Joins / Implementierung von Joins

Join

wichtigste Operation, insbesondere in relationalen DBS:

- komplexe Benutzeranfragen (Star-Joins)
- Normalisierung der Relationen (Snowflake-Schema)
- verschiedene Sichten (“views”) auf die Basisrelationen

Optimierungsziele:

- Reduzieren der I/O-Kosten durch
  - Verwendung von Indexen
  - durch vorberechnete Joins
  - gute Ausnutzung eines Puffers
- Reduzieren der Vergleiche
  (kann insbesondere bei komplexen Datentypen sehr wichtig sein)
- Reduzieren der Kommunikationskosten (besonders wichtig in verteilten DBS)
- Verbesserung der Abarbeitungsreihenfolge in einem Mehrweg-Join

Wichtige Joinoperationen

- gegeben zwei Relationen R und S mit n bzw. m Attributen
- | R | , | S | = #Tupel in R, S
- A sei ein Attribut von R und B ein Attribut von S, die bzgl. des Joins verträglich sind
- Q sei im Folgenden die sich aus dem Join ergebende Relation
- $\otimes$ sei ein Operator, der zwei Tupel zu einem Tupel verknüpft

Beispiel:

Relation R

Angesteller	Gehaltsgruppe
Müller	1
Schneider	2
Schuster	1
Schmidt	2

Relation S

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

(Müller,1) $\otimes$ (2, 20.000) = (Müller,1,2, 20.000)

1. θ-Join

$Q_\theta(R, S, A, B) = \{t \mid t = r \otimes s, r \in R, s \in S \text{ und } \prod_A(r)\ \theta\ \prod_B(s)\}$
$\theta \in \{=, <, >, \leq, \geq, \neq\}$

Beispiel:
$Q_<$(R,S, Gehaltsgruppe, Gehaltsgruppe) ergibt:

Angesteller	Gehaltsgruppe	Gehaltsgruppe	Gehalt
Müller	1	2	20.000
Schuster	1	2	20.000
Müller	1	3	30.000
Schneider	2	3	30.000
Schuster	1	3	30.000
Schmidt	2	3	30.000

- für θ = '=' wird der Join auch als Equi-Join bezeichnet
- $|Q_\theta| / (|R| * |S|)$ ist die Selektivität des θ-Joins

2. Natürlicher Join

$Q^*(R, S, A, B) = \{t \mid t = (r - \prod_A(r)) \otimes s, r \in R, s \in S \text{ und } \prod_A(r) = \prod_B(s)\}$,
wobei A=B die Menge aller gemeinsamer Attribute

Beispiel:

Q*(R,S, Gehaltsgruppe, Gehaltsgruppe) ergibt:

Angesteller	Gehaltsgruppe	Gehalt
Müller	1	10.000
Schneider	2	20.000
Schuster	1	10.000
Schmidt	2	20.000

- wichtigste Join-Operation in DBS

3. Semi-Join

$q^*(S, R, A, B) = \{t \mid t = s, s \in S, r \in R \text{ und } \prod_A(s) = \prod_B(r)\}$

Beispiel:
q^*(S, R, Gehaltsgruppe, Gehaltsgruppe) ergibt:

Gehaltsgruppe	Gehalt
1	10.000
2	20.000

- entspricht einem natürlichen Join mit anschließender Projektion
- q^* = alle Tupel der Relation S, die für die Berechnung eines natürlichen Joins mit der Relation R benötigt werden
- sehr wichtig in verteilten DBS zur Reduzierung der teuren Kommunikationskosten

4. Outer-Join

- $T_1 = \{t \mid t = r \otimes Null, r \in R \setminus q^*(R,S,A,B)\}$
- $T_2 = \{t \mid t = Null \otimes s, s \in S \setminus q^*(S,R,A,B)\}$
- ${}^{=}Q^{=}(R, S, A, B) = Q(R, S, A, B,=) \cup T_1 \cup T_2$ (full outer join)
- ${}^{=}Q(R, S, A, B) = Q(R, S, A, B,=) \cup T_1$ (left outer join)
  die linke Relation bleibt verlustfrei.
- $Q^{=}(R, S, A, B) = Q(R, S, A, B,=) \cup T_2$ (right outer join)
  die rechte Relation bleibt verlustfrei.

