Datenbanken

Anfrageverarbeitung

Dr. Özgür Özçep

Prof. Dr. Ralf Möller

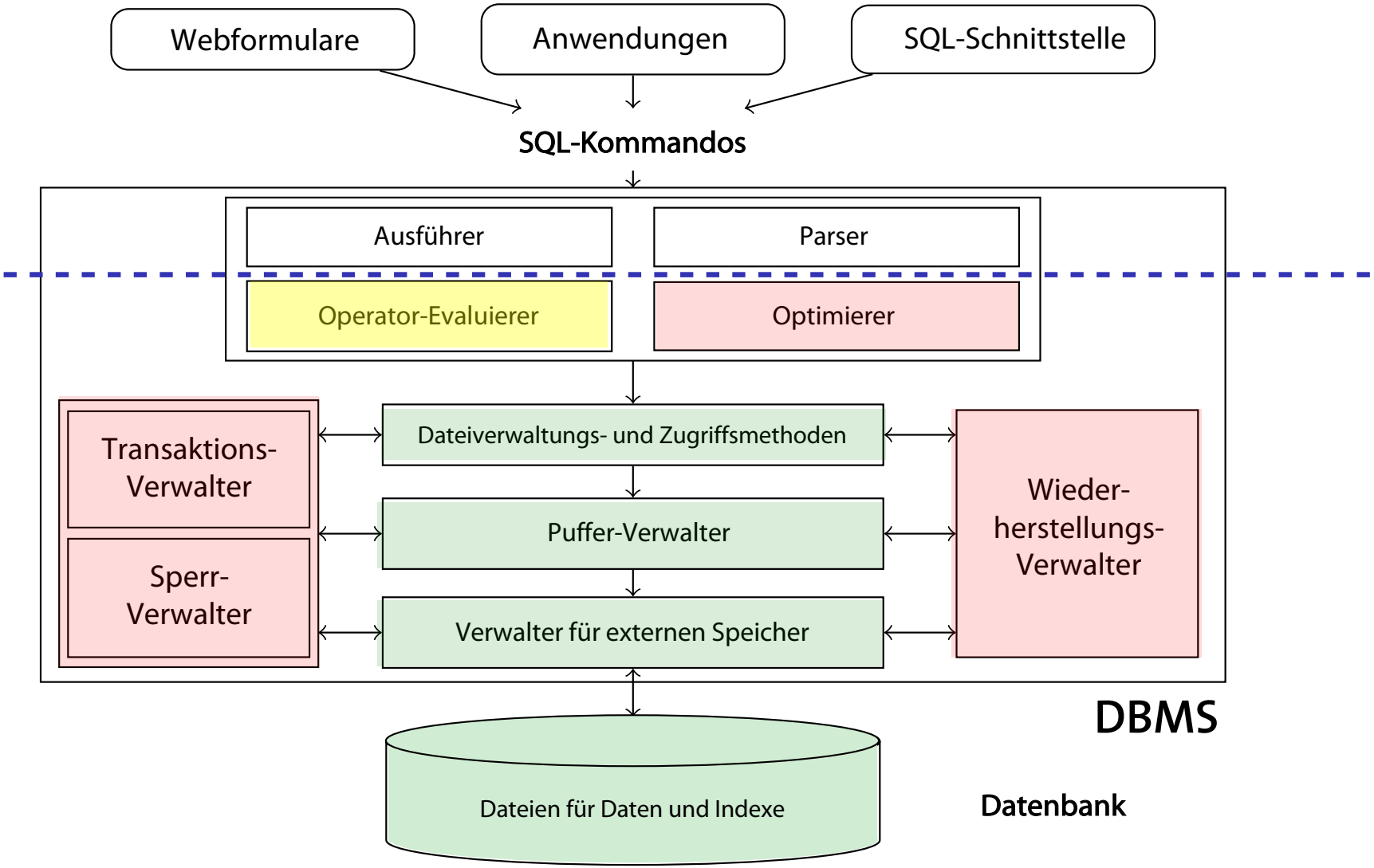
Universität zu Lübeck

Institut für Informationssysteme

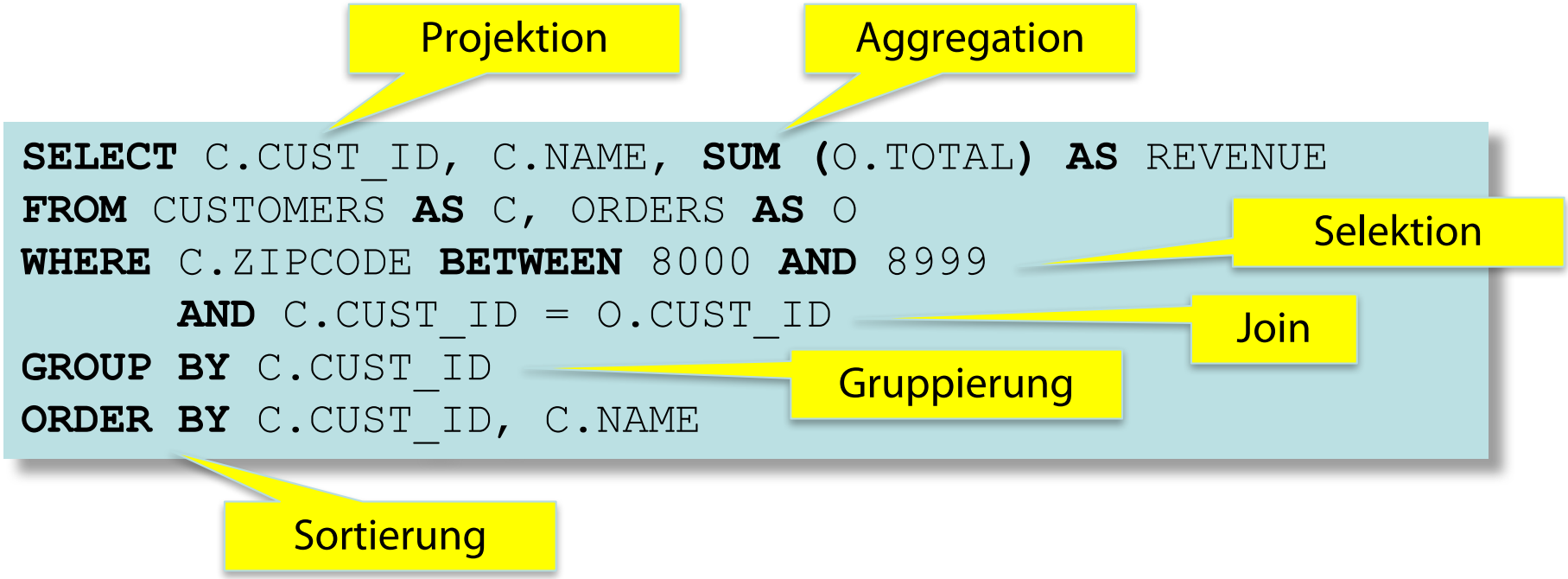
Felix Kuhr (Übungen)

und studentische Tutoren

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)?

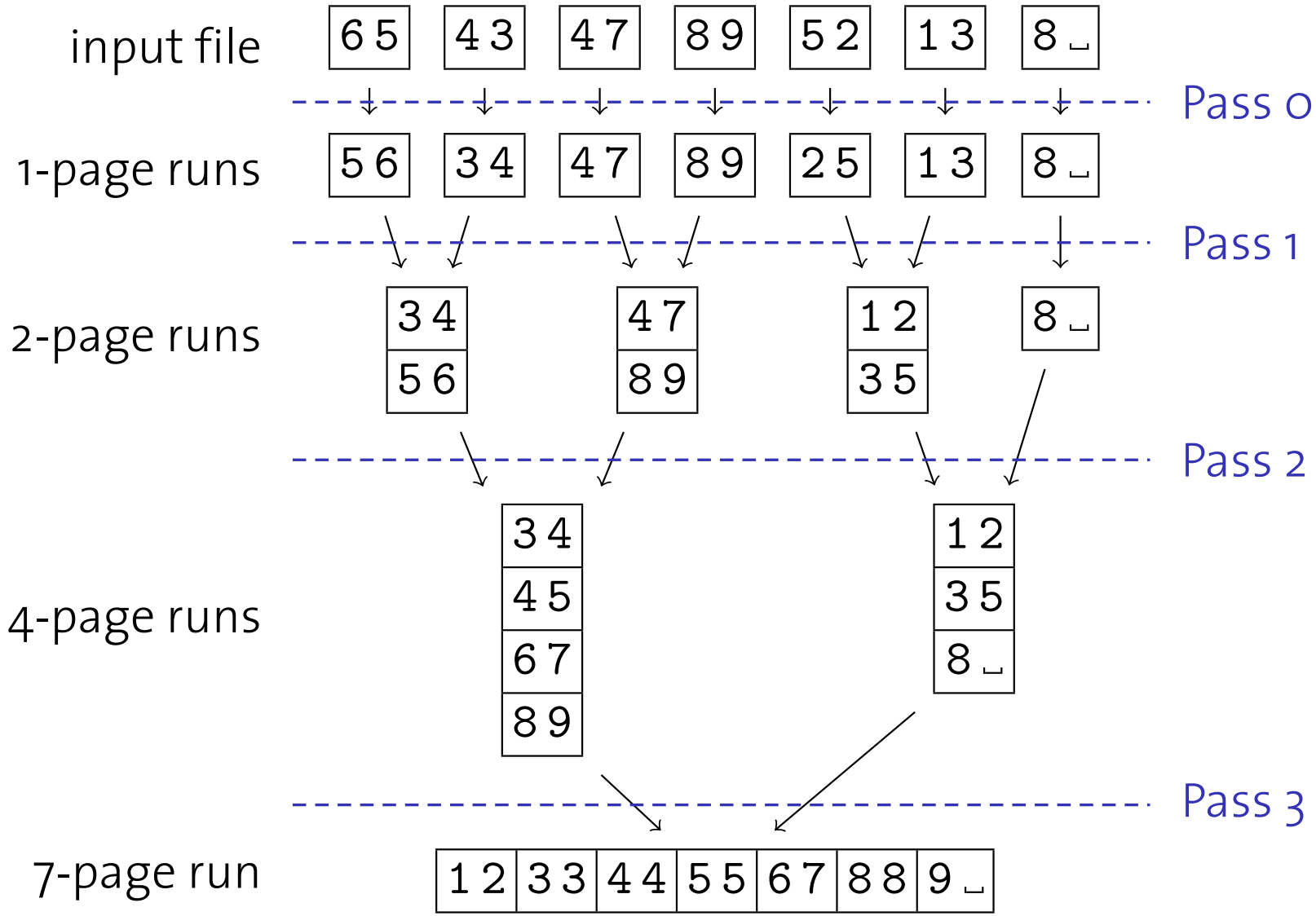
Zwei-Wege-Mischsortieren (Merge Sort)

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

- Sortierung von N Seiten in mehreren Durchgängen

Beispiel

n-page run = Datei aus n Seiten



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

  $\longrightarrow 2 \cdot N$ 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 bei einer Travelstar 7k200?

- 8KB pro Seite
- Suchzeit $t_s = 10$ ms
- Rotationsverzögerung $t_r = 4{,}17$ ms
- Transferzeit $t_{tr} = 0{,}16$ms für 8KB-Seite

Lösung

≈ 7d

- Wahlfreier Zugriff mit Zugriffszeit = 10ms + 4,17ms + 0,16ms = 14,33 ms
- Anzahl Seiten = 8GB / 8KB = 10^6
- I/O-Operationen = $2 \cdot 10^6(1 + \lceil 6 \log_2(10) \rceil) = 42 * 10^6$
- Zeit = $42*10^6$ * 14,33ms ≈ 7d

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 Runs 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?
- Antwort: Chunks von B Seiten sequenziell gelesen!

