

---

# Datenbanken

Prof. Dr. Ralf Möller

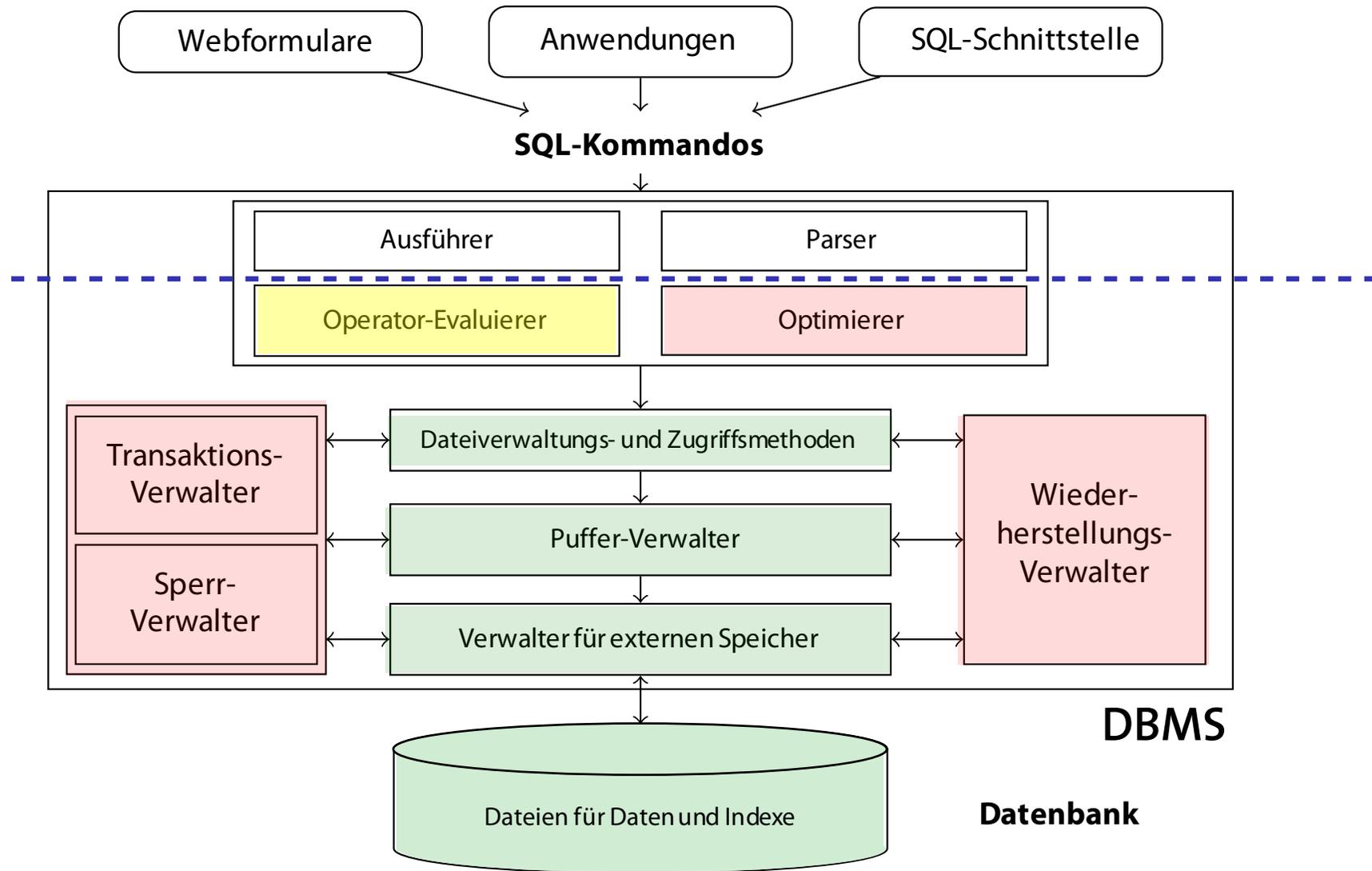
**Universität zu Lübeck**

**Institut für Informationssysteme**

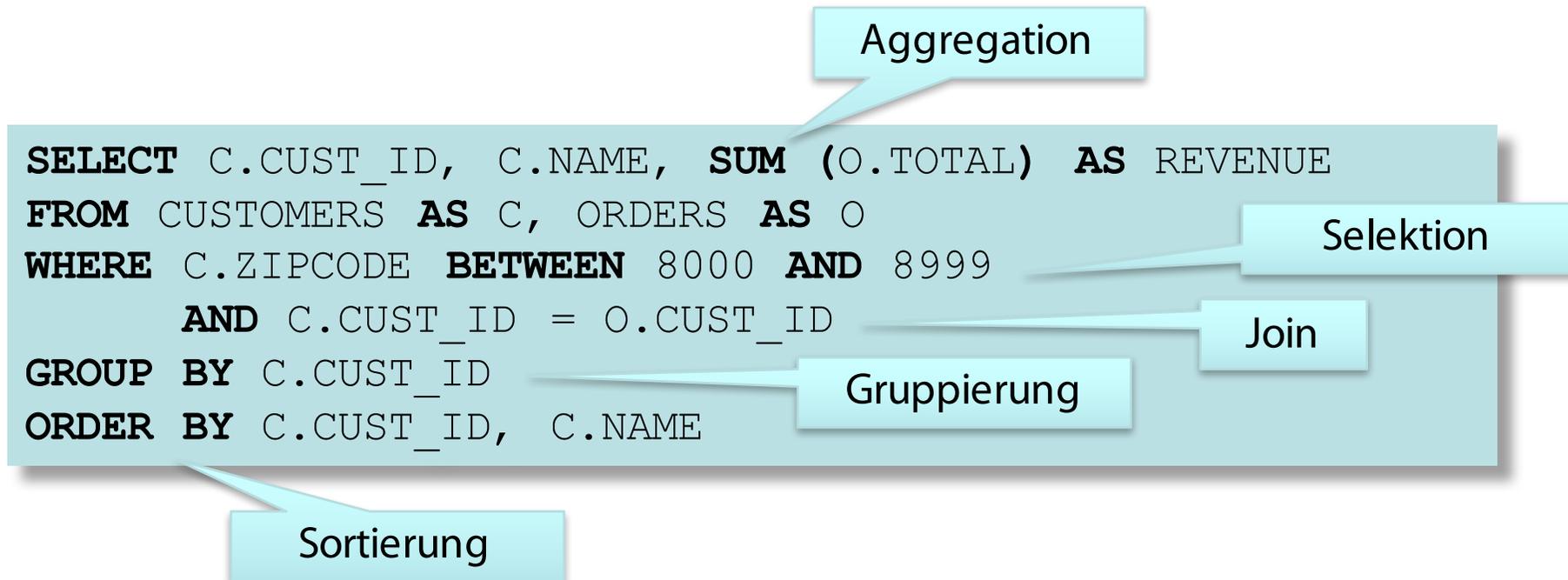
Karsten Martiny (Übungen)



# 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<sup>+</sup>-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)?*

