Datenbanken

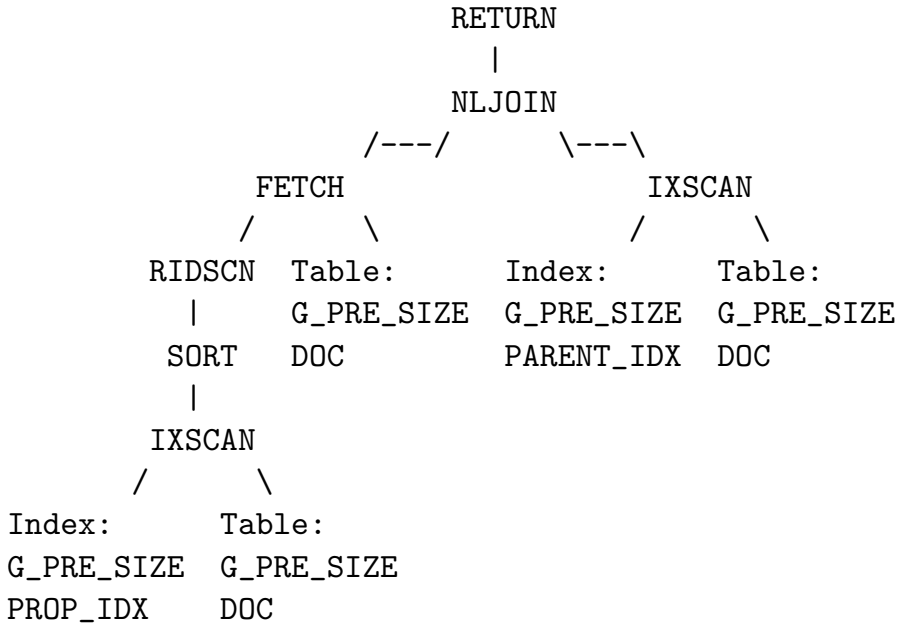
Anfrageverarbeitung Teil 2

Dr. Özgür Özçep

Universität zu Lübeck

Institut für Informationssysteme

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

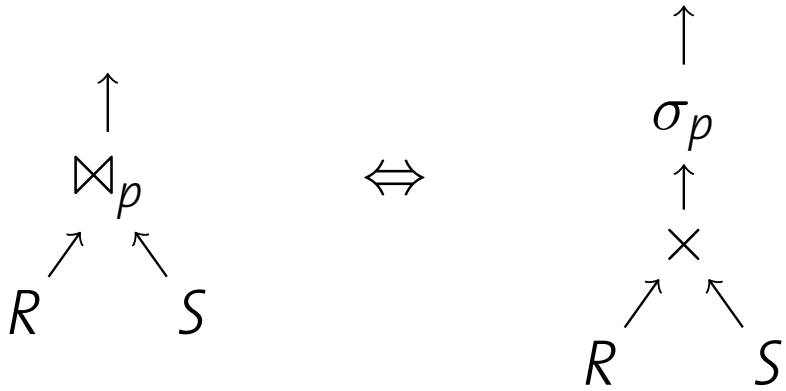
Als nächstes betrachten wir Verbundoperatoren

Physikalische Operatoren

- Zu einem logischen Operator kann es mehr als einen physikalischen Operator geben.
  (Worin müssten diese sich ähnlich sein?)
- Varianten in Abhängigkeit von
  - Vorhandensein eines Index
  - Sortiertheit des Inputs
  - Größe des Inputs
  - Verfügbare Pufferrahmen
  - ...
- Der Anfrageoptimierer ist für die Auswahl der richtigen Variante(n) zuständig (kommt später)

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:

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

Verbund-als-geschachtelte-Schleifen

Nur 3 Seiten nötig (zwei Seiten für das 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

