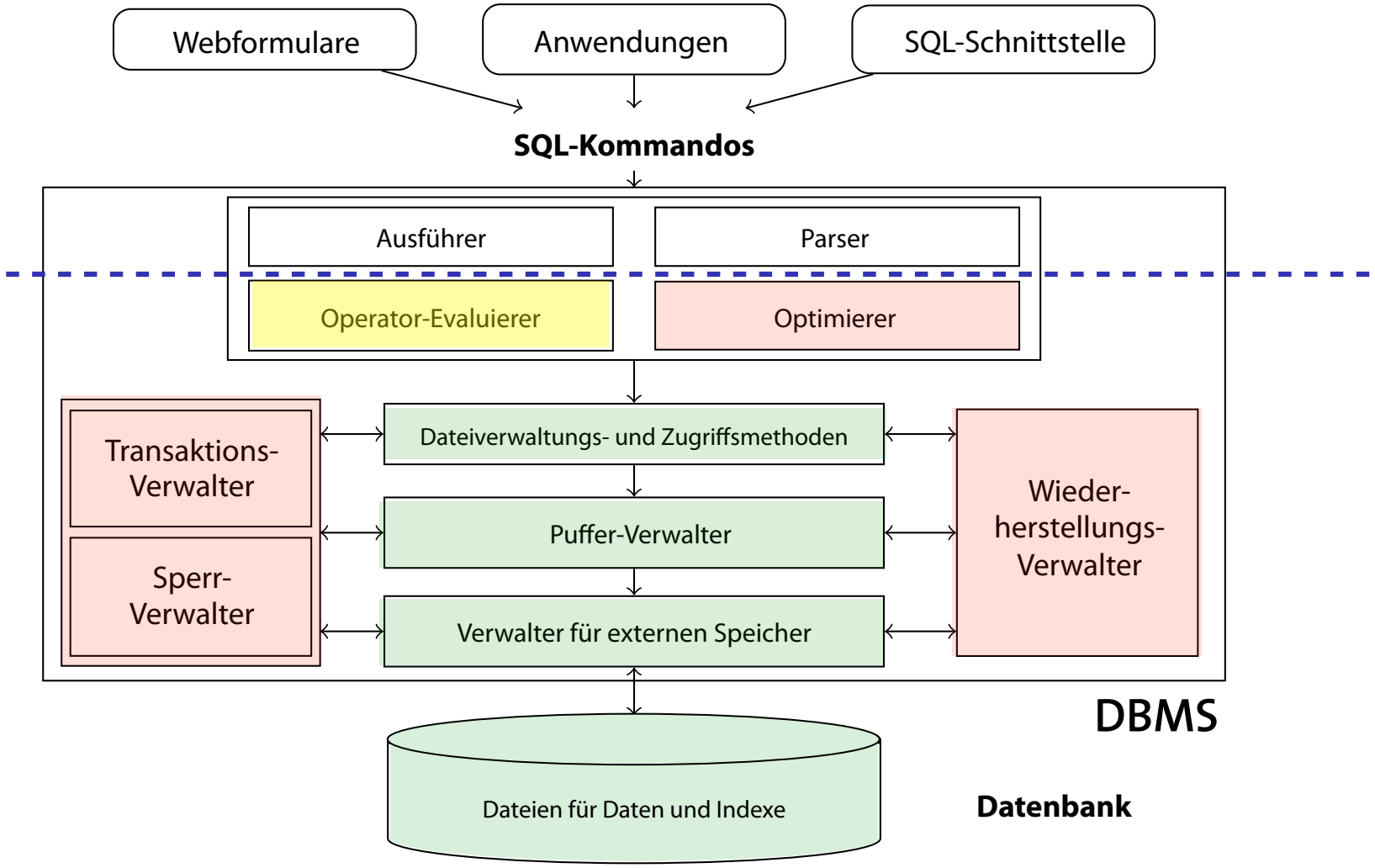
Datenbanken

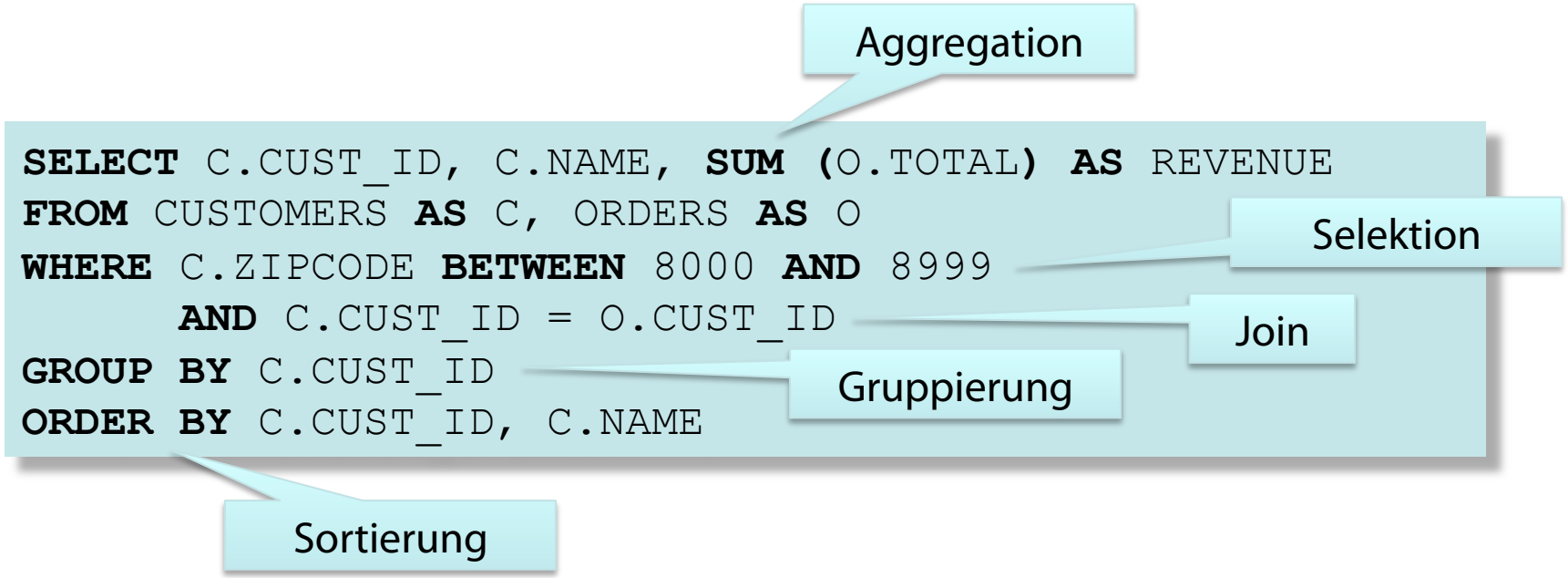
Prof. Dr. Ralf Möller
Universität zu Lübeck
Institut für Informationssysteme

