
Kapitel 5

Anfragebearbeitung

Vorlesung: PD Dr. Peer Kröger

Dieses Skript basiert auf den Skripten zur Vorlesung Datenbanksysteme II an der LMU München von Prof. Dr. Christian Böhm (Sommersemester 2007), PD Dr. Peer Kröger (Sommersemester 2008) und PD Dr. Matthias Schubert (Sommersemester 2009)

http://www.dbs.ifi.lmu.de/cms/Datenbanksysteme_II_12

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 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

3

5.1 Einleitung

HW-Grundlagen

- Von-Neumann Rechner Architektur: Flaschenhalse



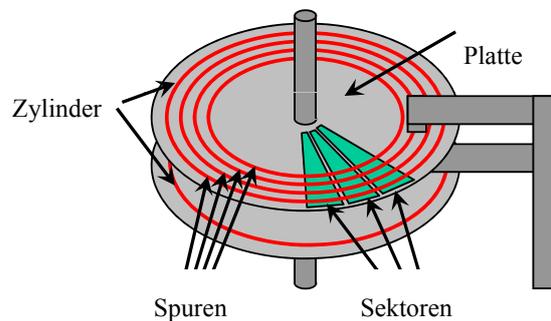
- Zur Vereinfachung unterscheidet man meist nur zwischen
 - **CPU-bound**
CPU, Arbeitsspeicher und Bus bilden den Hauptengpass
 - **I/O-bound**
Hintergrundspeicher und I/O bilden den Hauptengpass

4

5.1 Einleitung

HW-Grundlagen (cont.)

- Schematischer Aufbau einer Festplatte



- 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

5

5.1 Einleitung

HW-Grundlagen (cont.)

- 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)

6

5.1 Einleitung

Speichermedien

- I/O-Engpass auch mit modernen Platten nicht überwindbar
- Lösungsansatz: Verwende statt einer großen Festplatte mehrere kleine, die parallel betrieben werden können

=> RAID-Systeme

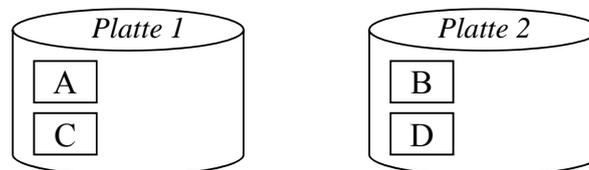
- Die Komplexität wird durch den RAID-Controller nach Außen verborgen => es gibt nur ein (virtuelles) Laufwerk
- Acht verschiedene RAID-Level die unterschiedliche Zugriffsprofile optimieren

7

5.1 Einleitung

Speichermedien (cont.)

- RAID 0
 - Datenmenge wird durch blockweise Rotation auf die Platten verteilt (**Striping**)
 - Beispiel: Striping von 4 Blöcken (A,B,C,D) auf zwei Platten



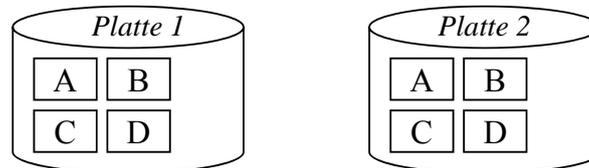
- Größtmögliche Beschleunigung: Anfrage an aufeinanderfolgende Blöcke kann parallel bearbeitet werden
- Fehleranfällig:
 - besteht eine Datei aus vielen Blöcken werden diese über die entsprechenden Platten verteilt
 - Ausfall einer Platte führt zur Beschädigung der Datei

8

5.1 Einleitung

Speichermedien (cont.)

- RAID 1
 - Jedes Laufwerk besitzt eine Spiegelkopie (**Mirror**)
 - Durch Redundanz ist Fehlerfall eines Laufwerks kein Problem
 - Beispiel:



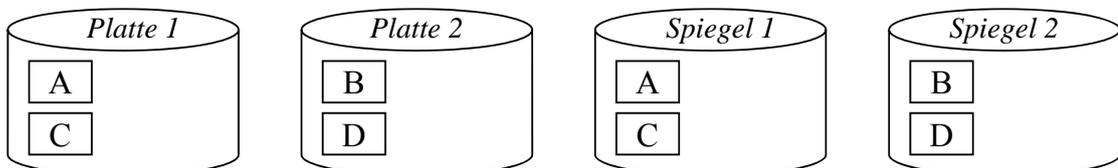
- Leseoperationen parallelisierbar (wie RAID 0)
- Schreiboperationen müssen auf beiden Mirrors (parallel) durchgeführt werden

9

5.1 Einleitung

Speichermedien (cont.)

