6. Anfragebearbeitung

Fokus: Effiziente Berecnung der Joinoperation

– Zur Vereinfachung: equi join

```
select *
from R r, S s
where r.A = s.B
```

also

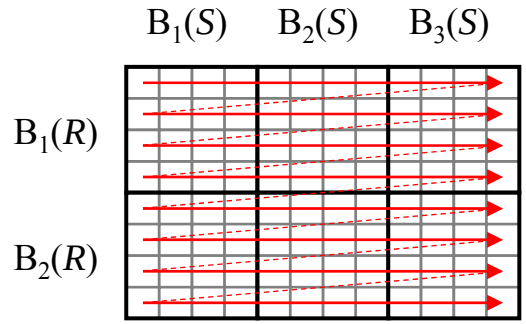
$$R \bowtie_{A=B} S$$

Einfacher Nested-Loop-Join

– Algoithmus

```
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.

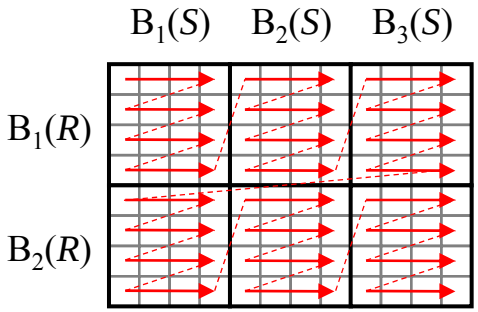
– Die Relation S wird |R| mal eingelesen: Performanz ist deshalb inakzeptabel

– S wird als *innere* Relation und R als *äußere* Relation bezeichnet

Nested-Block-Loop-Join

- Berücksichtige, dass Relationen R und S auf Blöcke verteilt sind
- Algoithmus Matrixnotation

```
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)
```



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	$B_R(2)$

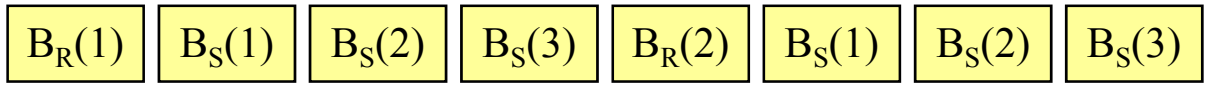
– Anzahl Blockzugriffe: $B_R + B_S \cdot B_R = 8$ Blockzugriffe ohne Cache
(B_R = Anzahl Blöcke der Relation R)

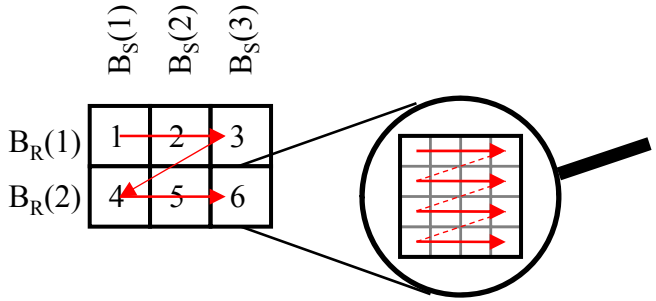
– D.h. die kleinere Relation sollte die äußere sein

Cache Strategien für Nested-Block-Loop-Join

Strategie 1

- Seiten der inneren Relation (S) 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)

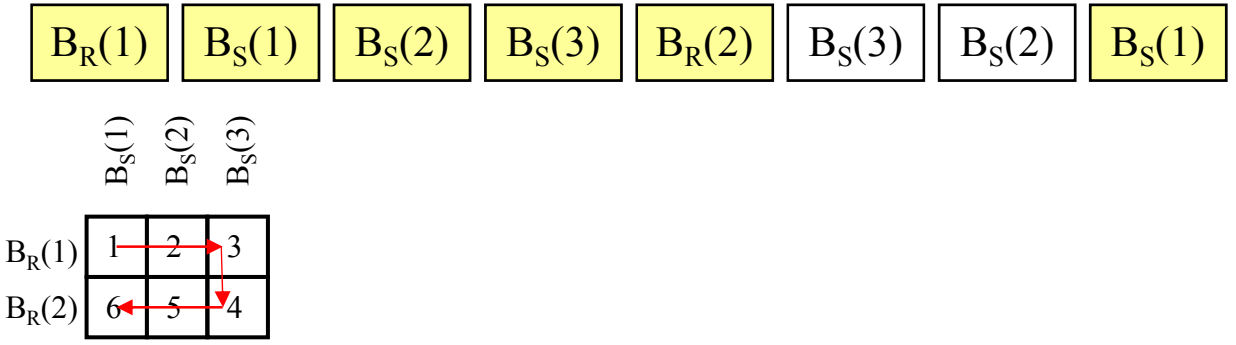




Cache Strategien für Nested-Block-Loop-Join

Strategie 2

- Seiten der inneren Relation (S) 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 äußere Relation (R) benötigt
- Blockzugriffe: BR+ BR · (BS - |C| + 1) + |C| - 1
- Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R



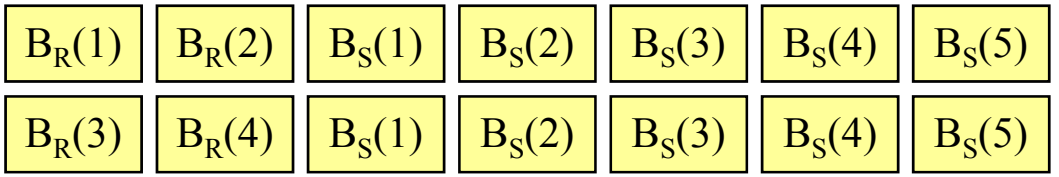
Cache Strategien für Nested-Block-Loop-Join

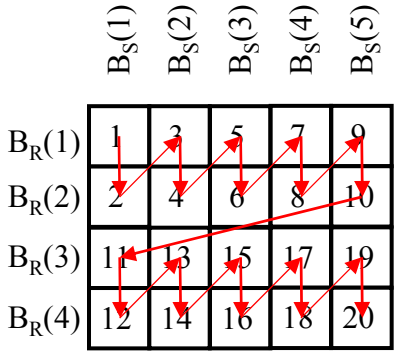
Strategie 3

- |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





Cache Strategien für Nested-Block-Loop-Join

Strategie 3 - Algorithmus