Marc Stelzner (Übungen)
Torben Matthias Kempfert (Tutor)
Maurice-Raphael Sambale (Tutor)

Architektur eines DBMS



Anfragebeantwortung



Ein DBMS muss eine Menge von Aufgaben erledigen:

- mit minimalen Ressourcen
- über großen Datenmengen
- und auch noch so schnell wie möglich

Danksagung

- Diese Vorlesung ist inspiriert von den Präsentationen zu dem Kurs:

  „Architecture and Implementation of Database Systems“ von Jens Teubner an der ETH Zürich

- Graphiken und Code-Bestandteile wurden mit Zustimmung des Autors und ggf. kleinen Änderungen aus diesem Kurs übernommen

Sortierung

Wichtige Datenbankoperation mit vielen Anwendungen

- Eine SQL-Anfrage kann Sortierung anfordern

```
SELECT A,B,C FROM R ORDER BY A
```

- Bulk-loading eines B^+-Baumes fußt auf sortierten Daten
- Duplikate-Elimination wird besonders einfach

```
SELECT DISTINCT A,B,C FROM R
```

- Einige Datenbankoperatoren setzen sortierte Eingabedateien voraus (kommt später)

Wie können wir eine Datei sortieren, die nicht in den Hauptspeicher passt (und auf keinen Fall in den vom Pufferverwalter bereitgestellten Platz)?

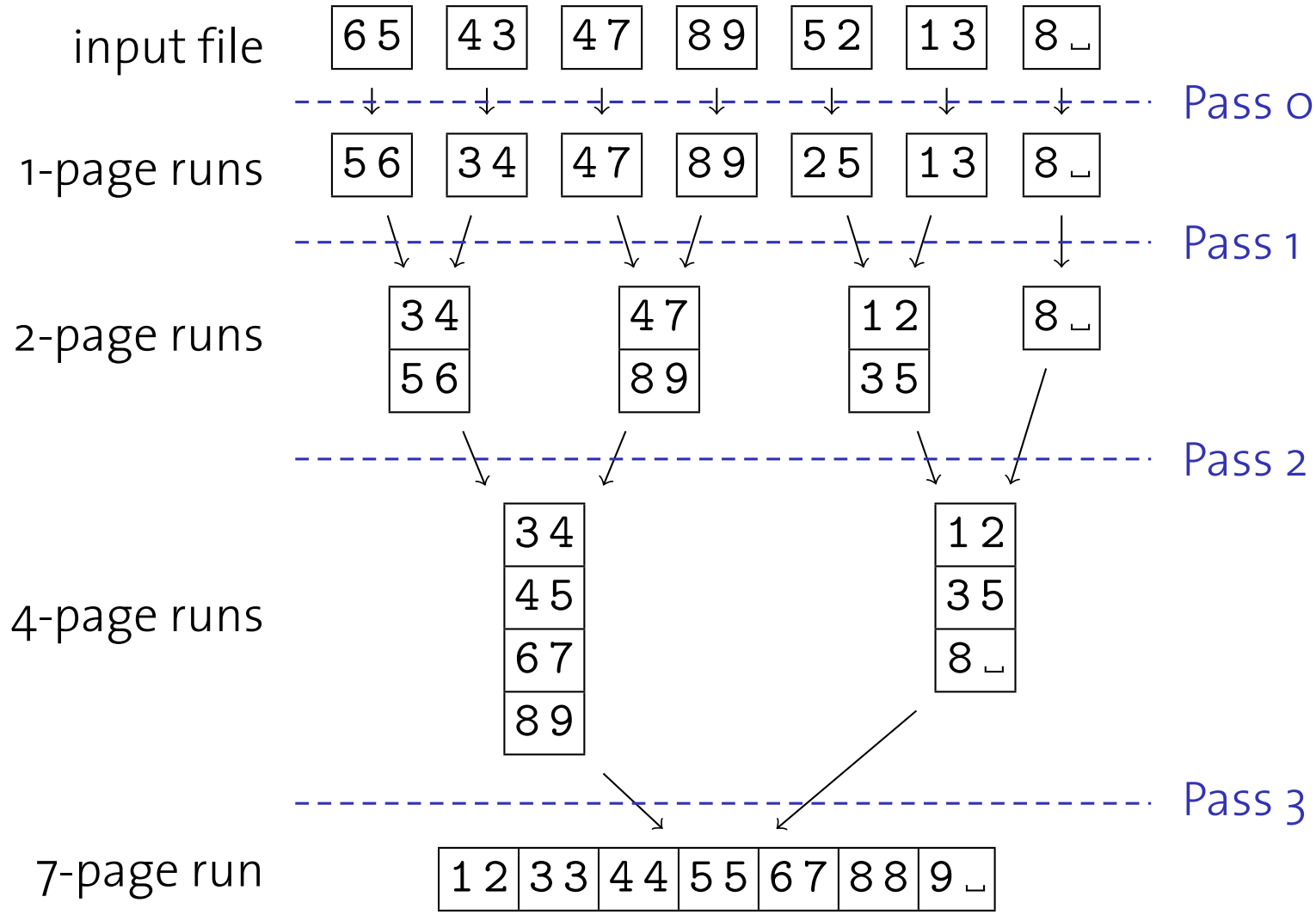
Zwei-Wege-Mischsortieren (Merge Sort)