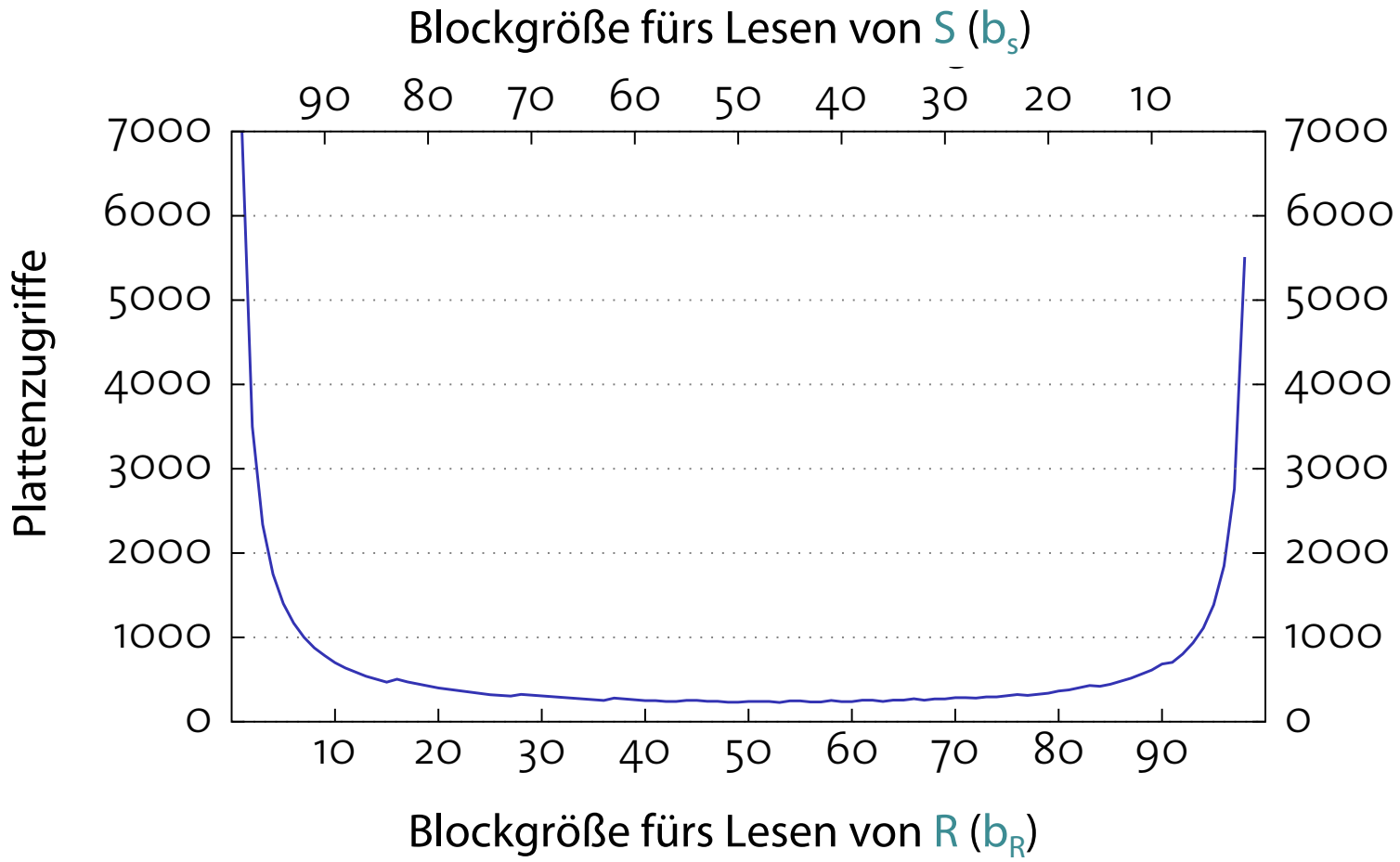
Einsparung von Kosten durch wahlfreien Zugriff durch blockweises Lesen von R und S mit b_R und b_S vielen Seiten

```
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 vorhandenen 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)
  - Ein zusammengesetzter Schlüssel kann durch eine Konjunktion von Atomen abgedeckt sein
  - Manchmal auch nur partielle Abdeckung nützlich (wenn z.B. jeweils ein Konjunkt mit einem jeweils anderen Index verträglich)
- Solche Verbundbedingungen heißen „sargable“ (SARG= search argument)
  - Beispiel für nicht-sargable Prädikate z.B. durch Funktionsanwendung auf Variable
  - ... WHERE SUBSTRING(Name,3) = ‚ups‘

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

Verträglichkeit

- Konjunktive Bedingung p deckt kompositionalen Hashindex $k = (A_1, ..., A_n)$ ab genau dann, wenn

  p die Form hat $A_1 = c_1$ AND .... AND $A_n = c_n$ AND p'.

  (Restbedinung p' wird erst nach dem Indexretrieval ausgewertet)

- Konjunktive Bedingung p deckt B^+-Index $k = (A_1, ..., A_n)$ ab genau dann, wenn p die Form hat

  $A_1 \theta c_1$ AND $A_2 \theta c_2$ AND .... AND $A_k \theta c_n$ AND p'

  für $k \leq n$ und $\theta \in \{=, <, >, \leq, \geq\}$

EXAMPLE

- A relation `R(a,b,c,d)`
- Does the index match the predicate?

