5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die 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**
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Auschnitt aus unserer Datenbank

Student		
MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

8 Tupel

hört	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

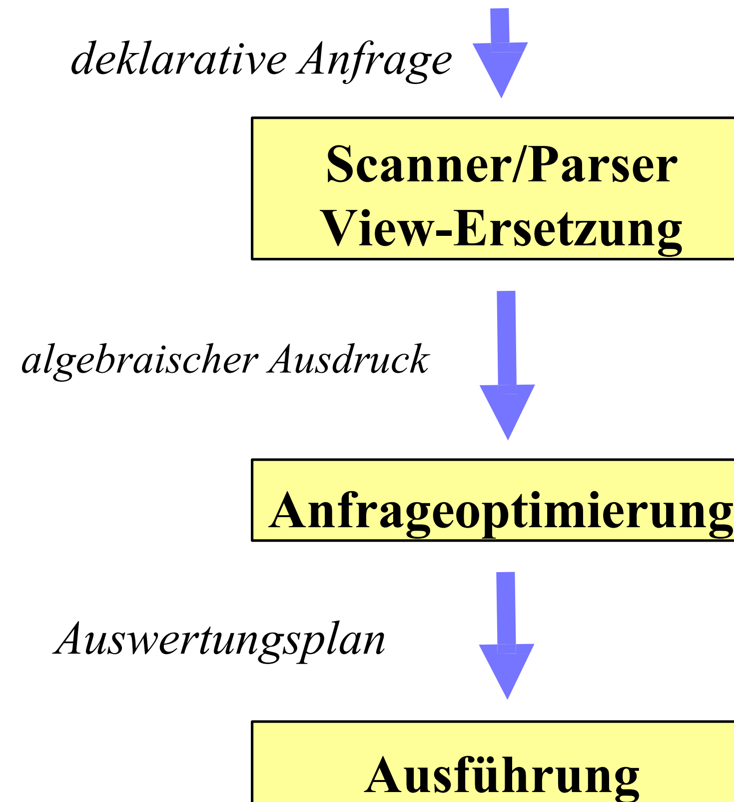
12 Tupel

Vorlesung			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

10 Tupel

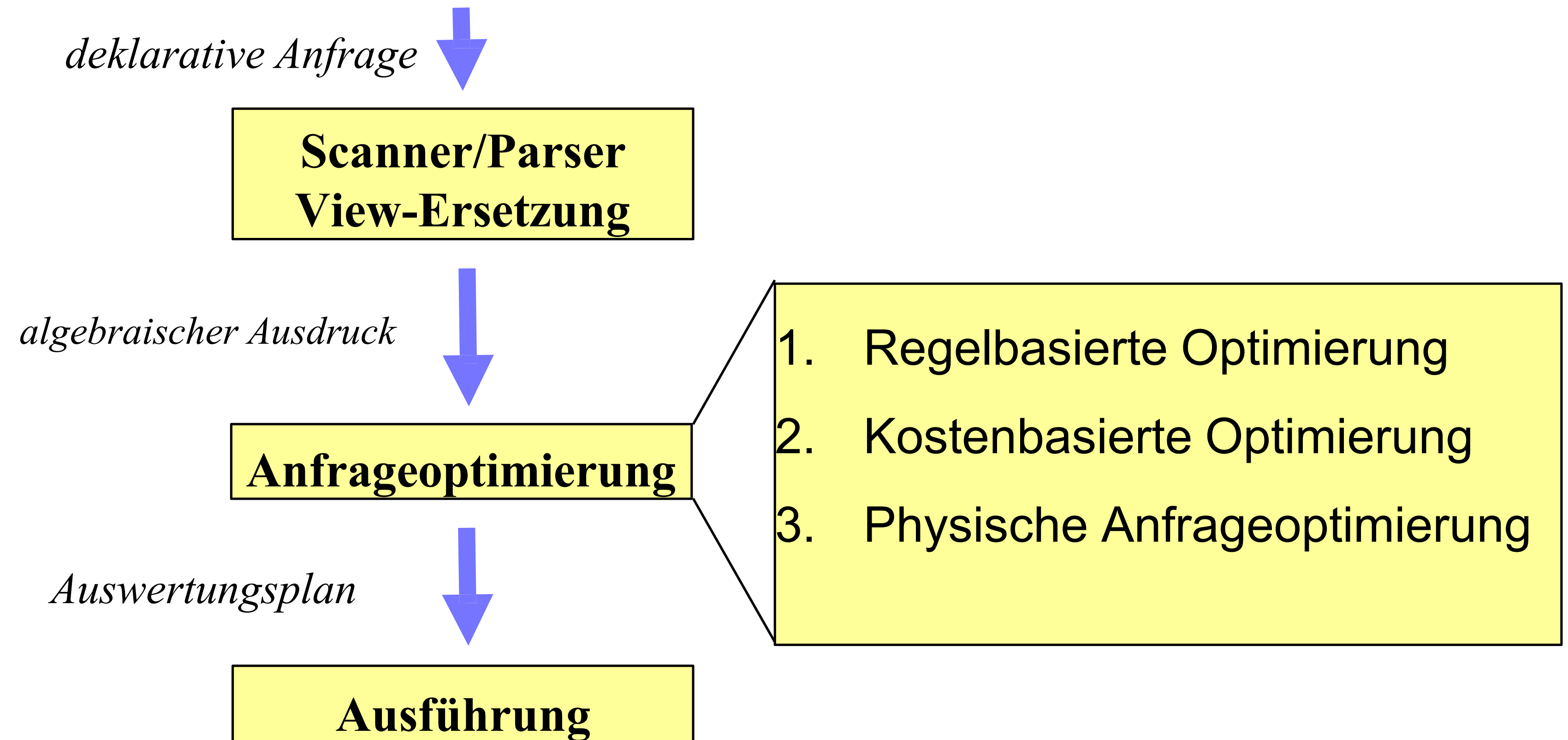
Aufgabe der Anfragebearbeitung

– Übersetzung der *deklarativen* Anfrage in einen *effizienten, prozeduralen* Auswertungsplan



Aufgabe der Anfragebearbeitung

– Übersetzung der *deklarativen* Anfrage in einen *effizienten, prozeduralen* Auswertungsplan



Drei 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
  - Zum Beispiel für die Join-Reihenfolge
- Physische Anfrageoptimierung
  - Auswahl einer geeigneten Auswertungsstrategie für die Join-Operation
  - Verwendung Index bei Selektionsoperation (und welche Art)
  - Pipelined vs. Materialisierung

Beobachtungen

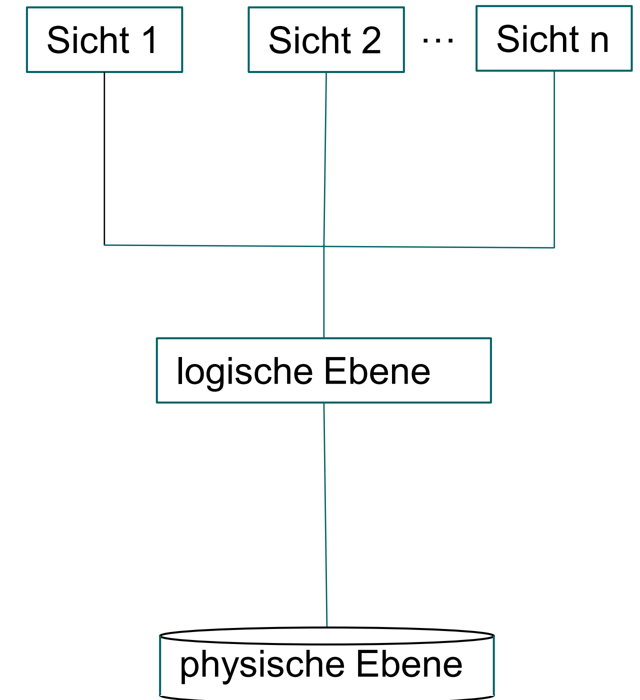
- Es kann viele verschiedene, *gleichwertige* Auswertungspläne für dieselbe Anfrage geben.
- Die Performanz gleichwertiger Auswertungspläne variiert häufig zwischen wenigen Sekunden (schnellster Plan) und vielen Stunden (Standardplan).
- Die Aufgabe der Anfrageoptimierung:

  ➡ 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.

Weitere Gründe für die Optimierung

- „physischen Datenunabhängigkeit“: Benutzer soll von der physischen Organisatisation der Daten abgeschirmt sein
- 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.



3-Ebenen Modell nach ANSI/SPARC (1975)

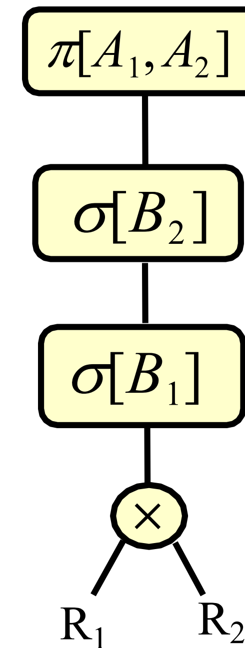
Kanonischer Auswertungsplan einer SQL Anfrage

SELECT A_1, A_2

FROM R_1, R_2

WHERE B_1 *AND* B_2

$$\pi_{A_1,A_2}(\sigma_{B_2}(\sigma_{B_1}(R_1 \times R_2)))$$



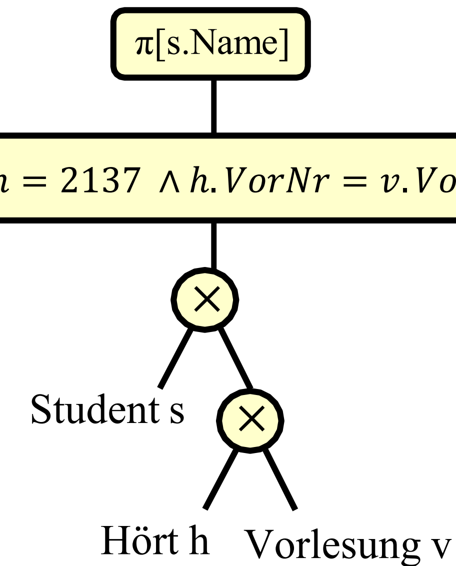
1. Bilde das kartesische Produkt der Relationen R_1, R_2
2. Führe Selektionen mit den Bedingungen B_1, B_2 durch.
3. Projiziere die Ergebnis-Tupel auf die erforderlichen Attribute A_1, A_2

Beispiel: SQL-Anfrage

- Anfrage: *Welche Studierenden (Name) sind mindestens im siebten Semester und haben eine Vorlesung besucht, die von Professor 2137 gelesen wurde?*

- SQL:

```
SELECT DISTINCT s.Name
FROM Student s CROSS JOIN Hört h CROSS JOIN Vorlesung v
WHERE s.MatrNr = h.MatrNr
   AND h.VorlNr = v.VorlNr
   AND v.gelesenVon = 2137
   AND s.Semester >= 7
```



- Übersetzung in die rel. Algebra (kanonisch):

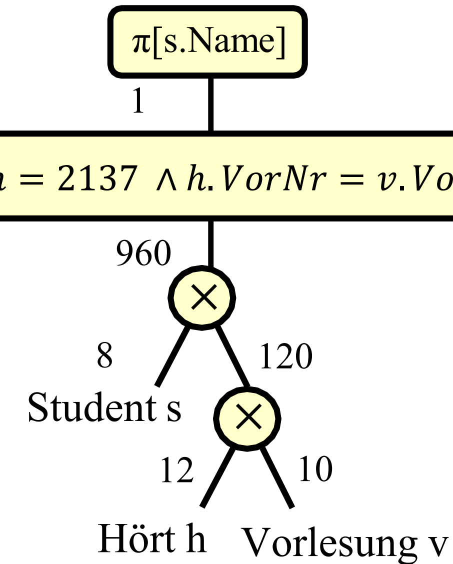
$$\pi_{s.Name}(\sigma_{s.Semster \geq 7 \wedge v.gelesenVon=2137 \wedge h.VorNr=v.VorlNr \wedge s.MatrNr=h.MatrNr}(Student \times (Hört \times Vorlesung)))$$

Beispiel: SQL-Anfrage

- Anfrage: *Welche Studierenden (Name) sind mindestens im siebten Semester und haben eine Vorlesung besucht, die von Professor 2137 gelesen wurde?*

- SQL:

SELECT DISTINCT s.Name
FROM Student s **CROSS JOIN** Hört h **CROSS JOIN** Vorlesung v
WHERE s.MatrNr = h.MatrNr
  AND h.VorlNr = v.VorlNr
  AND v.gelesenVon = 2137
  AND s.Semester >= 7



- Übersetzung in die rel. Algebra (kanonisch):

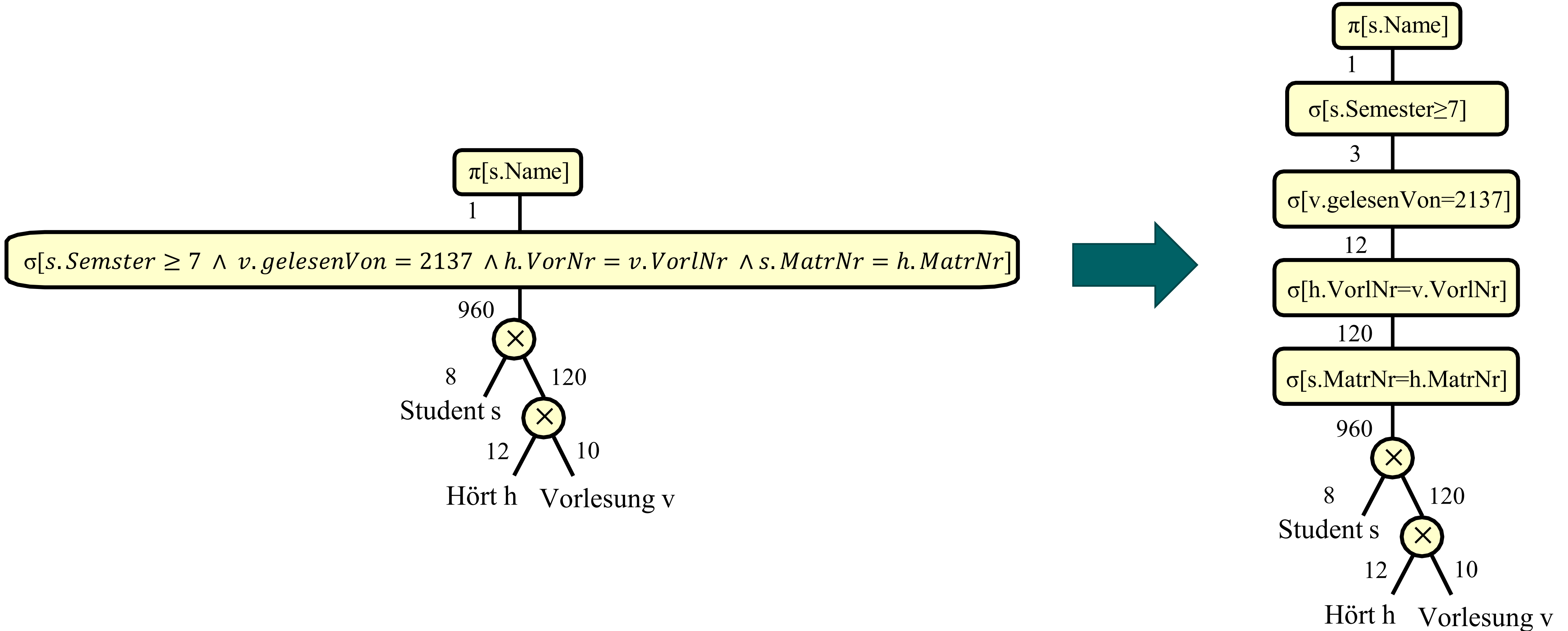
$$\pi_{s.Name}(\sigma_{s.Semster\geq 7 \wedge v.gelesenVon=2137 \wedge h.VorNr=v.VorlNr \wedge s.MatrNr=h.MatrNr}(Student \times (Hört \times Vorlesung)))$$

Prinzipien:

So früh wie möglich während der Abfragebearbeitung:

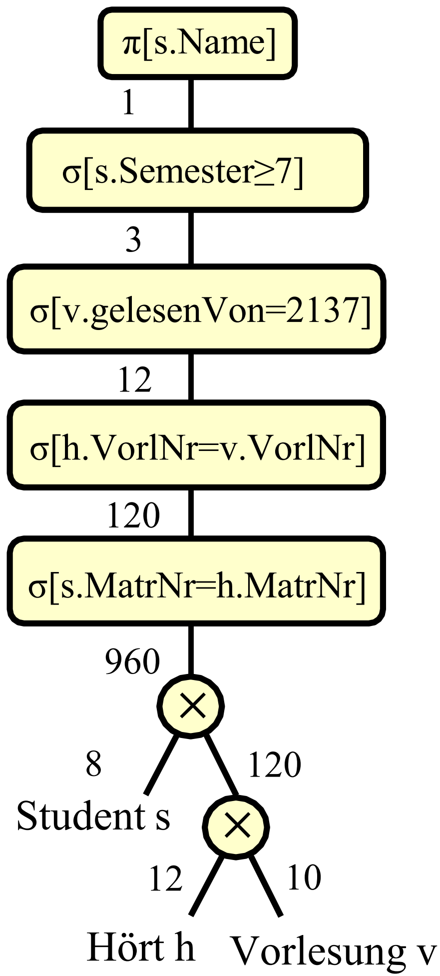
- Reduzierung der Anzahl der Tupel
- Reduzierung der Anzahl der Spalten

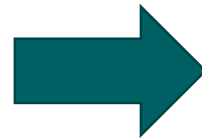
Anfrageoptimierung: Aufbrechen der Selektionen

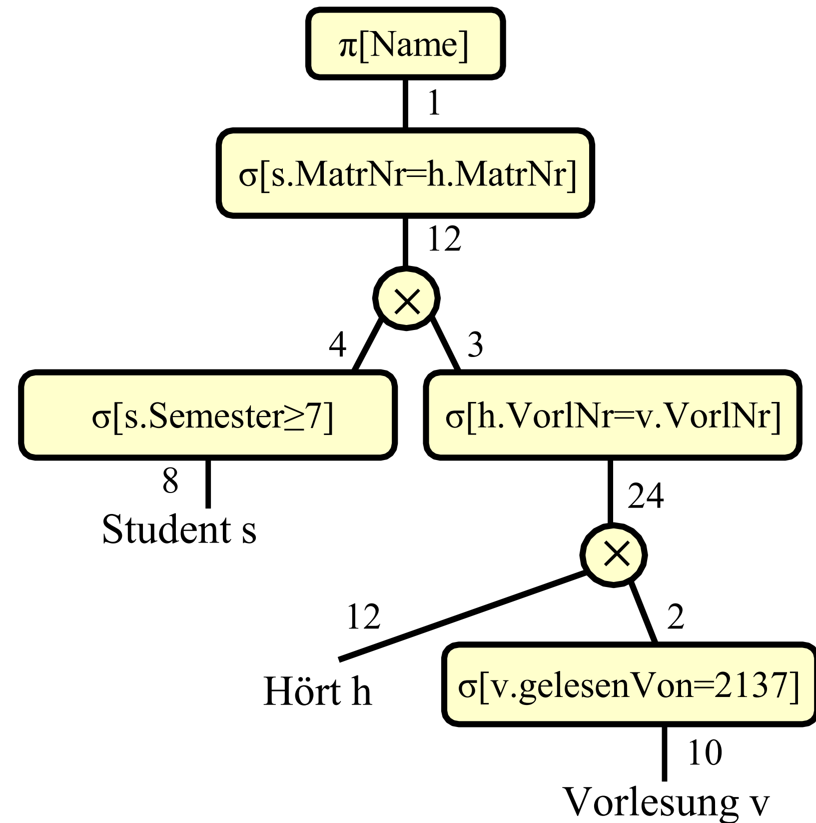


$$\pi_{s.Name}\left(\sigma_{s.Semster\geq 7}\left(\sigma_{v.gelesenVon=2137}\left(\sigma_{h.VorNr=v.VorlNr}\left(\sigma_{s.MatrNr=h.MatrNr}\left(Student \times (Hört \times Vorlesung)\right)\right)\right)\right)\right)$$

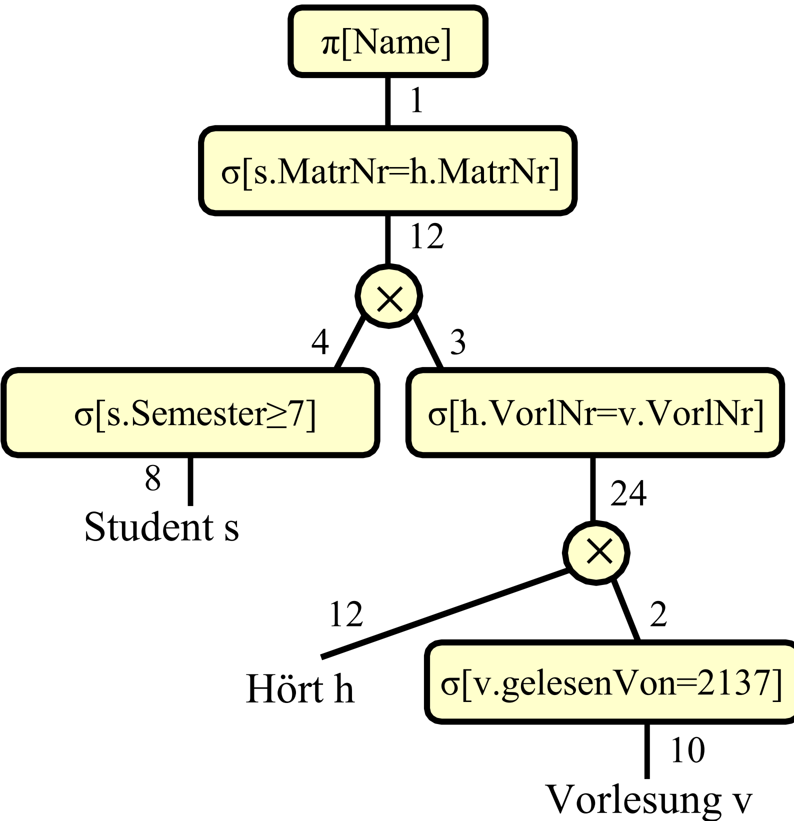
Anfrageoptimierung: Verschieben der Selektionen

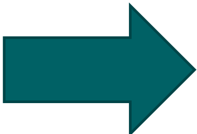


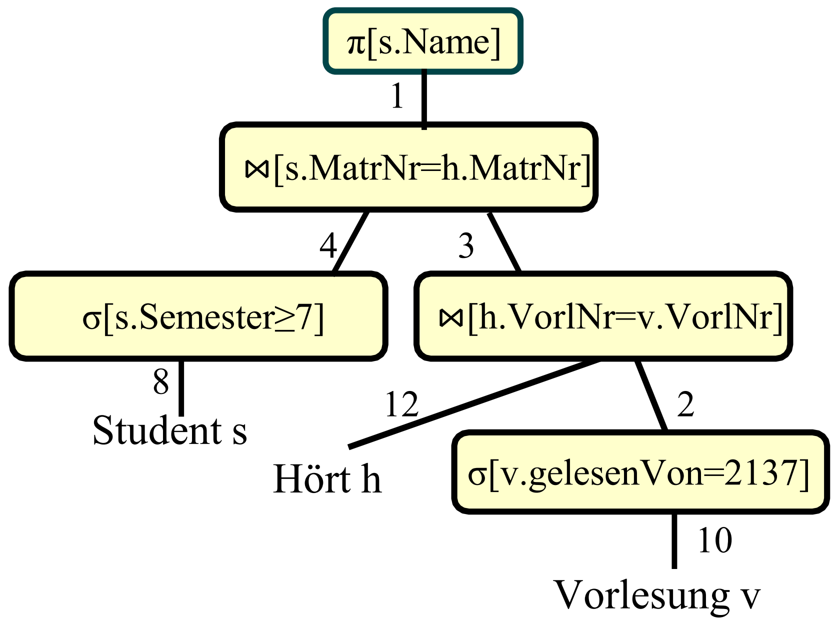




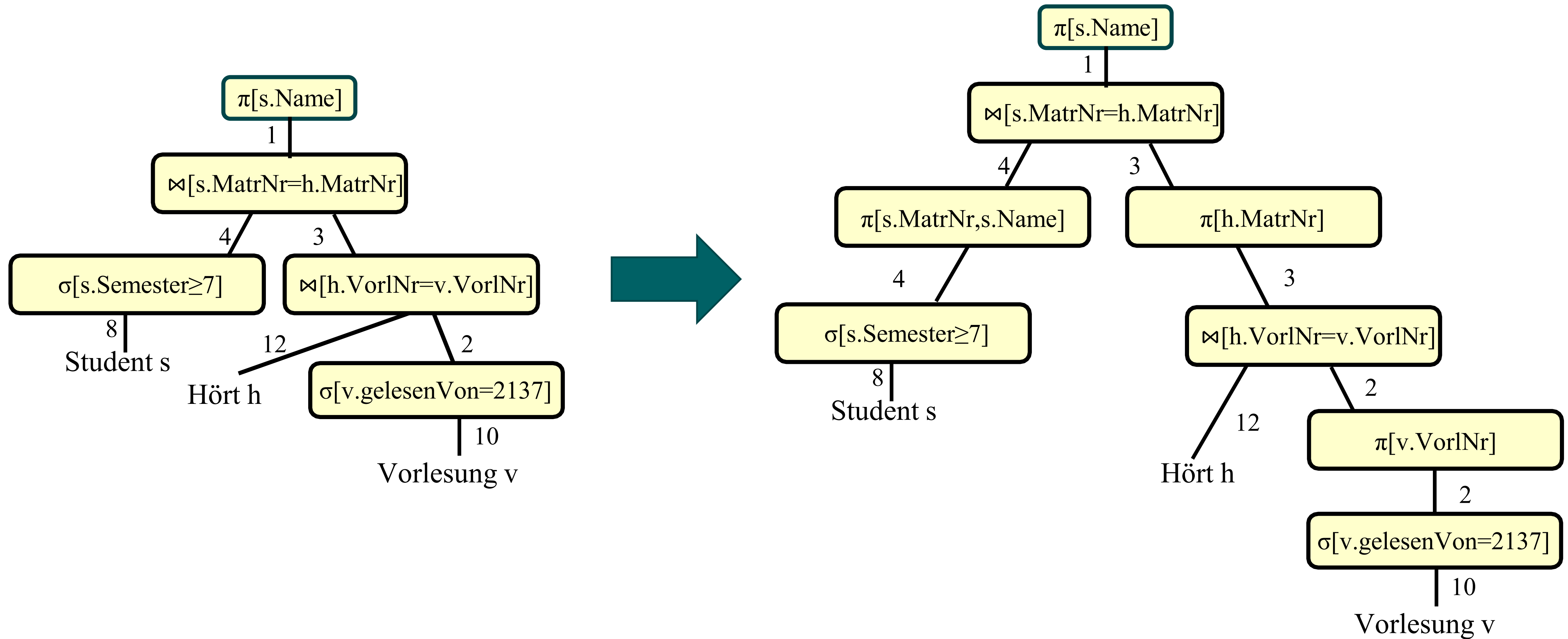
Anfrageoptimierung: Zusammenfassen zu Joins





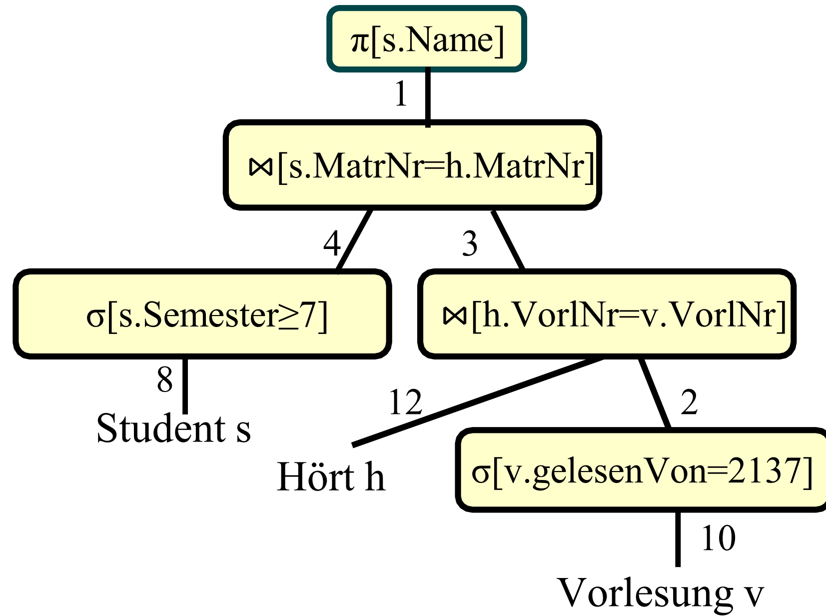


Logische Anfrageoptimierung: Einfügen Zusätzlicher Projektionen

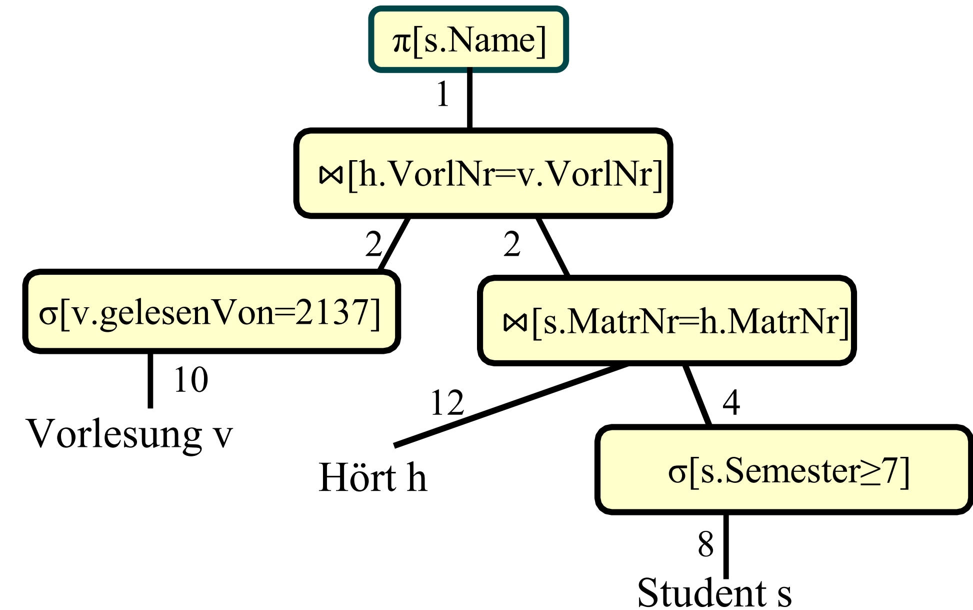


Kosten einer Anfrage: Wie wirkt sich die Join-Reihenfolge aus?

$Student \bowtie (Hört \bowtie Vorlesung))$



$Vorlesung \bowtie (Hört \bowtie Student))$



Betrachtete Tupel: $\Sigma = 40$ $\Sigma = 39$

Die beiden Pläne unterschieden sich nur in der Reihenfolge der ausgeführten Joins. Damit hängt die Performanz von Auswertungsplänen von der Datenverteilung der gespeicherten Informationen ab und der Reihenfolge der Joins ab.

