FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

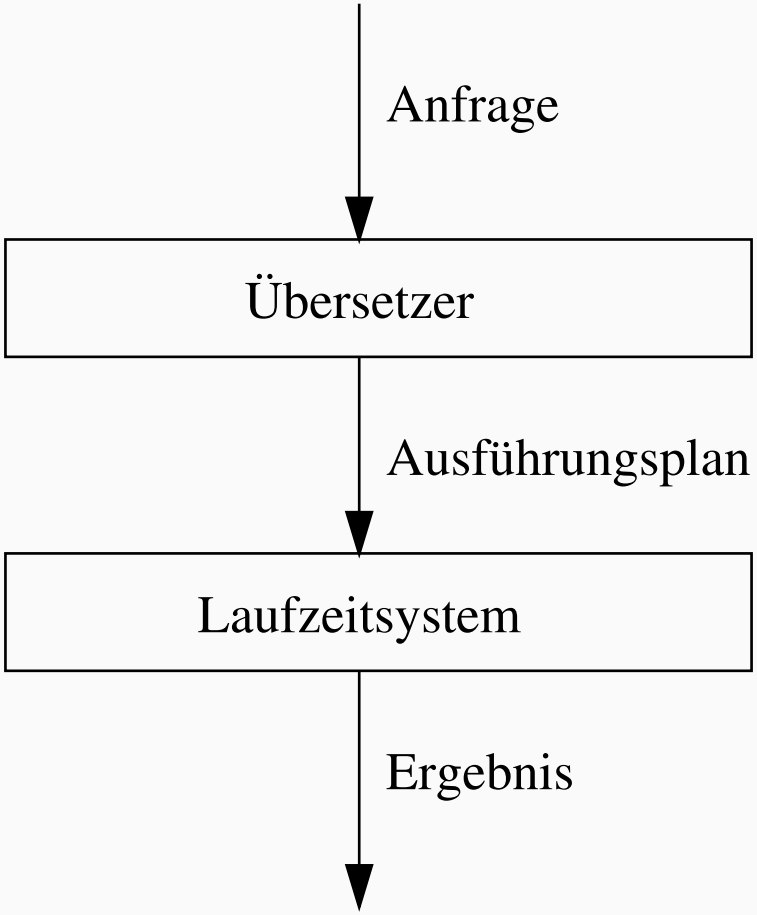
Datenbanksysteme I

Prof. Dr. Viktor Leis

Professur für Datenbanken und Informationssysteme

Anfragebearbeitung

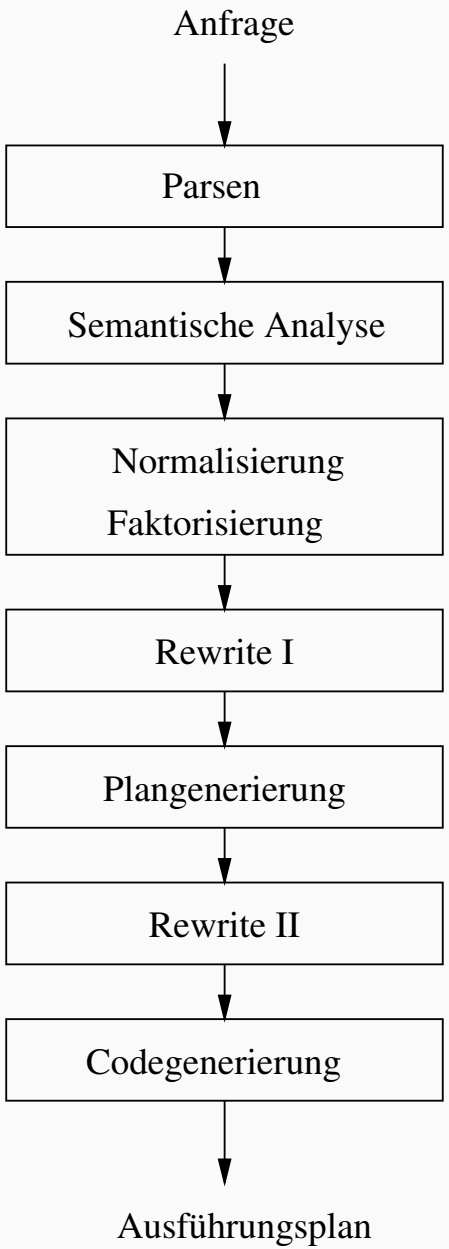
Übersicht


Anfrage
Übersetzer
Ausführungsplan
Laufzeitsystem
Ergebnis

Übersetzung

- SQL ist deklarativ, irgendwann muss Anfrage aber für Laufzeitsystem in etwas prozedurales übersetzt werden
- DBMS übersetzt SQL in eine interne Darstellung
- Ein weit verbreiteter Ansatz ist die Übersetzung in eine relationale Algebra

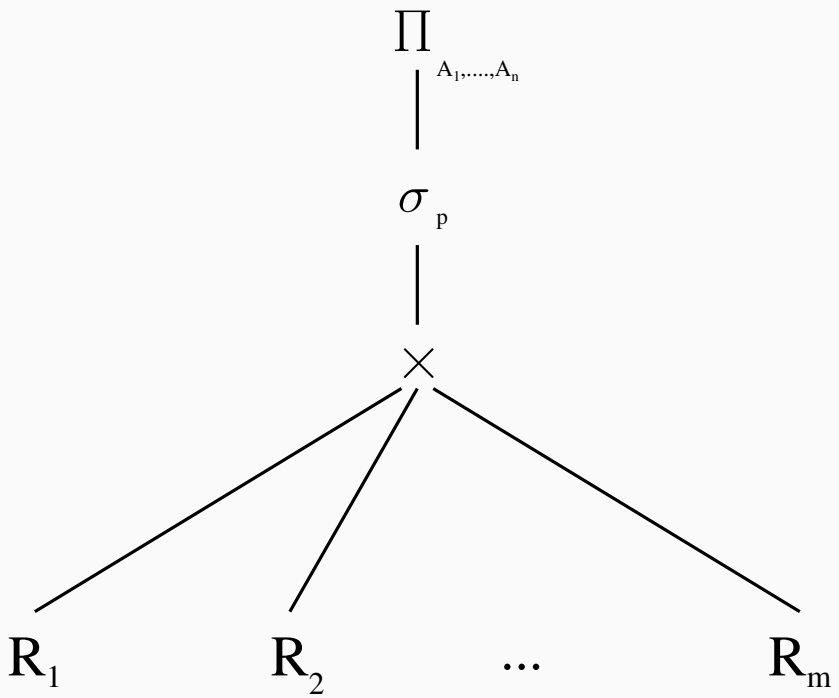
Übersetzung(2)



Kanonische Übersetzung

- Es gibt eine Standardübersetzung von SQL in relationale Algebra
- Algebraausdrücke werden oft auch graphisch repräsentiert

select $A_1, \ldots, A_n$
from $R_1, \ldots, R_m$
where p



Erweiterte Übersetzung

Erweiterungen zur ”klassischen” kanonischen Übersetzung:

- **select** a, **sum**(d) **as** s **from** ... **group by** a,b,c
  - $\Pi_{a,s}(\Gamma_{a,b,c;s:sum(d)}(C))$
  - wobei C die kanonische Übersetzung des inneren Teils ist
  - auf die Projektionsklausel achten!
- **select** ... **having** p
  - $\sigma_p(C)$, wobei C die Übersetzung inklusive group by ist
- **select** ... **order by** a, b, c
  - $sort_{a,b,c}(C)$, wobei C die Übersetzung inklusive having ist
  - *sort* tauchte in relationaler Algebra nicht auf weil Relationen unsortiert sind

