Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme II

Sommersemester 2007

Kapitel 4: Relationale Anfragebearbeitung

Vorlesung: Christian Böhm

Skript © 2007 Christian Böhm

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBSII

Inhalt

Inhalt

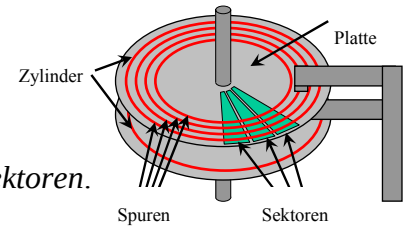
Engpassprobleme heutiger Von-Neumann-Rechner

Zur Vereinfachung unterscheidet man meist nur zwischen

- **CPU-bound**
  CPU, Arbeitsspeicher und Bus bilden den Hauptengpass
- **I/O-bound**
  Hintergrundspeicher und I/O bilden den Hauptengpass

Schematischer Aufbau einer Magnetplatte (1)

- Ein Plattenspeichersystem besteht aus *Platten*
- Die Oberfläche der Platten besteht aus *Spuren*
- Die Spuren bestehen aus *Sektoren*.
- *Zylinder* = alle Spuren mit konstantem Radius



- Platten rotieren um gemeinsame Achse, der Arm ist in radialer Richtung bewegbar

Schematischer Aufbau einer Magnetplatte (2)

- Zugriff auf eine Seite:
  - Setze den Arm auf den gewünschten Zylinder (*Suchen*)
  - Warte bis die Platte so rotiert ist, dass sich der Anfang der Seite unter dem Arm befindet (*Latenz*)
  - Übertrage die Seite in den Hauptspeicher (*Transfer*)
- Zugriffszeit = Suchzeit + Latenzzeit + Transferzeit
- I/O-Rate = erwartete Anzahl von Zugriffen pro Sekunde
- Übertragungsrate = maximale Anzahl übertragener Bytes pro Sekunde (Bandbreite)

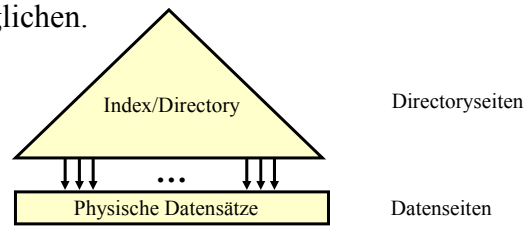
Indexstrukturen

- Zur effizienten Anfragebearbeitung werden oftmals geeignete Datenstrukturen und Speicherungsverfahren (**Indexstrukturen**) eingesetzt.
- Üblicherweise erfolgt die Verwaltung der Daten sequentiell auf einem Speichermedium. Die Bearbeitung einer Suchanfrage hat dabei einen Zeitaufwand von O(n), da im ungünstigsten Fall alle n Datensätze durchsucht werden müssen.
- Wird ein bestimmter Datensatz anhand eines Suchkriteriums gesucht, kann über eine Indexstruktur eine aufwändige Suche vermieden werden. Der Index erlaubt es, die Position des Datensatzes innerhalb des Mediums schnell zu bestimmen.

Indexstrukturen: Directory- vs. Datenseiten

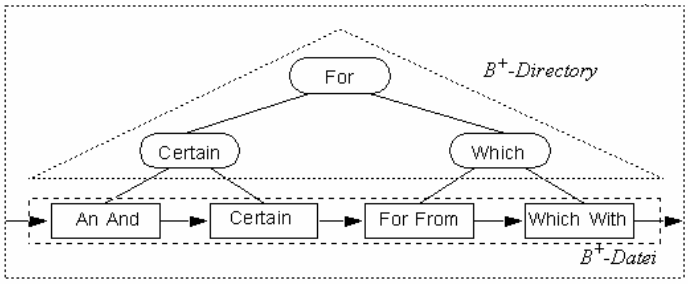
Baumartige Indexstrukturen bestehen üblicherweise aus Directory- und Datenseiten:

- Die eigentlichen physischen Datensätze werden in den **Datenseiten** gespeichert (Blattknoten des Baums).
- Die **Directoryseiten** sind die inneren Knoten des Baums und speichern die Directory-Einträge, die aus aggregierten Zugriffsinformationen bestehen und die Navigation im Baum ermöglichen.



