5 Anfragebearbeitung

Übersicht

5.4 Logische Anfrageoptimierung

Äquivalenzregeln der Relationalen Algebra

- Join, Vereinigung, Schnitt und Kreuzprodukt sind kommutativ

$$\begin{aligned} R \bowtie S &= S \bowtie R \\ R \cup S &= S \cup R \\ R \cap S &= S \cap R \\ R \times S &= S \times R \end{aligned}$$

- Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ

$$\begin{aligned} R \bowtie (S \bowtie T) &= (R \bowtie S) \bowtie T \\ R \cup (S \cup T) &= (R \cup S) \cup T \\ R \cap (S \cap T) &= (R \cap S) \cap T \\ R \times (S \times T) &= (R \times S) \times T \end{aligned}$$

- Selektionen sind untereinander vertauschbar

$$\sigma_{Bed1}(\sigma_{Bed2}(R)) = \sigma_{Bed2}(\sigma_{Bed1}(R))$$

5.4 Logische Anfrageoptimierung

Äquivalenzregeln d. Relationalen Algebra (cont.)

- Konjunktionen in einer Selektionsbedingung können in mehrere Selektionen aufgebrochen werden, bzw. nacheinander ausgeführte Selektionen können zu einer konjunktiven Selektion zusammengefasst werden

$$\sigma_{B1 \wedge B2 \wedge \ldots \wedge Bn}(R) = \sigma_{B1}(\sigma_{B2}(\ldots(\sigma_{Bn}(R))\ldots))$$

- Geschachtelte Projektionen können eliminiert werden

$$\pi_{A1}(\pi_{A2}(\ldots(\pi_{An}(R))\ldots)) = \pi_{A1}(R)$$

Damit eine solche Schachtelung sinnvoll ist, muss gelten:

$A1 \subseteq A2 \subseteq \ldots \subseteq An$

- Selektion und Projektion sind vertauschbar, falls die Projektion keine Attribute der Selektionsbedingung entfernt

$$\pi_A(\sigma_B(R)) = \sigma_B(\pi_A(R)) \quad , \text{falls } attr(B) \subseteq A$$

5.4 Logische Anfrageoptimierung

Äquivalenzregeln d. Relationalen Algebra (cont.)

- Selektion und Join (Kreuzprodukt) können vertauscht werden, falls die Selektion nur Attribute eines der beiden Join-Argumente verwendet

$$\sigma_B(R \bowtie S) = \sigma_B(R) \bowtie S$$
$$\sigma_B(R \times S) = \sigma_B(R) \times S$$
$$, \text{falls } attr(B) \subseteq attr(R)$$

- Projektionen können teilweise in den Join verschoben werden

$$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A1}(R) \bowtie_B \pi_{A2}(S))$$
$$, \text{falls } A1 = attr(R) \cap (A \cup attr(B))$$
$$A2 = attr(S) \cap (A \cup attr(B))$$

- Selektionen können mit Vereinigung, Schnitt und Differenz vertauscht werden

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

5.4 Logische Anfrageoptimierung

Äquivalenzregeln d. Relationalen Algebra (cont.)

- Der Projektionsoperator kann mit der Vereinigung, aber nicht mit Schnitt oder Differenz vertauscht werden(Siehe Übung!)

$$\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$$

- Selektion und ein Kreuzprodukt können zu einem Join zusammengefasst werden, wenn die Selektionsbedingung eine Joinbedingung ist (z.B. Equi-Join)

$$\sigma_{R.A1=S.A2}(R \times S) = R \bowtie_{R.A1=S.A2} S$$

- Auch an Bedingungen können Veränderungen vorgenommen werden
  - 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, z.B. $\neg(B_1 \wedge B_2) = \neg B_1 \vee \neg B_2$

5.4 Logische Anfrageoptimierung

Restrukturierungsalgorithmus

- Aufbrechen der Selektionen
- Verschieben der Selektionen so weit wie möglich nach unten im Operatorbaum
- Zusammenfassen von Selektionen und Kreuzprodukten zu Joins
- Einfügen und Verschieben von Projektionen so weit wie möglich nach unten
- Zusammenfassen einzelner Selektionen zu komplexen Selektionen

5.4 Logische Anfrageoptimierung

Beispiel

Fahrzeug-Datenbank

-Kunde(KNr, Name, Adresse, Region, Saldo)

KNr	Name	Adresse	Region	Saldo
201	Klein	Lilienthal	Bremen	200 000
337	Horn	Dieburg	Rhein-Main	100 000
444	Berger	München	München	300 000
108	Weiss	Würzburg	Unterfranken	500 000

-Bestellt(BNr, Datum, KNr, KNr, PNr)

BNr	Datum	KNr	PNr
221	10.05.04	201	12
312	11.05.04	201	4
401	20.05.04	337	330
456	13.05.04	444	330
458	14.05.04	444	98

-Produkt(PNr, Bezeichnung, Anzahl, Preis)

PNr	Bezeichnung	Anzahl	Preis
12	BMW 318i	10	40.000
4	Golf 5	40	25.000
330	Fiat Uno	5	18.000
98	Ferrari 380	1	180.000
14	Opel Corsa	14	17.000

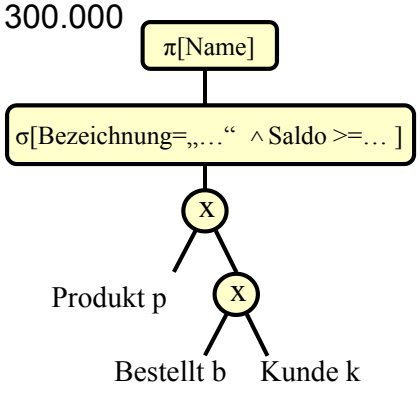
5.4 Logische Anfrageoptimierung

Beispiel (cont.)

