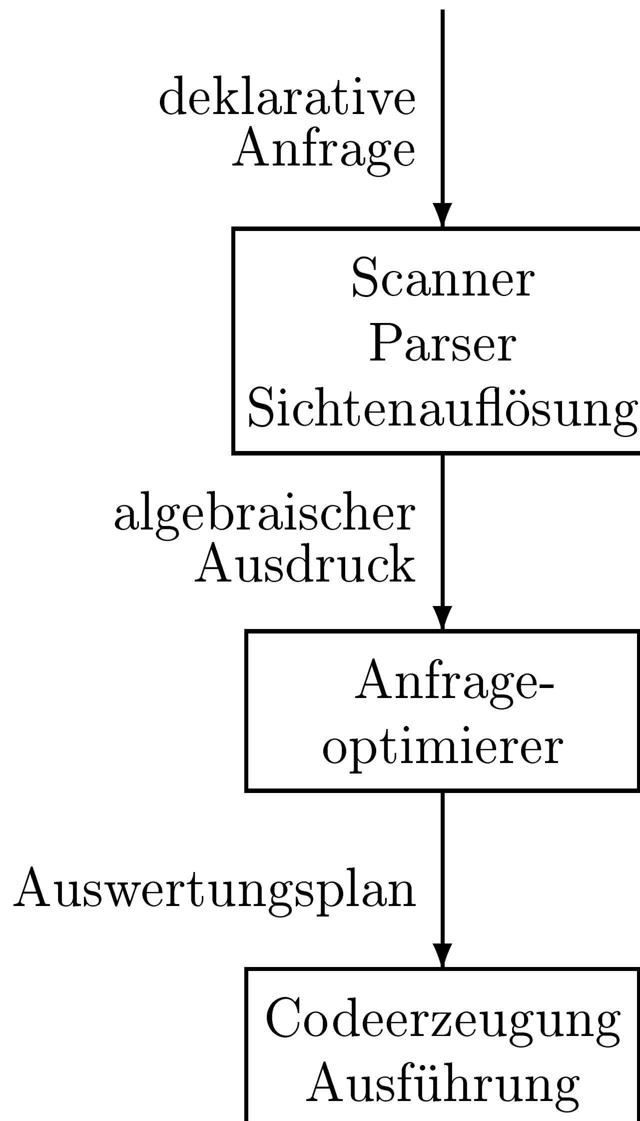
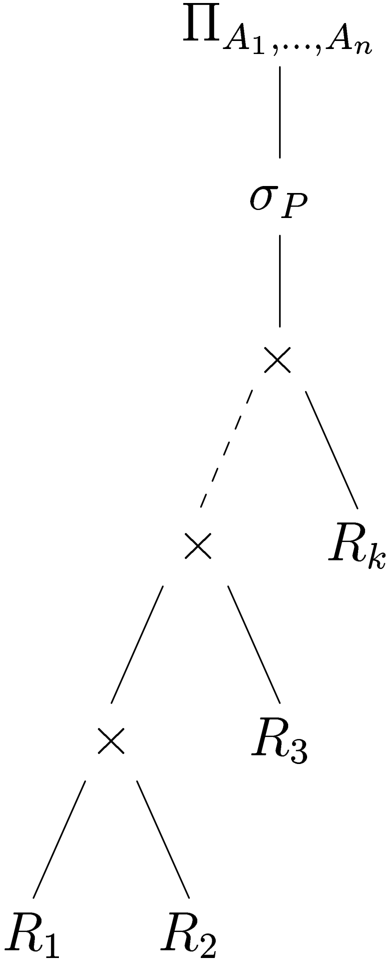
Ablauf der Anfragebearbeitung



Kanonische Übersetzung einer SQL-Anfrage

select $A_1, \ldots, A_n$
from $R_1, \ldots, R_k$
where P;

kanonische
$\Rightarrow$
Übersetzung



Demonstration der logischen Optimierung

```
select Titel
from Professoren, Vorlesungen
where Name = 'Popper' and PersNr = gelesenVon;
```

Kanonische Übersetzung:

$$\Pi_{\text{Titel}}(\sigma_{\text{Name}='\text{Popper}' \wedge \text{PersNr}=\text{gelesenVon}}(\text{Professoren} \times \text{Vorlesungen}))$$

Auswertung anhand der Beispielausprägung:

- 70 Tupel im Kreuzprodukt!

Demonstration der logischen Optimierung

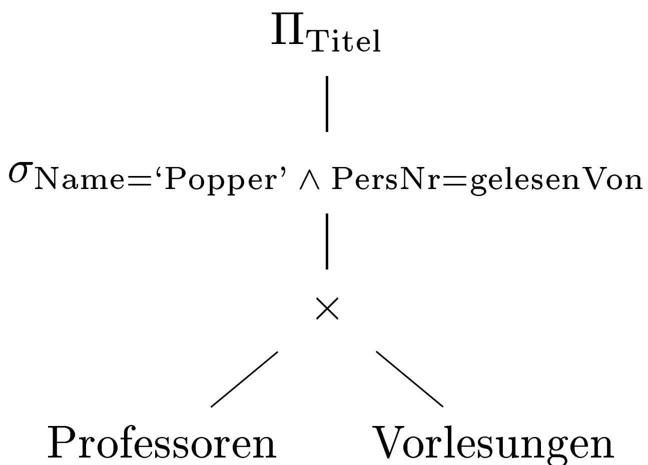
Verschiebung der Selektion:

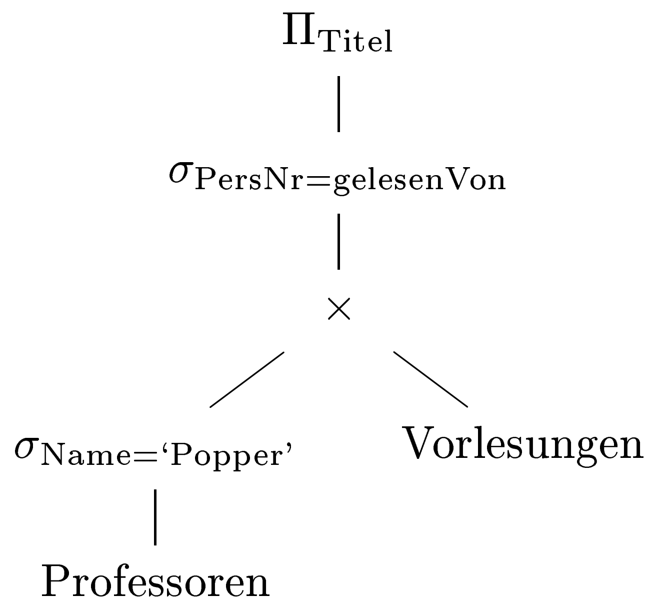
$$\Pi_{\text{Titel}}(\sigma_{\text{PersNr}=\text{gelesenVon}}(\sigma_{\text{Name}='\text{Popper}'}(\text{Professoren}) \times \text{Vorlesungen}))$$

Auswertung anhand der Beispielprägung:

- 7 Tupel aus *Professoren* für die Selektion „anfassen"
- 10 Tupel aus *Vorlesungen* für das Kreuzprodukt „anfassen"

Zum Vergleich:





Äquivalenzen in der relationalen Algebra

1. Join, Vereinigung, Schnitt und Kreuzprodukt sind kommutativ, also:

$$\begin{aligned} R_1 \bowtie R_2 &= R_2 \bowtie R_1 \\ R_1 \cup R_2 &= R_2 \cup R_1 \\ R_1 \cap R_2 &= R_2 \cap R_1 \\ R_1 \times R_2 &= R_2 \times R_1 \end{aligned}$$

2. Selektionen sind untereinander vertauschbar.

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

3. Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ, also:

$$\begin{aligned} R_1 \bowtie (R_2 \bowtie R_3) &= (R_1 \bowtie R_2) \bowtie R_3 \\ R_1 \cup (R_2 \cup R_3) &= (R_1 \cup R_2) \cup R_3 \\ R_1 \cap (R_2 \cap R_3) &= (R_1 \cap R_2) \cap R_3 \\ R_1 \times (R_2 \times R_3) &= (R_1 \times R_2) \times R_3 \end{aligned}$$