```
for i := 1 to B_R step |C| − 1 do
  lade Block B_R(i) ... B_R(i+|C|−2)
  for each Block B_S ∈ S do
    lade Block B_S
    for each Tupel r ∈ B_R(i) ... B_R(i+|C|−2) 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 ≠ '=' (also allen Joins außer Equi-Joins)

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)

- Parameter
  - 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)

Blockgrößen-Optimierung NBL-Join (cont)

– I/O-Kosten

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

Blockgrößen-Optimierung NBL-Join (cont)

- I/O-Kosten

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

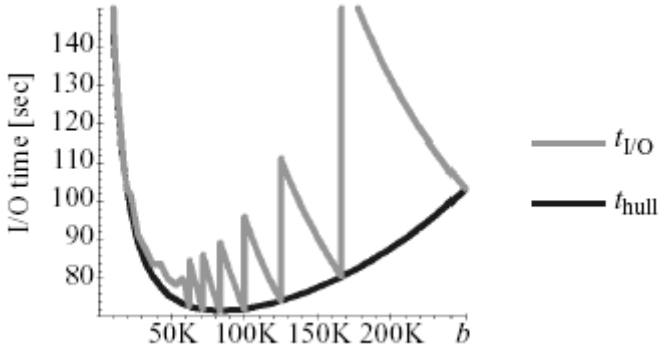
- Weglassen der Rundungsfunktion (unproblematisch für f_R, f_S >> b, d.h. relativer Fehler ist vernachlässigbar) ergibt stückweise differenzierbaren Term

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

Blockgrößen-Optimierung NBL-Join (cont)

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

– 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

- $f_R = f_S = 10$MByte
- 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 (aus 2. Ableitung erkennbar)
  - 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)

- Was sind denn eigentlich die CPU-Kosten?
  - Im wesentlichen müssen $|S|*|R|$ Vergleiche durchgeführt werden

- Vergleich: I/O- versus CPU-Kosten
  - Beispiel: |S| = |R| = 100.000 Tupel und f_S und f_R jeweils ca. 10 MB
  - Bearbeitungszeit eines Vergleichs durchschnittlich 0,1 µs
  - I/O-Zeit bei optimaler Seitengröße (85 KB) ca. 75 sec
  - CPU-Zeit: ca. 1000 sec

  => Der NBL-Join ist CPU-bound!!!

