Kapitel 5
Anfragebearbeitung

Skript zur Vorlesung: Datenbanksysteme II
Sommersemester 2008, LMU München

Dieses Skript basiert zu einem Teil auf dem Skript zur Vorlesung Datenbanksysteme II von Prof. Dr. Christian Böhm gehalten im Sommersemester 2007 an der LMU München

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

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

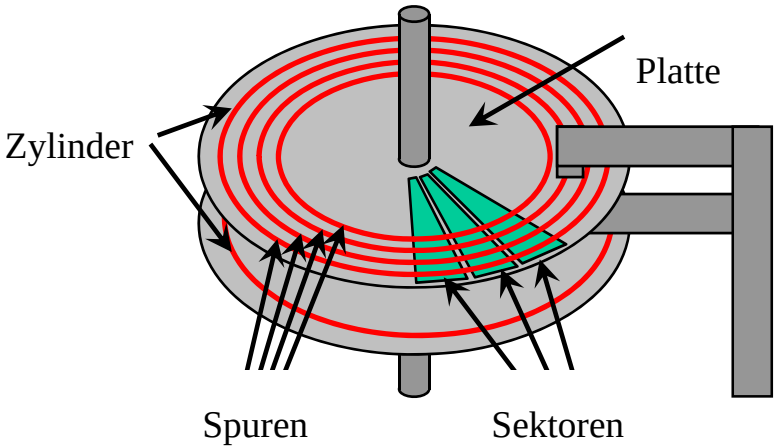
HW-Grundlagen

– Von-Neumann Rechner Architektur: Flaschenhälse

– 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

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

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)

Speichermedien

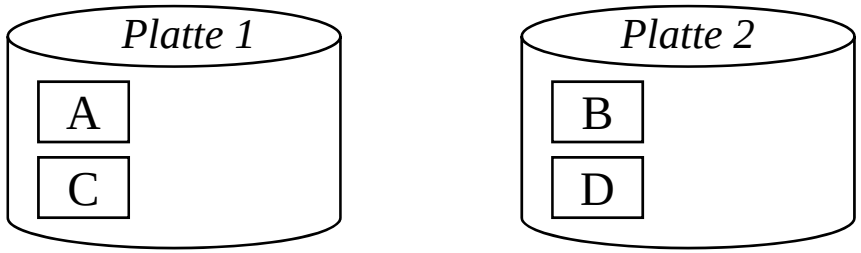
- I/O-Engpass auch mit modernen Platten nicht überwindbar
- Lösungsansatz: Verwende statt einer großen Festplatte mehrere kleine, die parallel betrieben werden können

=> 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

Speichermedien (cont.)

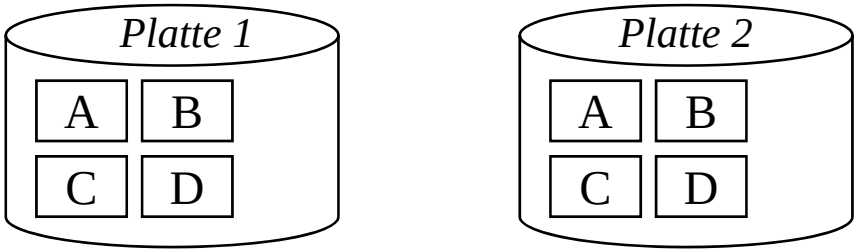
- RAID 0
  - Datenmenge wird durch blockweise Rotation auf die Platten verteilt (***Striping***)
  - Beispiel: Striping von 4 Blöcken (A,B,C,D) auf zwei Platten



  - 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

Speichermedien (cont.)

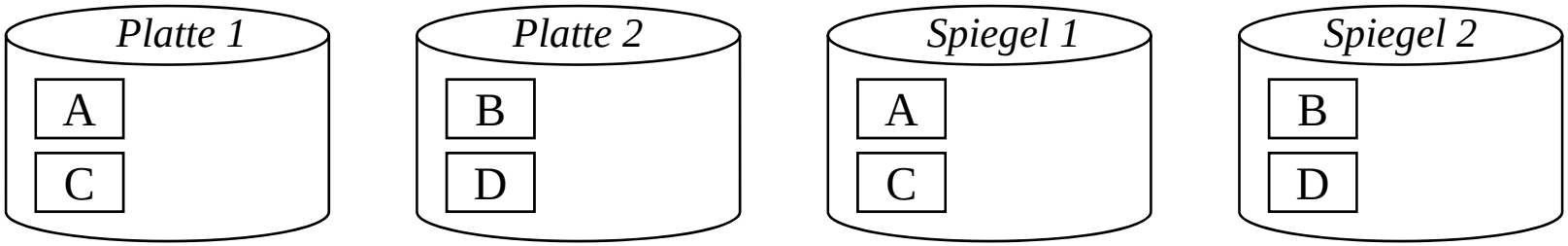
– RAID 1
  - Jedes Laufwerk besitzt eine Spiegelkopie (***Mirror***)
  - Durch Redundanz ist Fehlerfall eines Laufwerks kein Problem
  - Beispiel:



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

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



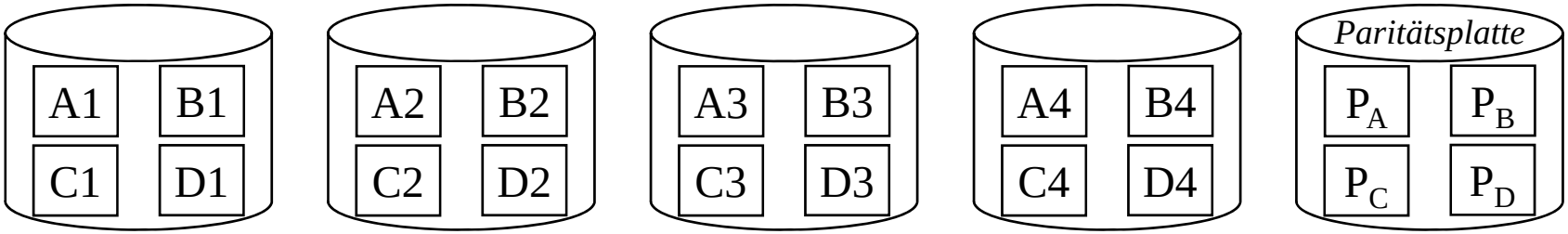
  - Vereinigt Vorteile von RAID 0 und RAID 1
  - ABER: Anzahl der benötigten Platten steigt!!!

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

Speichermedien (cont.)

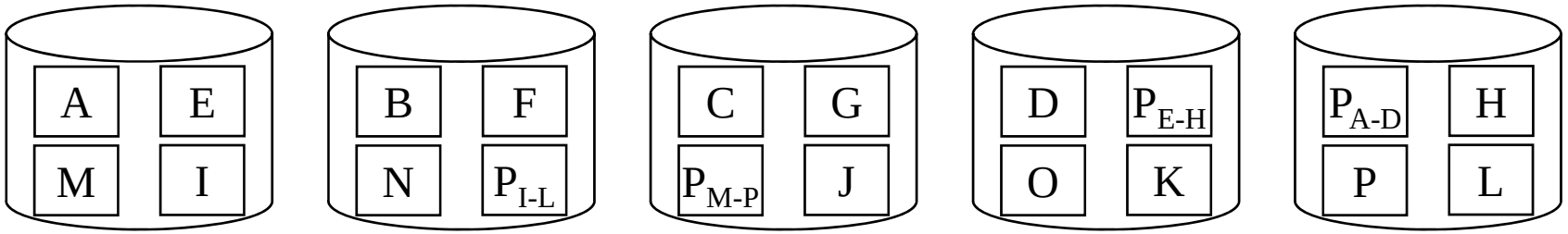
- RAID 2
  - Striping auf Bitebene
  - Paritätsinformationen auf separaten Platten
  - In der Praxis meist nicht eingesetzt
- RAID 3 und RAID 4
  - Striping auf Bit- oder Byte-Ebene (RAID 3) bzw. blockweise (RAID 4)
  - Paritätsinformationen auf einer speziellen Platte



  - Nachteil: jede Schreiboperation muss auf Paritätsplatte zugreifen

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!

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

Der DB-Puffer

- Operationen auf den Daten im Hauptsp. (DB-Puffer)
  - Seiten, die bearbeitet werden, müssen von der Platte in den Puffer übertragen werden
    - Puffergröße begrenzt => Seiten müssen verdrängt werden
    - Puffer-Verdrängungsstrategien (Steal/No Steal) und Einbringungsstrategien (update-in-place) => Kapitel 4
    - Ersetzungsstrategie, Prinzip der Lokalität =>Prozessverwaltung in BS
  - Schematisch:

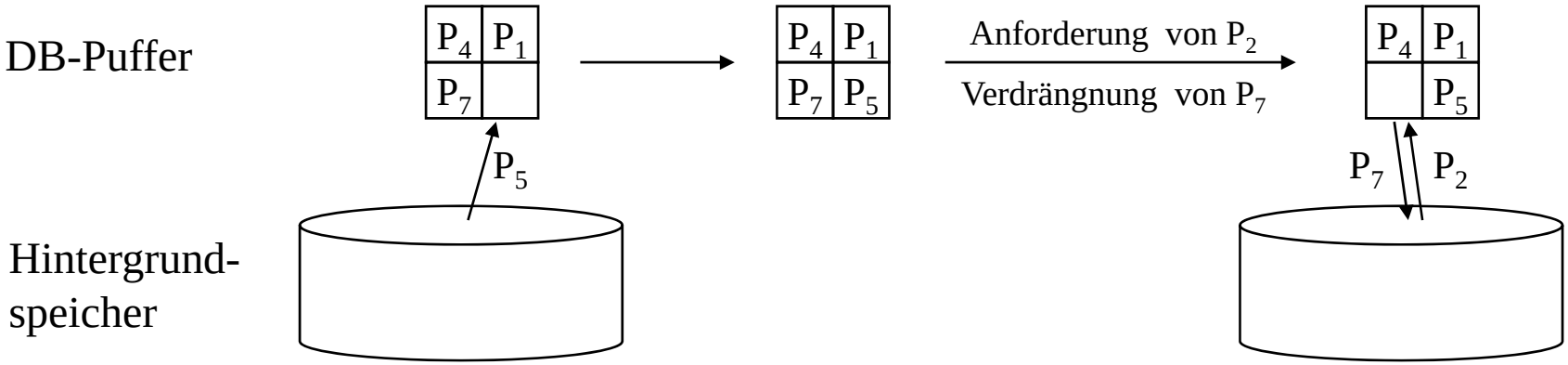


Abbildung von Relationen auf den HGSP

- Typischerweise umfasst eine Relation mehrere Seiten
- Diese werden zu einer Datei zusammengefasst
- Tupel nicht über mehrere Seiten verteilen, sonst:
  - Geschwindigkeitsverlust beim Zugriff (WARUM?)
  - Synchronisation und Recovery aufwändiger (WARUM?)
- Jede Seite enthält interne Datensatztabelle
- Tupel-Identifikator (TID): Seitennummer+Tabelleneintrag

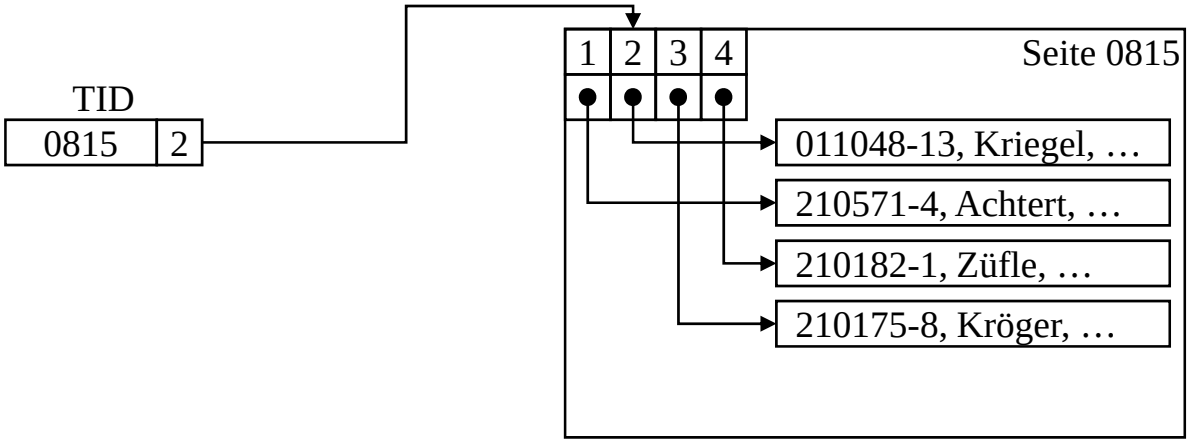
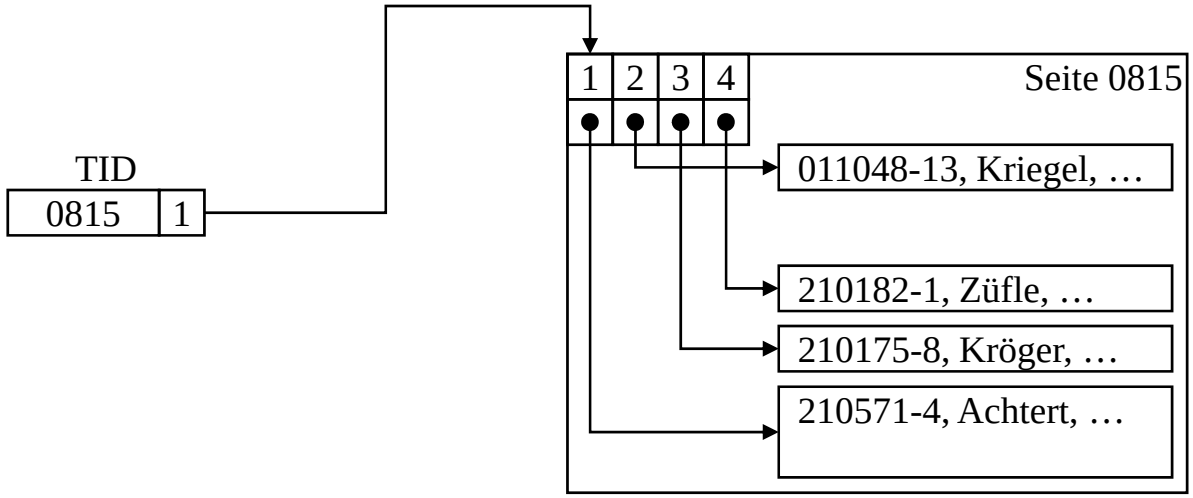