4. Konjunktionen in einer Selektionsbedingung können in mehrere Selektionen aufgebrochen, bzw. nacheinander ausgeführte Selektionen können durch Konjunktionen zusammengefügt werden.

$$\sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots))$$

5. Geschachtelte Projektionen können eliminiert werden.

$$\Pi_{l_1}(\Pi_{l_2}(\ldots(\Pi_{l_n}(R))\ldots)) = \Pi_{l_1}(R)$$

   Damit eine solche Schachtelung überhaupt sinnvoll ist, muß gelten:

$$l_1 \subseteq l_2 \subseteq \ldots \subseteq l_n \subseteq \mathcal{R} = sch(R)$$

Äquivalenzen in der relationalen Algebra

6. Eine Selektion kann an einer Projektion „vorbeigeschoben" werden, falls die Projektion keine Attribute aus der Selektionsbedingung entfernt. Es gilt also

$$\Pi_l(\sigma_p(R)) = \sigma_p(\Pi_l(R)), \text{ falls } attr(p) \subseteq l$$

7. Selektionen können an Joinoperationen (oder Kreuzprodukten) vorbeigeschoben werden, falls sie nur Attribute *eines* der beiden Join-Argumente verwenden. Enthält die Bedingung p beispielsweise nur Attribute aus $\mathcal{R}_1$, dann gilt

$$\sigma_p(R_1 \bowtie R_2) = \sigma_p(R_1) \bowtie R_2$$

8. Auf ähnliche Weise können auch Projektionen verschoben werden. Hier muß allerdings beachtet werden, daß die Join-Attribute bis zum Join erhalten bleiben.

$$\Pi_l(R_1 \bowtie_p R_2) = \Pi_l(\Pi_{l_1}(R_1) \bowtie_p \Pi_{l_2}(R_2)) \text{ mit}$$
$$l_1 = \{A | A \in \mathcal{R}_1 \cap l\} \cup \{A | A \in \mathcal{R}_1 \cap attr(p)\} \text{ und}$$
$$l_2 = \{A | A \in \mathcal{R}_2 \cap l\} \cup \{A | A \in \mathcal{R}_2 \cap attr(p)\}$$

9. Selektionen können mit Mengenoperationen wie Vereinigung, Schnitt und Differenz vertauscht werden, also:

$$\begin{aligned} \sigma_p(R \cup S) &= \sigma_p(R) \cup \sigma_p(S) \\ \sigma_p(R \cap S) &= \sigma_p(R) \cap \sigma_p(S) \\ \sigma_p(R - S) &= \sigma_p(R) - \sigma_p(S) \end{aligned}$$

Äquivalenzen in der relationalen Algebra

10. Der Projektions-Operator kann mit der Vereinigung vertauscht werden.

$$\Pi_l(R_1 \cup R_2) = \Pi_l(R_1) \cup \Pi_l(R_2)$$

    Eine Vertauschung der Projektion mit Durchschnitt und Differenz ist allerdings nicht zulässig.

11. Eine Selektion und ein Kreuzprodukt können zu einem Join zusammengefaßt werden, wenn die Selektionsbedingung eine Joinbedingung ist. Für Equijoins gilt beispielsweise

$$\sigma_{R_1.A_1=R_2.A_2}(R_1 \times R_2) = R_1 \bowtie_{R_1.A_1=R_2.A_2} R_2$$

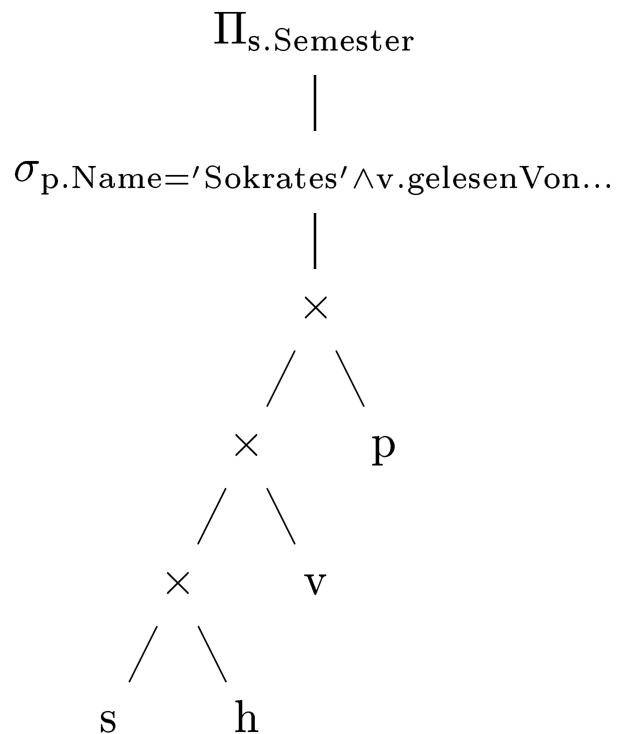
12. Auch an Bedingungen können Veränderungen vorgenommen werden. Beispielsweise kann eine Disjunktion mit Hilfe von DeMorgan's Gesetz in eine Konjunktion umgewandelt werden, um vielleicht später die Anwendung von Regel 4 zu ermöglichen:

$$\neg(p_1 \vee p_2) = \neg p_1 \wedge \neg p_2$$

Anwendung der Transformationsregeln

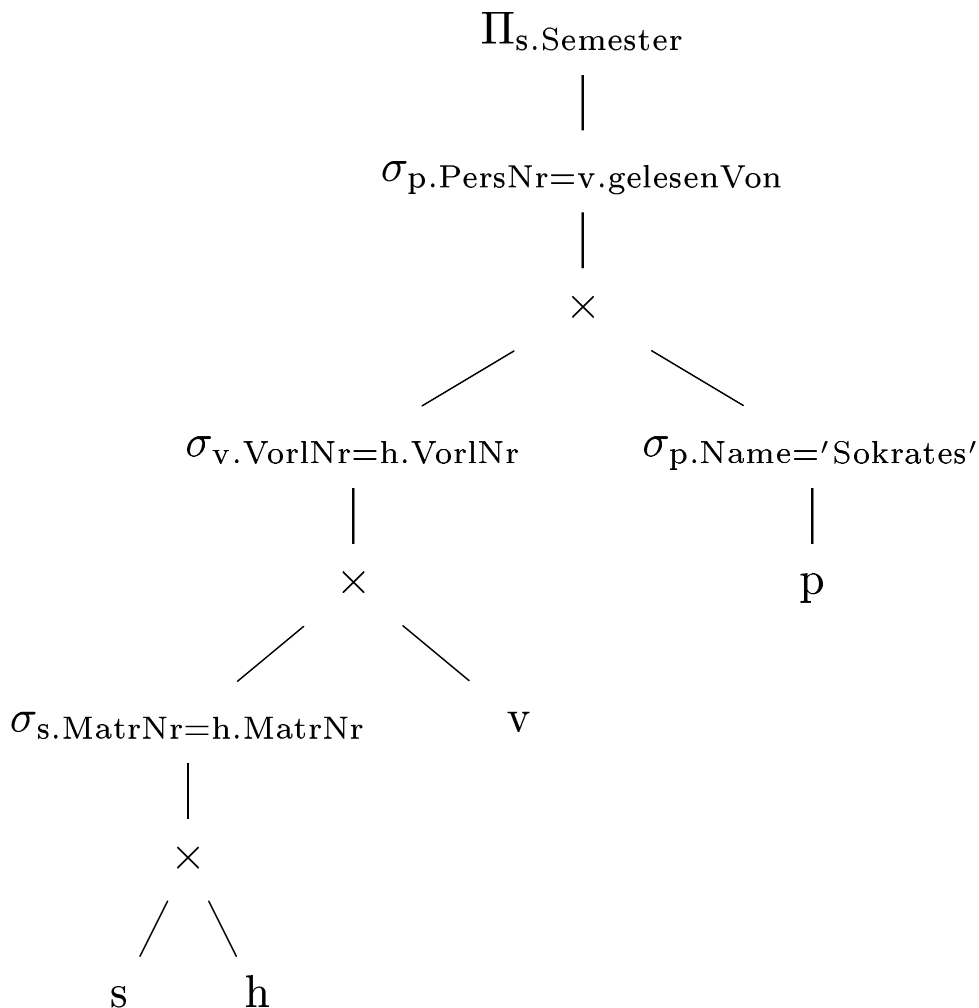
Die Ausgangsanfrage und ihre kanonische Übersetzung:

select distinct s.Semester
from Studenten s, hören h,
Vorlesungen v, Professoren p
where p.Name = 'Sokrates' **and**
v.gelesenVon = p.PersNr **and**
v.VorlNr = h.VorlNr **and**
h.MatrNr = s.MatrNr;