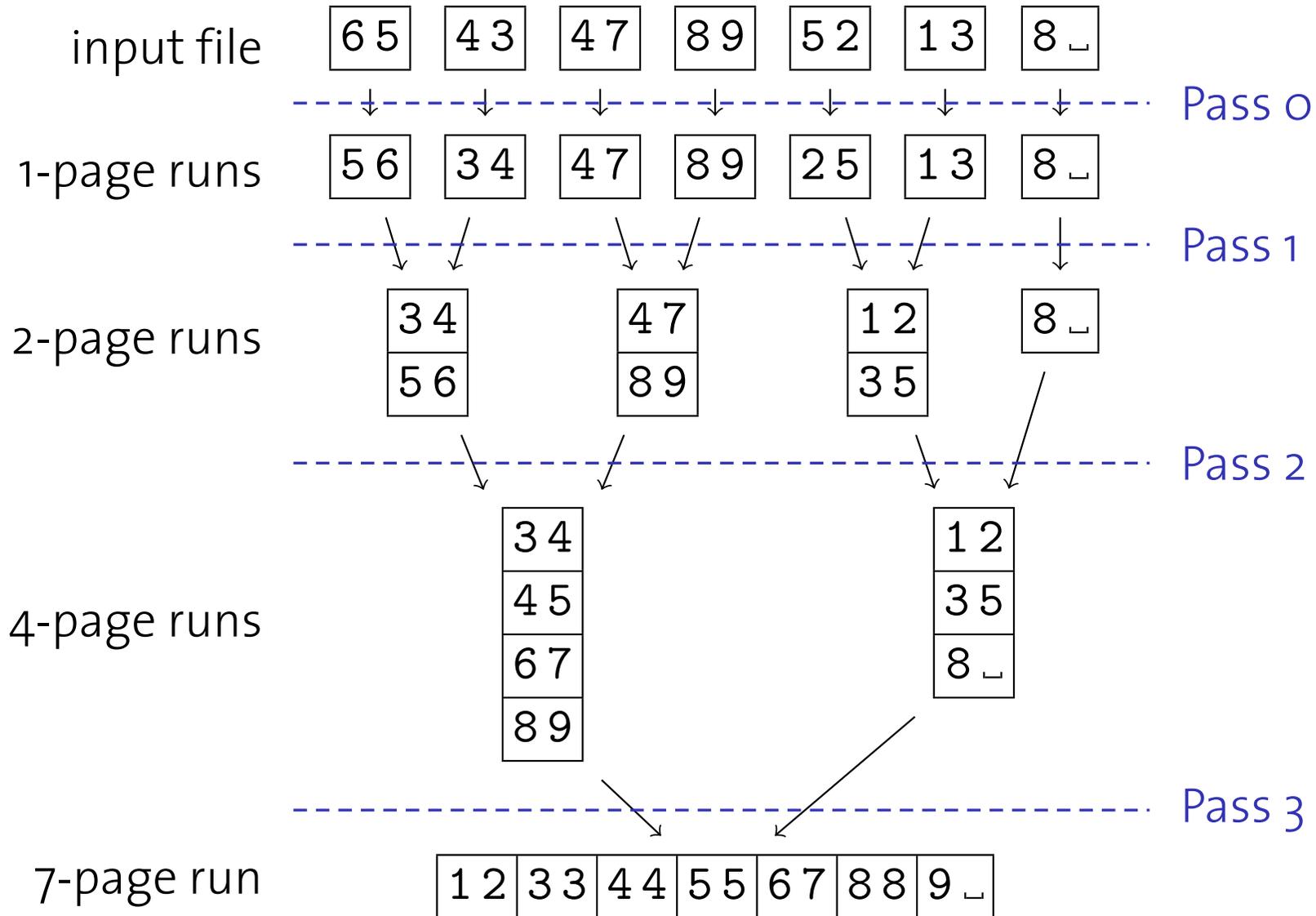
# Zwei-Wege-Mischsortieren (Merge Sort)

---

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

- Sortierung von **N** Seiten in mehreren Durchgängen

# Beispiel



# 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.



## 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 bei einer Travelstar 7k200?

- $t_s = 10 \text{ ms}$ ,  $t_r = 4.17 \text{ ms}$ ,  $t_{tr} = 0,16 \text{ ms}$ , 8KB pro Seite

# Lösung

---

$\approx 6d$

(Herleitung an der Tafel)



# Externes Mischsortieren

---

- Bisher freiwillig nur 3 Seiten verwendet
- Wie kann ein großer Pufferbereich genutzt werden:  
B Seiten auf einmal bearbeiten
- 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  $B$  Seiten eingelesen und im Hauptspeicher sortiert werden

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.

- Anzahl der I/O-Operationen:

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

- Was ist das Zugriffsmuster auf diese Ein-Ausgaben?

## Aufgabe:

Wie lange dauert die Sortierung einer 8GB Datei mit  $B=1000$  Pufferseiten je 8KB mit einer Travelstar 7k200?

- $t_s = 10 \text{ ms}$ ,  $t_r = 4.17 \text{ ms}$ ,  $t_{tr} = 0,16 \text{ ms}$ , 8KB pro Seite

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

Anzahl der I/O-Operationen:

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

# Lösung

---

$\approx 3d$

(Herleitung an der Tafel)



# Reduktion der Anzahl der Durchgänge

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

**Pass  $n$**  (Input:  $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$  sorted runs; Output:  $\frac{\lceil N/B \rceil}{(B-1)^n}$  sorted runs)

1. Open  $B - 1$  runs  $r_1 \dots r_{B-1}$  from Pass  $n - 1$  for reading.
2. **Merge** records from  $r_1 \dots r_{B-1}$ , reading input page-by-page.
3. **Write** new  $B \cdot (B - 1)^n$ -page run to disk (page-by-page).