Abbildung von Relationen auf den HGSP (cont.)

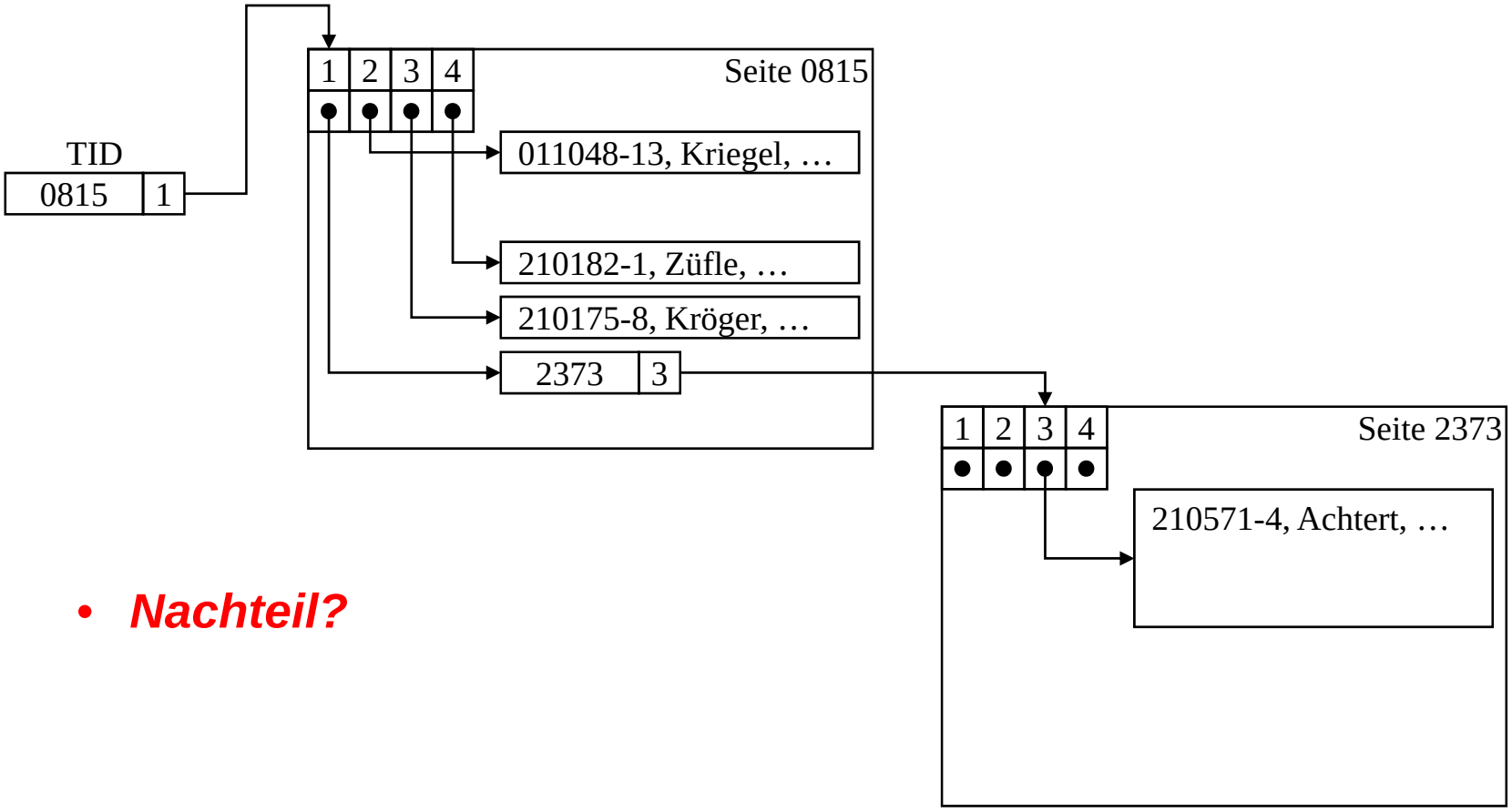
– Interne Reorganisation (z.B. Tupel von Achtert vergrößert sich)



