Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Übung Datenbanksysteme II

Anfrageausführung

Leon Bornemann

Folien basierend auf
Maximilian Jenders,
Thorsten Papenbrock

Rückblick:

Hausaufgabe 2

- Aufgabe 1b:
  - Jetzt Live

Einleitung:
Themen

- Iterator-Operatoren

- Algorithmen-Klassen
  - Sort-basierte
  - Hash-basierte
  - Index-basierte

- Algorithmen-Schwierigkeitsgrade
  - One-Pass-Algorithmen
  - Two-Pass-Algorithmen
  - Multipass-Algorithmen

Entwurf

- Skizziere den Aufbau eines *nicht blockierenden* Iterator-Operators:

Open():
```
r = relation.Open();
```

GetNext():
```
return = r.GetNext();
```

Close():
```
r.Close();
```

- Skizziere den Aufbau eines *blockierenden* Iterator-Operators:

Open():
```
r = relation.Open();
while ((t = r.GetNext()) != NotFound)
   c[i++] = t;
```

GetNext():
```
if (i < c.size())
   return c[i++];
return NotFound;
```

Close():
```
r.Close();
```

Iterator-Operatoren: Konstruktion

- Entwerfe für jeden der folgenden Operationen einen *nicht blockierenden* Iterator:

1. Projektion
2. Duplikateliminierung
3. Vereinigung (Menge)

Konstruktion

- Entwerfe für jeden der folgenden Operationen einen *nicht blockierenden* Iterator:

1. Projektion
2. Duplikateliminierung
3. Vereinigung (Menge)

Open():

```
r = relation.Open();
```

GetNext():

```
t = r.GetNext();
if (t == NotFound)
   return t;
return Projection(t);
```

Close():

```
r.Close();
```

Iterator-Operatoren: Konstruktion

- Entwerfe für jeden der folgenden Operationen einen *nicht blockierenden* Iterator:

1. Projektion
2. Duplikateliminierung
3. Vereinigung (Menge)

```
Open():
r = relation.Open();
s = empty set;

GetNext():
while (true)
   t = r.GetNext();
   if (t == NotFound)
      return t;
   if (! s.Contains(t))
      s.Add(t)
      return t;

Close():
r.Close();
```

Konstruktion

- Entwerfe für jeden der folgenden Operationen einen *nicht blockierenden* Iterator:

1. Projektion
2. Duplikateliminierung
3. Vereinigung (Menge)

Open():
```
r = rel1.Open();
s = empty set;
r1done = false;
```

GetNext():
```
while (true)
   t = r.GetNext();
   if (t == NotFound)
      if (r1done) return t;
      else
         r.Close();
         r = rel2.Open();
         r1done = true;
         continue;
if (! s.Contains(t))
      s.Add(t)
      return t;
```

Close():
```
r.Close();
```

Andere Iterator-Operatoren können auch in der GetNext()-Funktion geöffnet und geschlossen werden!