Sortierung von Dateien beliebiger Größe in nur 3 Seiten aus dem Pufferverwalter

- Sortierung von $N=2^k$ Seiten in mehreren Durchgängen
- Jeder Durchgang produziert Sub-Dateien (Läufe)
  - Durchgang 0 sortiert 2^k Eingabeseiten einzeln im Hauptspeicher. Ergibt 2^k sortierte Läufe
  - Nachfolgende Durchgänge mischen Paare von Läufen Durchgang $n \leq k$ produziert 2^{k-n} Läufe
  - Durchgang k hinterlässt einen Lauf, das sortierte Ergebnis

In jedem Durchgang wird jede Seite der Datei gelesen
Ergibt: $(k+1) \cdot N$ Lesevorgänge und $(k+1) \cdot N$ Schreibvorgänge

Beispiel


input file
65 43 47 89 52 13 8␣
Pass 0
1-page runs
56 34 47 89 25 13 8␣
Pass 1
2-page runs
34 47 12 8␣
56 89 35
Pass 2
4-page runs
34 12
45 35
67 8␣
89
Pass 3
7-page run
12 33 44 55 67 88 9␣

Zwei-Wege-Mischsortieren

Pass 0 (**Input:** $N = 2^k$ unsorted pages; **Output:** 2^k sorted runs)

1. **Read** N pages, **one page at a time**
2. **Sort** records in main memory.
3. **Write** sorted pages to disk (each page results in a **run**).

This pass requires **one page** of buffer space.

Pass 1 (**Input:** $N = 2^k$ sorted runs; **Output:** 2^{k-1} sorted runs)

1. Open two runs r_1 and r_2 from Pass 0 for reading.
2. **Merge** records from r_1 and r_2, reading input page-by-page.
3. **Write** new two-page run to disk (page-by-page).

This pass requires **three pages** of buffer space.

⋮

Pass n (**Input:** 2^{k-n+1} sorted runs; **Output:** 2^{k-n} sorted runs)

1. Open two runs r_1 and r_2 from Pass $n-1$ for reading.
2. **Merge** records from r_1 and r_2, reading input page-by-page.
3. **Write** new 2^n-page run to disk (page-by-page).

This pass requires **three pages** of buffer space.

⋮

I/O-Verhalten

- Um eine Datei mit N Seiten zu sortieren, werden in jedem Durchgang N Seiten gelesen und geschrieben

  ⟶ 2·N I/O-Operationen pro Durchgang

- Anzahl der Durchgänge: $1 + \lceil \log_2 N \rceil$

  Durchgang 0 — Durchgänge 1..k

- Anzahl der I/O-Operationen:

$$2 \cdot N \cdot (1 + \lceil \log_2 N \rceil)$$

Aufgabe:

Anzahl der Durchgänge: $1 + \lceil \log_2 N \rceil$

Anzahl der I/O-Operationen:

$$2 \cdot N \cdot (1 + \lceil \log_2 N \rceil)$$

Wie lange dauert die Sortierung einer 8GB Datei?

Externes Mischsortieren

- Bisher freiwillig nur 3 Seiten verwendet
- Wie kann ein großer Pufferbereich genutzt werden:
  B Seiten
- Zwei wesentliche Stellgrößen
  - Reduktion der initialen Durchgänge durch Verwendung des Pufferspeichers beim Sortieren im Hauptspeicher
  - Reduktion der Anzahl der Durchgänge durch Mischen von mehr als 2 Seiten

Reduktion der Anzahl der Durchgänge

Mit B Seiten im Puffer können B-1 Seiten gemischt werden (eine Seite dient als Schreibpuffer)

Pass 0 (**Input:** *N* unsorted pages; **Output:** $\lceil N/B \rceil$ sorted runs)

1. **Read** *N* pages, *B* **pages at a time**
2. **Sort** records in main memory.
3. **Write** sorted pages to disk (resulting in $\lceil N/B \rceil$ **runs**).

This pass uses *B* **pages** of buffer space.

(B-1)-Wege-Mischen:

- Anzahl der I/O-Operationen:

$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

- Wie lang dauert die Sortierung einer 8GB-Datei?

Externes Sortieren: I/O-Verhalten

- Sortierung von N Seiten mit B Pufferseiten benötigt

$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

  I/O-Operationen.

Was ist das Zugriffsmuster auf diese Ein-Ausgaben?

Blockweise Ein-Ausgabe

Man kann das I/O-Muster verbessern, in dem man Blöcke von b Seiten in den Mischphasen verarbeitet

- Alloziere b Seiten für jede Eingabe (statt nur eine)
- Reduktion der Ein-Ausgabe um Faktor b
- Preis: Reduzierte Einfächerung (was in mehr Durchgängen und mehr I/O-Operationen resultiert)
- In der Praxis meist genügend Hauptspeicher vorhanden, so dass Dateien in einem Mischdurchgang sortiert werden kann (mit blockweisem I/O).

