

5 Anfragebearbeitung

Übersicht

5.1 Einleitung

5.2 Indexstrukturen

5.3 Grundlagen der Anfrageoptimierung

5.4 Logische Anfrageoptimierung

5.5 Kostenmodellbasierte Anfrageoptimierung

5.6 Implementierung der Joinoperation

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)

5.5 Kostenmodellbasierte Anfrageoptimierung

Selektivität

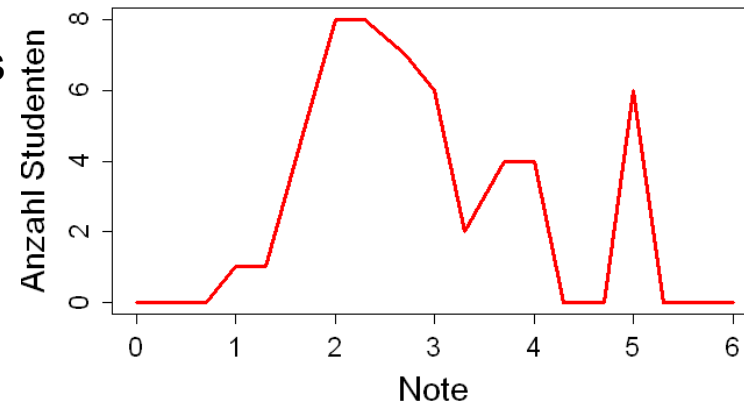
- Die Selektivität muss geschätzt werden, für Spezialfälle gibt es einfache Methoden:
 - Die Selektivität von $\sigma_{R.A=c}$, also Vergleich mit einer Konstante c beträgt $1 / |R|$, falls A ein Schlüssel ist
 - Falls A kein Schlüssel ist, aber die Werte gleichverteilt sind, ist $sel=1 / I$ (I ist dabei die *image size*, d.h. die Anzahl verschiedener A -Werte in R)
 - Besitzt bei einem Equi-Join $R \bowtie_{R.A=S.B} S$ 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: $sel_B(\sigma_{\neg B1}) = 1 - sel_B(\sigma_{B1})$

5.5 Kostenmodellbasierte Anfrageoptimierung

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

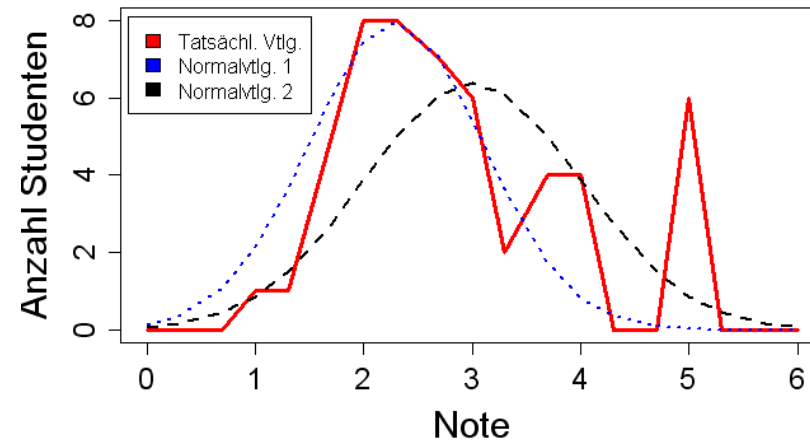
Beispiel: Schätzung der Verteilung der Noten der DBS II Klausur anhand des Ergebnisse von 2007:



5.5 Kostenmodellbasierte Anfrageoptimierung

Selektivität: Parametrische Verteilungen

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



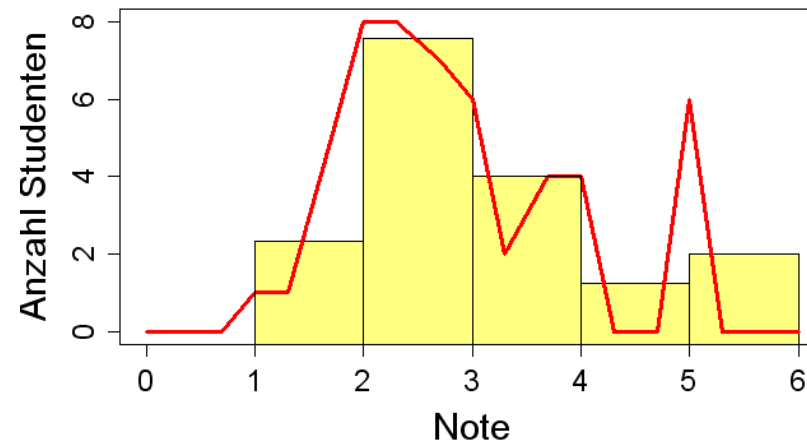
Probleme:

Wahl des Verteilungstyps (Normalverteilung, Exponentialverteilung...)
und Wahl der Parameter, besonders bei mehrdimensionalen Anfragen
(also z.B. bei Selektionen, die sich auf mehrere Attribute beziehen)

5.5 Kostenmodellbasierte Anfrageoptimierung

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

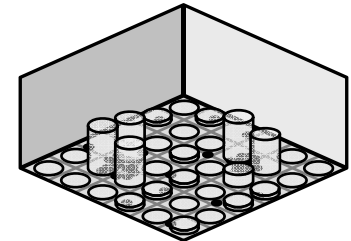
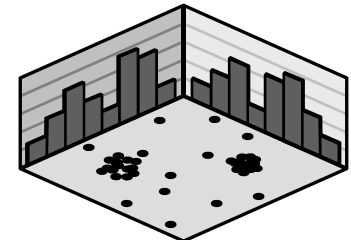
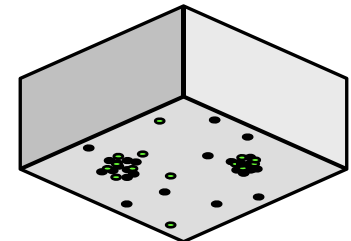
5.5 Kostenmodellbasierte Anfrageoptimierung

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

- Probleme bei Anfragen über mehrere Attribute (mehr-dimensionale Anfragen)
 - 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 max. 2-3 Attribute geeignet



5 Anfragebearbeitung

Übersicht

5.1 Einleitung

5.2 Indexstrukturen

5.3 Grundlagen der Anfrageoptimierung

5.4 Logische Anfrageoptimierung

5.5 Kostenmodellbasierte Anfrageoptimierung

5.6 Implementierung der Joinoperation

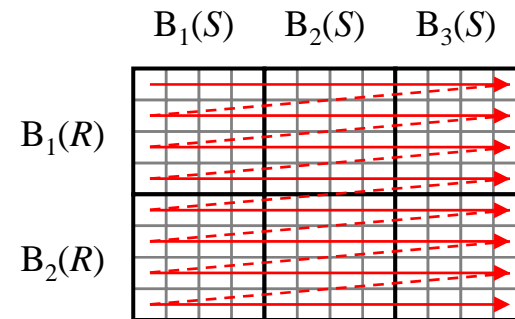
5.6 Implementierung der Joinoperation

Einfacher Nested-Loop-Join

– Algorithmus

```
for each Tupel  $r \in R$  do
  for each Tupel  $s \in S$  do
    if  $r.A = s.B$  then
       $result := result \cup (r \times s)$ 
```

– 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

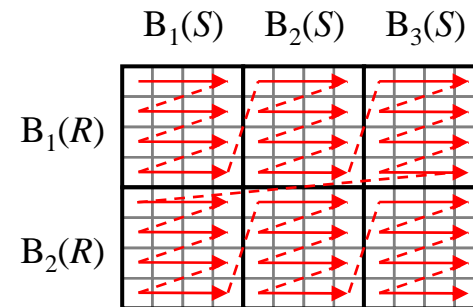
5.6 Implementierung der Joinoperation

Nested-Block-Loop-Join

– Algorithmus

```
for each Block  $B_R \in R$  do
  lade Block  $B_R$ 
  for each Block  $B_S \in S$  do
    lade Block  $B_S$ 
    for each Tupel  $r \in B_R$  do
      for each Tupel  $s \in B_S$  do
        if  $r.A = s.B$  then
           $result := result \cup (r \times s)$ 
```

– Matrixnotation



5.6 Implementierung der Joinoperation

Nested-Block-Loop-Join (cont.)

