HPI Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Datenbanksysteme II

Anfrageausführung

(Kapitel 15)

23.5.2007

Felix Naumann

Besprechung Evaluation DBS I

HPI Hasso Plattner Institut

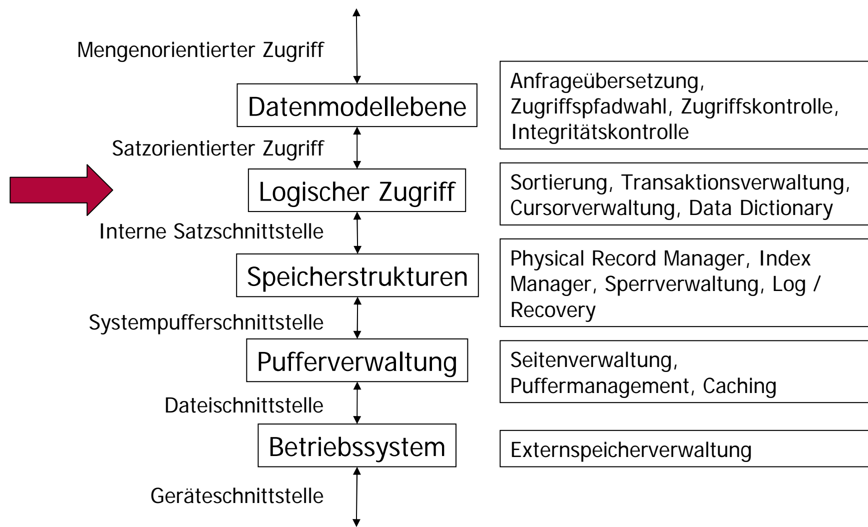
2

- 47 Stimmen von 66 Belegungen
- Beste Bachelorvorlesung – vielen Dank
- Kritische Kommentare
  - Fehler auf Folien – „online-Berichtigung": I
  - Fehler/Unklarheiten auf Aufgabenblättern: II
  - Zu ausschweifend: II
  - Leistungserfassung unklar
  - Zu schnell: IIIIII
    - Redet zu viel
  - Umgang mit DB2 zu simpel – Vertiefung wäre schön

Zoom in die interne Ebene: Die 5-Schichten Architektur

Hasso Plattner Institut

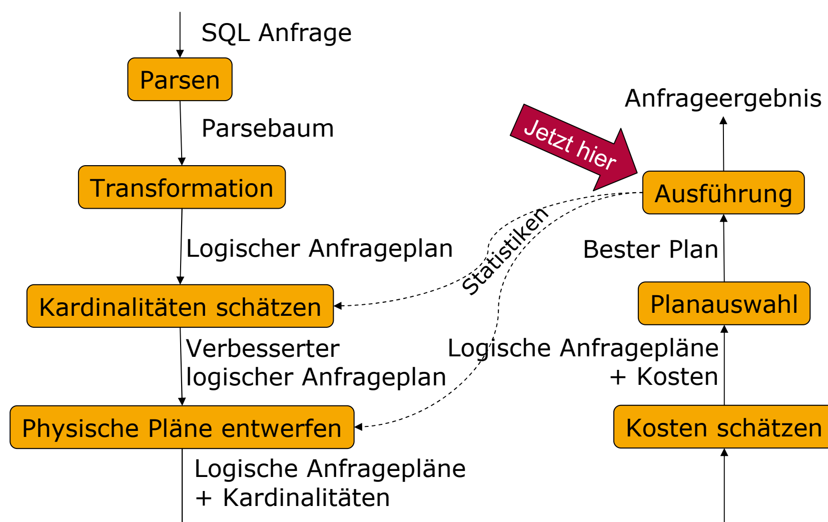
3



Ablauf der Anfragebearbeitung

Hasso Plattner Institut

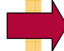
4



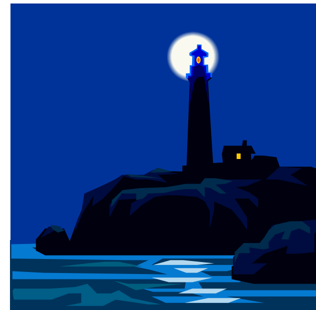
Überblick

HPI Hasso Plattner Institut

5



- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement



Grundbausteine

HPI Hasso Plattner Institut

6

- Anfragepläne bestehen aus Operatoren
  - Oft Operatoren der Relationalen Algebra
  - Aber auch: Scan einer Tabelle
- Physische Operatoren implementieren einen logischen Operator
  - Mehrere Implementierungen pro Operator