Themen

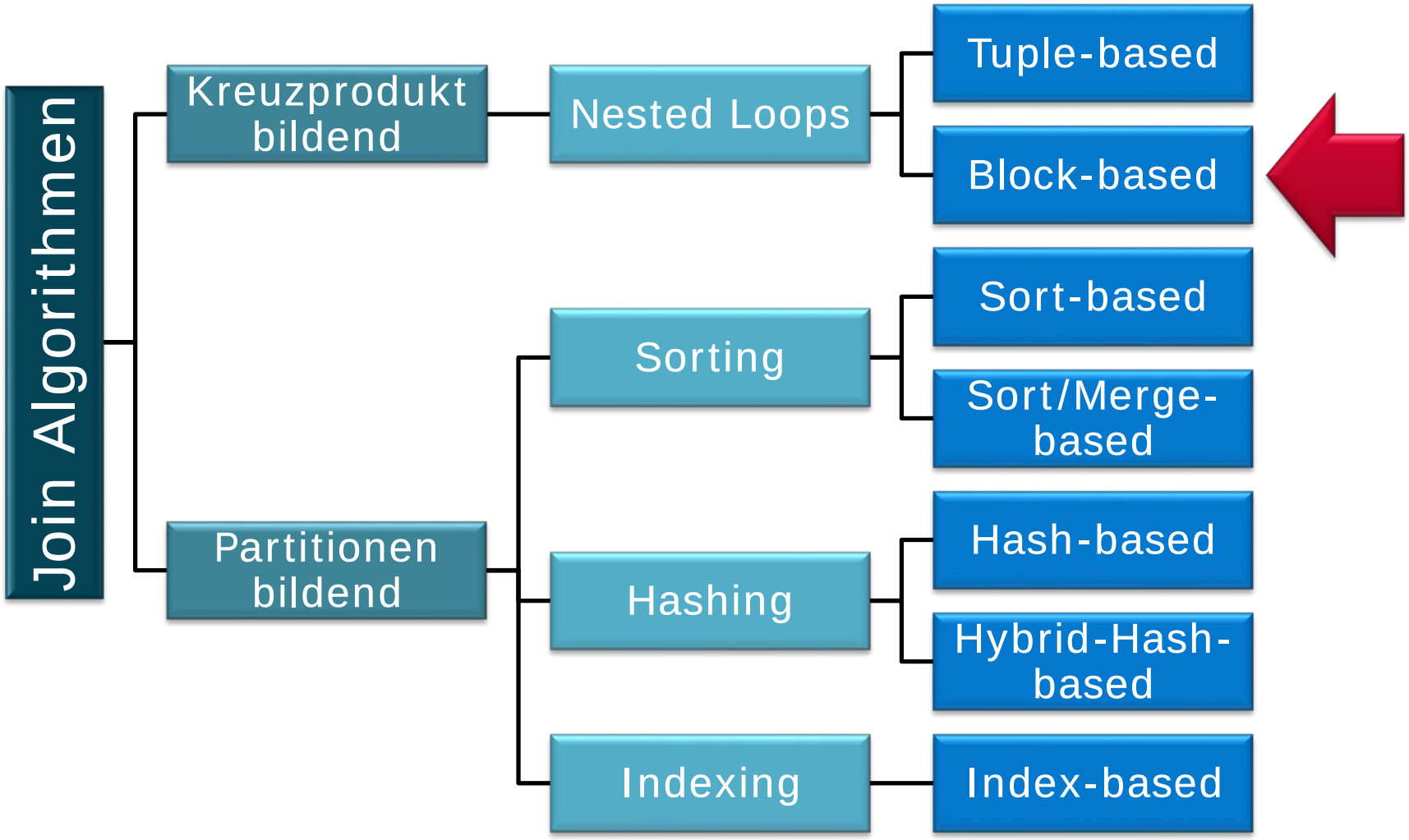
- Iterator-Operatoren

- Algorithmen-Klassen
  - Sort-basierte
  - Hash-basierte
  - Index-basierte

- Algorithmen-Schwierigkeitsgrade
  - One-Pass-Algorithmen
  - Two-Pass-Algorithmen
  - Multipass-Algorithmen

Algorithmen-Klassen:

Join-Algorithmen



Block-based Nested-Loop-Join

- Gegeben:
  - Zwei Relationen R und S mit je 10.000 Blöcken:
    B(R) = B(S) = 10.000
  - Hauptspeicher mit einer Kapazität von 1.001 Blöcken:
    M = 1.001

a) Berechne die Anzahl an I/O-Operationen für die Berechnung von R ⋈ S, wenn ein Block-basierter NLJ verwendet wird.

b) Wie groß muss M mindestens sein, wenn R ⋈ S unter Verwendung des Block-basierten NLJ mit nicht mehr als 100.000 I/O-Operationen berechnet werden soll?

c) Wie groß muss M mindestens sein, wenn R ⋈ S unter Verwendung des Block-basierten NLJ mit nicht mehr als 25.000 I/O-Operationen berechnet werden soll?

Block-based Nested-Loop-Join

- Gegeben: B(R) = B(S) = 10.000; M = 1.001

a) Berechne die Anzahl an I/O-Operationen für die Berechnung von R ⋈ S, wenn ein Block-basierter NLJ verwendet wird.

```
FOR EACH chunk of M-1 blocks of S DO BEGIN
   read blocks into main memory;
   organize tuples into efficient data structure;
   FOR EACH block b of R DO BEGIN
      read b into main memory;
      FOR EACH tuple t of b DO BEGIN
         find tuples of S in main memory that join;
         output those joined tuples;
      END;
   END;
END;
```

Insgesamt $B(S)$

$B(R)$

$\left\lceil \frac{B(S)}{M-1} \right\rceil$ Mal

$$IO = B(S) + \left\lceil \frac{B(S)}{M-1} \right\rceil * B(R)$$

Block-based Nested-Loop-Join

- Gegeben: B(R) = B(S) = 10.000; M = 1.001

a) Berechne die Anzahl an I/O-Operationen für die Berechnung von R ⋈ S, wenn ein Block-basierter NLJ verwendet wird.

$$IO = B(S) + \left\lceil \frac{B(S)}{M-1} \right\rceil * B(R)$$

$$IO = 10000 + \left\lceil \frac{10000}{1001-1} \right\rceil * 10000 = 10000 +$$

$$IO = 110000$$

Was, wenn
B(R) != B(S)?
Welche Relation außen?