- Dank der Seiteninternen Tabelle bleiben alle Verweise (insbesondere auch die TID) gültig

Abbildung von Relationen auf den HGSP (cont.)

– Verdrängung eines Tupels von einer Seite (z.B. Tupel von Achtert vergrößert sich weiter)



- ***Nachteil?***

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.

Übersicht

5.1 Einleitung

5.2 Indexstrukturen

5.3 Grundlagen der Anfrageoptimierung

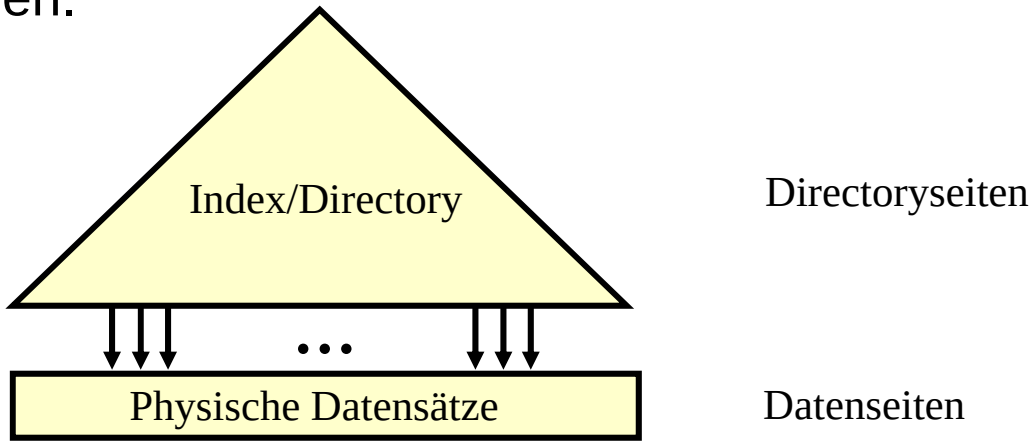
5.4 Logische Anfrageoptimierung

5.5 Kostenmodellbasierte Anfrageoptimierung

5.6 Implementierung der Joinoperation

Aufbau baumartiger Indexstrukturen

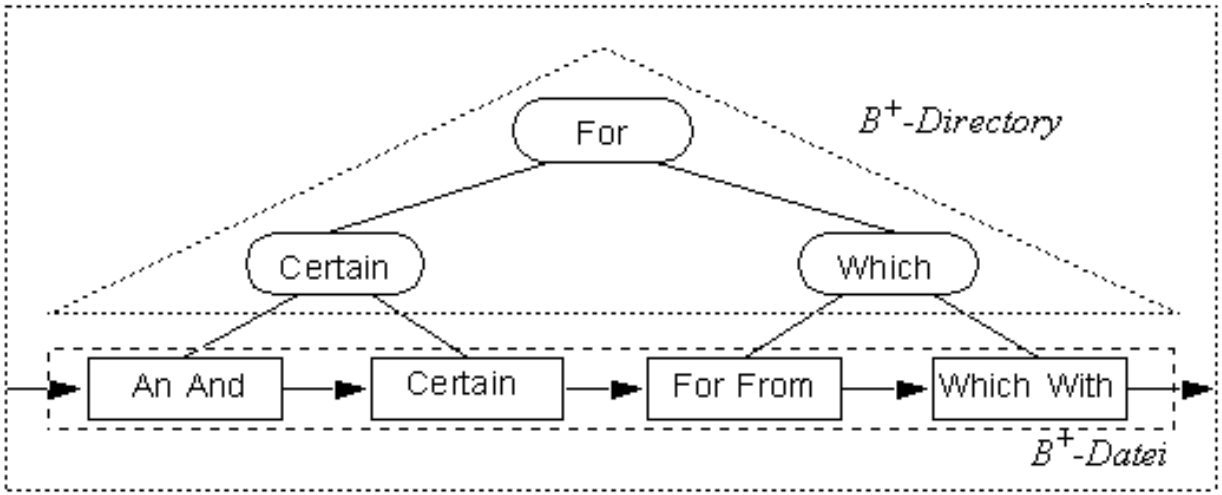
- Baumartige Indexstrukturen bestehen üblicherweise aus Directory- und Datenseiten:
  - Die eigentlichen physischen Datensätze werden in den **Datenseiten** gespeichert (Blattknoten des Baums).
  - Die **Directoryseiten** sind die inneren Knoten des Baums und speichern die Directory-Einträge, die aus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