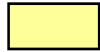
- Beispiel:

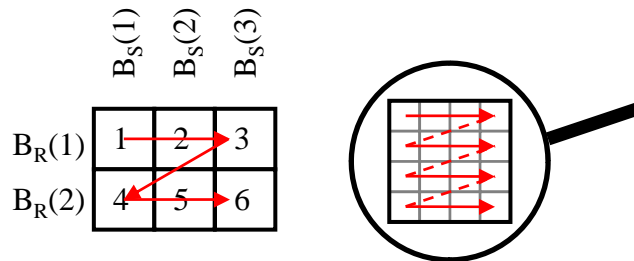
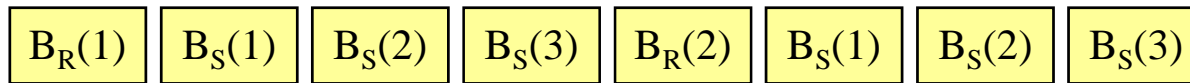
S	Angestellter	Gehaltsgruppe		R	Gehaltsgruppe	Gehalt	
	Müller	1	B _S (1)		1	10.000	B _R (1)
	Schneider	2			2	20.000	
	Schuster	1	B _S (3)		3	30.000	B _R (2)
	Schmidt	2					
	Schütz	1	B _S (3)				

- 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

5.6 Implementierung der Joinoperation

Cache Strategien für Nested-Block-Loop-Join

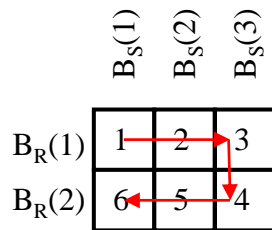
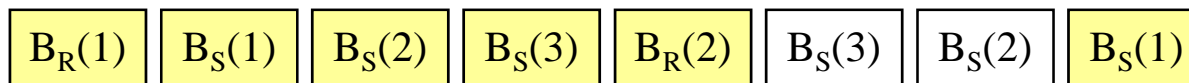
1. Seiten der inneren Relation im Cache halten
 - Cache wird überhaupt nicht ausgenutzt, wenn Cache kleiner als Relation S ist
 - Beispiel: 2 Seiten Cache für S , 1 Seite Cache für R
( Zugriff Platte)



5.6 Implementierung der Joinoperation

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 $(\lceil C \rceil - 1)$ Blockzugriffe eingespart (ab 2. Durchlauf)
 - $\lceil C \rceil =$ Anzahl Blöcke, die in den Cache passen, ein Cache-Block wird jeweils für R -Relation benötigt
 - Blockzugriffe: $B_R + B_R \cdot (B_S - \lceil C \rceil + 1) + \lceil C \rceil - 1$
 - Beispiel: 2 Seiten Cache für S , 1 Seite Cache für R



5.6 Implementierung der Joinoperation

Cache Strategien für NBL-Join (cont.)

- Algorithmus für Strategie 3:

```
for  $i := 1$  to  $B_R$  step  $|C| - 1$  do
  lade Block  $B_R(i) \dots B_R(i + |C| - 2)$ 
  for each Block  $B_S \in S$  do
    lade Block  $B_S$ 
    for each Tupel  $r \in B_R(i) \dots B_R(i + |C| - 2)$  do
      for each Tupel  $s \in B_S$  do
        if  $r.A = s.B$  then
           $result := result \cup (r \times s)$ 
```

- Leistung:
 - $|R|^*|S|$ Vergleiche von Tupel (ist nur bei schlechter Selektivität gerechtfertigt)
 - Effizienteste Ausführung von θ -Joins mit $\theta \neq '='$

5.6 Implementierung der Joinoperation

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?

5.6 Implementierung der Joinoperation

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

f_R bzw. f_S : Größe der Relationen in Bytes

c : Größe des Cache in Bytes

t_{tr} : Transferzeit pro Byte

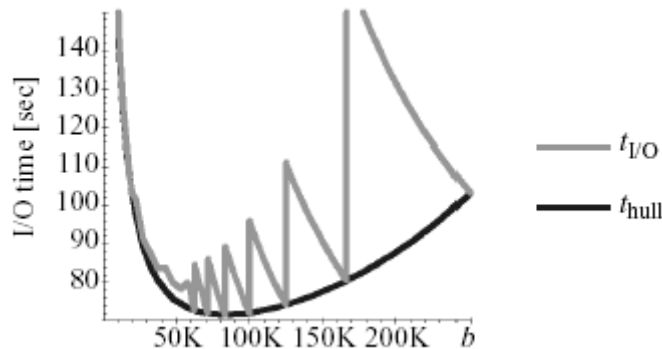
5.6 Implementierung der Joinoperation

Blockgrößen-Optimierung NBL-Join (cont.)

- Weglassen der Rundungsfunktion (unproblematisch für $f_R, f_S \gg b$, 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

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

5.6 Implementierung der Joinoperation

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:

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

5.6 Implementierung der Joinoperation

Blockgrößen-Optimierung NBL-Join (cont.)