Aufgabe:

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

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

≈ 3d

- Durchgänge = $1 + \lceil \log_2 \lceil N/B \rceil \rceil =$
  $1 + \lceil \log_2 \lceil 10^6/10^3 \rceil \rceil = 1 + \lceil \log_2 10^3 \rceil = 11$
- Mergedurchgänge = 10
- Im ersten Durchgang kann Seek und Rotationszeit vernachlässigt werden
- Damit bleibt nur für die Mergedurchgänge eine Seek und Rotationszeit von insgesamt $2 * 10^6 * 10 * 14{,}17\text{ms} \approx 3\text{d}$
- Gesamte Transferzeit ergibt sich zu
- $2 * 10^6 * 11 * 0{,}16\text{ms} \approx 1\text{h}$

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 $r_1 \ldots r_{B-1}$ from Pass $n-1$ for reading.
2. **Merge** records from $r_1 \ldots r_{B-1}$, 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?
- Antwort: In den Mergephasen muss ja entschieden werden, welcher der minimalen Elemente auf Outputbuffer gehen. Daher random access.

Aufgabe:

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

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

16h

- Durchgänge = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil =$
  $1 + \lceil \log_{999} \lceil 10^6 / 10^3 \rceil \rceil = 1 + \lceil \log_{999} 10^3 \rceil = 3$
- Mergedurchgänge = 2
- Im ersten Durchgang kann Seek und Rotationszeit vernachlässigt werden
- Damit bleibt nur für die Mergedurchgänge eine Seek und Rotationszeit von insgesamt $2 * 10^6 * 2 * 14{,}17\text{ms} \approx 16$ h
- Gesamte Transferzeit ergibt sich zu
- $2 * 10^6 * 3 * 0{,}16\text{ms} \approx 1\text{min}$
- Zusammen

≈ 16h

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 pro Seite
- 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 ?

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

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

Anzahl der I/O-Operationen:

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

Lösung

Mit blockweisem I/O (B=1000, Blöcke mit b=32 Seiten):
ca. $10 \cdot 31250$ Plattenzugriffe mit je 27.42 ms

$\approx$ 2,38 h

(Herleitung: Übungsaufgabe 4, Blatt 8)

Hauptspeicher 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: $1 + \lceil \log_{\lceil B/b \rceil - 1} \lceil N / \lceil B/b \rceil \rceil \rceil$

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

Lösung

Platzbedarf: 256MB

(Herleitung in Übungsaufgabe 5, Blatt 8)

Bisher nur IO-Zeit betrachtet

- Auswahl des nächsten Datensatzes für den Output unter B-1 (oder B/b -1) Eingabeläufen kann CPU-intensiv sein (B-2 Vergleiche)
- Beispiel: B-1 = 4, < -Ordnung

087 503 504 . . . (page from Run 1)
170 908 994 ... (page from Run 2)
154 426 653 ....
612 613 700 ... (page Run B-1)
(out buffer)

Bisher nur IO-Zeit betrachtet

- Auswahl des nächsten Datensatzes für den Output unter B-1 (oder B/b -1) Eingabeläufen kann CPU-intensiv sein (B-2 Vergleiche)
- Beispiel: B-1 = 4, < -Ordnung

087 503 504 . . .	(page from Run 1)
170 908 994 ...	(page from Run 2)
154 426 653	
612 613 700 ...	(page Run B-1)
	(out buffer)

503 504 . . .	(page from Run 1)
170 908 994 ...	(page from Run 2)
154 426 653	
612 613 700 ...	(page Run B-1)
087	(out buffer)

Bisher nur IO-Zeit betrachtet

- Auswahl des nächsten Datensatzes für den Output unter B-1 (oder B/b -1) Eingabeläufen kann CPU-intensiv sein (B-2 Vergleiche)
- Beispiel: B-1 = 4, < -Ordnung

087 503 504 . . . (page from Run 1)
170 908 994 ... (page from Run 2)
154 426 653 ....
612 613 700 ... (page Run B-1)
(out buffer)

503 504 . . . (page from Run 1)
170 908 994 ... (page from Run 2)
426 653 ....
612 613 700 ... (page Run B-1)
087 154 (out buffer)

503 504 . . . (page from Run 1)
170 908 994 ... (page from Run 2)
154 426 653 ....
612 613 700 ... (page Run B-1)
087 (out buffer)

Bisher nur IO-Zeit betrachtet

- Auswahl des nächsten Datensatzes für den Output unter B-1 (oder B/b -1) Eingabeläufen kann CPU-intensiv sein (B-2 Vergleiche)
- Beispiel: B-1 = 4, < -Ordnung

087 503 504 . . . (page from Run 1)
170 908 994 ... (page from Run 2)
154 426 653 ....
612 613 700 ... (page Run B-1)
(out buffer)

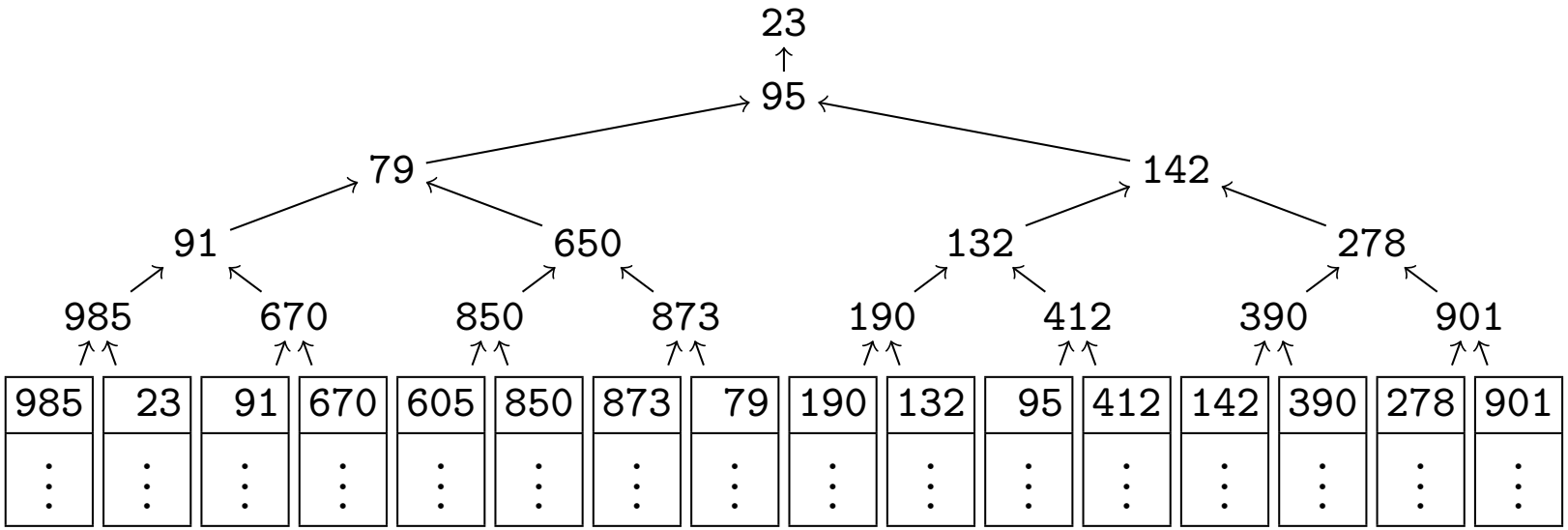
503 504 . . . (page from Run 1)
170 908 994 ... (page from Run 2)
426 653 ....
612 613 700 ... (page Run B-1)
087 154 (out buffer)

503 504 . . . (page from Run 1)
170 908 994 ... (page from Run 2)
154 426 653 ....
612 613 700 ... (page Run B-1)
087 (out buffer)

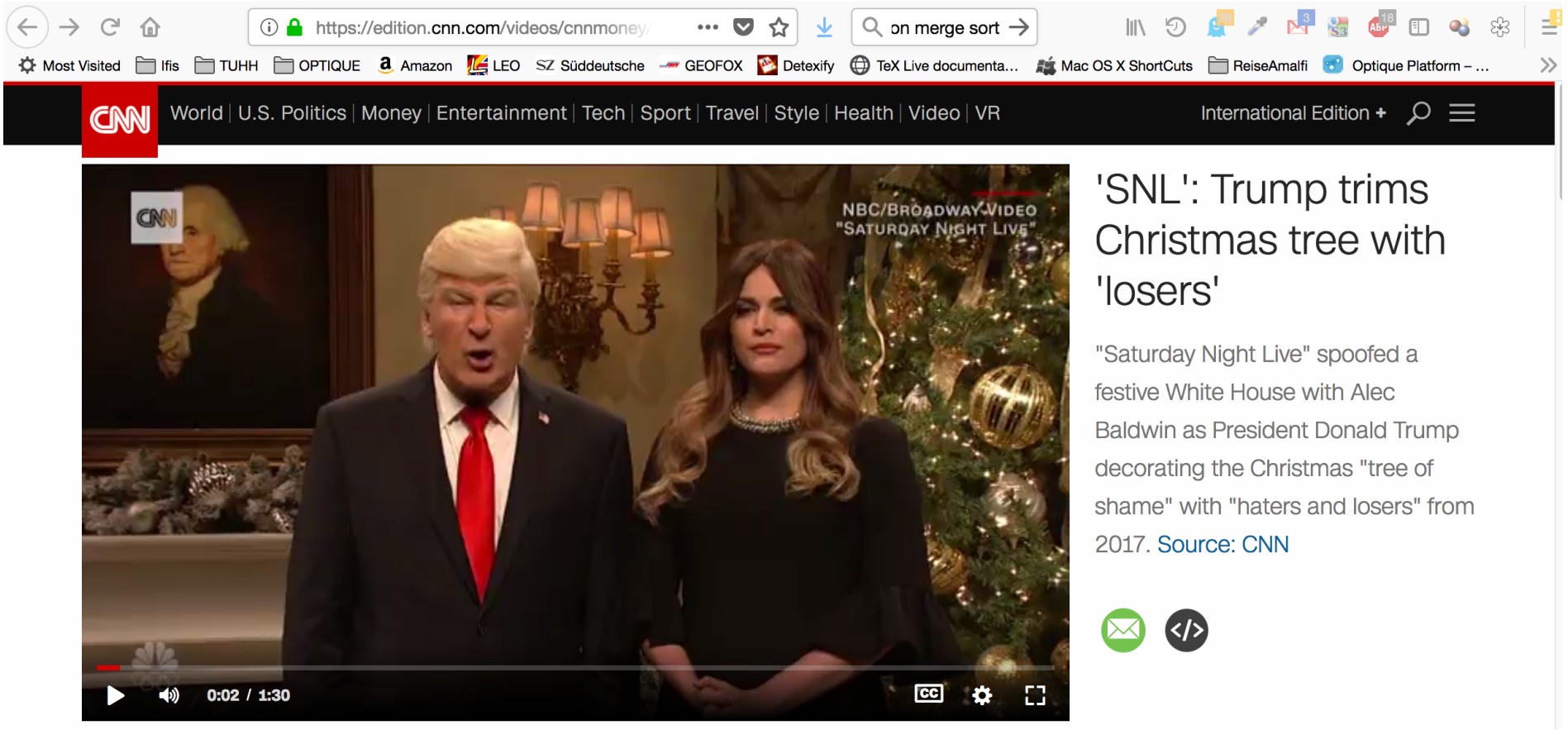
503 504 . . . (page from Run 1)
908 994 ... (page from Run 2)
426 653 ....
612 613 700 ... (page Run B-1)
087 154 170 (out buffer)