- SQL Anfrage:

```
select      Name,
from        Kunde k, Bestellt b, Produkt p
where       b.KNr = k.KNr
and         b.PNr = p.PNr
and         Bezeichnung = „Fiat Uno“
and         Saldo ≥ 300.000
```

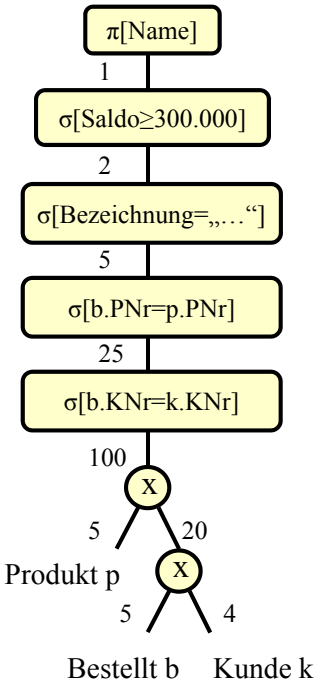
- Kanonischer Auswertungsplan:



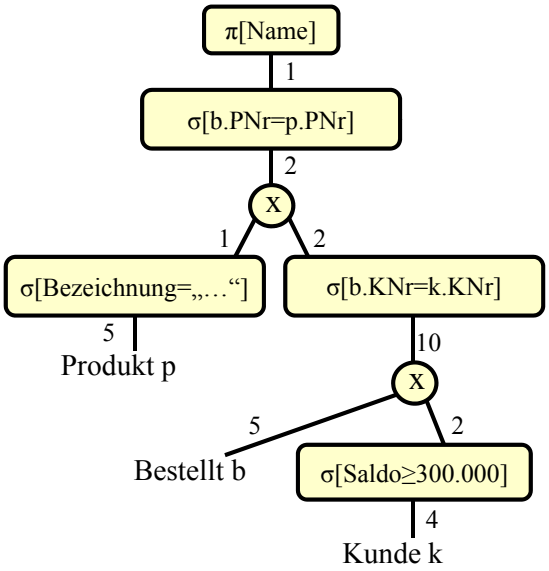
5.4 Logische Anfrageoptimierung

Beispiel (cont.)

- Aufbrechen der Selektionen



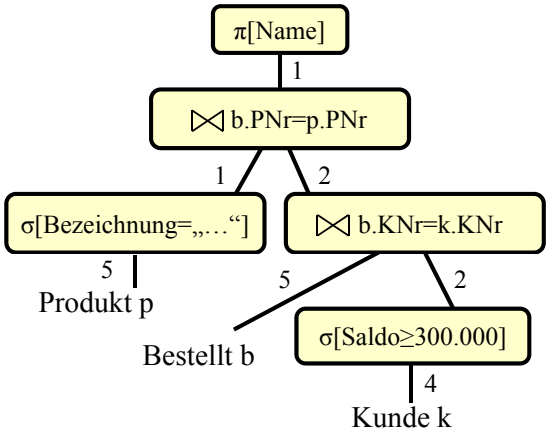
- Verschieben der Selektionen



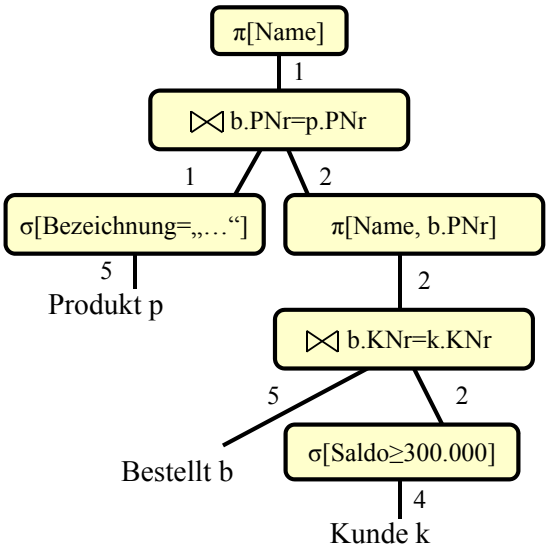
5.4 Logische Anfrageoptimierung

Beispiel (cont.)