Indexstrukturen: B$^+$-Baum

- Erweiterung des B-Baums (vgl. Vorlesung Effiziente Algorithmen)
- Bei einem B$^+$-Baum werden die eigentlichen Datenelemente nur in den Blattknoten gespeichert, während die inneren Knoten lediglich Schlüssel enthalten.
- B$^+$-Baum für die Zeichenketten:
  An, And, Certain, For, From, Which, With



Modellierung der I/O-Kosten (1)

- In DBS sind zwei verschiedene I/O-Zugriffsmuster vorherrschend:
  - *Sequentielles Lesen* einer großen Datei (Verarbeitung von Relationen ohne Index)
  - *Wahlfreies Lesen* von Blöcken konstanter Größe, wobei die einzelnen Blöcke an wahlfreien Positionen beginnen (Verarbeitung von Relationen mit Hilfe eines Index)
- Seien

f	Größe der Datei in MByte
a	Anzahl der Blockzugriffe
c_{puffer}	Größe des Puffers im Arbeitsspeicher in MByte
c_{index}	Blockgröße des Index in MByte
t_{seek}	Suchzeit in ms
t_{lat}	Latenzzeit in ms
t_{tr}	Transferleistung des Laufwerkes in ms/MByte

Modellierung der I/O-Kosten (2)

Sequentielles Lesen

- Da der Arbeitsspeicher begrenzt ist, erfolgt das sequentielle Lesen einer Datei in einzelnen Blöcken, die zwischen den I/O-Aufträgen verarbeitet werden:
  - Bei der ersten Leseoperation wird der Schreib-/Lesekopf auf die entsprechende Position gesetzt.
  - Bei jeder weiteren Leseoperation fallen nur noch Latenzzeit und Transferzeit an.

$$t_{scan} = t_{seek} + f \cdot t_{tr} + \left\lceil \frac{f}{c_{puffer}} \right\rceil \cdot t_{lat}$$

- Meist wählt man den Puffer so groß, dass die Transferzeit pro Leseoperation wesentlich höher als die Latenzzeit ist. In diesem Fall können Latenz- und Suchzeit vernachlässigt werden:

$$t_{scan} \approx f \cdot t_{tr}$$

Modellierung der I/O-Kosten (3)

Wahlfreies Lesen

- Bei wahlfreien Zugriffen fallen bei jedem Auftrag sowohl Transferzeit, Latenzzeit als auch Suchzeit an:

$$t_{random} = (t_{seek} + t_{lat} + c_{index} \cdot t_{tr}) \cdot a$$

- Verglichen mit der scanbasierten Verarbeitung ist die Größe c_{index} einer Transfereinheit hier nicht durch den zur Verfügung stehenden Arbeitsspeicher, sondern durch die Blockgröße des Indexes vorgegeben (und typischerweise wesentlich kleiner, z.B. 4-8 KBytes).

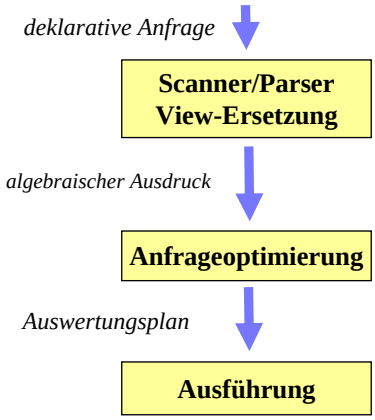
Wahlfreier vs. sequentieller Zugriff

- Beobachtung für große *n*:
  Ein sequentieller Zugriff auf *n* Datenblöcke ist in etwa *n*-mal schneller als *n* nacheinander ausgeführte wahlfreie Zugriffe auf die Datenblöcke
- Da Transferraten schneller wachsen als Zugriffsraten, verschlechtert sich das Verhältnis wahlfreier zu sequentieller Zugriff.
- Folgende Maßnahmen sind deshalb wichtig:
  - *Große Blöcke*: Die Wahl größerer Transfereinheiten verbessert das Verhältnis
  - *Clusterbildung der Daten*: Die Daten sollten von einer Indexstruktur so in Blöcken abgelegt werden, dass mit großen Blöcken in möglichst wenigen Zugriffen möglichst viele nützliche Daten übertragen werden

Inhalt

Aufgabe der Anfragebearbeitung

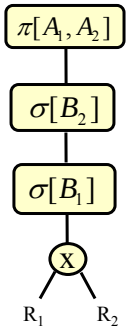
Übersetzung der *deklarativen* Anfrage in einen *effizienten, prozeduralen* Auswertungsplan