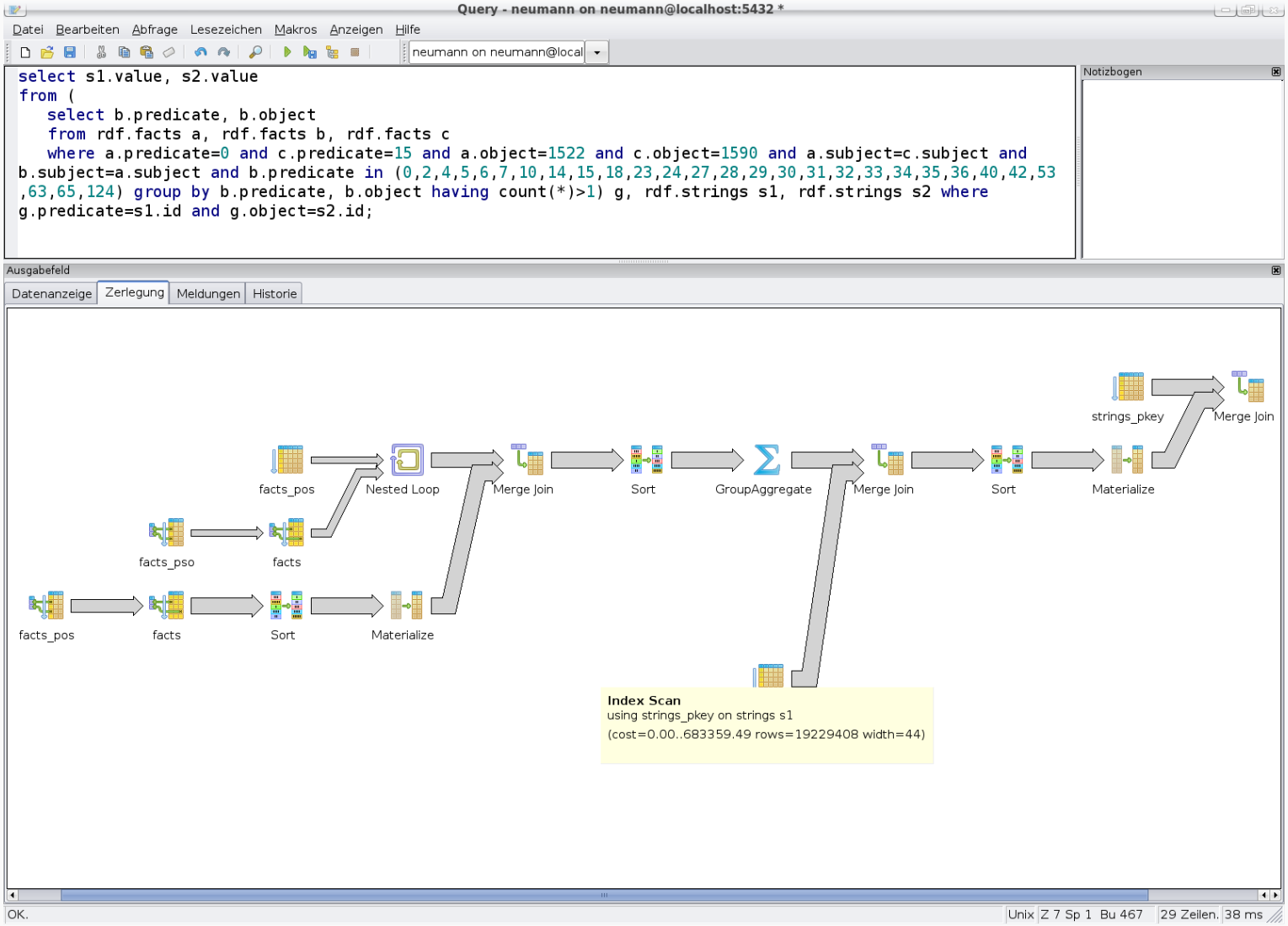
Optimierung

- Kanonischer Plan ist nicht sehr effizient (z.B. enthält er Kreuzprodukte)
- DBMS besitzt Optimierer, um einen Plan in eine effizientere Form zu überführen
- Das Finden eines Plans ist ein sehr schwieriges Problem, immer noch Gegenstand aktueller Forschung

Optimierung(2)

- Was hat ein gewöhnlicher Benutzer mit Anfrageoptimierung zu tun?
- DBMS Optimierer produziert manchmal suboptimale Pläne
- Die meisten DBMSe geben dem Benutzer Einblick in die generierten Pläne
- Benutzer kann generierten Plan analysieren und gegebenenfalls die Anfrage umbauen oder dem DBMS Hinweise zur Ausführung geben

Visualisierung von Plänen



Anfrageoptimierung 1x1

- DBMS kann die Kosten für die Ausführung eines Operators mit Hilfe von Kostenmodellen und Statistiken abschätzen
- Bei der Optimierung eines Plans werden Heuristiken angewandt, alle möglichen Pläne anzuschauen ist viel zu teuer
- Optimierung kann auf verschiedenen Ebenen stattfinden:
  - Logische Ebene
  - Physische Ebene

Logische Ebene

- Ausgangspunkt ist relationaler Algebraausdruck, der nach kanonischer Übersetzung entstanden ist
- Optimierung: Transformation relationaler Algebraausdrücke in äquivalente Ausdrücke (die zu schnellerem Ausführungsplan führen)
- Umformungen sollten so gewählt werden, dass die Ausgaben der einzelnen Operatoren möglichst klein werden

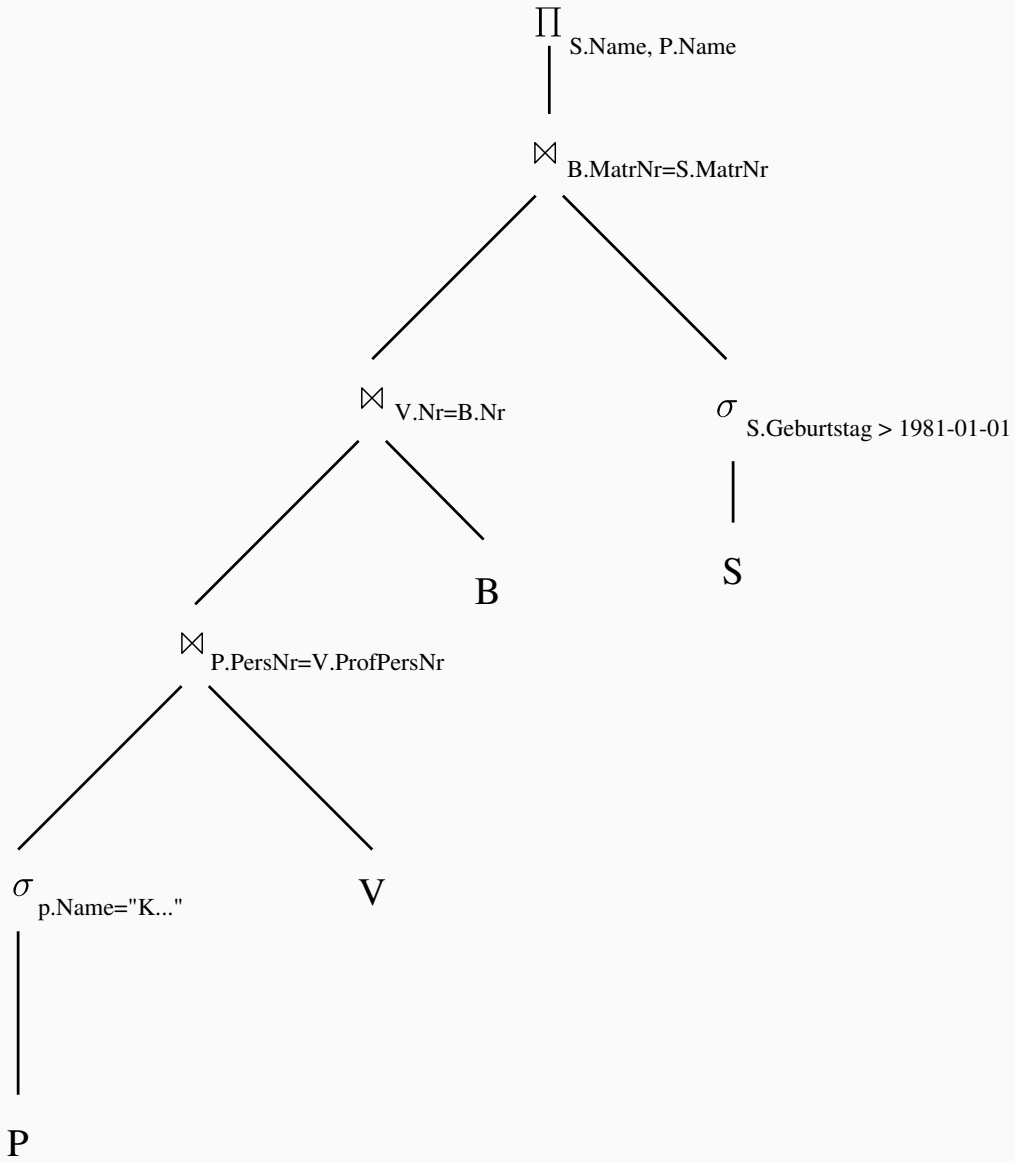
Logische Ebene(2)