Predicate	B+ tree on (a,b,c)	Hash index on (a,b,c)
a=5 and b=3		
a>5 and b<4		
b=3		
a=5 and c>10		
a=5 and b=3 and c=1		
a=5 and b=3 and c=1 and d >6		

a=5 and b=3 and c=1 are primary conjuncts here

Match mehrerer Indizes

- Beispiel
  - Tabelle R mit Indizes I1 und I2
  - Prädikat p = p1 AND p2
  - p1 matched I2 und p2 matched I2

  - Kann jeweils die Konjunkte auswerten und dann Mengenschnitt bilden.

I/O-Verhalten für Indexbasierten Verbund

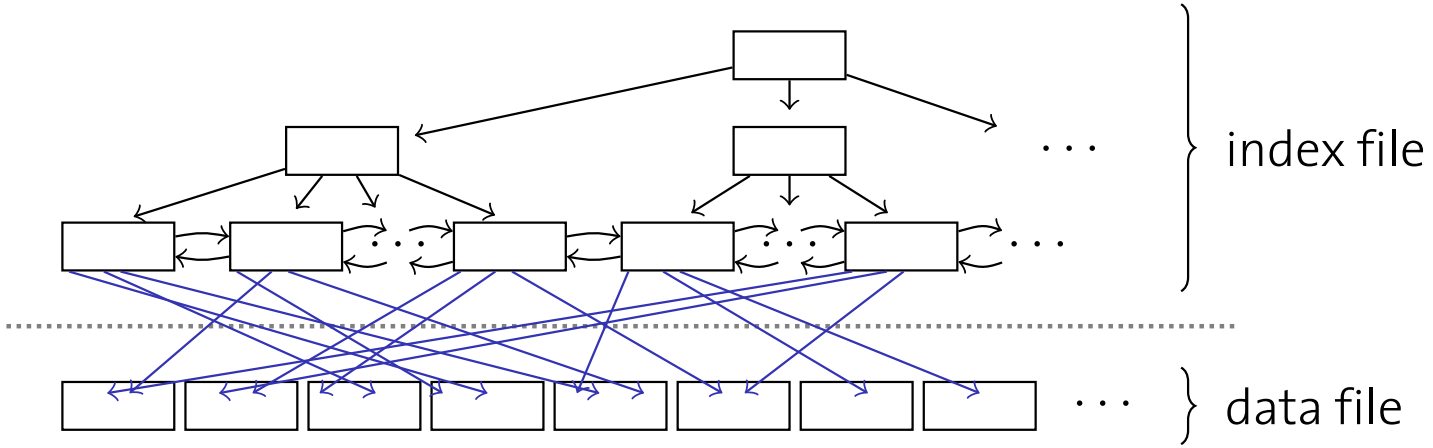
Für jeden Datensatz in R verwende Index zum Auffinden von korrespondierenden S-Tupeln. Für jedes R-Tupel sind folgende Kosten einzukalkulieren:

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

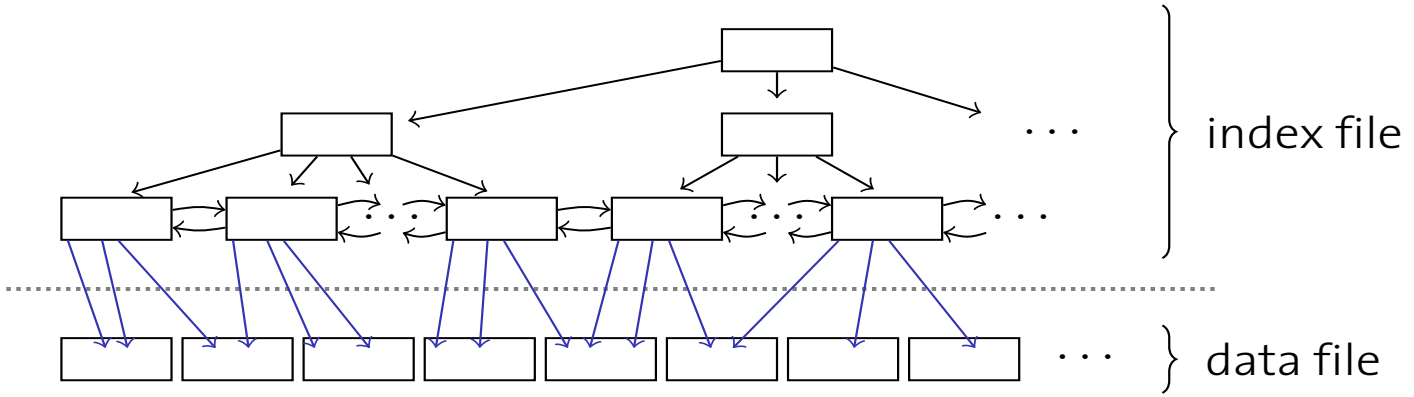
Wegen 2. und 3. Kosten von der Verbundgröße abhängig

Erinnerung: Clustering

Nicht geclustert



Geclustert



Zugriffskosten N_{idx} 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 N_{idx} 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 (gegenüber blockweisem Verbund etwa), wenn nur einige wenige 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 (stört sequenziellen Zugriff)

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 verwendet

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
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 ;
```