This pass requires  $B$  **pages** of buffer space.

( $B-1$ )-Wege-Mischen:

- Anzahl der I/O-Operationen:

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

- Was ist das Zugriffsmuster auf diese Ein-Ausgaben?

## Aufgabe:

Wie lange dauert die Sortierung einer 8GB Datei mit  $B=1000$  Pufferseiten je 8KB mit einer Travelstar 7k200?

- $t_s = 10 \text{ ms}$ ,  $t_r = 4.17 \text{ ms}$ ,  $t_{tr} = 0,16 \text{ ms}$ , 8KB pro Seite

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

---

ca.  $4 \cdot 10^6$  Plattenzugriffe mit je 14.33ms

≈ 16h

(Herleitung an der Tafel)



# 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 damit in mehr I/O-Operationen resultiert)

## Aufgabe:

Wie lange dauert die Sortierung einer 8GB Datei mit  $B=1000$  Pufferseiten je 8KB und blockweisem IO mit  $b=32$  bei einer Travelstar 7k200 ?

- $t_s = 10$  ms,  $t_r = 4.17$  ms,  $t_{tr} = 0,16$ ms, 8KB pro Seite
- 63 Sektoren pro Spur, Track-to-Track-Suchzeit  $t_{s,track-to-track} = 1$  ms
- Ein Block mit 8 KB benötigt 16 Sektoren

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

Anzahl der I/O-Operationen:

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

# Lösung

---

Mit blockweisem I/O (B=1000 Blöcke mit b=32 Seiten):  
ca.  $8 \cdot 31250$  Plattenzugriffe mit je 27.41 ms  
 $\approx 2 \text{ h}$

(Herleitung in der Übung)

# Hauptsspeicher als Ressource

---

In der Praxis meist genügend Hauptspeicher vorhanden, so dass Dateien in einem Mischdurchgang sortiert werden kann (mit blockweisem I/O).

## Aufgabe:

Wieviel Hauptspeicher wird bei Pufferseiten mit je 8KB und blockweisem IO mit  $b=32$  benötigt, so dass ein Mischvorgang für eine 8GB Datei reicht?

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

$$\lceil \log_{\lceil B/b \rceil - 1} \lceil N / (B/b) \rceil \rceil = 1$$

# Lösung

---

≈ 1h

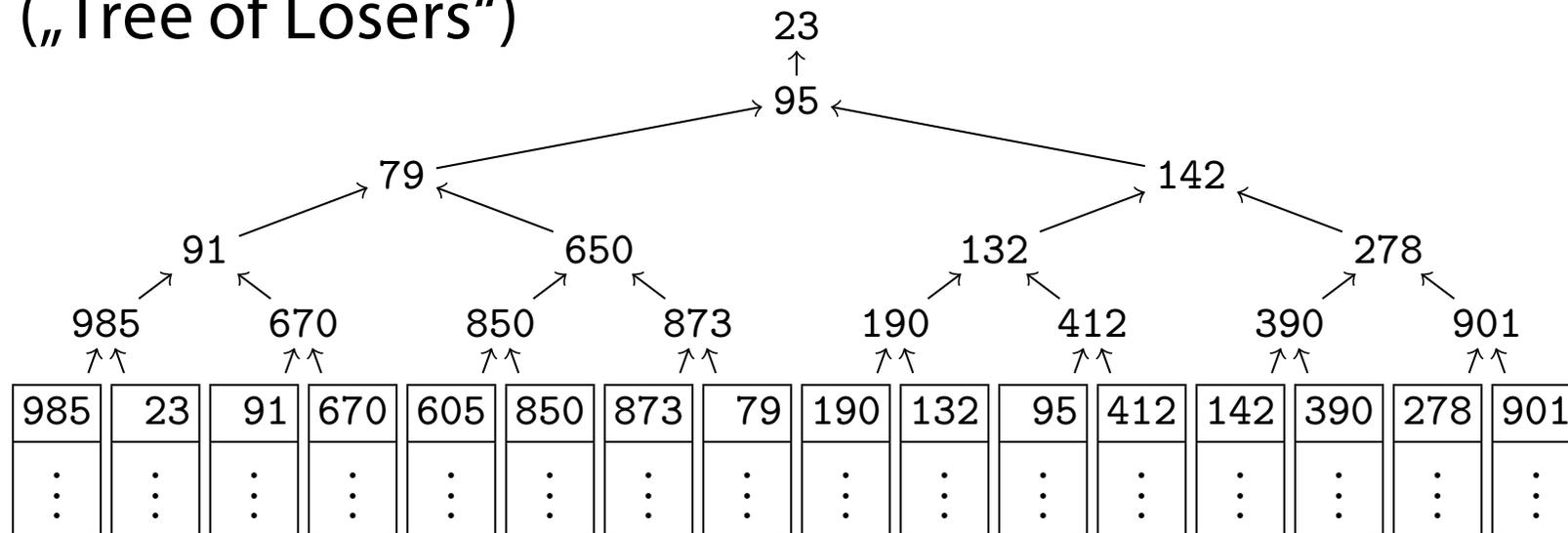
(Herleitung in der Übung)