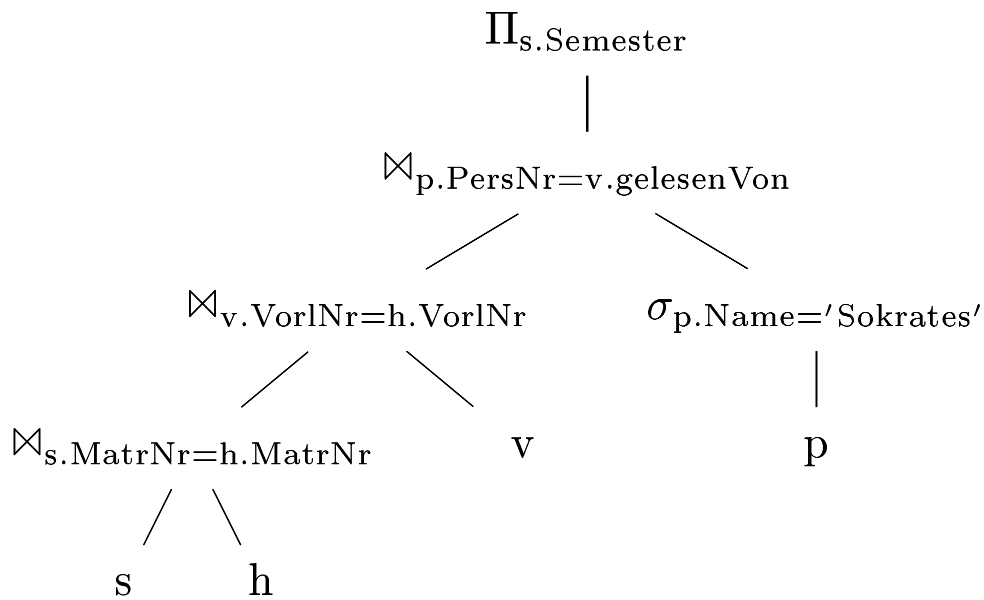
Verschieben der Selektionen

- Aufbrechen der Konjunktionen (Regel 4)
- Verschieben der Selektionen „nach unten“ (Regel 2, 6, 7 und 9)



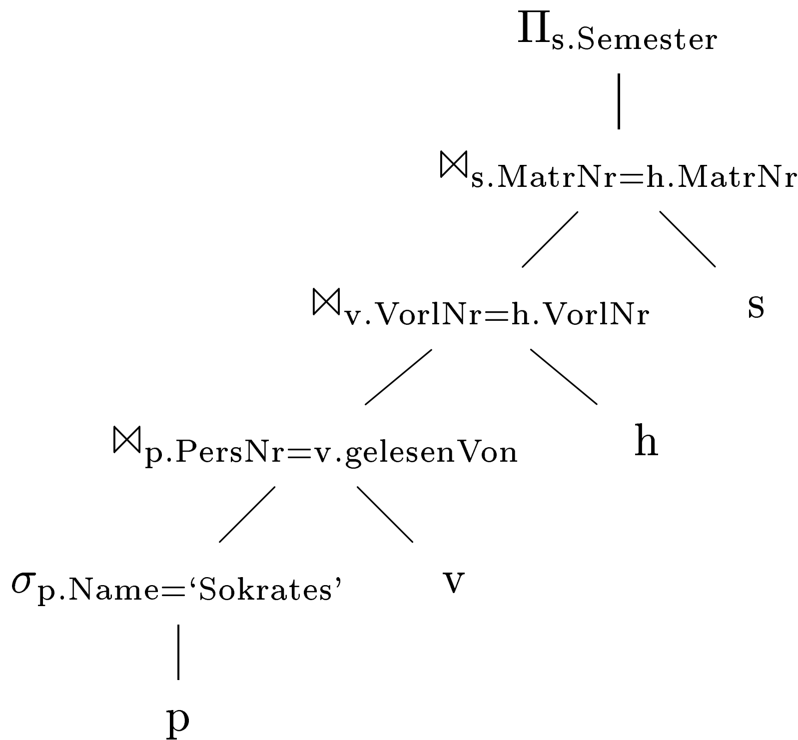
Erzeugen von Joins aus Kreuzprodukten

- Zusammenfassen von Selektionen und Kreuzprodukten (Regel 5 und 11)

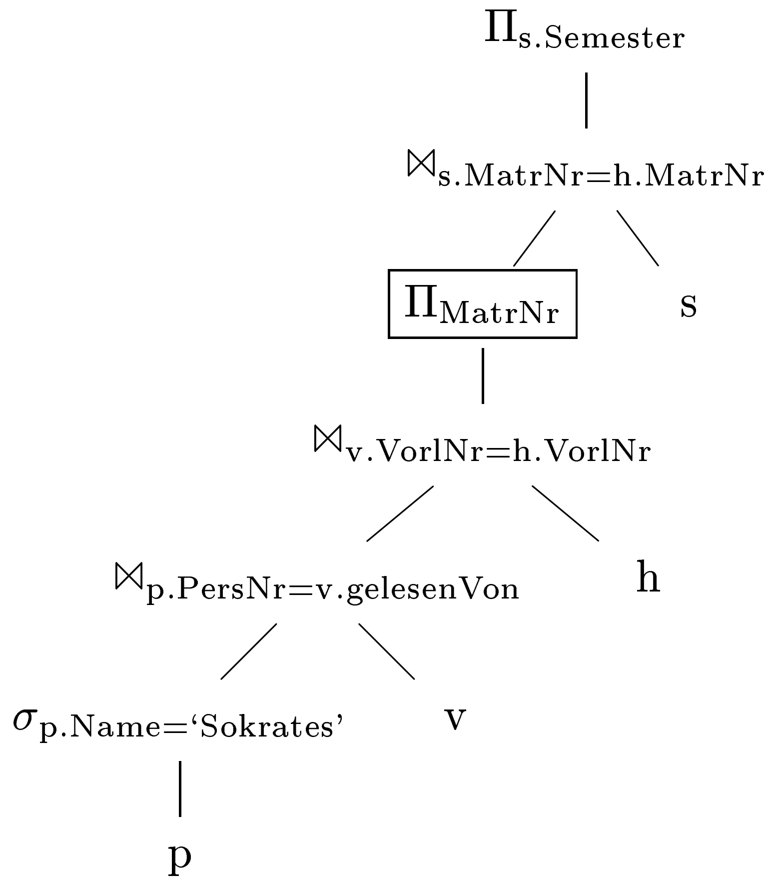


Bestimmung der Joinreihenfolge

- Kommutativität des Joins (Regel 1)
- Assoziativität des Joins (Regel 3)



Einfügen und Verschieben von Projektionen



Zusammenfassung

1. Aufbrechen von Selektionen
2. Verschieben der Selektionen soweit wie möglich nach unten im Operatorbaum
3. Zusammenfassen von Selektionen und Kreuzprodukten zu Joins
4. Bestimmung der Anordnung der Joins
5. u.U. Einfügen von Projektionen
6. Verschieben der Projektionen soweit wie möglich nach unten im Operatorbaum