Vom Algebraausdruck zum Ausführungsplan (QEP = Query Execution Plan)

- Ein Algebraausdruck ist noch kein Ausführungsplan
- Zusätzliche Entscheidungen müssen getroffen werden:
  - welche Indexe sollen verwendet werden, z.B. für Selektion oder Join?
  - welche Algorithmen sollen verwendet werde, z.B. Nested-Loop oder Hash Join?
- sollen Zwischenergebnisse materialisiert oder “pipelined” werden? usw.
  - Ausnutzung von Gleichungen der relationalen Algebra
  - Heuristiken auf Basis der Schemainformation
  - Keine Betrachtung der betroffenen Daten
- Für jeden Algebra Ausdruck können mehrere Ausführungspläne erzeugt werden.
- Alle Pläne ergeben dieselbe Relation, unterscheiden sich jedoch in der Ausführungszeit

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 Auswertungsplan 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

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - **Grundlagen: Regelbasierte Anfrageoptimierung**
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die 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 Optimierer*.
- Die Performanz von Auswertungsplänen hängt allerdings auch ganz wesentlich von der Datenverteilung der gespeicherten Informationen ab (siehe nächstes Teilkapitel)

15 Regeln fürs Leben (für die Anfrageoptimierung)

1. Selektionen können aufgebrochen werden: $\sigma_{B1 \wedge B2 \wedge \cdots \wedge Bn}(R) = \sigma_{B1}(\sigma_{B2}(\dots(\sigma_{Bn}(R))\dots))$
2. Selektionen sind kommutativ: $\sigma_{Bed_1}(\sigma_{Bed_2}(R)) = \sigma_{Bed_2}(\sigma_{Bed_1}(R))$
3. Geschachtelte Projektionen können eliminiert werden $\pi_X(\pi_Y(\dots(\pi_Z(R))\dots)) = \pi_X(R)$ (nur sinnvoll, falls $X \subseteq Y \subseteq \cdots \subseteq Z$)
4. Selektion und Projektion sind vertauschbar, falls die Projektion keine Attribute der Selektionsbedingung entfernt: $\pi_A(\sigma_B(R)) = \sigma_B(\pi_A(R))$ falls $attr(B) \subseteq A$
5. Das Kreuzprodukt und Joins sind kommutativ:
$$R \times S = S \times R$$
$$R \bowtie S = S \bowtie R$$
6. Selektion und Join (Kreuzprodukt) können vertauscht werden, falls die Selektion nur Attribute eines der beiden Join-Argumente verwendet: $\sigma_B(R \bowtie S) = \sigma_B(R) \bowtie S$ sowie $\sigma_B(R \times S) = \sigma_B(R) \times S$ falls $attr(B) \subseteq attr(R)$
7. Projektionen können teilweise in den Join verschoben werden:
$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A_1}(R) \bowtie_B \pi_{A_2}(S))$
wenn $A_1 = attr(R) \cap (A \cup attr(B))$ and $A_2 = attr(S) \cap (A \cup attr(B))$

15 Regeln fürs Leben (für die Anfrageoptimierung)

8. Kommutativität von Vereinigung und Schnitt:

$$R \cup S = S \cup R$$
$$R \cap S = S \cap R$$

9. Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ:

$$R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$$
$$R \cup (S \cup T) = (R \cup S) \cup T$$
$$R \cap (S \cap T) = (R \cap S) \cap T$$
$$R \times (S \times T) = (R \times S) \times T$$

10. Selektionen können mit Vereinigung, Schnitt und Differenz vertauscht werden:

$$\sigma_B(R \cup S) = \sigma_B(R) \cup \sigma_B(S)$$
$$\sigma_B(R \cap S) = \sigma_B(R) \cap \sigma_B(S)$$
$$\sigma_B(R - S) = \sigma_B(R) - \sigma_B(S)$$

11. Der Projektionsoperator kann mit der Vereinigung, aber nicht mit Schnitt oder Differenz vertauscht werden: $\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$

15 Regeln fürs Leben (für die Anfrageoptimierung)

12. Eine Selektion und ein Kreuzprodukt können zu einem Join zusammengefasst werden, wenn die Selektionsbedingung eine Joinbedingung ist (z.B. Equijoin): $\sigma_C\ (R\ \times\ S) =\ R \bowtie_C S$
13. Die Selektion braucht nur in das erste Argument der Mengendifferenz gezogen werden: Es gilt daher (neben Regel 10): $\sigma_B(R) - S$
14. Wenn in einer Vereinigung alle Attribute der Selektion aus einer Relation stammen, braucht die Selektion nur auf diese Relation angewandt werden: $\sigma_B(R \cap\ S)\ =\ \sigma_B(R) \cap S$
15. Wenn S die leere Relation ist, dann ist $R \cup\ S = R$. Wenn die Selektionsbedingung c für die gesamte Relation R erfüllt ist, dann ist $\sigma_B(R) = R$

Auch an Bedingungen in Selektionen oder Joins können Veränderungen vorgenommen werden, es gelten die üblichen Transformationsregeln, z.B.:

- Kommutativgesetze, Assoziativgesetze, z.B.: $B_1\ \wedge\ B_2\ =\ B_2\ \wedge\ B_1$
- Distributivgesetze, z.B.: $B_1\ \vee\ (B_2\ \wedge\ B_3)\ =\ (B_1 \vee\ B_2)\ \wedge\ (B_1\ \vee\ B_3)$
- De Morgan: $\neg(B_1\ \wedge\ B_2)\ =\ \neg B_1\ \vee\ \neg B_2$

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - **Grundlagen: Kostenbasierte Anfrageoptimierung**
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Kostenbasierte Optimierung

- Reihenfolge Selektions- und Projektionsoperationen durch regelbasierte Optimierung bestimmbar
- Die Reihenfolge der Join-Operationen nicht:
  - *Realen Kosten*
- Die kostenmodellbasierte Anfrageoptimierung behandelt deshalb insbesondere die Reihenfolge der Join-Operationen.
- Zwei interessante Fragestellungen:
  - Schätzung der Kosten eines bestimmten Query Execution Planes, 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
- Die *Selektivität* (sel) eines Anfrage schätzt die Anzahl der qualifizierenden Tupel relativ zur Gesamtanzahl der Tupel in der Relation.
- 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.

Kostenmodellbasierte Anfrageoptimierung

Selektivität (sel)

- Für die Selektion und den Join ist sie folgendermaßen definiert:

  - **Selektion** mit Bedingung *B*: $sel_B = \frac{|\sigma_B(R)|}{|R|}$

    (relativer Anteil der Tupel, die Bedingung *B* erfüllen)

  - **Join** von *R* und *S*: $sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$

    (Anteil relativ zur Kardinalität des Kreuzprodukts)

Schätzung der Zwischenergebnisse

Selektion mit Bedingung B : $sel_B = \frac{|\sigma_B(R)|}{|R|}$

Join von R und S : $sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$

Die Selektivität muss geschätzt werden, für Spezialfälle gibt es einfache Methoden:

- Die Selektivität von $\sigma_{R.A\,=\,c}$) ,Vergleich mit einer Konstante c) beträgt $1\ /\ |R|$, falls A ein *Schlüssel* ist.
- Falls A kein Schlüssel ist, aber Werte *gleichverteilt* sind, ist

  $sel(\sigma_{R.A=c}) = \frac{1}{I}$) I ist die Anzahl der unterschiedlichen Attributwerte)

- Besitzt bei einem Equijoin $R \bowtie_{R.A\,=\,S.B} S$ das Attribut A Schlüsseleigenschaft, kann die Größe des Join-Ergebnisses mit $|S|$ abgeschätzt werden, da jedes Tupel aus S maximal einen (genau einen bei *referential integrity*) Joinpartner findet. Also: $sel_{RS} = 1/|R|$

Schätzung der Zwischenergebnisse

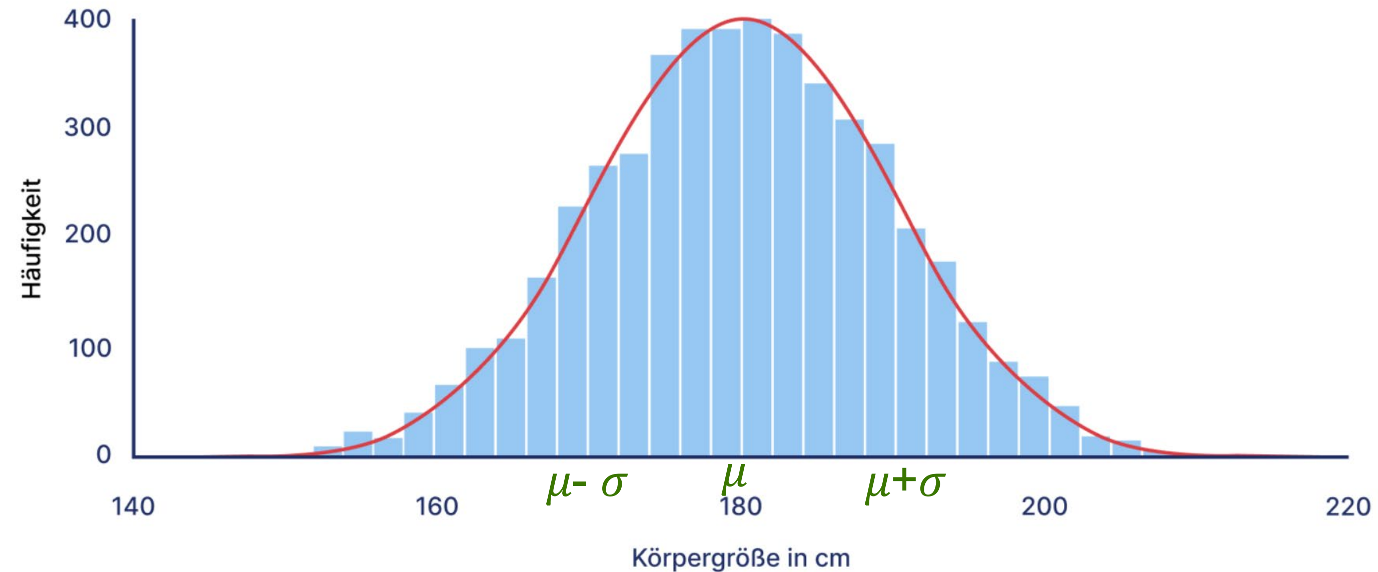
- Boolesche Verknüpfungen von Bedingungen (bei stochastischer Unabhängigkeit der Mengen):
  - logisches UND : $sel(\sigma_{B_1 \wedge B_2}) = sel(\sigma_{B_1}) \cdot sel(\sigma_{B_2})$
  - logisches ODER : $sel(\sigma_{B_1 \vee B_2}) = sel(\sigma_{B_1}) + sel(\sigma_{B_2}) - sel(\sigma_{B_1}) \cdot sel(\sigma_{B_2})$
  - logisches NICHT : $sel(\sigma_{\neg B_1}) = 1 - sel(\sigma_{B_1})$
- Nur in Spezialfällen kann man solche einfachen Rechenregeln anwenden. Im allgemeinen braucht man andere Methoden zur Selektivitätsabschätzung:

- Drei Grundsätzliche Arten von Schätzmethoden:
  1. Parametrische Verteilungen
  2. Histogramme
  3. Stichproben

Selektivität: Parametrische Verteilung

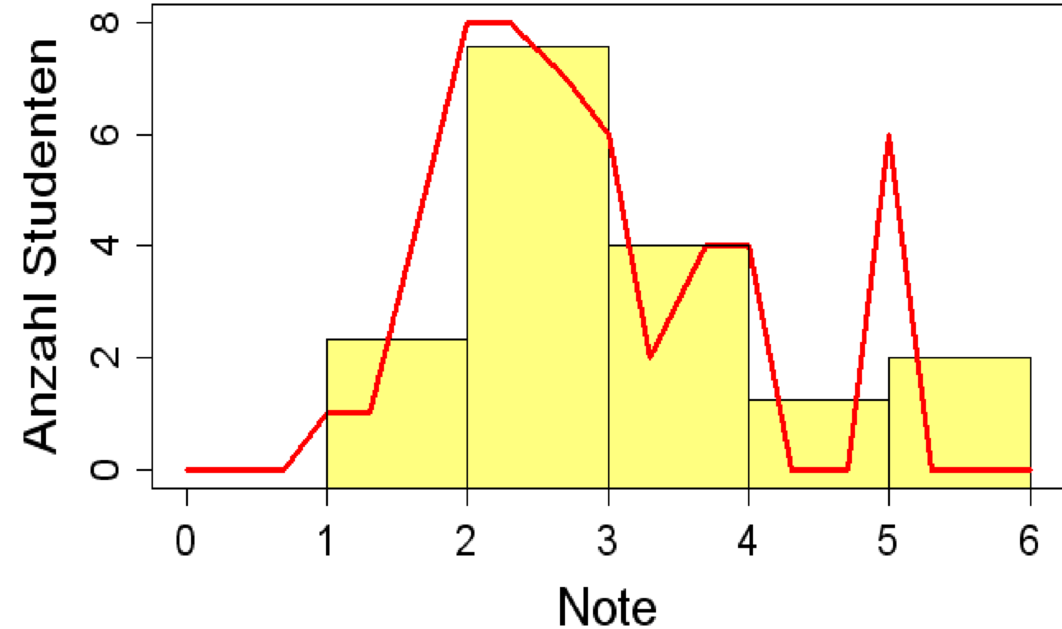
- Bestimme zu der vorhandenen Werteverteilung die Parameter einer Funktion so, dass die Verteilung möglichst gut angenähert wird.
- Z.B. Normalverteilung $N(\mu;\sigma\hat{}2)$ mit den Parametern μ (Mittelwert) und $\sigma\hat{}2$ (Varianz).

- Probleme: Wahl des Verteilungstyps (Normalverteilung, Exponentialverteilung…) und Wahl der Parameter, besonders bei mehrdimensionalen Anfragen (also z.B. bei Selektionen, die sich auf mehrere Attribute beziehen)



Selektivität: Histogramme

- Unterteile den Wertebereich des Attributs in Intervalle und zähle die Tupel, die in ein bestimmtes Intervall fallen
  - Equi-Width-Histograms: Intervalle gleicher Breite (starker Ungleichverteilung -> viele Unterteilungen in schwach besetzten Bereichen)
  - Equi-Depth-Histograms: Unterteilung so, dass in jedem Intervall gleich viele Tupel sind
- Equi-Depth-Histogramme = bessere Schätzgenauigkeit auf, haben aber auch einen höheren Verwaltungsaufwand.
- Z.B.: ORACLE benutzt Equi-Depth-Histogramme



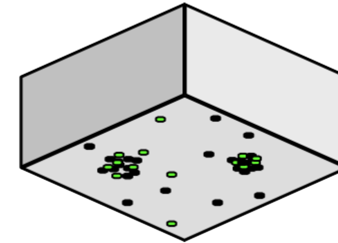
Selektivität: Stichproblem

Stichproben:

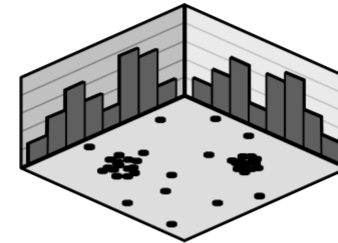
- Sehr einfaches Verfahren
- Ziehe eine zufällige Menge von n Tupeln aus einer Relation, und betrachte deren Verteilung als repräsentativ für die gesamte Relation.
- Problem der Größe des Stichprobenumfangs n:
  - n zu klein: Wenig repräsentative Stichprobe
  - n zu gross: Ziehen der Stichprobe erfordert zu viele „teure“ Zugriffe auf den Hintergrundspeicher

Selektivität: Anfragen über mehrere Attribute (mehr-dimensionale Anfragen)

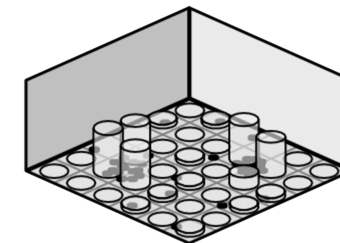
- Stichproblen:
  - Problem: Genauigkeit abhängig von der Samplegröße



- 1d Histogramme
  - Problem: Annahme der Unabhängigkeit zwischen den Attributen



- Multi-d Histogramme
  - Problem: Anzahl der Gridzellen steigt exponentiell mit d
- Parametrische Methoden
  - Problem: nur für max. 2-3 Attribute geeignet



5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
    - Einführung
    - Grundlagen: Regelbasierte Anfrageoptimierung
    - Grundlagen: Kostenbasierte Anfrageoptimierung
    - **Join-Reihenfolgen**
    - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
    - Indexstrukturen für eindimensionale Daten
    - Indexstrukturen für mehrdimensionale Daten

Join Reihenfolgen (1)

- Kostenbasierte Optimierung kann verwendet werden, um die beste Join Reihenfolge herauszufinden.

- Join Reihenfolgen der Relationen entstehen durch:
  - Assoziativgesetz: $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
  - Kommutativgesetz: $R \bowtie S = S \bowtie R$

- Die Join Reihenfolge hat große Auswirkung auf Effizienz:
  - Größe der Zwischenergebnisse
  - Auswahlmöglichkeit der Algorithmen (z.B. vorhandene Indexe wiederverwenden)

Join Reihenfolgen (2)

- Wieviele Reihenfolgen gibt es für: $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$?
- Assoziativgesetz:
  - Operatorbaum: es gibt C_{m-1} volle binäre Bäume mit m Blättern (es gibt C_{m-1} Klammerungen von m Operanden)
  - dabei ist $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{2n!}{(n+1)!n!}$, $n > 0$ die Catalan-Zahl:
- Kommutativgesetz:
  - Blätter des Operatorbaums sind die Relationen $R_1, R_{2,} \ldots, R_m$
  - für jeden Operatorbaum gibt es $\mathrm{m}!$ Permutationen
- Anzahl der Join-Reihenfolgen für m Relationen:

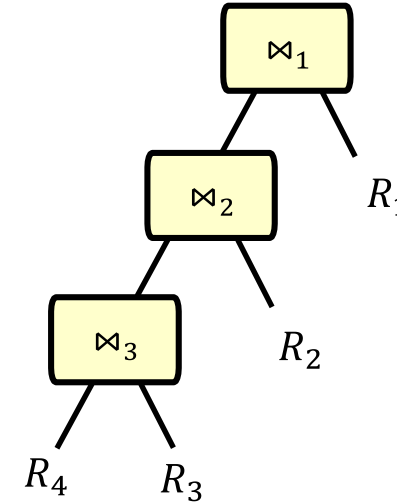
$$m!\, C_{m-1} = \frac{2(m-1)!}{(m-1)!}$$

Join Reihenfolgen (3)

- Anzahl aller Join-Reihenfolgen wächst sehr schnell an:
  - $m = 3$: 12 Reihenfolgen
  - $m = 7$: 665.280 Reihenfolgen
  - $m = 10$: > 17.6 Milliarden Reihenfolgen
- Dynamic Programming Ansatz:
  - Laufzeit Komplexität: $\mathrm{O}(3^{\mathrm{m}})$
  - Speicher Komplexität: $\mathrm{O}(2^{\mathrm{m}})$
- Beispiel: $m = 10$
  - Anzahl der Join-Reihenfolgen: 17.6 × 109
  - Dynamic Programming: $O(3^m)$ = 59049
- Trotz Dynamic Programming bleibt Aufzählung der Join-Reihenfolgen teuer.

