanz von Auswertungsplänen hängt allerdings auch ganz wesentlich von der Datenverteilung der gespeicherten Informationen ab (siehe nächstes Teilkapitel).
- Beispiel Join-Reihenfolge:





Äquivalenzregeln

Für die relationale Algebra gelten zahlreiche Äquivalenzregeln, die eine Quelle potentieller Optimierungsregeln sind.

- Join, Vereinigung, Schnitt und Kreuzprodukt sind kommutativ:
- Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ:
- Selektionen sind untereinander vertauschbar:

Äquivalenzregeln

- Konjunktionen in einer Selektionsbedingung können in mehrere Selektionen aufgebrochen werden, bzw. nacheinander ausgeführte Selektionen können zu einer konjunktiven Selektion zusammengefasst werden:

- Geschachtelte Projektionen können eliminiert werden:

  ⇨ Nur sinnvoll, falls für die jeweiligen Attributmengen:
- Selektion und Projektion sind vertauschbar, falls die Projektion keine Attribute der Selektionsbedingung entfernt:

  falls
- Selektion und Join (Kreuzprodukt) können vertauscht werden, falls die Selektion nur Attribute eines der beiden Join-Argumente verwendet:

  falls

Äquivalenzregeln

- Projektionen können teilweise in den Join verschoben werden:

- Selektionen können mit Vereinigung, Schnitt und Differenz vertauscht werden:

- Der Projektionsoperator kann mit der Vereinigung, aber nicht mit Schnitt oder Differenz vertauscht werden:

- Eine Selektion und ein Kreuzprodukt können zu einem Join zusammengefasst werden, wenn die Selektionsbedingung eine Joinbedingung ist (z.B. Equijoin):

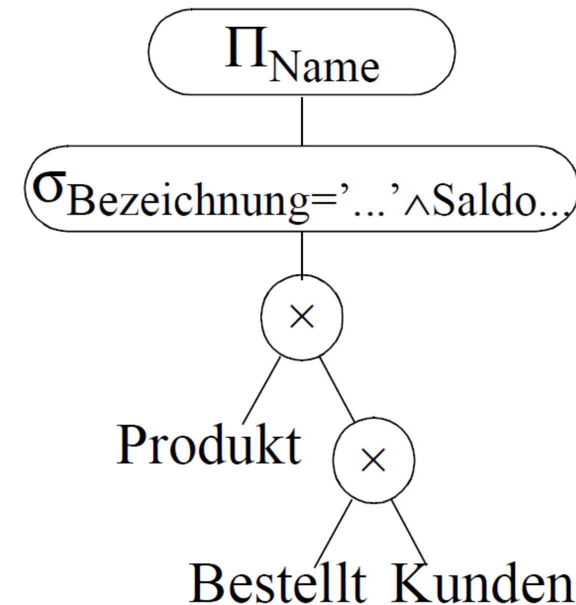
- Auch an Bedingungen können Veränderungen vorgenommen werden:
  Kommutativgesetze, Assoziativgesetze, z.B.:
  Distributivgesetze, z.B.:
  De Morgan:

Restrukturierungsalgorithmus: Anwendung von Regeln

- Aufbrechen der Selektionen
- Verschieben der Selektionen soweit wie möglich nach unten im Operatorbaum
- Zusammenfassen von Selektionen und Kreuzprodukten zu Joins
- Einfügen und Verschieben von Projektionen soweit wie möglich nach unten
- Zusammenfassen einzelner Selektionen zu komplexen Selektionen

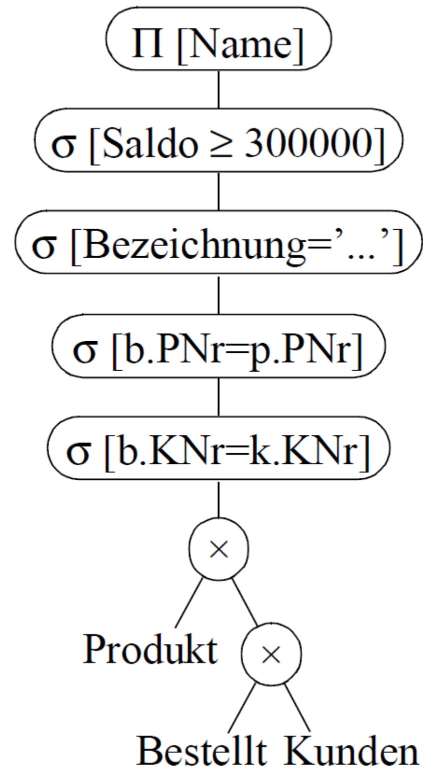
Beispiel:

select	Name
from	Kunden k, Bestellt b, Produkt p
where	b.KNr = k.KNr
and	b.PNr = p.PNr
and	Bezeichnung = 'Fiat Uno'
and	Saldo ≥ 300000

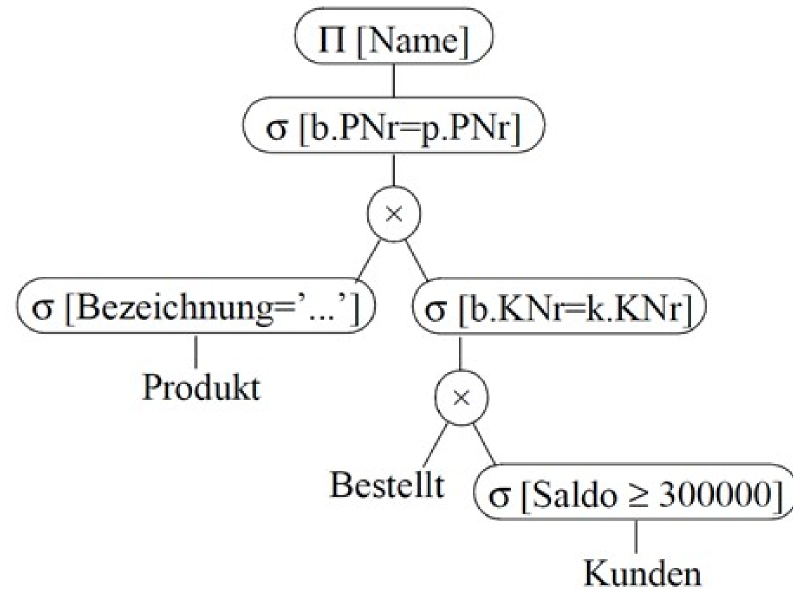


Restrukturierungsalgorithmus: Beispiel

1. Aufbrechen der Selektionen

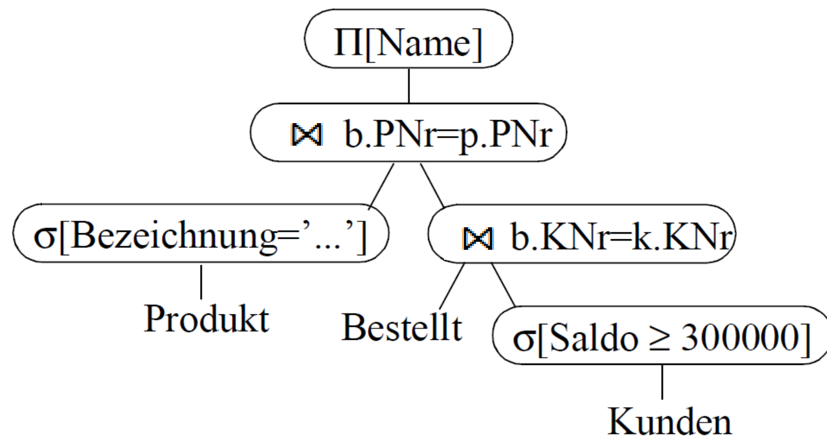


2. Verschieben der Selektionen

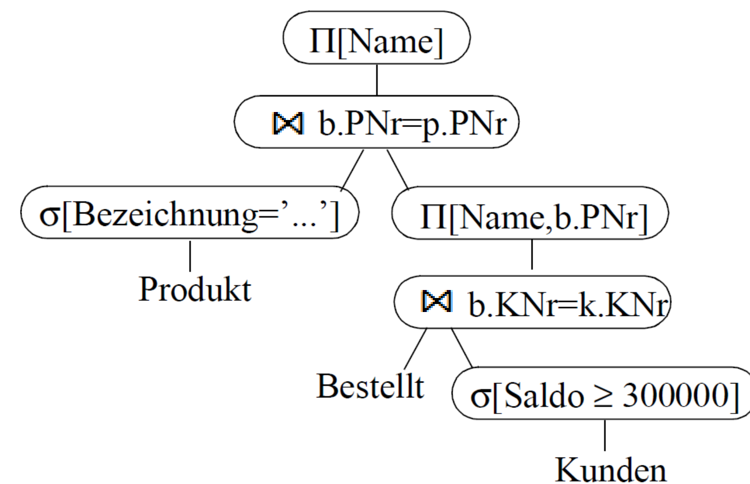


Restrukturierungsalgorithmus Beispiel

3. Zusammenfassen zu Joins



4. Einfügen und Verschieben von Projektionen



5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Regelbasierte Anfrageoptimierung
   - **Kostenbasierte Anfrageoptimierung**
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Kostenbasierte Optimierung

- Die Position von Selektions- und Projektionsoperationen im Query-Execution-Plan (QEP) wird durch die regelbasierte Optimierung bereits zufriedenstellend bestimmt.
- Die Reihenfolge der Join-Operationen sowie Operationen wie GROUP BY und neue Anfragearten (*top-n-queries*) kann durch den regelbasierten Ansatz nicht optimiert werden.
- Die kostenmodellbasierte Anfrageoptimierung behandelt deshalb insbesondere die Reihenfolge der Join-Operationen.
- Zwei interessante Fragestellungen:
  - Schätzung der Kosten, die die Auswertung eines bestimmten QEP verursachen würde, insbes. die Schätzung der Größe von Zwischenergebnissen
  - Generierung verschiedener Join-Reihenfolgen bei großen Mengen von Ausgangstabellen (die Anzahl verschiedener QEP ist exponentiell in der Anzahl der Tabellen)

Kostenbasierte Optimierung und Selektivität

- Ziel: Ergebnisse innerhalb kurzer Laufzeit liefern, d.h. (nahezu) optimale QEP
- Ein Kostenmodell ist notwendig, um den besten Auswertungsplan auszuwählen
- Ein Kostenmodell stellt Funktionen zur Verfügung, die den Aufwand, d.h. die Laufzeit, der Operationen der physischen Algebra abschätzen.
- Bei der Aufwandsbestimmung spielt in vielen Fällen eine Rolle, wie viele Tupel sich bei Auswertung einer Bedingung qualifizieren.
- Der Anteil der qualifizierenden Tupel heißt Selektivität (sel)
  - Die Selektivität ist kein Kostenmaß.
  - Die Laufzeit einer Operation hängt von der Eingabegröße ab, und damit von der Selektivität der im Operatorbaum darunter liegenden Operationen.

Schätzung der Zwischenergebnisse

- Definitionen für *Selektivität* der wichtigsten Algebraoperationen:

  Selektion mit Bedingung :

  Join von und :

  Aber die Werte der Zähler in diesen Brüchen müssen statistisch geschätzt werden, außer in den folgenden seltenen Spezialfällen!
- Die Selektivität von , also Vergleich mit einer Konstante beträgt , falls ein *Schlüssel* ist.
- Falls kein Schlüssel ist, aber Werte *gleichverteilt* sind,

  (: die Anzahl der unterschiedlichen Attributwerte)
- Besitzt bei einem Equijoin das Attribut Schlüsseleigenschaft, kann die Größe des Join-Ergebnisses mit abgeschätzt werden, da jedes Tupel aus maximal einen (genau einen bei *referential integrity*) Joinpartner findet.