Dynamische Programmierung

procedure DP:

    for i:=1 **to** n do

        optPlan($\{R_i\}$) := R_i

    for i:=2 **to** n do {

        for all $\mathcal{S} \subseteq \{R_1, \ldots, R_n\}$ mit $\|S\|$=i **do** {

            bestPlan($\mathcal{S}$) := dummy mit ∞ Kosten

            for all $\mathcal{R}$, $\mathcal{L}$ mit $\mathcal{S} = \mathcal{R} \cup \mathcal{L}$ **do** {

                p := optPlan($\mathcal{R}$) $\bowtie$ optPlan($\mathcal{S}$)

                if cost(p) $<$ cost(bestPlan($\mathcal{S}$))

                    bestPlan($\mathcal{S}$) := p

            }

        }

    }

    return optPlan($\{R_1, \ldots, R_n\}$)

Joinreihenfolge: Kostenbeispiel

$\text{Kosten}(A \times B) = |A| * |B|$

$|\text{Prof}| = 1; |\text{Vorl}| = 1; |\text{Stud}|{=}10000;$

$\text{Kosten}((\text{Prof} \times \text{Stud}) \times \text{Vorl}) = 20000$

$\text{Kosten}((\text{Prof} \times \text{Vorl}) \times \text{Stud}) = 10001$

Physische Optimimierung

Wie führt man die einzelnen Operatoren aus?

1. Zugriff auf eine Tabelle
    - mit oder ohne Index?
    - mit welchem Index?
2. Implementierung des Joins
    - welches Verfahren?
    - nested-loops, sortieren, hashing, ...?
3. ...

Fazit: Wiederum benötigt man ein Kostenmodell und einen Optimierer der verschiedene Alternativen aufzählt.

Dynamische Programmierung zur Physischen Optimierung

procedure DP:

  for i:=1 **to** n do

    optPlan($\{R_i\}$) := accessPlan(R_i)

  for i:=2 **to** n do {

    for all $\mathcal{S} \subseteq \{R_1, \ldots, R_n\}$ mit $\|S\|$=i **do** {

      bestPlan($\mathcal{S}$) := dummy mit ∞ Kosten

      for all $\mathcal{R}$, $\mathcal{L}$ mit $\mathcal{S} = \mathcal{R} \cup \mathcal{L}$ **do** {

        p := joinPlan(optPlan($\mathcal{R}$), optPlan($\mathcal{S}$))

        if cost(p) < cost(bestPlan($\mathcal{S}$))

          bestPlan($\mathcal{S}$) := p

      }

    }

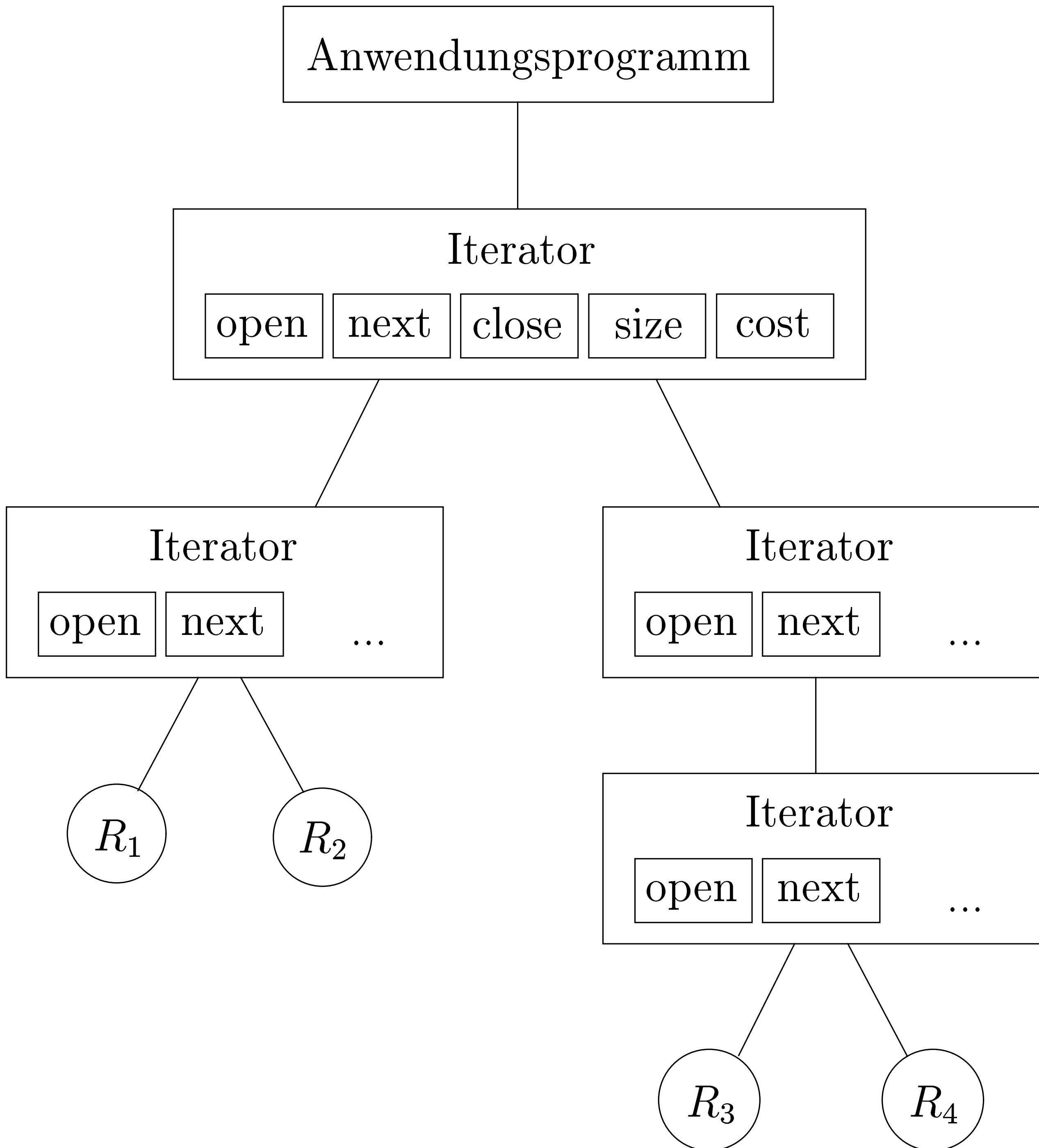
  }

  return optPlan($\{R_1, \ldots, R_n\}$)

Dynamische Programmierung: Bewertung

- exponentielle Laufzeit: $\mathcal{O}(3^n)$
- exponentieller Speicherbedarf: $\mathcal{O}(2^n)$
- (N.B.: Das Problem ist $\mathcal{NP}$ hart.)
- liefert ersten Plan erst ganz am Ende
- findet “optimalen” Plan
- sehr gut erweiterbar

- Bau von Auswertungsplänen mit Hilfe des *Iteratorkonzepts*



Implementierung der Selektion

a) **iterator** Scan_p

open

- Öffne Eingabe

next

- Hole solange nächstes Tupel, bis eines die Bedingung p erfüllt
- Gebe dieses Tupel zurück

close

- Schließe Eingabe

b) **iterator** IndexScan_p

open

- Schlage im Index das erste Tupel nach, das die Bedingung erfüllt
- Öffne Eingabe

next

- Gebe nächstes Tupel zurück, falls es die Bedingung p noch erfüllt

close

- Schließe Eingabe

Implementierung der Joinoperation

- Mengendifferenz und -durchschnitt können analog zum Join implementiert werden
- hier nur Equi-Joins betrachtet

Nested-Loop-Join:

for each $r \in R$
  for each $s \in S$
    if $r.A = s.B$ **then**
      $res := res \cup (r \times s)$