- Grundlegende Techniken:
  - Aufbrechen von Selektionen
  - Verschieben von Selektionen nach ”unten” im Plan
  - Zusammenfassen von Selektionen und Kreuzprodukten zu Joins
  - Bestimmung der Joinreihenfolge
  - Einfügen von Projektionen
  - Verschieben von Projektionen nach ”unten” im Plan

Beispielanfrage

```
select  S.Name, P.Name
from    Student S, besucht B, Vorlesung V, Professor P
where   S.MatrNr = B.MatrNr
and     B.Nr = V.Nr
and     V.ProfPersNr = P.PersNr
and     S.Geburtstag > 1981-01-01
and     P.Name = 'Kemper';
```

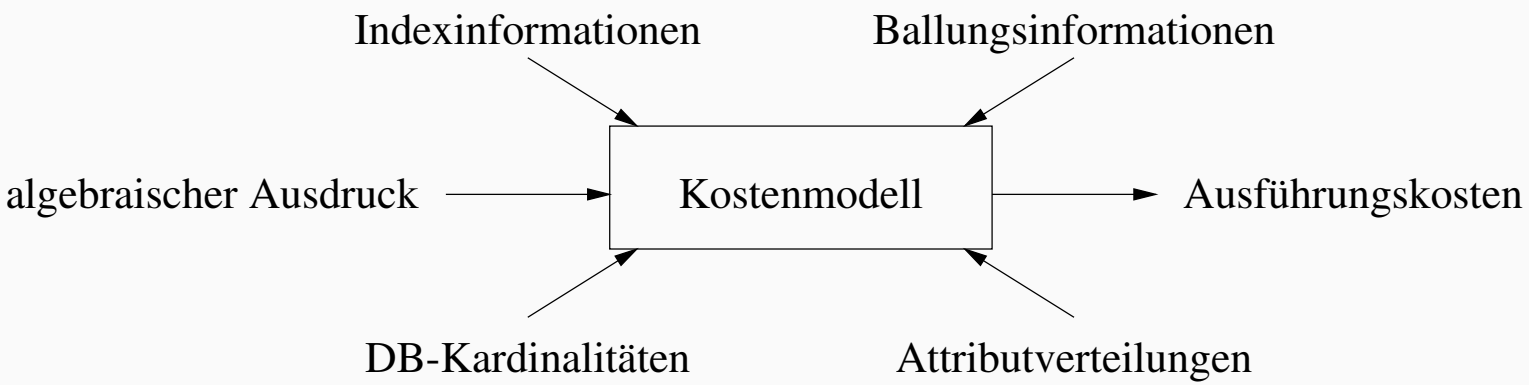
(Optimierter) Anfrageplan



Auszug aus Regeln

$$
\begin{aligned}
R_1 \bowtie R_2 &= R_2 \bowtie R_1 \\
R_1 \cup R_2 &= R_2 \cup R_1 \\
R_1 \cap R_2 &= R_2 \cap R_1 \\
R_1 \times R_2 &= R_2 \times R_1 \\
R_1 \bowtie (R_2 \bowtie R_3) &= (R_1 \bowtie R_2) \bowtie R_3 \\
R_1 \cup (R_2 \cup R_3) &= (R_1 \cup R_2) \cup R_3 \\
R_1 \cap (R_2 \cap R_3) &= (R_1 \cap R_2) \cap R_3 \\
R_1 \times (R_2 \times R_3) &= (R_1 \times R_2) \times R_3 \\
\sigma_{p_1 \wedge p_2 \wedge \cdots \wedge p_n}(R) &= \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots)) \\
\Pi_{l_1}(\Pi_{l_2}(\ldots(\Pi_{l_n}(R))\ldots)) &= \Pi_{l_1}(R) \\
&\text{mit} \quad l_1 \subseteq l_2 \subseteq \cdots \subseteq l_n \subseteq \mathcal{R} = sch(R) \\
\Pi_l(\sigma_p(R)) &= \sigma_p(\Pi_l(R)), \text{ falls } attr(p) \subseteq l
\end{aligned}
$$

Kostenmodelle



Statistiken

- um Kardinalitäten zu schätzen werden Statistiken über die gespeicherten Daten vorgehalten
- diese werden üblicherweise nicht ganz aktuell gehalten, sondern werden periodisch oder wenn sich die Daten substanziell geändert haben neu erzeugt
- damit die Erzeugen/Aktualisierung von Statistiken nicht zu lange dauert, werden sie oft auf Basis von Stichproben generiert

Statistiken (2)

- Statistiken werden meist pro Attribut vorgehalten:
  - Histogramme (gut für Bereichsanfragen)
  - Ausreißer für sehr häufige Werte
  - Domänengröße (Anzahl der eindeutigen Werte im Attribut)
  - manchmal Stichproben
- manche Systeme erlauben auch mehrdimensionale Statistiken, dies muss aber meist explizit angefordert werden

Selektivitäten

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

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

- Join von R mit S:

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

Kardinalitätsschätzung

- auf Basis von statistischen Annahmen und den gespeicherten Statistiken werden Selektivitäten/Kardinalitäten geschätzt
- Gleichverteilung:

$$|\sigma_{R.m=\text{'BMW'}}(R)| = \frac{|R|}{\text{dom}(R.m)}$$

- statistische Unabhängigkeit:

$$|\sigma_{R.m=\text{'BMW'} \wedge R.n=\text{'M5'}}(R)| = |\sigma_{R.m=\text{'BMW'}}(R)| \cdot |\sigma_{R.n=\text{'M5'}}(R)| : |R|$$

- Inklusion (Joins):

$$|R \bowtie_{R.x=S.y} S| = \frac{|R||S|}{\max(\text{dom}(R.x), \text{dom}(S.y))}$$

Joinreihenfolge

Eine der wichtigsten Optimierungen ist die Wahl der Joinreihenfolge

- Joins kommen sehr häufig vor
- Joins sind relativ teuer
- Joins verändern die Zahl der Tupel
- die Joinreihenfolge hat enormen Einfluss auf die Laufzeit

Praktisch alle Datenbanksysteme optimieren die Joinreihenfolge. Das Problem ist NP-hart im Hinblick auf die Anzahl der Relationen $\Rightarrow$ für große Anfragen müssen Heuristiken eingesetzt werden um exponentielle Laufzeit zu vermeiden

Beispiel

Wir betrachten als Beispiel den Nested-Loop-Join. Vereinfachte Kostenfunktion:

$$C(e_1 \bowtie^{NL} e_2) = |e_1||e_2|$$

Statistiken einer Beispielanfrage:

$$\begin{aligned} |R_1| &= 10 \\ |R_2| &= 100 \\ |R_3| &= 1000 \\ sel_{R_1 \bowtie R_2} &= 0.1 \\ sel_{R_1 \bowtie R_3} &= 1 \\ sel_{R_2 \bowtie R_3} &= 0.2 \end{aligned}$$

Beispiel(2)

Kosten für mögliche Pläne:

	$C_{\bowtie nl}$
$R_1 \bowtie R_2$	1000
$R_2 \bowtie R_3$	100000
$R_1 \bowtie R_3$	10000
$(R_1 \bowtie R_2) \bowtie R_3$	101000
$(R_2 \bowtie R_3) \bowtie R_1$	300000
$(R_1 \bowtie R_3) \bowtie R_2$	1010000

- riesige Unterschiede zwischen den Kosten
- Rückschlüsse von Teilen auf die Gesamtkosten schwierig
- $R_1 \bowtie R_3$ scheint zunächst besser zu sein als $R_2 \bowtie R_3$ usw.