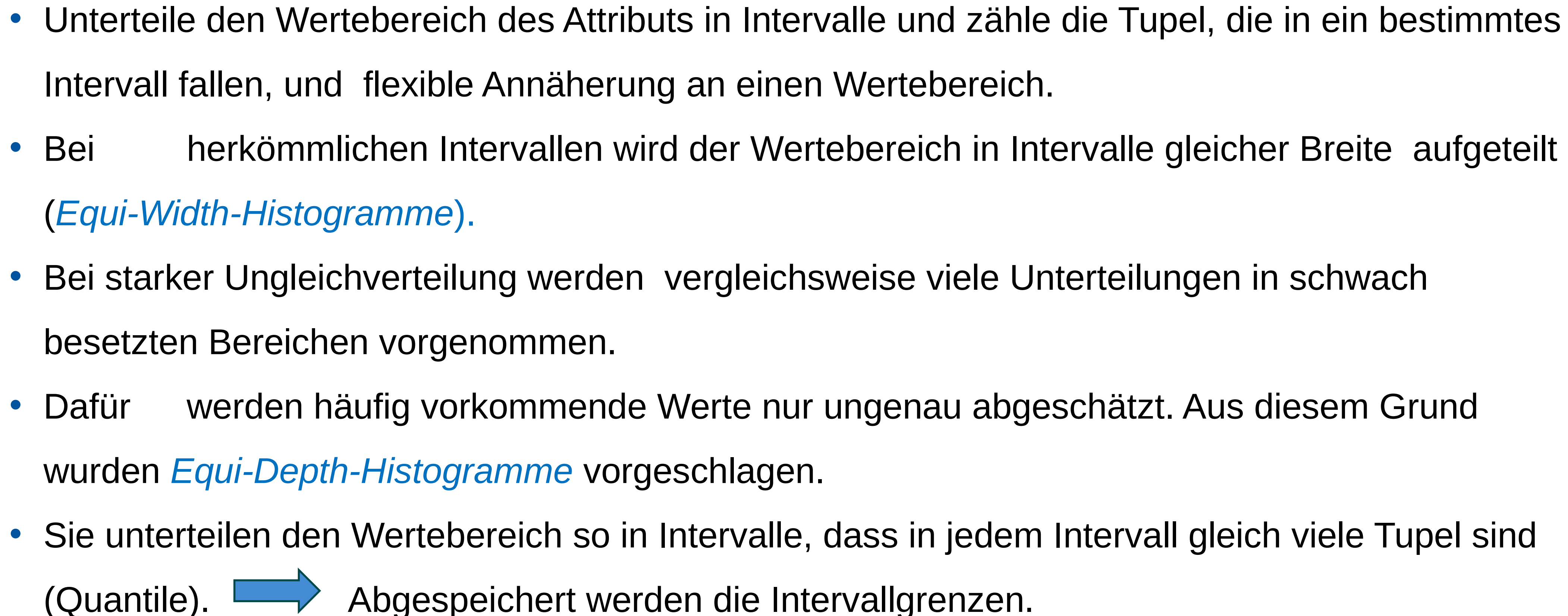
Schätzung der Zwischenergebnisse

- Bei der Verknüpfung von Selektionsbedingungen mit booleschen Operatoren lassen sich einfache Rechenregeln angeben, wenn die stochastische Unabhängigkeit der qualifizierenden Mengen angenommen werden kann, z.B.
  - logisches UND :
  - logisches ODER :
  - logisches NICHT :
- Nur in Spezialfällen kann man solche einfachen Rechenregeln anwenden.
- Im allgemeinen braucht man anspruchsvollere Methoden zur Selektivitätsabschätzung:

1) Parametrisierte Verteilungen:

- Bestimme zu der vorhandenen Werteverteilung die Parameter einer Funktion so, dass die Verteilung möglichst gut angenähert wird.
- Beispiel für eine solche Funktion: Die Normalverteilung mit den Parametern (Mittelwert) und (Varianz).

Schätzung der Zwischenergebnisse: Histogramme



2) Histogramme

- Unterteile den Wertebereich des Attributs in Intervalle und zähle die Tupel, die in ein bestimmtes Intervall fallen, und flexible Annäherung an einen Wertebereich.
- Bei herkömmlichen Intervallen wird der Wertebereich in Intervalle gleicher Breite aufgeteilt (*Equi-Width-Histogramme*).
- Bei starker Ungleichverteilung werden vergleichsweise viele Unterteilungen in schwach besetzten Bereichen vorgenommen.
- Dafür werden häufig vorkommende Werte nur ungenau abgeschätzt. Aus diesem Grund wurden *Equi-Depth-Histogramme* vorgeschlagen.
- Sie unterteilen den Wertebereich so in Intervalle, dass in jedem Intervall gleich viele Tupel sind (Quantile). → Abgespeichert werden die Intervallgrenzen.

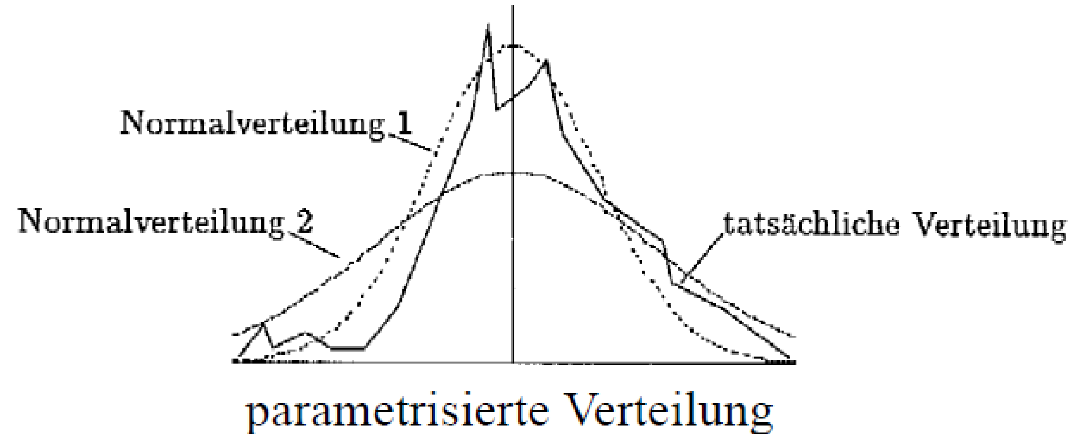
Schätzung der Zwischenergebnisse: Histogramme

Histogramme (weiter)

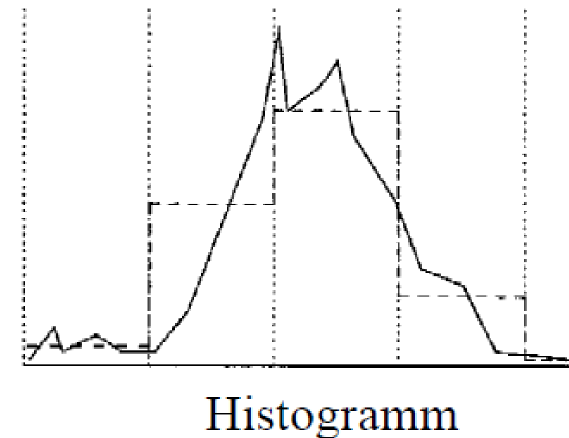
- Equi-Depth-Histogramme weisen eine bessere Schätzgenauigkeit auf, haben aber auch einen höheren Verwaltungsaufwand. ORACLE benutzt Equi-Depth-Histogramme:

  ANALYZE TABLE *name* ESTIMATE STATISTICS ;

- Die Histogramme werden in speziellen Tabellen des Data Dictionary abgelegt.
- Kein automatisches Update der Statistiken.



parametrisierte Verteilung


Histogramm

Schätzung der Zwischenergebnisse

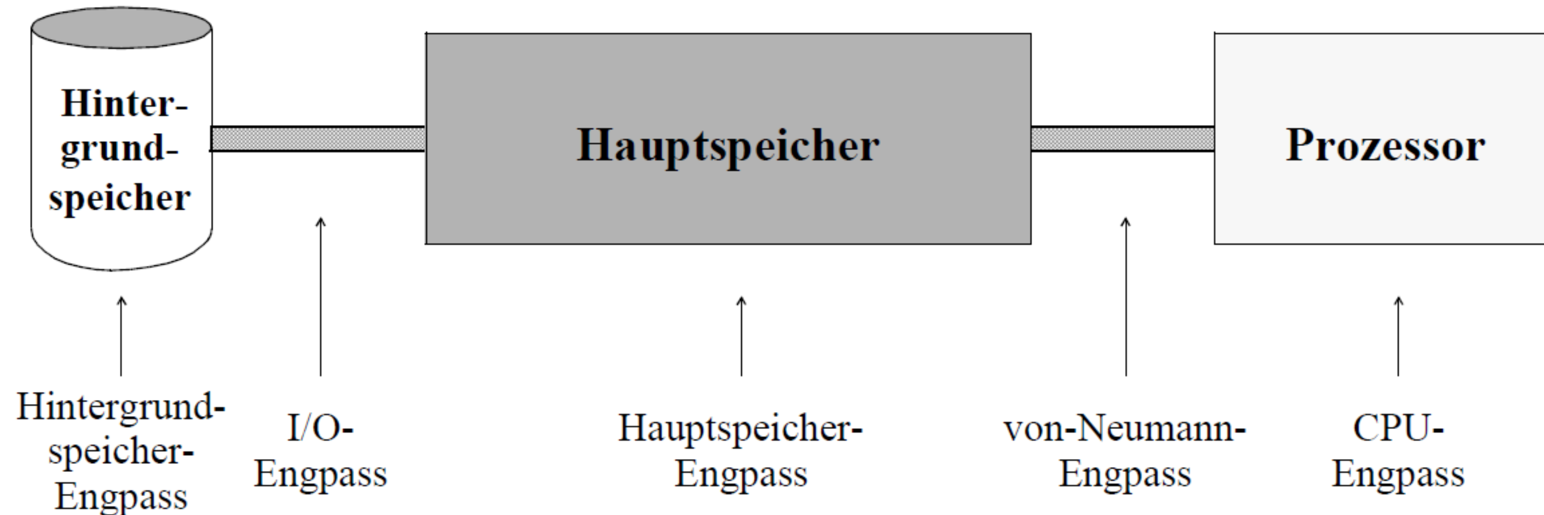
3) Stichproben:

- Außerordentlich einfach.
- Eine zufällige Menge von Tupeln einer Relation wird gezogen und als repräsentativ für die gesamte Relation betrachtet.
- Selektivitäten werden auf der Basis dieser Stichproben bestimmt.

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Regelbasierte Anfrageoptimierung
   - Kostenbasierte Anfrageoptimierung
2. **Algorithmen für Basisoperationen**
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Engpassprobleme



- Zur Vereinfachung unterscheidet man bei (sequentiellen) Algorithmen für Grundoperatoren der rel. Algebra meist nicht zwischen den 5 Engpässen sondern nur zwischen
  - *CPU-bound*: Das System aus CPU, Arbeitsspeicher und Bus bildet den Hauptengpass
  - *I/O-bound*: Hintergrundspeicher und I/O bilden den Hauptengpass
- Auch bei nicht-parallelen Datenbanksystemen wird immer die Parallelität zwischen CPU und Hintergrundspeicher ausgenutzt. Deshalb sind Algorithmen zur Anfragebearbeitung häufig mit Hilfe mehrerer Prozesse oder "multithreaded" implementiert.

Optimierungsziele

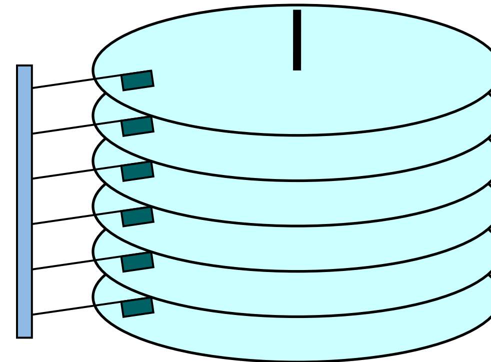
Optimierung auf verschiedenen Ebenen:

- Reduzieren der I/O-Kosten durch
  - gute Ausnutzung eines Puffers
  - Verwendung von Indexen
  - durch vorberechnete Joins (Cluster)
- Reduzieren der Vergleiche (CPU-Kosten, kann insbesondere bei komplexen Datentypen sehr wichtig sein)
- Reduzieren der Kommunikationskosten (besonders wichtig in verteilten DBS)
- Verbesserung der Abarbeitungsreihenfolge in einem Mehrwege-Join

Verwendung von Hintergrundspeicher („Sekundärspeicher“)

- Motivation
  - Persistente Speicherung von Daten: „Daten überleben Prozesse“
  - Datenmengen sind größer als der Hauptspeicher: viele GB, TB, PB
  - Gemeinsame Nutzung von Daten: nebenläufiges Arbeiten

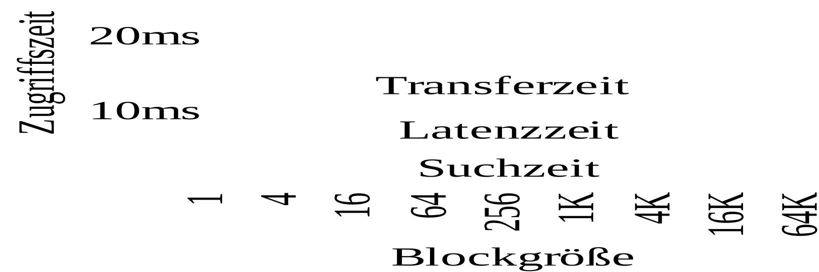
- Festplatten/SSD als gebräuchliche Sekundärspeicher
  - Platten: Übereinanderliegende Platten mit magnetischen / optischen Oberflächen
  - Zur Adressierung sind die Platten in Spuren und Sektoren eingeteilt
  - Mechanische Bewegungen
    - Platten rotieren um gemeinsame Achse
    - Schreib-Lese-Köpfe werden zwischen den Platten synchron in radialer Richtung bewegt



Blockweiser Zugriff auf Festplatten

- Zugriffszeit bei Festplatten
  - Armpositionierung: Suchzeit (5 ms)
  - Rotation bis Blockanfang: Latenzzeit (5 ms)
  - Datenübertragung: Transferzeit (ms/MB)

- Blockorientierter Zugriff
  - Größere Transfereinheiten (Blöcke, Seiten) sind günstiger als einzelne Bytes
  - Gebräuchliche Seitengrößen: 2kB oder 4kB



Algorithmen für Basisoperationen: Selektion und Projektion

Selektion

- Sequentieller Scan oder Verwendung von Indexstrukturen
- Auswahl hängt unter anderem vom Anfragetyp ab
  - Punktanfragen („exact match query“)
    - SELECT * FROM Stud WHERE Matrnr = 123456
  - Bereichsanfragen („range query“)
    - SELECT * FROM Stud WHERE 123456 <= Matrnr AND Matrnr <= 123465

Projektion

- Teiloperationen:

  Projektion auf die Projektionsattribute & Eliminierung von Duplikaten
- Aufwendigere Teiloperation: Eliminierung von Duplikaten
  - ➢ Projektion durch Sortieren
  - ➢ Projektion durch Hashverfahren

Algorithmen für Basisoperationen: Join

Join

- Wichtigste Operation, insbesondere in relationalen DBS:
  - komplexe Benutzeranfragen
  - Normalisierung der Relationen
  - verschiedene Sichten (“views”) auf die Basisrelationen

Beispiele von Join Algorithmen:

1. *Nested Loop Join*:
   - erzeuge alle Tupel des kartesischen Produktes und prüfe die Join-Bedingung
2. *Nested Block Loop Join*
   - *Berücksichtigt die Block-Struktur des verwendeten Speichers*
3. *Indexed Loop Join*:
   - betrachte alle Tupel der einen Relation und greife auf die Joinpartner über einen passenden Index der anderen Relation zu
4. *Hash-Join:*
   - *Join-Partner eines Tupels wird mit Hilfe eines Hash-Verfahrens gesucht*
5. *Sort Merge Join*:
   - sortiere beide Relationen nach dem Joinattribut und filtere passende Paare

Es gibt i.A. kein *bester* Join-Algorithmus! Es ist von der jeweiligen Situation abhängig (Datenverteilung, Existenz von Index, Anfrage usw.), welcher Algorithmus sinnvoll ist.

