



FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Datenbanksysteme I

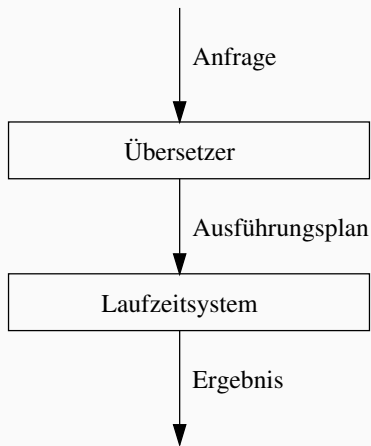
Datenbanken und Informationssysteme

Prof. Dr. Viktor Leis

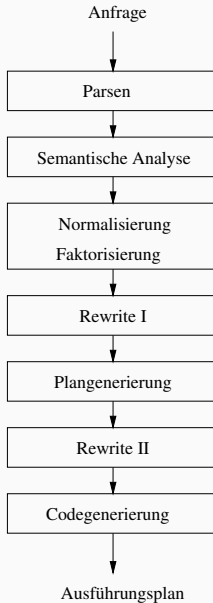
WS 2019/2020

Professur für Datenbanken und Informationssysteme

Anfragebearbeitung



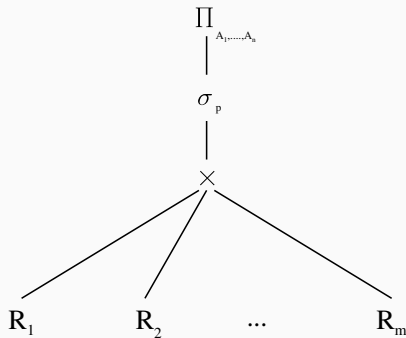
- SQL ist deklarativ, irgendwann muss Anfrage aber für Laufzeitsystem in etwas prozedurales übersetzt werden
- DBMS übersetzt SQL in eine interne Darstellung
- Ein weit verbreiteter Ansatz ist die Übersetzung in eine relationale Algebra



Kanonische Übersetzung

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

```
select A1, ..., An  
from R1, ..., Rm  
where ρ
```



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

- 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

- 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

Query - neumann on neumann@localhost:5432 *

```
select s1.value, s2.value
from (
  select b.predicate, b.object
  from rdf.facts a, rdf.facts b, rdf.facts c
  where a.predicate=0 and c.predicate=15 and a.object=1522 and c.object=1590 and a.subject=c.subject and
  b.subject=a.subject and b.predicate in (0,2,4,5,6,7,10,14,15,18,23,24,27,28,29,30,31,32,33,34,35,36,40,42,53,65,124) group by b.predicate, b.object having count(*)>1) g, rdf.strings s1, rdf.strings s2 where
g.predicate=s1.id and g.object=s2.id;
```

Ausgabefeld

Datenanzeige | Zerlegung | Meldungen | Historie

The diagram illustrates the execution plan for the provided query. It starts with an **Index Scan** operation on the `strings` table, which is highlighted in a yellow box. The scan uses the `strings_pkey` index on the `strings s1` table, with a cost of 0.00, 683359.49 rows, and a width of 44. The data then flows through a **Sort** operation, followed by a **GroupAggregate** operation. This is followed by a **Merge Join** operation, which then leads to another **Sort** operation. The data then passes through a **Materialize** operation, which is connected to a **Merge Join** operation. The plan also shows a **Nested Loop** join between `facts_pos` and `facts` tables, and another **Merge Join** operation connecting the `strings_pkey` table to the `facts` table.

Index Scan
Using strings_pkey on strings s1
(cost=0.00..683359.49 rows=19229408 width=44)

OK

Unix/2.7 Sp.1 Bu.467 29 Zeilen, 38 ms

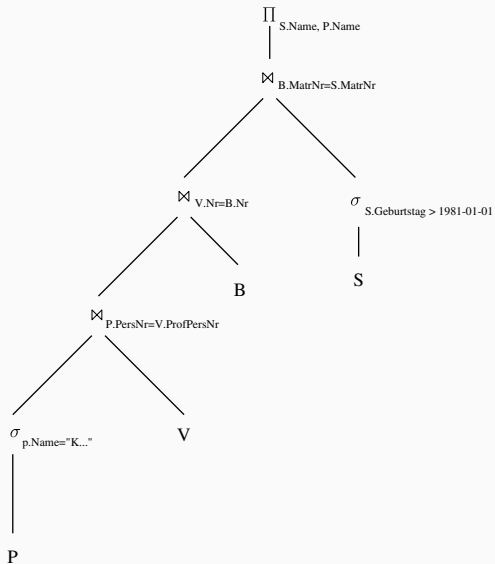
- DBMS kann die Kosten für die Ausführung eines Operators mit Hilfe von Kostenmodellen und Statistiken abschätzen
- Bei der Optimierung eines Plans werden Heuristiken angewandt, alle möglichen Pläne anzuschauen ist viel zu teuer
- Optimierung kann auf verschiedenen Ebenen stattfinden:
 - Logische Ebene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