Komplexes Optimierungsproblem

Greedy-Heuristiken

- Bestimmung der optimalen Joinreihenfolge ist NP-hart
- für große Anfragen sehr aufwendig
- deshalb häufig Heuristiken

Einfacher Ansatz:

- beginne mit einer Relation
- joine die ”günstigste” Relation dazu
- wiederhole bis alle gejoined sind

Vermeidet die größten Fehler

Greedy-Heuristiken (2)

Einfache Strategie um $R = \{R_1, \ldots, R_n\}$ anzuordnen:

1. wähle $R_i \in R$ mit $|R_i|$ minimal
2. $S = R_i$ und $R = R \setminus \{R_i\}$.
3. solange $|R| > 0$
   - 3.1 wähle $R_i \in R$ mit $C(S \bowtie R_i)$ minimal.
   - 3.2 $S = S \bowtie R_i$. $R = R \setminus \{R_i\}$
4. liefere S als Joinreihenfolge

stark vereinfacht, liefert nur links-tiefe Bäume, oft suboptimal, noch sehr viele (bessere) Varianten

Dynamisches Programmieren

Standardtechnik für die optimal Joinreihenfolge: Dynamisches Programmieren (DP)

> *Optimalitätsprinzip: Die optimale Lösung des Gesamtproblems kann aus optimalen Lösungen von Teilproblemen zusammengesetzt werden.*

Generelles Vorgehen:

- löse zunächst einfache Teilprobleme optimal
- löse immer kompliziertere Probleme und nutze dabei bekannte Lösungen

Dynamisches Programmieren (2)

1 erzeuge eine leere DP Tabelle
2 trage die Basisrelationen als optimale Lösungen der Größe 1 ein
3 für alle Problemgrößen s von 2 bis n
4 für alle gelösten Probleme (l, r), so dass $|l| + |r| = s$
5 ist $l \bowtie r$ möglich?
6 wenn nein, weiter bei 4
7 $T = \{x | x \text{ Relation in } l \text{ oder in } r\}$
8 $p = l \bowtie r$
9 ist $dpTabelle[T]$ leer oder $C(p) < C(dpTabelle[T])$?
10 wenn ja, $dpTabelle[T] = p$
11 liefere $dpTabelle[\{x | x \text{ ist Basisrelation}\}]$ als optimale Lösung

- betrachtet alle relevanten Relationenkombinationen
- Zeile 5 verhindert ungültige Joins, typischerweise auch Kreuzprodukte

Dynamisches Programmieren (3)

- findet die optimale Lösung
- aber im worst case exponentielle Laufzeit
- für sehr große/komplexe nicht mehr durchführbar
- es gibt aber sehr effiziente DP Algorithmen (komplexer als der hier vorgestellte)

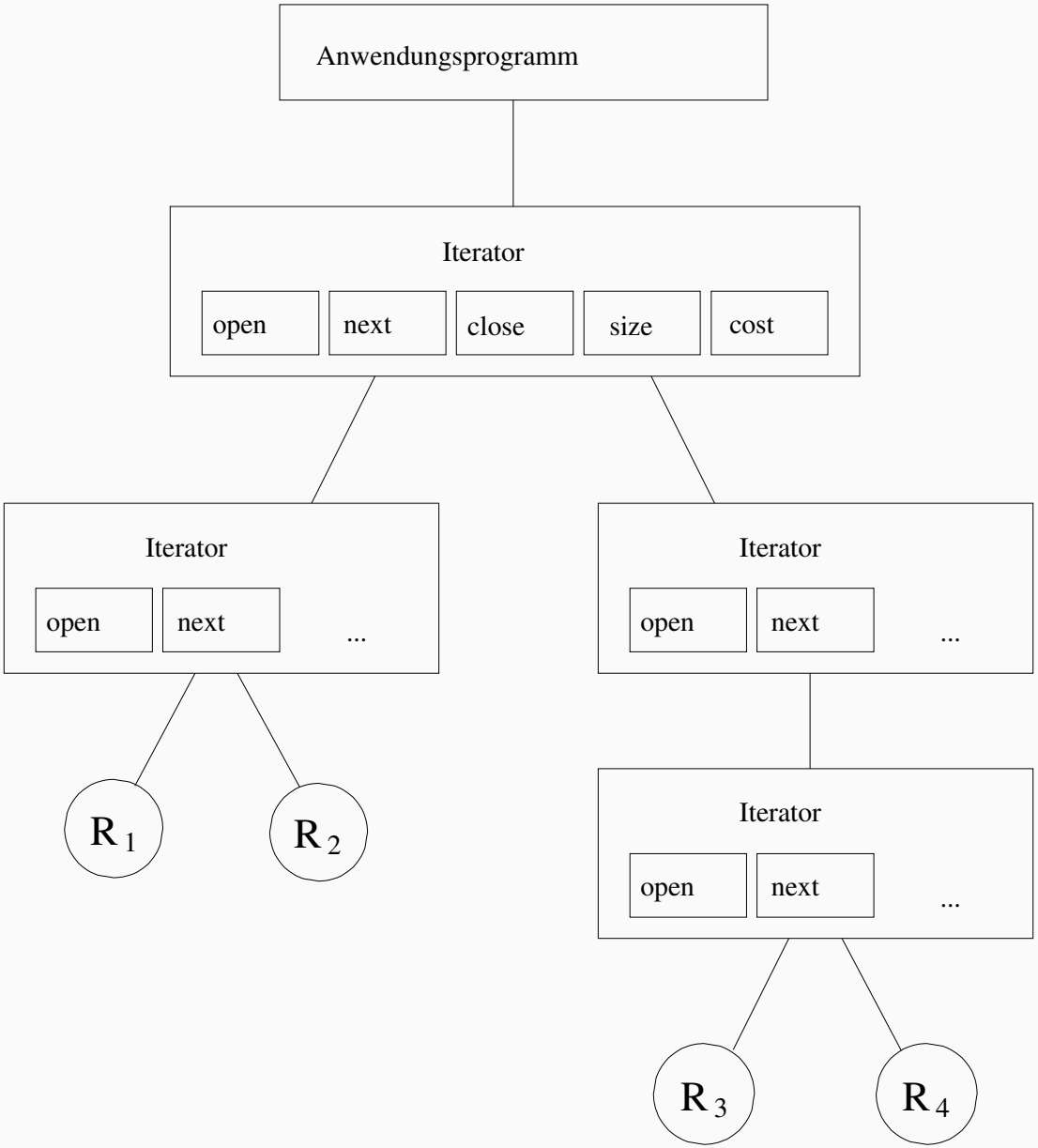
- grundlegende Annahme: Kostenmodell stimmt
- erfordert oft ein `run stats`, `analyze` o.ä.
- selbst dann noch große Schätzfehler möglich

”Echte” Algorithmen wählen nicht nur die Reihenfolge sondern auch die physischen Operatoren

Physische Optimierung

- Man unterscheidet zwischen logischen Algebraoperatoren und physischen Algebraoperatoren
- Physische Algebraoperatoren stellen die Realisierung der logischen dar
- Es kann mehrere physische für einen logischen Operator geben
- Optimierung auf der physischen Ebene bedeutet, einen dieser Operatoren auszuwählen, zu entscheiden, ob Indexe benutzt werden sollen, Zwischenergebnisse zu materialisieren, etc.