Annahmen

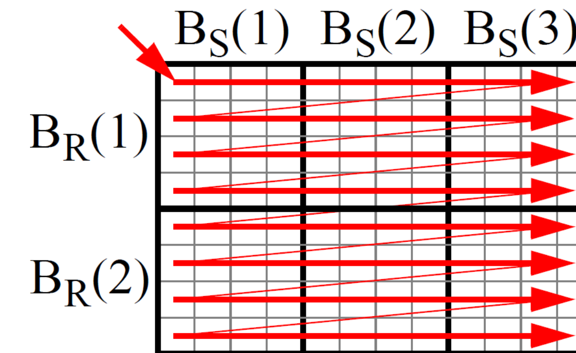
- Equijoin mit einem gemeinsamen Attribut A
- R und S sind zwei Relationen mit einem gemeinsamen Attribut A (sonst Attribut umbenennen)
- Notation:
  - $\otimes$ sei ein Operator, der zwei Tupel zu einem Tupel verknüpft
  - Sei r ein Tupel und A ein Attribut. Dann ist r(A) der Attributwert von A und r-r(A) das Tupel r ohne den Attributwert von A.
  - Sei R Relation und A ein Attribut, dann ist R\A die Relation R ohne das Attribut A

Einfacher Nested Loop Join

for each Tupel in do

for each Tupel in do

if then

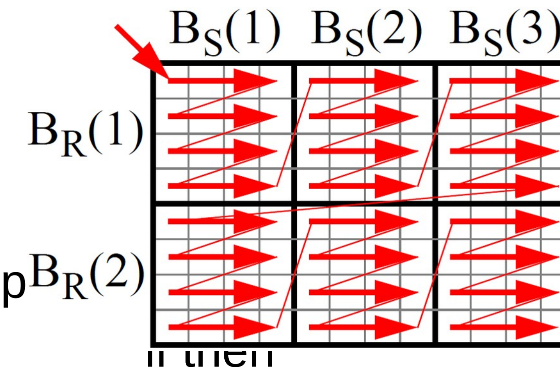


- Die Laufzeit ist
- Performanz ist deshalb manchmal inakzeptabel.
- Der einfache Nested Loop Join entspricht der Bildung des kartesischen Produktes in kanonischer Ordnung mit anschließender Selektion.
- S wird als innere Relation und R als äußere Relation bezeichnet
- Matrixdarstellung der Joinoperation (stellt Reihenfolge von Block- und Tupelpaarungen dar)
- Nested Loop Joins sind geeignet für alle Join-Prädikate (“, ”, “, ”, usw.)

Nested Block Loop Join

- Beobachtung: Die innere Relation S wird -mal gelesen (das kann teuer sein).
- Reduktion der I/O-Kosten durch blockweise Verarbeitung der Relationen

```
for each Block  in  do {
        Lade Block ;
        for each Block  in  do {
                Lade Block ;
                for each Tupel  in  do
                        for each Tup
                                if then
        }
}
```



Nested Block Loop Join: Beispiel

Relation S

Angesteller	Gehaltsgruppe	
Müller	1	$B_S(1)$
Schneider	2	
Schuster	1	$B_S(2)$
Schmidt	2	
Schütz	1	$B_S(3)$

Relation R

Gehaltsgruppe	Gehalt	
1	10.000	$B_R(1)$
2	20.000	
3	30.000	$B_R(2)$

- Blockzugriffe , wobei Anzahl der Blöcke der Relation ist
  - Im Beispiel: 8 Zugriffe
- Empfehlung: Die kleinere Relation sollte die äußere sein (ohne Cache).
- Wenn ein Cache (= Hauptspeicherplatz für viele Blöcke gleichzeitig) zur Verfügung steht, kann u.U. die kleinere Relation ganz im Hauptspeicher gehalten werden
  - Dann nur Zugriffe, im Beispiel: 5 Zugriffe

Index Join

- Grundidee: Ersetzt innere Schleife durch Suche in einer Indexstruktur
- Index-Join kann in folgendem Fall verwendet werden:
  - Equi-Join zwischen und bezüglich eines (ggf. nicht-eindeutigen) Attributs .
  - hat einen *Index* auf dem Attribut .
  - kann ohne Index gespeichert sein.
- Folgende Schleife berechnet den Join:

for each Tupel in do

find join partner in index on *S(A)*

- Kostenschätzung:
  - Also z.B. im Fall der Verwendung von B-Bäumen (siehe Kap. 5.3)

Einfacher Hash-Join

Weitere Idee: Erzeugung eines Index zur Laufzeit

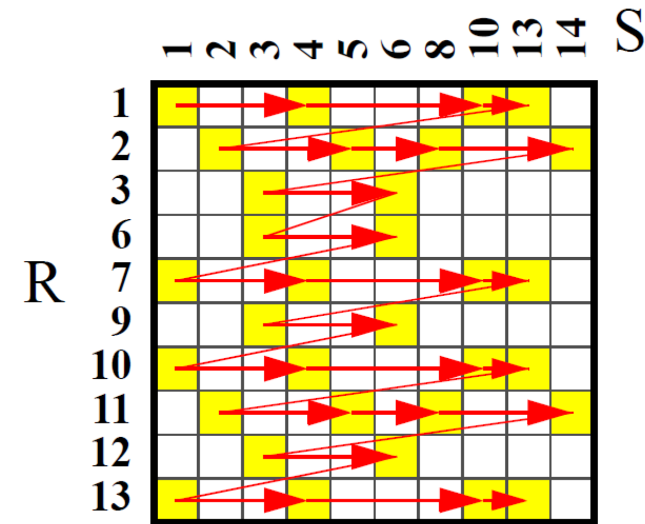
Reduktion des CPU-Aufwandes bei der Join-Berechnung:

- Der Join-Partner eines -Tupels wird gezielt mit Hilfe eines Hash-Verfahrens gesucht, statt sequentiell mit jedem Tupel der -Relation zu vergleichen
- Zu diesem Zweck wird die -Relation gehasht, d.h. zu allen Tupeln der Hash-Key bestimmt und die Tupel in einer Tabelle unter diesem Key eingetragen
- Nicht alle -Tupel, die den passenden Hash-Key haben, sind Join-Partner eines -Tupels, aber alle Join-Partner haben denselben Hashkey (**Bedingung der Hashfunktion!**)

Einfacher Hash-Join

```
for each Tupel  in  do {                /* Erzeugen der Hashtabelle  */
        berechne  = Hash;
        speichere das Tupel  in
        }
        for each Tupel  in  do {                /* Prüfen in der Hashtabelle  */
        berechne  Hash;
        for each Tupel  in  do {
                if then

        }
        }
```



$$h(x) = x \text{ MOD } 3$$

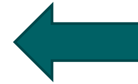
- Im Idealfall soll der Join im Hauptspeicher ablaufen:

  Hashtabelle wird für die kleinere Relation erzeugt.
- Hash-Join Verfahren können nur für Equi-Join und Natürlichen Join effizient genutzt werden.

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

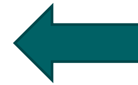
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

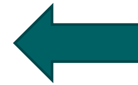
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**
1	**10.000**	**Schuster**

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Relation S

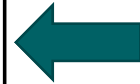
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**
1	**10.000**	**Schuster**

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

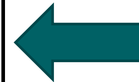
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**
1	**10.000**	**Schuster**

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

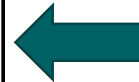
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**
1	**10.000**	**Schuster**
2	**20.000**	**Schneider**

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

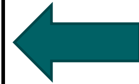
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**
1	**10.000**	**Schuster**
2	**20.000**	**Schneider**

Sort-Merge Join: Beispiel

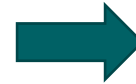
Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt



Gehaltsgruppe	Gehalt	Angestellter
1	**10.000**	**Müller**
1	**10.000**	**Schuster**
2	**20.000**	**Schneider**
2	**20.000**	**Schmidt**

Sort-Merge Join

1. Sort-Phase (falls Relationen nicht schon sortiert)
   - Sortiere Relation R bzgl. Attribut A;
   - Sortiere Relation S bzgl. Attribut A;

2. Merge-Phase: Paralleles Durchlaufen der Relationen

while
if
else if
else
return ;

Hilfsfunktion zur Bestimmung der nächsten Gruppe:

function

while

;

return

Sort-Merge Join

Analyse:

- Sortieren der Relationen kostet
- Sortieren ist nicht notwendig, wenn bereits Index existiert
- Wenn beide Relationen keine Duplikate in den Join-Attributen habe wird jede Relation genau einmal durchlaufen: Vergleiche
- Bei Duplikaten für A bestimmt das kartesische Produkt die worst case Performanz.
- Wenn nur eine der Relationen Duplikate in den Join-Attributen hat, kann das Verfahren optimiert werden: auch dann kann jede Relation genau einmal durchlaufen werden (Idee: Berechnung des kartesischen Produktes im gleichen Durchlauf wie Bestimmung der Gruppen)
- Weitere Optimierungen: Sortierung und Merging kombinieren, Blöcke berücksichtigen

2. Merge-Phase: Paralleles Durchlaufen der Relationen unter der Annahme: R enthält für A keine Duplikate!

```
while
        while
                    if

return ;
```

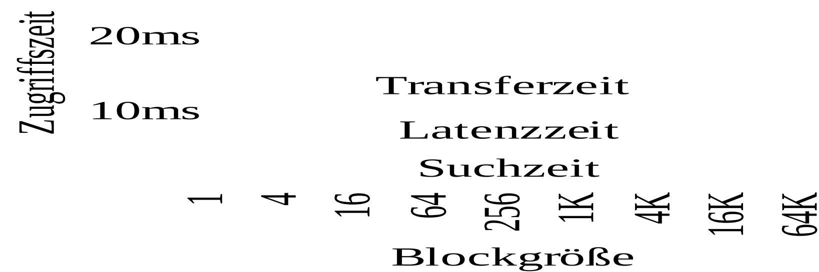
5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Regelbasierte Anfrageoptimierung
   - Kostenbasierte Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. **Indexstrukturen**
   - **Indexstrukturen für eindimensionale Daten**
   - Indexstrukturen für mehrdimensionale Daten

Blockweiser Zugriff auf Festplatten

- Zugriffszeit bei Festplatten
  - Armpositionierung: Suchzeit (□ 5 ms)
  - Rotation bis Blockanfang: Latenzzeit (□ 5 ms)
  - Datenübertragung: Transferzeit (ms/MB)

- Blockorientierter Zugriff
  - Größere Transfereinheiten (Blöcke, Seiten) sind günstiger als einzelne Bytes
  - Gebräuchliche Seitengrößen: 2kB oder 4kB



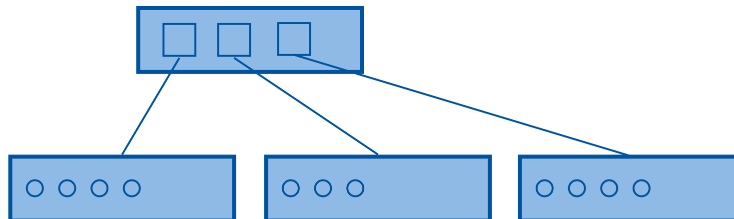
Verwendungsarten von Indexen

- Zielsetzung: Effiziente Unterstützung von Selektionen
- Beispiel: Primärindex
  - Die Tupel sind eindeutig durch einen Primärschlüssel oder einen Schlüsselkandidaten bestimmt.
  - Verwendung eines Clusterindex möglich, d.h. die Daten sind gemäß dem Schlüssel geordnet gespeichert minimale Such- und Latenzzeit auf Platten!
- Beispiel: Sekundärindex
  - Indexe dürfen auch über anderen Attributen angelegt werden.
  - In diesem Fall können in den Attributwerten auch Duplikate auftreten.
- Speicherhierarchie impliziert wichtige Nebenbedingungen:
  - Vorhersagbarer Suchaufwand: AVL-Binärbäume sinnlos balancierte Mehrwegbäume
  - Möglichst wenig I/O : Ausnutzen der Blockstruktur B-Baum-Familie als Standard