Bisher nur IO-Zeit betrachtet

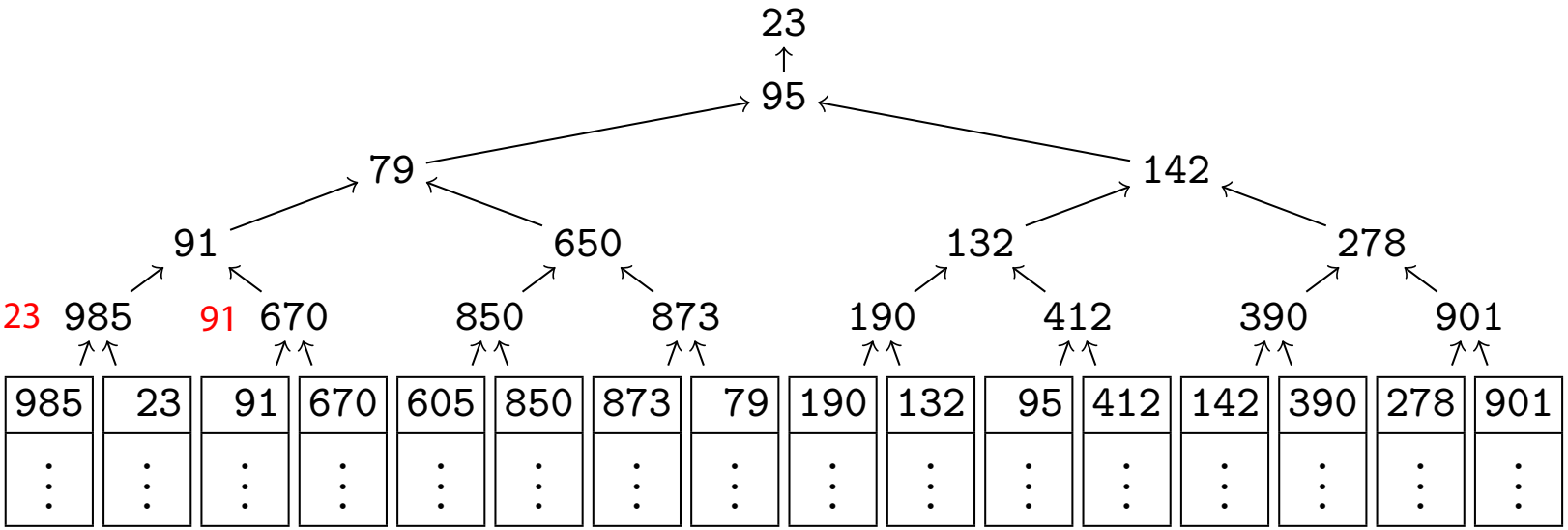
- Verwende Auswahlbaum zur Kostenreduktion („Tree of Losers“)
- Reduktion der Vergleiche auf $\log_2 (B-1)$ bzw. $\log_2 (B/b-1)$



D. Knuth, The Art of Computer Programming,vol.3, 1973

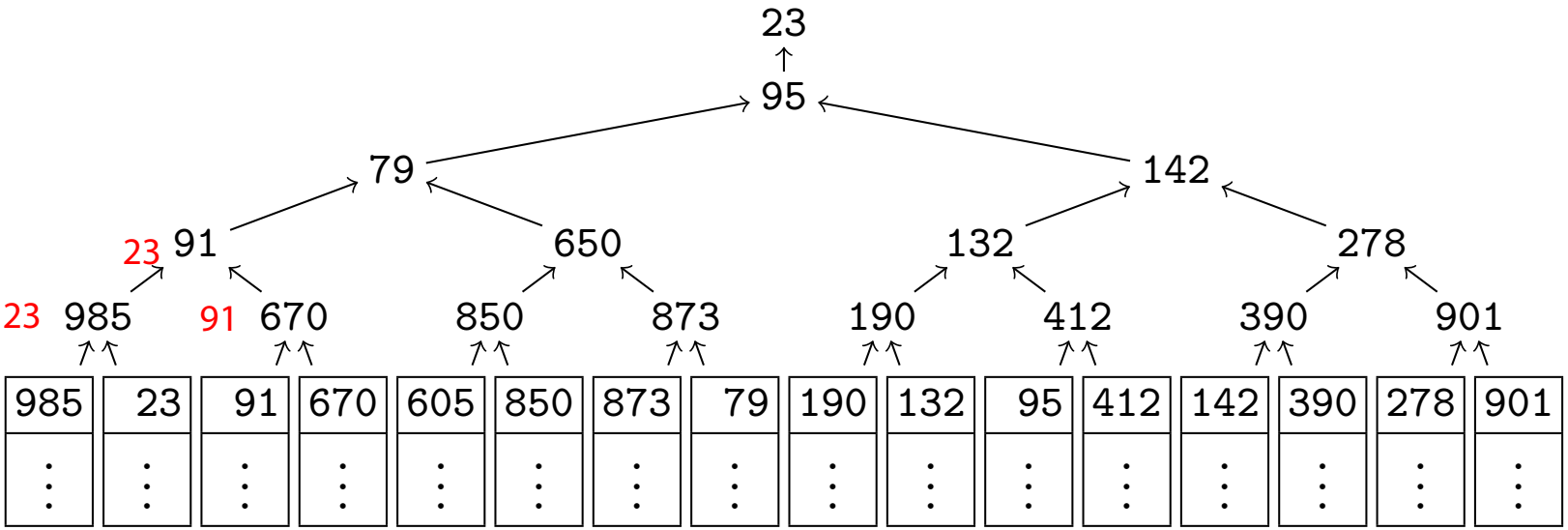

https://edition.cnn.com/videos/cnnmoney/
on merge sort
Most Visited
Ifis
TUHH
OPTIQUE
Amazon
LEO
Süddeutsche
GEOFOX
Detexify
TeX Live documenta...
Mac OS X ShortCuts
ReiseAmalfi
Optique Platform – ...
CNN
World | U.S. Politics | Money | Entertainment | Tech | Sport | Travel | Style | Health | Video | VR
International Edition +
NBC/BROADWAY VIDEO
"SATURDAY NIGHT LIVE"
0:02 / 1:30
'SNL': Trump trims Christmas tree with 'losers'
"Saturday Night Live" spoofed a festive White House with Alec Baldwin as President Donald Trump decorating the Christmas "tree of shame" with "haters and losers" from 2017. Source: CNN

Tree of Losers (and hidden winners)



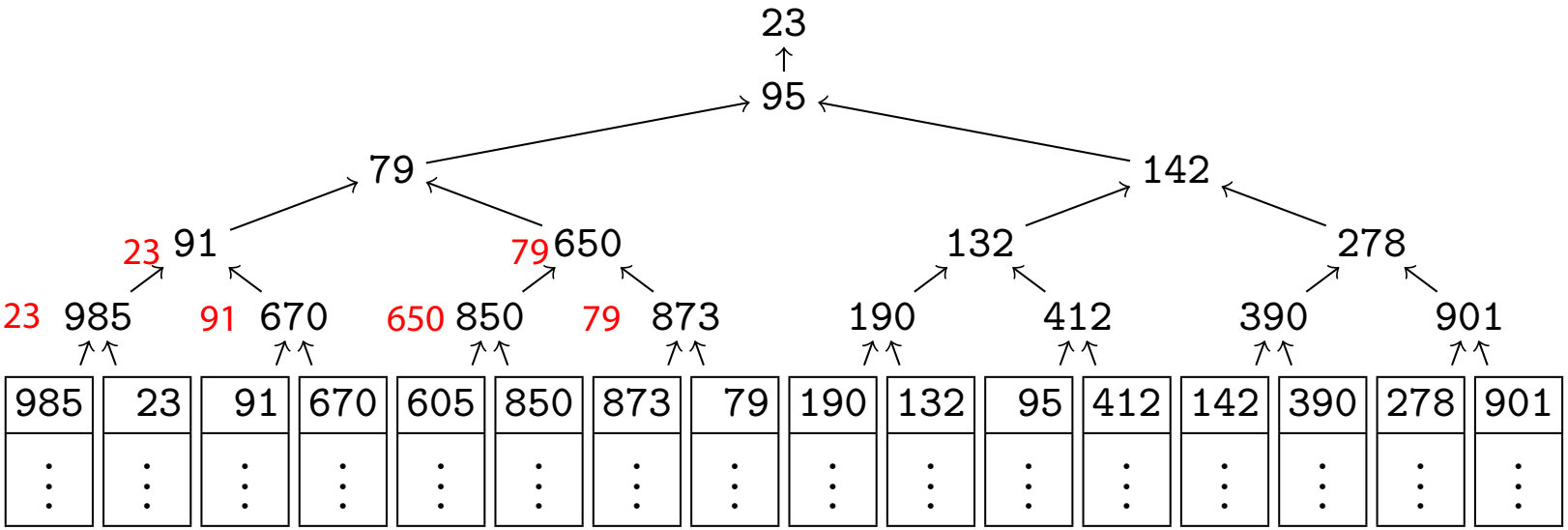
D. Knuth, The Art of Computer Programming,vol.3, 1973

Tree of Losers (and hidden winners)



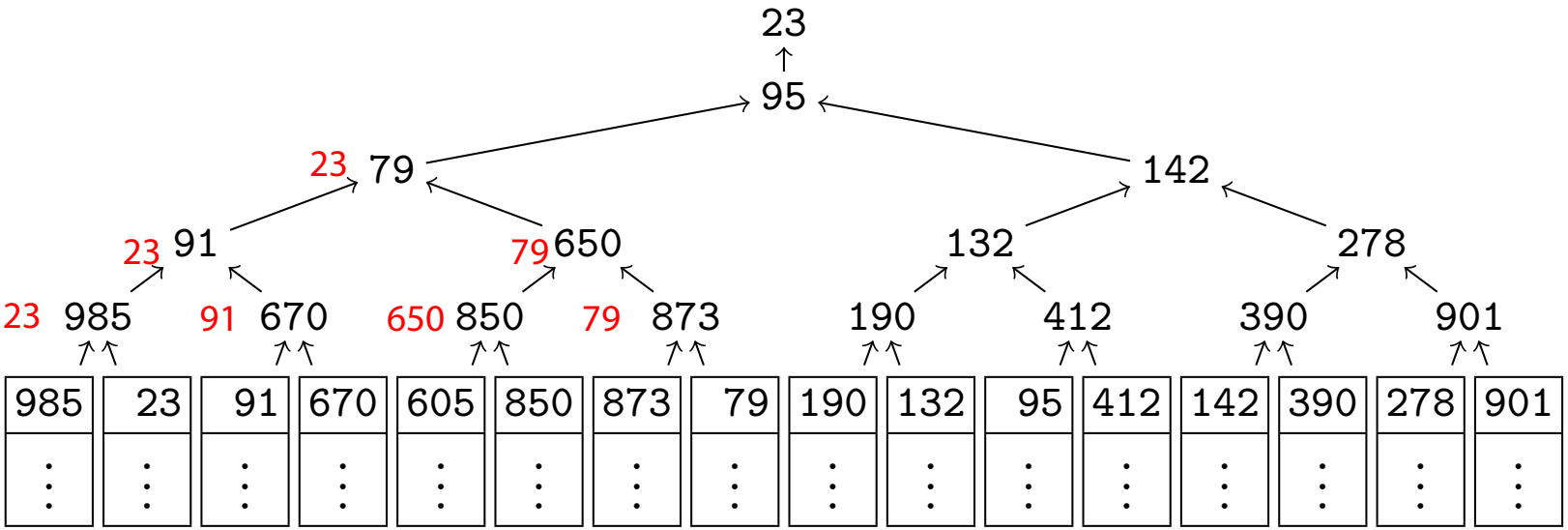
D. Knuth, The Art of Computer Programming,vol.3, 1973

Tree of Losers (and hidden winners)