Beispiel B+-Baum

- Erweiterung des B-Baums (vgl. Vorlesungen Effiziente Algorithmen, Index- und Speicherstrukturen)
  - Datenelemente nur in den Blattknoten speicheren
  - Innere Knoten enthalten lediglich Schlüssel
- B^+-Baum für die Zeichenketten:

  An, And, Certain, For, From, Which, With



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

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.

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

- 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 (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).

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

Übersicht

5.1 Einleitung

5.2 Indexstrukturen

5.3 Grundlagen der Anfrageoptimierung

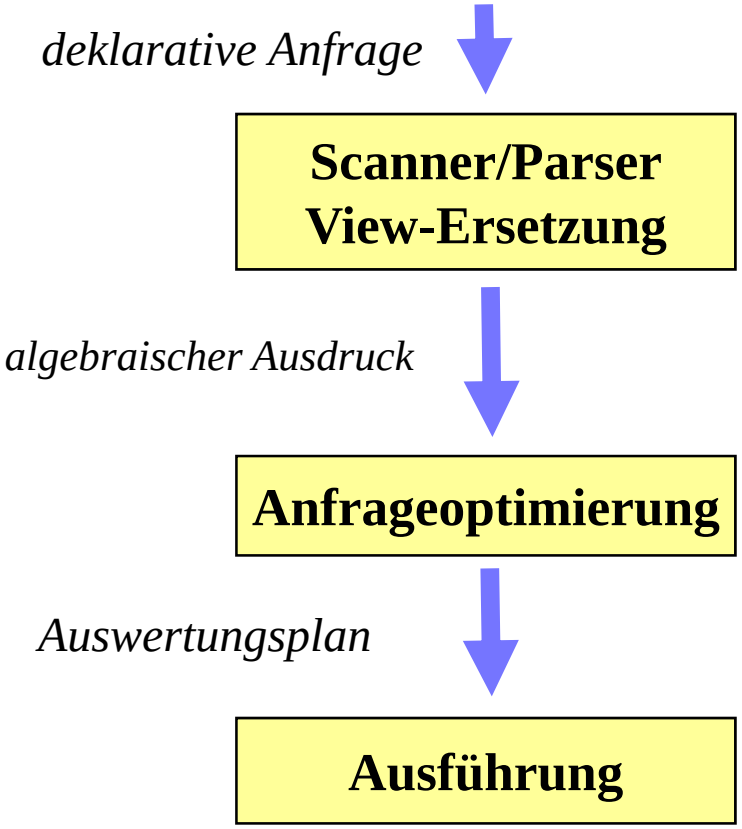
5.4 Logische Anfrageoptimierung

5.5 Kostenmodellbasierte Anfrageoptimierung

5.6 Implementierung der Joinoperation

Aufgabe der Anfragebearbeitung

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



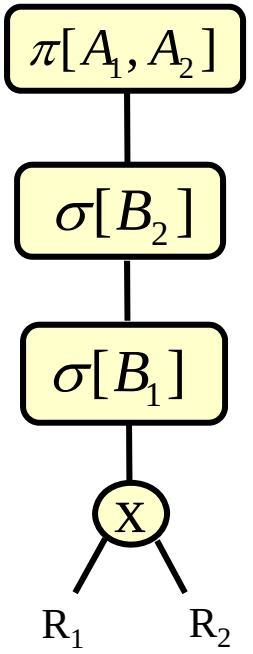
Kanonischer Auswertungsplan

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

– 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-/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