Join Reihenfolgen (4)

- Beschränkung auf Left-deep Join Reihenfolgen
  - rechter Join-Operator ist immer eine Relation (nicht Join-Ergebnis)
  - dadurch ergeben sich sog. left-deep Operatorbäume (im Gegensatz zu "bushy", wenn alle Operatorbäume erlaubt sind)
- Anzahl der left-deep Join Reihenfolgen für m Relationen: $O(m!)$
- Dynamic Programming: Laufzeit $O(m!)$
- Vergleich für 10 Relationen:



	bushy	left-deep	m = 10	bushy	left-deep
#Baumformen	1	C_{m-1}		1	4.862
#Join Reihenfolgen	$\frac{2(m-1)!}{(m-1)!}$	m!		1.76×10^{10}	3.63×10^{6}
Dynamic Programming	$O(3^m)$	$O(m\ 2^m)$		59.049	10.240

- Aber:

  keine Garantie auf optimale Reihenfolge

Greedy Algorithmus für Join Reihenfolgen (1)

- Ansatz: In jedem Schritt wird der Join mit dem kleinsten Zwischenergebnis verwendet.
- Überblick: Greedy Algorithms für Join Reihenfolge:
  - nur left-deep Join Reihenfolgen werden betrachtet
  - Relationen-Paar mit dem kleinsten Join Ergebnis kommt zuerst dran
  - Dann wird immer die Relation ausgewählt, die mit dem vorhanden Operatorbaum das kleinste Join-Ergebnis erzeugt

- Algorithmus: Join Reihenfolge von $S = \{R_1, \ldots, R_m\}$
  1. $O \leftarrow R_i \bowtie R_j$, sodass $|R_i \bowtie R_j|$ minimal ist $(i \neq j)$
  2. $S = S - \{R_i, R_j\}$
  3. **while** $S \neq \emptyset$ **do**
     a) wähle $R_i \in S$ sodass $|O \bowtie R_i|$ minimal ist
     b) $O \leftarrow O \bowtie R_i$
     c) $S = S - \{R_i\}$
  4. **return** Operatorbaum O

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - **Ein Algorithmus für die Anfrageoptimierung**
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Ein Algorithmus für die Anfrageoptimierung

1. Zerlege komplexe Selektions -Operationen in eine Kaskade einfacher Selektionen (Regel 1).
2. Verschiebe Selektionen so weit wie möglich den Abfragebaum hinunter (Regeln 2, 4, 6, 10, 13, 14).
3. Ordne Blattknoten nach restriktiven Selektionen und vermeide Kreuzprodukte (Regeln 5 und 9).
4. Kombiniere Kreuzprodukte und Selektionen zu JOINs, wenn möglich (Regel 12).
5. Verschiebe und teile Projektions-Operationen, um nur benötigte Attribute zu behalten (Regeln 3, 4, 7, 11).
6. Identifiziere Teilbäume, die von einem einzigen Auswertungs-Algorithmus ausgeführt werden können (siehe nächsten Abschnitt in der Vorlesung).

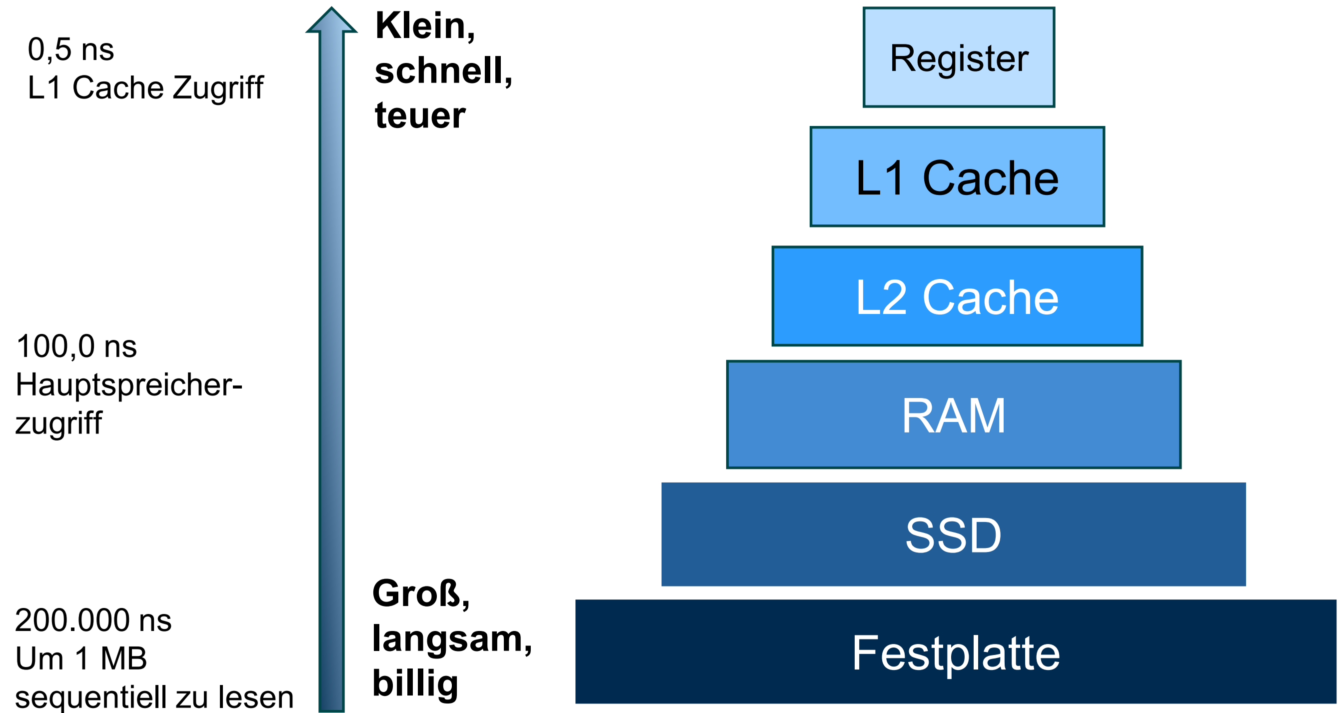
Nach Elmasri & Navathe, Fundamentals of Database Systems, 7th Edition)

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. **Algorithmen für Basisoperationen**
3. Indexstrukturen
   - Indexstrukturen für eindimensionale Daten
   - Indexstrukturen für mehrdimensionale Daten

Aufgabe: Finde geeignete Algorithmen & Datenstrukturen für einzelne Operationen
Herausforderungen: Physische Datenorganisation (Cache, RAM, Festplatte, SSD)

Vereinfachung:
- CPU-beschränkt: Das System aus CPU, Arbeitsspeicher und Bus bildet den Hauptengpass
- I/O-beschränkt: Hintergrundspeicher und I/O bilden den Hauptengpass
- Parallelität zwischen CPU und Hintergrundspeicher: Algorithmen zur Anfragebearbeitung “multithreaded” implementiert



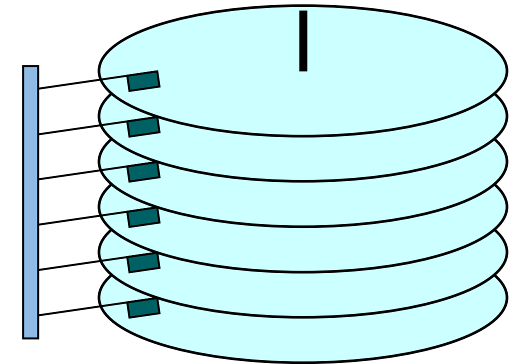
Optimierungsziele

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

Speicherhierarchie

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

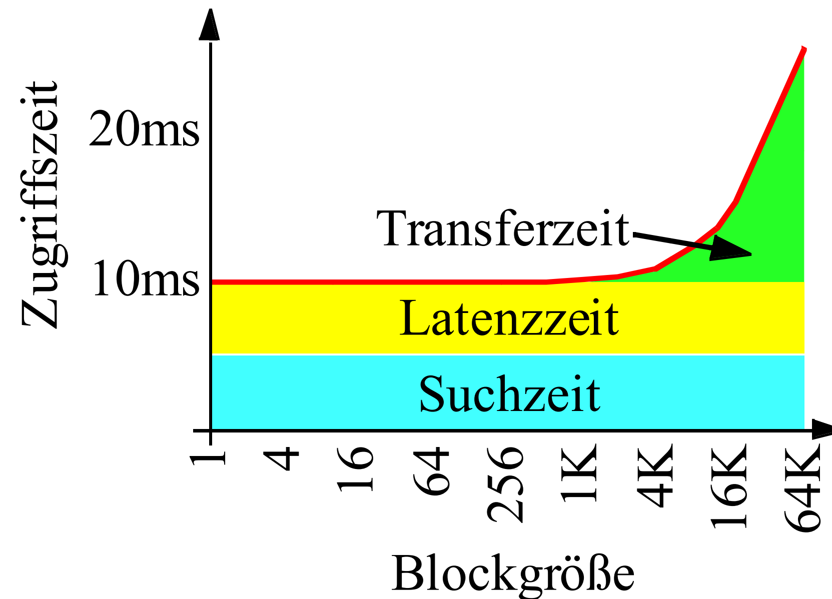


- Festplatten/SSD als gebräuchliche Sekundärspeicher
  - Platten: Übereinanderliegende Platten mit magnetischen / optischen Oberflächen
    - Zur Adressierung sind die Platten in Spuren und Sektoren eingeteilt
    - Mechanische Bewegungen
      - Platten rotieren um gemeinsame Achse
      - Schreib-Lese-Köpfe werden zwischen den Platten synchron in radialer Richtung bewegt
  - SSD:
    - Unterschied zu magnetischen Festplatten: SSDs haben Zellenorganisation
    - Schnelle wahlfreier Zugriff bei SSDs
    - "Wear Leveling" zur Verteilung von Schreibvorgängen und Vermeidung von Verschleiß
    - Feines blockweises lesen (4-8kb Block), grobes schreiben („indirection unit“, z.B., 16 KB)
    - SSDs sind teurer als Festplatten

Blockweiser Zugriff auf Festplatten

- Zugriffszeit bei Festplatten (nicht für SSDs)
  - Armpositionierung: Suchzeit (ca. 5 ms)
  - Rotation bis Blockanfang: Latenzzeit (ca. 5 ms)
  - Datenübertragung: Transferzeit (ms/MB)

- Blockorientierter Zugriff (gilt auch für SSDs)
  - 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

- Wir schauen uns im folgenden die Join-Operation anhand folgenden Beispiels an:
- Zur Vereinfachung: equi join

```
select *
   from R r, S s
   where r.A = s.B
```

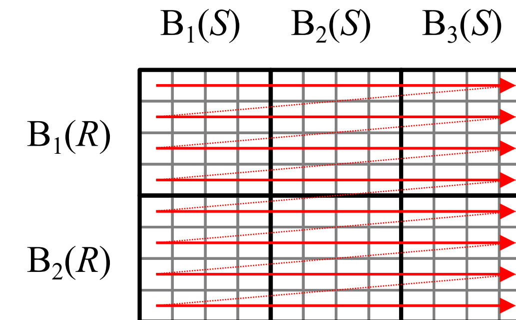
  also $R \bowtie_{A=B} S$

- Die Relationen R und S sind in Blöcken $B_i(R)$ und $B_i(S)$ auf dem Hintergrundspeicher abgelegt.

- 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

Matrixnotation (S 3 Blöcke, R 2 Blöcke)



for each Tupel $r \in R$ do

  for each Tupel $s \in S$ do

    if $r(A) = s(A)$ then

      $Result := Result \cup \{(r - r(A)) \otimes s\}$

- Der einfache Nested Loop Join entspricht der Bildung des kartesischen Produktes in kanonischer Ordnung mit anschließender Selektion.
- Die Laufzeit ist $|S| \cdot |R|$. Relation S wird $|R|$ mal eingelesen: Performanz ist deshalb inakzeptabel
- S wird als *innere* Relation und R als *äußere* Relation bezeichnet

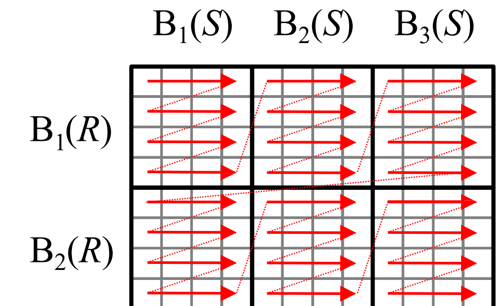
- Matrixdarstellung der Joinoperation (stellt Reihenfolge von Block- und Tupelpaarungen dar)
- Nested Loop Joins sind geeignet für alle Join-Prädikate θ (’$=$‘, ’$<$‘, ’$>$‘, ’$\leq$‘, usw.)

Nested Block Loop Join

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

for each Block B_R in R do {
    lade Block B_R;
    for each Block B_S in S do {
        lade Block B_S;
        for each Tupel r in B_R do
            for each Tupel s in B_S do
                if $r(A) = s(A)$ then
                    $Result := Result \cup \{(r - r(A)) \otimes s\}$
    }
}

Matrixnotation



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	$B_S(3)$

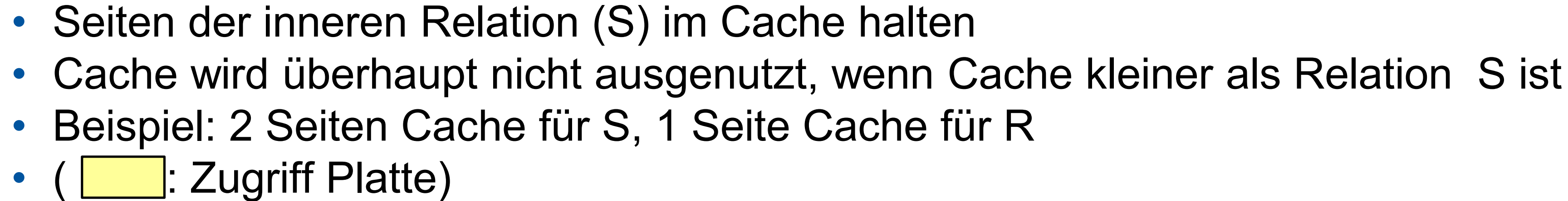
Relation R

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

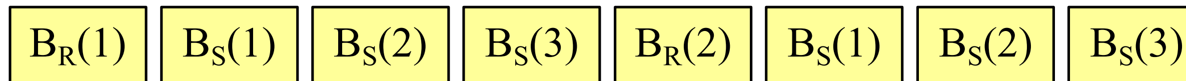
- Blockzugriffe $= b_R + b_S \cdot b_R$ =8, (b_R = Anzahl der Blöcke der Relation R ist)
- Empfehlung: Die kleinere Relation sollte die äußere sein (ohne Cache).
- Wenn ein Cache (= Hauptspeicherplatz für viele Blöcke gleichzeitig) zur Verfügung steht, kann u.U. die kleinere Relation ganz im Hauptspeicher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

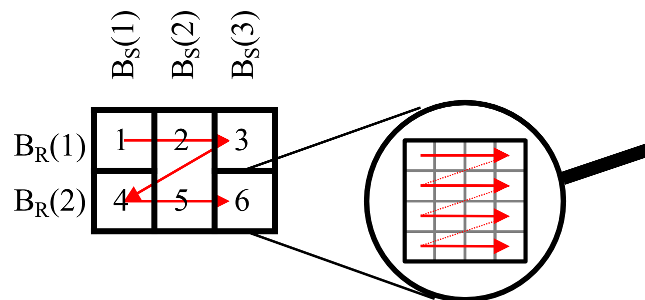
Cache Strategien für Nested-Block-Loop-Join

Strategíe 1



- Seiten der inneren Relation (S) im Cache halten
- Cache wird überhaupt nicht ausgenutzt, wenn Cache kleiner als Relation S ist
- Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R
- ([ ]: Zugriff Platte)