# Bisher nur IO-Zeit betrachtet

Auswahl des nächsten Datensatzes 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 kaschieren

---

# Datenbanken

Prof. Dr. Ralf Möller

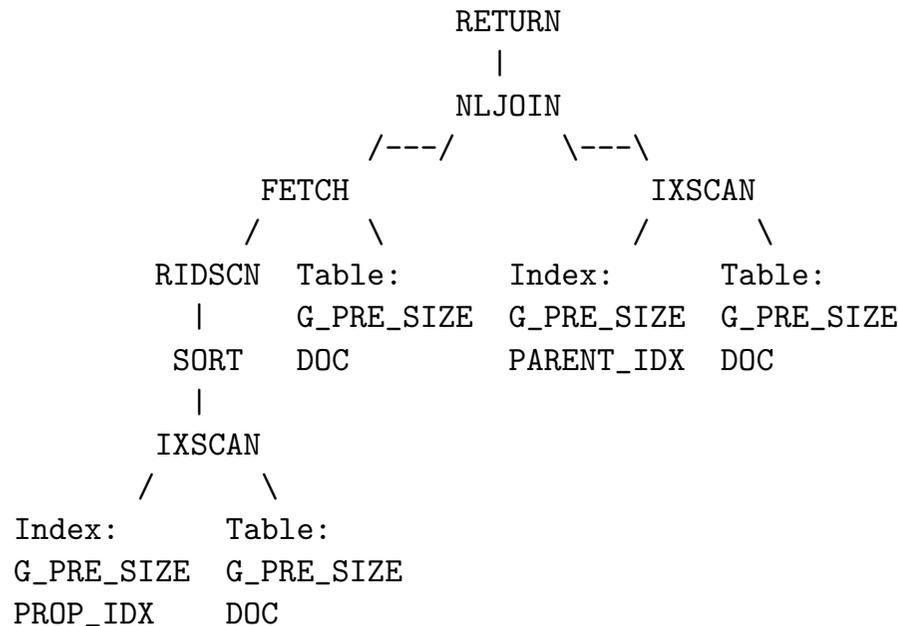
**Universität zu Lübeck**

**Institut für Informationssysteme**

Karsten Martiny (Übungen)



# Ausführungspläne



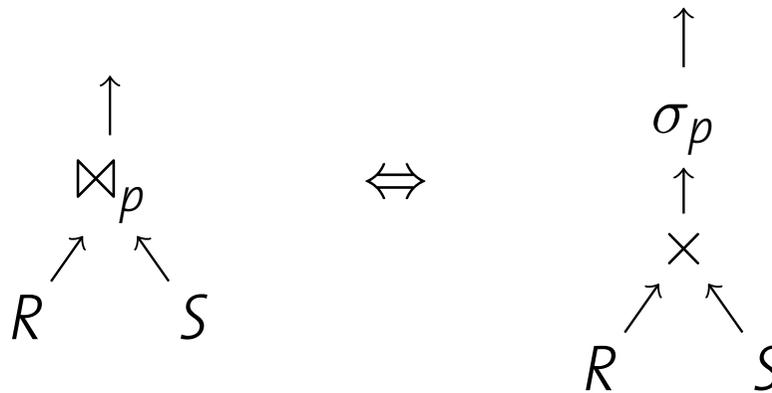
Ausführungsplan (DB2)

- 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) $R \bowtie S$

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



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  $r \in R$  do
3   foreach record  $s \in S$  do
4     if  $\langle r, s \rangle$  satisfies  $p$  then
5       append  $\langle r, s \rangle$  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**:

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

# 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  $p_R = p_S = 100$ ,  $N_R = 1000$ ,  $N_S = 500$ :  
 $1000 + 5 \cdot 10^7$  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

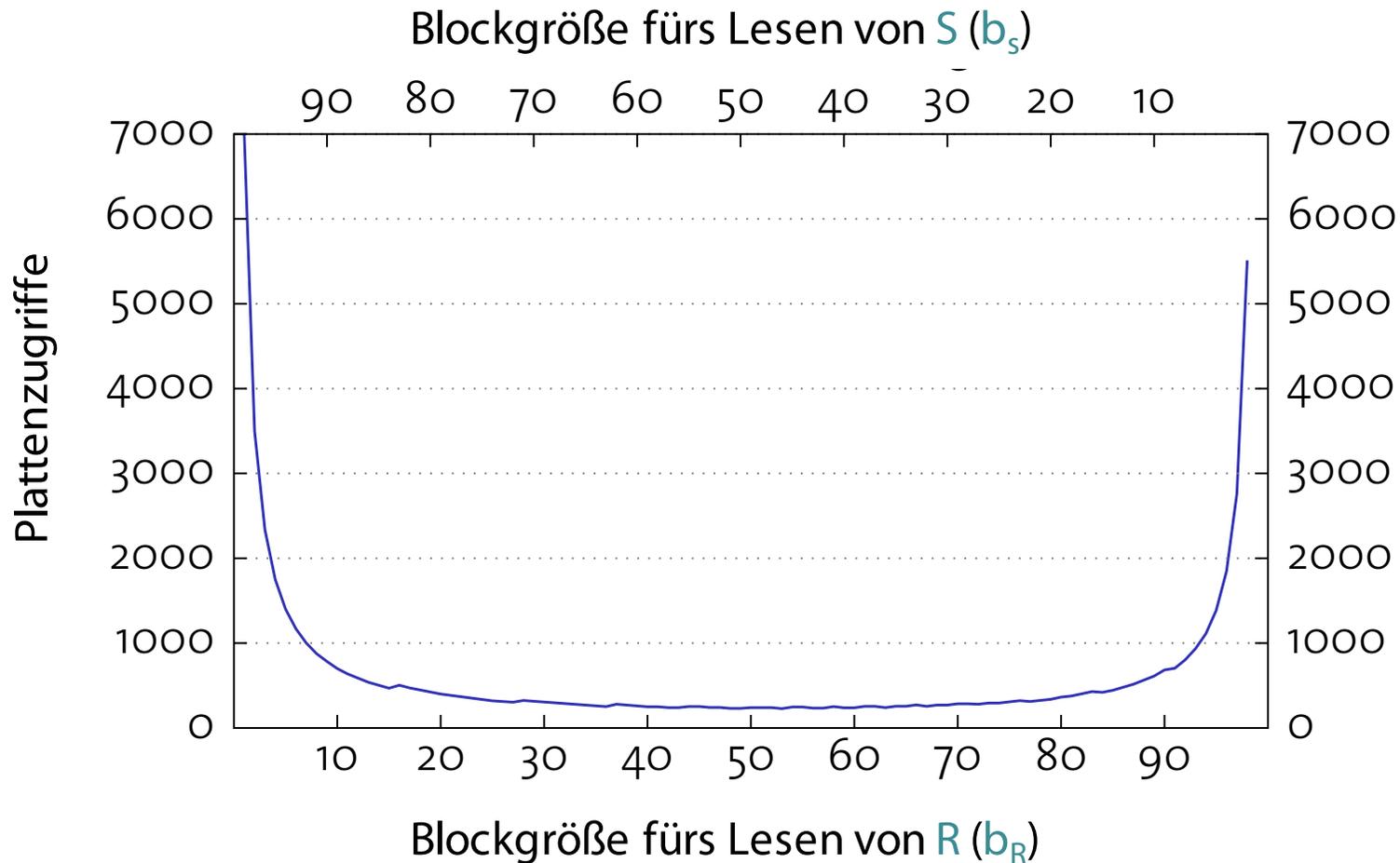
Einsparung von Kosten durch wahlfreien Zugriff durch blockweises Lesen von  $R$  und  $S$  mit  $b_R$  und  $b_S$  vielen Seiten

```
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,  $N_R = 1000$ ,  $N_S = 500$ :



---

# Datenbanken

Prof. Dr. Ralf Möller

**Universität zu Lübeck**

**Institut für Informationssysteme**

Karsten Martiny (Übungen)



# 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

# 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  $\langle r, s \rangle$  tuples to result ;
```