Iteratorkonzept



Implementierung Selektion

iterator Scan_p

- open
  - Öffne Eingabe
- next
  - Hole solange nächstes Tupel, bis eines die Bedingung p erfüllt
  - Gib dieses Tupel zurück
- close
  - Schließe Eingabe

Implementierung Selektion(2)

iterator IndexScan$_p$

- open
  - Schlage im Index das erste Tupel nach, das die Bedingung erfüllt
  - Öffne Eingabe
- next
  - Gib nächstes Tupel zurück, falls es die Bedingung p noch erfüllt
- close
  - Schließe Eingabe

Implementierung Join

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

Nested-Loop-Join:

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

Implementierung Join(2)

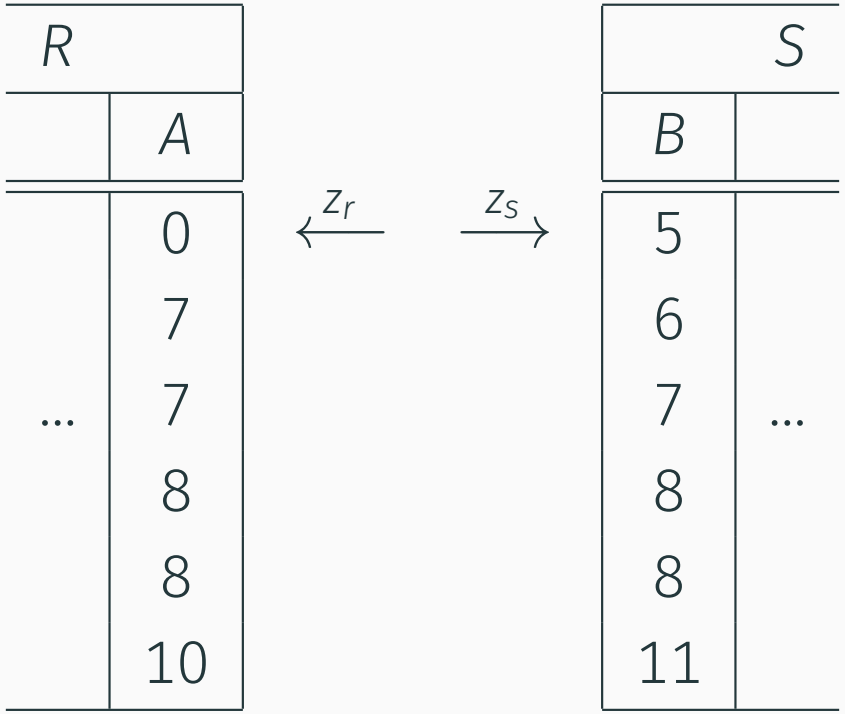
iterator NestedLoop$_p$

- 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
    - Hole nächstes Tupel der linken Eingabe
    - Starte **next** neu
  - Gib den Verbund von aktuellem linken und aktuellem rechten Tupel zurück
- close
  - Schließe beide Eingabequellen

(Sort-)Merge-Join

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

Beispiel:



(Sort-)Merge-Join(2)

iterator MergeJoin_p

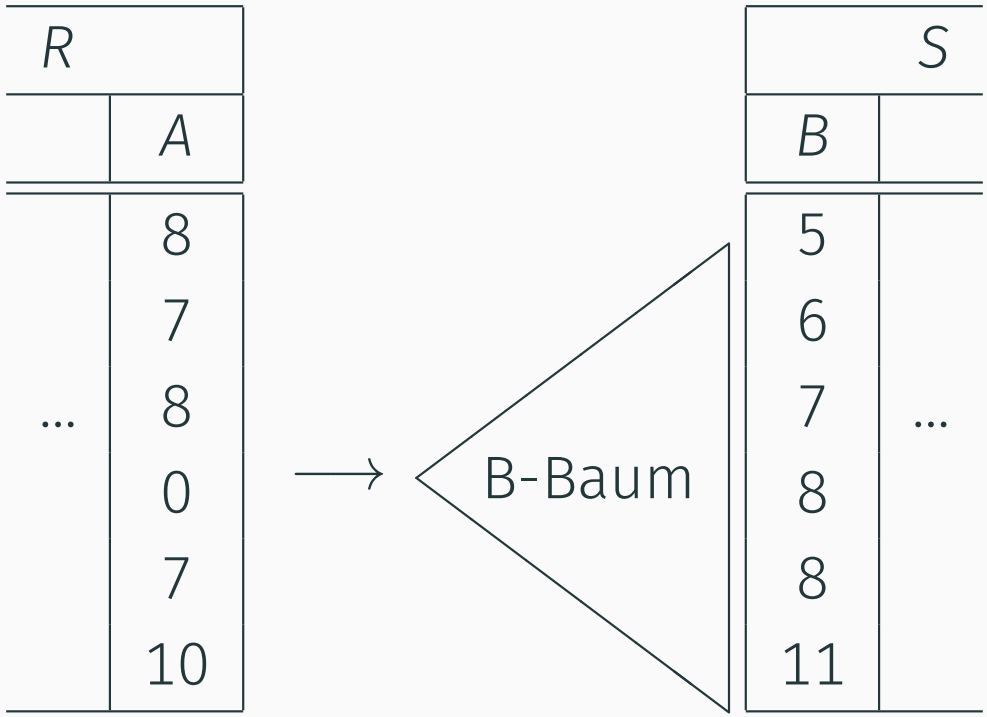
- open
  - Öffne beide Eingaben
  - Setze *akt* auf linke Eingabe
  - Markiere rechte Eingabe
- close
  - Schließe beide Eingabequellen

(Sort-)Merge-Join(3)

iterator MergeJoin_p

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

Index-Join



Index-Join(2)

iterator IndexJoin$_p$

- open
  - Öffne die linke Eingabe
  - Hole erstes Tupel aus linker Eingabe
  - Schlage Joinattributwert im Index nach
- next
  - Bilde Join, falls Index (weiteres) Tupel zu diesem Wert liefert
  - Ansonsten bewege linke Eingabe vor und schlage Joinattributwert im Index nach
- close
  - Schließe die Eingabe

Index-Join(3)

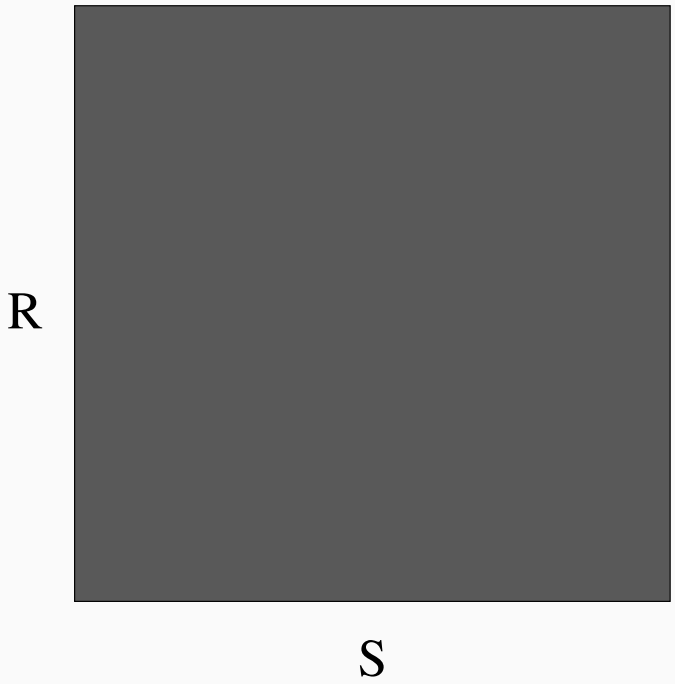
- Nachteile des Index-Joins:
  - auf Zwischenergebnissen existieren keine Indexstrukturen
  - temporäres Anlegen i.A. zu aufwendig
  - Nachschlagen im Index i.A. zu aufwendig