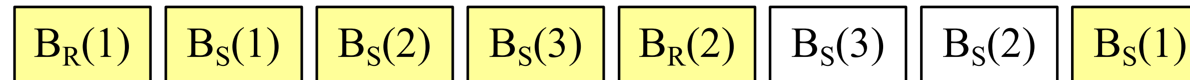


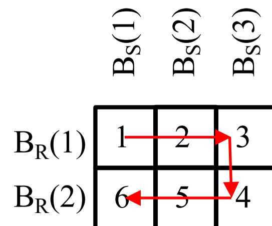


Cache Strategien für Nested-Block-Loop-Join

Strategie 2

- Seiten der inneren Relation (S) im Cache, aber innere Relation jedes zweite mal rückwärts
- Pro Durchlauf der äußeren Schleife werden (|C|-1) Blockzugriffe
- eingespart (ab 2. Durchlauf)
- |C| = Anzahl Blöcke, die in den Cache passen, ein Cache-Block wird jeweils für äußere Relation (R) benötigt
- Blockzugriffe: BR+ BR · (BS - |C| + 1) + |C| - 1
- Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R





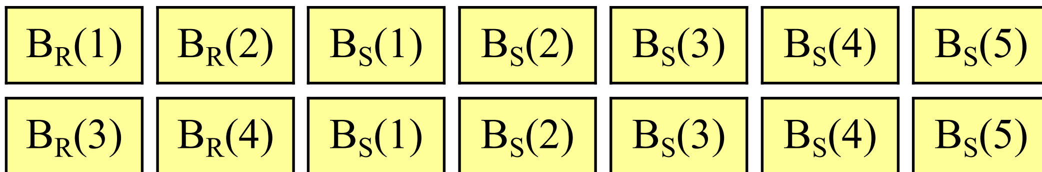
Cache Strategien für Nested-Block-Loop-Join

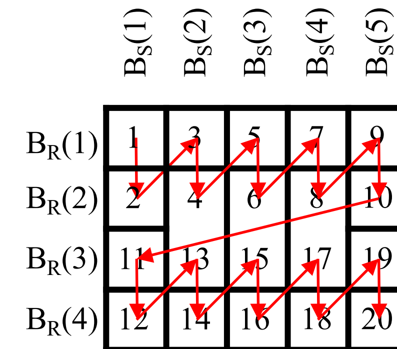
Strategie 3

- |C|-1 Blöcke der äußeren Relation werden in den Cache eingelesen, zu jedem Block der inneren Relation werden diese Blöcke gejoint
- Blockzugriffe:

$$B_R + B_S \cdot \left\lceil \frac{B_R}{|C| - 1} \right\rceil$$

- Beispiel: 2 Seiten Cache für R, 1 Seite Cache für S





Datenbanken und Informationssysteme 2023
Prof. Dr. Stefan Decker
Lehrstuhl Informatik 5

Informatik 5 - Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

Cache Strategien für Nested-Block-Loop-Join

Strategie 3 - Algorithmus

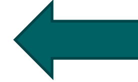
for $i := 1$ to B_R step $|C| - 1$ do
  Lade Block $B_R(i) \ldots B_R(i + |C| - 2)$
  for each Block $B_S \in S$ do
    Lade Block B_S
    for each Tupel $r \in B_R(i) \ldots B_R(i + |C| - 2)$ do
      for each Tupel $s \in B_S$ do
        if $r.A = s.B$ then
          $result := result \cup \{(r - r(A)) \otimes s\}$

- Leistung:
  - |R|*|S| Vergleiche von Tupel (ist nur bei schlechter Selektivität gerechtfertigt)
  - Effizienteste Ausführung von θ–Joins mit θ ≠ ‘=‘ (also allen Joins außer Equi-Joins)

Sort-Merge Join: Beispiel – Annahme R und S sind sortiert

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

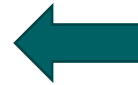
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Gehaltsgruppe	Gehalt	Angestellter

Sort-Merge Join: Beispiel

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Relation S

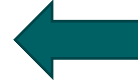
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

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

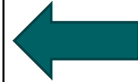
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

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

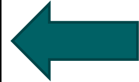
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

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

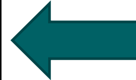
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

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

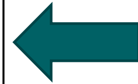
Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

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

Zweistufiger Algorithmus

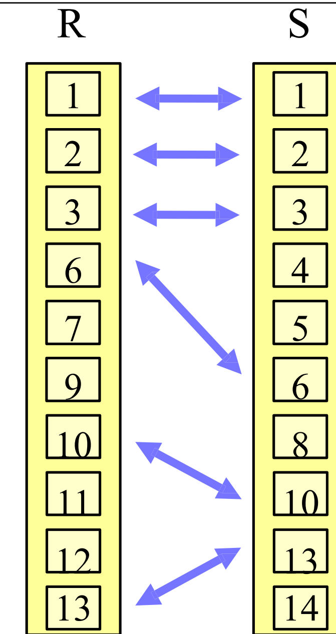
1.Schritt:

sortiere R bzgl. Attribut A
sortiere S bzgl. Attribut B

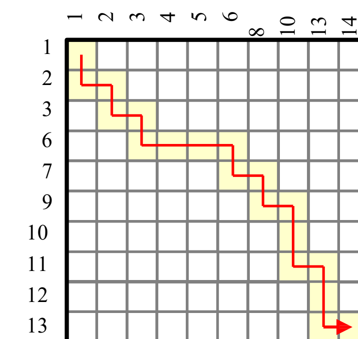
2.Schritt:

```
Result :=∅;
s:=erstes Tuple in S;
r:=erstes Tuple in R;
while (r≠null ∧s≠null)
        while (s≠null∧ r(A) ≥s(A))
                if ( r(A)=s(A)) Result :=Result∪{(r−r(A)⊗s}
                s:=nächstes Tuple in S nach s;
        r:=nächstes Tuple in R nach r;

return Result;
```



Matrixnotation



- Achtung: Dieser Algorithmus funktioniert nur, falls R auf dem Joinattribut A keine Duplikate enthalten.
- Wie muss der Algorithmus erweitert werden um Duplikate zu erfassen?

Sort-Merge Join (Duplikate möglich in beiden Relationen)

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

$Result := \emptyset;$
$Rvalue, Rtuple, Rgroup := \text{nextGroup}(R, \text{erstes Tupel in R});$
$Svalue, Stuple, Sgroup := \text{nextGroup}(S, \text{erstes Tupel in S});$
$\text{while } (Rgroup \neq \emptyset \ \land \text{Sgroup} \neq \emptyset)$
$\quad \text{if } (Rvalue = Svalue)$
$\quad\quad Result = Result \cup (Rgroup \setminus A) \times Sgroup)$
$\quad\quad Rvalue, Rgroup := \text{nextGroup}(R, Rtuple);$
$\quad\quad Svalue, Sgroup := \text{nextGroup}(S, Stuple);$
$\quad \text{else if } (Rvalue < Svalue)$
$\quad\quad\quad Rvalue, Rtuple, Rgroup := \text{nextGroup}(R, Rtuple);$
$\quad\quad \text{else } Svalue, Stuple, Sgroup := \text{nextGroup}(S, Stuple);$
$\text{return } Result;$

Hilfsfunktion zur Bestimmung der nächsten Gruppe:

function $value, t, group$ $\text{nextGroup}(T, tuple)$
$\quad group := \emptyset;$
$\quad t := tuple;$
$\quad value := t\ (A)\ ;$
$\quad \text{while } (t \neq null \land \text{value} = t\ (A))$
$\quad\quad group := group \cup \{t\};$
$\quad\quad t := \text{nächstes Tuple in } T \text{ nach } t;$
$\quad$ return $value, t, group;$

Analyse:

- Sortieren der Relationen kostet $O(|R| \log |R| + |S| \log |S|)$
- Sortieren ist nicht notwendig, wenn bereits Index existiert
- Wenn beide Relationen keine Duplikate in den Join-Attributen habe wird jede Relation genau einmal durchlaufen: $O(|R| + |S|)$ Vergleiche
- Bei Duplikaten in beiden Relationen bestimmt das kartesische Produkt die worst case Performanz.
- Weitere Optimierungen: Sortierung und Merging kombinieren, Blöcke berücksichtigen

Index Join

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

  for each Tupel r in R do

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

  $$Result := Result \cup ((r - r(A)) \otimes s)$$

- Kostenschätzung:
  - $|R| \cdot cost\left(Indexzugriff\ auf\ S(\hat{A})\right)$
  - Also z.B. $|R| \cdot \log|S|$ im Fall der Verwendung von B-Bäumen (siehe Indexstrukturen)

Einfacher Hash-Join

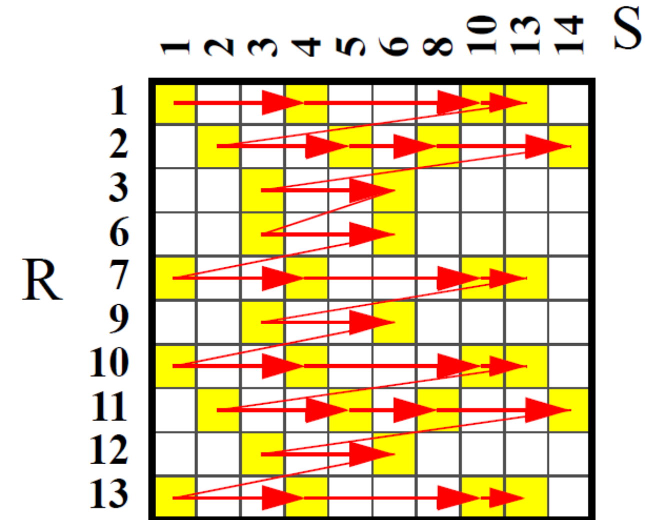
Weitere Idee: Erzeugung eines Index zur Laufzeit

Reduktion des CPU-Aufwandes bei der Join-Berechnung:

- Der Join-Partner eines R-Tupels wird gezielt mit Hilfe eines Hash-Verfahrens gesucht, statt sequentiell mit jedem Tupel der S-Relation zu vergleichen
- Zu diesem Zweck wird die S-Relation gehasht, d.h. zu allen Tupeln der Hash-Key bestimmt und die Tupel in einer Tabelle unter diesem Key eingetragen
- Nicht alle S-Tupel, die den passenden Hash-Key haben, sind Join-Partner eines R-Tupels, aber alle Join-Partner haben denselben Hashkey (**Bedingung der Hashfunktion!**)
- Im Idealfall soll der Join im Hauptspeicher ablaufen: die Hashtabelle soll für die kleinere Relation erzeugt werden.
- Hash-Join Verfahren können nur für Equi-Join und Natürlichen Joint effizient genutzt werden.

Einfacher Hash-Join

```
for each Tupel s in S do {            /* Erzeugen der Hashtabelle HT */
        berechne adr = Hash(s);
        speichere das Tupel s in HT[adr]
}
for each Tupel r in R do {            /* Prüfen in der Hashtabelle HT */
        berechne adr = Hash(r);
        for each Tupel s in HT[adr] do {
                if r(A) = s(A) then
                        Result := Result ∪ {(r − r(A) ⊗ s}
        }
}
```



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

Leistung

- hängt stark ab von der Güte der Hashfunktion: O(|R| + |S|) im Idealfall
- verschlechtert sich, wenn Werte ungleichmäßig belegt sind
- Modifikation ist notwendig, wenn Hauptspeicher zu klein (kleiner als R)

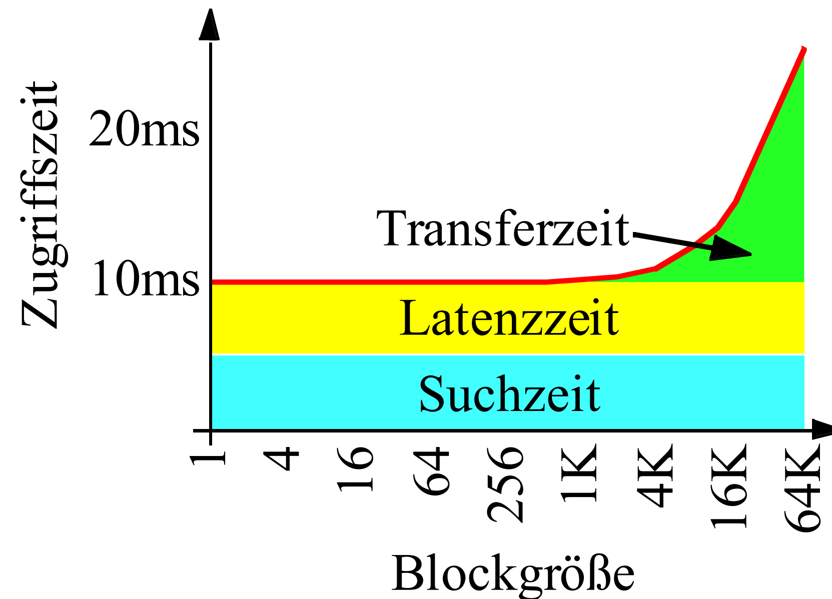
5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - **Indexstrukturen für eindimensionale Daten**
   - Indexstrukturen für mehrdimensionale Daten

Blockweiser Zugriff auf Festplatten

- Zugriffszeit bei Festplatten (nicht für SSDs)
  - Armpositionierung: Suchzeit (ca. 5 ms)
  - Rotation bis Blockanfang: Latenzzeit (ca. 5 ms)
  - Datenübertragung: Transferzeit (ms/MB)

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



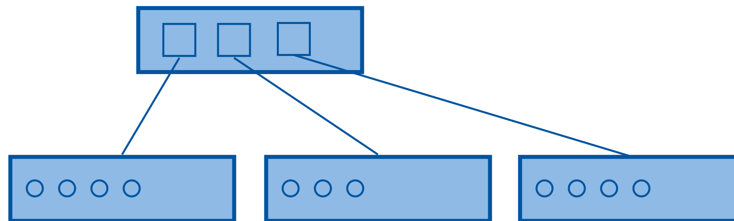
Verwendungsarten von Indexen

- 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

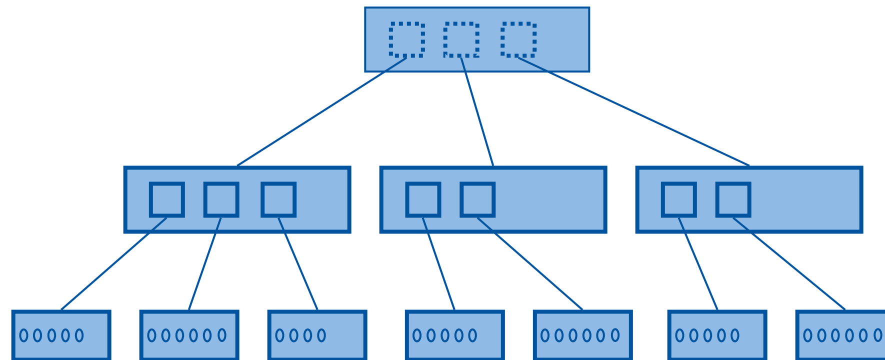
- Block- oder seitenweise Speicherung
  - Speicherung vieler Datensätze auf ein- und derselben Seite
  - Für effiziente Suche: Speichere ähnliche Werte auf derselben Seite
  - Bei Überlauf einer Seite: Aufspaltung auf mehrere Seiten

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



Mehrstufige Hierarchien (Bäume)

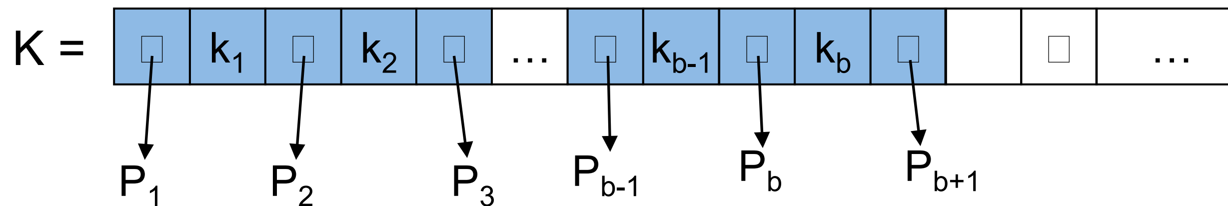
- Rekursive Aufspaltung liefert Baumstruktur