CPU-Kosten

- Im wesentlichen müssen $|S|*|R|$ Vergleiche durchgeführt werden
- Bei $0.1 \mu\text{s}$ 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

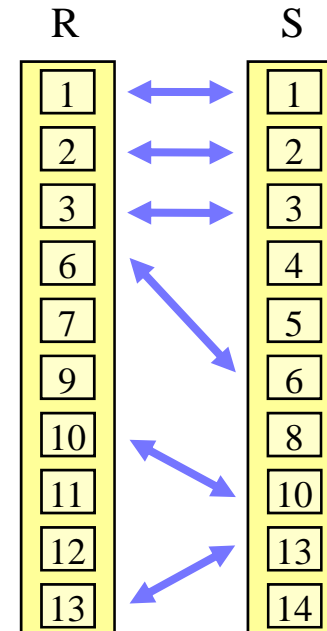
– 1.Schritt:

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

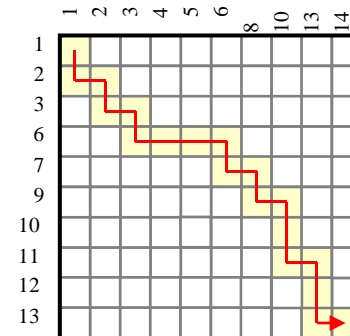
– 2.Schritt:

```
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 $O(|R| \cdot \log |R| + |S| \cdot \log |S|)$
- 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 ghasht, 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)

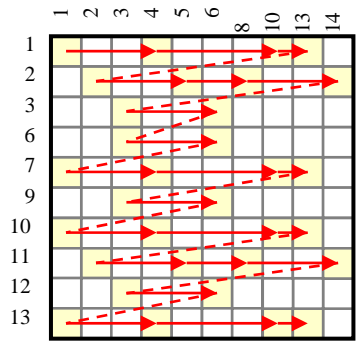
5.6 Implementierung der Joinoperation

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**



$hash(x) = \text{MOD } 3$

5.6 Implementierung der Joinoperation

Hashed-Loop-Join

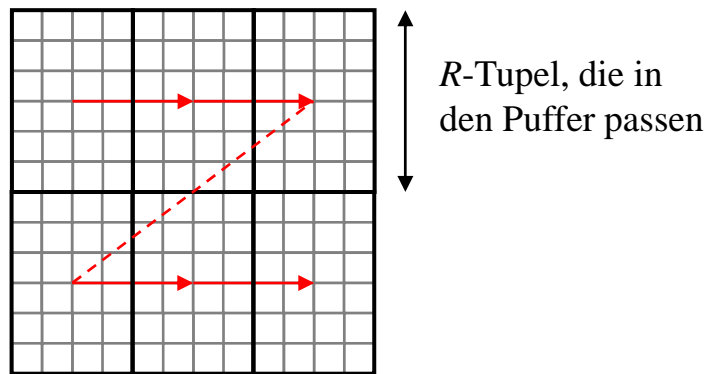
- Kombination aus dem Nested-Block- Loop-Join und dem einfachen Hash-Join
- Relation R wird in große Blöcke eingeteilt, deren Hashtabellen in den Puffer passen
- Für jeden dieser Blöcke wird die Relation S gescannt und ein einfacher Hash-Join durchgeführt
- **Algorithmus**