Kanonischer Auswertungsplan

SELECT A_1, A_2
FROM R_1, R_2
WHERE B_1 *AND* B_2

$$\pi_{A_1,A_2}(\sigma_{B_2}(\sigma_{B_1}(R_1 \times R_2)))$$



1. Bilde das kartesische Produkt der Relationen R_1, R_2
2. Führe Selektionen mit den Bedingungen B_1, B_2 durch.
3. Projiziere die Ergebnis-Tupel auf die erforderlichen Attribute A_1, A_2

Logische vs. physische Anfrageoptimierung

- Optimierungstechniken, die den Auswertungsplan umbauen (d.h. die Reihenfolge der Operatoren verändern), werden als **logische Anfrageoptimierung** bezeichnet.
- Unter **physischer Anfrageoptimierung** versteht man z.B. die Auswahl einer geeigneten Auswertungsstrategie für die Join-Operation oder die Entscheidung, ob für eine Selektionsoperation ein Index verwendet wird oder nicht und wenn ja, welcher (bei unterschiedlichen Alternativen). Es handelt sich also um die Auswahl eines geeigneten Algorithmus für jede Operation im Auswertungsplan.

Regel- vs. kostenbasierte Anfrageoptimierung (1)

- Es gibt zahlreiche Regeln (Heuristiken), um die Reihenfolge der Operatoren im Auswertungsplan zu modifizieren und so eine Performanz-Verbesserung zu erreichen, z.B.:
  - Push Selection: Führe Selektionen möglichst frühzeitig (vor Joins) aus
  - Elimination leerer Teilbäume
  - Erkennen gemeinsamer Teilbäume
- Optimierer, die sich ausschließlich nach diesen starren Regeln richten, nennt man **regelbasierte** oder auch **algebraische Optimierer**.

Regel- vs. kostenbasierte Anfrageoptimierung (2)

- Optimierer, die zusätzlich zu den starren Regeln die voraussichtliche Performanz der Auswertungspläne ermitteln und den leistungsfähigsten Plan auswählen, werden als **kostenbasierte** oder auch (irreführend) **nicht-algebraische Optimierer** bezeichnet.

- Die Vorgehensweise kostenbasierter Anfrageoptimierer ist meist folgende:
  - Generiere einen initialen Plan (z.B. Standardauswertungsplan)
  - Schätze die bei der Auswertung entstehenden Kosten
  - Modifiziere den aktuellen Plan gemäß vorgegebener Heuristiken
  - Wiederhole die Schritte 2 und 3 bis ein Stop-Kriterium erreicht ist
  - Gib den besten erhaltenen Plan aus

Inhalt

Äquivalenzregeln der Relationalen Algebra (1)

- 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

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

Äquivalenzregeln der Relationalen Algebra (2)

- 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

$$\pi_A(\sigma_B(R)) = \sigma_B(\pi_A(R))$$, falls $attr(B) \subseteq A$

Äquivalenzregeln der Relationalen Algebra (3)

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

$$\sigma_B(R \bowtie S) = \sigma_B(R) \bowtie S$$
$$\sigma_B(R \times S) = \sigma_B(R) \times S$$

, falls $attr(B) \subseteq attr(R)$

- Projektionen können teilweise in den Join verschoben werden

$$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A1}(R) \bowtie_B \pi_{A2}(S))$$

, 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

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

Äquivalenzregeln der Relationalen Algebra (4)

- Der Projektionsoperator kann mit der Vereinigung, aber nicht mit Schnitt oder Differenz vertauscht werden

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

- Eine 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: $\neg(B_1 \wedge B_2) = \neg B_1 \vee \neg B_2$

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

Beispiel (1)

Auto-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)

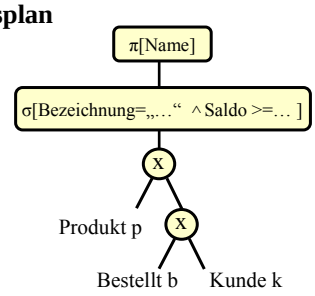
PNr	Bezeichnung	Anzahl	Preis
12	BMW 318i	10	40.000
4	Golf 5	40	25.000
330	Fiat Uno	5	18.000
98	Ferrari 380	1	180.000
14	Opel Corsa	14	17.000

Beispiel (2)