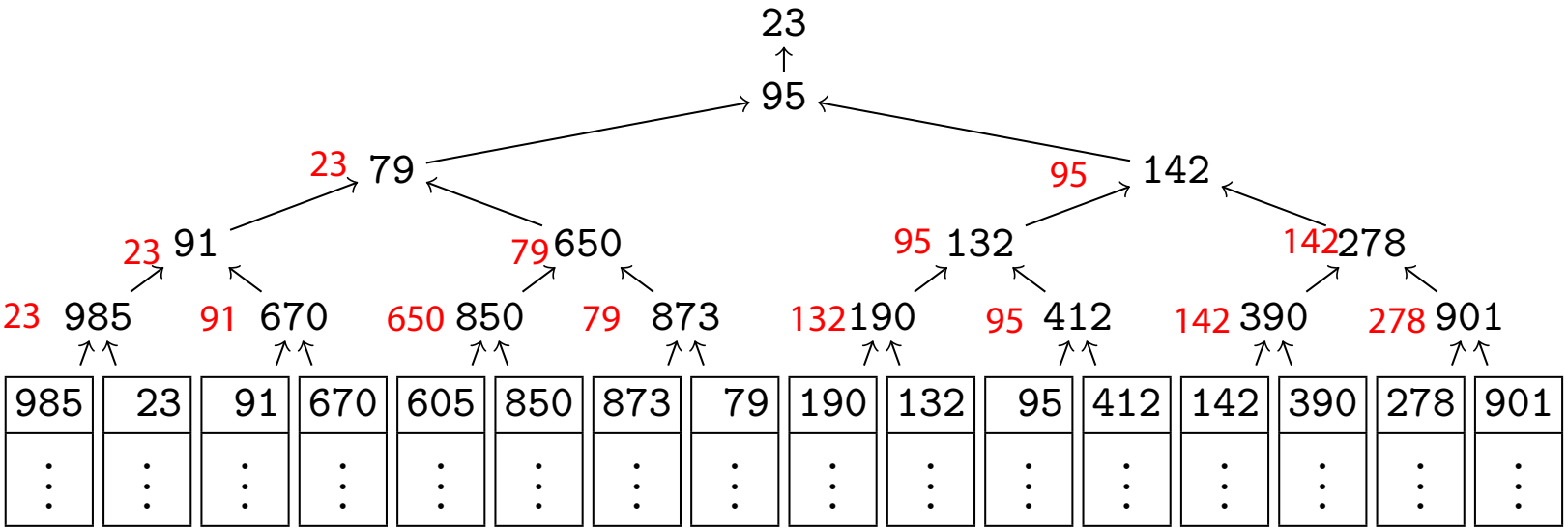
D. Knuth, The Art of Computer Programming,vol.3, 1973

Tree of Losers (and hidden winners)



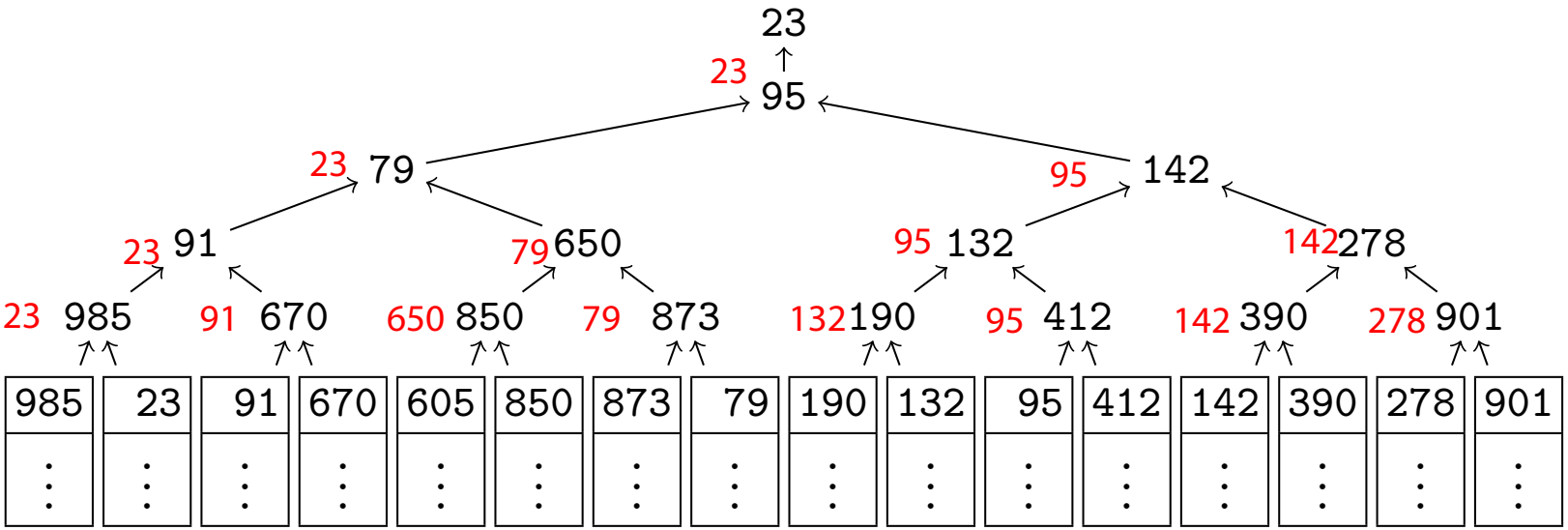
D. Knuth, The Art of Computer Programming,vol.3, 1973

Tree of Losers (and hidden winners)



D. Knuth, The Art of Computer Programming,vol.3, 1973

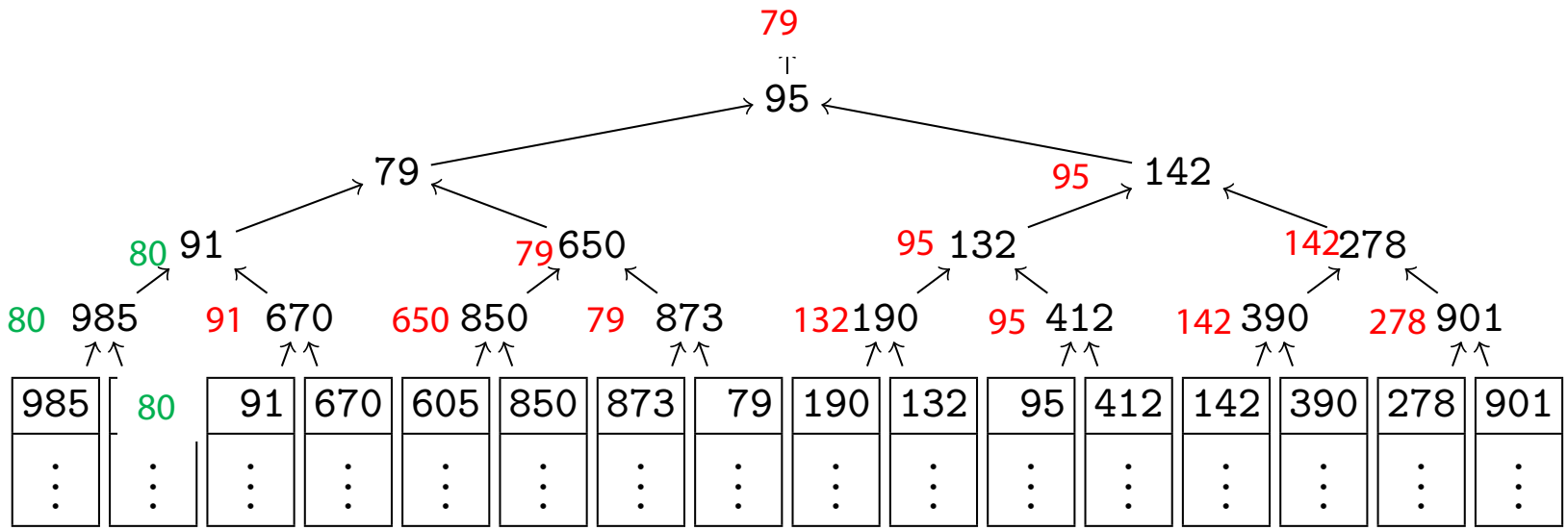
Tree of Losers and hidden winners (not stored)



D. Knuth, The Art of Computer Programming,vol.3, 1973

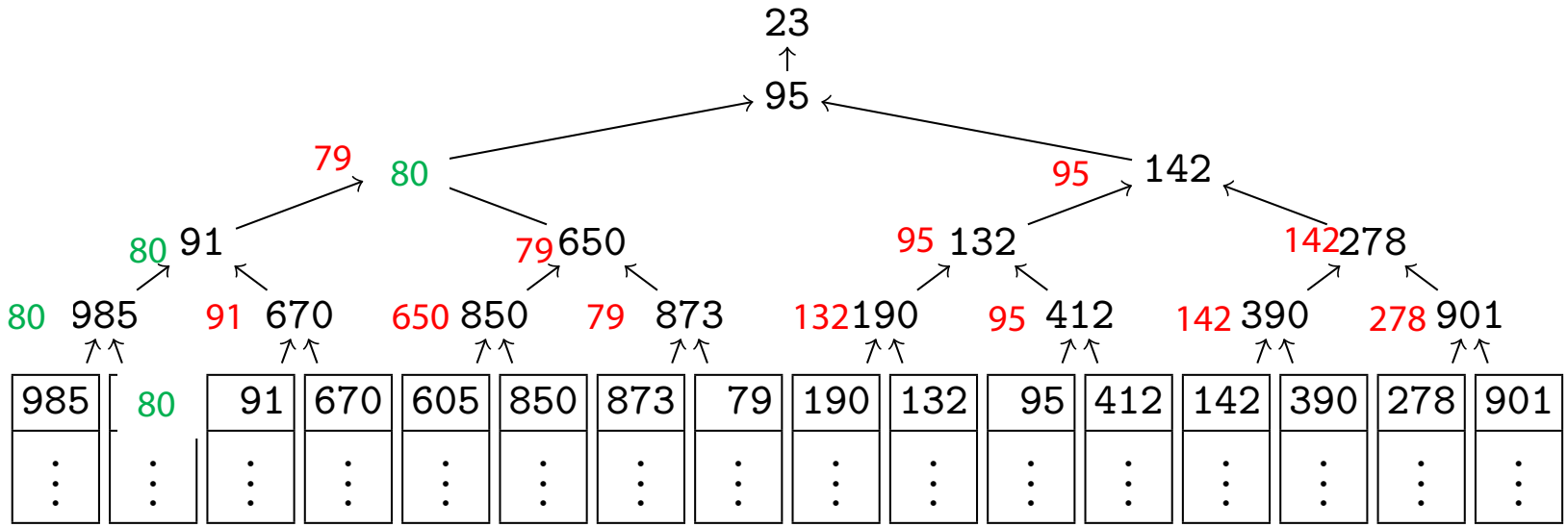
Nächstes Element in Run vom Winner

Muss nicht die hidden Gewinner betrachten (Referenz: Self->parent-> sibling)
Sondern nur Vaterknoten (self-> parent)



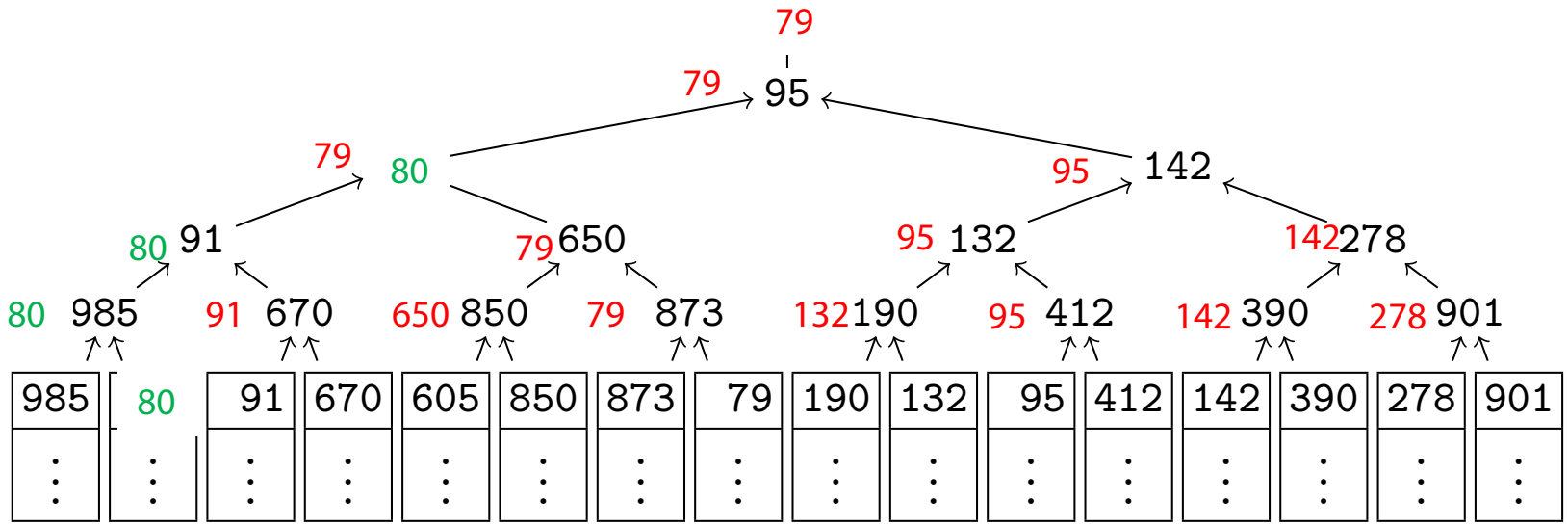
Nächstes Element in Run vom Winner

Muss nicht die hidden Gewinner betrachten (Referenz: Self->parent-> sibling)
Sondern nur Vaterknoten (self-> parent)