- Funktioniert nur für Equi-Verbunde

# Indexbasierte Verbunde $R \bowtie S$

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

```
1 Function: index_nljoin ( $R, S, p$ )
2 foreach record  $r \in R$  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)

# 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 **ungeclusterten** Index:  $n$  I/O-Operationen
  - Für **geclusterten** Index:  $\lceil n/p_s \rceil$  I/O-Operationen

Wegen 2. und 3. Kosten von der Verbundgröße abhängig

# Zugriffskosten für Index

---

Falls Index ein **B<sup>+</sup>-Baum**:

- Einzelner Indexzugriff benötigt Zugriff auf **h** Indexseiten<sup>1</sup>
- 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

# 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		C	D
"foo"	1	$\bowtie$ $B=C$	1	false
"foo"	2		2	true
"bar"	2		2	false
"baz"	2		3	true
"baf"	4			

- Misch-Verbund **nur für Equi-Verbünde** verwendbar

# Misch-Verbund

```
1 Function: merge_join ( $R, S, \alpha = \beta$ ) //  $\alpha, \beta$ : join columns in  $R, S$ 
2  $r \leftarrow$  position of first tuple in  $R$ ; //  $r, s, s'$ : cursors over  $R, S, S$ 
3  $s \leftarrow$  position of first tuple in  $S$ ;
4 while  $r \neq \text{eof}$  and  $s \neq \text{eof}$  do // eof: end of file marker
5     while  $r.\alpha < s.\beta$  do
6          $\lfloor$  advance  $r$ ;
7     while  $r.\alpha > s.\beta$  do
8          $\lfloor$  advance  $s$ ;
9      $s' \leftarrow s$ ; // Remember current position in  $S$ 
10    while  $r.\alpha = s'.\beta$  do // All  $R$ -tuples with same  $\alpha$  value
11         $s \leftarrow s'$ ; // Rewind  $s$  to  $s'$ 
12        while  $r.\alpha = s.\beta$  do // All  $S$ -tuples with same  $\beta$  value
13             $\lfloor$  append  $\langle r, s \rangle$  to result;
14             $\lfloor$  advance  $s$ ;
15         $\lfloor$  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 **blockweises** 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?

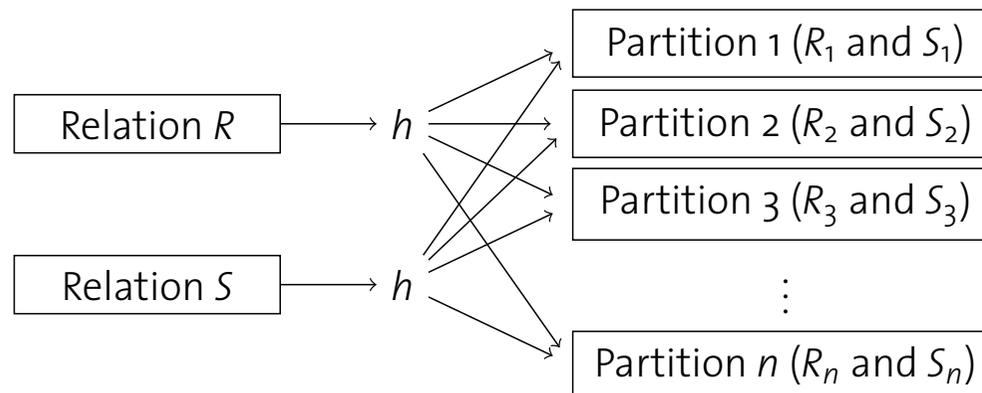
# Lösung

---

- 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  $R_1, \dots, R_n$  und  $S_1, \dots, S_n$  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 korrespondierende Partitionen der Relationen
- 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)

Warum eine andere Hashfunktion?