- Zusammenfassen zu Joins



- Einfügen zusätzlicher Selektionen



5 Anfragebearbeitung

Übersicht

5.5 Kostenmodellbasierte Anfrageoptimierung

Selektivität

Der Anteil der qualifizierenden Tupel wird *Selektivität sel* genannt.
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 *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)

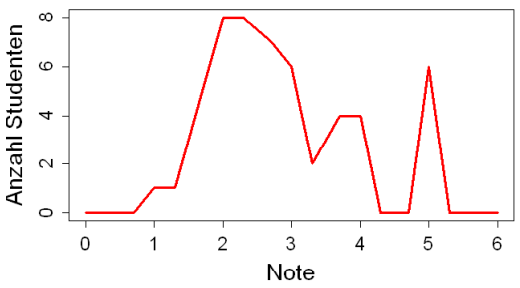
Selektivität

- Die Selektivität muss geschätzt werden, für Spezialfälle gibt es einfache Methoden:
  - Die Selektivität von $\sigma_{R.A=c}$, also Vergleich mit einer Konstante *c* beträgt 1 / |*R*|, falls *A* ein Schlüssel ist
  - Falls *A* kein Schlüssel ist, aber die Werte gleichverteilt sind, ist *sel*=1 / *I* (*I* ist dabei die *image size*, d.h. die Anzahl verschiedener *A*-Werte in *R*)
  - Besitzt bei einem Equi-Join $R \underset{R.A=S.B}{} S$ das Attribut *A* Schlüsseleigenschaft, kann die Größe des Join-Ergebnisses mit |*S*| abgeschätzt werden, da jedes Tupel aus *S* maximal einen Joinpartner findet. Die Selektivität ist also $sel_{RS} = 1/|R|$
  - logisches UND: $sel_B(\sigma_{B1 \wedge B2}) = sel_B(\sigma_{B1}) \cdot sel_B(\sigma_{B2})$
  - logisches ODER: $sel_B(\sigma_{B1 \vee B2}) = sel_B(\sigma_{B1}) + sel_B(\sigma_{B2}) - sel_B(\sigma_{B1}) \cdot sel_B(\sigma_{B2})$
  - logisches NICHT: $sel_B(\sigma_{\neg B1}) = 1 - sel_B(\sigma_{B1})$

Selektivität

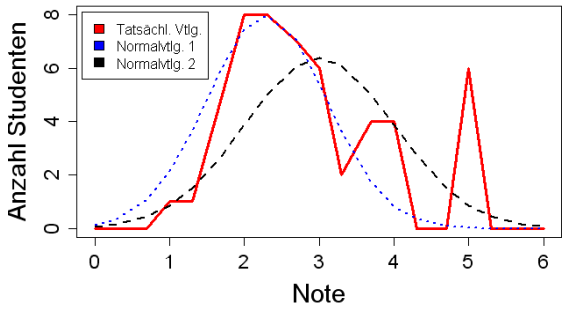
- Im Allgemeinen benötigt man anspruchsvollere Methoden um zu schätzen, wieviele Tupel sich in einem bestimmten Wertebereich befinden.
- Drei Grundsätzliche Arten von Schätzmethoden:
  - Parametrische Verteilungen
  - Histogramme
  - Stichproben

Beispiel: Schätzung der Verteilung der Noten der DBS II Klausur anhand des Ergebnisse von 2007:



Selektivität: Parametrische Verteilungen

- Bestimme zu der vorhandenen Werteverteilung die Parameter einer Funktion so, dass die Verteilung möglichst gut angenähert wird.

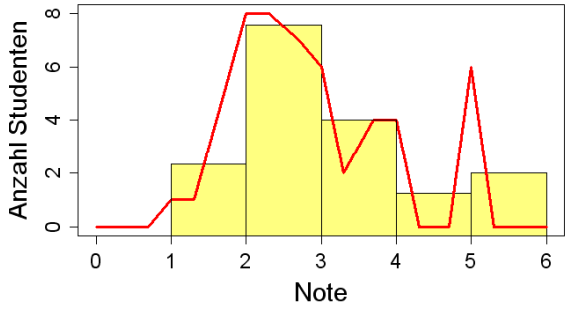


Probleme:

- 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
  - *Equi-Depth-Histograms:* Unterteilung so, dass in jedem Intervall gleich viele Tupel sind



- Flexible Annäherung an die Verteilung

Selektivität: Stichproben

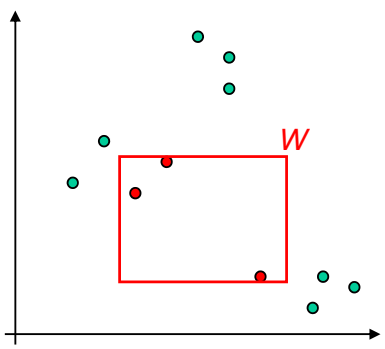
- Sehr einfaches Verfahren
- Ziehe eine zufällige Menge von n Tupeln aus einer Relation, und betrachte deren Verteilung als repräsentativ für die gesamte Relation.
- Problem der Größe des Stichprobenumfangs n:

  - n zu klein: Wenig repräsentative Stichprobe
  - n zu gross: Ziehen der Stichprobe erfordert zu viele „teure“ Zugriffe auf den Hintergrundspeicher