- 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

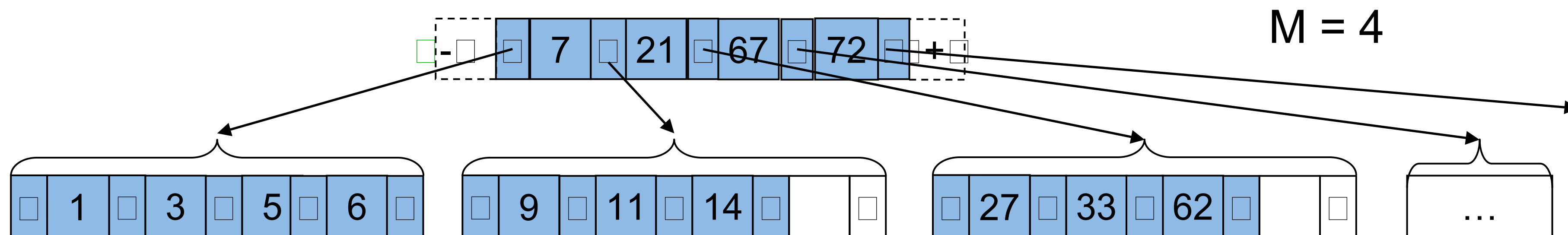
- Charakterisierung
  - Knoten haben bis zu M+1 viele Nachfolger
  - Blockorientierte Speicherung der Bäume: Speichere Knoten auf Plattenseiten
  - Damit ergibt sich M aus Seitengröße und Größe der Datensätze + Zeiger



- 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

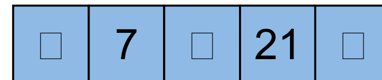
- Definition:
  Ein B-Baum ist ein (M+1)-Wege Suchbaum (für eine gerade Zahl M)

- 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)

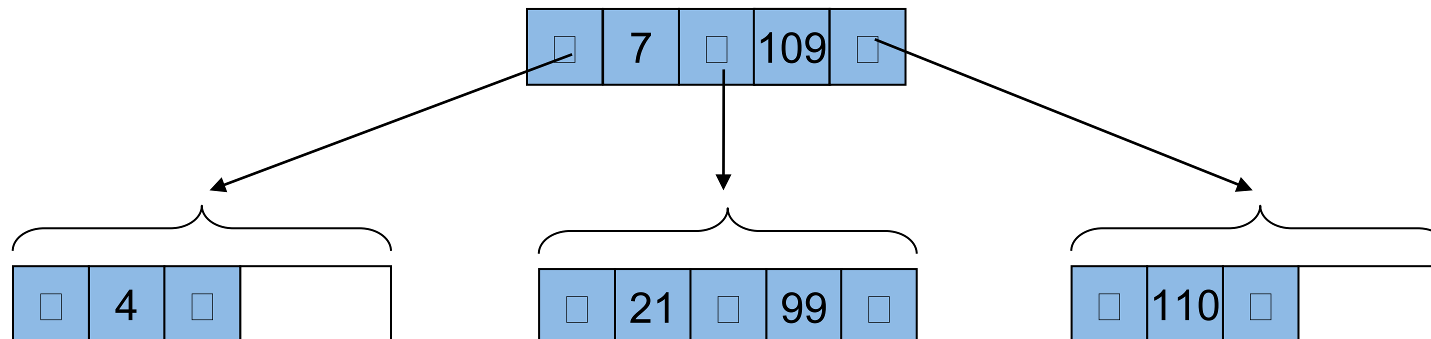
- Bedeutung des „B“:
  - **B**alanced Tree, **B**locked Tree (technische Beschreibung)
  - **B**ushy Tree, **B**road Tree (Hinweis auf hohen Verzweigungsgrad)
  - Prof. Dr. Rudolf **B**ayer (mit Ed McCreight Erfinder der B-Bäume)
  - The **B**oeing Company (Bayer arbeitete in deren Forschungslabor)
  - **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

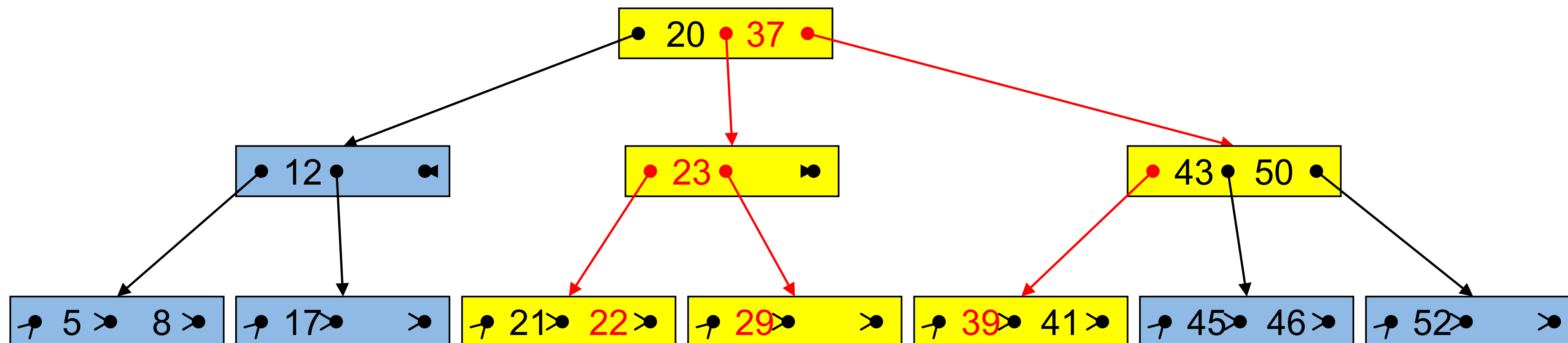
- Komplexität der Suche
  - In einem B-Baum der Höhe h werden maximal h viele Knoten besucht
  - Die Höhe eines B-Baums mit N Objekten ist maximal $h = \log_m N$ (s.später)
  - Die binäre Suche auf einem Knoten benötigt maximal $\log_2 M$ Vergleiche
  - Anzahl der Vergleiche insgesamt: höchstens $\log_2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

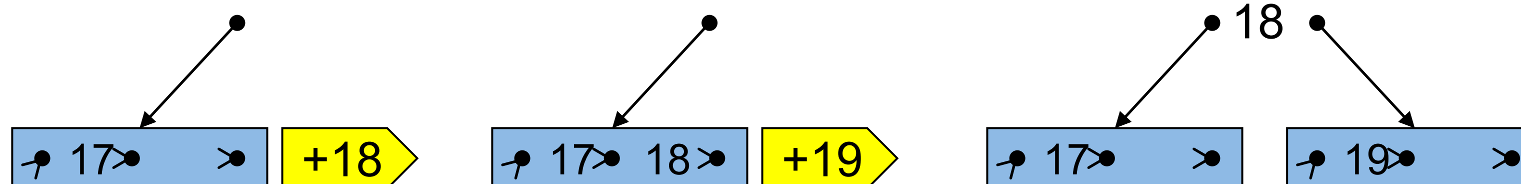


Einfügen im B-Baum

- 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

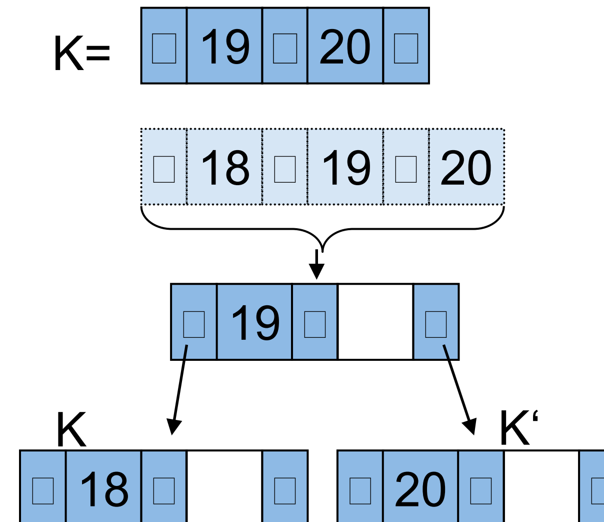


Einfügen im B-Baum: Split

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

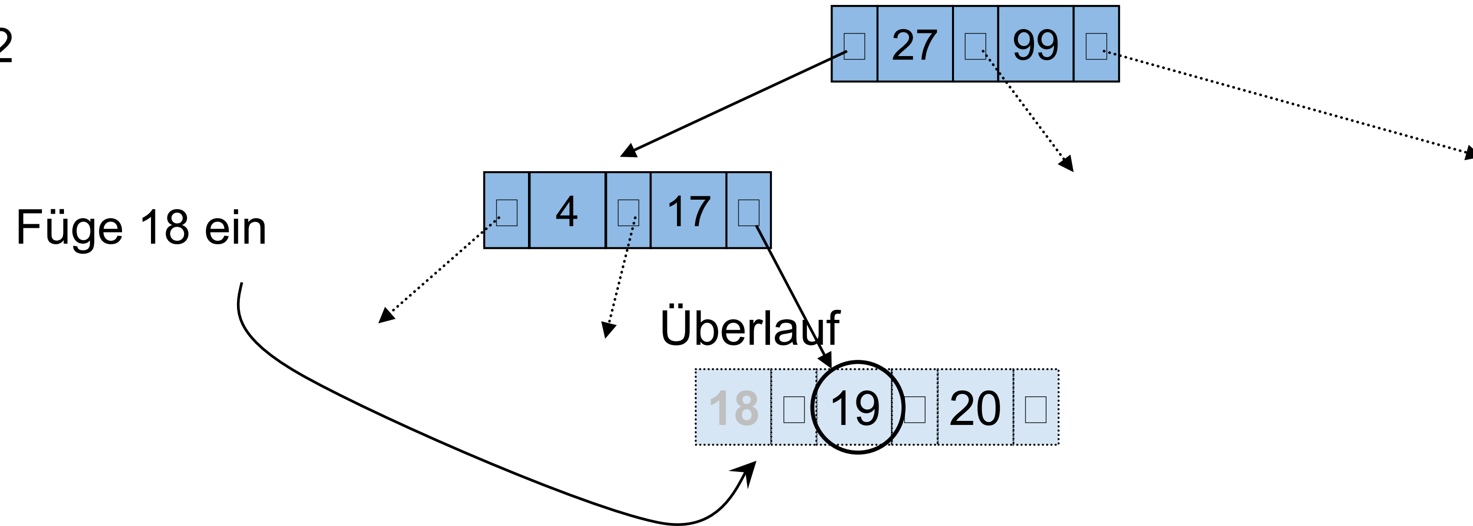
- Falls Vorgänger nicht existiert
  - Knoten war die Wurzel: Schaffe neue Wurzel
  - Die Höhe wächst um Eins

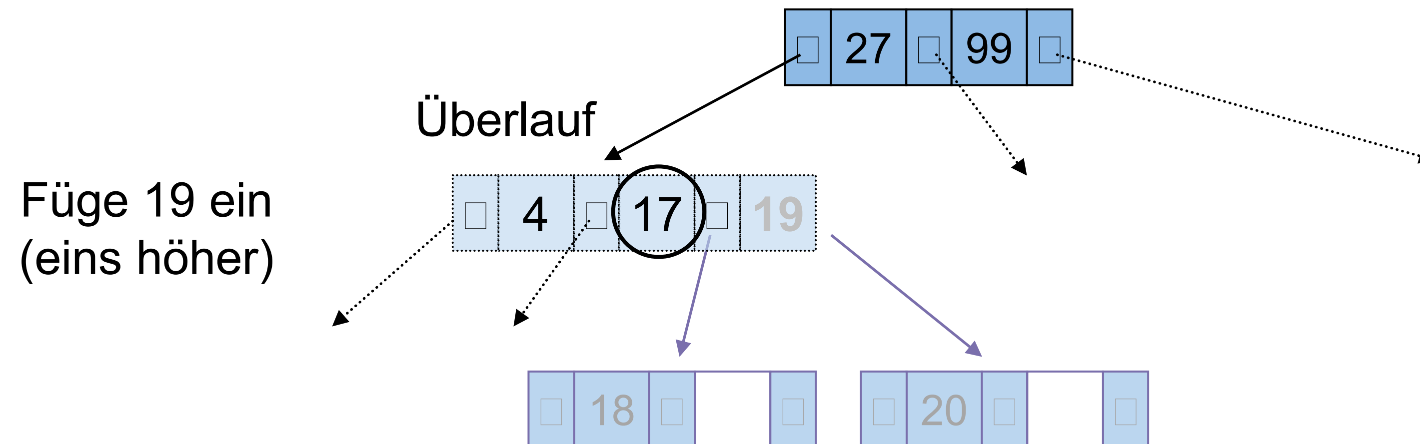
- Falls Vorgänger überläuft
  - Wende denselben Split-Algorithmus an
  - Split kann rekursiv bis zur Wurzel laufen
  - Komplexität des Einfügens: $O(\log_m N)$



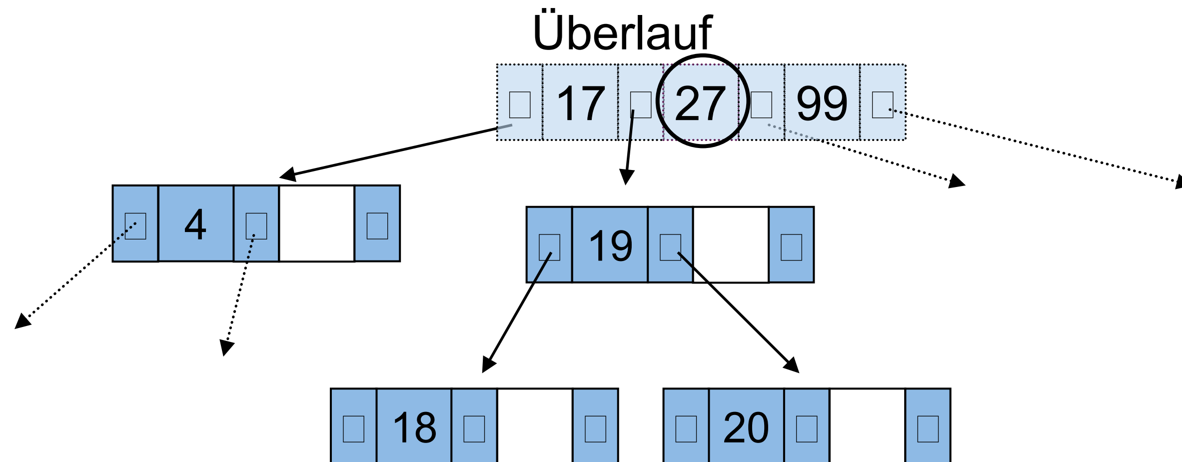
Beispiel für Einfügen im B-Baum

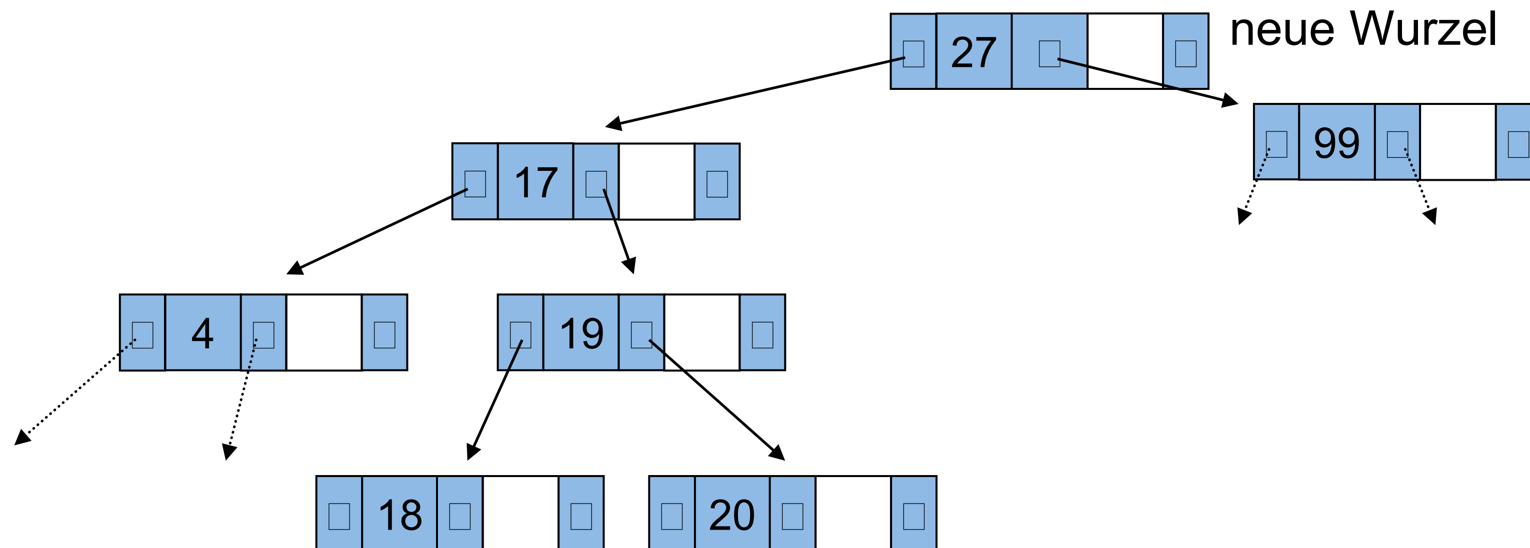
- Beispiel: M = 2





Beispiel (2)