Aufgabe:

Wie lange dauert die Sortierung einer 8GB Datei mit 1000 Pufferseiten je 8KB mit 10ms Plattenlatenz

Anzahl der Durchgänge: $1 + \lceil \log_{B-1} N/B \rceil$

Anzahl der I/O-Operationen:

$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

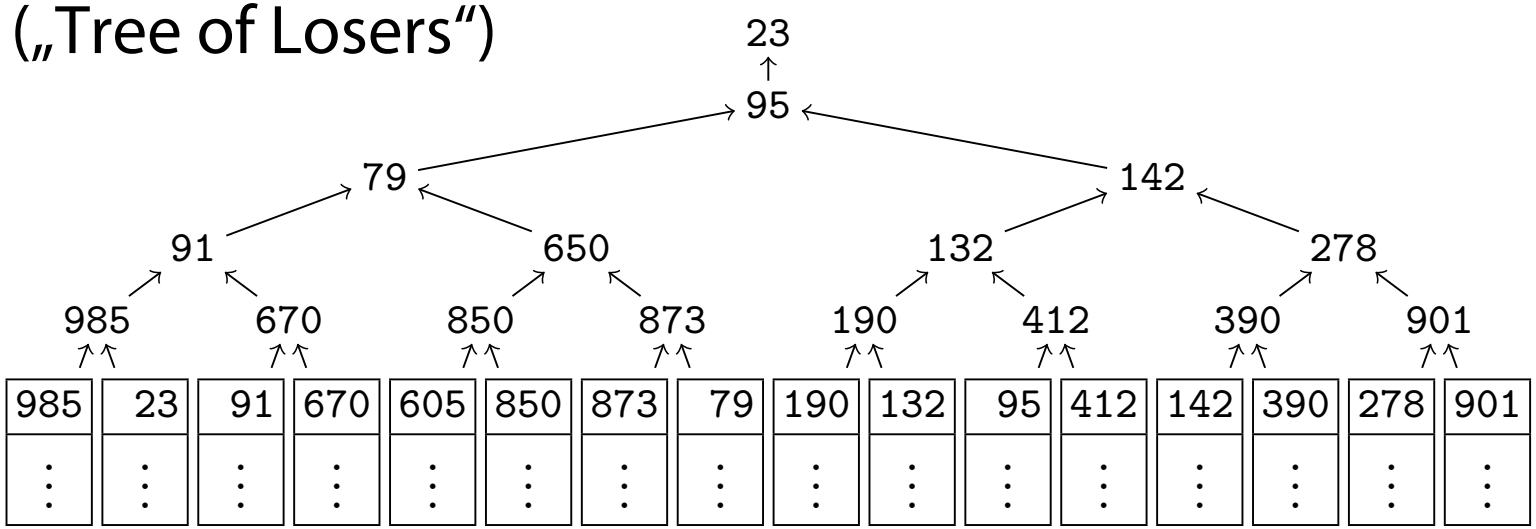
Lösung

- Ohne blockweises I/O:
  $\approx 4 \cdot 10^6$ Plattenzugriffe (11h)
  + Transfer von $\approx 6 \cdot 10^6$ Seiten (17min)
- Mit blockweisem I/O (Blöcke von 32 Seiten):
  $\approx 6 \cdot 32768$ Plattenzugriffe (33min)
  + Transfer von $\approx 8 \cdot 10^6$ Seiten (22min)

Auswahlbäume

Auswahl des nächsten Datensätzes aus B-1 (oder B/b -1) Eingabeläufen kann CPU-intensiv sein (B-2 Vergleiche)

- Verwende Auswahlbaum zur Kostenreduktion („Tree of Losers“)

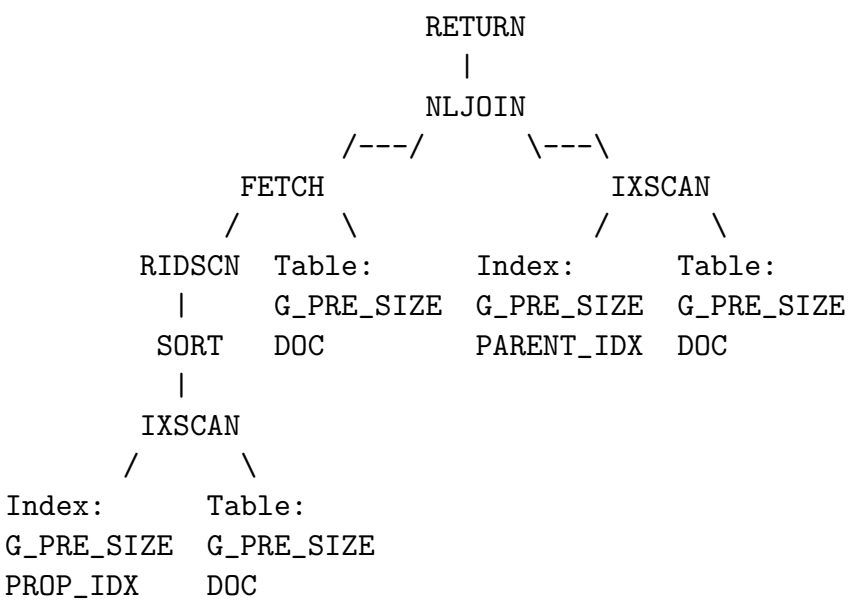


- Reduktion der Vergleiche auf $\log_2$ (B-1)