5.5 Kostenmodellbasierte Anfrageoptimierung

Beispiel: Selektivität von Fensteranfragen

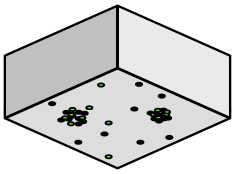
- Szenario:
  - Datenobjekte (Tupel) sind Punkte in einem d-dimensionalen Featureraum (z.B. geographische Objekte auf einer 2D Karte)
  - Fensteranfragen:
    - Gegeben ein Anfragefenster *W* (d-dimensionales Hyper-Rechteck)
    - Gesucht: alle Objekte innerhalb des Anfragefensters *W*



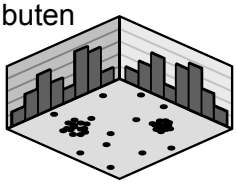
5.5 Kostenmodellbasierte Anfrageoptimierung

Beispiel: Selektivität von Fensteranfragen (cont)

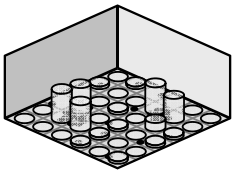
- Bekannte Ansätze:
  - Sampling
    - ***Problem:*** Genauigkeit abhängig von der Samplegröße



  - 1D Histogramme
    - ***Problem:*** Annahme der Unabhängigkeit zwischen den Attributen



  - Mutli-D Histogramme
    - ***Problem:*** Anzahl der Gridzellen steigt exponentiell mit *d*

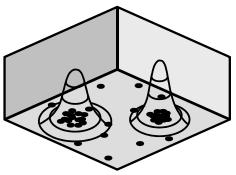


  - Parametrische Methoden
    - ***Problem:*** nur für 2D und 3D Daten geeignet

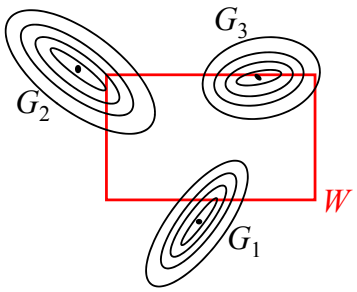
5.5 Kostenmodellbasierte Anfrageoptimierung

Beispiel: Selektivität von Fensteranfragen (cont)

- Stattdessen:
  - Modellierung der Datenverteilung durch eine Menge von Gauss-Verteilungen
  - EM-Algorithmus:
    - Input: Datenbank, Anzahl der Gaussverteilungen *k*
    - Output: *k* Gaussverteilungen, die die Objekte der Datenbank optimal repräsentieren



- Selektivitätsabschätzung:
  - Integral des Schnitts von *W* mit allen *k* Gauss-Verteilungen



5 Anfragebearbeitung

Übersicht

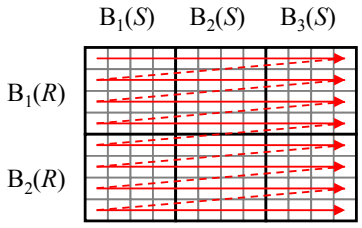
5.6 Implementierung der Joinoperation

Einfacher Nested-Loop-Join

- Algorithmus

```
for each Tupel r ∈ R do
  for each Tupel s ∈ S do
    if r.A = s.B then
      result := result ∪ (r × s)
```

- Matrixnotation



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

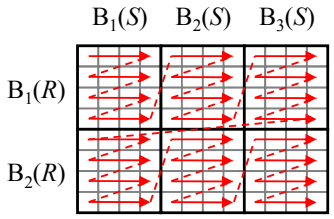
5.6 Implementierung der Joinoperation

Nested-Block-Loop-Join

- Algorithmus

```
for each Block B_R ∈ R do
  lade Block B_R
  for each Block B_S ∈ S do
    lade Block B_S
    for each Tupel r ∈ B_R do
      for each Tupel s ∈ B_S do
        if r.A = s.B then
          result := result ∪ (r × s)
```

- Matrixnotation



5.6 Implementierung der Joinoperation

Nested-Block-Loop-Join (cont.)

- Beispiel:

S

Angestellter	Gehaltsgruppe	
Müller	1	$B_S(1)$
Schneider	2	
Schuster	1	$B_S(3)$
Schmidt	2	
Schütz	1	$B_S(3)$

R

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

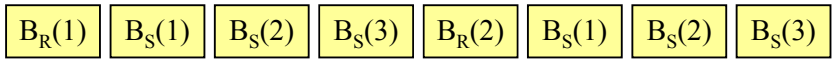
- Anzahl Blockzugriffe: $B_R + B_S \cdot B_R = 8$ Blockzugriffe ohne Cache
  (B_R = Anzahl Blöcke der Relation R)

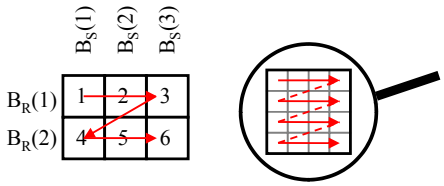
- D.h. die kleinere Relation sollte die äußere sein

Cache Strategien für Nested-Block-Loop-Join

1. Seiten der inneren Relation 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)

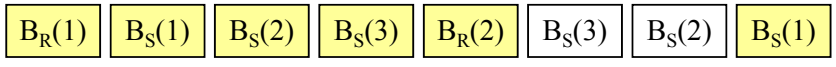


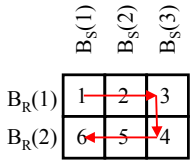


Cache Strategien für NBL-Join (cont.)