Iteratordarstellung:

iterator NestedLoop_p

open

- Öffne die linke Eingabe

next

- Rechte Eingabe geschlossen?
  - Öffne sie
- Fordere rechts solange Tupel an, bis Bedingung p erfüllt ist
- Sollte zwischendurch rechte Eingabe erschöpft sein
  - Schließe rechte Eingabe
  - Fordere nächstes Tupel der linken Eingabe an
  - Starte **next** neu
- Gib den Verbund von aktuellem linken und aktuellem rechten Tupel zurück

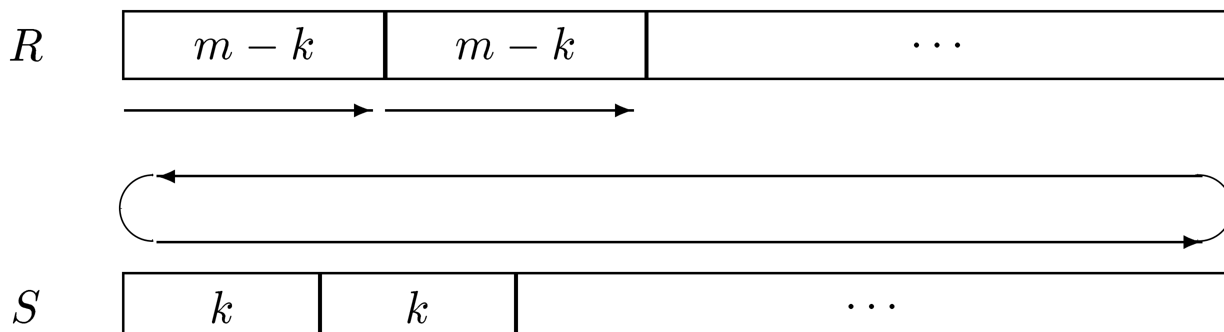
close

- Schließe beide Eingabequellen

Ein verfeinerter Join-Algorithmus

- Relationen sind *seitenweise* abgespeichert
- Es stehen m Pufferrahmen im Hauptspeicher zur Verfügung:
  - k für die innere Schleife des Nested Loop
  - $m - k$ für die äußere

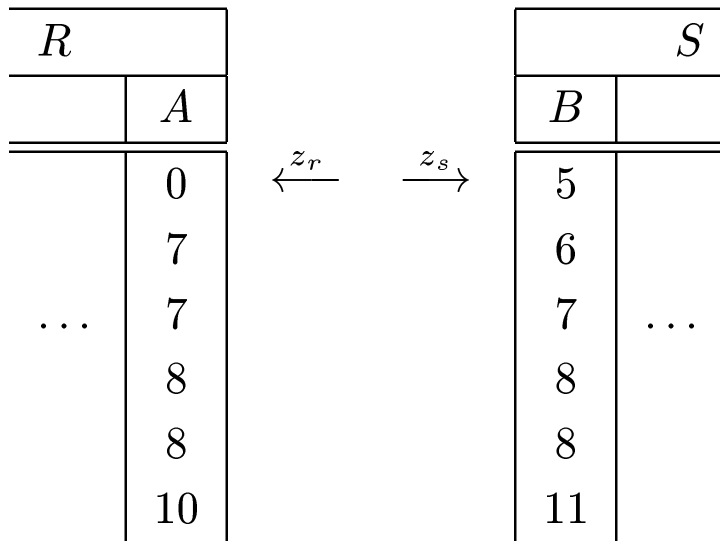
Join von R und S:



Der Merge-Join

- Voraussetzung: R und S sind sortiert (notfalls vorher sortieren)

Beispiel:



Der Merge-Join

iterator MergeJoin_p

open

- Öffne beide Eingaben
- Setze *akt* auf linke Eingabe
- Markiere rechte Eingabe

next

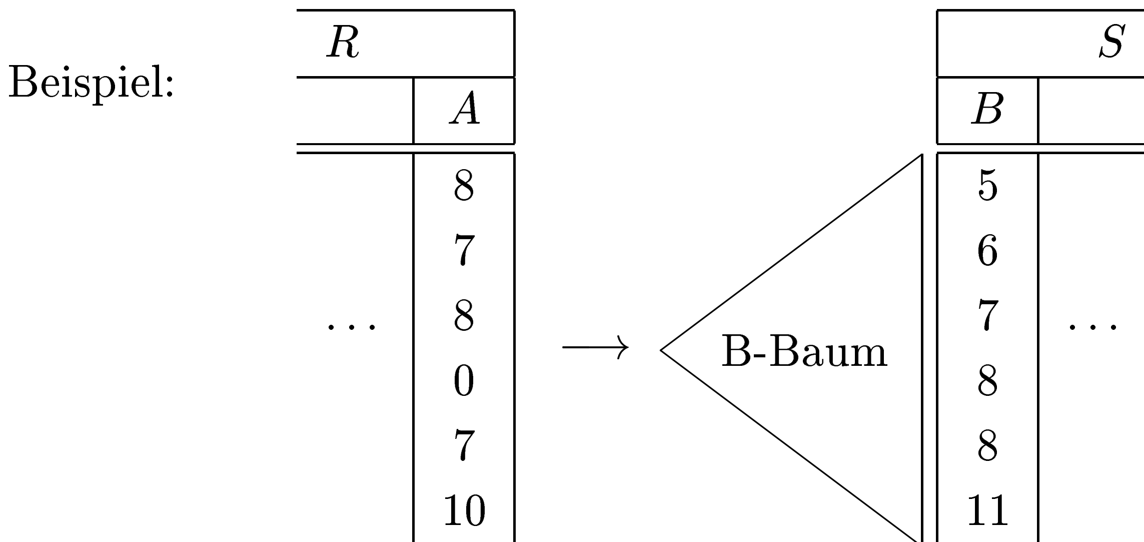
- Solange Bedingung nicht erfüllt
  - Setze *akt* auf Eingabe mit dem kleinsten anliegenden Wert im Joinattribut
  - Rufe **next** auf *akt* auf
  - Markiere andere Eingabe
- Gebe Verbund der aktuellen Tupel der linken und rechten Eingabe zurück
- Bewege andere Eingabe vor
- Ist Bedingung nicht mehr erfüllt oder andere Eingabe erschöpft?
  - Bewege *akt* vor
  - Wert des Joinattributes in *akt* verändert?
    - Nein, dann setze andere Eingabe auf Markierung zurück
    - Ansonsten markiere andere Eingabe

close

- Schließe beide Eingabequellen

Index-Join

Beispiel:



Iteratorstellung:

iterator IndexJoin_p

open

- Sei Index auf Joinattribut der rechten Eingabe vorhanden
- Öffne die linke Eingabe
- Hole erstes Tupel aus linker Eingabe
- Schlage Joinattributwert im Index nach

next

- Bilde Join, falls Index weiteres Tupel zu diesem Attributwert liefert
- Ansonsten bewege linke Eingabe vor und schlage Joinattributwert im Index nach

close

- Schließe die Eingabe

Hash-Join

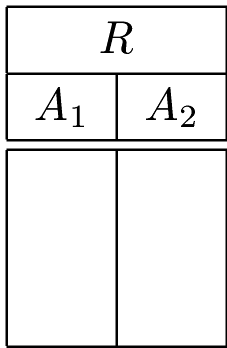
Nachteile des Index-Joins:

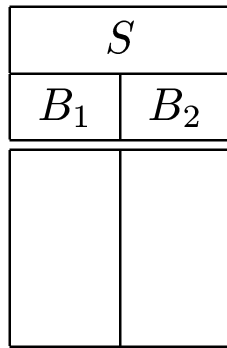
- auf Zwischenergebnissen existieren keine Indexstrukturen
- temporäres Anlegen i.A. zu aufwendig
- Nachschlagen im Index i.A. zu aufwendig