Beispiel:
$Q^{=}$ (R, S, Gehaltsgruppe, Gehaltsgruppe) ergibt:

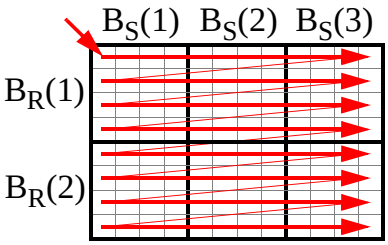
Angesteller	Gehaltsgruppe	Gehaltsgruppe	Gehalt
Müller	1	1	10.000
Schneider	2	2	20.000
Schuster	1	1	10.000
Schmidt	2	2	20.000
⊥	⊥	3	30.000

Implementierung von Joins

1. Einfacher Nested Loop Join

```
for each Tupel r in R do
    for each Tupel s in S do
        if r(A) = s(B) then
            Q* := Q* ∪ (r ⊗ s)
```

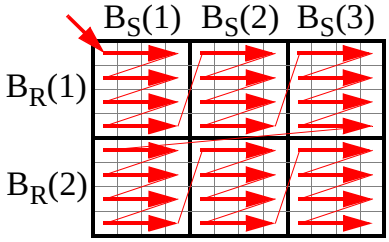
- Die Relation S wird |R| mal (|R|: Anzahl der Tupel von R) eingelesen
  Performanz ist deshalb inakzeptabel.
- Der einfache Nested Loop Join entspricht der Bildung des kartesischen Produktes in kanonischer Ordnung mit anschließender Selektion.
- S wird als innere Relation und R als äußere Relation bezeichnet
- Matrixdarstellung der Joinoperation
  (stellt Reihenfolge von Block- und Tupelpaarungen dar)



2. **Nested Block Loop Join** (oft auch: “Nested Loop Join”)

```
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
                    Q* := Q* ∪ (r ⊗ s)
}   }
```

- Nested Loop Joins sind geeignet für alle Join-Prädikate θ (’=‘, ’<‘, ’>‘, ’≤‘, usw.)
- S wird als innere Relation und R als äußere Relation bezeichnet



Beispiel

Relation S

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

Relation R

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

- Ohne Cache: 8 Zugriffe
- Blockzugriffe = $B_R + B_S \cdot B_R$, B_R Anzahl der Blöcke der Relation R, d.h. die kleinere Relation sollte die äußere sein.

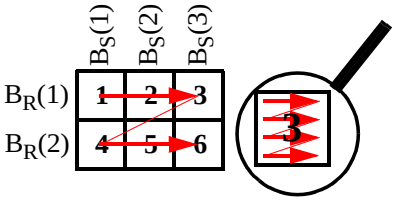
Cache-Strategien für Nested-Loop-Join:

Bei jeder Cache-Strategie wird die äußere Relation genau ein mal eingelesen.

- Seiten der inneren Relation S in Cache halten:
  Cache wird überhaupt nicht ausgenutzt, wenn Cache kleiner als Relation S ist.

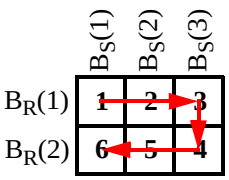
Beispiel (2 Seiten Cache für S, 1 Seite Arbeitspuffer für R):

$B_R(1)$ $B_S(1)$ $B_S(2)$ $B_S(3)$ $B_R(2)$ $B_S(1)$ $B_S(2)$ $B_S(3)$: Zugriff (Platte)



- Seiten der inneren Relation in Cache, aber innere Relation jedes zweite mal rückwärts:

$B_R(1)$ $B_S(1)$ $B_S(2)$ $B_S(3)$ $B_R(2)$ $B_S(3)$ $B_S(2)$ $B_S(1)$



Pro Durchlauf der äußeren Schleife werden $(|C| - 1)$ Blockzugriffe eingespart (ab 2. Durchlauf, |C| ist Anzahl der Blöcke, die in den Cache passen). Ein Cache-Block wird jeweils für die R-Relation benötigt:

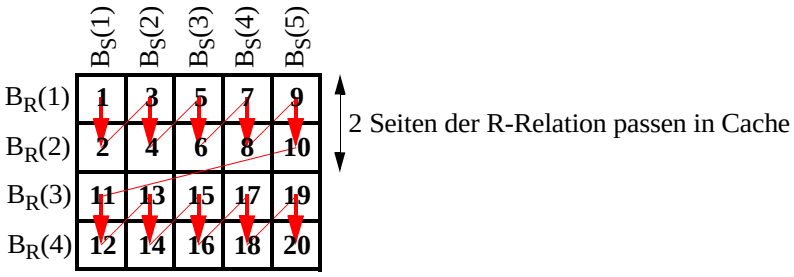
Blockzugriffe = $B_R + B_R \cdot (B_S - |C| + 1) + |C| - 1$

- LRU Strategie: wie Strategie 1 (Seiten der inneren Relation in Cache)

  $B_R(1)$ $B_S(1)$ $B_S(2)$ $B_S(3)$ $B_R(2)$ $B_S(1)$ $B_S(2)$ $B_S(3)$

- $(|C| - 1)$ Blöcke der äußeren Relation werden in den Cache eingelesen. Zu jedem Block der inneren Relation werden diese Blöcke gejoint:

  $B_R(1)$ $B_R(2)$ $B_S(1)$ $B_S(2)$ $B_S(3)$



Blockzugriffe = $B_R + B_S \cdot \left\lceil \frac{B_R}{|C| - 1} \right\rceil$

Algorithmus für die 3. Variante:

```
for i:=0 to B_R step |C| do {
    Lade Block B_R(i)..B_R(i+|C|-1) ;
```



```
        for each Block B_S in S do {
            Lade Block B_S;
            for each Tupel r in B_R(i)..B_R(i+|C|-1) do
                for each Tupel s in B_S do
                    if r(A) = s(B) then
                        Q* := Q* ∪ (r ⊗ s)
}   }
```

Leistung

- |R|*|S| Vergleiche von Tupel (ist nur bei schlechter Selektivität gerechtfertigt)
- effizienteste Ausführung von θ-Joins mit θ ≠ '='

Optimierung der Blockgröße bei NL-Join

Problem:

- bei kleiner Blockgröße: innere Relation wird in sehr kleinen Schritten eingelesen bei jedem I/O-Auftrag Latenzzeit des Plattenlaufwerks
- bei großer 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: Wieviel von Cache für äußere/innere Relation

Kosten:

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

Dann ergibt sich folgende I/O-Zeit für den gesamten Join:

$$t_{\text{NLJoin}} \approx \frac{B_R}{|C|-1} \cdot (2t_{\text{seek}} + t_{\text{lat}} + b \cdot (|C|-1) \cdot t_{\text{tr}}) + B_S \cdot \frac{B_R}{|C|-1} \cdot (t_{\text{lat}} + b \cdot t_{\text{tr}})$$

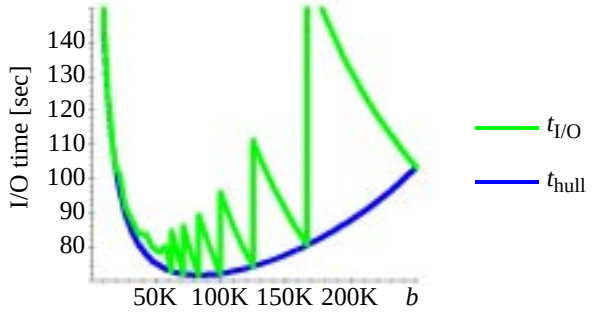
Der B_R-Scan wird vernachlässigt, da nur 1 mal und in großen Blöcken.