2. Seiten der inneren Relation 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 *R*-Relation benötigt
- Blockzugriffe: $B_R + B_R \cdot (B_S - |C| + 1) + |C| - 1$
- Beispiel: 2 Seiten Cache für *S*, 1 Seite Cache für *R*





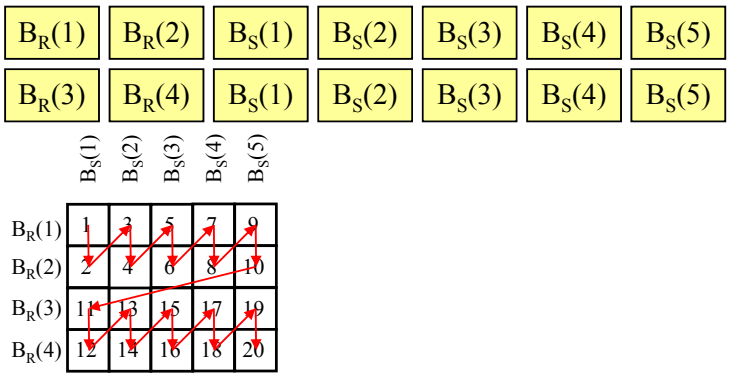
5.6 Implementierung der Joinoperation

Cache Strategien für NBL-Join (cont.)

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



5.6 Implementierung der Joinoperation

Cache Strategien für NBL-Join (cont.)

– Algorithmus für Strategie 3:

```
for i := 0 to B_R step |C| do
  lade Block B_R(i) ... B_R(i+|C|-1)
  for each Block B_S ∈ S do
    lade Block B_S
    for each Tupel r ∈ B_R(i) ... B_R(i+|C|-1) do
      for each Tupel s ∈ B_S do
        if r.A = s.B then
          result := result ∪ (r × s)
```

– Leistung:
  – |R|*|S| Vergleiche von Tupel (ist nur bei schlechter Selektivität gerechtfertigt)
  – Effizienteste Ausführung von θ–Joins mit θ ≠ '='

Blockgrößen-Optimierung NBL-Join

Problem:

- Zu kleine Blockgröße:
  - Innere Relation wird in sehr kleinen Schritten eingelesen
  - Bei jedem I/O-Auftrag Latenzzeit des Plattenlaufwerks
- Zu große Blockgröße (z.B.: Cache wird in 2-3 Blöcke geteilt):
  - Zu wenig Cache steht für die äußere Relation zur Verfügung
  - Innere Relation muss öfter gescanned werden

Äquivalente Frage:

Wie viel vom Cache für äußere/innere Relation?

Blockgrößen-Optimierung NBL-Join (cont.)

I/O-Kosten für den gesamten Join:

$$t_{NL-Join} \approx \left\lceil \frac{B_R}{|C|-1} \right\rceil \cdot (2t_{seek} + t_{lat} + b \cdot (|C|-1) \cdot t_{tr}) + B_S \cdot \left\lceil \frac{B_R}{|C|-1} \right\rceil \cdot (t_{lat} + b \cdot t_{tr})$$

- f_R bzw. f_S: Größe der Relationen in Bytes
- c: Größe des Cache in Bytes
- t_{tr}: Transferzeit pro Byte
- t_{lat}: durchschnittliche Latenzzeit des Disk-Laufwerkes
- b: Blockgröße (Parameter, der optimiert wird)

- Vernachlässigung des B_R-Scans (da nur 1 mal und in großen Blöcken)

$$t_{NL-Join} \approx \left(\left\lceil \frac{f_s}{b} \right\rceil \cdot \left\lceil \frac{\lceil f_R / b \rceil}{\lfloor c/b \rfloor - 1} \right\rceil \right) \cdot (t_{lat} + b \cdot t_{tr})$$

5.6 Implementierung der Joinoperation

Blockgrößen-Optimierung NBL-Join (cont.)

	Äußere Relation R	**Innere Relation S**
Anzahl Block-zugriffe	B_R	$B_R + B_S \cdot \left\lceil \frac{B_R}{\lvert C \rvert - 1} \right\rceil$
	Suchen zum aktuellen Block von R + Suchen zum Start von S	
$t_{NL-Join} \approx$	$\left\lceil \frac{B_R}{\lvert C \rvert - 1} \right\rceil \cdot (2t_{seek} + t_{lat} + b \cdot (\lvert C \rvert - 1) \cdot t_{tr})$	$+ \quad B_S \cdot \left\lceil \frac{B_R}{\lvert C \rvert - 1} \right\rceil \cdot (t_{lat} + b \cdot t_{tr})$
	in einer Leseoperation werden |C|-1 Blöcke der äußeren Relation gelesen	Jeweils ein Block wird gelesen, aber nächster Block startet meist auf gleicher Spur
$t_{NL-Join} \approx$	*ignorieren, da nur 1x und in großen Blöcken*	$\left(\left\lceil \frac{f_s}{b} \right\rceil \cdot \left\lceil \frac{\lceil f_R / b \rceil}{\lfloor c / b \rfloor - 1} \right\rceil \right) \cdot (t_{lat} + b \cdot t_{tr})$

f_R bzw. f_S: Größe der Relationen in Bytes
c: Größe des Cache in Bytes
t_{tr}: Transferzeit pro Byte