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

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 (von der SQL-Anfrage erzwungener) 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?

Aufgabe:

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

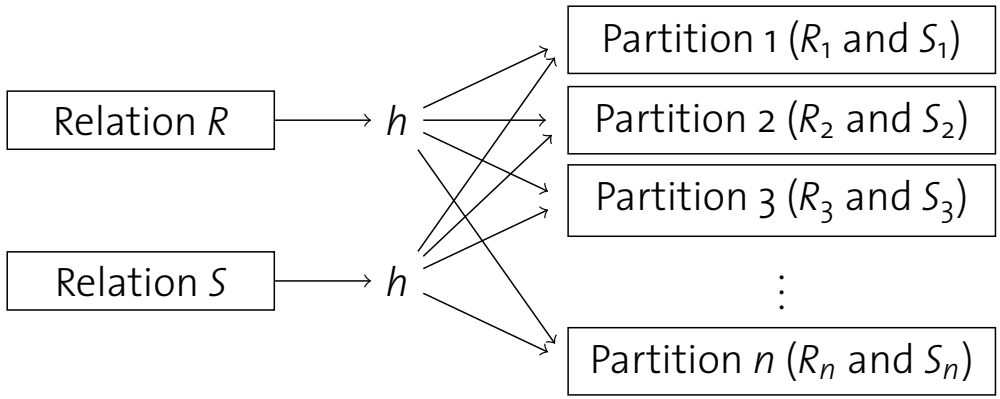
Lösung:

- 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 (d.h. $N_{Ri} < B - 1$; wobei B = Pufferrahmenzahl)
  - 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, 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)

Aufgabe:

Warum reicht es aus zu fordern, dass R aus höchstens $(B-1)^2$ Seiten besteht, damit man die n Partitionen in einem Durchgang erstellt (Bedenke, dass $N_{Ri} < B-1$)?

Warum ist dies nur eine Approximation?

Aufgabe:

Warum reicht es aus zu fordern, dass R aus höchstens $(B-1)^2$ Seiten besteht, damit man die n Partitionen in einem Durchgang erstellt (Bedenke, dass $N_{Ri} < B-1$)? Warum ist dies nur eine Approximation?

