IMPLEMENTIERUNG VON OPERATIONEN AUF RELATIONEN

Literatur

A. Kemper, A. Eickler: ‘Datenbanksysteme – Eine Einführung ‘, 8. Auflage Oldenburg Verlag, 2011, ISBN 978-3-486-59834-6 (als E-Book mit dem Übungsbuch über die Informatik Bibliothek herunterladbar!) – Kapitel 8

Priti Mishara, Maragaret H. Eich, ‘Join Processing in Relational Databases’, ACM Computing Surveys, Vol. 24, No. 1, March 1992

Goetz Graefe, ‘Query Evaluation Techniques for Large Databases’, ACM Computing Surveys, Vol. 25, No. 2, June 1993

Abfrage-Auswertungs-Techniken

- Effiziente Algorithmen für die Auswertung von komplexen Abfragen an „große“ Datenbanken.

- Eine „große“ Datenbank in diesem Zusammenhang ist eine Datei mit mehreren (hundert) Megabytes.

Abfrageverarbeitung in einem Datenbanksystem:

Benutzerschnittstelle (User Interface) Abfragesprache
Abfrageoptimierer Abfrageausführungs- Einheit
Dateien und Indizes I/O Puffer Festplatten

Abfrageoptimierer (Query Optimizer)

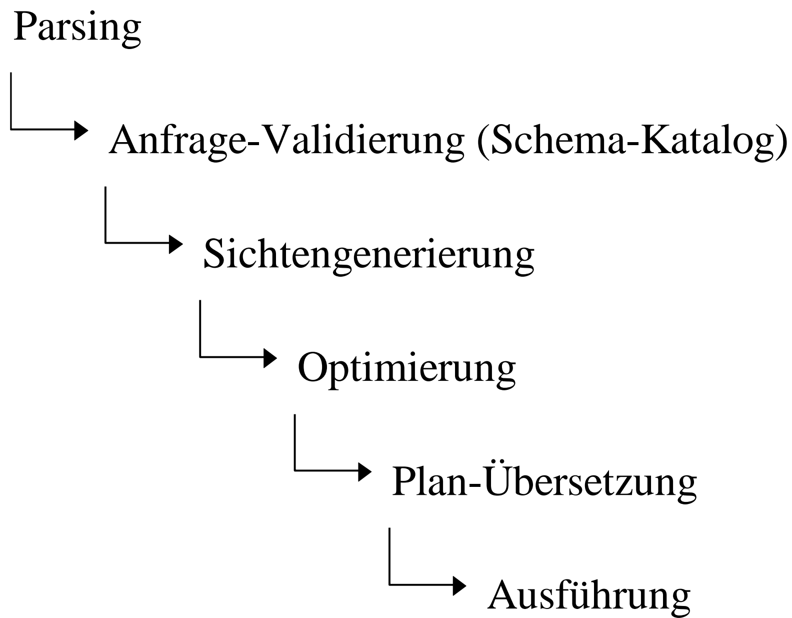
Der Optimierer übersetzt eine Anfrage (SQL) in eine Sequenz von Operationen, die in der Ausführungseinheit oder dem Dateisystem implementiert sind.

Ziel ist es, einen Anfrage-Ausführungsplan (query execution plan) zu finden, der den Verbrauch von CPU, I/O, Speicher etc. minimiert.

Ausführungseinheit

Die Ausführungseinheit ist eine Sammlung von Funktionen und Mechanismen zur Kommunikation und Synchronisation zwischen Operatoren.

Schritte bei der Ausführung einer Abfrage



- Interaktive oder
  Eingebettete Anfragen (Cobol, PL/1, C, Fortran)

- Die Anfragebearbeitung konzentriert sich auf das Extrahieren von Informationen ohne Daten zu verändern.

- Für Updates siehe den Vorlesungsabschnitt über Transaktionen und ACID-Semantik.

  Atomic
  Consistent
  Isolated
  Durable

Anfragebearbeitung:

- Eingabe sind Relationen und Mengen

Alle Implementierungen von Anfrage-Algorithmen iterieren über den Elementen der Eingabemengen.

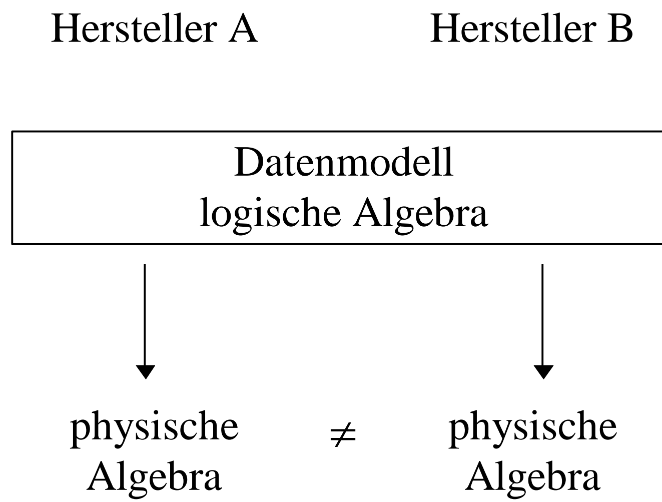
Mengen sind als Sequenzen verwirklicht.