5.6 Implementierung der Joinoperation

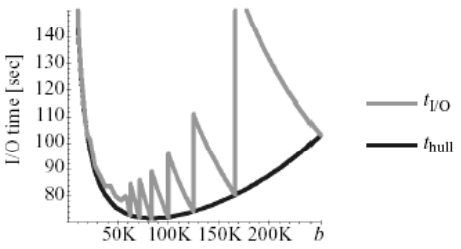
Blockgrößen-Optimierung NBL-Join (cont.)

– Weglassen der Rundungsfunktion (unproblematisch für f_R, $f_S >> b$, d.h. relativer Fehler ist vernachlässigbar) ergibt stückweise differenzierbaren Term

$$t_{NL-Join} \approx \left(\frac{f_S \cdot f_R}{b^2 \cdot (\lfloor c / b \rfloor - 1)} \right) \cdot (t_{lat} + b \cdot t_{tr})$$

– Optimierung der Hüllfunktion

$$t_{hull} \approx \left(\frac{f_S \cdot f_R}{b^2 \cdot ((c / b) - 1)} \right) \cdot (t_{lat} + b \cdot t_{tr})$$



Joinkosten bei
- $f_R = f_S$ = 10MByte
- c = 500 KByte
- t_{lat} = 5 ms
- t_{tr} = 0,25 s /MByte
- b_{opt} = 85 KByte

Blockgrößen-Optimierung NBL-Join (cont.)

- **Optimierung durch Differenzieren**
  - Gleichsetzen der 1. Ableitung mit 0
  - 2 Lösungen, von denen nur eine positiv ist

$$0 = \frac{\partial}{\partial b} t_{hull} \Rightarrow b_{opt} = \frac{\sqrt{t_{lat}^2 + t_{tr} \cdot t_{lat} \cdot c} - t_{lat}}{t_{tr}}$$

  - Lösung ist Minimum (s. 2. Ableitung)
  - An den Stellen, an denen $\lfloor c / b \rfloor$ konstant ist, ist t_{NLJoin} streng monoton fallend (negative Ableitung)
  - Deshalb kann das Minimum von t_{NLJoin} nur an der ersten Sprungstelle links oder rechts vom Minimum von t_{hull} sein:

$$b_1 = c / \left\lfloor \frac{c}{b_{opt}} \right\rfloor, \quad b_2 = c / \left\lceil \frac{c}{b_{opt}} \right\rceil$$

Blockgrößen-Optimierung NBL-Join (cont.)

CPU-Kosten

- Im wesentlichen müssen $|S|*|R|$ Vergleiche durchgeführt werden
- Bei 0.1 µs pro Vergleich und 100.000 Tupel pro Relation ergibt sich eine Bearbeitungszeit von 1000 s.
- D.h. wesentlich mehr als die 75 s I/O-Zeit
- Der NLB-Join ist also *CPU-bound*
- Maßnahmen zur Senkung des CPU-Aufwands später

5.6 Implementierung der Joinoperation

Sort-Merge-Join

– **Zweistufiger Algorithmus**

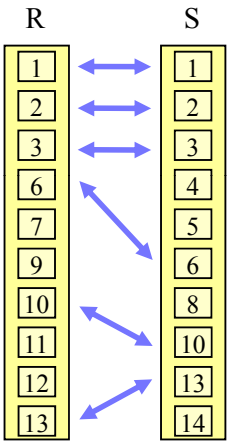
– 1.Schritt:

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

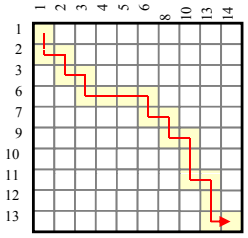
– 2.Schritt:

```
j = 1;
s = erstes Tupel von S;
for i = 1 to |R| do
  r = i - tes Tupel von R;
  while s.B < r.A
    j = j + 1;
    s = j - tes Tupel von S;
  if r.A = s.B then
    result := result ∪ ((r − r.A) × s);
```

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



- **Matrixnotation**



5.6 Implementierung der Joinoperation

Sort-Merge-Join (cont.)

Leistung

- Jede Relation wird genau einmal durchlaufen: O(|R| + |S|) Vergleiche
- Sortieren der Relation kostet O(|R| · log |R| + |S| · log |S|)
- Sortieren ist nicht notwendig, wenn bereits ein Index existiert
- Verfahren versagt, wenn in beiden Relationen sehr viele Duplikate (d.h. mehr als in den Puffer passen) auftreten. In diesem Fall muss auf Nested-Loop-Join umgeschaltet werden

5.6 Implementierung der Joinoperation

Einfacher Hash-Join

Reduktion des CPU-Aufwandes bei der Join-Berechnung

- Der Join-Partner eines *S*-Tupels wird gezielt mit Hilfe eines Hash-Verfahrens gesucht, anstatt das *S*-Tupel sequentiell mit jedem Tupel der Relation *R* zu vergleichen.
- Zu diesem Zweck wird die Relation *R* gehasht, d.h. es wird zu allen Tupeln der Hash-Key bestimmt und die Tupel in einer Tabelle unter diesem Key eingetragen.
- Nicht alle *R*-Tupel, die den passenden Hash-Key haben, sind Join-Partner eines *S*-Tupels, aber alle Join-Partner haben denselben Hash-Key.
- 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 Join effizient genutzt werden.