Block-based Nested-Loop-Join

- Gegeben: B(R) = B(S) = 10.000

b) Wie groß muss M mindestens sein, wenn R ⋈ S unter Verwendung des Block-basierten NLJ mit nicht mehr als 100.000 I/O-Operationen berechnet werden soll?

$$\mathrm{I/O} \geq B(S) + \left\lceil \frac{B(S)}{M-1} \right\rceil * B(R)$$

$$\rightarrow 100.000 \geq 10.000 + \left\lceil \frac{10.000}{M-1} \right\rceil * 10.000$$

$$\rightarrow 90.000 \geq \left\lceil \frac{10000}{M-1} \right\rceil * 10.000$$

$$\rightarrow 9 \geq \left\lceil \frac{10000}{M-1} \right\rceil \rightarrow \frac{10000}{M-1} \leq 9$$

$$\rightarrow M \geq \frac{10000}{9} + 1 = 1112{,}11 \rightarrow M = 1113$$

Vorsicht bei Kommazahlen
Siehe c)

Block-based Nested-Loop-Join

- Gegeben: B(R) = B(S) = 10.000

c) Wie groß muss M mindestens sein, wenn R ⋈ S unter Verwendung des Block-basierten NLJ mit nicht mehr als 25.000 I/O-Operationen berechnet werden soll?

$$\mathrm{I/O} \geq B(S) + \left\lceil \frac{B(S)}{M-1} \right\rceil * B(R)$$

$$\rightarrow 25.000 \geq 10.000 + \left\lceil \frac{10.000}{M-1} \right\rceil * 10.000$$

$$\rightarrow 15.000 \geq \left\lceil \frac{10000}{M-1} \right\rceil * 10.000$$

$$\rightarrow 1{,}5 \geq \left\lceil \frac{10000}{M-1} \right\rceil \geq \rightarrow \frac{10000}{M-1} \leq \mathbf{1}$$

$$\rightarrow M \geq 10000 + 1 = 10001$$

Index-basierte Algorithmen

- Gegeben ist eine Relation R sowie ein Index auf R.a (kein Primärindex!). Beschreibe, wie dieser Index genutzt werden kann, um die Ausführung der folgenden Operationen zu verbessern.

a) $R \cup_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

"S" für Mengensemantik!

b) $R \cap_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

Annahme: M < R + S, ansonsten trivial!

c) $\delta(R)$

R kann Duplikate enthalten.

Index-basierte Algorithmen

- Gegeben ist eine Relation R sowie ein Index auf R.a (kein Primärindex!). Beschreibe, wie dieser Index genutzt werden kann, um die Ausführung der folgenden Operationen zu verbessern.

a) $R \cup_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

b) $R \cap_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

c) $\delta(R)$

R kann Duplikate enthalten.

Was, wenn s.a nicht in R existiert? Auch ausgeben!

1. Lese Index von R in den Hauptspeicher
2. Lade und Schreibe alle Tupel von R in den Output
3. Lese die Tupel von S blockweise
4. Lade für jedes S-Tupel s über den Index alle R-Tupel r bei denen s.a=r.a und falls ∀r: s≠r, dann gib s aus

Index-basierte Algorithmen

- Gegeben ist eine Relation R sowie ein Index auf R.a (kein Primärindex!). Beschreibe, wie dieser Index genutzt werden kann, um die Ausführung der folgenden Operationen zu verbessern.

a) $R \cup_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

b) $R \cap_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

c) $\delta(R)$

R kann Duplikate enthalten.

1. Lese Index von R in den Hauptspeicher
2. Lese die Tupel von S blockweise
3. Lade für jedes S-Tupel s über den Index alle R-Tupel r bei denen s.a=r.a und falls ∃r: s=r, dann gib s aus

Index-basierte Algorithmen

- Gegeben ist eine Relation R sowie ein Index auf R.a (kein Primärindex!). Beschreibe, wie dieser Index genutzt werden kann, um die Ausführung der folgenden Operationen zu verbessern.

Problem: Wir lesen Tupelweise! eventuell müssen dadurch viele Blöcke mehrfach geladen werden.

Wenn die Relation clustered vorliegt Ist dieser Algorithmus in vielen Fällen langsamer als beispielsweise Duplikateliminierung durch sortieren

c) $\delta(R)$
R kann Duplikate enthalten.

1. Lese Index von R in den Hauptspeicher
2. Für jeden Wert v im Index lade alle R-Tupel r mit r.a = v
3. Eliminiere Duplikate in geladenen Tupeln und gib verbleibende Tupel aus

Index-basierte Algorithmen

- Gegeben ist eine Relation R sowie ein Index auf R.a (kein Primärindex!). Beschreibe, wie dieser Index genutzt werden kann, um die Ausführung der folgenden Operationen zu verbessern.

a) $R \cup_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

b) $R \cap_S S$

R und S haben keine Duplikate, können jedoch Tupel gemeinsam haben.