Algorithmen:

→ Physische Algebra → Systemspezifisch

vs.

Logische Algebra → Datenmodell



Beispiel:

Hersteller A: Nested-Loops Join

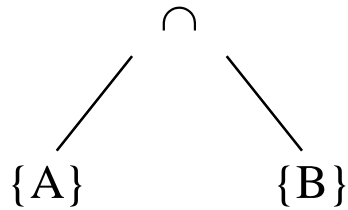
Hersteller B: Nested-Loops Join
&
Merge Join

- Spezifische Algorithmen (mit Kostenfunktion) sind nur physischen Operatoren zugeordnet, nicht aber logischen.

- Ein Ausdruck der logische Algebra ist nicht direkt ausführbar.

- Er muss in einen Ausdruck der physischen Algebra übersetzt werden.

Beispiel:



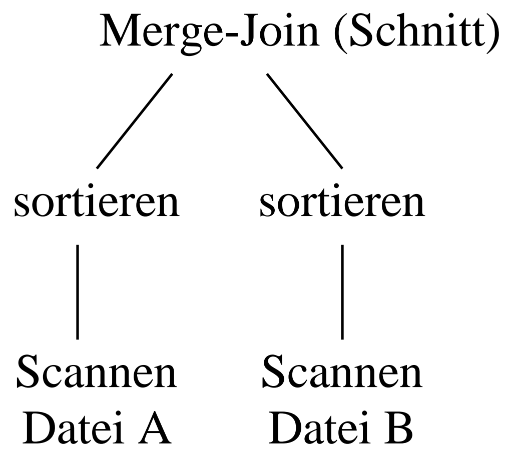


Abbildung von logischen Operatoren auf physische Operatoren

Diese Abbildung ist **schwierig** und **komplex**, denn ...

- Einige Operatoren der physischen Algebra können mehrere logische Operatoren verwirklichen.

  Beispiel: Implementierung von Joins
  (gängige Implementierungen ermöglichen bereits eine Projektion für die Ausgabemenge, jedoch ohne Duplikat-Entfernung – diese Art der Projektion nennt man auch *Delta-Projektion*)

- Einige physische Operatoren verwirklichen nur einen Teil eines logischen Operators.

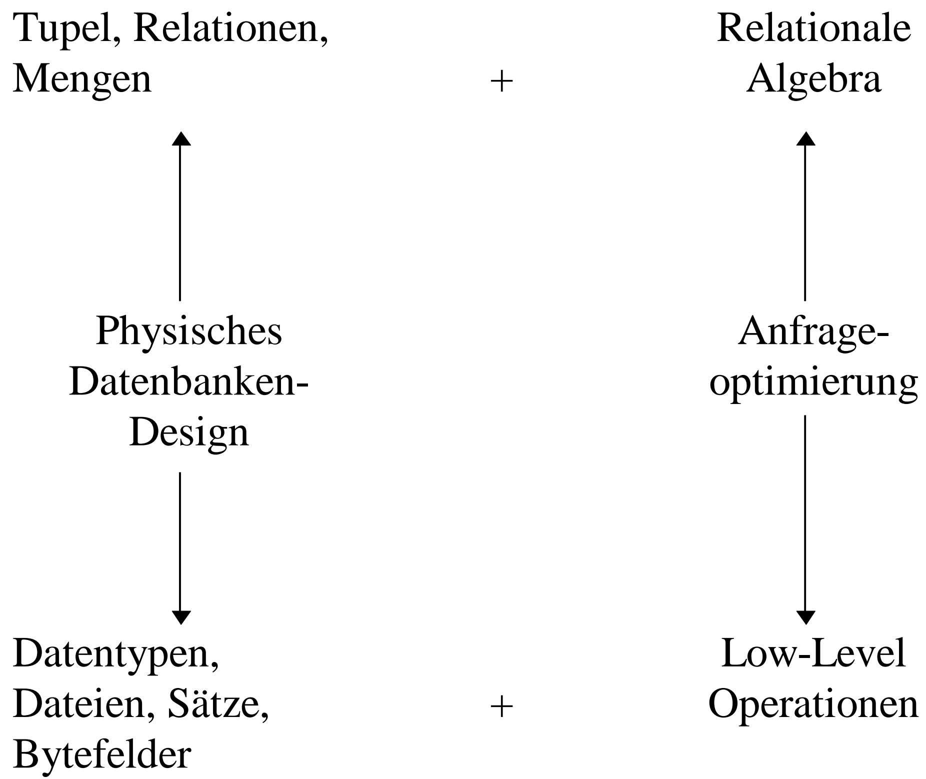
  Beispiel: Ein Algorithmus, der Duplikate eliminiert verwirklicht nur einen Teil des relationalen Projektions-Operators.