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*)

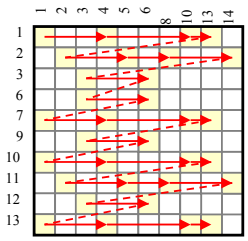
5.6 Implementierung der Joinoperation

Einfacher Hash-Join (cont.)

- **Algorithmus**

for each Tupel $r \in R$ do
  berechne $adr = hash(r)$;
  speichere r in $HT[adr]$ ab;
for each Tupel $s \in S$ do //prüfe in der Hashtabelle HT
  berechne $adr = hash(s)$;
  for each Tupel $r \in HT[adr]$ do
    if $r.A = s.B$ then
      $result := result \cup ((r - r.A) \times s)$

- **Matrixnotation**



$hash(x)$ = MOD 3

5.6 Implementierung der Joinoperation

Hashed-Loop-Join

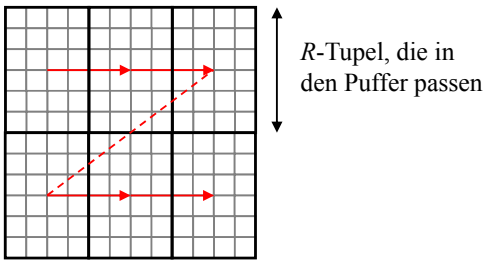
- Kombination aus dem Nested-Loop-Join und dem einfachen Hash-Join
- Relation R wird in große Blöcke eingeteilt, deren Hashtabellen in den Puffer passen
- Für jeden dieser Blöcke wird die Relation S gescannt und ein einfacher Hash-Join durchgeführt
- **Algorithmus**

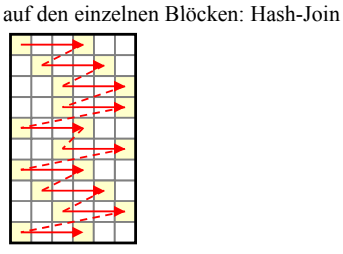
```
repeat
    lese soviel Tupel von R in Hauptspeicher bis der Platz aufgebraucht ist;
    erzeuge für diese Tupel eine Hashtabelle HT;
    for each Tupel s ∈ S do
        berechne adr = hash(s);
        for each Tupel r ∈ HT[adr] do
            if r.A = s.B then
                result := result ∪ ((r − r.A) × s)
until alle Tupel der Relation R sind eingelesen;
```


5.6 Implementierung der Joinoperation

Hashed-Loop-Join (cont.)

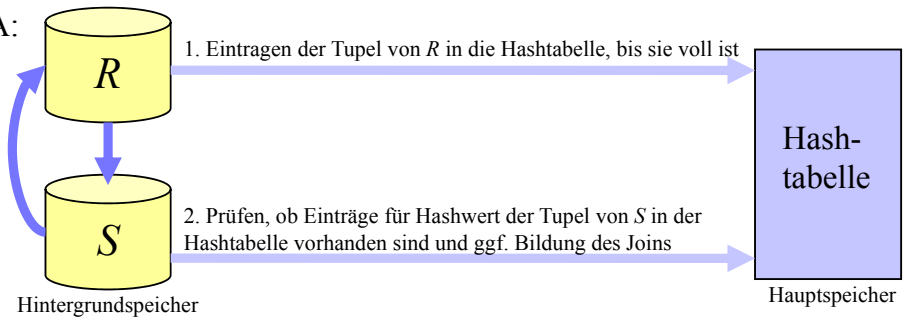
- **Matrixnotation**





- **Ablauf**

Schritt A:



Schritt B: Wiederhole Schritt A für die restlichen Tupel von R

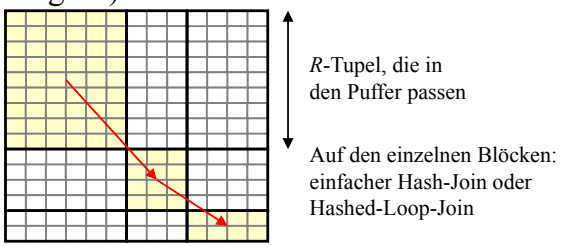
5.6 Implementierung der Joinoperation

Hash-Partitioned-Join (GRACE)

- Der Hashed-Loop-Join zerlegt die Relationen willkürlich in Blöcke, jeder Block der *R*-Relation muss mit jedem Block der *S*-Relation kombiniert werden
- Idee: Zerlege die Relationen *R* und *S* mit Hilfe einer Hashfunktion in Partitionen, so dass nur Partitionen mit demselben Hash-Key kombiniert werden müssen
- Zweistufiges Verfahren