Externes Sortieren: Diskussion

- Misch-Schritte können auch parallel ausgeführt werden
- Bei ausreichend Speicher reichen zwei Durchgänge auch für große Dateien
- Mögliche Optimierungen:
  - Seitenersetzung während des Sortierens: Erneutes Laden neuer Seiten während des initialen Laufs (dadurch Erhöhen der initialen Lauflänge)
  - Doppelpufferung: Verschränkung des Seitenladevorgangs und der Verarbeitung, um Latenzzeiten der Festplattenspeicher zu cachieren

Ausführungspläne



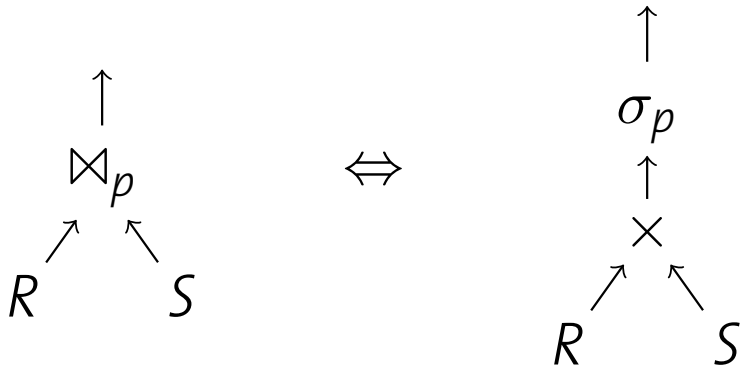
Ausführungsplan (DB2)

- Externes Sortieren ist eine Instanz eines physikalischen Datenbankoperators
- Operatoren können zu Ausführungsplänen zusammengesetzt werden
- Jeder Planoperator führt zur Verarbeitung einer vollständigen Anfrage eine Unteraufgabe aus

Wir werden zunächst Verbundoperatoren betrachten

Verbundoperator (Join) $\bowtie$

Ein Verbundoperator $\bowtie_p$ ist eine Abkürzung für die Zusammensetzung von Kreuzprodukt $\times$ und Selektion σ_p



Daraus ergibt sich eine einfache Implementierung von $\bowtie_p$

1. Enumeriere alle Datensätze aus $R \times S$
2. Wähle die Datensätze, die p erfüllen

Ineffizienz aus Schritt 1 kann überwunden werden
(Größe des Zwischenresultats: $|R| \times |S|$)

Verbund-als-geschachtelte-Schleifen

Einfache Implementierung des Verbundes:

```
1 Function: nljoin(R, S, p)
2 foreach record r ∈ R do
3     foreach record s ∈ S do
4         if ⟨r, s⟩ satisfies p then
5             append ⟨r, s⟩ to result
```

Sei N_R und N_S die Seitenzahl in R und S, sei p_R und p_S die Anzahl der Datensätze pro Seite in R und S

Anzahl der Plattenzugriffe:

$$N_R + \underbrace{p_r \cdot N_R}_{\text{\#Tupel in R}} \cdot N_S$$

Verbund-als-geschachtelte-Schleifen

Nur 3 Seiten nötig (zwei Seiten fürs Lesen von R und S und eine um das Ergebnis zu schreiben)

I/O-Verhalten: Leider sehr viele Zugriffe

- Annahme $p_R = p_S = 100$, $N_R = 1000$, $N_R = 500$:

  $1000 + 5 \cdot 10^7$ Seiten zu lesen

- Mit einer Zugriffszeit von 10ms für jede Seite dauert der Vorgang 140 Stunden
- Vertauschen von R und S (kleinere Relation S nach außen) verbessert die Situation nur marginal

Seitenweises Lesen bedingt volle Plattenlatenz, obwohl beide Relationen in sequentieller Ordnung verarbeitet werden.