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:

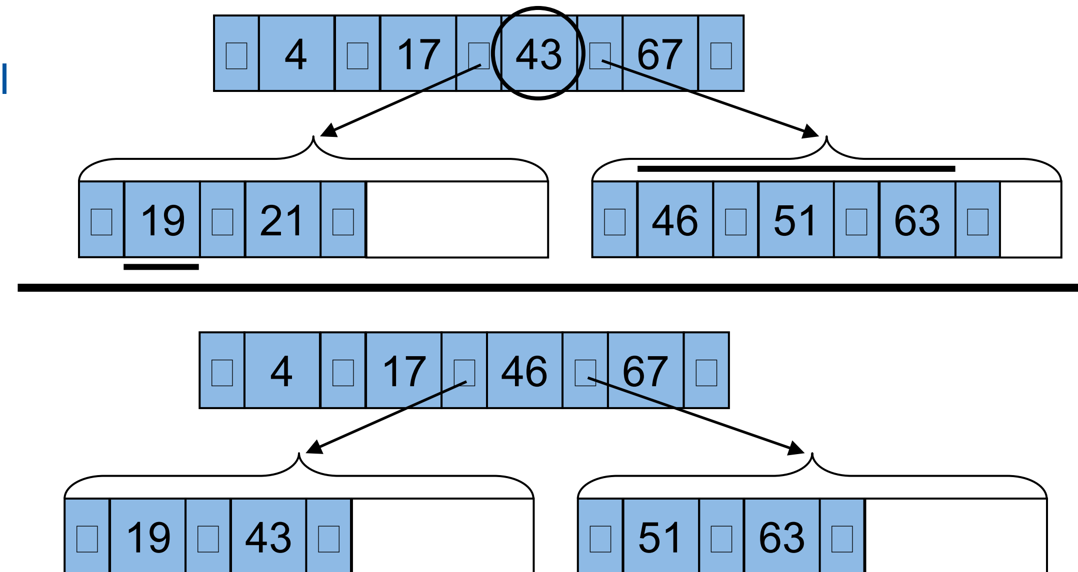
- Falls es einen Nachbarknoten K‘ = (o‘$_1$,…,o‘$_n$) mit **mehr** als m = M/2 Schlüsseln gibt, kann ein Ausgleich durchgeführt werden:
  - O.B.d.A. sei K‘ rechts von K, und p der Trennschlüssel im Vorgänger
  - Verteile die Schlüssel o_1 … o_b, p, o‘$_1$ … o‘$_n$ auf die Knoten K und K‘, und ersetze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



Informatik 5 - Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

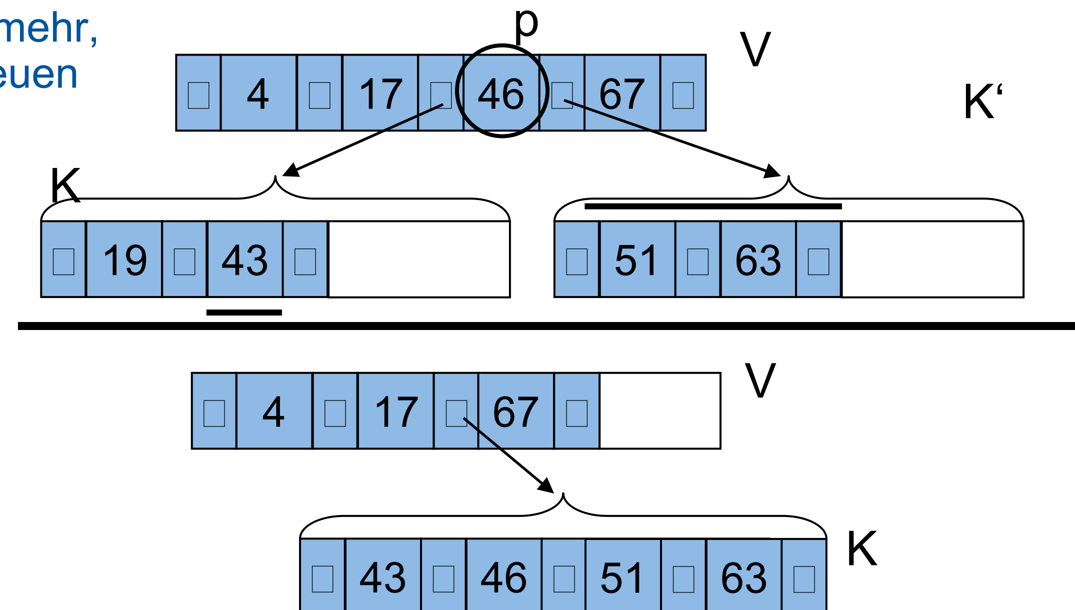
Löschen im B-Baum (Verschmelzen)

Falls es keinen Nachbarknoten mit mehr als m = $M/2$ Elementen gibt, so existiert mindestens ein Nachbarknoten K' = $(o'_1, \ldots, o'_m)$ 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:

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

  - Maximal:

$$N \leq M + (M+1) \cdot M + (M+1)^2 \cdot M + \ldots$$
$$= M \cdot \sum_{i=0}^{h-1} (M+1)^i = (M+1)^h - 1$$

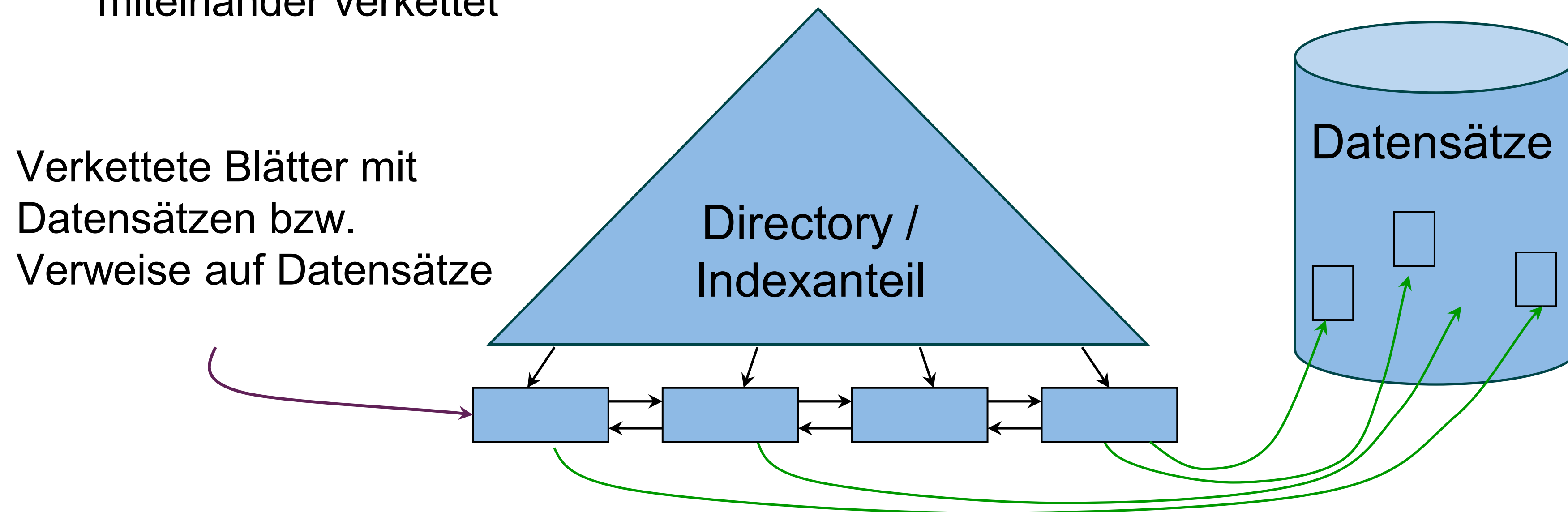
- Auflösen nach h ergibt die Höhenabschätzung:

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

- 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

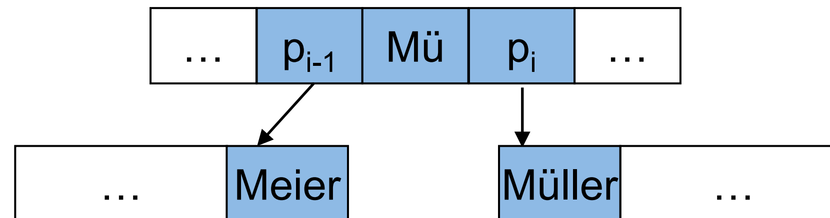
- Ein B⁺-Baum ist eine B-Baum-Variante mit zwei Knotentypen
  - Blätter enthalten Schlüssel mit Datensätzen oder Schlüssel mit Verweisen auf Datensätze
  - Innere Knoten enthalten keine Datensätze, nur Trennschlüssel
  - Als Trennschlüssel (Separatoren, Wegweiser) nutzt man z.B. die Schlüssel selbst 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

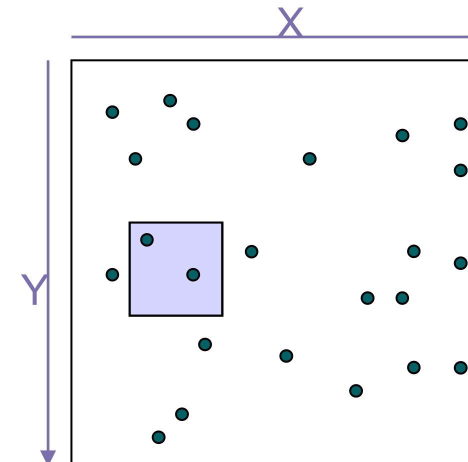
1. Anfragebearbeitung und -optimierung
   - Einführung
   - Grundlagen: Regelbasierte Anfrageoptimierung
   - Grundlagen: Kostenbasierte Anfrageoptimierung
   - Join-Reihenfolgen
   - Ein Algorithmus für die Anfrageoptimierung
2. Algorithmen für Basisoperationen
3. Indexstrukturen
   - Indexstrukturen 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
```

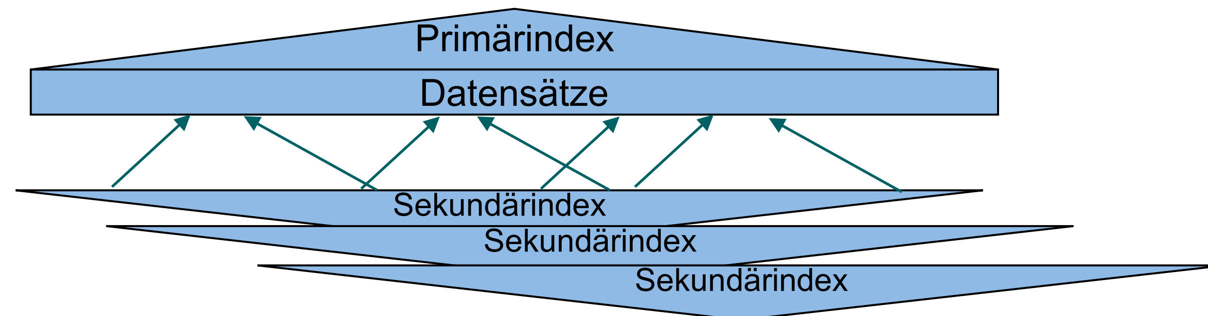



```
SELECT *
FROM Geometry
WHERE 2.5 <= x AND x <= 4.1
      AND 3.8 <= y AND y <= 6.2
```

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.

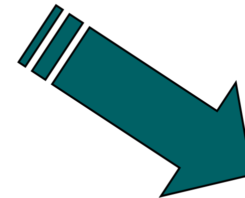
Multiattributsuche für invertierte Listen:

- 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
```

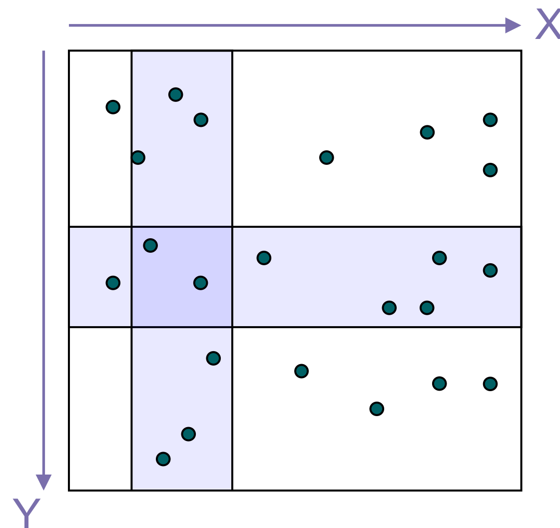
„Athen“ „30“

$\{1,5\} \cap \{1,2,5,8\} = \{1,5\}$

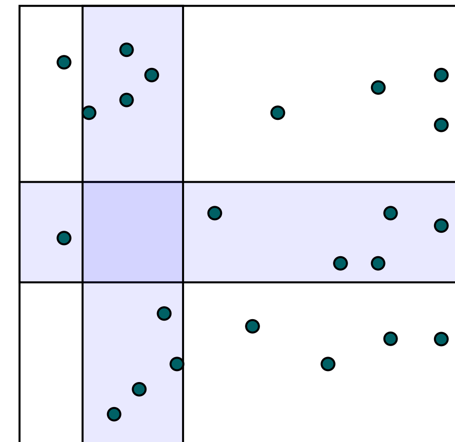
Invertierte Listen: Punktdaten

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.

Bitmap Index

- Bitmap-Indizes sind ein spezieller Indextyp, der für effiziente Abfragen mit mehreren Schlüsseln konzipiert ist.
- Sehr effektiv bei Attributen, die eine relativ kleine Anzahl *unterschiedlicher Werte* annehmen
  - z.B. Geschlecht, Land, Staat, ...
  - Z.B. Einkommensstufe (Einkommen unterteilt in eine kleine Anzahl von Stufen wie 0-9999, 10000-19999, 20000-50000, 50000-unendlich)
- Eine Bitmap ist einfach ein Bit-Array
  - Jedem Geschlecht wird eine Bitmap zugeordnet, wobei jedes Bit angibt, ob der entsprechende Datensatz dieses Geschlecht hat oder nicht.