- **SQL-Anfrage**

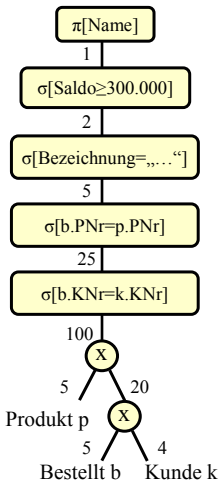
select	Name
from	Kunde k, Bestellt b, Produkt p
where	b.KNr = k.KNr
and	b.PNr = p.PNr
and	Bezeichnung = ‚Fiat Uno‘
and	Saldo ≥ 300.000

- **kanon. Auswertungsplan**

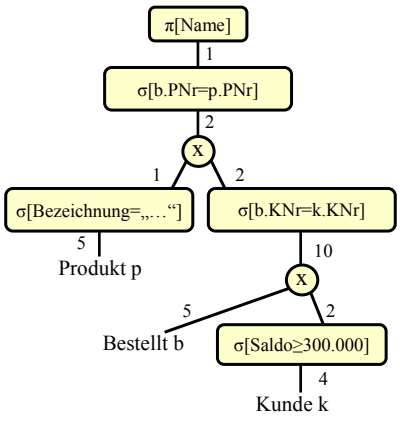


Beispiel (3)

- **Aufbrechen der Selektionen**

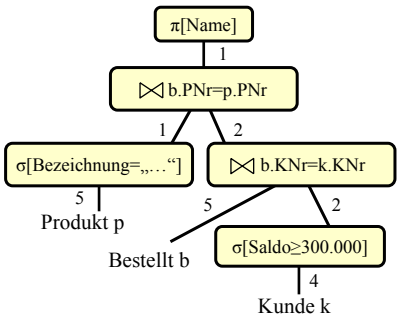


- **Verschieben der Selektionen**

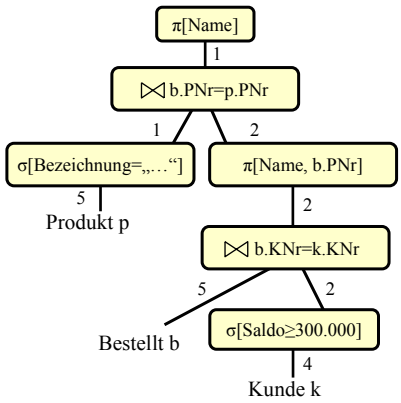


Beispiel (4)

- **Zusammenfassen zu Joins**



- **Einfügen zusätzlicher Selektionen**



Inhalt

Definition 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)

Schätzung der Selektivität (1)

- 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: $sel_B(\sigma_{\neg B1}) = 1 - sel_B(\sigma_{B1})$

Schätzung der Selektivität (2)

- I. a. benötigt man anspruchsvollere Methoden zur Selektivitäts-abschätzung.
- In der Literatur findet man 3 Arten von Schätzmethoden:
  - **Parametrisierte Verteilungen**
    Bestimme zu der vorhandenen Werteverteilung die Parameter einer Funktion so, dass die Verteilung möglichst gut angenähert wird.
  - **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
  - **Stichproben**
    Eine zufällige Menge von Tupeln einer Relation wird gezogen und als repräsentativ für die gesamte Relation betrachtet. Selektivitäten werden auf der Basis dieser Stichproben bestimmt (sehr einfache Methode).

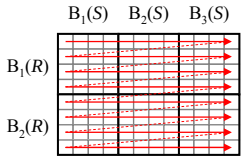
Inhalt

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**



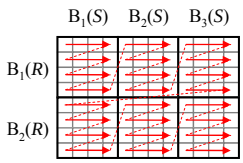
- Der einfache Nested-Loop-Join entspricht der Bildung des kartesischen Produktes in kanonischer Ordnung mit anschließender Selektion.
- Die Relation S wird $|R|$ mal eingelesen: Performanz ist deshalb inakzeptabel
- S wird als innere Relation und R als äußere Relation bezeichnet

Nested-Block-Loop-Join (1)

- **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**



Nested-Block-Loop-Join (2)

- **Beispiel**

S

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

R

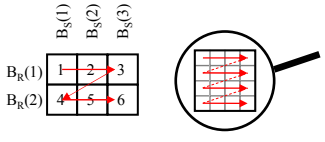
Gehaltsgruppe	Gehalt	
1	10.000	$B_R(1)$
2	20.000	
3	30.000	$B_R(2)$