- Einige physische Operatoren existieren in der logischen Algebra überhaupt nicht.

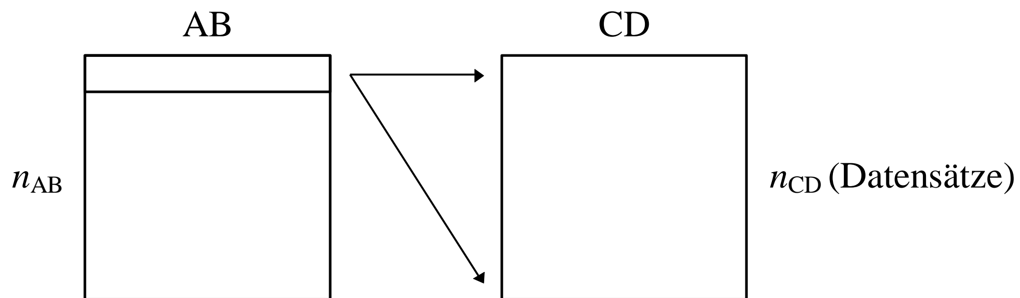
  Beispiel: Sortieren.

- Einige Eigenschaften, die für logische Operatoren gelten, gelten nicht (oder nur teilweise) für physische Operatoren.

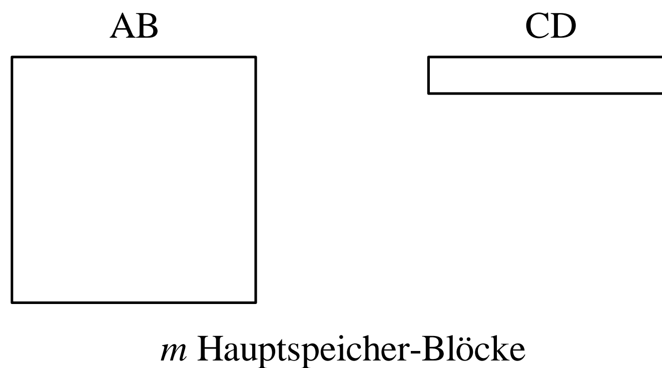
  Beispiel: $\cap, \cup$: Symmetrie und Kommutativität gelten (bezogen auf den Kostenfaktor) nicht, z.B. für Implementierung durch Nested-Loops.


Tupel, Relationen,
Mengen
+
Relationale
Algebra
Physisches
Datenbanken-
Design
Anfrage-
optimierung
Datentypen,
Dateien, Sätze,
Bytefelder
+
Low-Level
Operationen

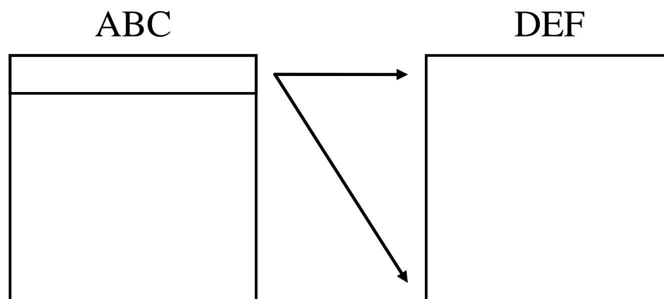
Kartesisches Produkt



Strategie: Lade so viele *Blöcke* von AB wie möglich in den Hauptspeicher und lasse dabei Platz für einen Block von BC.



Kartesisches Produkt



A	B	C
4	32	72,89
7	50	43,54
5	42	53,22
3	70	33,94
6	50	42,74
1	20	23,09

D	E	F
12	2,1	Peter
17	2,3	Meier

A	B	C	D	E	F
4	32	72,89	12	2,1	Peter
4	32	72,89	17	2,3	Meier
7	50	43,54	12	2,1	Peter
7	50	43,54	17	2,3	Meier
5	42	53,22	12	2,1	Peter
5	42	53,22	17	2,3	Meier
3	70	33,94	12	2,1	Peter
3	70	33,94	17	2,3	Meier
6	50	42,74	12	2,1	Peter
6	50	42,74	17	2,3	Meier
1	20	23,09	12	2,1	Peter
1	20	23,09	17	2,3	Meier

- n_{AB}, n_{CD} : Sätze.
- b_{AB}, b_{CD} : Sätze/Block.
- m : Anzahl der Blöcke im Hauptspeicher.
- Anzahl der Block-Zugriffe um AB zu lesen: n_{AB}/b_{AB} .
- CD muss $n_{AB}/(m-1)b_{AB}$ - mal gelesen werden.
  Jedes Mal werden dazu n_{CD}/b_{CD} Zugriffe benötigt.