Blockweise Speicherung von Daten

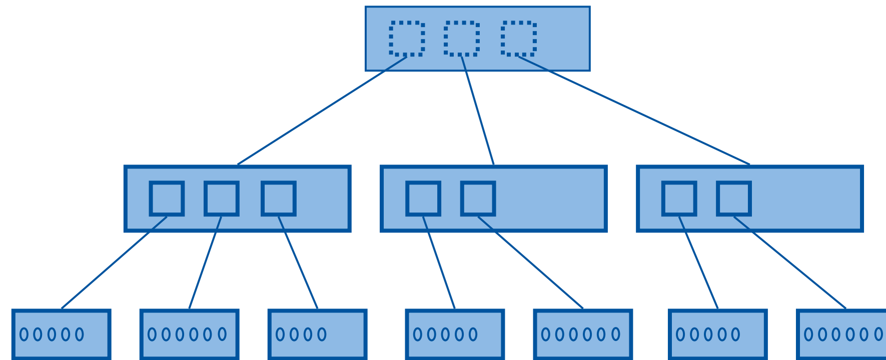
- Block- oder seitenweise Speicherung
  - Speicherung vieler Datensätze auf ein- und derselben Seite
  - Für effiziente Suche: Speichere ähnliche Werte auf derselben Seite
  - Bei Überlauf einer Seite: Aufspaltung auf mehrere Seiten

- Hierarchische Organisation
  - Übergeordnete Knoten zur Erschließung der Datenblöcke (Directory)



Mehrstufige Hierarchien (Bäume)

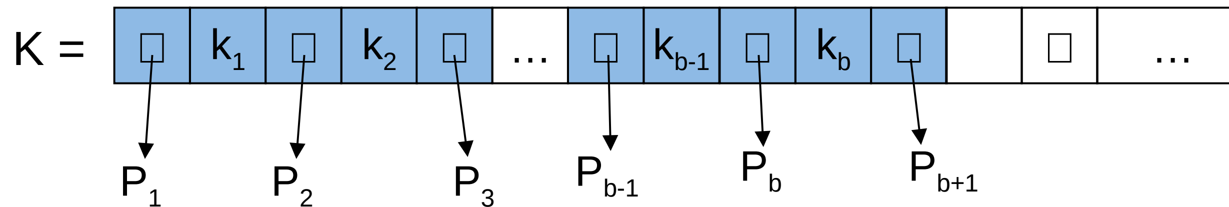
- Rekursive Aufspaltung liefert Baumstruktur



- Eigenschaften der Datenstruktur
  - Blattknoten enthalten Datensätze
  - Innere Knoten enthalten Knotenbeschreibungen und Zeiger
  - Alle Blätter haben dieselbe Entfernung von der Wurzel
  - Jeder Knoten hat höchstens M viele Einträge
  - Jeder Knoten (außer Wurzel) hat mindestens $m \geq M/2$ Einträge

Mehrweg-Bäume

- Charakterisierung
  - Knoten haben bis zu M+1 viele Nachfolger
  - Blockorientierte Speicherung der Bäume: Speichere Knoten auf Plattenseiten
  - Damit ergibt sich M aus Seitengröße und Größe der Datensätze + Zeiger



- Knoten K eines (M+1)-Wege Suchbaums besteht aus:
  - Verzweigungsgrad b+1 = Grad(K) $\leq$ M+1
  - Datensätze mit Schlüsseln k_i ($1 \leq i \leq b$)
  - Zeiger P_i auf die Unterbäume ($1 \leq i \leq b+1$)

Mehrweg-Suchbäume

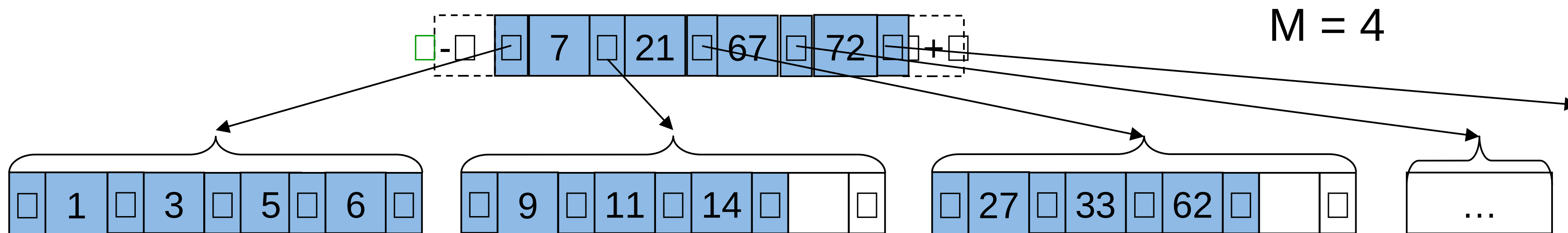
- Suchbaumeigenschaft für Mehrwegebäume
  - Die Schlüssel $k_1, k_2, \ldots, k_b$ in einem Knoten K sind geordnet, d.h. für i = 1, ..., b–1 gilt:

    $$k_i \leq k_{i+1}$$

  - Für alle Schlüssel k‘ im Teilbaum, der zwischen den Schlüsseln k_{p-1} und k_p liegt, gilt (setze $k_0 := -\infty$ und $k_b := +\infty$):

    $$k_{p-1} < k' \leq k_p$$

- Beispiel



B-Baum

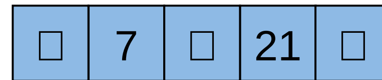
- Definition:
  Ein B-Baum ist ein (M+1)-Wege Suchbaum (für eine gerade Zahl M)

- Für einen nicht leeren B-Baum gilt:
  1. Jeder Knoten enthält höchstens M Schlüssel
  2. Die Wurzel enthält mindestens einen Schlüssel
  3. Jeder Knoten außer der Wurzel enthält mindestens m = M/2 Schlüssel
  4. Ein innerer Knoten mit b Schlüsseln hat genau b+1 Kinder
  5. Alle Blätter befinden sich auf demselben Level (Balanciertheit)

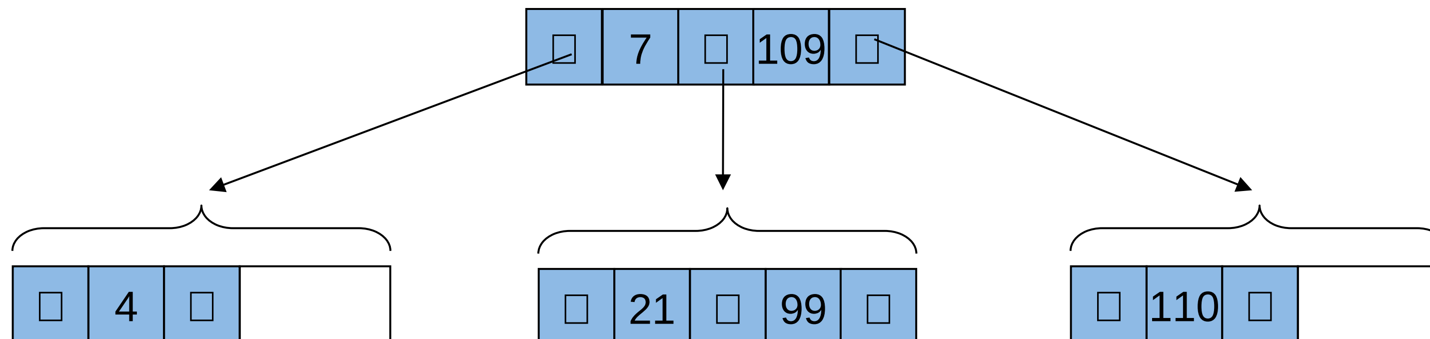
- Bedeutung des „B“:
  - **B**alanced Tree, **B**locked Tree (technische Beschreibung)
  - **B**ushy Tree, **B**road Tree (Hinweis auf hohen Verzweigungsgrad)
  - Prof. Dr. Rudolf **B**ayer (mit Ed McCreight Erfinder der B-Bäume)
  - The **B**oeing Company (Bayer arbeitete in deren Forschungslabor)
  - **B**arbara (Vorname von Bayers Ehefrau)
  - **B**anyan Tree (australischer Baum, wächst durch Wurzelteilung)
  - **B**inary Tree (falsch, da Mehrwegebaum; richtig, da binäre Suche)

Beispiele für B-Bäume

- Beispiele für B-Bäume mit M = 2 (d.h. maximal 3 Nachfolger)
  - Bsp. Höhe 1



  - Bsp. Höhe 2



Suchen im B-Baum

- Suche nach einem Datensatz
  - Beginne in der Wurzel und suche binär auf dem jeweiligen Knoten
  - Falls nicht gefunden: Suche im entsprechenden Teilbaum rekursiv weiter

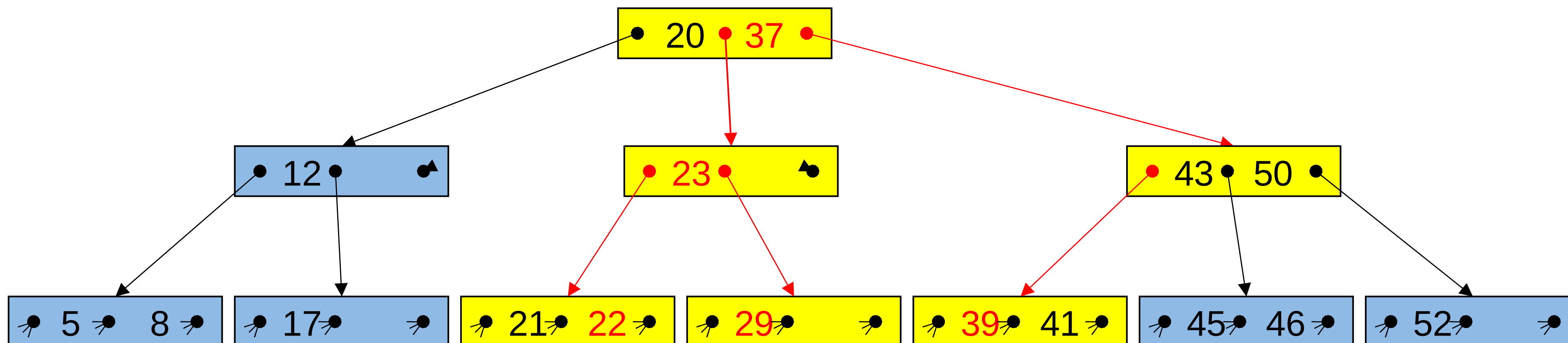
- Komplexität der Suche
  - In einem B-Baum der Höhe h werden maximal h viele Knoten besucht
  - Die Höhe eines B-Baums mit N Objekten ist maximal $h = \log_m N$ (s.später)
  - Die binäre Suche auf einem Knoten benötigt maximal $\log_2 M$ Vergleiche
  - Anzahl der Vergleiche insgesamt: höchstens $\log_2 M \cdot \log_m N$ □ $\log_2 N$
  - Anzahl der Plattenzugriffe (jeweils ca. 10ms): höchstens $\log_m N$

- Beispiel
  N = 1 Mio, M = 100 gibt 20 Vergleiche, 3 Plattenzugriffe (Wurzel im Cache)

Bereichsanfrage im B-Baum

- Suche Objekte, die in einen Bereich (min, max) fallen
  - Suche rekursiv jeweils binär den kleinsten Eintrag e ≥ min
  - Gehe von e aus mit Inorder-Durchlauf bis zum größten Eintrag e‘ ≤ max
  - Sei r die Anzahl der dabei gefundenen Elemente
  - Anzahl Vergleiche: $O(r + \log_2 N)$
  - Anzahl Plattenzugriffe: $O(r/m + \log_m N)$

- Beispiel: (22, 40)

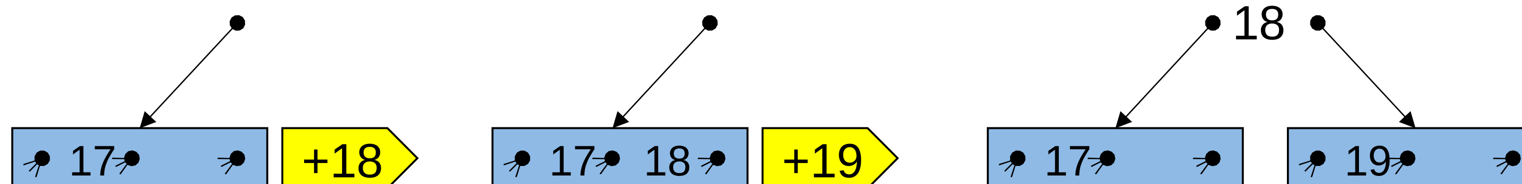


Einfügen im B-Baum

- Grundidee
  - Neue Objekte werden nur in Blättern eingefügt
  - Bei Überlauf eines Blatts wird ein neues Blatt erzeugt;
    die Einträge werden zwischen den beiden Nachbarknoten verteilt
  - Der Baum wächst nicht in die Tiefe, sondern in die Höhe

- Algorithmus: Einfügen eines neuen Objekts
  - Suche das Blatt, in welches das neue Objekt gehört
  - Füge das Objekt sortiert in das Blatt ein
  - Wenn hierdurch der Blattknoten überläuft, spalte ihn auf