# Hash-Verbund-Algorithmus

---

```
1 Function: hash_join ( $R, S, \alpha = \beta$ )
2 foreach record  $r \in R$  do
3   └ append  $r$  to partition  $R_{h(r.\alpha)}$ 
4 foreach record  $s \in S$  do
5   └ append  $s$  to partition  $S_{h(s.\beta)}$ 
6 foreach partition  $i \in 1, \dots, n$  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) basierend 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 $\pi$

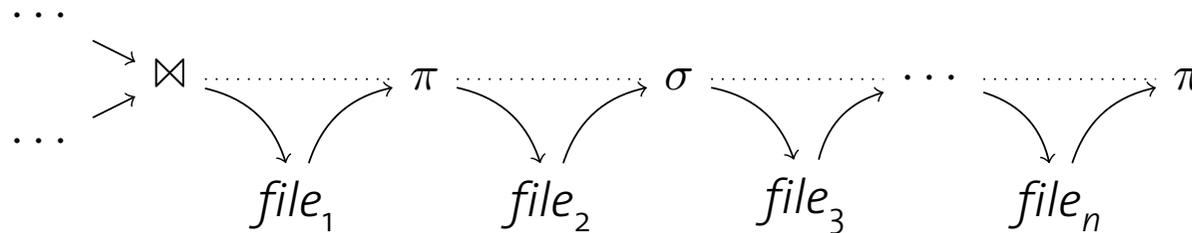
- Implementierung durch
  - a. Entfernen nicht benötigter Spalten
  - b. Eliminierung von Duplikaten
- Die Implementierung von a) bedingt das Ablaufen (scan) aller Datensätze in der Datei, b) siehe oben
- Systeme vermeiden b) sofern möglich (in SQL muss Duplikate-Eliminierung angefordert werden)

## Selektion $\sigma$

- 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

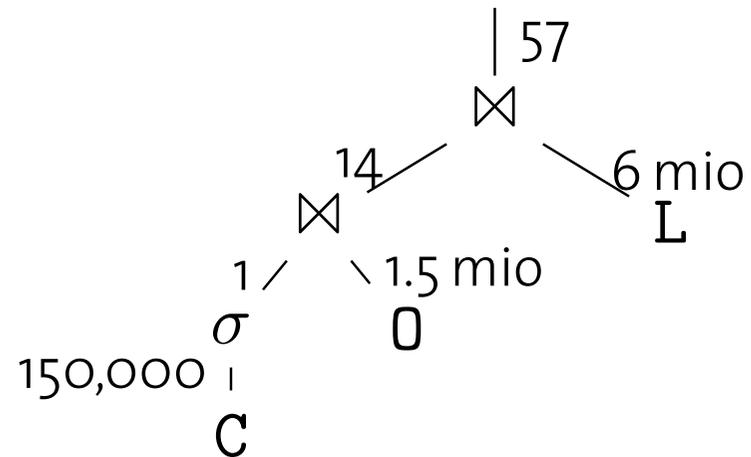
---

- 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

# Auswirkungen auf die Performanz

```
SELECT L.L_PARTKEY, L.L_QUANTITY, L.L_EXTENDEDPRICE
FROM LINEITEM L, ORDERS O, CUSTOMER C
WHERE L.L_ORDERKEY = O.O_ORDERKEY
        AND O.O_CUSTKEY = C.C_CUSTKEY
        AND C.C_NAME = 'IBM Corp.'
```

- Tupelweises Verarbeiten der Relation C
- Sofortiges Weiterleiten nach der Selektion
- Kombiniert mit tupelweisem Verarbeiten der Relationen O und L



# 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() SchlieÙe allozierte Ressourcen
- Zustandsinformation wird Operator-lokal vorgehalten

# Beispiel: Selektion ( $\sigma$ )

---

Eingabe: Relation  $R$ , Prädikat  $p$

1 **Function:** open ()

2  $R.open () ;$

---

1 **Function:** close ()

2  $R.close () ;$

---

1 **Function:** next ()

2 **while** ( $(r \leftarrow R.next ()) \neq eof$ ) **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  $r \leftarrow R.next ()$  ;
```

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

---

```
1 Function: next ()  
2 while ( $r \neq eof$ ) do  
3   while ( $(s \leftarrow S.next ()) \neq eof$ ) do  
4     if  $p(r, s)$  then  
5       return  $\langle r, s \rangle$  ;  
6   S.close () ;  
7   S.open () ;  
8    $r \leftarrow R.next()$  ;  
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 gesamte 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...