Anzahl der Block-Zugriffe:

$$\frac{n_{AB}}{b_{AB}} + \frac{n_{AB}}{(m-1) \cdot b_{AB}} \cdot \frac{n_{CD}}{b_{CD}} =$$

$$\frac{n_{AB}}{b_{AB}} \cdot \left(1 + \frac{n_{CD}}{(m-1)b_{CD}}\right)$$

Beispiel:

$$n_{AB} = n_{CD} = 10\,000$$
$$b_{AB} = b_{CD} = 5$$
$$m = 100$$

Anzahl der Zugriffe = 42.400.

Bei 20 Block-Zugriffen pro Sekunde wird dieses kartesische Produkt ca. 35 Minuten benötigen.

Wähle AB als die Relation, mit dem kleineren Quotienten $\frac{n_{AB}}{b_{AB}}$.

Das heißt, die Relation, die in weniger Blocks passt.

IMPLEMENTIERUNG VON JOINS

Gegeben sei folgende Abfrage:

$$\pi_A\left(\sigma_{B=C \wedge D=99}\left(AB \times CD\right)\right)$$

Nach den, in Teil 1 der Vorlesung behandelten, Verfahren kann man diesen **relationalen Ausdruck** zunächst so umformen:

$$\pi_A\left(\sigma_{B=C}\left(AB \times \sigma_{D=99}\left(CD\right)\right)\right)$$

Durch Ersetzen des kartesischen Produktes durch einen natürlichen Verbund ergibt sich folgende optimierte Abfrage:

$$\pi_A\left(AB \underset{B=C}{\bowtie} \sigma_{D=99}\left(CD\right)\right)$$

$$\pi_A\left(AB \underset{B=C}{\bowtie} \sigma_{D=99}(CD)\right)$$

Wie kann diese Abfrage **implementiert** werden?

1. $\bowtie, \sigma, \pi$

oder

2. $\sigma, \bowtie, \pi$ ← ist besser

wenn CD: Index(D)

dann ist $\sigma_{d=99}$ (CD) schnell.

sonst Scanne CD (einmal) mit

$\frac{n_{CD}}{b_{CD}}$ Blockzugriffen

→ { Menge von Tupeln aus CD mit D=99 }

<u>Beispiel</u>:

$n_{AB} = n_{CD} = 10.000$

$b_{AB} = b_{CD} = 5$

m = 100 (Hauptspeicher)

AB×CB:

→ 42.000 Blockzugriffe

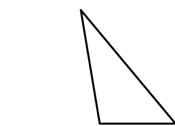
bei 20 Blocks/sec ≈ 35 Minuten

Wenn aber die Selektion vorher ausgeführt wird:

$\frac{n_{CD}}{b_{CD}} = 2.000$

C	D
	99

← $\sigma_{D=99}(CD)$



nur dieser Teil ist von Interesse!

Die Menge von C-Werten wird verglichen mit

einer Menge von B-Werten (AB)

- wenn # C-Werte klein (passt in den Hauptspeicher)

  $\wedge$ AB hat einen Index(B)

  dann benötigt $AB \underset{B=C}{\bowtie} \sigma_{D=99}(CD)$ **wenige** Blockzugriffe.

- wenn # C-Werte klein, kein Index

  dann Scanne AB (einmal) mit

  $\frac{n_{AB}}{b_{AB}}$ = 2.000 Blockzugriffen

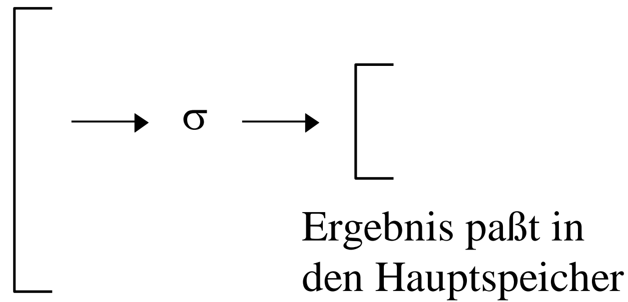
2.000 + 2.000 = 4.000 $\approx$ 3,5 Minuten

vs.

AB x CD = 42.400 $\approx$ 35 Minuten

→ Umformen der Abfrage **!!**

Bisherige Annahme:



Jetzt:

$$AB \underset{B=C}{\bowtie} CD$$

AB und CD passen nicht in den Hauptspeicher

→ Implementierungsstrategien (Join)

- Nested-Loops Join
- Sort-Merge Join
- Hash Join

Generelle Strategien (physikalisch)

- Vorverarbeitung der Dateien

z.B. - Sortieren,
- Erzeugen von Indizes.