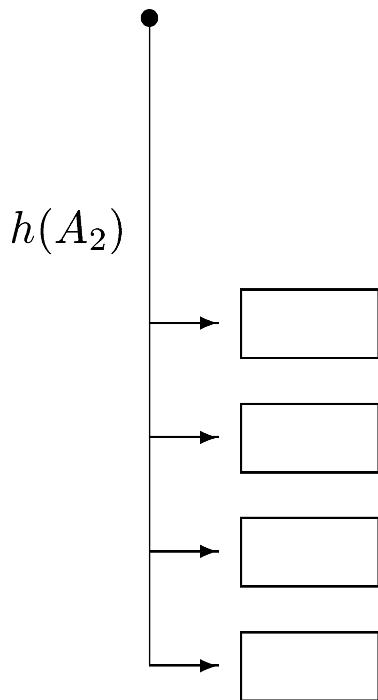
Idee:

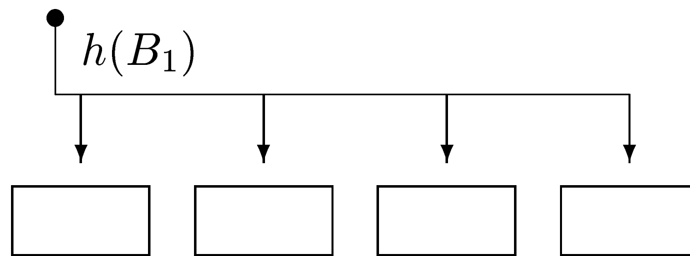
- Partitionieren der Relationen
- Anlegen von *Hauptspeicher*-Indexstrukturen (Hashtabellen) je Partition

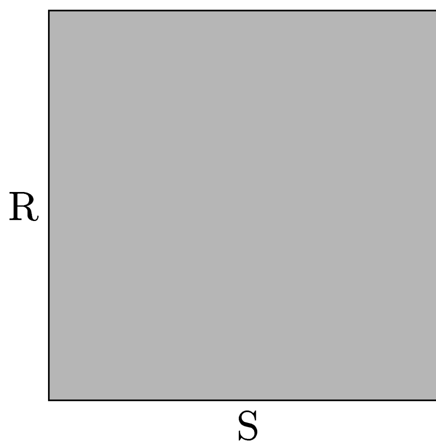
Vergleich der Tupel in der “Diagonalen”