```
repeat
  lese soviel Tupel von  $R$  in Hauptspeic her bis der Platz aufgebrauc ht ist;
  erzeuge für diese Tupel eine Hashtabell e  $HT$ ;
  for each Tupel  $s \in S$  do
    berechne  $adr = hash(s)$ ;
    for each Tupel  $r \in HT[adr]$  do
      if  $r.A = s.B$  then
         $result := result \cup ((r - r.A) \times s)$ 
until alle Tupel der Relation  $R$  sind eingelesen ;
```

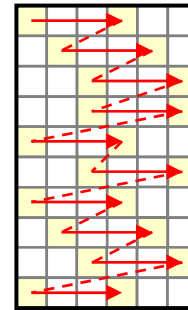
5.6 Implementierung der Joinoperation

Hashed-Loop-Join (cont.)

- Matrixnotation**

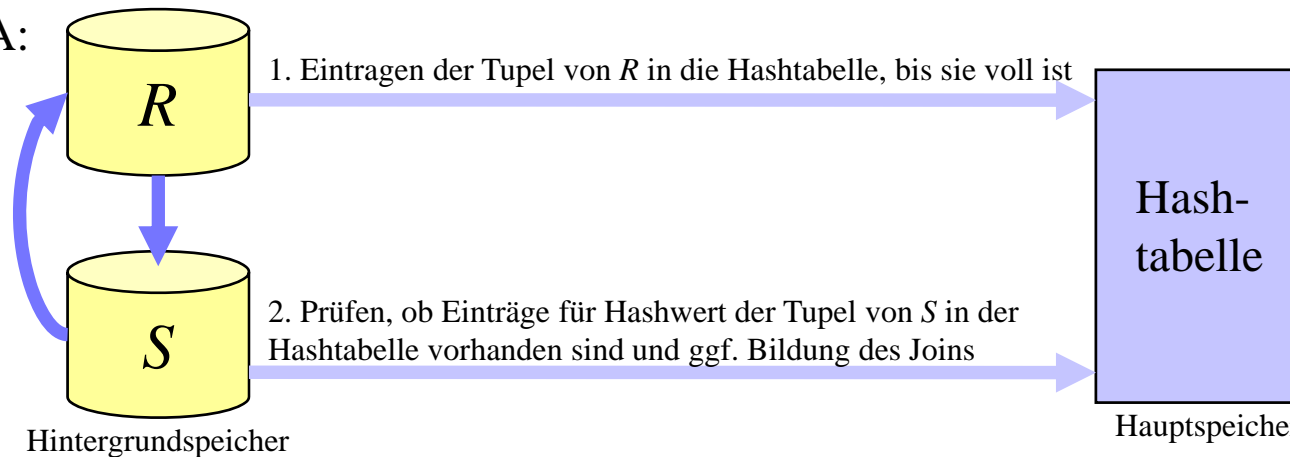


auf den einzelnen Blöcken: Hash-Join



- Ablauf**

Schritt A:



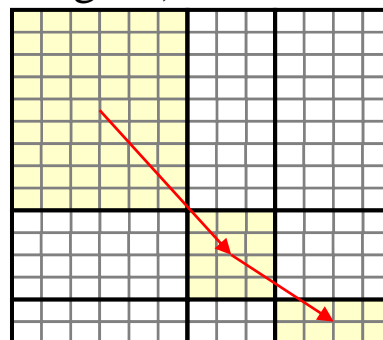
Schritt B: Wiederhole Schritt A für die restlichen Tupel von *R*

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, \dots, R_N und S_1, \dots, 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