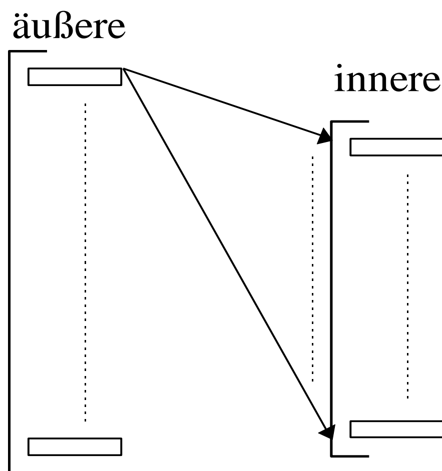
- Berechnung von Optionen vor der Ausführung

z.B. AB $\times$ BC

$$\frac{n_{AB}}{b_{AB}} \quad vs. \quad \frac{n_{CD}}{b_{CD}}$$

Nested-Loops Join

$$AB \underset{B=C}{\bowtie} CD \quad \Big| \quad R \underset{r(a)\,\theta\, s(b)}{\bowtie} S \qquad \Big|\ \theta \in \{=, \neq, \leq, \geq, <, >\}$$



Für jedes Tupel der äußeren Relation werden alle Tupel der inneren Relation gelesen und verglichen.

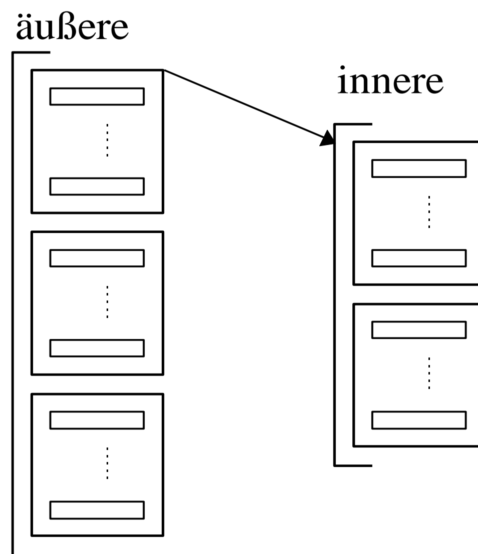
$\forall$ s $\in$ S do
{ $\forall$ r $\in$ R do
{ if r(a) θ s(b)
then r · s
Speichern in Q } }

-- Einfachste Methode --

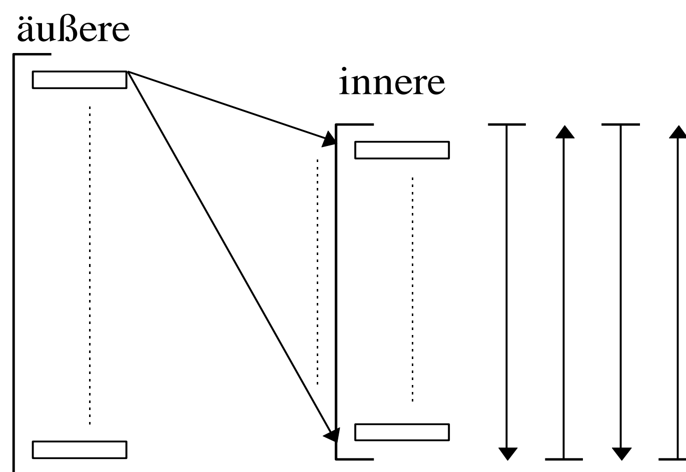
Effizienz: Die innere Relation sollte die mit der größeren Kardinalität sein.

Nested Loops → Nested Blocks

Da Sätze immer nur in ganzen Blöcken gelesen werden können, liegt folgende Optimierung nahe:



„Rocking" der inneren Relation



Performanz:

$$O(n \times m)$$

Anwendung:

- Unpassend für sehr große Relationen

- Gut für parallele Implementierung
  (Hardware Datenbankmaschinen)

Sort-Merge Join

$$AB \underset{B=C}{\bowtie} CD$$

Schritt 1:

- Sortiere A<u>B</u> ,
- Sortiere <u>C</u>D.

Schritt 2:

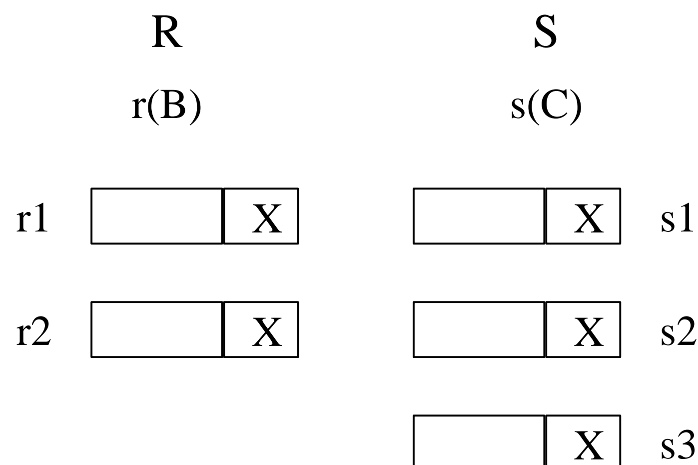
- Vergleiche & Mische

$$\frac{n_{AB}}{b_{AB}} + \frac{n_{CD}}{b_{CD}}$$

Falls B und C Schlüssel:

```
read 1st Tuple ∈ R
read 1st Tuple ∈ S
∀ r do
   { while s(C) < r(B)
        read next s ∈ S;
     if r(B) = s(C) then
        join r and s
        store in Q }
```

Falls die Join-Attribute keine Schlüssel sind, dann kann ein Wert mehr als einmal vorkommen, deshalb können mehrere Durchläufe der inneren Relation notwendig sein.



→ Modifiziere den Algorithmus so, dass er den letzten Wert r(B) und den Punkt in S, an dem der letzte innere Loop begann, speichert. Immer wenn ein doppelter (mehrfacher) r(B)-Wert erkannt wird, wird die Schleife an dem vorherigen Startpunkt in S neu begonnen.

S

A	B
1	20
3	70
4	32
5	42
6	50
7	50

C	D
1	20
7	21
8	21
9	62

$$AB \underset{A=C}{\bowtie} CD$$

A	B	C	D
1	20	1	20
7	50	7	21

Hauptvorteil:

Die Anzahl der Vergleiche zwischen Tupeln wird reduziert.

Performanz:

- Wenn die Relationen bereits sortiert vorliegen, ist der Algorithmus besser als Nested-Loop.
- Jede Relation wird nur einmal gelesen.
- Die Ausführungszeit wird durch **Sortieren** und **Mischen** beschränkt.

Sortieren: $O(n \log n)$

HASHED JOIN METHODE

<u>Idee:</u> Versuche die Tupel der ersten Relation zu isolieren, die zu einem gegebenen Tupel der zweiten Relation unter der Joinbedingung passen.

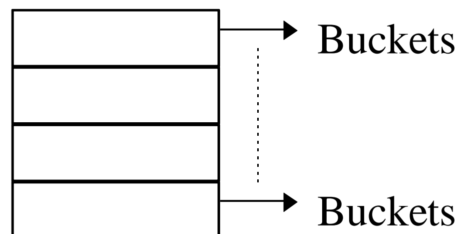
SIMPLE HASH JOIN

Erste Relation:

- Die Werte der Join-Attribute der ersten Relation werden mittels einer Hashfunktion „gehashed“.

- h(v) → Hashtabelle
  Inhalt:

    1. Tupel, oder
    2. TID, Schlüssel



2-te Relation:

- Für jedes Tupel der zweiten Relation werden die Join-Attribute mit **der selben** Hashfunktion gehashed.

- Wenn die Werte auf ein nicht leeres Bucket gehashed werden, dann werden die entsprechenden Tupel miteinander verglichen.

Simple Hash

$$r \underset{r.A=s.B}{\bowtie} s$$

Algorithmus:

$\forall$ u $\in$ s do

{ Hashe über den Join-Attributen u.B;

stelle die Tupel in die Hashtabelle entsprechend

den gehashten Werten; }

$\forall$ t $\in$ r do

{ Hashe über den Join-Attributen t.A;

if t auf ein nicht-leeres Bucket hashed

then

if t.A = u.B then result := result ∪ (t x u) }

A	B
1	20
3	70
4	32
5	42
6	50
7	50

S

C	D
1	D1
7	D2
8	D3
11	D4

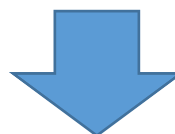
$$AB \underset{A=C}{\bowtie} CD$$

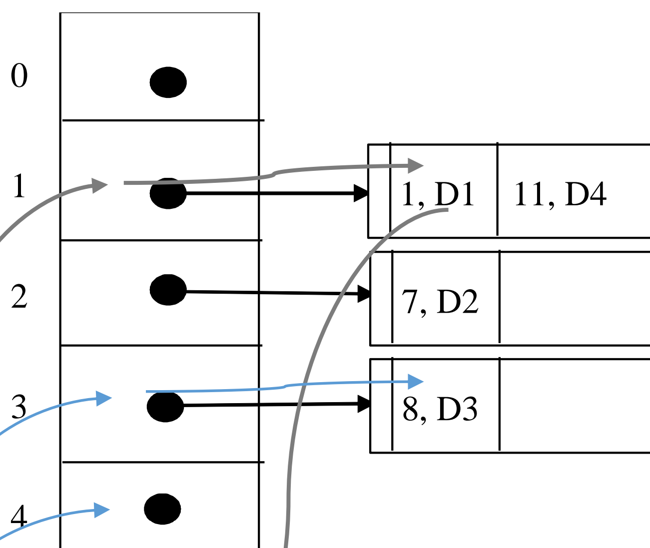
A	B	C	D
1	20	1	D1
7	50	7	D2

h(v)

= v mod 5

C	D
1	D1
7	D2
8	D3
11	D4





h(v) = v mod 5

A	B
1	20
3	70
4	32
5	42
6	50
7	50

A	B	C	D
1	20	1	D1
7	50	7	D2

- Die Hashtabelle wird üblicherweise für die kleinere der beiden Relationen angelegt.