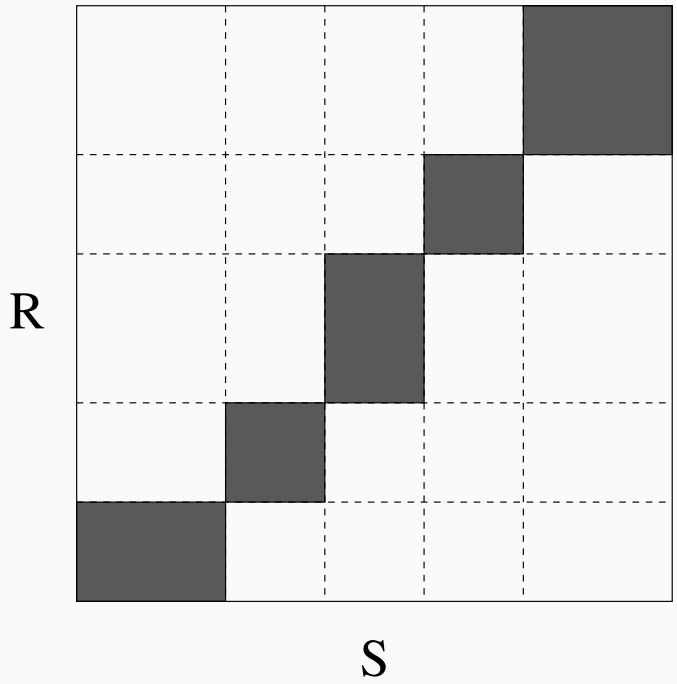
Hash-Join

- Hashtabellen sind oft effizienter als B-Bäume
- Idee: Indexierung der kleineren Eingaberelation mit einer (Hauptspeicher-)Hashtabelle
- Hashtabelle wird nach Ausführung der Anfrage wieder verworfen
- gute Lösung wenn eine Eingaberelation kleiner als der Arbeitsspeicher ist

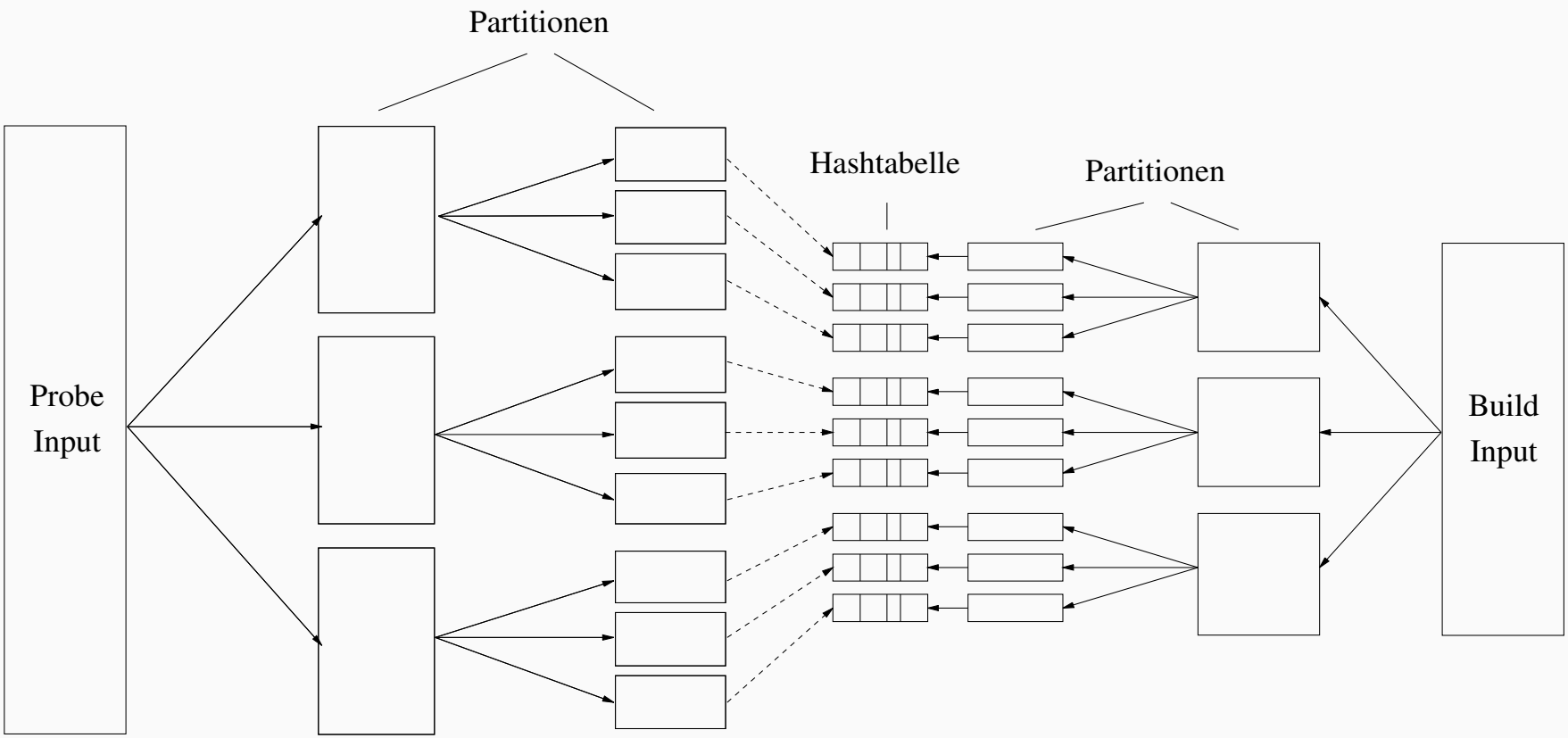
Hash-Join (2)

- Auch wenn beide Eingaberelationen größer als der Hauptspeicher kann Hash-Join benutzt werden
- Idee: Partitionieren der Relationen
- Anlegen von einer Hashtabelle je Partition
- auch bekannt als Grace Hash Join





Partitionierung von Relationen



Join Komplexität

- hängt von Ergebnisgröße ab (kann quadratisch sein)
- für $R \bowtie_{R.x=S.y} S$ mit $R.x$ ist Primärschlüssel und $n = max(|R|, |S|)$:

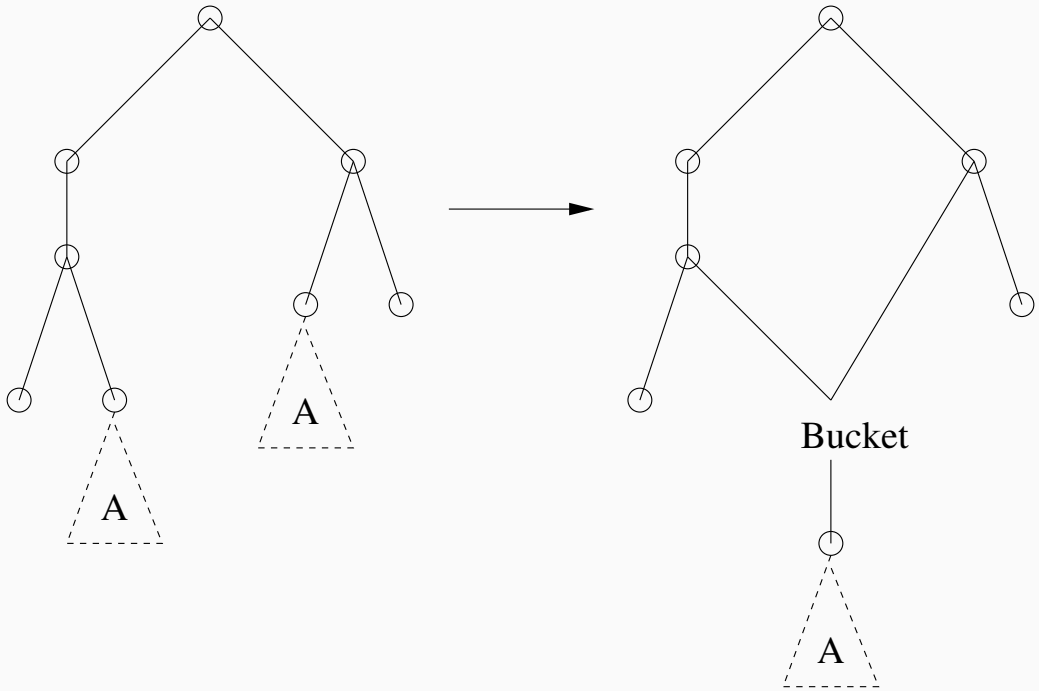
	Schritte	Komplexität
Nested Loop	$\|R\|\|S\|$	$O(n^2)$
Merge Join[1]	$\|R\| + \|S\|$	$O(n)$
Sort+Merge Join	$\|R\| \log \|R\| + \|S\| \log \|S\| + \|R\| + \|S\|$	$O(n \log n)$
Index Join[2]	$\|S\| \log \|R\|$	$O(n \log n)$
Hash Join	$\|R\| + \|S\|$	$O(n)$
Grace Hash Join	$\|R\| + \|S\| + \|R\| + \|S\|$	$O(n)$