Nächstes Element in Run vom Winner

Muss nicht die hidden Gewinner betrachten (Referenz: Self->parent-> sibling)
Sondern nur Vaterknoten (self-> parent)



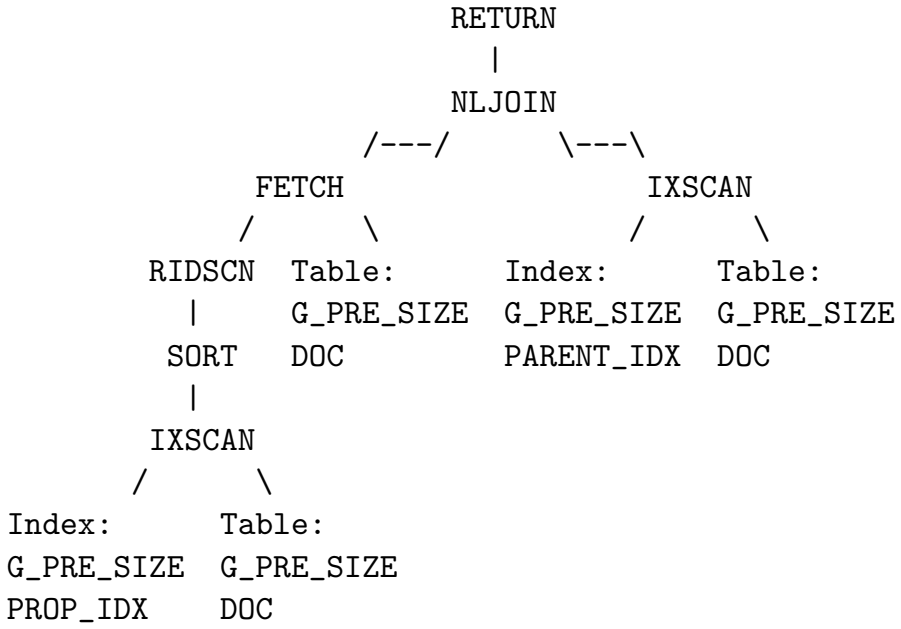
Warum nicht ein „tree of winners“?

- Tree of losers lässt sich leicht updaten: Pfad von neuem Element zur Wurzel ohne auf Geschwisterknoten zu schauen.

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

Ausführungspläne



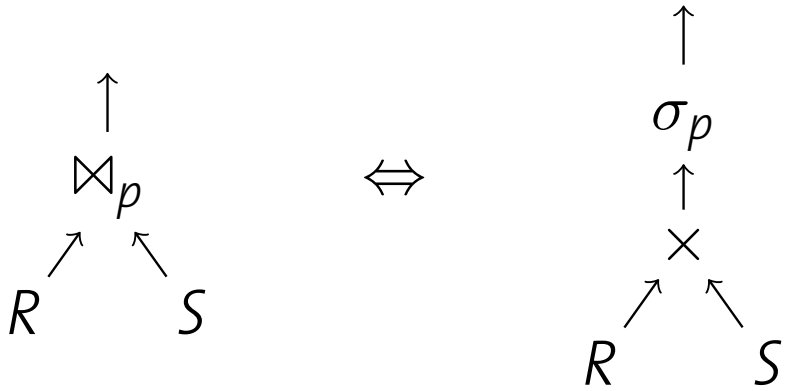
Ausführungsplan (Skizze, DB2)

- 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

Als nächstes betrachten wir Verbundoperatoren

Verbundoperator (Join) R⋈S

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 × S
2. Wähle die Datensätze, die p erfüllen

Ineffizienz aus Schritt 1 kann überwunden werden (Größe des Zwischenresultats: |R| × |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:

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

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 sequenzieller Ordnung verarbeitet werden.

Blockweiser Verbund mit Schleifen

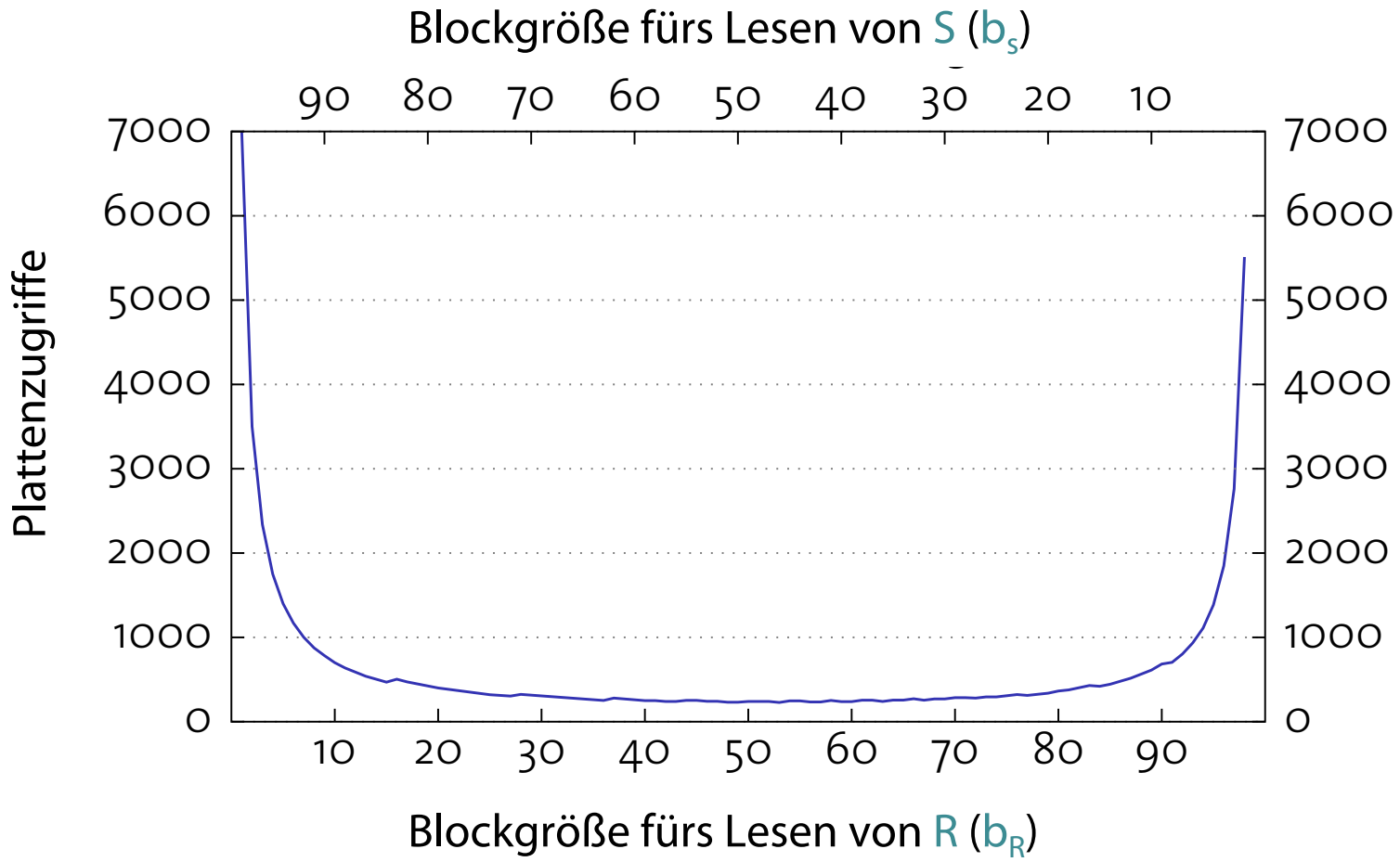
Einsparung von Kosten durch wahlfreien Zugriff durch blockweises Lesen von R und S mit b_R und b_S vielen Seiten

```
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 (einmal) 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 Blöcken 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 $R \bowtie S$

Verwendung eines vorhanden Index für die innere Relation S (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

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

Indexbasierte Verbunde $R \bowtie S$

- Index muss verträglich mit der Verbundbedingung sein
  - Hash-Index (nur für Gleichheitsprädikate)
  - Abdeckung einer Konjunktion von Atomen durch zusammengesetzte Schlüssel
  - Manchmal auch nur partielle Abdeckung nützlich (wenn z.B. jeweils ein Konjunkt mit einem jeweils anderen Index verträglich)

- Argumente heißen „sargable“ (SARG= search argument)

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

I/O-Verhalten

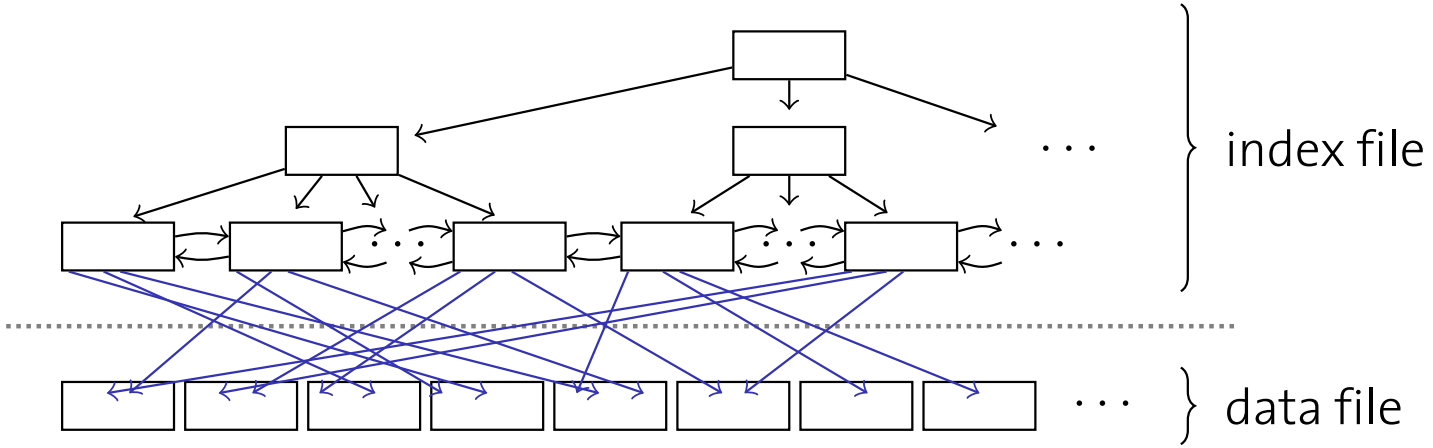
Für jeden Datensatz in R verwende Index zum Auffinden von korrespondierenden S-Tupeln. Für jedes R-Tupel sind folgende Kosten einzukalkulieren:

1. Zugriffskosten für den Index zum Auffinden des ersten Eintrags: N_{idx} I/O-Operationen
2. Entlanglaufen an den Indexwerten (Scan), um passende n 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

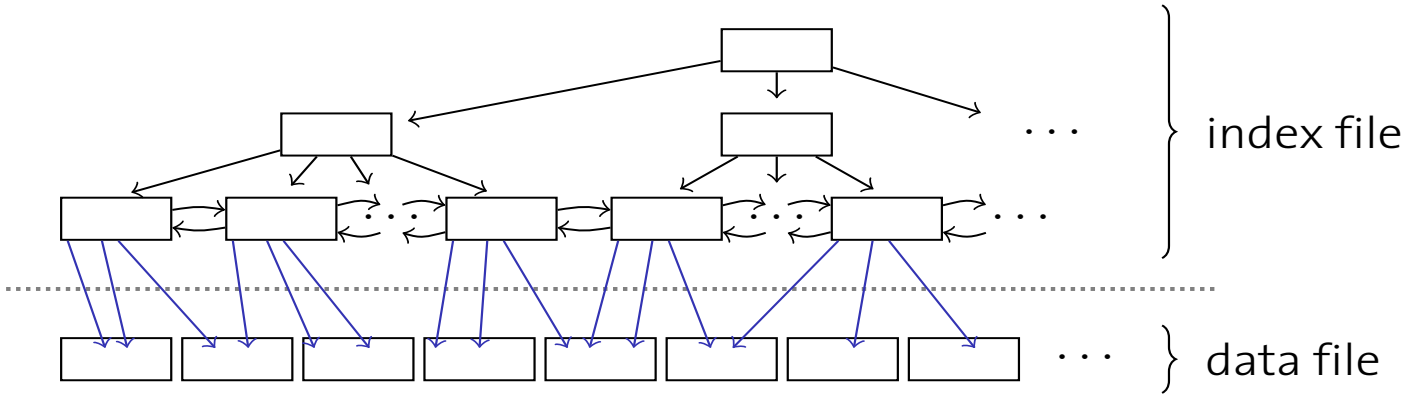
Wegen 2. und 3. Kosten von der Verbundgröße abhängig

Erinnerung: Clustering

Nicht geclustert



Geclustert



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 blockweises I/O treten fast immer sequenzielle 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?

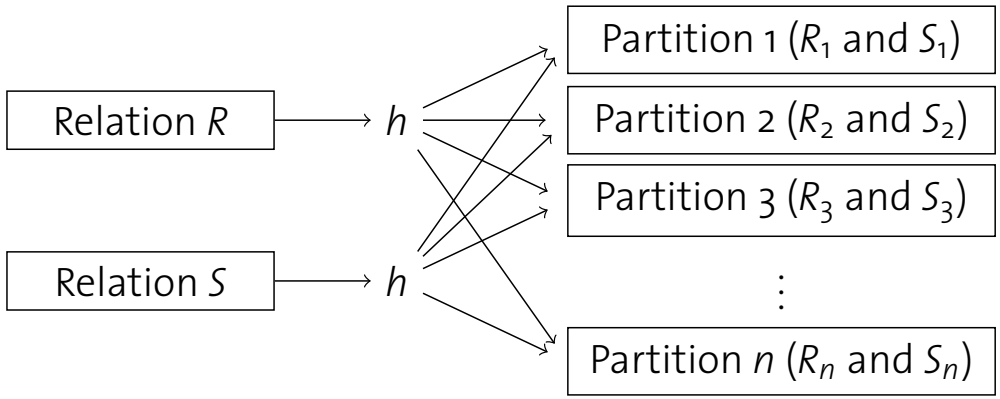
Lösung

- Wenn alle 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

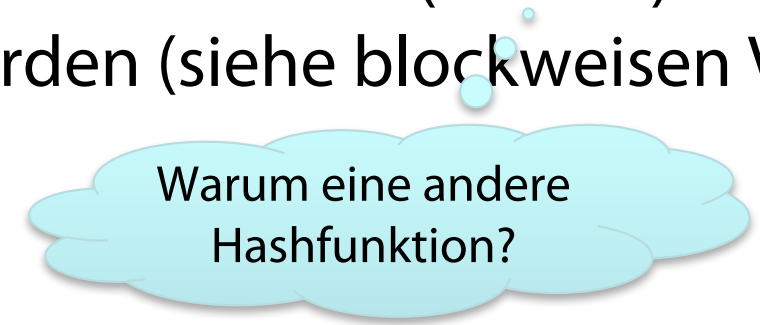
- Sortierung bringt korrespondierende Tupel in eine „räumliche Nähe“, so dass eine effiziente Verarbeitung möglich ist
- Ein ähnlicher Effekt erreichbar mit Hash-Verfahren
- Zerlege R und S in Teilrelationen R_1, ..., R_n und S_1,...,S_n mit der gleichen Hashfunktion (angewendet auf die Verbundattribute)



- $R_i \bowtie S_j = \varnothing$ für alle $i \neq j$

Hash-Verbund

- Durch Partitionierung werden kleine Relationen R_i und S_i geschaffen
- Korrespondierende Datensätze kommen garantiert in korrespondierende Partitionen der Relationen
- Es muss $R_i \bowtie S_i$ (für alle 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 ;
```