- Beispiel

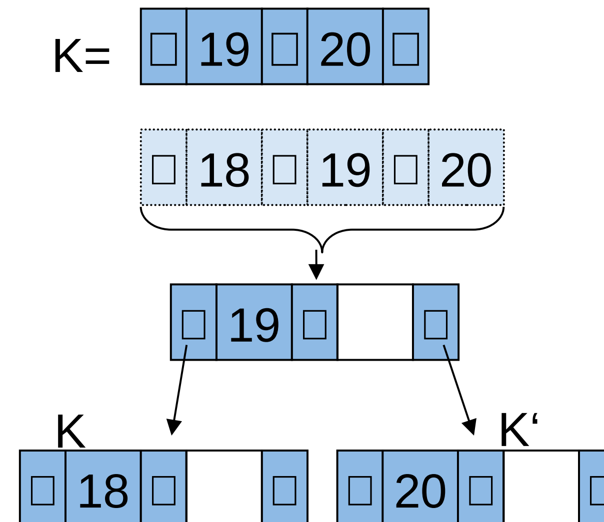


Einfügen im B-Baum: Split

- Überlauf eines Knotens
  - Knoten K kann M+1 Objekte (o_1, o_2, …, o_{M+1}) nicht fassen
  - Erzeuge einen Nachbarknoten K‘
  - Verteile die M+1 Objekte auf die beiden Knoten
    K = (o_1, o_2, …, o_m) und K‘ = (o_{m+2}, …, o_{M+1})
  - Das mittlere Objekt o_{m+1} wird dem Vorgängerknoten hinzugefügt

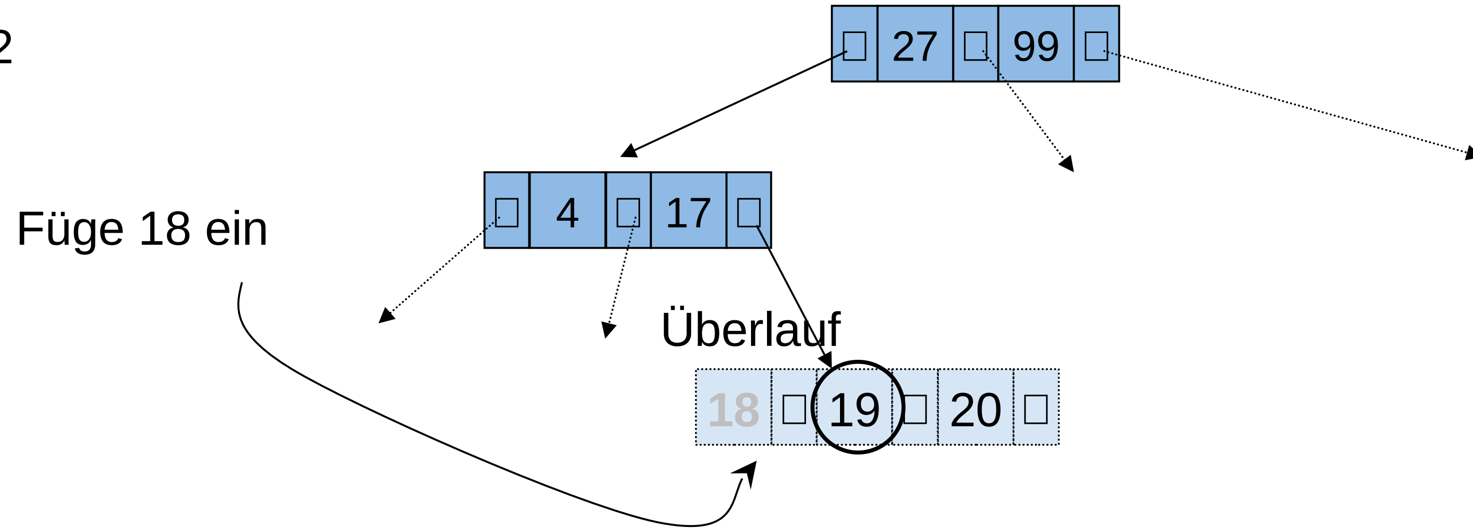
- Falls Vorgänger nicht existiert
  - Knoten war die Wurzel: Schaffe neue Wurzel
  - Die Höhe wächst um Eins

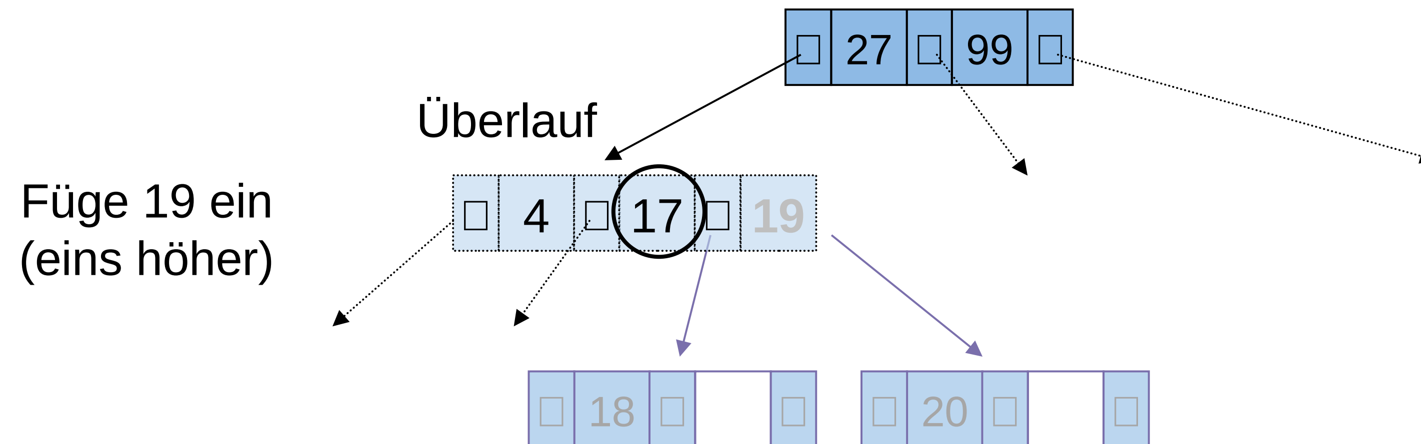
- Falls Vorgänger überläuft
  - Wende denselben Split-Algorithmus an
  - Split kann rekursiv bis zur Wurzel laufen
  - Komplexität des Einfügens: $O(\log_m N)$



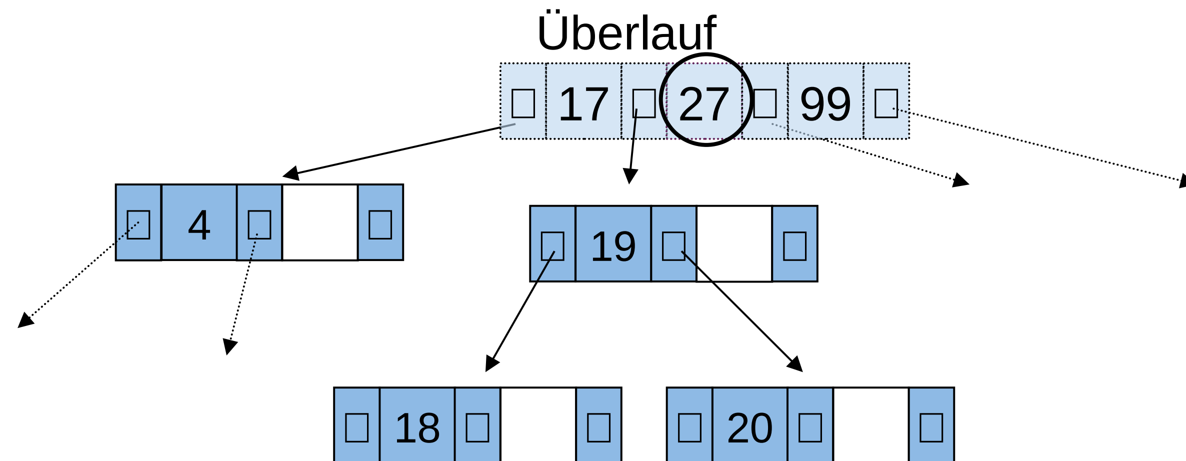
Beispiel für Einfügen im B-Baum

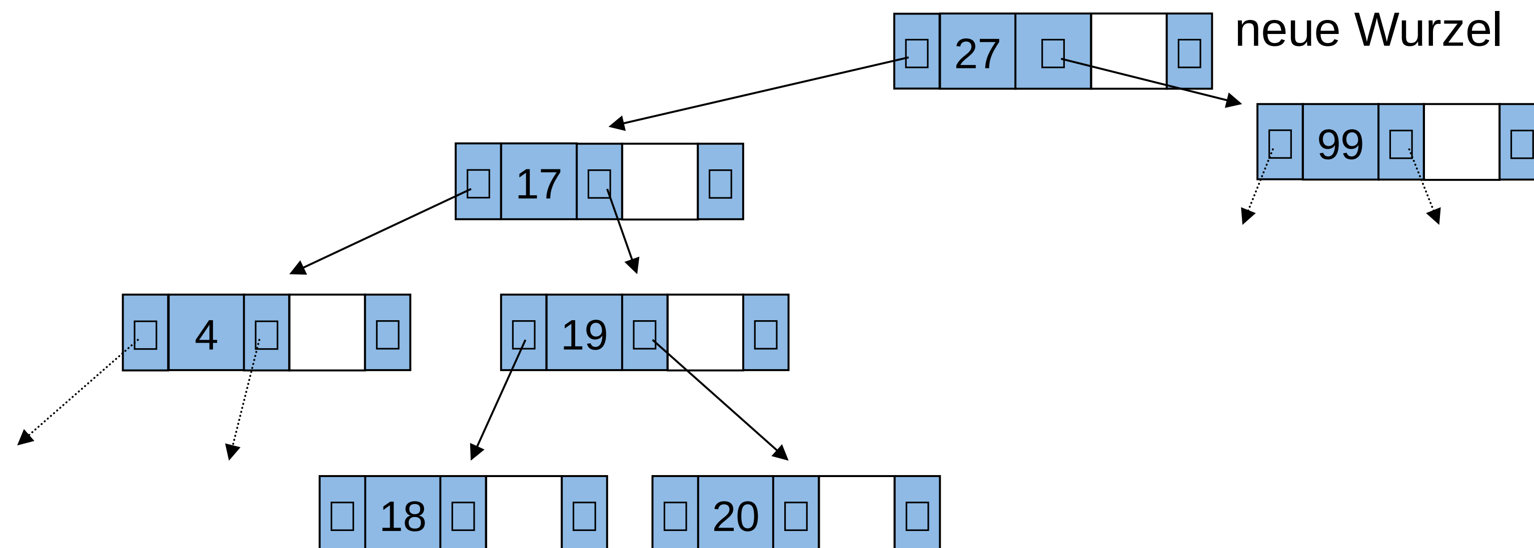
- Beispiel: M = 2





Beispiel (2)





Informatik 5 - Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Löschen im B-Baum

- Suche den Knoten K, der den zu löschenden Schlüssel o enthält

- Falls K ein Blatt ist: Lösche den Schlüssel o aus dem Blatt
  - Es ist möglich, dass K nun weniger als m = M/2 Schlüssel beinhaltet
  - Reorganisation unter Einbeziehung der Nachbarknoten

- Falls K ein innerer Knoten ist
  - Suche den größten Schlüssel o‘ im Teilbaum links von Schlüssel o
  - Ersetze o im Knoten K durch o‘
  - Lösche o‘ aus seinem ursprünglichen Knoten (das ist ein Blatt)

- Falls K die Wurzel ist
  - Die Wurzel hat keine Nachbarn und darf weniger als m = M/2 Schlüssel beinhalten

Löschen im B-Baum (Ausgleich)

Entferne Schlüssel o_i aus dem Knoten K = (o_1, …, o_b) eines B-Baums:

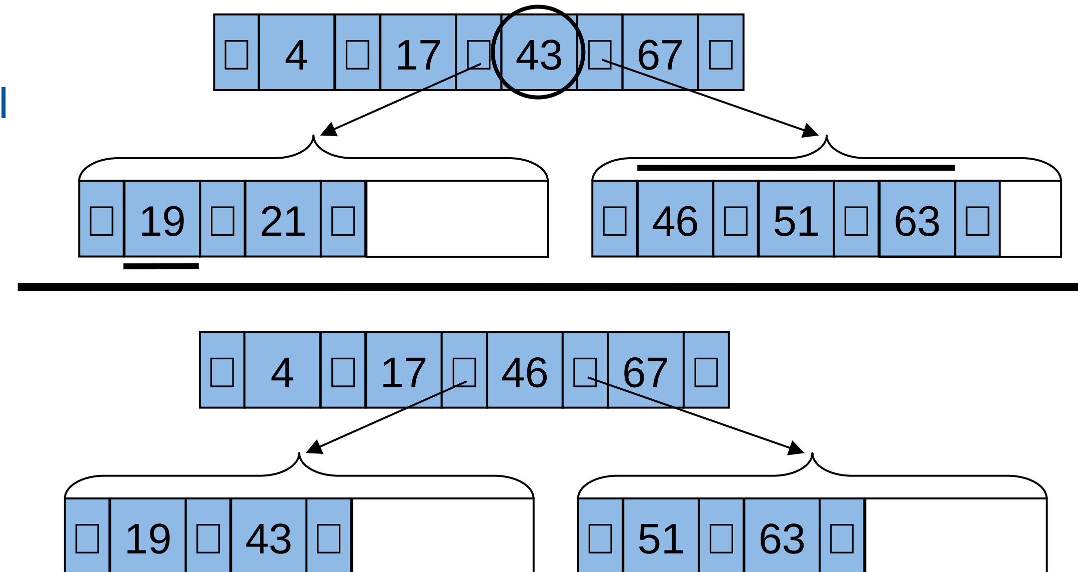
- Falls es einen Nachbarknoten K‘ = (o'_1,…,o'_n) mit **mehr** als m = M/2 Schlüsseln gibt, kann ein Ausgleich durchgeführt werden:
  - O.B.d.A. sei K‘ rechts von K, und p der Trennschlüssel im Vorgänger
  - Verteile die Schlüssel o_1 … o_b, p, o'_1 … o'_n auf die Knoten K und K‘, und ersetze den Schlüssel p im Vorgänger durch den mittleren Schlüssel
  - K und K‘ haben nun jeweils mindestens m = M/2 Schlüssel