[1] R bereits sortiert nach R.x, S nach S.y

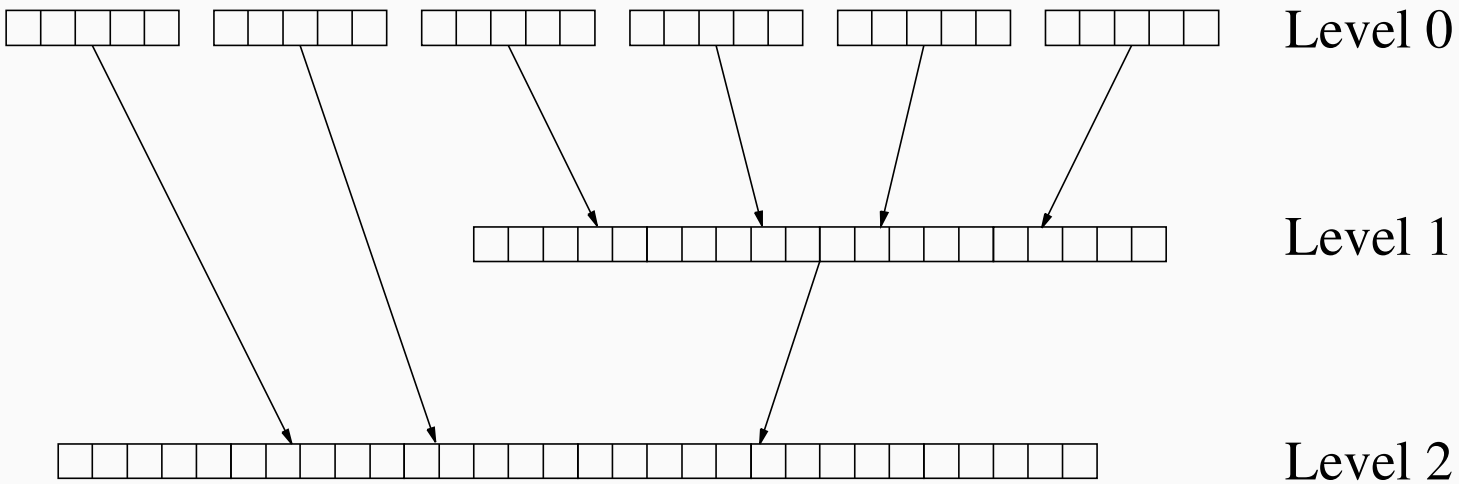
[2] B-Baum-Index auf R.x

Zwischenspeicherung

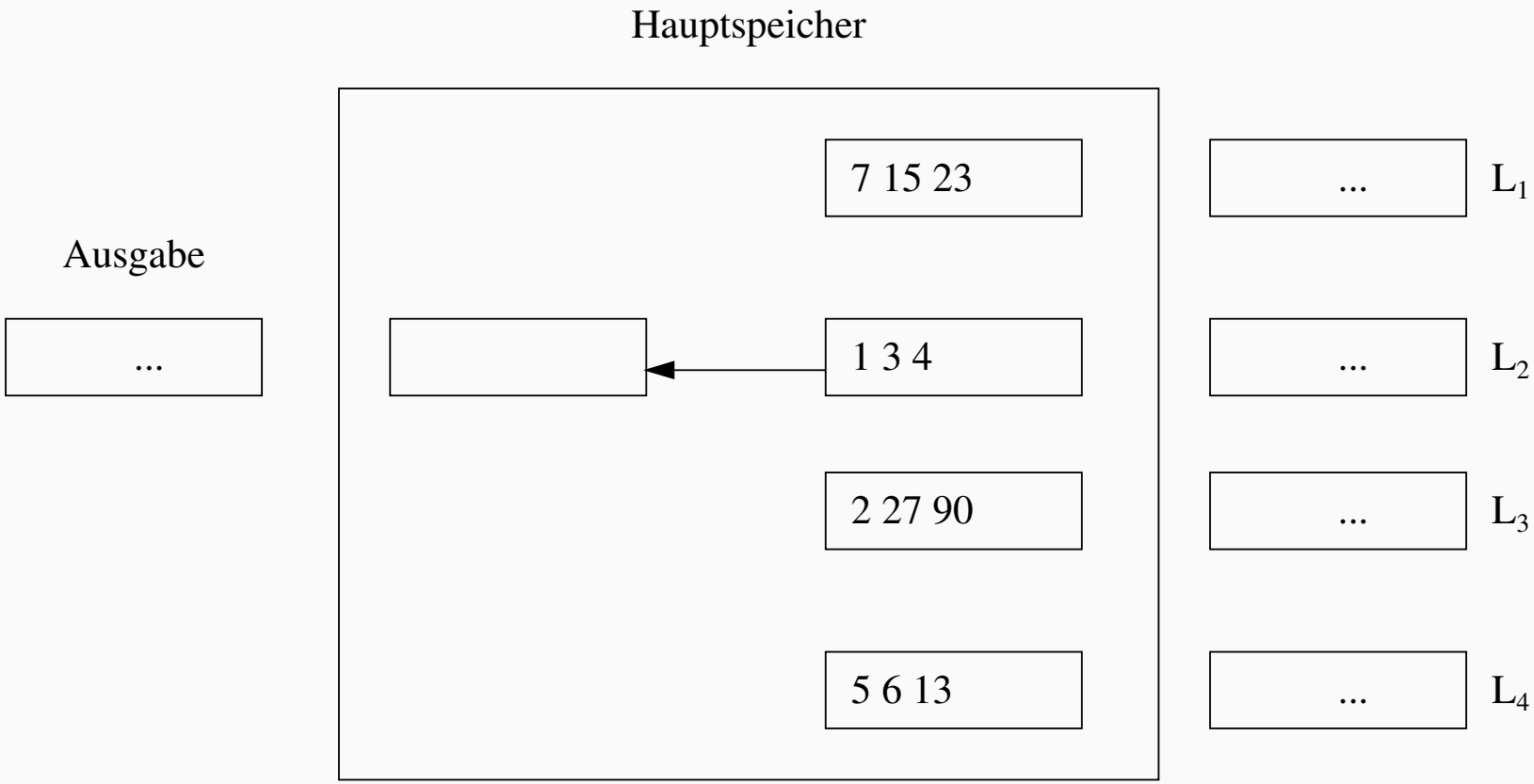
- Wenn mehrere Operationen mit hohem Hauptspeicherverbrauch vorkommen (z.B. Hash-Join)
- Wenn gemeinsame Teilausdrücke eliminiert werden sollen



Mergesort



Ein Merge-Vorgang



Zusammenfassung

- Anfragebearbeitung und -optimierung sind wichtige Aufgaben eines DBMS
- Es hilft auch als Benutzer Ahnung davon zu haben, da Entwurfsentscheidungen und Anfrageformulierung einen Einfluss auf die Performanz eines DBMS haben

Themen

1. Datenbankentwurf (ER-Modell)
2. relationales Modell (relationale Algebra)
3. SQL (SELECT, CREATE TABLE, INSERT, UPDATE)
4. Relationale Entwurfstheorie (funktionale Abhängigkeiten, Normalformen)
5. physische Datenorganisation (B+Bäume)
6. Transaktionen (ACID, grundlegende Funktionsweise)
7. Anfragebearbeitung (kanonische Übersetzung, logische Optimierung, Joinalgorithmen)