- Daher im Folgenden:

  Maßnahmen zur Senkung des CPU-Aufwands

Sort-Merge-Join

– Zweistufiger Algorithmus

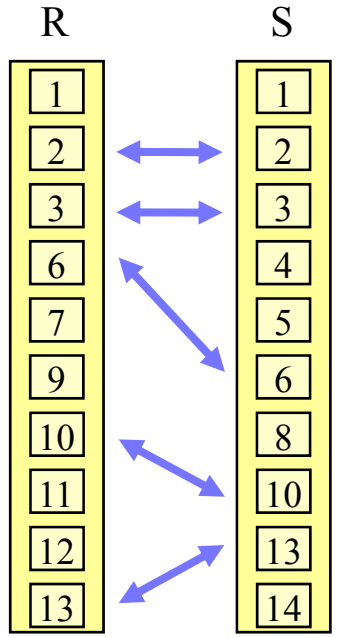
- 1.Schritt:

sortiere R bzgl. Attribut A

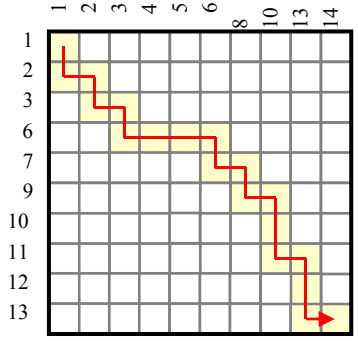
sortiere S bzgl. Attribut B

- 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);
```



Matrixnotation



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?

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

Einfacher Hash-Join

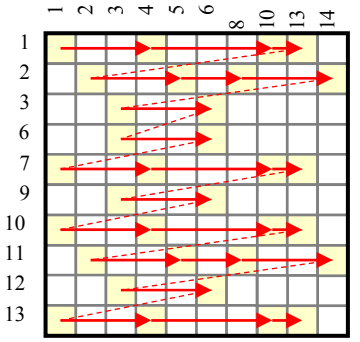
- Reduktion des CPU-Aufwandes bei der Join-Berechnung
  - Der Join-Partner eines S-Tupels wird gezielt mit Hilfe eines Hash-Verfahrens gesucht, anstatt das S-Tupel sequentiell mit jedem Tupel der Relation R zu vergleichen.
  - Zu diesem Zweck wird die Relation R gehasht, d.h. es wird zu allen Tupeln 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.

Einfacher Hash-Join (cont.)

– Algoithmus

```
for each Tupel r ∈ R do
    berechne adr = hash(r);
    speichere r in HT[adr] ab;
for each Tupel s ∈ S do //prüfe in der Hashtabelle HT
    berechne adr = hash(s);
    for each Tupel r ∈ HT[adr] do
        if r.A = s.B then
            result := result ∪ ((r – r.A) × s)
```

Matrixnotation



hash(x) = MOD 3

– 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)

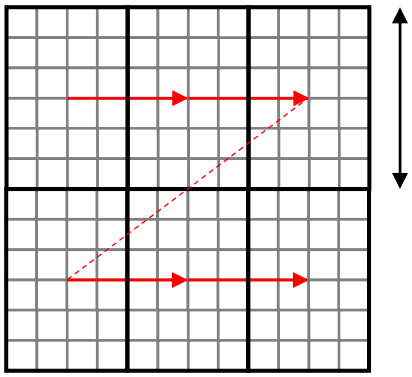
Hashed-Loop-Join

- 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;
```

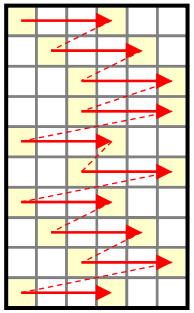
Hashed-Loop-Join (cont.)