Performanz:

- Hash-basierte Joins sind die effektivsten Join-Techniken.
- Komplexität:

$$O(n+m)$$

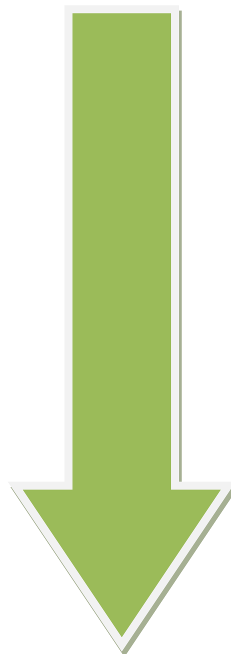
  da beide Relationen nur einmal gelesen werden.
  Die Geschwindigkeit hängt von der Hashfunktion ab.

- Hash-Kollisionen vermindern die Performanz !
- Jedes Tupel, dass auf ein nicht-leeres Bucket hashed, muss überprüft werden, ob es der Join-Bedingung genügt.
- Falls die Join-Bedingung $\theta \neq =$ ist, ist die Implementierung schwierig!
- Die Eliminierung von Duplikaten in der Ergebnismenge ist schwierig.

WAHL DES AUSFÜHRUNGSPLANS

select r.A, s.B, t.D from r, s, t

where r.A = s.A and s.B = t.C;



sei:

r(A,F) ∧

s(A,B,C) ∧

t(C,D,E)

$$\pi_{r.A,s.B,t.D}\left(r \underset{r.A=s.A}{\bowtie} s \underset{s.B=t.C}{\bowtie} t \right)$$

$$\pi_{r.A,s.B,t.D}\left(s \underset{s.B=t.C}{\bowtie} t \underset{r.A=s.A}{\bowtie} r \right)$$

…

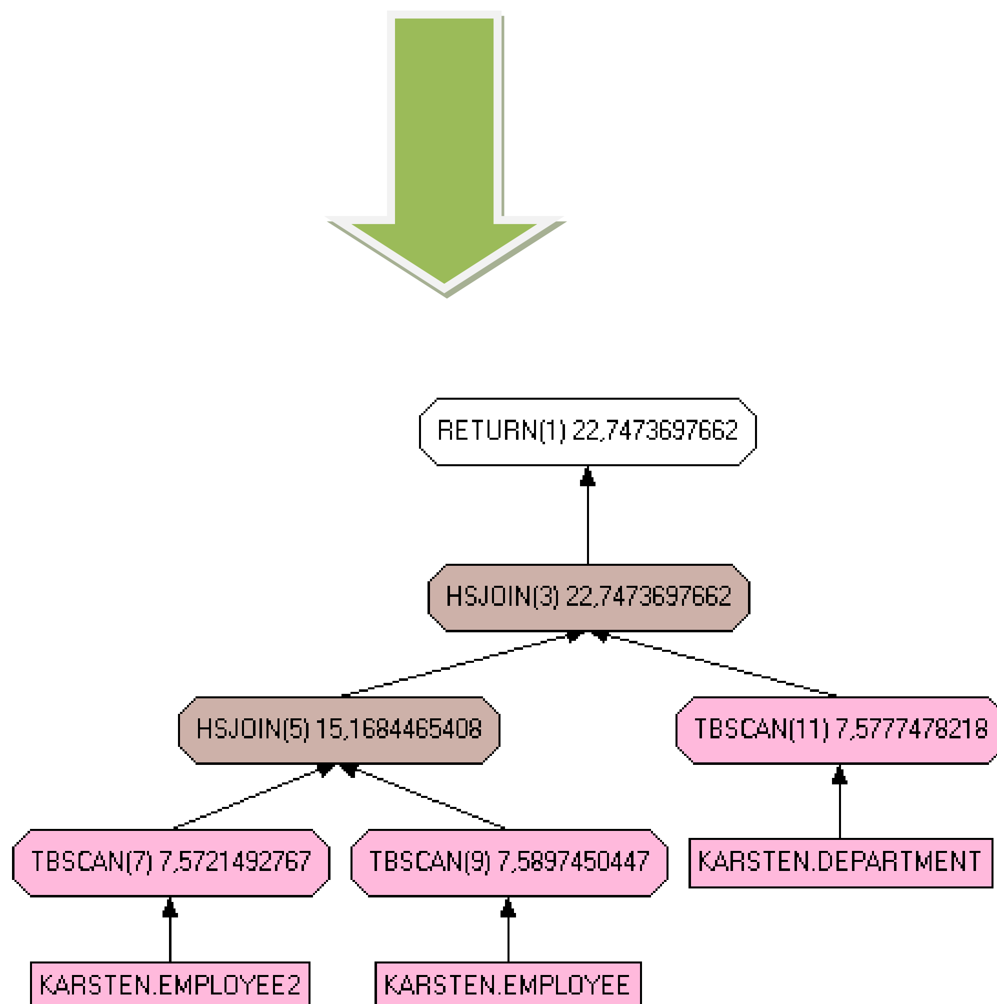
*select * from employee2 e2, employee e, department d*