- RAID 0+1
 - Kombination aus RAID 0 und RAID 1
 - Verteilung der Datenblöcke wie bei RAID 0
 - Spiegelung der Platten wie bei RAID 1



- Vereinigt Vorteile von RAID 0 und RAID 1
- ABER: Anzahl der benötigten Platten steigt!!!

10

5.1 Einleitung

Speichermedien (cont.)

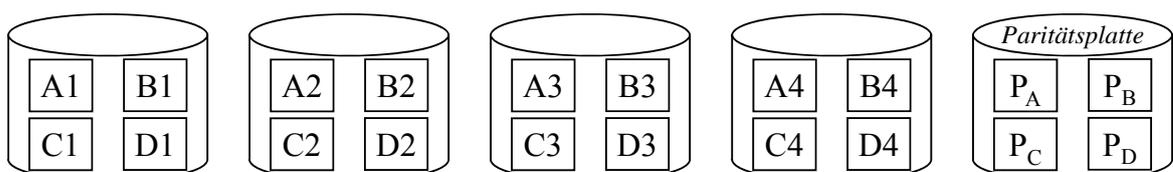
- Ab RAID 2 wird Datensicherheit ökonomisch günstiger umgesetzt
- Hilfsmittel: Paritätsinformationen
 - Prüfsumme für mehrere Daten
 - Verwendbar, um Daten auf Korrektheit zu überprüfen
 - Verwendbar, um Daten im Fehlerfall zu rekonstruieren
 - Vorgehen:
 - Speichere zu N Datenbereichen auf unterschiedlichen Platten zusätzlich deren Prüfsumme auf einer anderen Platte
 - Ist einer der N Datenbereiche defekt kann dieser aus der Prüfsumme und den $N-1$ übrigen (intakten) Datenbereichen wiederhergestellt werden

11

5.1 Einleitung

Speichermedien (cont.)

- RAID 2
 - Striping auf Bitebene
 - Paritätsinformationen auf separaten Platten
 - In der Praxis meist nicht eingesetzt
- RAID 3 und RAID 4
 - Striping auf Bit- oder Byte-Ebene (RAID 3) bzw. blockweise (RAID 4)
 - Paritätsinformationen auf einer speziellen Platte



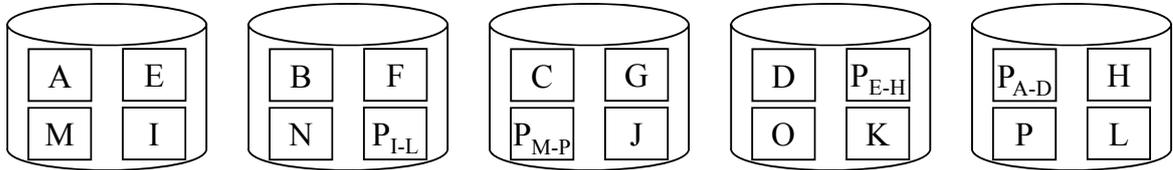
- Nachteil: jede Schreiboperation muss auf Paritätsplatte zugreifen

12

5.1 Einleitung

Speichermedien (cont.)

- RAID 5
 - Striping blockweise (wie RAID 4)
 - Verteilung der Paritätsinformationen auf alle Platten



- Damit ist der Flaschenhals der Paritätsplatte beseitigt
- Schreibeoperationen setzt aber nach wie vor die Neuberechnung und das Ablegen des neuen Paritätsblocks voraus
- ACHTUNG: Paritätsblock P_x kann nur ein Fehler in den Daten X korrigieren!

13

5.1 Einleitung

Speichermedien (cont.)

- Abschließende Bemerkungen
 - Wahl des RAID-Levels hängt vom Anwendungsprofil ab (z.B. Anteil der Leseoperationen im Vergleich zu Schreiboperationen)
 - Kommerzielle RAID-System erlauben typischerweise eine flexible Konfiguration
 - Viele DBS unterstützen Striping von Tupeln auf unterschiedliche Platten auch ohne Einsatz von RAID-Systemen
 - Trotz der Fehlertoleranz von RAID-Systemen ist der Einsatz von Recovery-Techniken (Kapitel 4) wichtig

14

5.1 Einleitung

Anfragebearbeitung

- Zu bearbeitende Seiten müssen vom HGSP in den DB-Puffer geladen werden
- Problem: Verwaltung der Daten auf einem Speichermedium sequentiell
 - Zeitaufwand für Bearbeitung einer Suchanfrage: $O(n)$
(im ungünstigsten Fall alle n Datensätze durchsuchen)
- 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.