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

Um den Term differenzierbar zu machen, läßt man die Rundungs-Funktion weg. Dies ist unproblematisch für f_R, f_S (>> b, d.h. der relative Fehler ist vernachlässigbar).

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

Wir optimieren zunächst die Hüllfunktion: $t_{\text{hull}} = \left(\frac{f_S \cdot f_R}{b^2 \cdot ((c/b) - 1)} \right) \cdot (t_{\text{lat}} + b \cdot t_{\text{tr}})$



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

Optimierung durch Differenzieren:

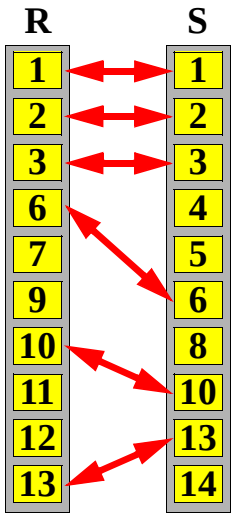
Die Ableitung wird mit Null gleichgesetzt. 2 Lösungen, von denen nur eine positiv ist:

$$0 = \frac{\partial}{\partial b} t_{\text{hull}} \Rightarrow b = \frac{\sqrt{t_{\text{lat}}^2 + t_{\text{tr}} \cdot t_{\text{lat}} \cdot c} - t_{\text{lat}}}{t_{\text{tr}}}$$ Lösung ist Minimum (siehe 2. Ableitung).

An den Stellen, an denen $\lfloor c/b \rfloor$ konstant ist, ist t_{NLJoin} streng monoton fallend (die Ableitung ist negativ). Deshalb kann das Minimum von t_{NLJoin} nur an der ersten Sprungstelle links oder rechts vom Minimum von t_{hull} sein: $b_1 = c / \left\lfloor \frac{c}{b} \right\rfloor$ $\quad$ $b_2 = c / \left\lceil \frac{c}{b} \right\rceil$

CPU-Kosten:
Im Wesentlichen müssen |S|*|R| Vergleiche durchgeführt werden. Bei 0.1 µs pro Vergleich und 100.000 Tupel pro Relation ergeben sich 1000 s Bearbeitungszeit, d.h. wesentlich mehr als die 75 s I/O-Zeit. Der NL-Join ist also *CPU-bound*. Maßnahmen zur Senkung des CPU-Aufwandes später (Hashed-Loop-Join).

3. Sort-Merge Join



zweistufiger Algorithmus
1. Schritt:
   - Sortiere Relation R bzgl. dem Attribut A;
   - Sortiere Relation S bzgl. dem Attribut B;

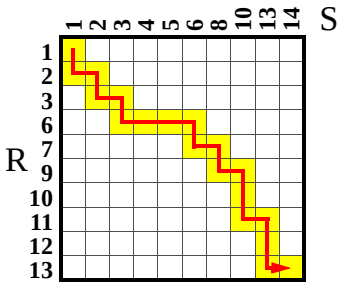
2. Schritt:

```
j=1; s = erstes Tupel der Relation S;
for i = 1 to |R| do {
    r = i-tes Tupel der Relation R;
    while s(B) < r(A)
        {j = j+1; s = j-tes Tupel der Relation S }
    if r(A) = s(B) then
        Q* = Q* ∪ (((r - r(A)) ⊗ s)
    while s(B) =r(A)
```



```
        {j = j+1; s = j-tes Tupel der Relation S; Q* = Q* ∪ (((r - r(A)) ⊗ s)}
}
```

Matrixnotation:



Beispiel

Relation S

Angesteller	Gehaltsgruppe
Müller	1
Schuster	1
Schneider	2
Schmidt	2

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

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 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 NL-Join umgeschaltet werden.