- 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

- 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

```
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';
```

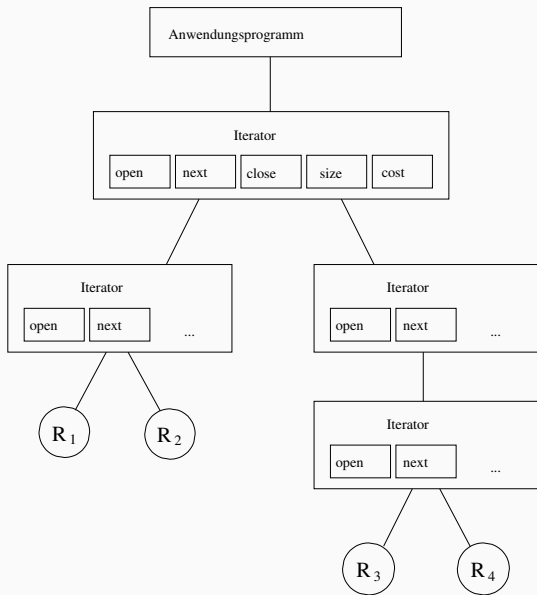
(Optimierter) Anfrageplan



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

- 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



iterator Scan_p

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

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

- 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)

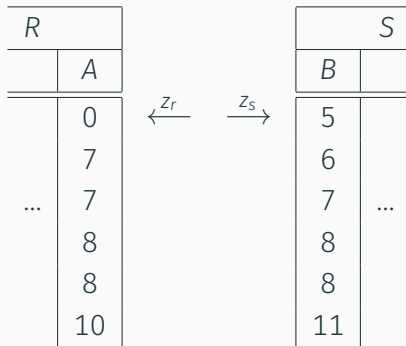
iterator NestedLoop_p

- 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:



iterator MergeJoin_p

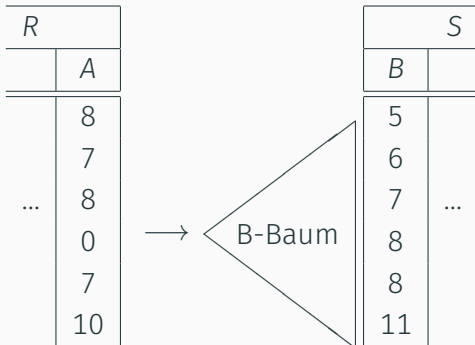
- **open**
 - Öffne beide Eingaben
 - Setze *akt* auf linke Eingabe
 - Markiere rechte Eingabe
- **close**
 - Schließe beide Eingabequellen

(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?
 - ⇒ Nein, dann setze andere Eingabe auf Markierung zurück
 - ⇒ Ansonsten markiere die andere Eingabe

Index-Join



iterator IndexJoin_p

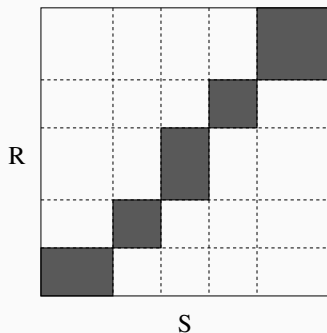
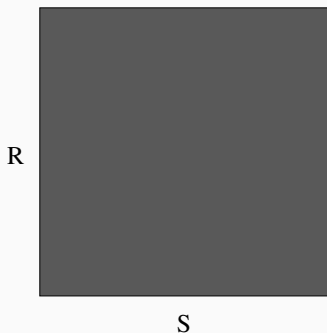
- **open**
 - Öffne die linke Eingabe
 - Hole erstes Tupel aus linker Eingabe
 - Schlage Joinattributwert im Index nach
- **next**
 - Bilde Join, falls Index (weiteres) Tupel zu diesem Wert liefert
 - Ansonsten bewege linke Eingabe vor und schlage Joinattributwert im Index nach
- **close**
 - Schließe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

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

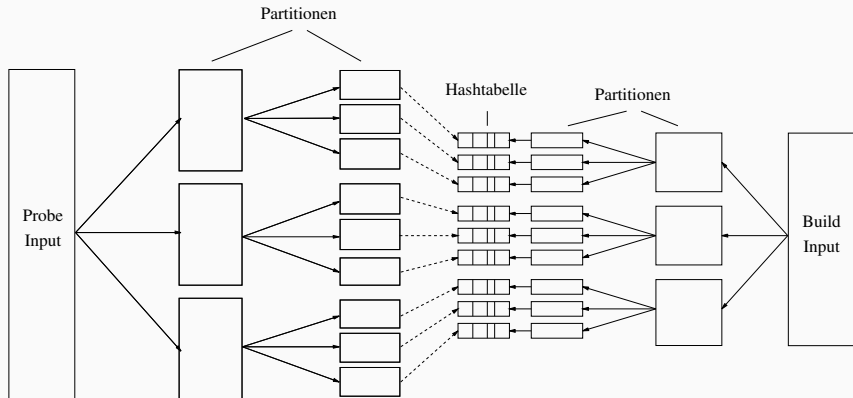
- Hashtabellen sind oft effizienter als B-Bäume
- Idee: Indexierung der kleineren Eingaberelation mit einer (Hauptspeicher-)Hashtabelle
- Hashtabelle wird nach Ausführung der Anfrage wieder verworfen
- gute Lösung wenn eine Eingaberelation kleiner als der Arbeitsspeicher ist