15

5 Anfragebearbeitung

Übersicht

5.1 Einleitung

5.2 Indexstrukturen

5.3 Grundlagen der Anfrageoptimierung

5.4 Logische Anfrageoptimierung

5.5 Kostenmodellbasierte Anfrageoptimierung

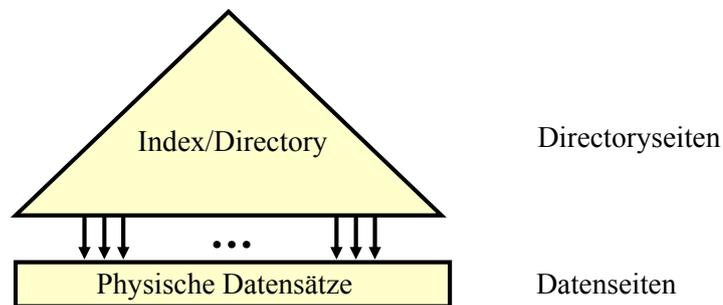
5.6 Implementierung der Joinoperation

16

5.2 Indexstrukturen

Aufbau baumartiger Indexstrukturen

- 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.

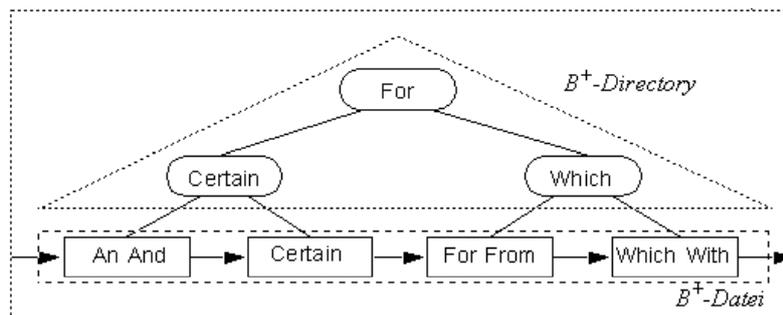


17

5.2 Indexstrukturen

Beispiel B+-Baum

- Erweiterung des B-Baums (vgl. Vorlesungen Effiziente Algorithmen, Index- und Speicherstrukturen)
 - Datenelemente nur in den Blattknoten speichern
 - Innere Knoten enthalten lediglich Schlüssel
- Knoten entsprechen Seiten auf der Platte
- B⁺-Baum für die Zeichenketten:
An, And, Certain, For, From, Which, With



18

5.2 Indexstrukturen

Modellierung der I/O-Kosten

- 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

19

5.2 Indexstrukturen

Modellierung der I/O-Kosten (cont.)

- **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.

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

- 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:

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

20

5.2 Indexstrukturen

Modellierung der I/O-Kosten (cont.)

- **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).

21

5.2 Indexstrukturen

Wahlfreier vs. sequentieller Zugriff

- Ein sequentieller Zugriff auf n Datenblöcke ist in etwa n -mal schneller als n nacheinander ausgeführte wahlfreie Zugriffe auf die Datenblöcke (für große n)
- Transferraten wachsen schneller als Zugriffsraten
=> Verhältnis wahlfreier zu sequentieller Zugriff verschlechtert sich
- 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