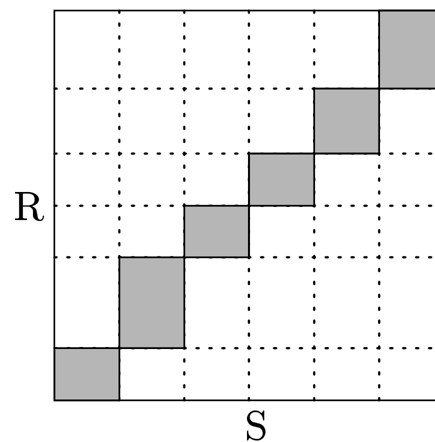




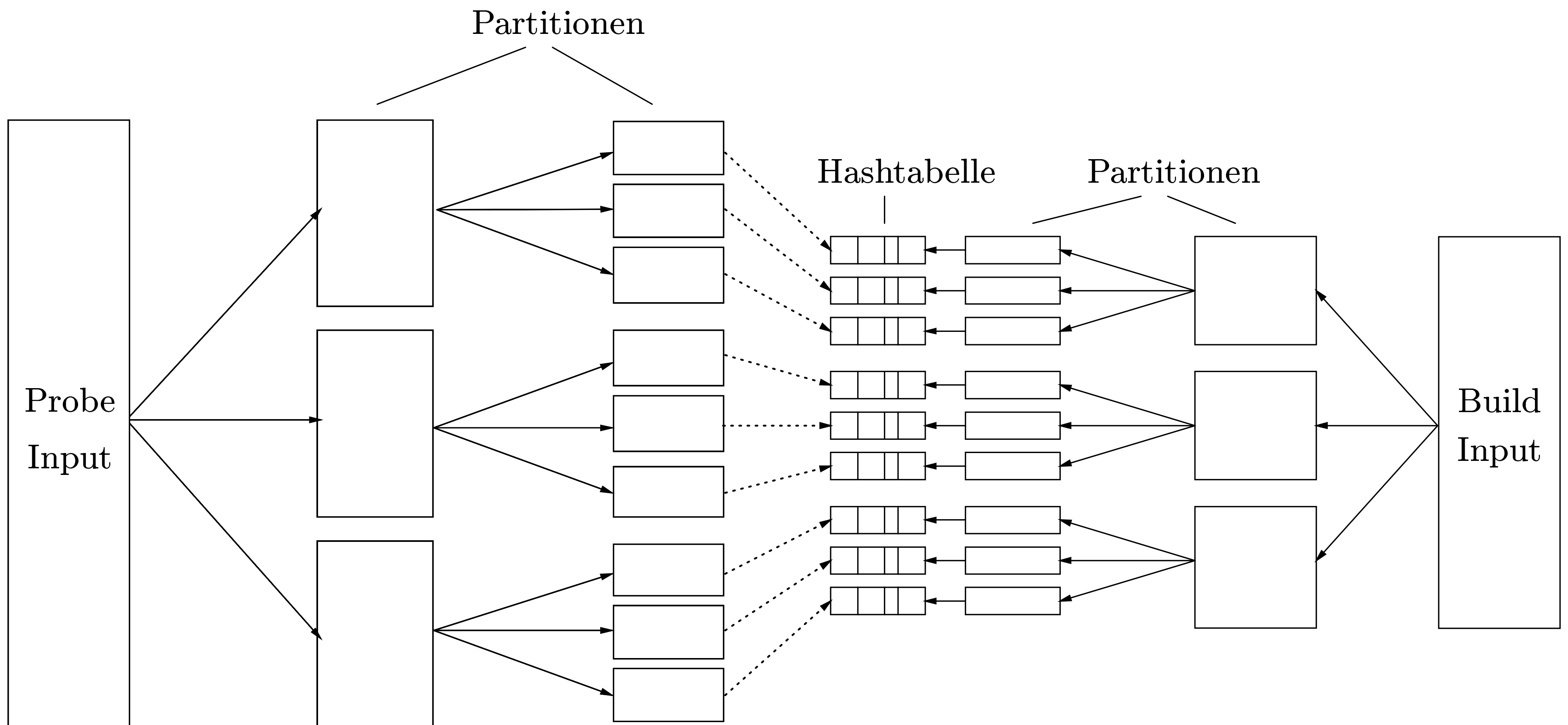




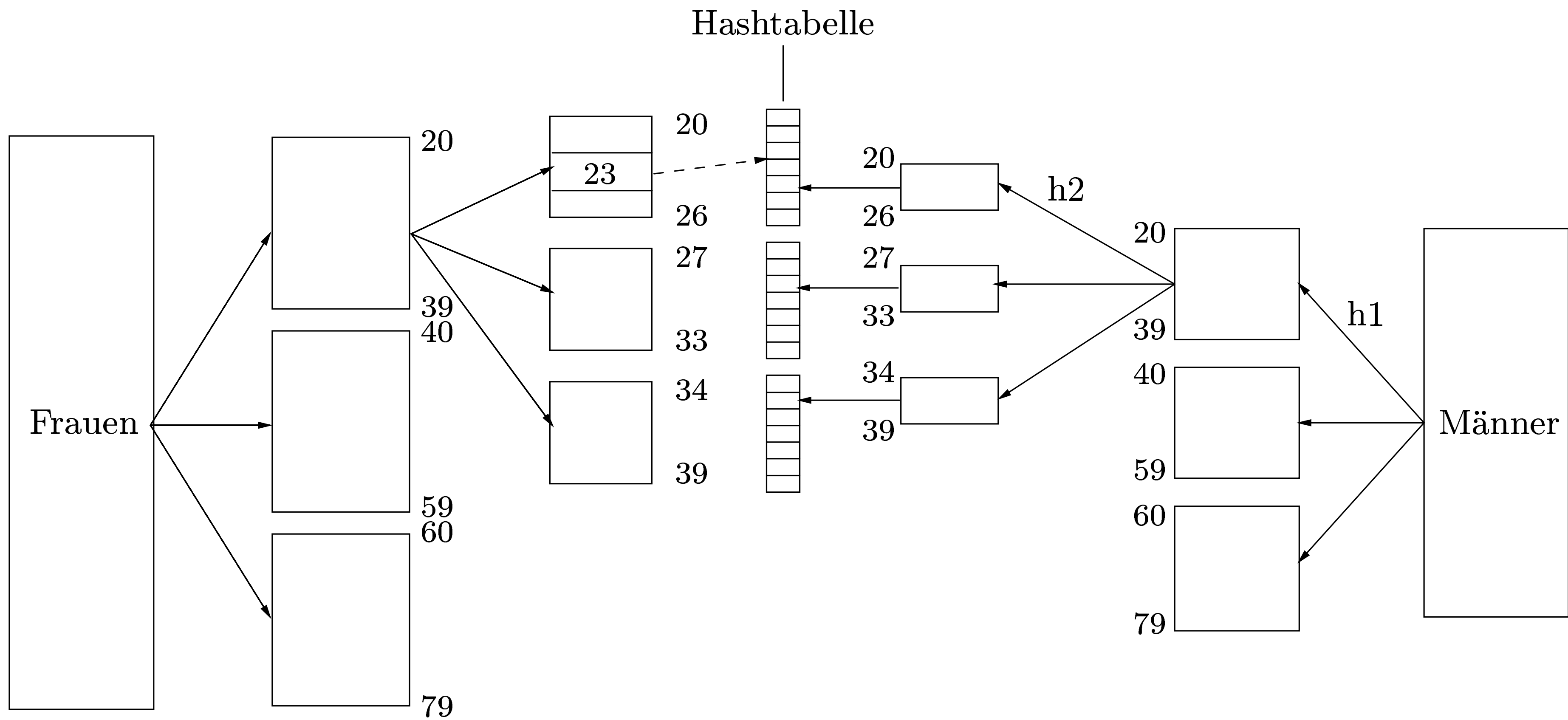




Partitionierung von Relationen



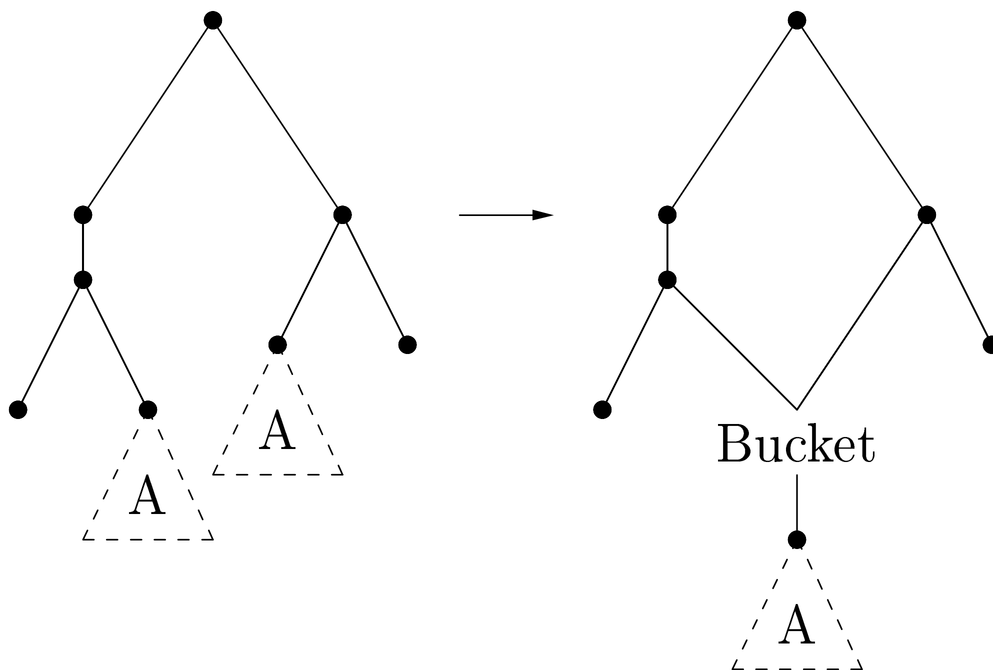
Demonstration der Partitionierung



Zwischenspeicherung

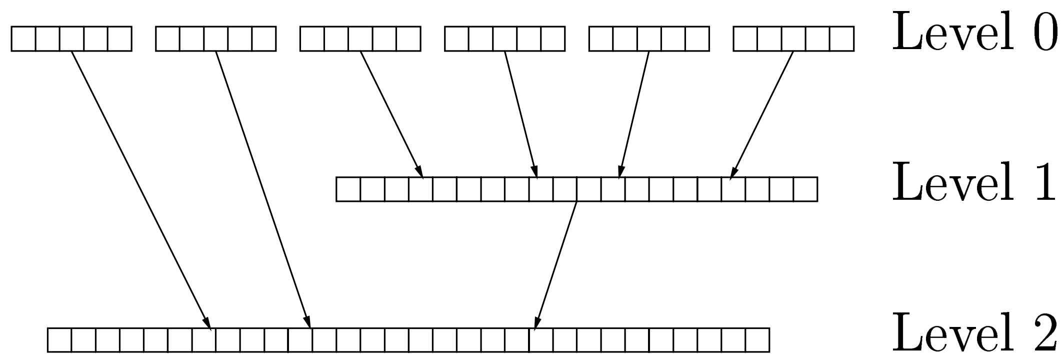
Speicherung von Zwischenergebnissen notwendig, falls

- mehrere Operationen mit hohem Hauptspeicherverbrauch vorkommen (z.B. Hash-Join)
- gemeinsame Teilausdrücke eliminiert werden sollen

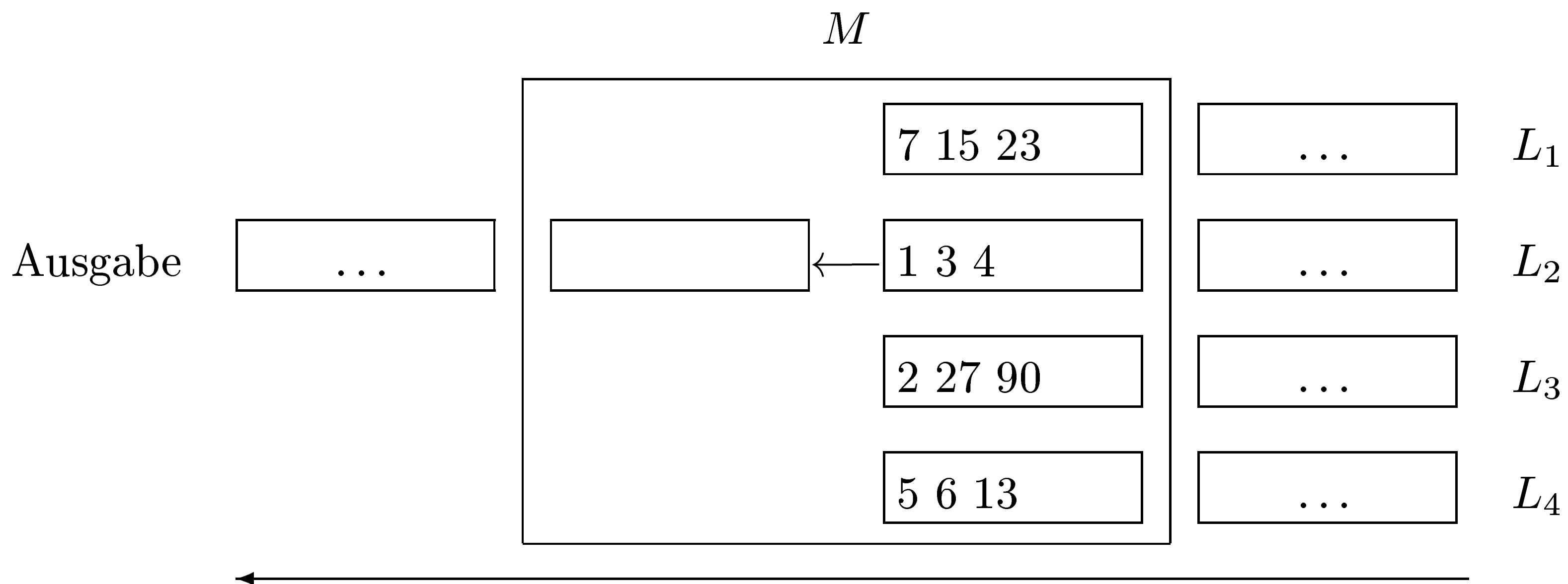


Sortierung auf dem Hintergrundspeicher

Mergesort:



Ein Mischvorgang:

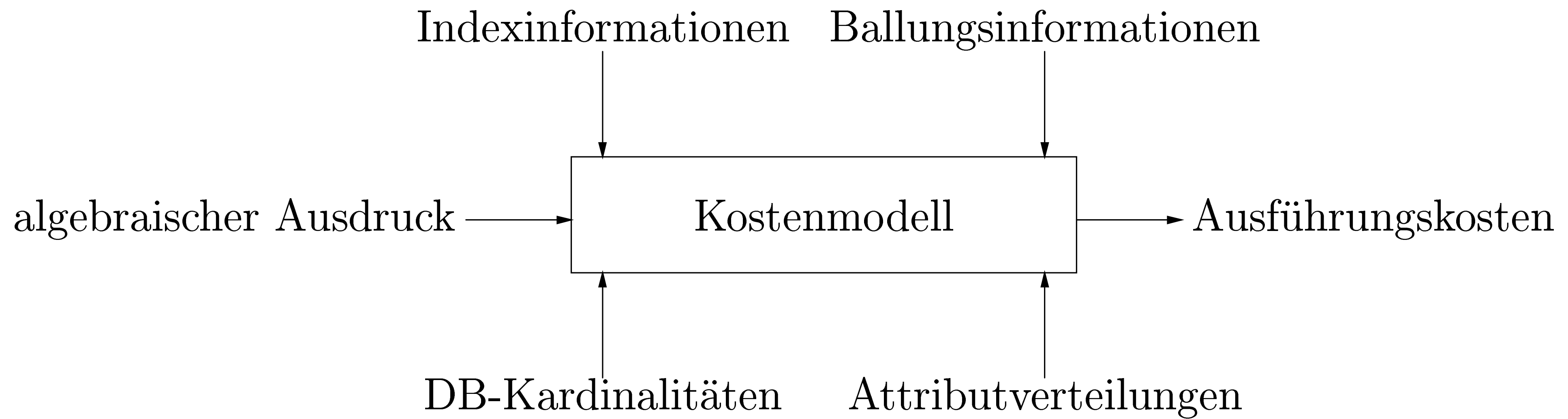


Replacement Selection

Ausgabe						Speicher				Eingabe						
						10	20	30	40	25	73	16	26	33	50	31
					10	20	25	30	40	73	16	26	33	50	31	
				10	20	25	30	40	73	16	26	33	50	31		
			10	20	25	(16)	30	40	73	26	33	50	31			
		10	20	25	30	(16)	(26)	40	73	33	50	31				
	10	20	25	30	40	(16)	(26)	(33)	73	50	31					
10	20	25	30	40	73	(16)	(26)	(33)	(50)	31						
					16	26	31	33	50							

- Vergrößerung der initialen Läufe um den Faktor 2 gegenüber:
  1. Laden des Hauptspeicherbereichs
  2. Sortieren mit Quicksort
  3. Ausschreiben des Laufs

Kostenmodelle



Selektivitäten

- Anteil der qualifizierenden Tupel einer Operation
- Selektion mit Bedingung p:

$$sel_p := \frac{|\sigma_p(R)|}{|R|}$$

- Join von R mit S:

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

Abschätzung der Selektivität:

- $sel_{R.A=C} = \frac{1}{|R|}$
  falls A Schlüssel von R
- $sel_{R.A=C} = \frac{1}{i}$
  falls i die Anzahl der Attributwerte von $R.A$ ist (Gleichverteilung)
- $sel_{R.A=S.B} = \frac{1}{|R|}$
  bei Equijoin von R mit S über Fremdschlüssel in S

Ansonsten z.B. Stichprobenverfahren

Kostenabschätzungen

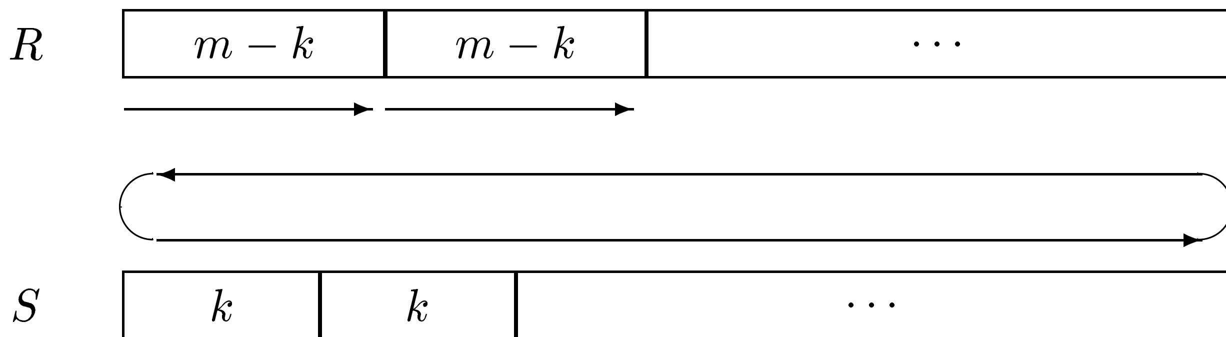
Selektion:

- Brute Force: Lesen aller Seiten von R
- B^{+}-Baum-Index: $t + \lceil sel_{A\theta c} \cdot b_R \rceil$
  - Absteigen der Indexstruktur
  - Lesen der qualifizierenden Tupel
- Hash-Index: für jeden die Bedingung erfüllenden Wert einen Look-up

Kostenabschätzungen

Blockorientierte Nested-Loops

Join:



- Durchlaufen aller Seiten von R: b_R
- Durchläufe der inneren Schleife: $\lceil b_R/(m-k) \rceil$
- Insgesamt: $b_R + k + \lceil b_R/(m-k) \rceil \cdot (b_S - k)$
- minimal, falls $k = 1$ und R die kleinere Relation

„Tuning“ von Datenbankanfragen

- viele DBMS-Produkte bieten unterschiedliche Optimierungslevel an
- Fast alle DBMS-Produkte haben beute u.a. einen kostenbasierten Optimierer
- Der kostenbasierte Optimierer benötigt Statistiken über die sepeicherten Daten, wie z.B.
  - Kardinalitäten der Relationen
  - Attributverteilungen (Histogramme) für Selektivitätsabschätzungen
  - Größe der Tupel
  - Clustering der Tupel
  - Indexkonfiguration
  - etc
- Die Datenbankadministratoren müssen die Generierung der Statistiken explizit anstoßen. Dazu dient z.B. in Oracle7 der Befehl

  analyze table Professoren **compute statistics for table**;
- in DB2:

  runstats on table ...

Analysieren der Auswertungspläne

- Man kann sich die generierten Anfrageauswertungspläne anzeigen lassen
- Dazu gibt es den **explain plan**-Befehl

```
explain plan for
   select distinct s.Semester
   from Studenten s, hören h, Vorlesungen v, Professoren p
   where p.Name = 'Sokrates' and v.gelesenVon = p.PersNr and
         v.VorlNr = h.VorlNr and h.MatrNr = s.MatrNr;
```

Beispiel-Plan

```
SELECT STATEMENT   Cost = 37710
  SORT UNIQUE
    HASH JOIN
      TABLE ACCESS FULL STUDENTEN
      HASH JOIN
        HASH JOIN
          TABLE ACCESS BY ROWID PROFESSOREN
            INDEX RANGE SCAN PROFNAMEINDEX
          TABLE ACCESS FULL VORLESUNGEN
        TABLE ACCESS FULL HOEREN
```