- **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 NBL-Join (1)

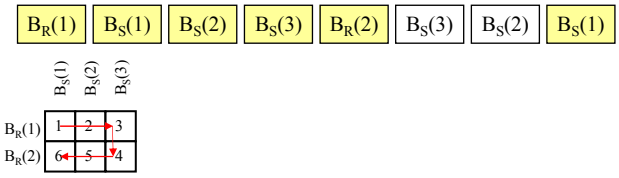
1. Seiten der inneren Relation im Cache halten
   - Cache wird überhaupt nicht ausgenutzt, wenn Cache kleiner als Relation S ist
   - Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R (: Zugriff Platte)

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



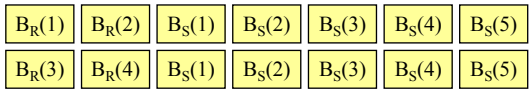
Cache Strategien für NBL-Join (2)

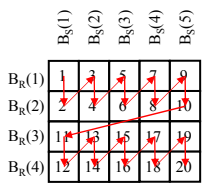
2. Seiten der inneren Relation im Cache, aber innere Relation jedes zweite mal rückwärts
   - Pro Durchlauf der äußeren Schleife werden ($|C|$-1) Blockzugriffe eingespart (ab 2. Durchlauf)
   - $|C|$ = Anzahl Blöcke, die in den Cache passen, ein Cache-Block wird jeweils für R-Relation benötigt
   - Blockzugriffe: $B_R + B_R \cdot (B_S - |C| + 1) + |C| - 1$
   - Beispiel: 2 Seiten Cache für S, 1 Seite Cache für R



Cache Strategien für NBL-Join (3)

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 NBL-Join (4)

– 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 (1)

- **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 von Cache für äußere/innere Relation?

Blockgrößen-Optimierung NBL-Join (2)

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)

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

Blockgrößen-Optimierung NBL-Join (3)

	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	
$t_{NL-Join} \approx$	$\left\lceil \frac{B_R}{\lvert C \rvert - 1} \right\rceil \cdot (2t_{seek} + t_{lat} + b \cdot (\lvert C \rvert - 1) \cdot t_{tr})$	+ $B_S \cdot \left\lceil \frac{B_R}{\lvert C \rvert - 1} \right\rceil \cdot (t_{lat} + b \cdot t_{tr})$
	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
t_{lat}: durchschnittliche Latenzzeit des Laufwerkes
b: Blockgröße (Parameter, der optimiert wird)

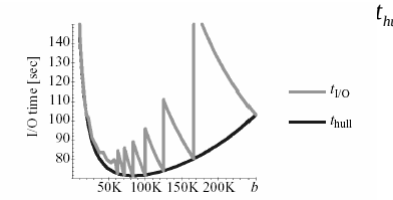
Blockgrößen-Optimierung NBL-Join (4)

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

Blockgrößen-Optimierung NBL-Join (5)

- 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 (6)

CPU-Kosten

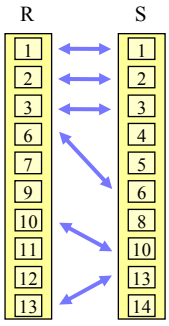
- Im wesentlichen müssen |*S*|*|*R*| Vergleiche durchgeführt werden
- Bei 0.1 µs pro Vergleich und 100.000 Tupel pro Relation ergibt sich eine Bearbeitungszeit von 1000 s.
- D.h. wesentlich mehr als die 75 s I/O-Zeit
- Der NLB-Join ist also *CPU-bound*
- Maßnahmen zur Senkung des CPU-Aufwands später

Sort-Merge-Join (1)

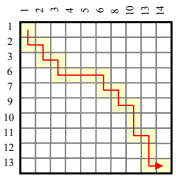
- **Zweistufiger Algorithmus**
  - 1. Schritt:

    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);
  while s.B = r.A
    j = j + 1;
    s = j-tes Tupel von S;
    result := result ∪ ((r − r.A) × s);
```



- **Matrixnotation**



Sort-Merge-Join (2)

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 (1)

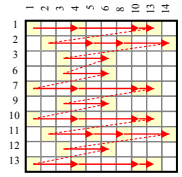
- **Reduktion des CPU-Aufwandes bei der Join-Berechnung**
  - Der Join-Partner eines *S*-Tupels wird gezielt mit Hilfe eines Hash-Verfahrens gesucht, anstatt das *S*-Tupel sequentiell mit jedem Tupel der Relation *R* zu vergleichen.
  - Zu diesem Zweck wird die Relation *R* gehasht, d.h. es wird zu allen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

Einfacher Hash-Join (2)

- **Algorithmus**