22

5 Anfragebearbeitung

Übersicht

5.1 Einleitung

5.2 Indexstrukturen

5.3 Grundlagen der Anfrageoptimierung

5.4 Logische Anfrageoptimierung

5.5 Kostenmodellbasierte Anfrageoptimierung

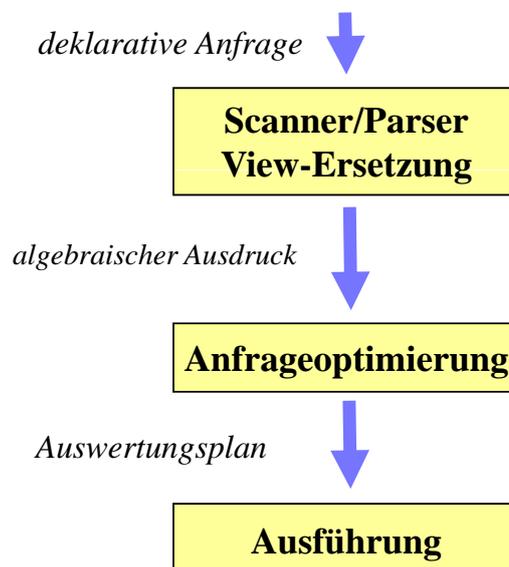
5.6 Implementierung der Joinoperation

23

5.3 Grundlagen der Anfrageoptimierung

Aufgabe der Anfragebearbeitung

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



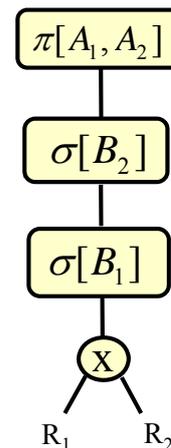
24

5.3 Grundlagen der Anfrageoptimierung

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

25

5.3 Grundlagen der Anfrageoptimierung

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.

26

5.3 Grundlagen der Anfrageoptimierung

Regel- vs. kostenbasierte Anfrageoptimierung

- 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**.

27

5.3 Grundlagen der Anfrageoptimierung

Regel-/kostenbasierte Optimierung (cont.)

- 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

28

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

29

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

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

30

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

$$\sigma_{B_1 \wedge B_2 \wedge \dots \wedge B_n}(R) = \sigma_{B_1}(\sigma_{B_2}(\dots(\sigma_{B_n}(R))\dots))$$

- Geschachtelte Projektionen können eliminiert werden

$$\pi_{A_1}(\pi_{A_2}(\dots(\pi_{A_n}(R))\dots)) = \pi_{A_1}(R)$$

Damit eine solche Schachtelung sinnvoll ist, muss gelten: $A_1 \subseteq A_2 \subseteq \dots \subseteq A_n$

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

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

31

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

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

- Projektionen können teilweise in den Join verschoben werden

$$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A_1}(R) \bowtie_B \pi_{A_2}(S)), \text{ falls } \begin{aligned} A_1 &= \text{attr}(R) \cap (A \cup \text{attr}(B)) \\ A_2 &= \text{attr}(S) \cap (A \cup \text{attr}(B)) \end{aligned}$$

- Selektionen können mit Vereinigung, Schnitt und Differenz vertauscht werden

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

32

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.

$$\neg(B_1 \wedge B_2) = \neg B_1 \vee \neg B_2$$

33

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

34

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, 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

35

5.4 Logische Anfrageoptimierung

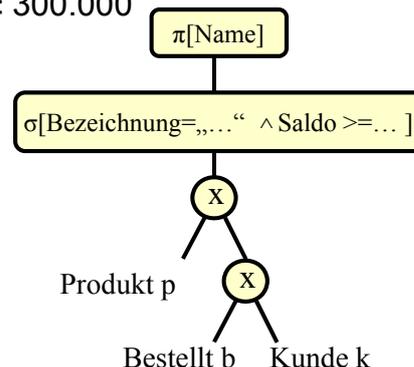
Beispiel (cont.)

- 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:

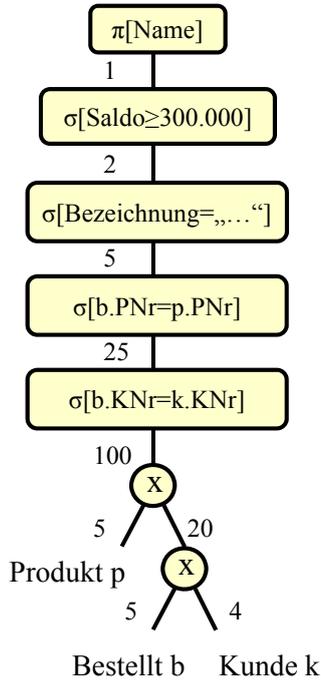


36

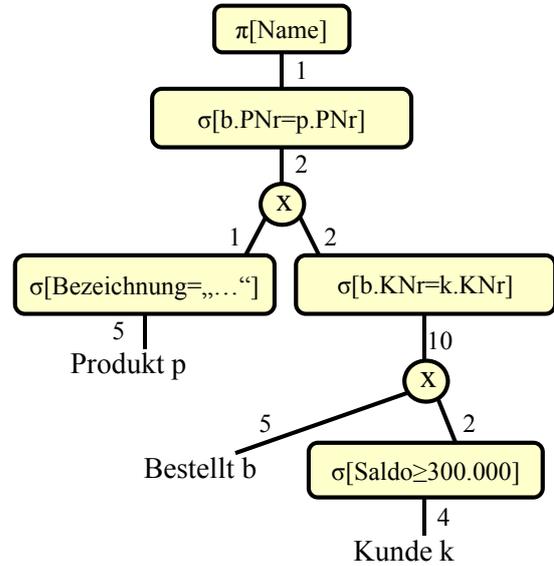
5.4 Logische Anfrageoptimierung

Beispiel (cont.)

- Aufbrechen der Selektionen



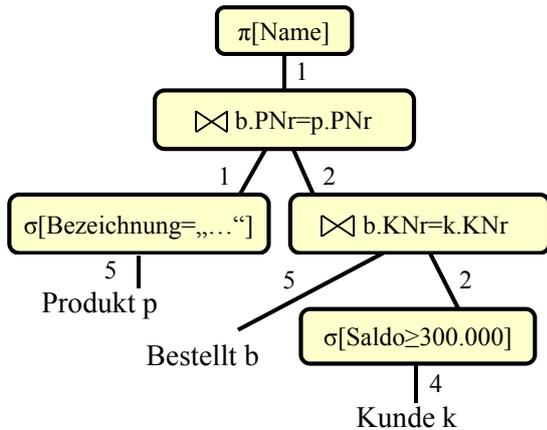
- Verschieben der Selektionen



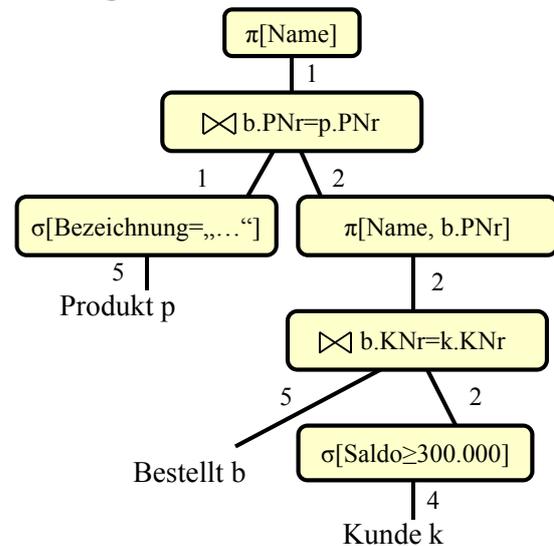
5.4 Logische Anfrageoptimierung

Beispiel (cont.)

- Zusammenfassen zu Joins



- Einfügen zusätzlicher Selektionen



5 Anfragebearbeitung

Übersicht

5.1 Einleitung

5.2 Indexstrukturen

5.3 Grundlagen der Anfrageoptimierung

5.4 Logische Anfrageoptimierung

5.5 Kostenmodellbasierte Anfrageoptimierung

5.6 Implementierung der Joinoperation

39

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)

40

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 Schlüsseligenschaft, 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: $sel_B(\sigma_{B_1 \wedge B_2}) = sel_B(\sigma_{B_1}) \cdot sel_B(\sigma_{B_2})$
 - logisches ODER: $sel_B(\sigma_{B_1 \vee B_2}) = sel_B(\sigma_{B_1}) + sel_B(\sigma_{B_2}) - sel_B(\sigma_{B_1}) \cdot sel_B(\sigma_{B_2})$
 - logisches NICHT: $sel_B(\sigma_{\neg B_1}) = 1 - sel_B(\sigma_{B_1})$

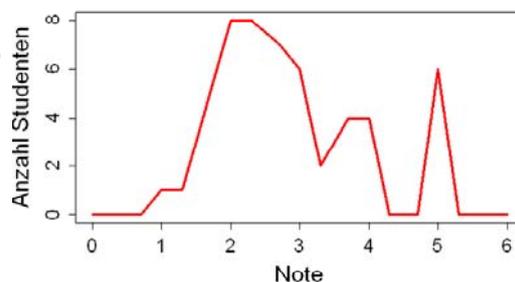
41

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 Schätzmethode:
 - Parametrische Verteilungen
 - Histogramme
 - Stichproben