Lösung:

- Bei einem Partitionierungsdurchgang können höchtens B-1 Partitionen rausgeschrieben werden (eine Seite nötig für Input). Da $N_{Ri} < B-1$, sind also höchstens (B-1)*(B-1) Seiten erlaubt für R.
- Das Hashen garantiert keine gleichmäßige Verteilung in die Partitionen. Daher muss R tatsächlich noch etwas kleiner sein.
- Wir lernen: Größere Dateien benötigen mehrere Partitionierungsdurchgänge (rekursives Partitionieren)

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

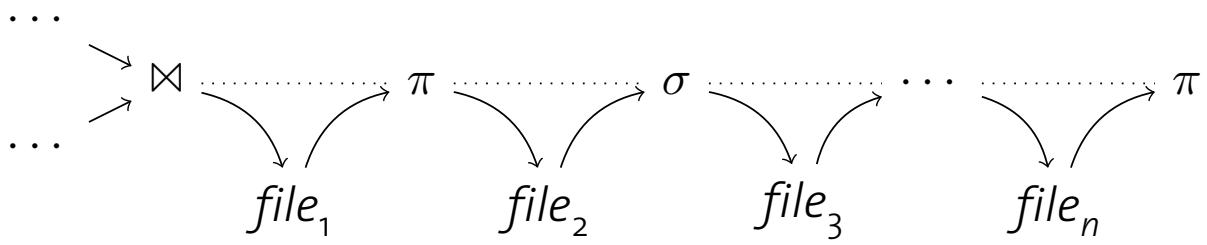
- Implementierung durch
  a. Entfernen nicht benötigter Spalten
  b. Eliminierung von Duplikaten
- Die Implementierung von a) bedingt das Ablaufen (scan) aller Datensätze in der Datei, b) siehe oben
- Systeme vermeiden b) sofern möglich (in SQL muss Duplikate-Eliminierung angefordert werden)

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

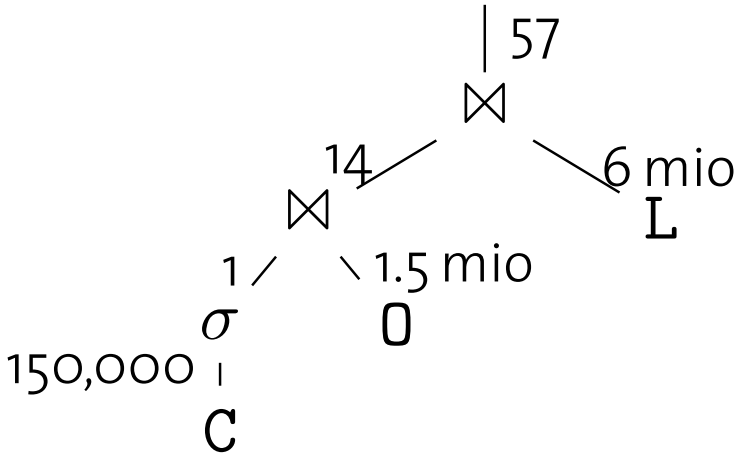
- Alternativ könnte jeder Operator seine Ergebnisse direkt an den nachfolgenden senden, ohne die Ergebnisse erst auf die Platte zu schreiben
- Ergebnisse werden so früh wie möglich weitergereicht und verarbeitet (Pipeline-Prinzip)
- Granularität ist bedeutsam:
  - Kleinere Brocken reduzieren Antwortzeit 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 ;
```

Evaluierung eines Ausführungsplans

- Wurzel.Open() vom Anfrageevaluierer (AE)
- Open() wird propagiert durch die Operatoren
- Kontrolle zurück an AE
- AE ruft Wurzel.next() auf
- Next() wird durch die Operatoren so weit propagiert wie nötig. Sobald neues Ergebnis-Tupel produziert, geht Kontrolle zurück an AE.

Blockierende Operatoren

- Pipelining reduziert Speicheranforderungen und Antwortzeiten, da jeder Datensatz gleich weitergeleitet
- Funktioniert so nicht für alle Operatoren
- Für welche nämlich nicht?

Blockierende Operatoren

- Pipelining reduziert Speicheranforderungen und Antwortzeiten, da jeder Datensatz gleich weitergeleitet
- Funktioniert so nicht für alle Operatoren
- Für welche nämlich nicht?
  - 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)