Beispiel:

B-Baum mit M = 4

Lösche Schlüssel 21

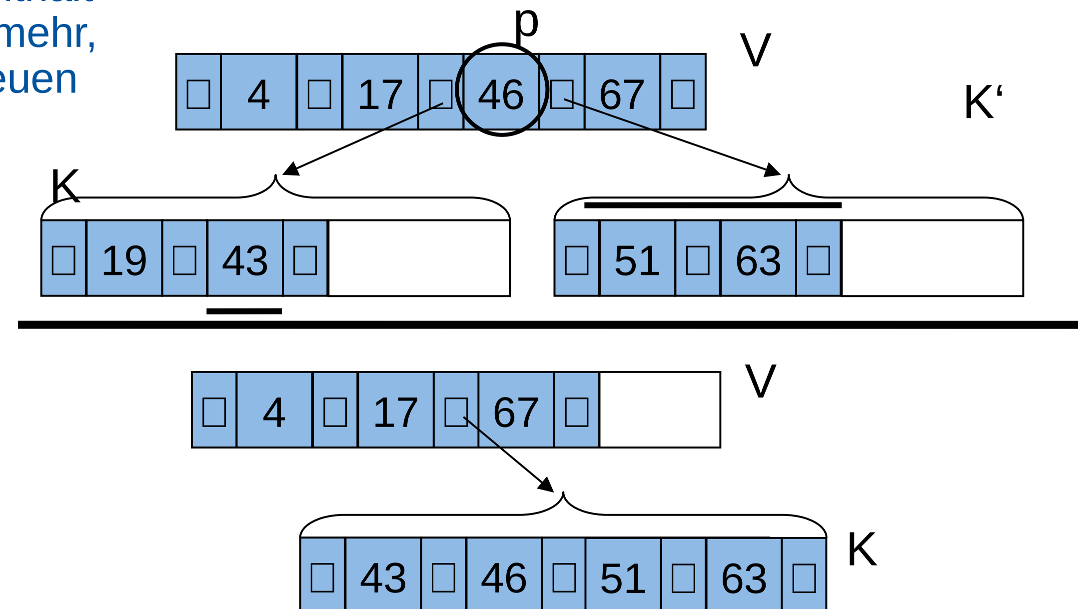
Ausgleich(19 , 43 , 46 , 51 , 63)



Löschen im B-Baum (Verschmelzen)

Falls es keinen Nachbarknoten mit mehr als m = $M/2$ Elementen gibt, so existiert mindestens ein Nachbarknoten K‘ = (o‘$_1$, …, o‘$_m$) mit **genau** *m* Schlüsseln:

- O.B.d.A. sei K‘ rechts von K, und *p* der Trennschlüssel im Vorgänger V
- Verschmelze die Knoten K‘ und K zu K, füge *p* in K hinzu und lösche K‘
- Entferne *p* sowie den Verweis auf K‘ aus dem Vorgänger V
- Ggf. rekursiv bis zur Wurzel (enthält diese danach keine Schlüssel mehr, so wird das einzige Kind zur neuen Wurzel)



Beispiel:
B-Baum mit M = 4
Lösche Schlüssel 19

Verschmelze (43 , 46 , 51 , 63)
Entferne (p=46)

Höhenabschätzung für B-Bäume

- Schlüsselanzahl N in einem Baum der Höhe h
  - Minimal:

$$N \geq 1 + 2m + 2(m+1) \cdot m + 2(m+1)^2 \cdot m + \ldots$$

$$= 1 + 2m \cdot \sum_{i=0}^{h-2} (m+1)^i = 2(m+1)^{h-1} - 1$$

$$N \leq M + (M+1) \cdot M + (M+1)^2 \cdot M + \ldots$$

  - Maximal:

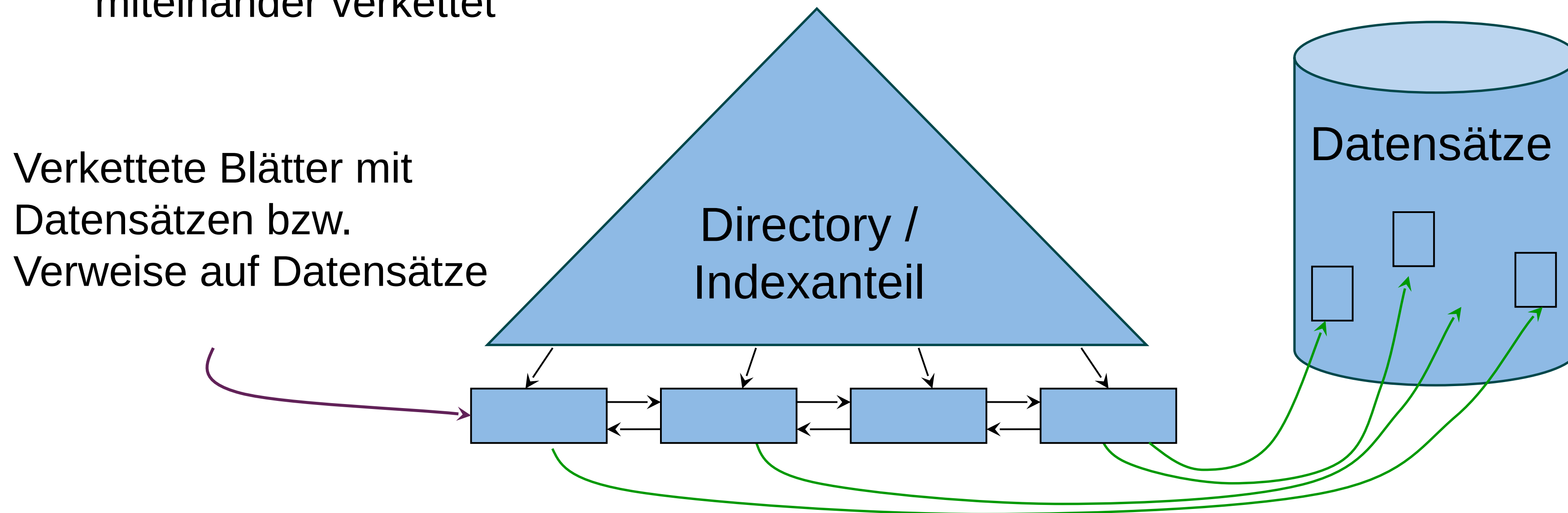
$$= M \cdot \sum_{i=0}^{h-1} (M+1)^i = (M+1)^h - 1$$

$$\log_{M+1}(N+1) \leq h \leq \log_{m+1}\left(\frac{N+1}{2}\right) + 1$$

- Auflösen nach h ergibt die Höhenabschätzung:
- Betrachtung der Schranken
  - Die Höhe eines B-Baumes ist durch den Logarithmus zur Basis der maximalen bzw. minimalen Anzahl Nachfolger eines Knotens beschränkt

Wichtige Variante B⁺-Baum

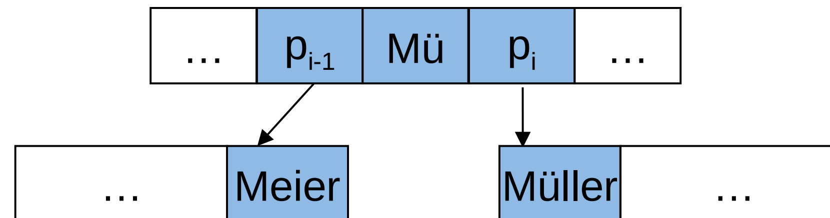
- Ein B⁺-Baum ist eine B-Baum-Variante mit zwei Knotentypen
  - Blätter enthalten Schlüssel mit Datensätzen oder Schlüssel mit Verweisen auf Datensätze
  - Innere Knoten enthalten keine Datensätze, nur Trennschlüssel
  - Als Trennschlüssel (Separatoren, Wegweiser) nutzt man z.B. die Schlüssel selbst oder geeignete Präfixe (bei Strings)
  - Für ein effizientes Durchlaufen großer Bereiche der Daten sind die Blätter miteinander verkettet



Vergleich B^+-Baum und B-Baum

- Da in den inneren Knoten nur Schlüssel ohne Daten gespeichert werden, haben auf einer B^+-Baum-Seite mehr Einträge Platz

- Schlüssel dienen nur als Wegweiser und können deswegen oft verkürzt werden:
  - Verwende als Trennschlüssel k_i (Wegweiser) z.B. das kürzeste Präfix des ersten Schlüssels im rechten Teilbaum p_i von k_i, das größer ist als der größte Schlüssel im linken Teilbaum p_{i-1} von k_i



- Dadurch B+-Baum in der Regel breiter und weniger hoch als B-Baum
- In der Praxis werden wegen dieser Vorteile überwiegend nur noch Varianten von B^+-Bäumen eingesetzt.

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Regelbasierte Anfrageoptimierung
   - Kostenbasierte Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - **Indexstrukturen für mehrdimensionale Daten**

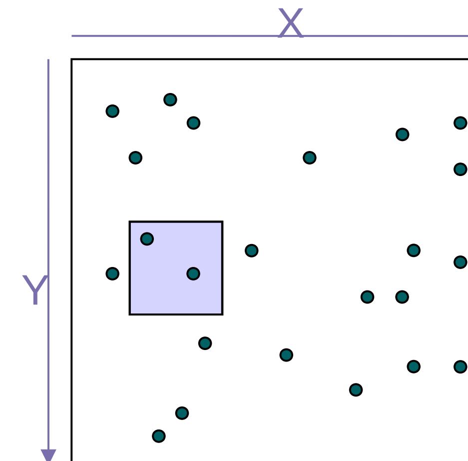
Mehrdimensionale Daten

- Problemstellung:
  - Gesucht wird anhand mehrdimensionaler Schlüssel $K = (a_1, \ldots, a_n)$ basierend auf verschiedenen Attributen $A_1, \ldots, A_n$
  - Gesuchte Attribute $A_1, \ldots, A_n$ sind gleichwertig

- Beispiel:

Matrikel-nummer	**Nebenfach**	**Semester**
171283	BWL	9
184238	Medizin	7
191373	Elektrotechnik	5
…	…	…

```
SELECT *
FROM Studenten
WHERE Nebenfach = „BWL“
    AND Semester >= 7
    AND   Semester <= 9
```

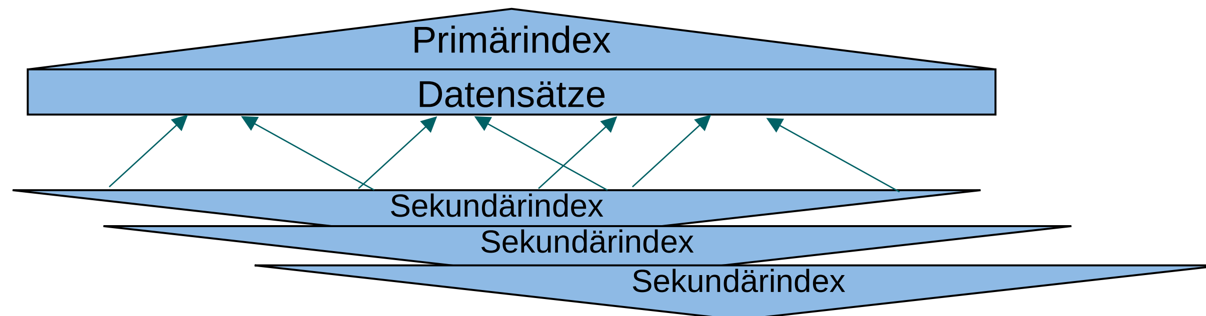



```
SELECT *
FROM Geometry
WHERE 2.5 <= x AND x <= 4.1
    AND 3.8 <= y AND y <= 6.2
```

Invertierte Listen

Ziel: Unterstützung von Anfragen über mehrere Attribute

- Multiattributssuche anhand *invertierter Listen*
  - Für jedes Attribut gibt es einen (Sekundär-) Index
  - Suche in allen Indexen unabhängig von den anderen
  - Kombiniere Ergebnis über Durchschnittsbildung
- Indexgefüge
  - *Primärindex:* Index über den Primärschlüssel
  - *Sekundärindex:* Index über ein Attribut, das kein Primärschlüssel ist
    - Im Gegensatz zu einem Primärindex beeinflusst der Sekundärindex den Ort der Speicherung eines Datensatzes nicht. Es werden nur Verweise gespeichert.



Invertierte Listen (2)

Konzept der invertierten Listen:

- Für anfragerelevante Attribute werden Sekundärindexe ***(invertierte Listen)*** angelegt.
- Damit steht für jedes relevante Attribut eine eindimensionale Indexstruktur zur Verfügung.