where

e2.lastname = e.lastname

and

d.deptno = e.workdept;



$$r \bowtie s \bowtie t$$

Die Reihenfolge der Joins ist durch den Optimierer hier wählbar! Möglichst sollte zuerst der Join ausgeführt werden, der das kleinste Ergebnis liefert! Doch …

Wie lässt sich die Größe eines Joins abschätzen?

Selectivity Factor

Der Selectivity Factor (Join Selection Factor) gibt **die Prozent der Tupel** an, die bei einem Join verbunden werden (im Verhältnis zum kartesischen Produktes).

Selectivity Factor $\left(R \bowtie S\right) := \frac{|R \bowtie S|}{|R \times S|}$

klein → Nested Loop schlecht
Hashed Join besser

groß → Nested Loop $\approx$ Hashed Join
oder besser!

Partitioned Hash Join besser

Frage: Wie kann der Selectivity Factor bestimmt werden?

$$r(R) \bowtie s(S)$$

Nutze die Katalog Informationen:

- Falls $R \cap S = \emptyset \Rightarrow$ entspricht dem Kartesischen Produkt
- Falls $R \cap S$ ein Schlüssel von r ist, können die Tupel aus s höchstens mit einem Tupel aus r übereinstimmen. Die Tupelanzahl des Join-Ergebnisses entspricht also höchstens der Anzahl von Tupeln in s.
- Falls $R \cap S$ ein Fremdschlüssel von r ist, so ist die Tupelanzahl des Join-Ergebnisses exakt der Anzahl von Tupeln in s.
- Falls $R \cap S$ weder Schlüssel von r noch von s ist, ist das Abschätzen schwer (aufwendig). Übliche Verfahren sind:
  - parametrisierte Verteilungen,
  - Histogramme und
  - Stichproben.

Stichprobenverfahren

Es wird eine zufällige Menge von Tupeln einer Relation gezogen und deren Verteilung als repräsentativ für die ganze Relation angesehen.

➔ Das Lesen der Tupel erfordert jedoch „teure“ Zugriffe auf den Hintergrundspeicher.

Übersicht

- **Nested-Loops Join** – O($n \times m$)
    - mit Nested-Blocks
    - mit Roching
- **Sort-Merge Join** – O($n \log n$)
- **Hash Join**
    - Simple Hash Join – O($n + m$)
    - Hash-Partitioned Join
        - Grace Hash Join – O($(n + m) / K$)
          | K Speicherblöcken und $2K$ Prozessoren

Table 4-2 Join strategies

Criteria	Nested loop join	Merge scan or Hash join
Only limited memory available	Better	Worse, since memory is needed for sorting or building the hash table.
Need the first row quickly	Better	Worse, since a sort or hash is needed before a row can be returned.
Selecting only a few rows from the inner table	Better	Worse.
Parallelism	Best	Okay.
Index available on inner table	Best	Okay.
No equality predicates	Okay	Requires at least one equality predicate. For hash join, the data type, scale and precision must match perfectly.
Very large numbers of qualifying rows for each input to the join	Worse	Not much better. Hash and sort will almost have equal costs.
One very small input and one very large input to the join	Worse	Hash join better than merge.

DB2 II: Performance Monitoring, Tuning and Capacity Planning Guide

November 2004

www.redbooks.ibm.com/redbooks/pdfs/sg247073.pdf

Literatur:

A. Kemper, A. Eickler: ‘Datenbanksysteme – Eine Einführung ‘,
8. Auflage Oldenburg Verlag, 2011, ISBN 978-3-486-59834-6
(als E-Book mit dem Übungsbuch über die Informatik
Bibliothek herunterladbar!) – Kapitel 8

„Query Evaluation Techniques for Large Databases“

S. Graefe

ACM Computing Survey, Vol. 25, No. 2, Juni 1993

„Join Processing in Relational Databases“

P. Mishra, M. H. Eich

ACM Computing Survey, Vol. 24, No. 1, März 1992