I/O-Aufwand wenn $|R \bowtie S|$ "klein"

$3 \cdot (N_R + N_S)$

(Lesen und Schreiben beider Relationen für Partitionierung + Lesen beider Relationen für Join)

Gruppierung und Duplikate-Elimination

- Herausforderung: Finde identische Datensätze in einer Datei
- Ähnlichkeiten zum Eigenverbund (self-join) basierend auf allen Spalten der Relation
- Duplikate-Elimination oder Gruppierung mit Hash-Verbund oder Sortierung

Andere Anfrage-Operatoren

Projektion π

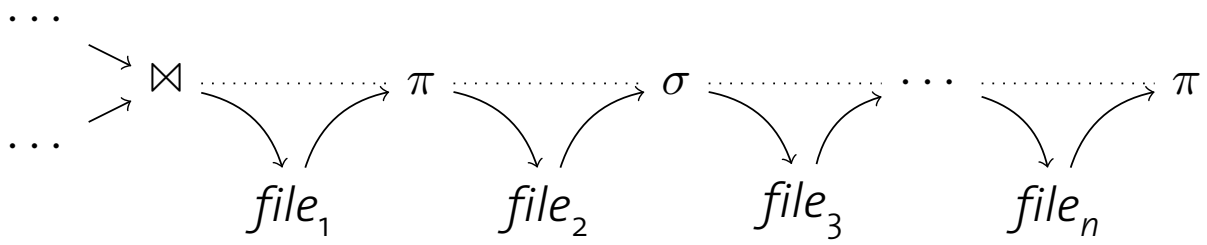
- Implementierung durch
  a. Entfernen nicht benötigter Spalten
  b. Eliminierung von Duplikaten
- Die Implementierung von a) bedingt das Ablaufen (scan) aller Datensätze in der Datei, b) siehe oben
- Systeme vermeiden b) sofern möglich (in SQL muss Duplikate-Eliminierung angefordert werden)

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

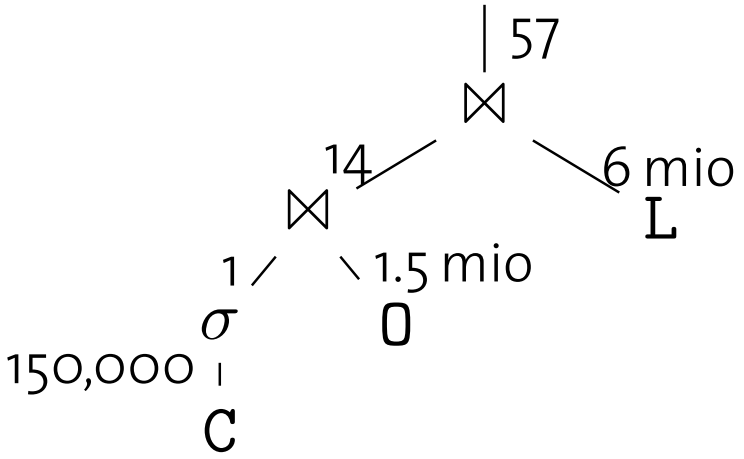
- Alternativ könnte jeder Operator seine Ergebnisse direkt an den nachfolgenden senden, ohne die Ergebnisse erst auf die Platte zu schreiben
- Ergebnisse werden so früh wie möglich weitergereicht und verarbeitet (Pipeline-Prinzip)
- Granularität ist bedeutsam:
  - Kleinere Brocken reduzieren Antwortzeit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

- Siehe auch Gebiet der Stromverarbeitung

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() Schließe allozierte Ressourcen
- Zustandsinformation wird Operator-lokal vorgehalten

Goetz Graefe. Volcano—An Extensible 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?

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)

Rekursive Anfragen

Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW
LTSW	Leitung SW	**NULL**
PERS	Personal	**NULL**

Abteilungen

```
create recursive view Unterabteilungen
  select r.kurz, r.oberabt
  from Abteilungen r
union
  select u.kurz, o.oberabt
  from Abteilungen o,
       Unterabteilungen u
  where o.kurz = u.oberabt
```

PostgreSQL-Syntax

- Beim Start der Rekursion enthält „Unterabteilungen“ nur die Tupel der ersten Teilanfrage.
- Tupel, die sich durch den Join in der zweiten Teilanfrage ergeben, werden der Extension von „Unterabteilungen“ für die nächste Iteration hinzugefügt.
- Abbruch der Rekursion, sobald die zweite Teilanfrage bei Verwendung der Ergebnisse aus der vorigen Iteration keine zusätzlichen Ergebnistupel mehr liefert ⟶ Fixpunkt.

Rekursive Anfragen in Datalog

Kurz	*Name*	*Oberabt*
MFSW	Mainframe SW	LTSW
UXSW	Unix SW	LTSW
PCSW	PC SW	LTSW
LTSW	Leitung SW	**NULL**
PERS	Personal	**NULL**

Abteilungen

```
create recursive view Unterabteilungen
  select r.kurz, r.oberabt
  from Abteilungen r
union
  select u.kurz, o.oberabt
  from Abteilungen o,
       Unterabteilungen u
  where o.kurz = u.oberabt
```

PostgreSQL-Syntax

$$\forall\, x,y\ (\ \text{Unterabteilung}(x,y) \leftarrow \exists\, w\ \text{Abteilung}(x,w,y)\ )$$

$$\forall\ x,y,z\ (\ \text{Unterabteilung}(x,y) \leftarrow \exists\, w\ \exists\, z(\text{Abteilung}(z,w,y) \wedge \text{Unterabteilung}(x,z))\ )$$

Datalog: Rekursive Anfragen

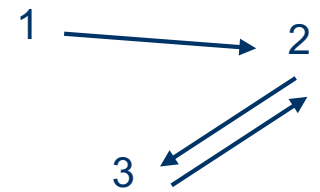
Einige Darstellungen wurden mit Änderungen übernommen aus einer Präsentation von Stephanie Scherzinger

$$\forall\ x,y\ (\ T(x,y) \leftarrow G(x,y)\ )$$

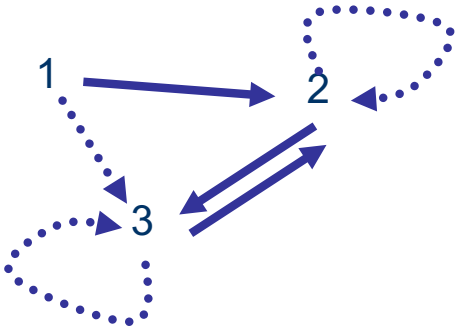
$$\forall\ x,y,z\ (\ T(x,y) \leftarrow (\ G(x,z) \wedge T(z,y)\ )\ )$$

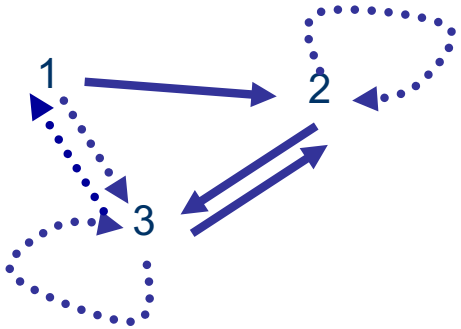
$$G(\ 1,\ 2\ ),\ G(\ 2,\ 3\ ),\ G(\ 3,\ 2\ )$$