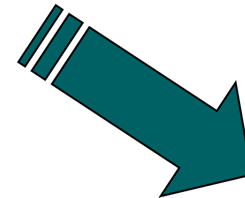
Multiattributsuche für invertierte Listen:

- Eine Anfrage spezifiziere die Attribute A_1, ... ,A_m:
  - *m Anfragen über m Indexstrukturen*
- Ergebnis:
  m Listen mit Verweisen auf die entsprechenden Antwortkandidaten in der Datei.
- *Mengentheoretische Verknüpfung* (z.B. Durchschnitt) der m Listen gemäß der Anfrage

Invertierte Listen (Beispiel)

Primärindex (über Name)

Speicher-adresse	Name	Stadt	Alter
1	*Adams*	Athen	30
2	*Blake*	Paris	30
3	*Clark*	London	50
4	*Hart*	Chicago	40
5	*James*	Athen	30
6	*Jones*	Paris	40
7	*Parker*	New York	40
8	*Smith*	London	30



invertierte Listen:

Stadt	Speicher-Adresse
Athen	1, 5
Chicago	4
London	3, 8
New York	7
Paris	2, 6

Alter	Speicher-Adresse
30	1, 2, 5, 8
40	4, 6, 7
50	3

```
SELECT *
FROM ….
WHERE Stadt = „Athen“
    AND Alter = 30
```

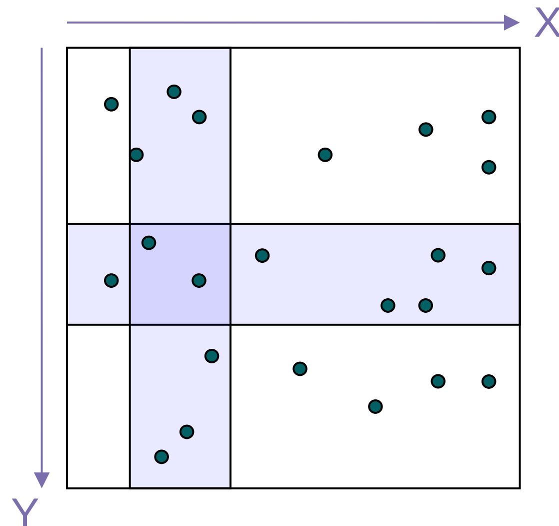
„Athen“ „30“

{1,5} ∩ {1,2,5,8} = {1,5}

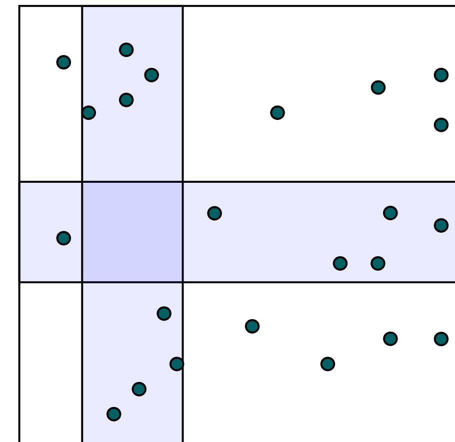
Invertierte Listen: Punktdaten

Bei 2-dimensionalen Punktdaten:

- Speicherung der X- und Y-Werte in jeweils einem Sekundärindex
- bei Suchen von Punkten in Bereichen (Rechtecken):
  - Bilde Punktmenge, deren X-Koordinaten im Anfragebereich liegen
  - Bilde Punktmenge, deren Y-Koordinaten im Anfragebereich liegen
  - Bilde die Schnittmenge beider Mengen



Antwort: zwei Punkte



„worst-case“

Invertierte Listen: Eigenschaften

- Die Antwortzeit ist nicht proportional zur Anzahl der Antworten.
- Die Suche dauert umso länger, je mehr Attribute spezifiziert sind.
- *Ursache für beide Beobachtungen:*

  Die Attributwerte eines Datensatzes sind nicht in einer Struktur miteinander verbunden.
- Invertierte Listen sind einigermaßen effizient, wenn die Antwortlisten sehr klein sind.
- Invertierte Listen haben hohe Kosten für Update-Operationen.
- Sekundärindexe beeinflussen die physische Speicherung der Datensätze nicht.
  - Ordnungserhaltung über den Sekundärschlüssel nicht möglich.
  - schlechtes Leistungsverhalten von invertierten Listen.

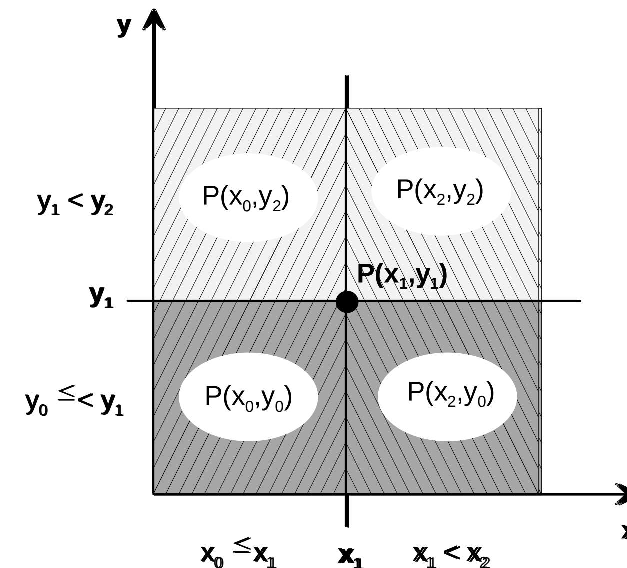
Beispiel: viele Attribute spezifiziert,
schlechte Antwortzeit:
 Bibliotheksrecherche

ggf. verursacht jeder Treffer
einen Plattenzugriff

Quadtree (2D)

3D: Octree
1D: Segmenttree

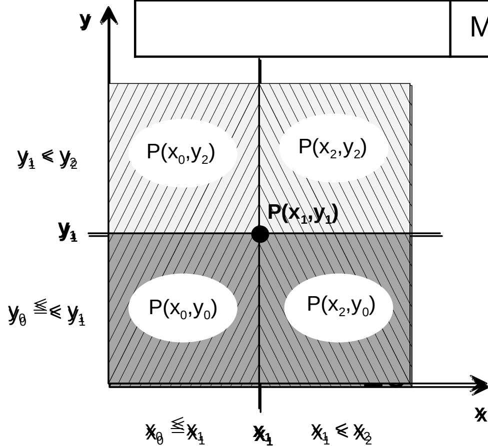
- nicht-balancierte Datenstruktur für mehrdimensionale Punkt-Objekte (hier: 2-dimensionale)
- Ansatz: Jeder Knoten hat sowohl für die x- als auch für die y-Koordinate je zwei Kindknoten: x (<=,>) und y (<=,>) (je Knoten also vier Kindknoten „*Quad*“-Tree)
- Aufbau des Baums:
  - erster Punkt bildet Wurzel
  - bei jedem weiteren Punkt:
    - Bestimmung des Quadranten, in dem Punkt liegt
    - Abstieg in entsprechenden Kindknoten
    - falls kein Kindknoten für diesen Quadranten existiert: Hinzufügen eines Kindknotens



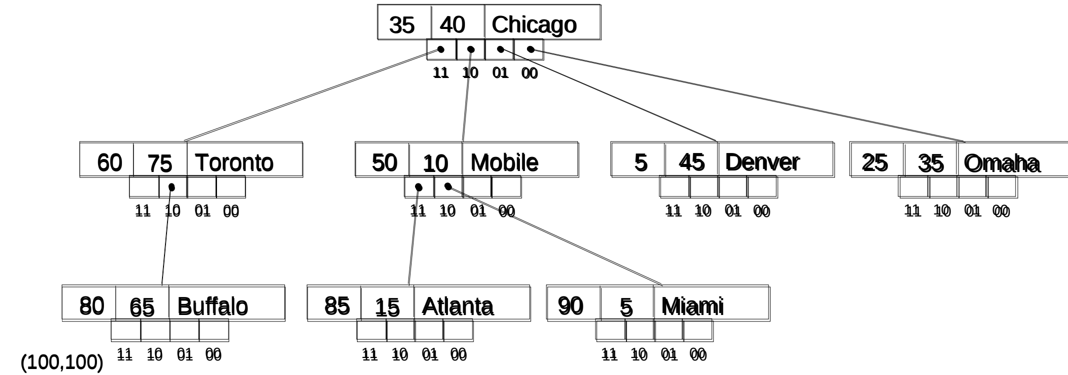
- hier: Hauptspeicherstruktur (Verzweigungsgrad = 2^d für d Dimensionen)
- später: Quadrant = Bucket zur Verwendung als Sekundärspeicherstruktur

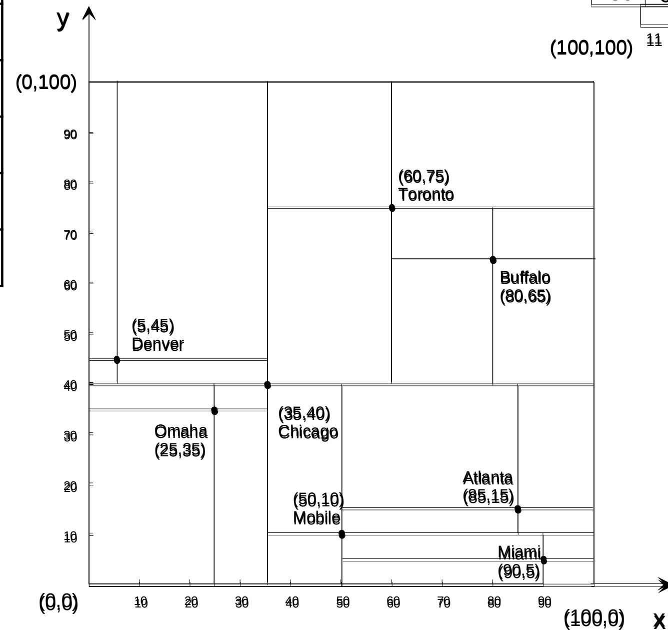
Quadtree (Beispiel)

einzufügen:	Vergleichs-knoten	$D_{binär}$
Chicago (35,40)	-	
Denver (5,45)	Chicago (35,40)	01
Mobile (50,10)	Chicago (35,40)	10
Toronto (60,75)	Chicago (35,40)	11
Buffalo (80,65)	Chicago (35,40)	11
	Toronto (60,75)	10
Miami (90,5)	Chicago (35,40)	10
	Mobile (50,10)	10
Omaha (25,35)	Chicago (35,40)	00
Atlanta (85,15)	Chicago (35,40)	10
	Mobile (50,10)	11



Nach allen Einfügungen:





R-Baum

(Guttman A.: ‘R-trees: A Dynamic Index Structure for Spatial Searching’, Proc. ACM SIGMOD Int. Conf. on Management of Data,1984, pp. 47-57.)

R (Rectangle)-Baum: höhenbalancierter Baum zur Speicherung von Punkt- und Rechteckdaten

Idee:

- basiert auf der Technik überlappender Seitenregionen
- Approximation der Objekte durch minimale umgebende Rechtecke
  (Abk. MUR, engl. MBR “minimum bounding rectangle“)
- verallgemeinert die Idee des B^+-Baums auf den mehrdimensionalen Raum

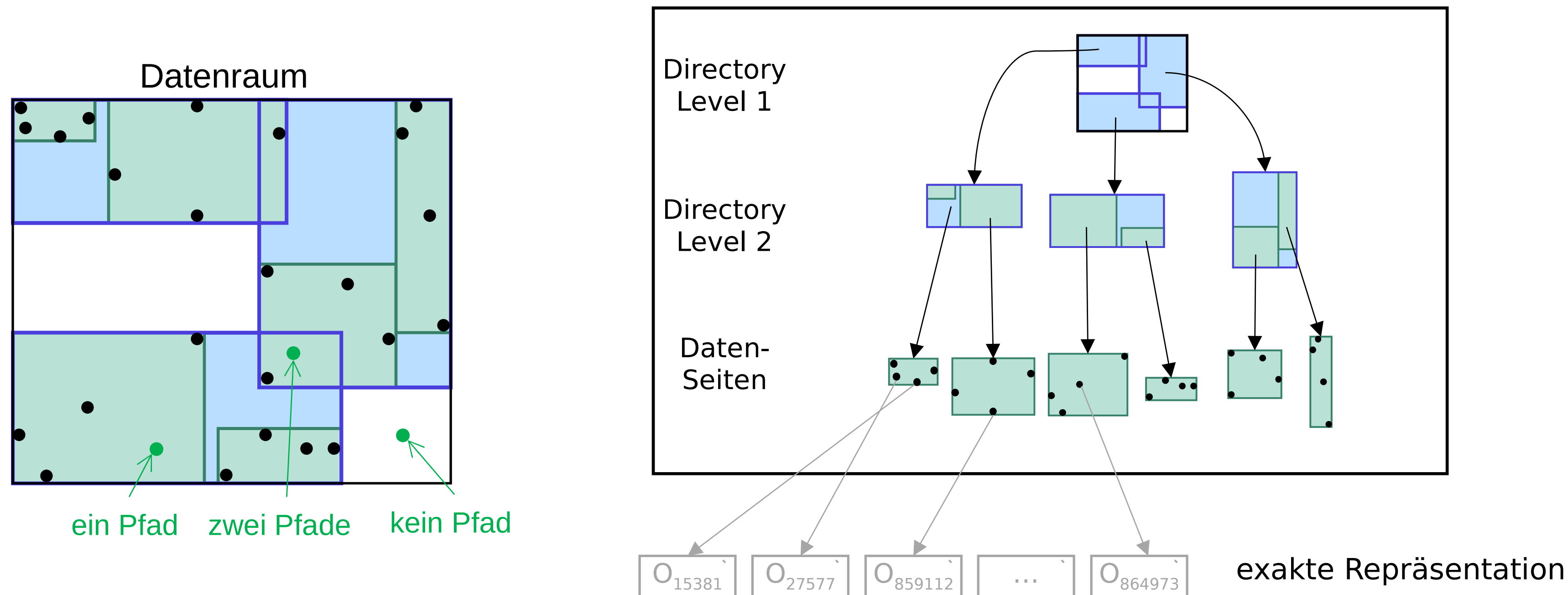
Aufbau einer Seite:

- Seite besteht aus mehreren Einträgen
- Einträge in Directory-Seiten bestehen aus MURs und Verweisen auf andere Seiten
- Einträge in Datenseiten bestehen aus MURs und Verweisen auf die exakte Objekt-Repräsentation, bzw. einfach aus Punkten

R-Baum (2)

“Partitionierung“ des Datenraums:

- jedes Rechteck in einer Directoryseite umfasst als MUR alle Rechtecke in allen Directory- oder Datenseiten, die im zugehörigen Teilbaum liegen
- nicht disjunkt: die Rechtecke einer Seite können sich überlappen
- „Partitionierung“ des Datenraumes der Directoryseite muss nicht vollständig sein, d.h. es existiert „leerer“ Raum



R-Baum: Eigenschaften

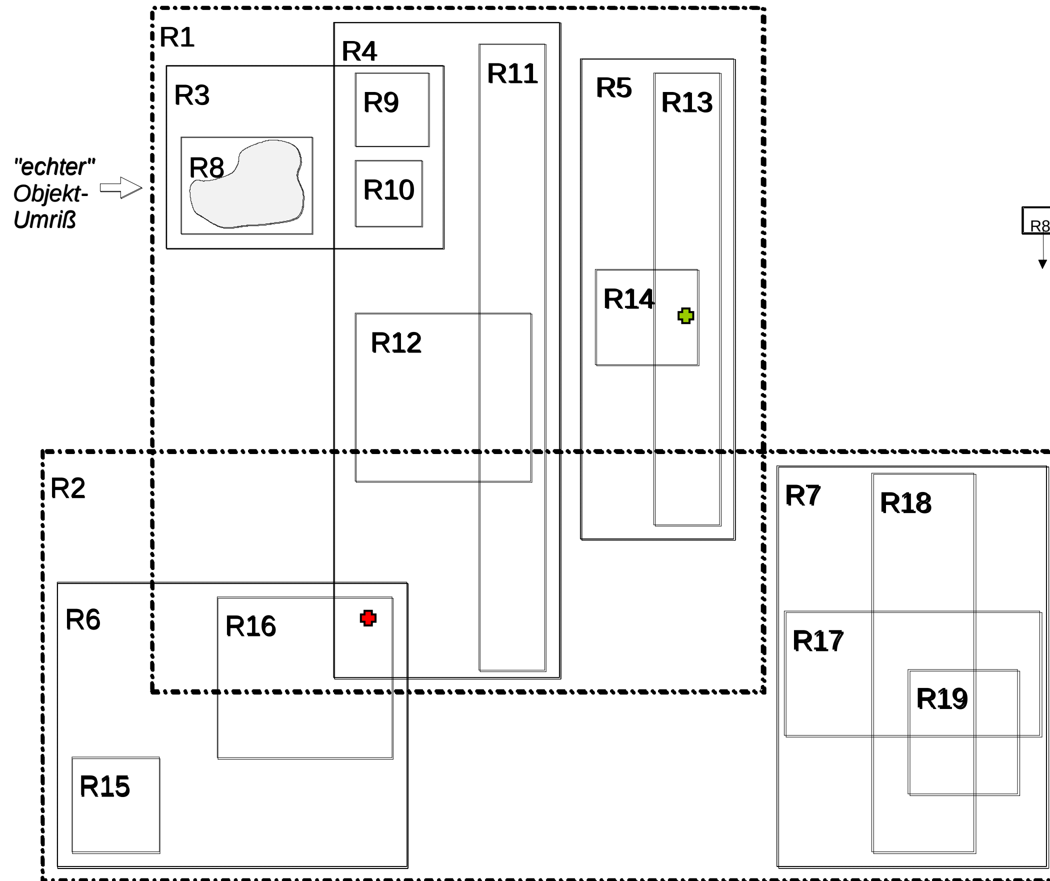
Parameter:

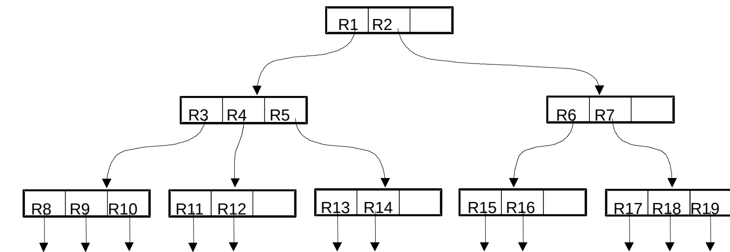
- M : maximale Anzahl von Einträgen pro Knoten (abhängig von Blockgröße)
- $m \leq M/2$: Mindestbelegung pro Knoten (z.B. m = 40 % · M)

Eigenschaften:

- Anzahl der Index-Einträge pro Blatt-Knoten zwischen m und M
- Anzahl der Kindknoten von Nichtblatt-Knoten (Directory-Knoten) zwischen m und M
- in inneren Knoten ist das kleinste Rechteck gespeichert, welches Rechtecke der Kindknoten umfasst (MUR: **M**inimal **U**mgebendes **R**echteck, engl. MBR: **M**inimum **B**ounding **R**ectangle)
- in Blatt-Knoten (Datenknoten) ist Verweis auf Objekt und sein kleinstes umschließendes Rechteck gespeichert
- höhenbalanciert (Blattknoten auf derselben Höhe)
- Zerlegung des Datenraums nicht disjunkt (also überlappende Regionen möglich)
- Höhe des Baums ≤ $\lceil \log_m N \rceil - 1$ (bei N gespeicherten Objekten)

R-Baum: Punktanfrage



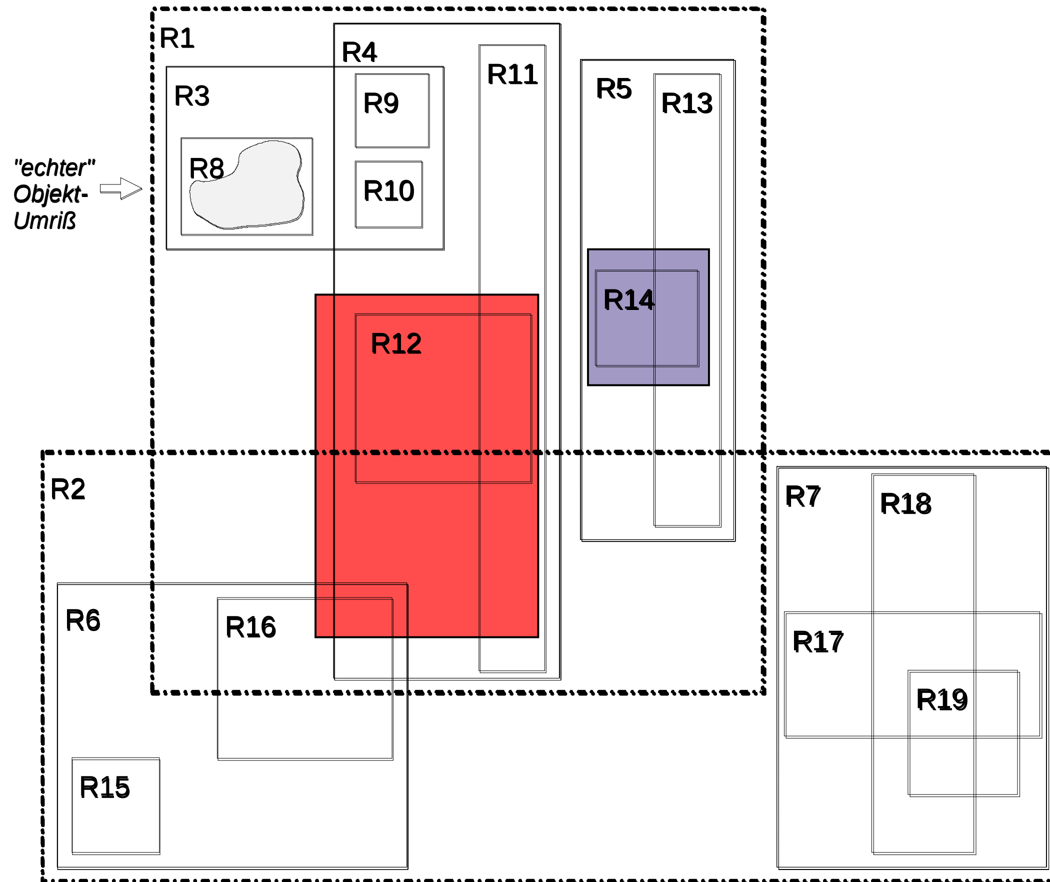


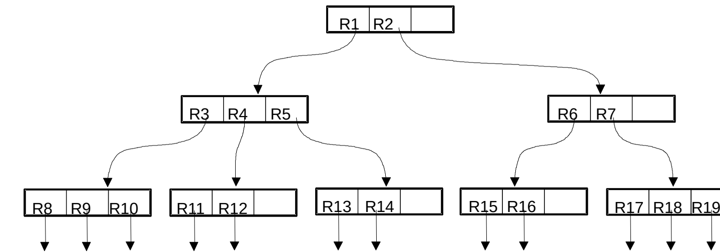
Eingabe: *Punkt p*

1. Starten bei Wurzel
2. Tiefensuche im R-Baum
3. Untersuche jeweils Kindknoten von Nichtblattknoten, deren Rechteck den Punkt p enthält
4. Überprüfe in Blattknoten, ob eines der Rechtecke den Punkt p enthält

- untersuche R1 □ R3 □ **R8** □ R9 □ R10 □ R4 □ R5 □ R2
- untersuche R1 □ R3 □ R4 □ R5 □ **R13** □ **R14** □ R2
- untersuche R1 □ R3 □ R4 □ R11 □ R12 □ R5 □ R2 □ R6 □ R15 □ **R16** □ R7

R-Baum: Rechteckanfrage (Schnitt)





Eingabe: Rechteck r

1. Starten bei Wurzel
2. Tiefensuche im R-Baum
3. Untersuche jeweils Kindknoten von Nichtblattknoten, deren Rechteck das Rechteck R schneidet
4. Überprüfe in Blattknoten, ob eines der Rechtecke das Anfragerechteck schneidet

R1 □ R3 □ R4 □ R5 □ **R13** □ **R14** □ R2

R1 □ R3 □ R4 □ **R11** □ **R12** □ R5 □ R2 □ R6 □ R15 □ **R16** □ R7

R-Baum: Einfügen

- ähnlich wie im B^+-Baum
- Einfügungen erfolgen stets in den Blattknoten
- im Gegensatz zum B-Baum kommen hier i. a. mehrere Blattknoten in Frage (Überlappungen von minimal umgebenden Rechtecken)
- Wahl des Blattknotens/Teilbaumes mit minimaler Vergrößerung der MURs
- Durch Einfügen eines neuen Elements kann ein Knoten überlaufen: Verschiedene Heuristiken:
  - Quadratischer Split
    - Laufzeitkomplexität ist quadratisch in der Anzahl der Rechtecke
    - Verteile Einträge auf zwei Knoten, so dass die Flächenvergrößerung des minimal umgebenden Rechteck am geringsten ist
  - Linearer Split
    - Laufzeitkomplexität ist linear in der Anzahl der Rechtecke
    - Basierend auf größter normalisierter Separierung in den Dimensionen

R-Baum: Löschen

- Beginnend bei der Wurzel, durchsuche alle Teilbäume, in denen der zu löschende Eintrag sein könnte, bis Eintrag gefunden ist.
- Entferne Objekt aus Plattenblock.
- Passe minimal umgebende Rechtecke auf dem Pfad zurück zur Wurzel an (falls nötig).
- Zwei Strategien, falls Unterlauf in Blattknoten auftritt:
  - Unterlauf behandeln: Verschmelze Nachbarknoten, propagiere Entfernung nach oben.
  - Unterlauf ignorieren: Beliebte Vorgehensweise; falls mehr Einfügungen als Entfernungen auftreten, wird die Seite vermutlich schon bald wieder weiter belegt. Verschmelzung und erneuter Split kann eingespart werden.

5. Relationale Anfragebearbeitung: Zusammenfassung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Regelbasierte Anfrageoptimierung (Logik-/Algebra-Transformation)
   - Kostenbasierte Anfrageoptimierung (Statistik+Kombinatorik)

2. Algorithmen für Basisoperationen (Speicherhierarchie)

3. Indexstrukturen (Blockorientierte Erweiterung von DS&A)
   - Indexstrukturen für eindimensionale Daten (Struktur-/Textdaten)
   - Indexstrukturen für mehrdimensionale Daten (Geo-/Raum-Daten)