Bitmap Index (2)

- In seiner einfachsten Form hat ein Bitmap-Index für ein Attribut eine Bitmap für jeden Wert des Attributs
  - Eine Bitmap hat so viele Bits wie die Relation Tupel
  - In einer Bitmap für den Wert v ist das Bit für einen Datensatz 1, wenn der Datensatz den Wert v für das Attribut hat, und sonst 0

Tupel	Name	Geschlecht	Addresse	Einkommenstufe
0	Klaus	m	Aachen	L1
1	Diana	w	Köln	L2
2	Maria	w	Bonn	L1
3	Peter	m	Köln	L4
4	Johannes	w	Aachen	L3

Bitmaps für Geschlecht

m	1 0 0 1 0
f	0 1 1 0 1

Bitmaps für Einkommenstufe

L1	1 0 1 0 0
L2	0 1 0 0 0
L3	0 0 0 0 1
L4	0 0 0 1 0
L5	0 0 0 0 0

Bitmap Index

- Bitmap-Indizes sind nützlich für Abfragen über mehrere Attribute
  - nicht besonders nützlich für Abfragen über ein einzelnes Attribut
- Abfragen werden mit Bitmap-Operationen beantwortet
  - Schnittmenge (und)
  - Vereinigung (oder)
  - Komplement (nicht)
- Jede Operation nimmt zwei Bitmaps der gleichen Größe und wendet die Operation auf die entsprechenden Bits an, um die Ergebnis-Bitmap zu erhalten
  - Z.B. 100110 AND 110011 = 100010
    100110 ODER 110011 = 110111
    NICHT 100110 = 011001
  - Männer mit Einkommensstufe L1:
  - Und-Verknüpfung der Bitmap Männer mit der Bitmap Einkommensstufe L1
  - 10010 UND 10100 = 10000
- Kann dann die gewünschten Tupel abrufen.
- Das Zählen der Anzahl der übereinstimmenden Tupel ist noch schneller

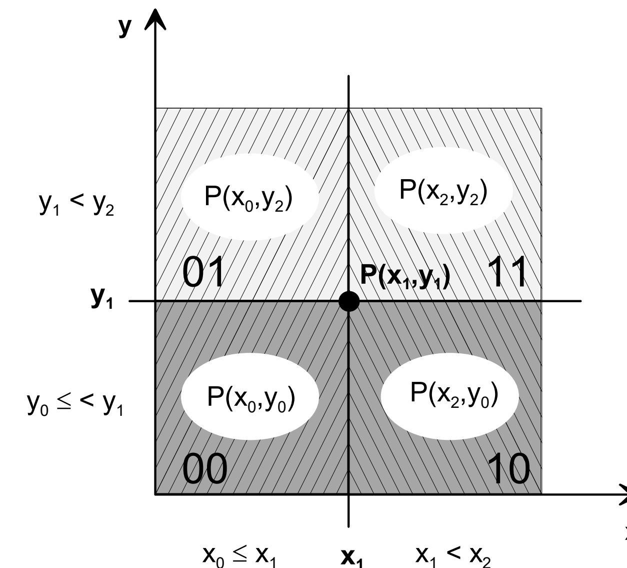
Bitmap Index

- Bitmap-Indizes sind im Allgemeinen sehr klein im Vergleich zur Größe der Beziehung
  - Wenn z.B. ein Datensatz 100 Bytes groß ist, beträgt der Platz für eine einzelne Bitmap 1/800 des von der Relation belegten Platzes.
  - Wenn die Anzahl der unterschiedlichen Attributwerte 8 beträgt, ist die Bitmap nur 1% der Größe der Beziehung.
- Das Löschen muss richtig gehandhabt werden
  - Existenz-Bitmap, um zu vermerken, ob ein gültiger Datensatz an einer Datensatzposition vorhanden ist
  - Erforderlich für die Komplementierung
    - not(A=v): (NOT bitmap-A-v) AND ExistenceBitmap
- Sollte Bitmaps für alle Werte behalten, auch für Nullwerte
  - Um die SQL-Null-Semantik für NOT(A=v) korrekt zu behandeln:
  - obiges Ergebnis mit (NOT bitmap-A-Null) schneiden

Quadtree (2D)

3D: Octree
1D: Segmenttree

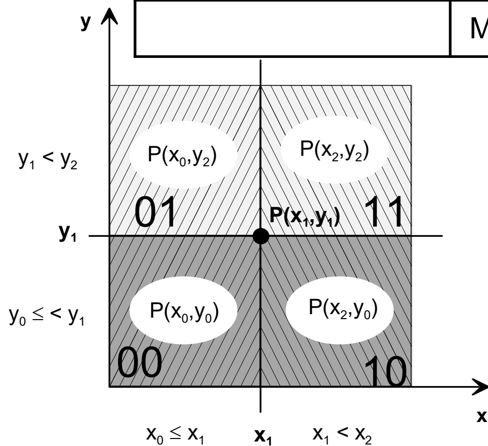
- nicht-balancierte Datenstruktur für mehrdimensionale Punkt-Objekte (hier: 2-dimensionale)
- Ansatz: Jeder Knoten hat sowohl für die x- als auch für die y-Koordinate je zwei Kindknoten: x (<=,>) und y (<=,>) (je Knoten also vier Kindknoten □ „*Quad*“-Tree)
- Aufbau des Baums:
  - erster Punkt bildet 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



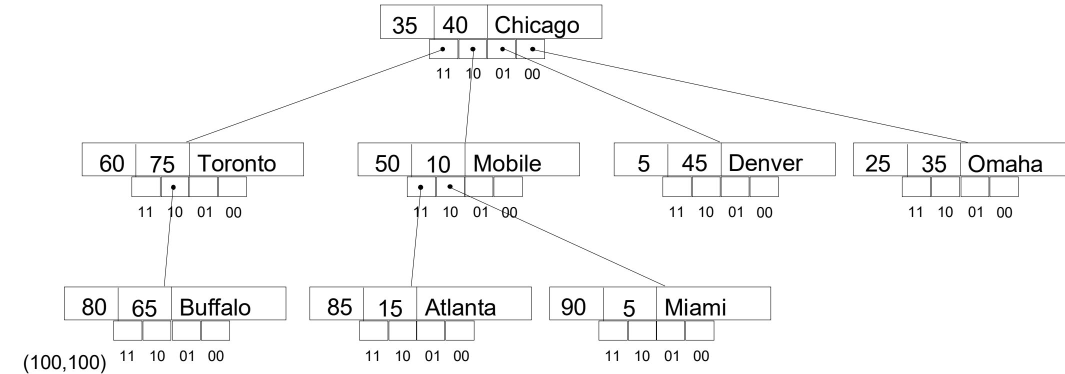
Informatik 5 - Informationssysteme und Datenbanken
RWTH AACHEN UNIVERSITY

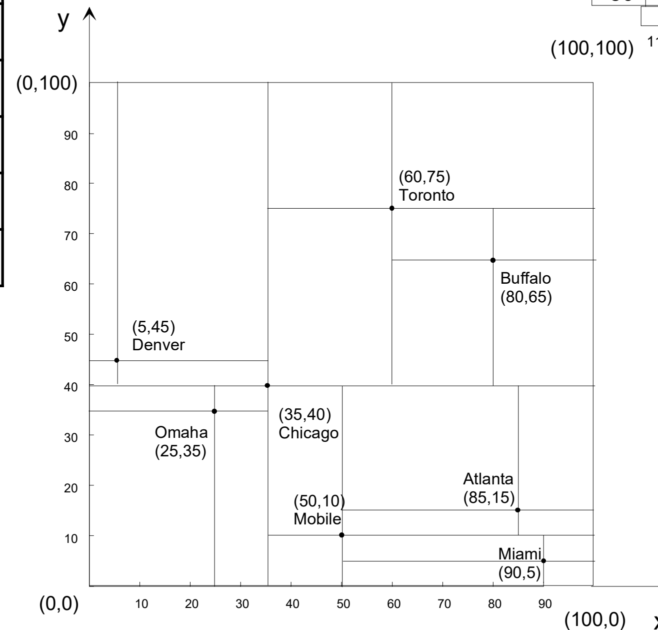
Quadtree (Beispiel)

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

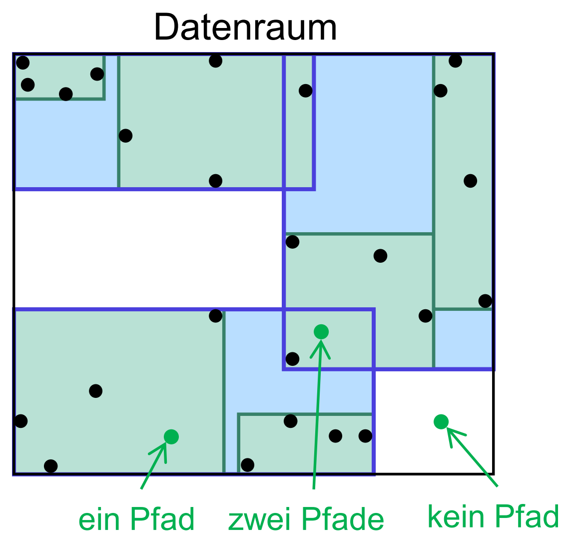
Aufbau einer Seite:

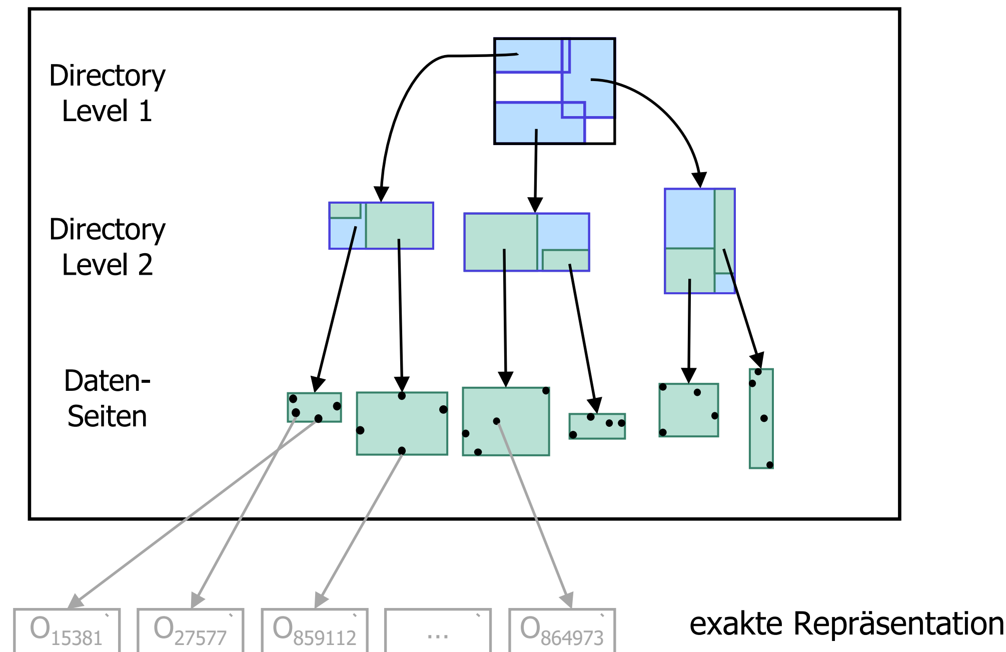
- Seite besteht aus mehreren Einträgen
- Einträge in Directory-Seiten bestehen aus MURs und Verweisen auf andere Seiten
- Einträge in Datenseiten bestehen aus MURs und Verweisen auf die exakte Objekt-Repräsentation, bzw. einfach aus Punkten

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

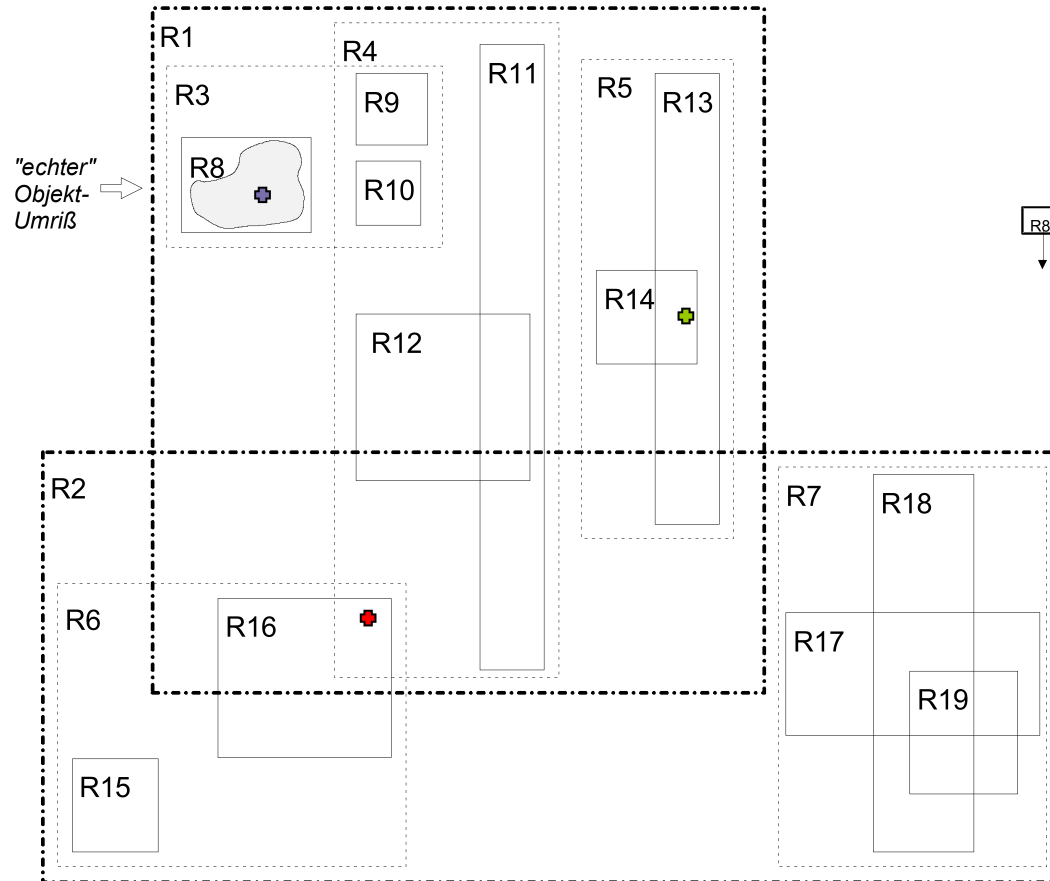
Parameter:

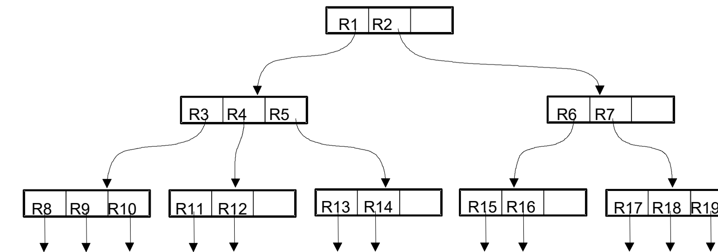
- *M* : maximale Anzahl von Einträgen pro Knoten (abhängig von Blockgröße)
- $m \leq M/2$: Mindestbelegung pro Knoten (z.B. $m = 40\ \% \cdot M$)

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 <u>nicht</u> disjunkt (also überlappende Regionen möglich)
- Höhe des Baums ≤ $\lceil \log_m N \rceil - 1$ (bei N gespeicherten Objekten)

R-Baum: Punktanfrage



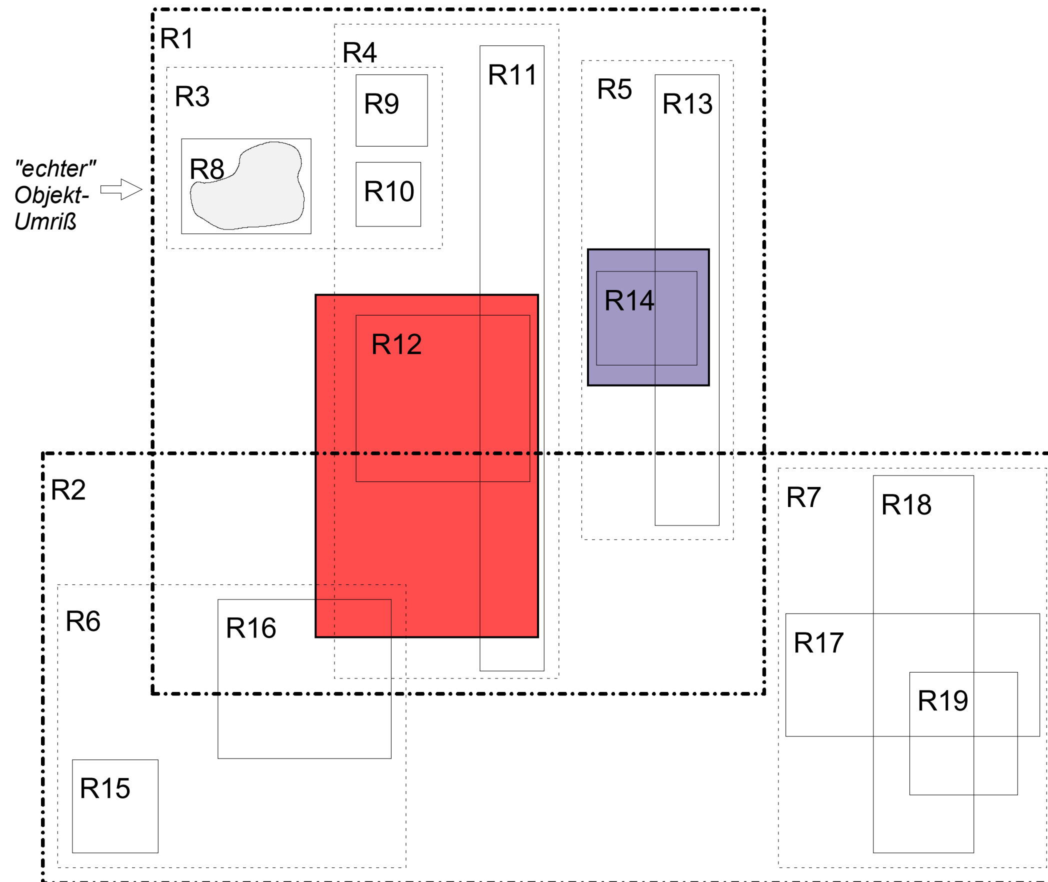


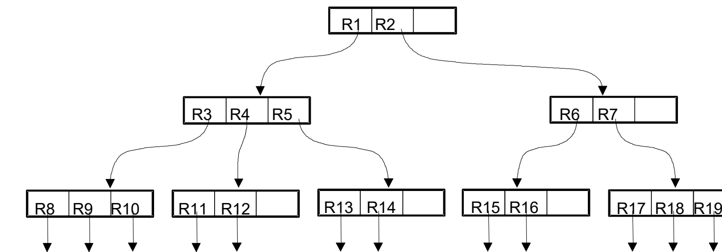
Eingabe: *Punkt p*

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.