Matrixnotation



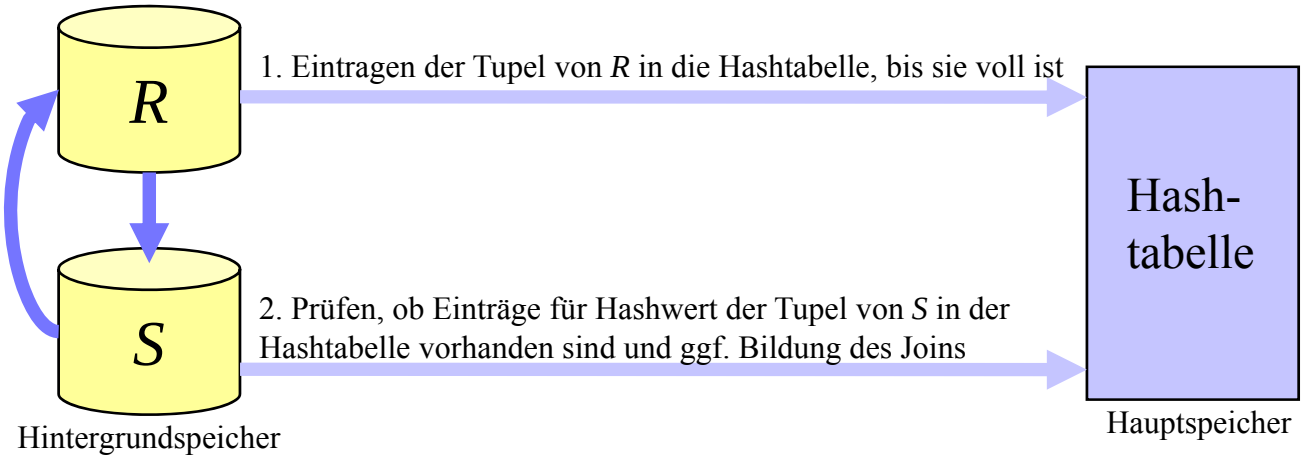
R-Tupel, die in den Puffer passen

auf den einzelnen Blöcken: Hash-Join



Ablauf

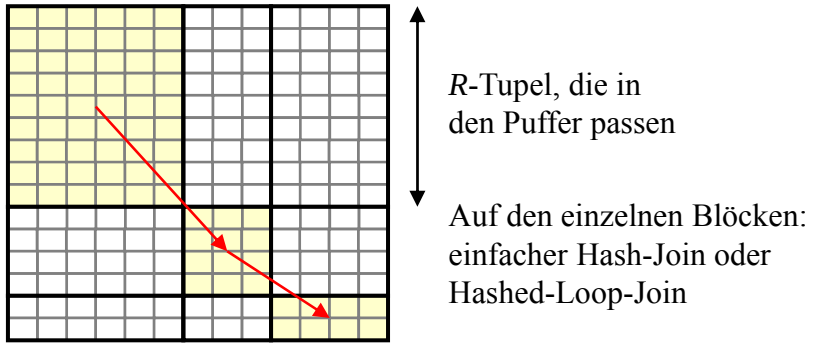
Schritt A:



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

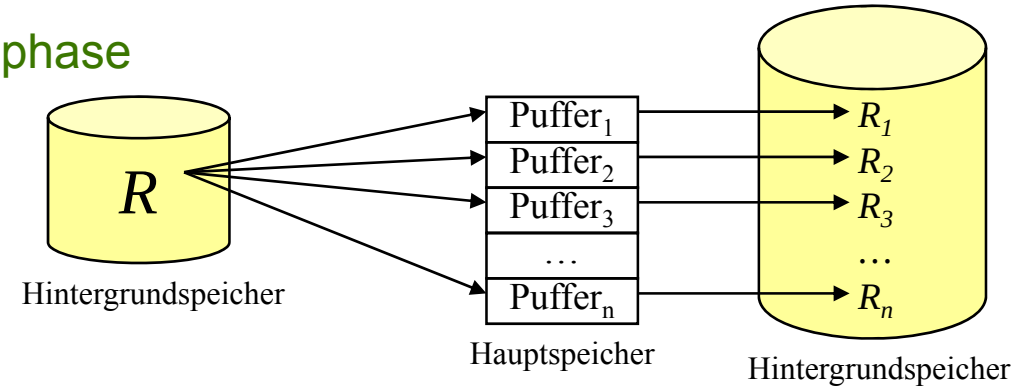
Hash-Partitioned-Join (a.k.a. GRACE)

- Hashed-Loop-Join zerlegt Relationen willkürlich in Blöcke, jeder Block von R muss mit jedem Block von S kombiniert werden
- Idee: Zerlege R und S mit einer Hashfunktion in Partitionen, so dass nur Partitionen mit demselben Hash-Key kombiniert werden müssen
- Zweistufiges Verfahren
  - Partitioniere die Relationen R und S in $R_1,\ldots,R_N$ und $S_1,\ldots,S_N$
  - Berechne den Join der einzelnen Partitionen R_i und S_i mit einem beliebigen Join Verfahren
- Matrixnotation



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

- Ablauf
  - Partitionierungsphase



  - Join-Phase