4. 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, statt sequentiell mit jedem Tupel der R-Relation zu vergleichen
- Zu diesem Zweck wird die R-Relation gehasht, d.h. 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 Hashkey

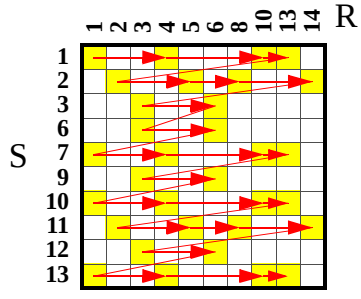
```
for each Tupel r in R do /* Erzeugen der Hashtabelle HT */
      { berechne adr = Hash(r);
      speichere das Tupel r in HT[adr] }
for each Tupel s in S do /* Prüfen in der Hashtabelle HT */
```



```
      { berechne adr = Hash(s);
      for each Tupel r in HT[adr] do
            {if r(A) = s(B) then
                  Q* = Q* ∪ (((r - r(A)) ⊗ s)
      }     }
```

- im Idealfall soll der Join im Hauptspeicher ablaufen:
  Hashtabelle soll für die kleinere Relation erzeugt werden
- Hash-Join Verfahren können nur für Equi-Join und Natürlichen Join effizient genutzt werden

Matrixnotation:



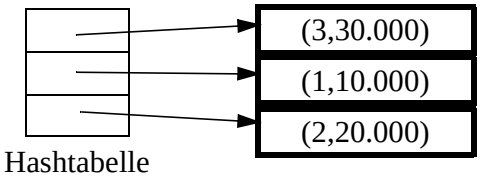
h(x) = x MOD 3

Beispiel

Hash = Gehaltsgruppe *mod* 3

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000



Hash((Müller,1)) = 1
Hash((Schneider,2)) = 2
…

Leistung:

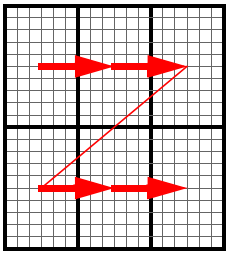
- hängt stark ab von der Güte der Hashfunktion: O(|R| + |S|) im Idealfall.
- verschlechtert sich, wenn Werte ungleichmäßig belegt sind.
- Modifikation ist notwendig, wenn Hauptspeicher zu klein (kleiner als R) ist.

5. Hashed-Loop Join

- Kombination aus dem Nested-Loop-Join und dem einfachen Hash-Join
- Relation R wird in große Blöcke eingeteilt, deren Hash-Tabellen in den Puffer passen
- Für jeden dieser Blöcke wird die Relation S gescannt und ein einfacher Hash-Join durchgeführt.
- Blockgrößenoptimierung wie bei NL-Join

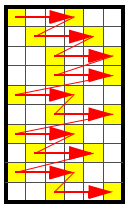
```
repeat
    { lese soviel Tupel von R in Hauptspeicher bis der Platz aufgebraucht ist;
    erzeuge für diese Tupel eine Hashtabelle HT;
    for each Tupel s in Relation S do
        { berechne adr := Hash(s);
        for each Tupel r in HT[adr] do
            {if r(A) = s(B) then
                Q* := Q* ∪ (((r - r(A)) ⊗ s)
        }
    }
} until alle Tupel der Relation R sind eingelesen;
```

Matrix-Notation:



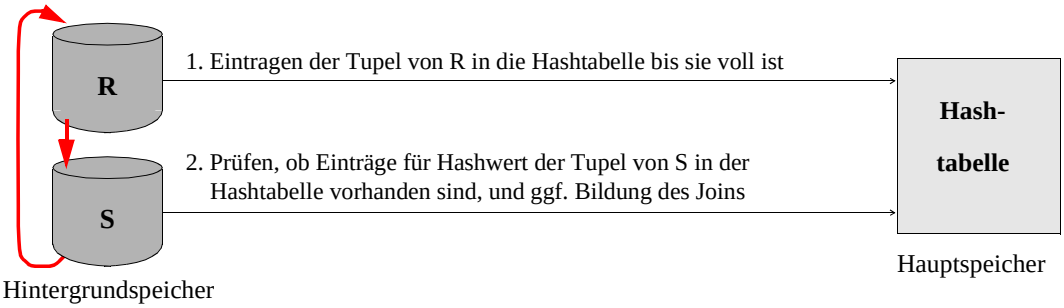
R-Tupel, die in den Puffer passen

auf den einzelnen Blöcken: Hash-Join



Ablauf des Hashed-Loop Join

Schritt A:

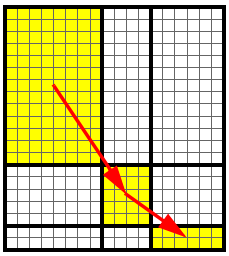


Schritt B: Wiederhole Schritt A für die restlichen Tupel von R

5. 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 Hashkey 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 einfachen Hash-Join oder einem Hashed-Loop Join (wenn Partition zu groß)

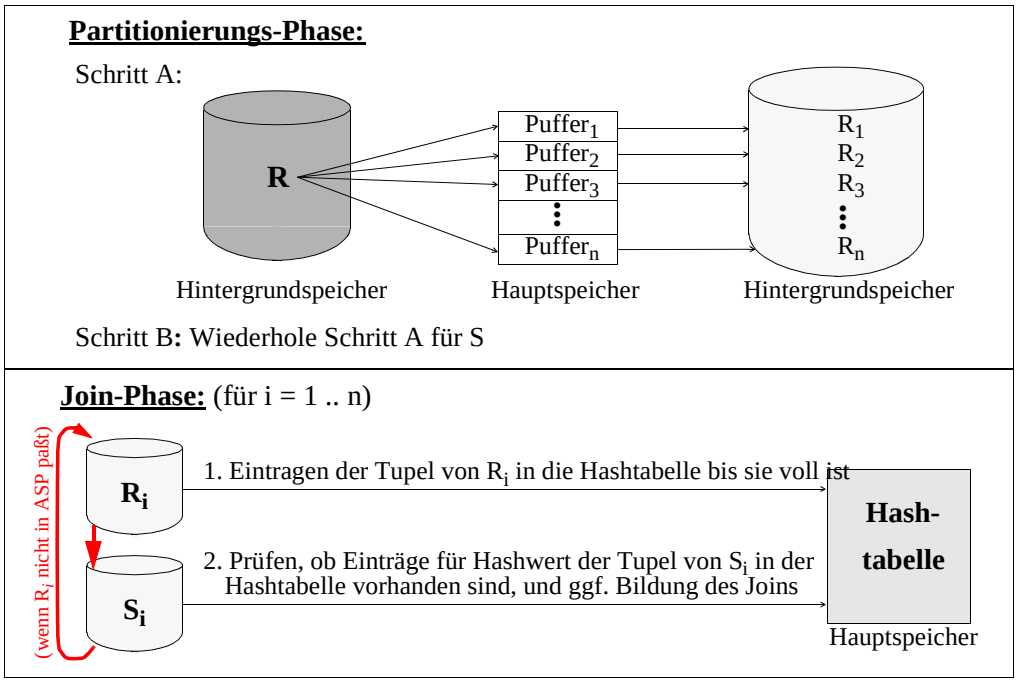
Matrix-Notation:



R-Tupel, die in den Puffer passen

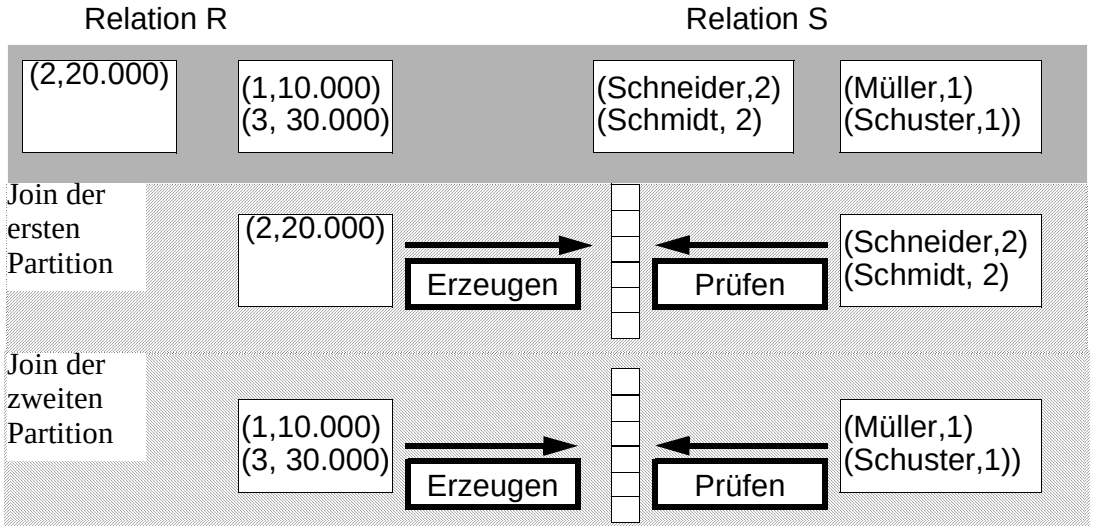
auf den einzelnen Blöcken:
einfacher Hash-Join
oder Hashed Loop-Join

Ablauf des Hashed-Partitioned Join (GRACE)



Beispiel

- Partitionierungsphase:
- Hash(Tupel) = Gehaltsgruppe *mod* 2 (zwei Datensätze pro Pufferseite)



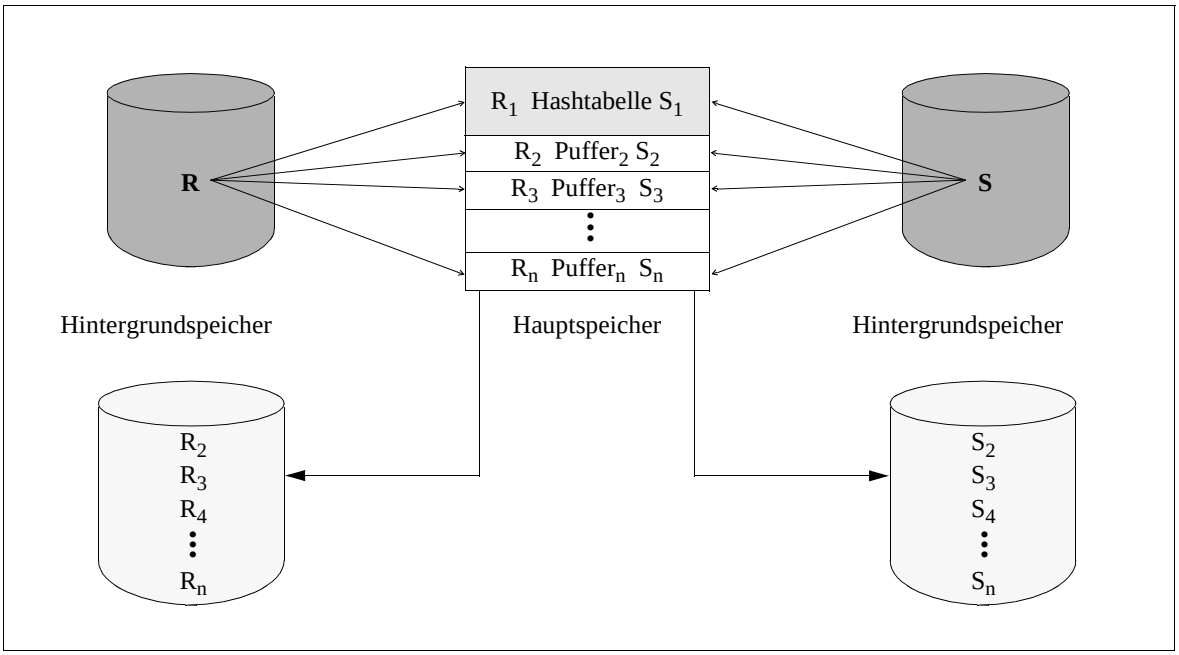
7. Hybrid-Hash Join

for each Tupel r in R do
{ adr = Hash(r);
if adr=1 then
füge das Tupel in eine Hashtabelle HT ein (bzgl. neuer Hashfunktion);
else
speichere das Tupel in einem Puffer BR_{adr};
/* wenn der Puffer voll ist, wird er stets auf die Platte geschrieben */