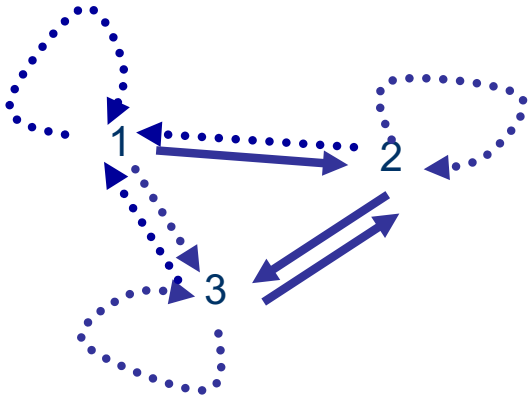
G:



Mögliche Lösungen:







Herve Gallaire und Jack Minker *Logic and Data Bases*. Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse in „Advances in Data Base Theory". Plenum Press, New York **1978**

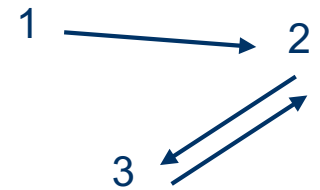
Minimale-Modell-Semantik

$\forall$ x,y (T(x,y) $\leftarrow$ G(x,y))

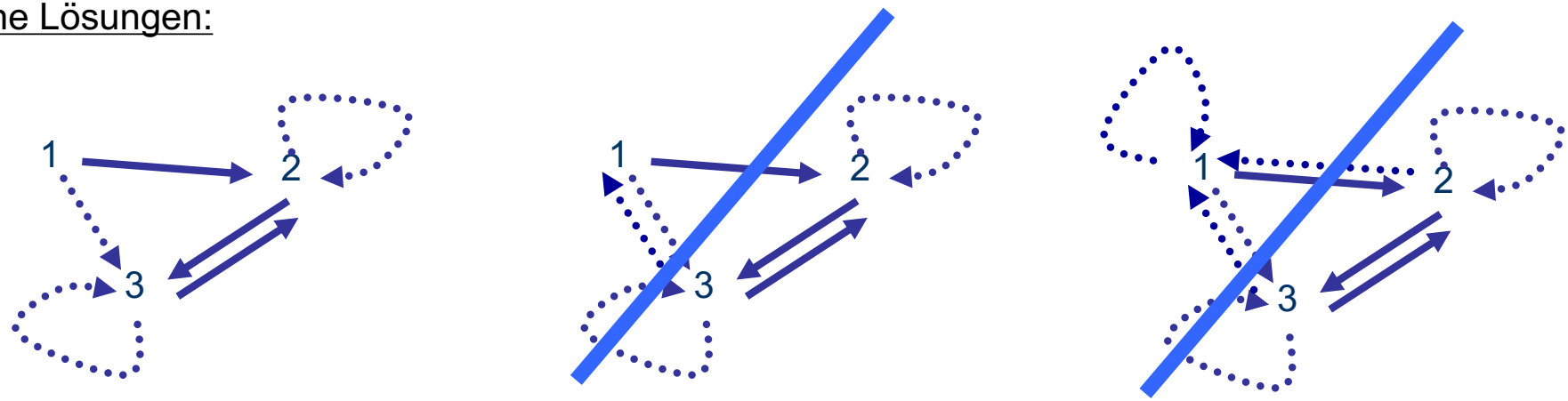
$\forall$ x,y,z (T(x,y) $\leftarrow$ (G(x,z) $\wedge$ T(z,y)))

G(1, 2), G(2, 3), G(3, 2)

G:



Mögliche Lösungen:



→ ***Wähle minimales Modell*** ←

T soll kleinste Menge von Fakten enthalten, so dass Regeln wahr sind

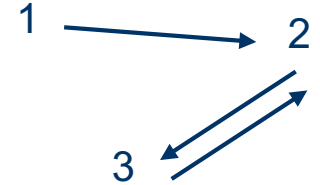
Minimale-Modell-Semantik

$\forall$ x,y (T(x,y) $\leftarrow$ G(x,y))

$\forall$ x,y,z (T(x,y) $\leftarrow$ (G(x,z) $\wedge$ T(z,y)))

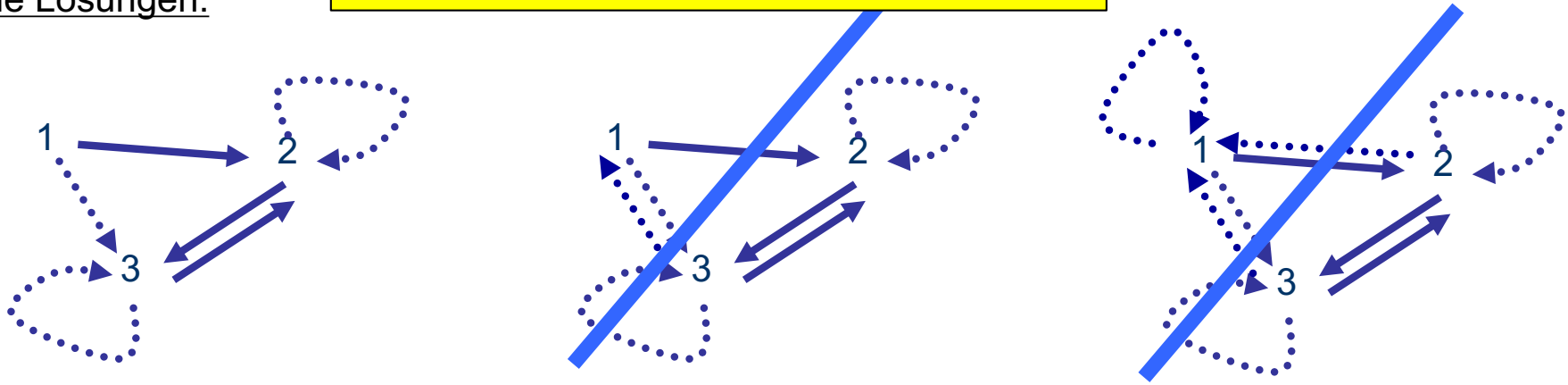
G(1, 2), G(2, 3), G(3, 2)

G:



Terminologie:
G is ein EDB-Prädikat (extensionale DB)
T ist ein IDB-Prädikat (intensionale DB)

Mögliche Lösungen:



→ ***Wähle minimales Modell*** ←
T soll kleinste Menge von Fakten enthalten, so dass Regeln wahr sind

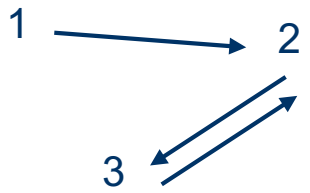
Datalog$_{(rec, no\text{-}neg)}$

Erreichbarkeit in Graphen (T = transitive Hülle von G):

```
T(x,y)  :- G(x,y)
T(x,y)  :- G(x,z), T(z,y)
```

Intuition

Transitive Hülle:




```
T(x,y)  :- G(x,y)
T(x,y)  :- G(x,z), T(z,y)
```

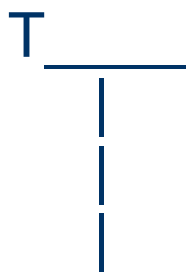
T

G

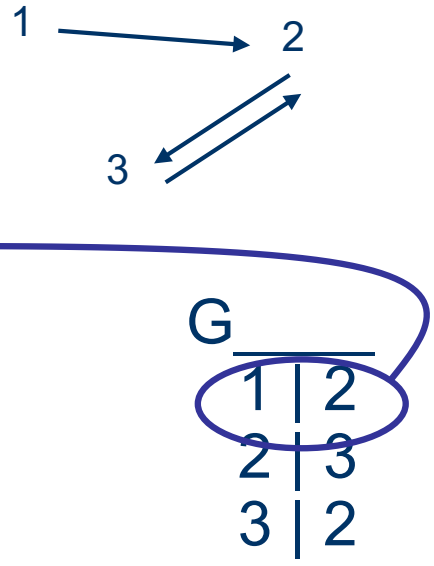
1	2
2	3
3	2

Intuition

Transitive Hülle:



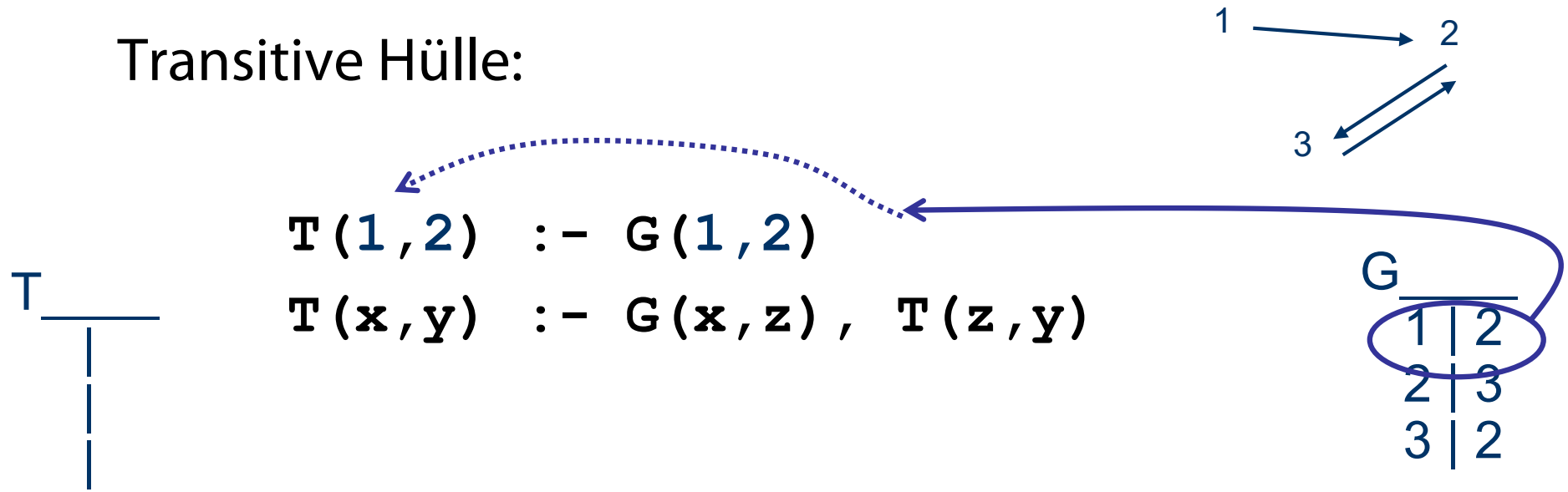

```
T(x,y)  :- G(1,2)
T(x,y)  :- G(x,z), T(z,y)
```