Blockweiser Verbund mit Schleifen

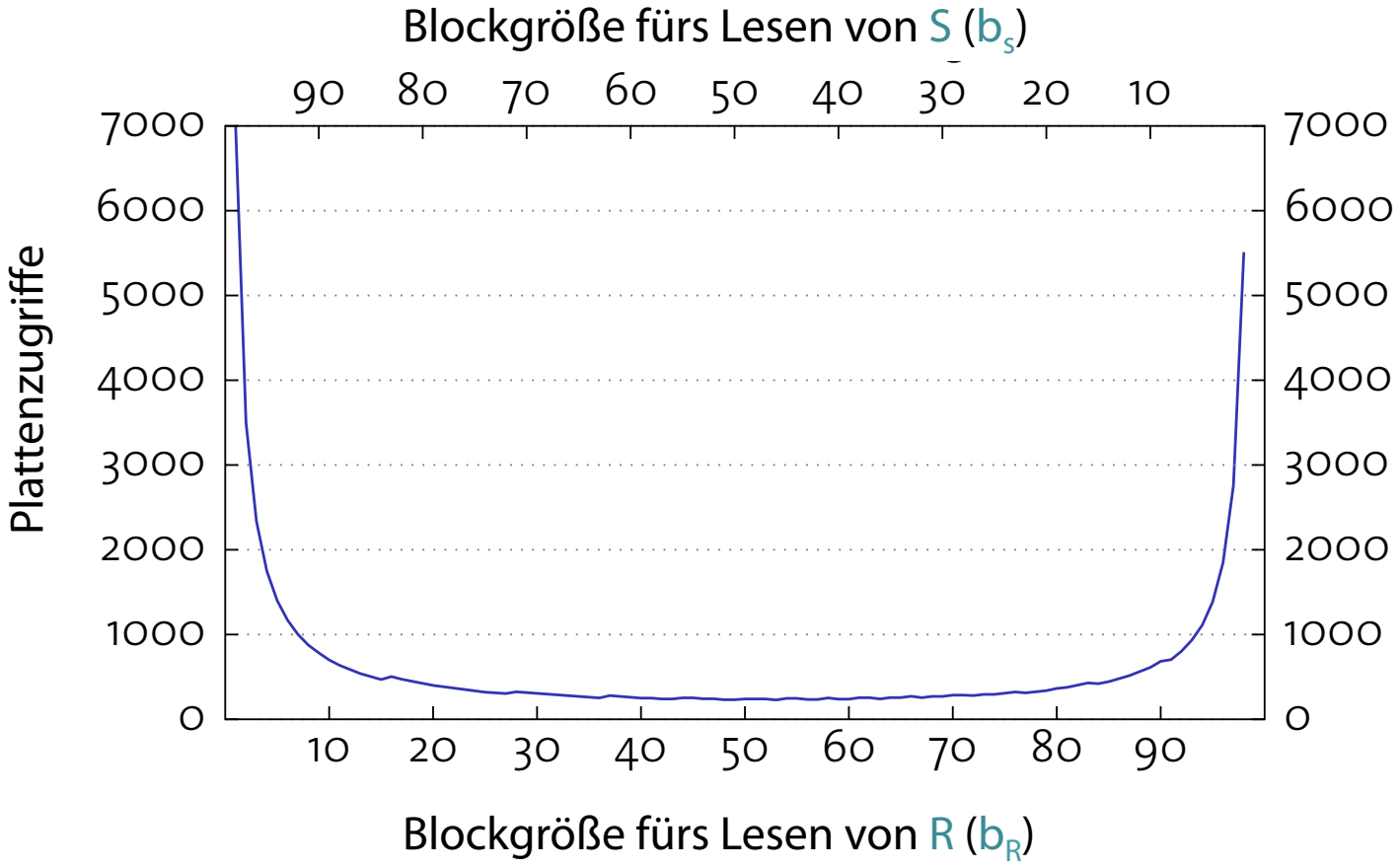
Einsparung von Kosten durch wahlfreien Zugriff durch blockweises Lesen von R und S mit b_R und b_S vielen Seiten

```
1 Function: block_nljoin(R, S, p)
2 foreach b_R-sized block in R do
3     foreach b_S-sized block in S do
4         find matches in current R- and S-blocks and
          append them to the result ;
```

- R wird vollständig gelesen, aber mit nur $\lceil N_R/b_R \rceil$ Lesezugriffen
- S nur $\lceil N_R/b_R \rceil$ mal gelesen, mit $\lceil N_R/b_R \rceil \cdot \lceil N_S/b_S \rceil$ Plattenzugriffen

Wahl von b_R und b_S

Pufferbereich mit $B = 100$ Rahmen, $N_R = 1000$, $N_S = 500$:



Performanz des Hauptspeicher-Verbunds

- Zeile 4 in block_nljoin(R, S, p) bedingt einen Hauptspeicherverbund zwischen Blocken aus R und S
- Aufbau einer Hashtabelle kann den Verbund erheblich beschleunigen

```
1 Function: block_nljoin' (R, S, p)
2 foreach b_R-sized block in R do
3     build an in-memory hash table H for the current R-block ;
4     foreach b_S-sized block in S do
5         foreach record s in current S-block do
6             probe H and append matching ⟨r, s⟩ tuples to result ;
```

- Funktioniert nur für Equi-Verbunde

Indexbasierte Verbunde

Verwendung eines Index für die innere Relation
(ggf. innere und äußere vertauschen)

```
1 Function: index_nljoin (R, S, p)
2 foreach record r ∈ R do
3     probe index using r and append all matching
      tuples to result ;
```

- Index muss verträglich mit der Verbundbedingung sein
  - Hash-Index (nur für Gleichheitsprädikate)
  - Vergleiche auch die Diskussion über zusammengesetzte Schlüssel in B^+-Bäumen
- Argumente heißen „sargable“ (SARG= search argument)

Access Path Selection in a Relational Database Management System, Selinger et al., SIGMOD **1979**

I/O-Verhalten

Für jeden Datensatz in R verwende Index zum Auffinden von korrespondierenden S-Tupeln. Für jedes R-Tupel sind folgende Kosten einzukalkulieren:

1. Zugriffskosten für den Index zum Auffinden des ersten Eintrags: N_{idx} I/O-Operationen
2. Entlanglaufen an den Indexwerten (Scan), um passende Rids zu finden (I/O-Kosten vernachlässigbar)
3. Holen der passenden S-Tupel aus den Datenseiten
   - Für ungelusterten Index: n I/O-Operationen
   - Für geclusterten Index: $\lceil n/Ps \rceil$ I/O-Operationen

Wegen 2. und 3. Kosten von der Verbundgröße abhängig

Zugriffskosten für Index

Falls Index ein B^+-Baum:

- Einzelner Indexzugriff benötigt Zugriff auf h Indexseiten[1]
- Bei wiederholtem Zugriff sind diese Seiten im Puffer
- Effektiver Wert der I/O-Kosten 1-3 I/O-Operationen

Falls Index ein Hash-Index:

- Caching nicht effektiv (kein lokaler Zugriff auf Hashfeld)
- Typischer Wert für I/O-Kosten: 1,2 I/O-Operationen (unter Berücksichtigung von Überlaufseiten)

Index rentiert sich stark, wenn nur einige Tupel aus einer großen Tabelle im Verbund landen

[1] h = Höhe des B^+-Baumes

Sortier-Misch-Verbund

Verbundberechnung wird einfach wenn Eingaberelationen bzgl. Verbundattribut(en) sortiert

- Misch-Verbund mischt Eingabetabellen ähnlich wie beim Sortieren
- Es gibt aber mehrfache Korrespondenzen in der anderen Relation