for each Tupel s in S do
{adr = Hash(s);
if adr=1 then
suche in der Hashtabelle HT nach entsprechenden Tupel r mit r(A) = s(B);
else
speichere das Tupel 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

Ablauf der Partitionierungsphase des Hybrid-Hash Joins



Leistung

- Reduzierung der I/O-Kosten (im Vergleich zu GRACE), da eine Partition im Hauptspeicher gehalten wird
- ist immer dann vorteilhaft, wenn viel Hauptspeicher zur Verfügung steht, aber nicht die Relation R komplett im Hauptspeicher gehalten werden kann

Probleme aller Hash-Join-Verfahren

- ungleiche Datenverteilung (extrem hohe Belegung eines Wertes durch Datensätze)
- Wie können die Partitionen der einzelnen Verfahren gewählt werden?

8. Index Join

- Natürlicher Join zwischen R und S bezüglich eines nicht-eindeutigen Attributs B
- S hat einen *clustering index* auf dem Attribut B
- R ist ohne Index gepackt gespeichert
- Annahme: Jedes Tupel in R hat mindestes einen Join-Partner in S (R.B $\subseteq$ S.B)

Folgende Schleife berechnet den Join:

for each block *P* of *R* do
  for each tuple *ab* on *P* do
    join *ab* with each tuple in $\sigma_{B=ab.B}(S)$

Kostenschätzung:

Sei I die *image size*, d.h. die Anzahl *verschiedener* B-Werte in S.
Falls die *image size* größer als die Anzahl der S-Blöcke ist, muß für jedes Tupel in R ein Block von S eingelesen werden (eigentlich ein ganzer Index-Pfad, aber dies wird wegen Cache vernachlässigt; das Directory wird hier als Cache-resident angenommen).

$$\text{Blockzugriffe} = B_R + |R|$$

Falls die *image size* kleiner als die Anzahl der S-Blöcke ist, müssen in jedem Schleifendurchlauf im Durchschnitt B_S/I Blöcke eingelesen werden:

$$\text{Blockzugriffe} = B_R + |R| \bullet \max(1, B_S/I)$$

Beispiel:

- $|R| = 2000$; $B_R = 200$
- $|S| = 5000$; $B_S = 500$
- $I = 100$ (Anzahl versch. B-Werte in S).

$$\text{Blockzugriffe} = 200 + 2000(500/100) = 10200$$

9. Two-Index-Join

Bestimme zunächst aus der kleineren Relation die verschiedenen B-Werte
Selektiere dann aus beiden Relationen die passenden Tupel für jeden B-Wert:

for each B-value b do
join the tuples of $\sigma_{B=b}(R)$ with $\sigma_{B=b}(S)$

Analyse:

- Zunächst werden B_R Blöcke eingelesen um die verschiedenen B-Werte zu bestimmen
- In jedem Iterationsschritt werden max {1, B_R/I} Blöcke der R-Relation eingelesen
- In jedem Iterationsschritt werden max {1, B_S/J} Blöcke der S-Relation eingelesen, wobei I, J die Anzahl der verschiedenen B-Werte in der R- bzw. S-Relation sind

  Blockzugriffe = B_R + I (max {1, B_R/I} + max {1, B_S/J})

Beispiel:

- |R| = 2000; B_R = 200
- |S| = 5000; B_S = 500
- I = 50; J = 100;
  Blockzugriffe = 200 + 50 (max {1, 200/50} + max {1, 500/100}) = 650

⇒ es lohnt sich oft, temporär für den Join Clustering Indexe anzulegen