Tabellen Scannen

7

- Einfachste Operation
- Gesamte Relation einlesen
  - Join, Union, …
- Gegebenenfalls kombiniert mit Selektionsbedingung
- Zwei Varianten
  - *Table-scan*: Blöcke liegen in einer (bekannten) Region der Festplatte. Lies alle Blöcke ein
  - *Index-scan*: Index besagt, welche Blöcke zur Relatione gehören und wo diese liegen
    - Hier Kombination mit Selektionen besonders effizient (-> später)

Sortiertes Einlesen

8

Sortiertes Einlesen von Relationen kann nützlich sein:

- ORDER BY in der Anfrage
- Spätere Operatoren nutzen Sortierung aus
- *Sort-scan*:
  - Gegeben Sortierschlüssel (ein oder mehr Attribute + Sortierreihenfolge)
  - Gegeben Relation
  - Gebe gesamte Relation sortiert zurück
- Implementierungsvarianten
  - B-Baum mit Sortierschlüssel als Suchschlüssel
  - Sequentielle Datei, sortiert nach Sortierschlüssel
  - Relation ist klein und kann im Hauptspeicher sortiert werden
    - Table-scan + Sortierung
    - Index-scan + Sortierung
  - Relation ist groß: TPMMS
    - Ausgabe nicht auf Festplatte sondern als Iterator

Berechnungsmodell

9

- Kosten eines Operators
  - Nur I/O-Kosten werden gezählt
  - CPU-Kosten werden von I/O-Kosten dominiert
  - Ausnahme: Netzwerkübertragung -> nicht hier
- Annahme
  - Input eines Operators wird von Disk gelesen
  - Output eines Operators muss nicht auf Disk geschrieben werden.
    - Falls letzter Operator im Baum:
      - » Anwendung verarbeitet Tupel einzeln
      - » I/O Kosten hängen von Anfrage ab, soweiso nicht vom Plan
    - Falls innerer Operator: Pipelining möglich

Kostenparameter / Statistiken

10

- Verfügbarer Hauptspeicher für einen Operator: M Einheiten
  - Eine Einheit entspricht Blockgröße auf Festplatte
  - Hauptspeicherverbrauch nur für Input und Operator, nicht für Output
  - Meist: M entspricht fast gesamtem Hauptspeicher
  - Kann dynamisch (während Anfragebearbeitung) bestimmt werden
  - Deswegen: M ist nur Schätzung
    - => Gesamtkosten sind nur geschätzt
    - => Gewählter Plan nicht unbedingt optimal

Kostenparameter / Statistiken

11

- Anzahl Blocks: *B*
  - Anzahl benötigt Blocks einer Relation: *B(R)*
  - Annahme: *B(R)* = Anzahl tatsächlich belegter Blocks
- Anzahl Tupel: *T*
  - Anzahl Tupel einer Relation: *T(R)*
  - *T/B* = Anzahl Tupel pro Block
- Anzahl unterschiedlicher Werte: *V*
  - Anzahl unterschiedlicher Werte einer Relation im Attribut *a*: *V(R,a)*
  - DISTINCT values
  - $V(R, [a_1,a_2,...,a_n]) = \delta(\pi_{a1,a2,...,an}(R))$

Scan Kosten - Beispiele

12

- R geclustered
  - Table-scan: Kosten *B*
  - Sort-scan
    - Kosten *B* falls *R* in Hauptspeicher passt
    - Kosten 3*B*, falls TPMMS nötig
- R nicht geclustered (also verteilt zusammen mit Tupeln anderer Relationen)
  - Table-Scan: Kosten *T*
  - Sort-scan
    - Kosten *T* falls R in Hauptspeicher passt
    - Kosten *T* + 2*B* falls TPMMS nötig
- Index-scan
  - Annahme: Kosten *B* bzw. T, auch wenn Index selbst einige Blöcke groß ist