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

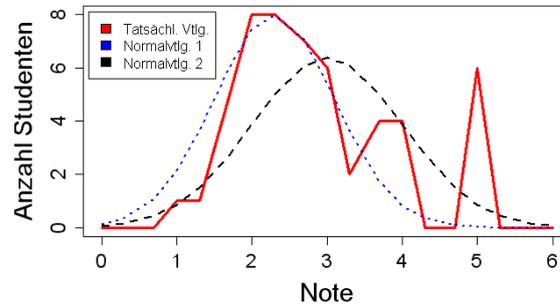


42

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:

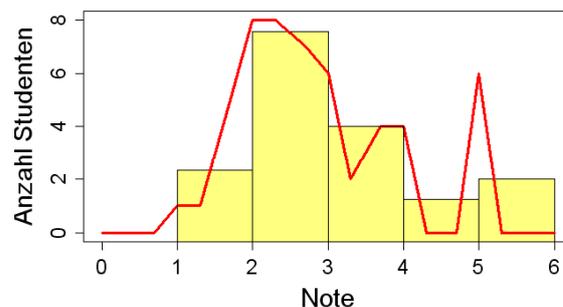
Wahl des Verteilungstyps (Normalverteilung, Exponentialverteilung...)
und Wahl der Parameter, besonders bei mehrdimensionalen Anfragen
(also z.B. bei Selektionen, die sich auf mehrere Attribute beziehen)

43

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

44

5.5 Kostenmodellbasierte Anfrageoptimierung

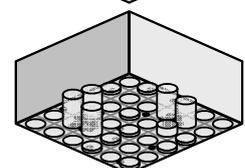
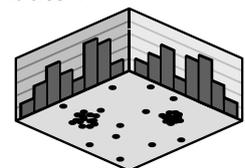
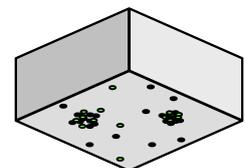
Selektivität: Stichproben

- 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

45

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



46

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

47

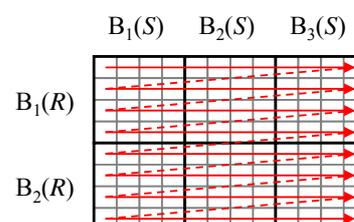
5.6 Implementierung der Joinoperation

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

48

5.6 Implementierung der Joinoperation

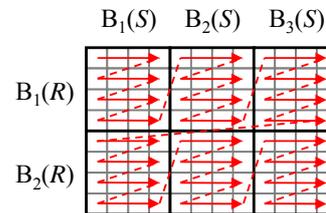
Nested-Block-Loop-Join

– 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



49

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)

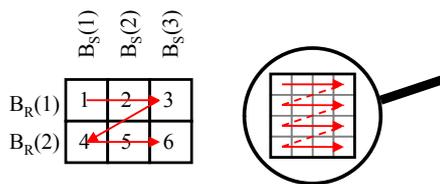
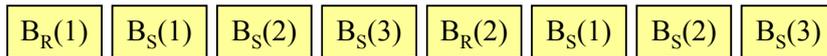
– D.h. die kleinere Relation sollte die äußere sein

50

5.6 Implementierung der Joinoperation

Cache Strategien für Nested-Block-Loop-Join

- 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)

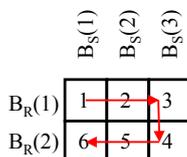
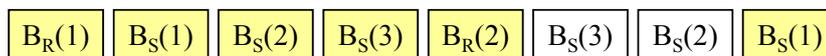


51

5.6 Implementierung der Joinoperation

Cache Strategien für NBL-Join (cont.)

- 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



52

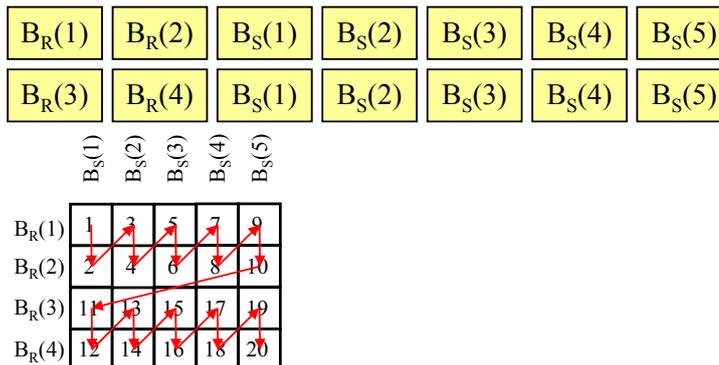
5.6 Implementierung der Joinoperation

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



53

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

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

54

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?