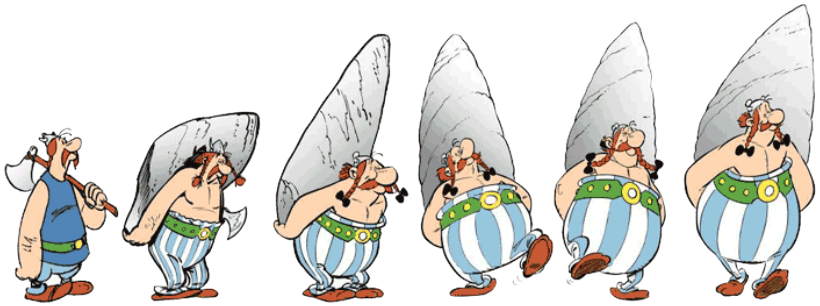
c) $\delta(R)$

R kann Duplikate enthalten.

Themen

- Iterator-Operatoren

- Algorithmen-Klassen
  - Sort-basierte
  - Hash-basierte
  - Index-basierte



- Algorithmen-Schwierigkeitsgrade
  - One-Pass-Algorithmen
  - Two-Pass-Algorithmen
  - Multipass-Algorithmen

Sort-based Two-Pass Union

- Gegeben:
  - Zwei Relationen R(X,Y,Z) und S(X,Y,Z)
  - Die Tupel in R und S belegen folgende Block-Zahlen:
    B(R) = 3.000
    B(S) = 200
- Berechne die I/O-Kosten für die Sort-basierte Two-Pass Mengen-Vereinigung – *ohne* das Schreiben des Ergebnisses auf die Platte.

Pass 1: R und S lesen und als sortierte Teillisten wieder schreiben
→ B(R) und B(S) lesen + B(R) und B(S) schreiben

Pass 2: Sortierte Teillisten lesen und dabei Vereinigung bestimmen
→ B(R) und B(S) lesen

I/O = 3 · (B(R) + B(S)) = 3 · (3.000 + 200) = 9.600

Sort-based Two-Pass Join

Annahme: Wir wissen nicht, ob S nach Y sortiert ist!

- Gegeben:
  - Zwei Relationen R(X,Y) und S(Y,Z), für die die folgende FK gilt: R.Y → S.Y
  - Die Tupel in R und S belegen folgende Block-Zahlen:
    B(R) = 3.000
    B(S) = 200
- Berechne die I/O-Kosten für den *einfachen* Sort-basierten Two-Pass Join – *ohne* das Schreiben des Ergebnisses auf die Platte.

Pass 1: R und S lesen und als sortierte Teillisten wieder schreiben
→ B(R) lesen und schreiben + B(S) lesen und schreiben
Sortierte Teillisten lesen und Sortierung für R und S schreiben
→ B(R) lesen und schreiben + B(S) lesen und schreiben
Pass 2: Sortierte Relationen R und S parallel lesen und joinen
→ B(R) und B(S) lesen

I/O = 5 · (B(R) + B(S)) = 5 · (3.000 + 200) = 16.000

Sort-based Two-Pass Join

- Gegeben:
  - Zwei Relationen R(X,Y) und S(Y,Z), für die die folgende FK gilt: R.Y → S.Y
  - Die Tupel in R und S belegen folgende Block-Zahlen:
    B(R) = 3.000
    B(S) = 200
- Berechne die I/O-Kosten für den *einfachen* Sort-basierten Two-Pass Join – *ohne* das Schreiben des Ergebn[illegible]uf die Platte.

Als Übung für die Klausur:

Selbe Aufgabe mit *Sort-Merge* Two-Pass Join!

(d.h. Merge und Join in einem Schritt)

Operatoren und Algorithmen

- Alle Operatoren der relationalen Algebra unterstützen Pipelining.

  Falsch Die relationalen Operatoren Sortierung und Gruppierung sind inhärent blockierend und daher Beispiele für Operatoren, die kein Pipelining unterstützen.

- Wenn die für den Operator notwendigen Indexe bereits vorhanden sind sind index-basierte Operatoren meistens am effizientesten.

  Wahr Durch die Nutzung des Index kann auf zusätzliches Hashen oder Sortieren verzichtet werden. Falls die Relation aber schon gehashed oder sortiert vorliegt können diese Ansätze auch effizienter sein!

- Tabellen, die mindestens einen Index besitzen, sollten immer mit Index-basierten Operatoren angefragt werden.

  Falsch Der Index-basierte Operator bietet sich nur dann an, wenn er den vorhandenen Index auch tatsächlich nutzen kann.

- Two-Pass-Algorithmen sind stets effizienter als One-Pass-Algorithmen und sollten daher bevorzugt genutzt werden.

  Falsch One-Pass-Algorithmen effizienter, da sie Daten nur einmal von der Festplatte lesen. Allerdings können Daten ab einer bestimmten Größe nicht mehr per One-Pass-Algorithmus verarbeitet werden.