Relationale Datenbanksysteme: Drei Ideen

1. normalisierte Speicherung von Daten in Relationen
   - effiziente Änderungen der Daten
   - keine Verschwendung von Speicherplatz
2. deklarative Anfragen (SQL)
   - Beantwortung *beliebiger* Anfragen
   - automatische Anfrageoptimierung
3. Transaktionen
   - Sicherstellung von Persistenz
   - ermöglicht gleichzeitiger Datenzugriff

Hierarchisches Datenmodell

- bevor das relationale Modell erfunden wurde, gab es hierarchische Datenbanken
- moderne Inkarnation: JSON, XML
- Beispiel JSON 1 (Vorlesungen unter Studenten):

```
[{ "name": "Xenokrates", "semester": 18, "vorlesungen":
  [ {"titel": "DBMS I", "SWS": 6} ] },
 { "name": "Jonas", "semester": 12, "vorlesungen":
  [ {"titel": "DBMS I", "SWS":6}, {"titel": "DBMS 2", "SWS":6}]}]
```

- Beispiel JSON 2 (Studenten unter Vorlesungen):

```
[ {"titel": "DBMS I", "SWS": 6, "studenten":
    [{ "name": "Xenokrates", "semester": 18}]},
  {"titel": "DBMS II", "SWS": 6, "studenten":
    [{ "name": "Xenokrates", "semester": 18},
     { "name": "Jonas", "semester": 12 }]}]
```

- Redundanz

Netzwerkmodell

- Turing Award für Charles Bachman, 1973
- moderne Inkarnation: Modellierung in den meisten objektorientierten Programmiersprachen
- Beispiel:

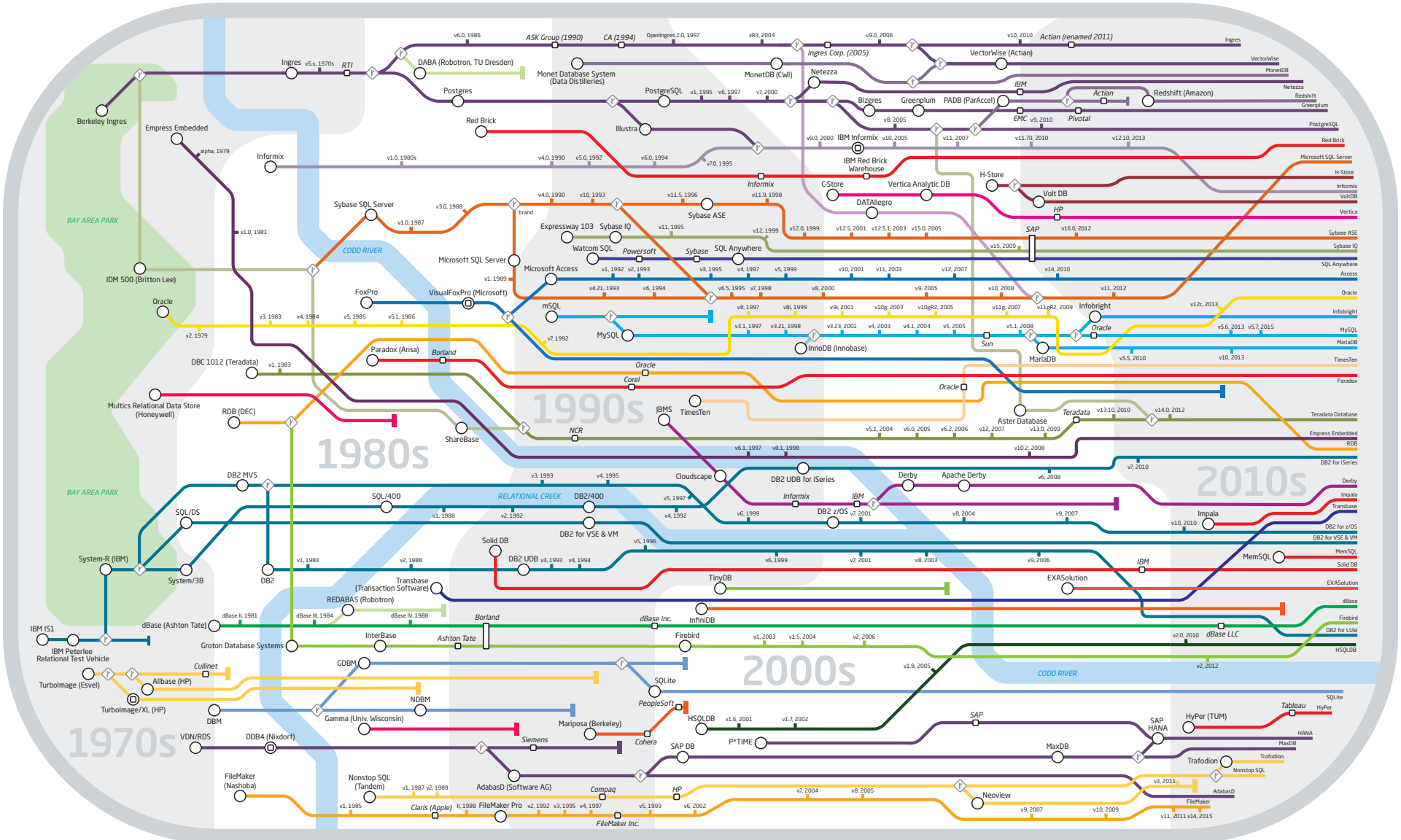
```
class Vorlesung { String titel; int SWS; }
class Professor { String name; String rang; Vorlesung v[]; }
class Student { int matrnr; String name; int semester; }
class Pruefung { Vorlesung v; Professor p; Student s; float note;
```

- komplexe Anfragen, komplexe Datenbanksysteme

Geschichte Relationaler Datenbanksysteme

1. Idee durch Edgar F. Codd 1970 (IBM Research), Turing Award 1981
2. 1980: erste kommerzielle Produkte (Oracle, IBM, Ingres)
3. 1990: Technologie setzt sich durch
4. 2000: Dominanz der “Dinosaurier” (Oracle, Microsoft, IBM)
5. 2010-jetzt: viele neue Systeme (auch NoSQL)

Genealogy of Relational Database Management Systems



Felix Naumann, Jana Bauckmann, Claudia Exeler, Jan-Peer Rudolph, Fabian Tschirschnitz
Contact - Hasso Plattner Institut, University of Potsdam, felix.naumann@hpi.de
Design - Alexander Sandt Grafik-Design, Hamburg

Version 6.0 - October 2018
https://hpi.de/naumann/projects/rdbms-genealogy.html

NoSQL, Key/Value Stores, Document Stores

- Hintergrund: relationale Datenbanksysteme und SQL sind (angeblich) zu kompliziert und ineffizient
- “Lösung”: hierarchisches Datenmodell, kein Schema, keine Transaktionen, keine deklarativen Anfragen, keine Normalisierung
- Datenbanksystem speichert komplexe Objekte (z.B. JSON)
- Zugriff zu Objekten über definierte Schlüssel
- “schema-free” bedeutet, dass das Schema implizit in den Anfragen steckt
- Beispiele: Redis, MongoDB

Wintersemester

- Datenbanksystem*implementierung* (2V+2U)
- Datenbanksysteme Projekt (2P)
- Datenbanksysteme (Spezializierung) (2V)
- Datenbanksysteme I (2V+2U)
- Seminar Datenbanksysteme

Sommersemester

- Datenbanksysteme II (2V+2U)
  - Fortgeschrittenes SQL (Window-Funktionen, rekursive Ausdrücke, Dekorrelation von Anfragen)
  - Recovery (ARIES)
  - Mehrbenutzersynchronisation (2 Phase Locking)
  - Verteilte Datenbanken (Two-Phase Commit)
  - Anfrageoptimierung (Join-Optimierungsalgorithmen)
- Seminar “Moderne Datenbanksysteme”
- Software-Entwicklungsprojekt
- Datenbanksysteme I (2V+2U)