55

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-Laufwerks
- b : Blockgröße (Parameter, der optimiert wird)

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

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

56

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>	$\left(\left\lceil \frac{f_S}{b} \right\rceil \cdot \left\lceil \frac{f_R/b}{c/b-1} \right\rceil \right) \cdot (t_{lat} + b \cdot t_{tr})$

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

57

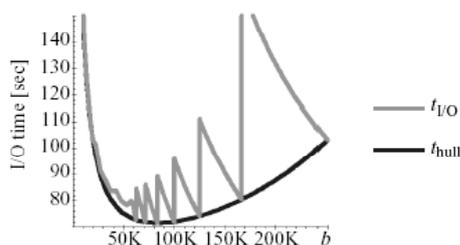
5.6 Implementierung der Joinoperation

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

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

- 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

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

58

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:

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

59

5.6 Implementierung der Joinoperation

Blockgrößen-Optimierung NBL-Join (cont.)

CPU-Kosten

- Im wesentlichen müssen $|S| \cdot |R|$ Vergleiche durchgeführt werden
- Bei 0.1 μ 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

60

5.6 Implementierung der Joinoperation

Sort-Merge-Join

– Zweistufiger Algorithmus

– 1. Schritt:

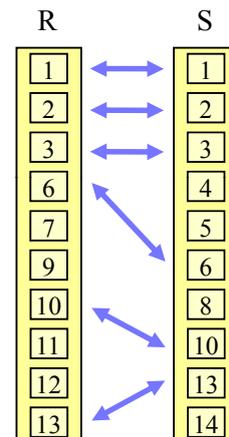
sortiere R bzgl. Attribut A
 sortiere S bzgl. Attribut B

– 2. Schritt:

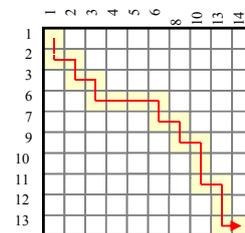
```

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



61

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

62

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)

63

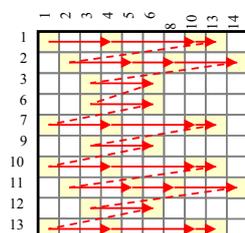
5.6 Implementierung der Joinoperation

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



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

64

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**

```

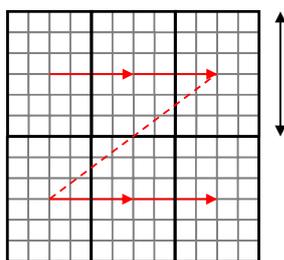
repeat
  lese soviel Tupel von  $R$  in Hauptspeic her bis der Platz aufgebrauc ht ist;
  erzeuge für diese Tupel eine Hashtabell e  $HT$ ;
  for each Tupel  $s \in S$  do
    berechne  $adr = hash(s)$ ;
    for each Tupel  $r \in HT[adr]$  do
      if  $r.A = s.B$  then
         $result := result \cup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

65

5.6 Implementierung der Joinoperation

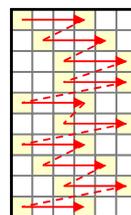
Hashed-Loop-Join (cont.)

- **Matrixnotation**



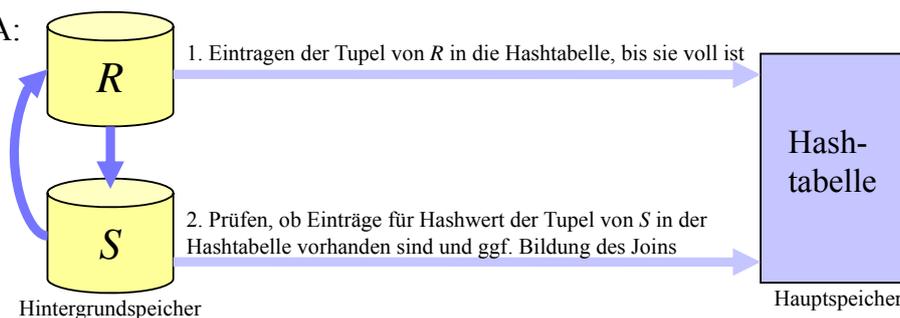
R -Tupel, die in den Puffer passen

auf den einzelnen Blöcken: Hash-Join



- **Ablauf**

Schritt A:



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

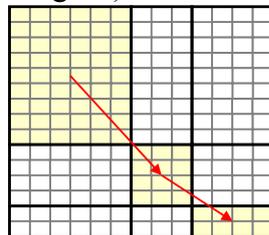
66

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

67

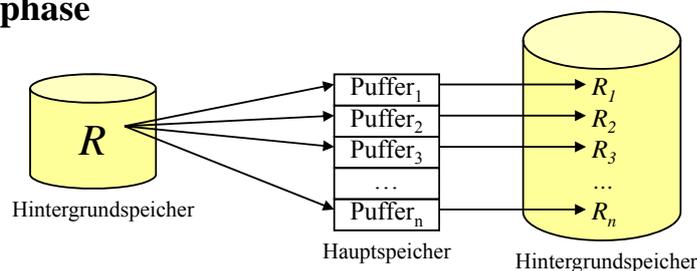
5.6 Implementierung der Joinoperation

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

• Ablauf

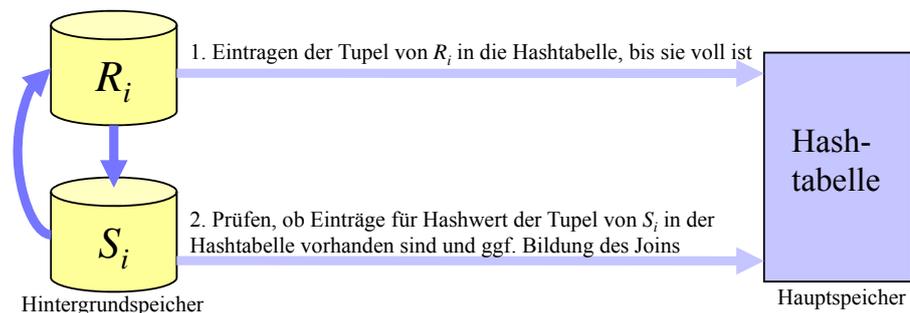
– Partitionierungsphase

Schritt A:



Schritt B: Wiederhole Schritt A für S

– Join-Phase



68

5.6 Implementierung der Joinoperation

Hybrid Hash-Join

- **Algorithmus**

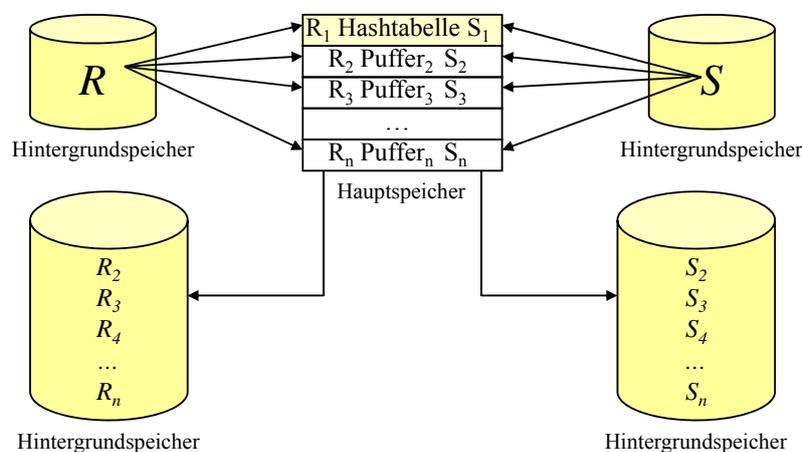
```
for each Tupel  $r \in R$  do
  berechne  $adr = hash(r)$ ;
  if ( $adr = 1$ ) then
    füge  $r$  in eine Hashtabelle  $HT$  ein (bzgl. neuer Hashfkt.);
  else
    speichere  $r$  in einem Puffer  $BR_{adr}$ 
    /* wenn der Puffer voll ist, wird er stets auf Platte geschrieben */
for each Tupel  $s \in S$  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
```

69

5.6 Implementierung der Joinoperation

Hybrid Hash-Join (cont.)

Ablauf der Partitionierungsphase:



70

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?