R -Tupel, die in den Puffer passen

Auf den einzelnen Blöcken:
einfacher Hash-Join oder
Hashed-Loop-Join

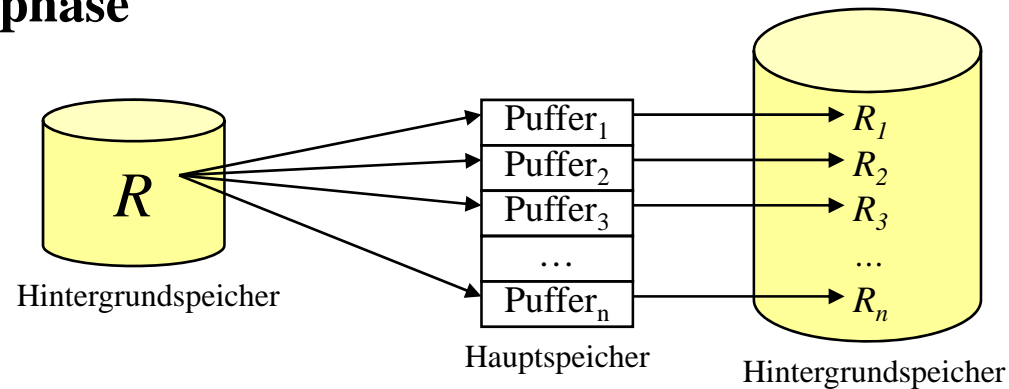
5.6 Implementierung der Joinoperation

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

- **Ablauf**

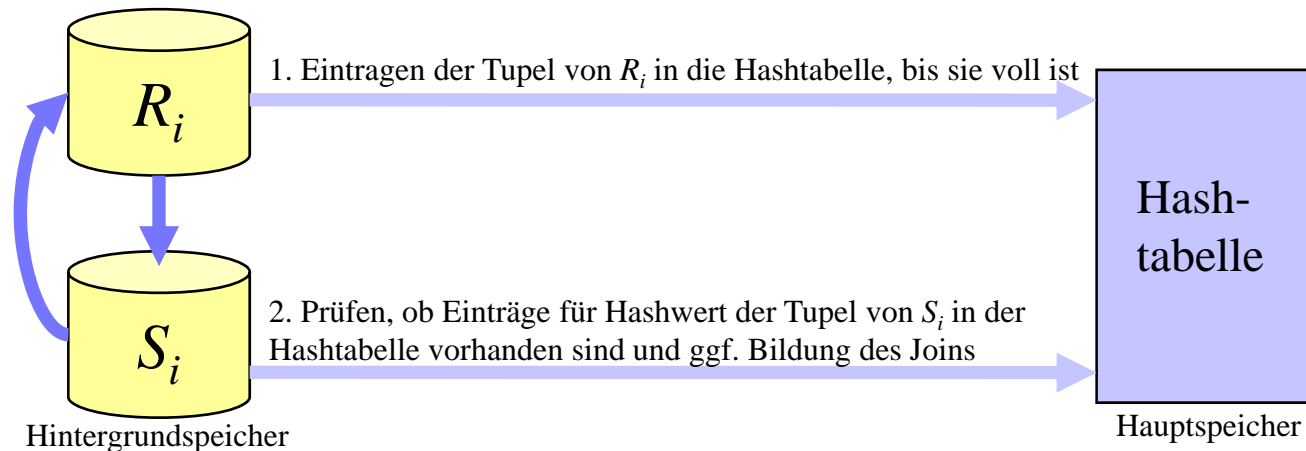
- **Partitionierungsphase**

Schritt A:



Schritt B: Wiederhole Schritt A für S

- **Join-Phase**



5.6 Implementierung der Joinoperation

Hybrid Hash-Join

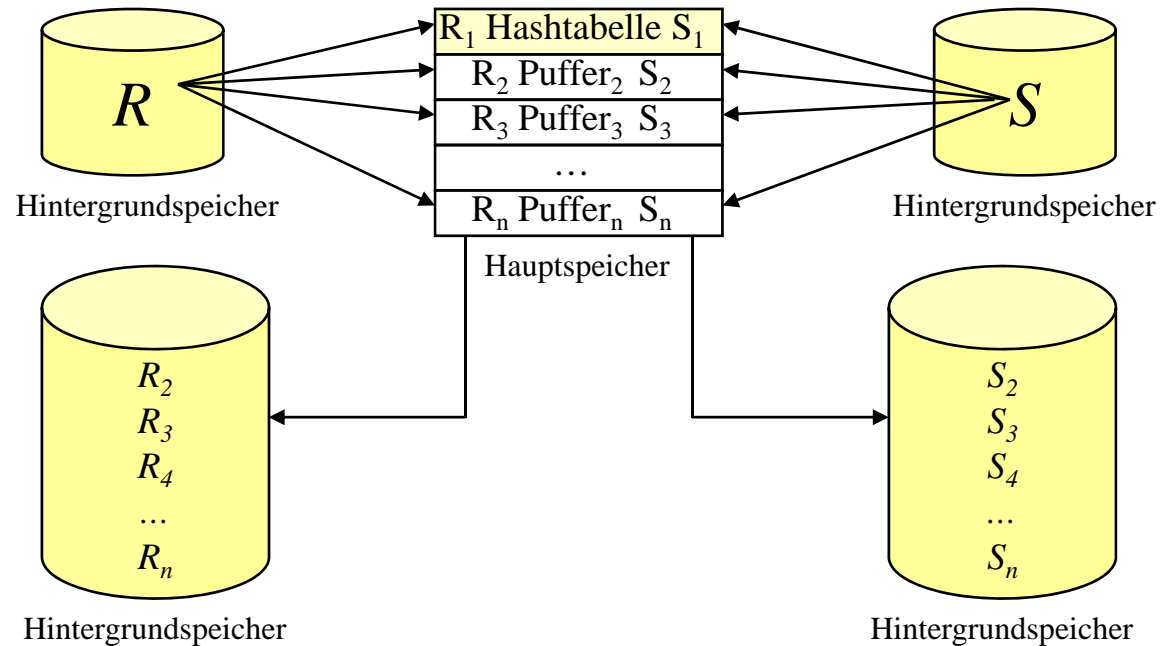
- **Algorithmus**

```
for each Tupel  $r \in R$  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  $s \in S$  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:



5.6 Implementierung der Joinoperation

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?