A	*B*
"foo"	1
"foo"	2
"bar"	2
"baz"	2
"baf"	4

$\bowtie_{B=C}$

C	*D*
1	false
2	true
2	false
3	true

- Misch-Verbund nur für Equi-Verbünde verwendbar

Misch-Verbund

```
1  Function: merge_join (R, S, α = β)    // α, β: join columns in R, S
2  r ← position of first tuple in R ;    // r, s, s′: cursors over R, S, S
3  s ← position of first tuple in S ;
4  while r ≠ eof and s ≠ eof do          // eof: end of file marker
5      while r.α < s.β do
6          advance r ;
7      while r.α > s.β do
8          advance s ;
9      s′ ← s ;                          // Remember current position in S
10     while r.α = s′.β do               // All R-tuples with same α value
11         s ← s′ ;                      // Rewind s to s′
12         while r.α = s.β do            // All S-tuples with same β value
13             append ⟨r, s⟩ to result ;
14             advance s ;
15         advance r ;
```

I/O-Verhalten

- Wenn beide Eingaben sortiert und keine außergewöhnlich langen Sequenzen mit identischen Schlüsselwerten vorhanden, dann ist der I/O-Aufwand $N_R + N_S$ (das ist dann optimal)
- Durch blockweise I/O treten fast immer sequentielle Lesevorgänge auf
- Es kann sich für die Verbundberechnung auszahlen, vorher zu sortieren, insbesondere wenn später eine Sortierung der Ausgabe gefordert wird
- Ein abschließender Sortiervorgang kann auch mit einem Misch-Verbund kombiniert werden, um Festplattentransfers einzusparen

Aufgabe:

Bei welchen Eingaben tritt beim Sortier-Misch-Verbund das schlimmste Verhalten auf?

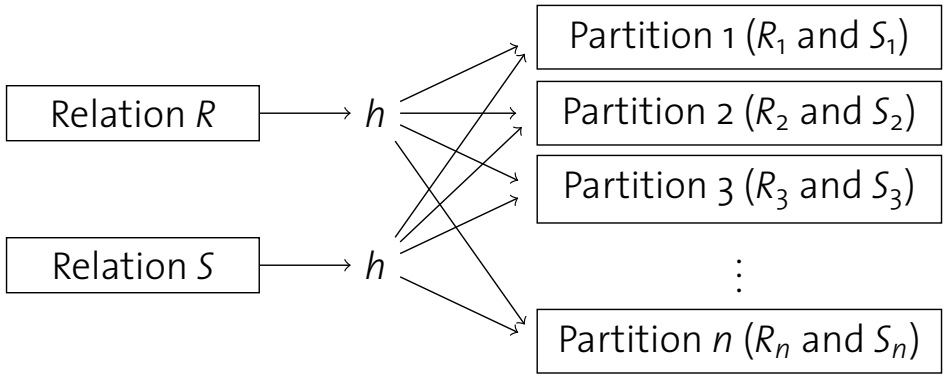
Lösung

- Wenn all verbundenen Attribute gleiche Werte beinhalten, dann ist das Ergebnis ein Kreuzprodukt.

- Der Sortier-Misch-Verbund verhält sich dann wie ein geschachtelte-Schleifen-Verbund.

Hash-Verbund

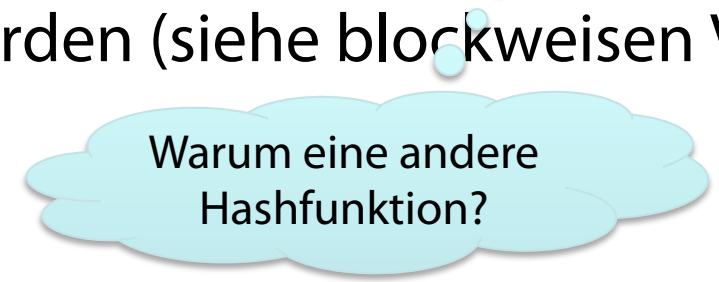
- Sortierung bringt korrespondierende Tupel in eine „räumliche Nähe“, so dass eine effiziente Verarbeitung möglich ist
- Ein ähnlicher Effekt erreichbar mit Hash-Verfahren
- Zerlege R und S in Teilrelationen $R_1, ..., R_n$ und $S_1,...,S_n$ mit der gleichen Hashfunktion (angewendet auf die Verbundattribute)



- $R_i \bowtie R_j = \emptyset$ für alle $i \neq j$

Hash-Verbund

- Durch Partitionierung werden kleine Relationen R_i and S_i geschaffen
- Korrespondierende Datensätze kommen garantiert in die gleiche Relation
- Es muss $R_i \bowtie S_i$ (für all i) berechnet werden (einfacher)
- Die Anzahl der Partitionen n (d.h. die Hashfunktion) sollte mit Bedacht gewählt werden, so dass $R_i \bowtie S_i$ als Hauptspeicher-Verbund berechnet werden kann
- Hierzu kann wiederum eine (andere) Hashfunktion verwendet werden (siehe blockweisen Verbund mit Schleifen)



Hash-Verbund-Algorithmus

```
1  Function: hash_join (R, S, α = β)
2  foreach record r ∈ R do
3      append r to partition R_{h(r.α)}
4  foreach record s ∈ S do
5      append s to partition S_{h(s.β)}
6  foreach partition i ∈ 1, ..., n do
7      build hash table H for R_i, using hash function h′;
8      foreach block in S_i do
9          foreach record s in current S_i-block do
10             probe H and append matching tuples to result ;
```

Gruppierung und Duplikate-Elimination

- Herausforderung: Finde identische Datensätze in einer Datei
- Ähnlichkeiten zum Eigenverbund (self-join) basieren auf allen Spalten der Relation
  - Man könnte einen Hash-Verbund-ähnlichen Algorithmus verwenden oder Sortierung, um Duplikate-Elimination oder Gruppierung zu realisieren