1. Partitioniere die Relationen *R* und *S* in $R_1,\ldots,R_N$ und $S_1,\ldots,S_N$
2. Berechne den Join der einzelnen Partitionen R_i und S_i mit einem beliebigen Join Verfahren (z.B. einfacher Hash-Join oder Hashed-Loop-Join wenn Partition zu groß)

Matrixnotation

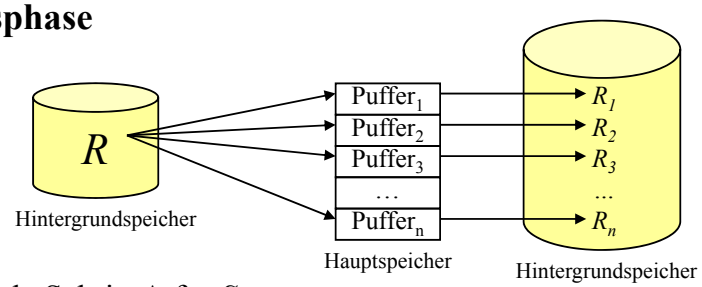


5.6 Implementierung der Joinoperation

Hash-Partitioned-Join (GRACE) (cont.)

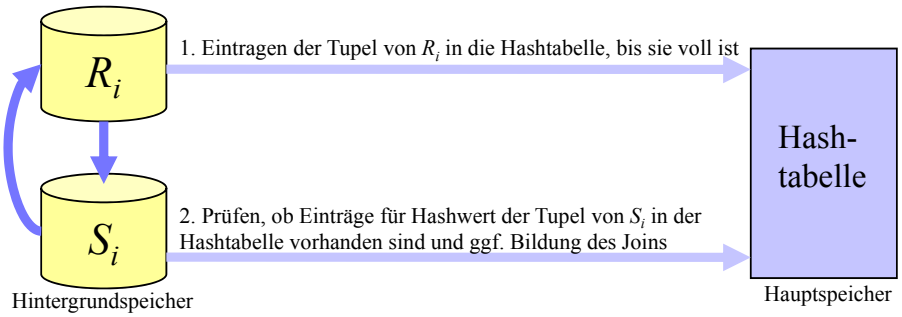
- **Ablauf**
  - **Partitionierungsphase**

Schritt A:



Schritt B: Wiederhole Schritt A für *S*

  - **Join-Phase**



5.6 Implementierung der Joinoperation

Hybrid Hash-Join

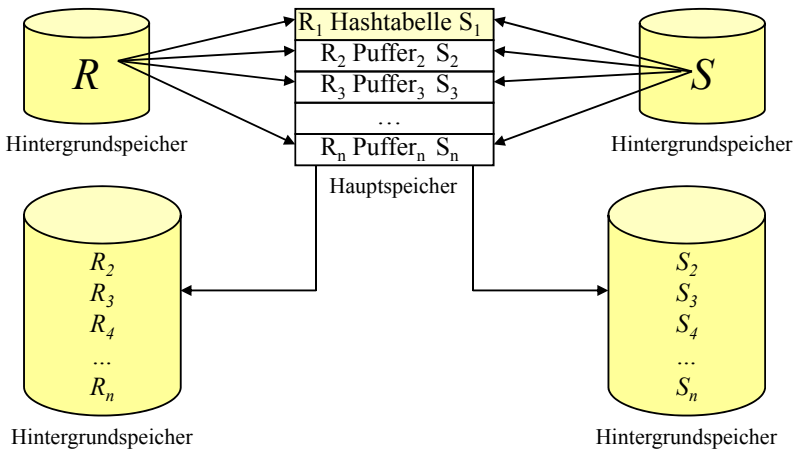
- **Algorithmus**

```
for each Tupel r ∈ R do
    berechne adr = hash(r);
    if (adr = 1) then
        füge r in eine Hashtabelle HT ein (bzgl. neuer Hashfkt.);
    else
        speichere r in einem Puffer BR_adr
        /* wenn der Puffer voll ist, wird er stets auf Platte geschrieben */
for each Tupel s ∈ S do
    berechne adr = hash(s);
    if (adr = 1) then
        suche in HT nach entsprechenden Tupel r mit r.A = s.B;
    else
        speichere s in einem Puffer BS_adr
for i = 2 to N do
    berechne den Join der Partitionen R_i und S_i mit dem Hashed - Loop - Join
```


5.6 Implementierung der Joinoperation

Hybrid Hash-Join (cont.)

Ablauf der Partitionierungsphase:



Hybrid Hash-Join (cont.)

- **Leistung**
  - Reduzierung der I/O-Kosten (im Vergleich zu GRACE), da eine Partition im Hauptspeicher gehalten wird
  - vorteilhaft, wenn viel Hauptspeicher zur Verfügung steht, aber die Relation *R* nicht komplett im Hauptspeicher gehalten werden kann
- **Probleme aller Hash-Join-Verfahren**
  - ungleiche Datenverteilung (extrem hohe Belegung eines Wertes durch Datensätze)
  - Wie wird die Hashfunktion (und damit die Partitionen) der einzelnen Verfahren gewählt?