(1) Map from instances over the relations in the rule body

Intuition

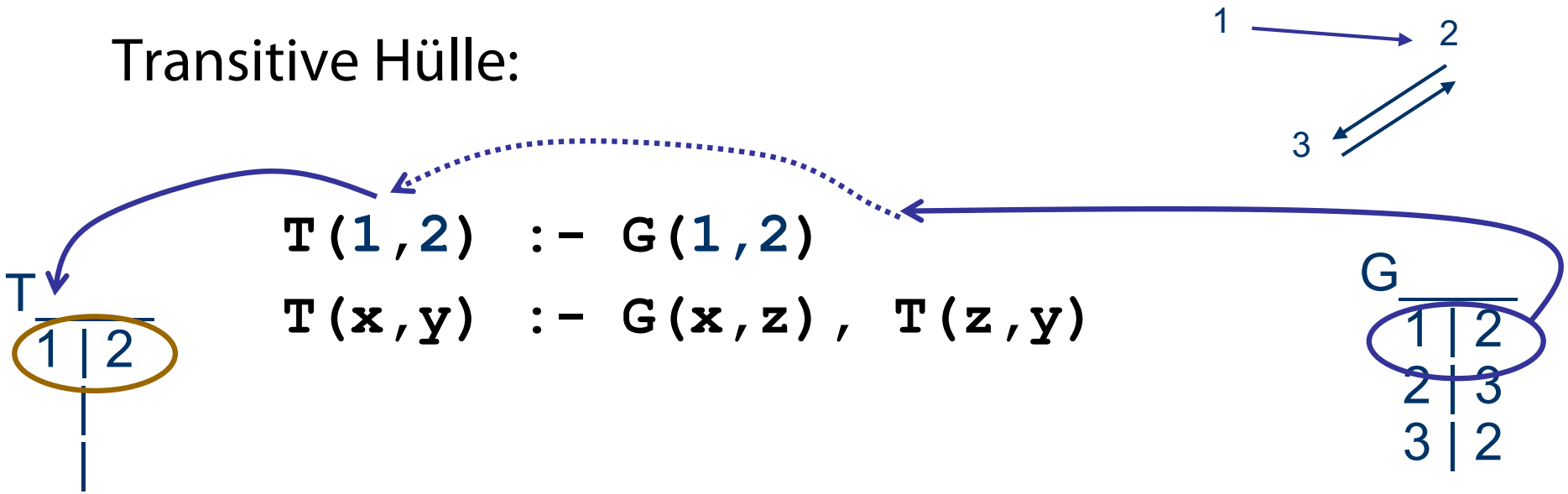
Transitive Hülle:



(2) ... map to instances over the relations in the rule head

Intuition

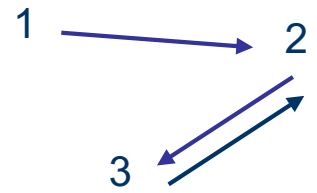
Transitive Hülle:



(2) ... map to instances over the relations in the rule head

Intuition

Transitive Hülle:




```
T(x,y) :- G(x,y)
T(x,y) :- G(x,z), T(z,y)
```

T

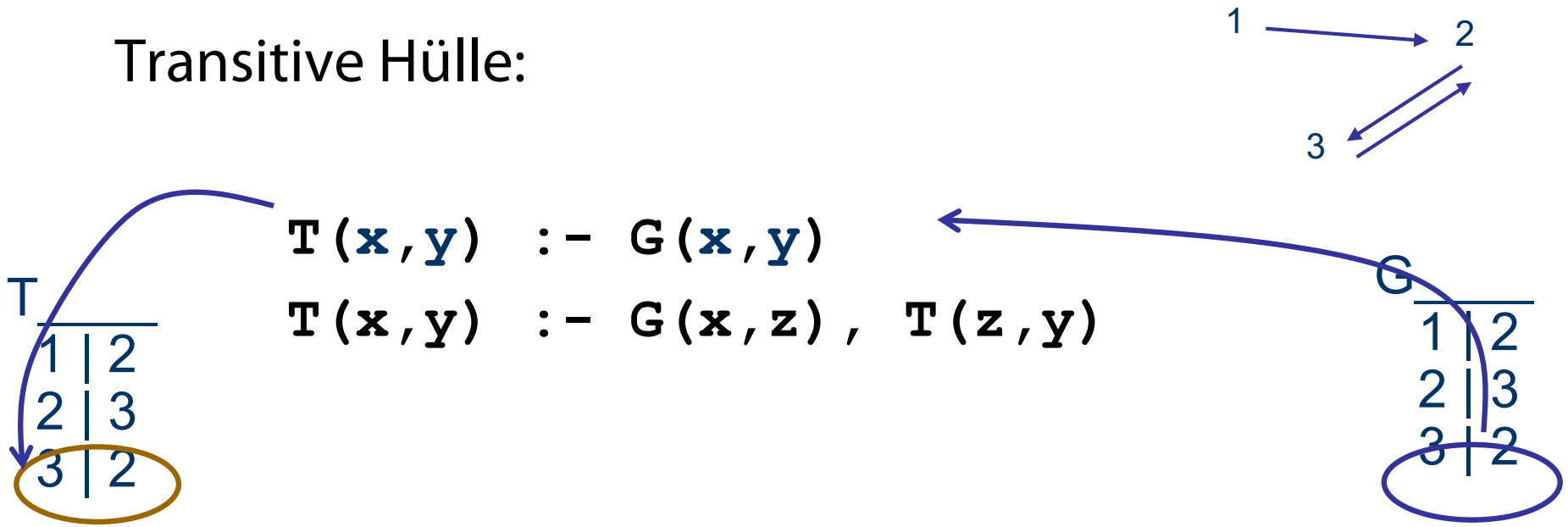
1	2
2	3

G

1	2
2	3
3	2

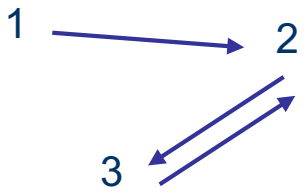
Intuition

Transitive Hülle:



Intuition

Transitive Hülle:




```
T(x,y)  :- G(x,y)
T(x,y)  :- G(1,2), T(2,3)
```

T

1	2
2	3
3	2

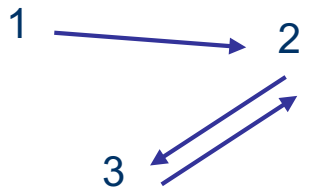
G

1	2
2	3
3	2

(1) Setze Tupel aus den Relationen in den Regelrujmpf ein

Intuition

Transitive Hülle:




```
T(x,y) :- G(x,y)
T(1,3) :- G(1,2), T(2,3)
```

T

1	2
2	3
3	2

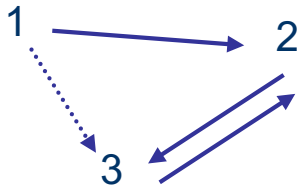
G

1	2
2	3
3	2

(2) Übertrage Bindungen in den Regelkopf

Intuition

Transitive Hülle:




```
T(x,y)  :- G(x,y)
T(1,3)  :- G(1,2), T(2,3)
```

T

1	2
2	3
3	2
1	3

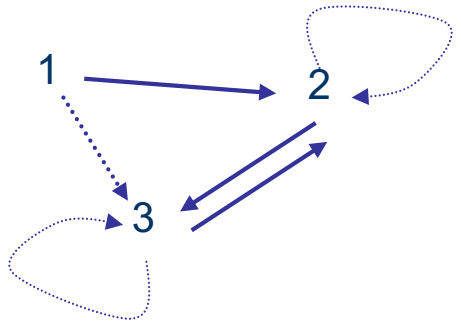
G

1	2
2	3
3	2

(2) Übertrage Bindungen in den Regelkopf

Intuition

Transitive Hülle:



T

1	2
2	3
3	2
1	3
2	2
3	3

```
T(x,y) :- G(x,y)
T(x,y) :- G(x,z), T(z,y)
```

G

1	2
2	3
3	2

... Wiederhole bis Fixpunkt erreicht

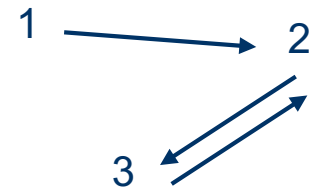
Correctness

$\forall$ x,y (T(x,y) $\leftarrow$ G(x,y))

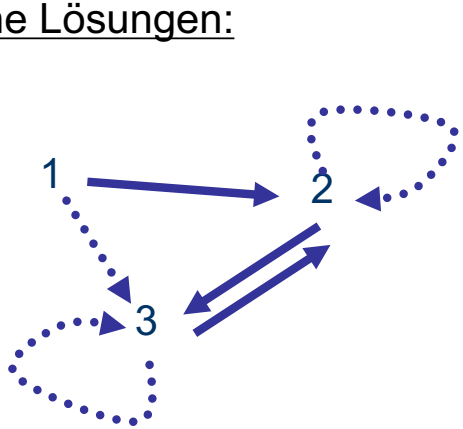
$\forall$ x,y,z (T(x,y) $\leftarrow$ (G(x,z) $\wedge$ T(z,y)))

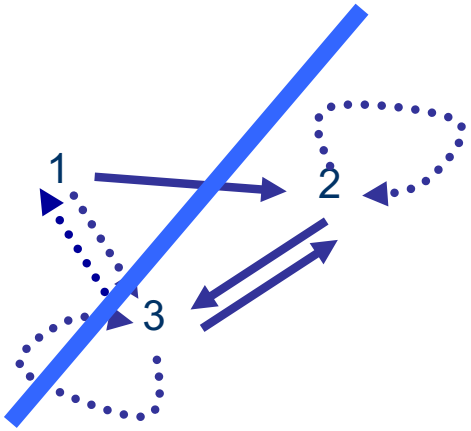
G(1,2), G(2,3), G(3, 2)

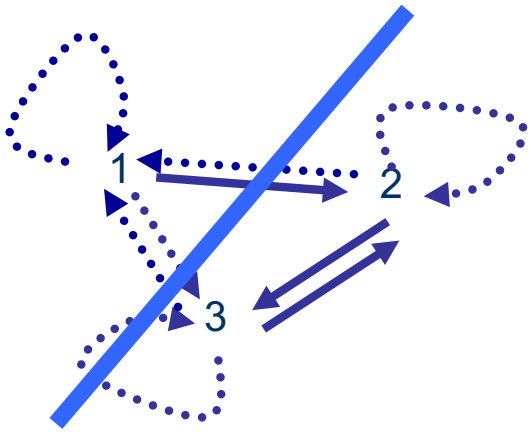
G:



Mögliche Lösungen:







→ ***Wähle minimales Modell*** ←

T soll kleinste Menge von Fakten enthalten, so dass Regeln wahr sind

Einschränkungen

- Negation in reinem Datalog nicht erlaubt

- Aber: „Sichere“ Erweiterung von Datalog mit Negation möglich
  1. Negation nur als „Filter“ (variablen in negierten Atomen bereits durch positive Atome gebunden)
     Verboten: `T(x,y) :- ¬ H(x,y);`
     Erlaubt: `T(x,y) :- ¬ H(x,y),G(x,w), R(w,y)`
  2. und negiertes Atom nicht in Schleife eines IDB-Prädikats
     Verboten:

```
T(x,y) :- ¬ H(x,y), G(x,y);
H(x,y) :- T(y,x)
```

     Erlaubt:

```
T(x,y) :- ¬ H(x,y), G(x,y);
H(x,y) :- G(y,x)
```

  → Stratifizierte Programme

Einschränkungen

- Syntaxeinschränkungen bzgl. (freier) Variablen
- Unsichere Anfrage:
  Verboten: `Colored_edges(x,y,col) :- G(x,y)`

→ Variablen im Kopf müssen im Rumpf vorkommen (Wertebereichsbeschränktheit)

→ Safety-Bedingungen: Keine Probleme mit Endlosschleifen

- Aufweichung der Einschränkungen möglich (hier nicht vertieft)
  - Läuft auf sogenannte semantisches Kriterium der Domänenunabhängigkeit (domain independence) hinaus
  - Sicher Programme eine echte Teilmenge der domain-independent Programme; leider domain independence nicht entscheidbar

Auswertung durch iterative Verbunde

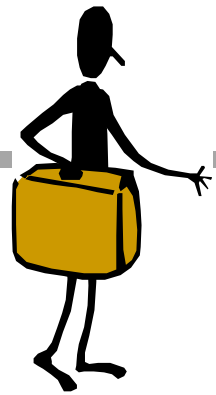
```
T(x,y) :- G(x,y).
T(x,y) :- G(x,z), T(z,y).
```

- Daten in Datenbank:

  ```
  G(1,2). G(2,3). G(3,2).
  ```

- Berechne Verbund G(x,z) $\wedge$ T(z,y) für weitere G-Tupel
- Verwende Verbunde über Daten bis Fixpunkt erreicht (naive Auswertung)
- Betrachte nur Datensätze, die im vorigen Schritt hergeleitet (semi-naive Auswertung)
- Transformation der Datalog-Regeln für Konstanten in der Anfrage: Magic-Set-Transformation (hier nicht vertieft)

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

Rekursive Anfragen

Beim nächsten Mal...