Andere Anfrage-Operatoren

Projektion π

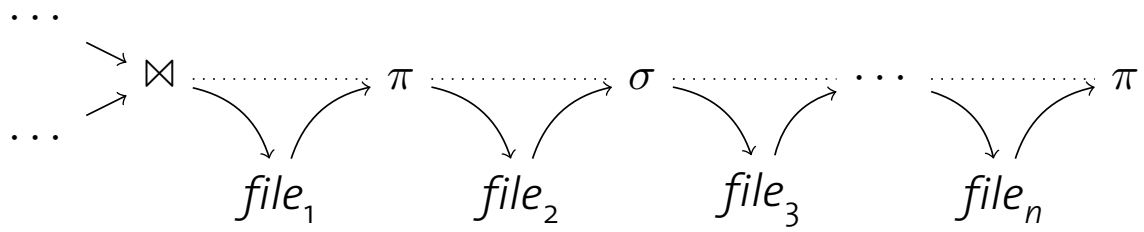
- Implementierung durch
  a. Entfernen nicht benötigter Spalten
  b. Eliminierung von Duplikaten
- Die Implementierung von a) bedingt das Ablaufen (scan) aller Datensätze in der Datei, b) siehe oben
- Systeme vermeiden b) sofern möglich (in SQL muss Duplikate-Eliminierung angefordert werden)

Selektion σ

- Ablaufen (scan) aller Datensätze
- Eventuell Sortierung ausnutzen oder Index verwenden

Organisation der Operator-Evaluierung

- Bisher gehen wir davon aus, dass Operatoren ganze Dateien verarbeiten



- Das erzeugt offensichtlich viel I/O
- Außerdem: lange Antwortzeiten
  - Ein Operator kann nicht anfangen solange nicht seine Eingaben vollständig bestimmt sind (materialisiert sind)
  - Operatoren werden nacheinander ausgeführt

Pipeline-orientierte Verarbeitung

- Alternativ könnte jeder Operator seine Ergebnisse direkt an den nachfolgenden senden, ohne die Ergebnisse erst auf die Platte zu schreiben
- Ergebnisse werden so früh wie möglich weitergereicht und verarbeitet (Pipeline-Prinzip)
- Granularität ist bedeutsam:
  - Kleinere Brocken reduzieren Antwortzeit des Systems
  - Größere Brocken erhöhen Effektivität von Instruktions-Cachespeichern
  - In der Praxis meist tupelweises Verarbeiten verwendet

Volcano Iteratormodell

- Aufrufschnittstelle wie bei Unix-Prozess-Pipelines
- Im Datenbankkontext auch Open-Next-Close-Schnittstelle oder Volcano Iteratormodell genannt
- Jeder Operator implementiert
  - open() Initialisiere den internen Zustand des Operators
  - next() Produziere den nächsten Ausgabe-Datensatz
  - close() Schließe allozierte Ressourcen
- Zustandsinformation wird Operator-lokal vorgehalten

Goetz Graefe. Volcano—An Extensibel and Parallel Query Evaluation System. Trans. Knowl. Data Eng. vol. 6, no. 1, February **1994**

Beispiel: Selektion (σ)

Eingabe: Relation R, Prädikat p

```
1 Function: open ()
2 R.open () ;
```

```
1 Function: close ()
2 R.close () ;
```

```
1 Function: next ()
2 while ((r ← R.next ()) ≠ eof) do
3     if p(r) then
4         return r ;
5 return eof ;
```

Geschachtelte Schleifen Verbund: Volcano-Stil

```
1 Function: open ()
2 R.open () ;
3 S.open () ;
4 r ← R.next () ;
```

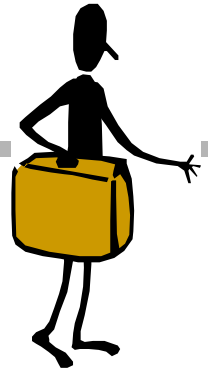
```
1 Function: close ()
2 R.close () ;
3 S.close () ;
```

```
1 Function: next ()
2 while (r ≠ eof) do
3     while ((s ← S.next ()) ≠ eof) do
4         if p(r, s) then
5             return ⟨r, s⟩ ;
6     S.close () ;
7     S.open () ;
8     r ← R.next();
9 return eof ;
```

Blockierende Operatoren

- Pipelining reduziert Speicheranforderungen und Antwortzeiten, da jeder Datensatz gleich weitergeleitet
- Funktioniert so nicht für alle Operatoren
- Welche?
  - Externe Sortierung
  - Hash-Verbund
  - Gruppierung und Duplikate-Elimination über einer unsortierten Eingabe
- Solche Operatoren nennt man blockierend
- Blockierende Operatoren konsumieren die gesamt Eingabe in einem Rutsch bevor die Ausgabe erzeugt werden kann (Daten auf Festplatte zwischengespeichert)

Zusammenfassung



Teile-und-Herrsche

- Zerlegung einer großen Anfrage in kleine Teile
- Beispiel:
  - Laufgenerierung und externe Sortierung
  - Partitionierung mit Hashfunktion (Hash-Verbund)

Blockweises Durchführen von I/O

- Lesen und Schreiben von größeren Einheiten kann die Verarbeitungszeit deutlich reduzieren, da wahlfreier Zugriff auf Daten vermieden wird

Pipeline-orientierte Verarbeitung

  - Speicherreduktion und Laufzeitverbesserung durch Vermeidung der vollständigen Materialisierung von Zwischenresultaten

Beim nächsten Mal...