```
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

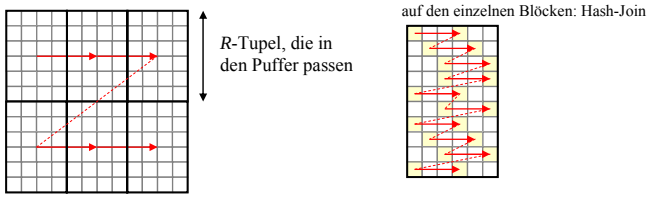
Hashed-Loop-Join (1)

- 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;
```

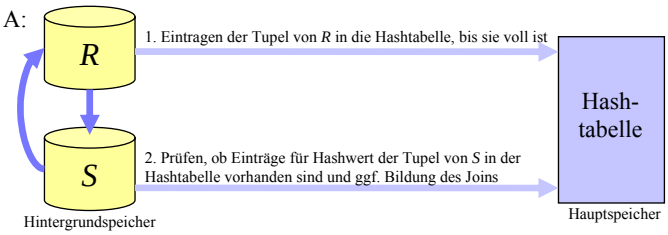
Hashed-Loop-Join (2)

- **Matrixnotation**



- **Ablauf**

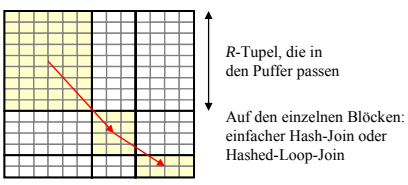
Schritt A:



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

Hash-Partitioned-Join (GRACE) (1)

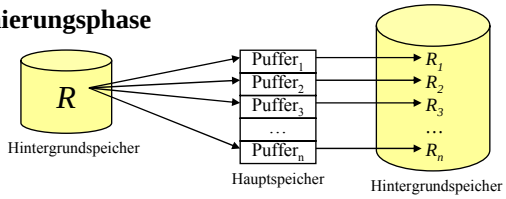
- Der Hashed-Loop-Join zerlegt die Relationen willkürlich in Blöcke, jeder Block der *R*-Relation muss mit jedem Block der *S*-Relation kombiniert werden
- Idee: Zerlege die Relationen *R* und *S* mit Hilfe einer Hashfunktion in Partitionen, so dass nur Partitionen mit demselben Hash-Key kombiniert werden 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 einfachen Hash-Join oder einem Hashed-Loop-Join (wenn Partition zu groß)
- **Matrixnotation**



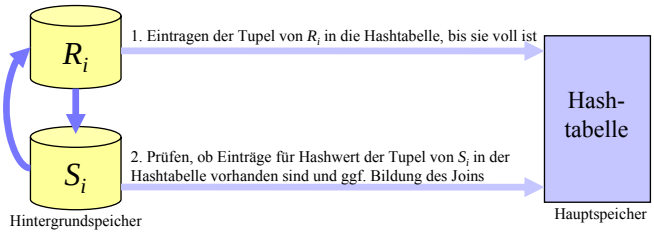
Hash-Partitioned-Join (GRACE) (2)

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

    Schritt A:



    Schritt B: Wiederhole Schritt A für S
  - **Join-Phase**



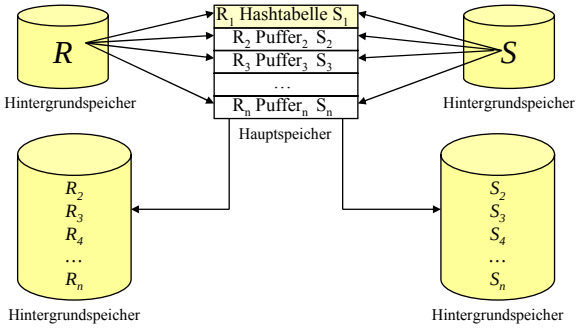
Hybrid Hash-Join (1)

- **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
```

Hybrid Hash-Join (2)

- **Ablauf der Partitionierungsphase**



Hybrid Hash-Join (3)

- **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 können die Partitionen der einzelnen Verfahren gewählt werden?