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

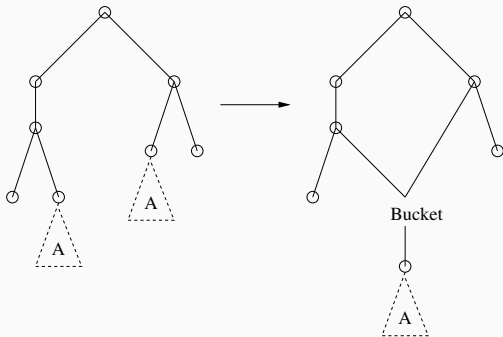


Partitionierung von Relationen

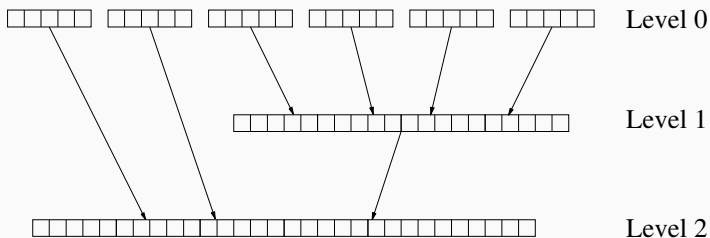


Zwischenspeicherung

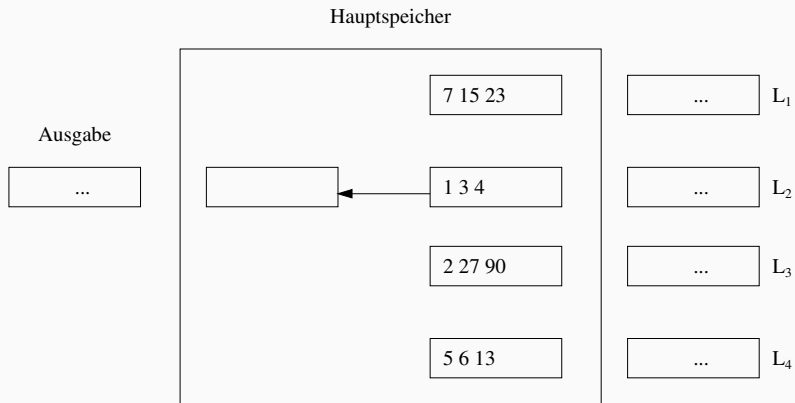
- Wenn mehrere Operationen mit hohem Hauptspeicherverbrauch vorkommen (z.B. Hash-Join)
- Wenn gemeinsame Teilausdrücke eliminiert werden sollen

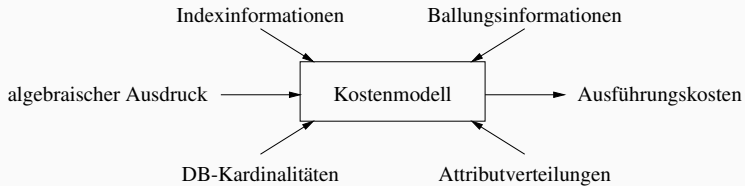


Mergesort



Ein Merge-Vorgang





- 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 substantiell geändert haben neu erzeugt
- damit die Erzeugen/Aktualisierung von Statistiken nicht zu lange dauert, werden sie oft auf Basis von Stichproben generiert

- 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

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

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

- statistische Unabhängigkeit:

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

- ad hoc Konstanten:

$$|\sigma_{R.m \text{ LIKE } 'BM\%'}(R)| = 0.1|R|$$

- Inklusion (Joins):

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

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

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

- Join von R mit S :

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

Eine der wichtigsten Optimierungen ist die Wahl der Joinreihenfolge

- Joins kommen sehr häufig vor (fast immer)
- 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 häufig 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:

$$|R_1| = 10$$

$$|R_2| = 100$$

$$|R_3| = 1000$$

$$sel_{R_1 \bowtie R_2} = 0.1$$

$$sel_{R_1 \bowtie R_3} = 1$$

$$sel_{R_2 \bowtie R_3} = 0.2$$

Beispiel(2)

Kosten für mögliche Pläne:

	C_{int}
$R_1 \bowtie R_2$	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

- 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