Iteratoren

HPI Hasso Plattner Institut

13

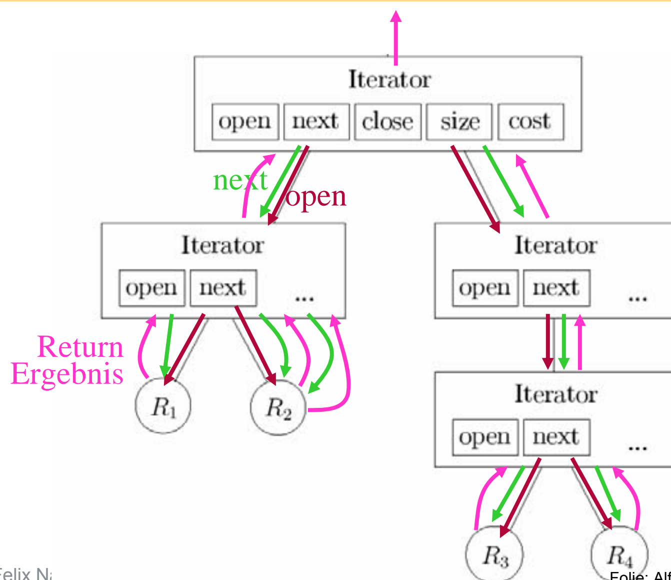
Viele physische Operatoren werden als Iterator implementiert.

- **`Open()`**
  - Öffnet Iterator, initialisiert Datenstrukturen
  - Ruft wiederum Open für Input-Operator(en) auf
  - Holt noch kein Tupel
- **`GetNext()`**
  - Holt nächstes Tupel
  - Ruft wiederum GetNext für Input-Operator(en) auf
  - Falls kein Tupel mehr vorhanden: NotFound
- **`Close()`**
  - Beendet und schließt Iterator
  - Ruft wiederum Close für Input-Operator(en) auf

Pull-basierte Anfrageauswertung

HPI Hasso Plattner Institut

14



Folie: Alfons Kemper, TU München

Iterator – Beispiel

15

- Open()
  - b := the first block of R;
  - t := the first tuple of block b;
- GetNext()
  - If (t is past the last tuple on block b)
    - Increment b to the next block;
    - If (there is no next block)
      - » RETURN NotFound;
    - ELSE
      - » t := first tuple on block b;
  - oldt := t;
  - increment t to the next tuple of b;
  - RETURN oldt;
- Close()
  - Do Nothing

Annahme: durch Pointer implementiert

Annahme: durch Block-organisation implementiert

Frage: Was wird hier implementiert?

Iterator – Beispiel

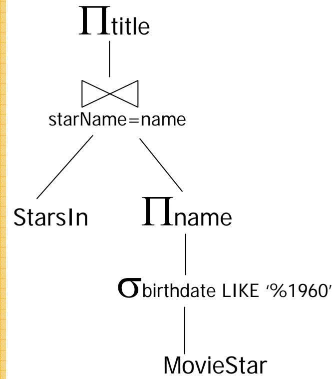
16

```
Open(R,S) {
        R.open();
        CurRel := R;
}
GetNext(R,S) {
        IF (CurRel = R) {
                t := R.GetNext();
                IF(t <> NotFound) /*R ist nicht erschöpft*/
                    RETURN t;
                ELSE /*R ist erschöpft*/ {
                   S.Open();
                   CurRel := S;
        }}
        RETURN S.GetNext();
}
Close(R,S) {
        R.Close();
        S.Close()
}
```

Was passiert falls S erschöpft ist?

Frage: Was wird hier implementiert?

Iterator – Beispiele



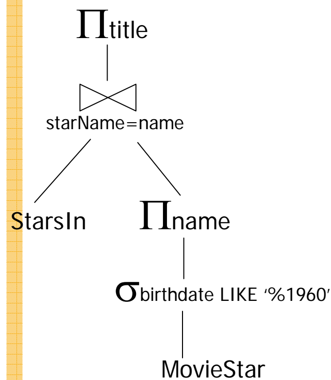

```
p = projection.Open();
while (t <> NotFound)
  t = p.GetNext()
  return t;
p.Close();

class projection {
Open() {
  j = join.Open();
  while (t <> NotFound)
    t:=j.GetNext()
    tmp[i++]=t.title;
  j.Close();
}
GetNext( ) {
  if (cnt < tmp.size())
    return tmp[cnt++];
  else return NotFound;
}
Close() {
  discard(tmp);
  j.Close();
}
}
```

```
class join {
Open() {
  l = table.open();
  while (tl <> NotFound)
    tl = l.GetNext();
    r = projection.Open();
    while (tr <> NotFound)
      tr = r.GetNext();
      if tl.starname=tr.name
        tmp[i++]=tl⋈tr;
    end while;
    l.Close();
  end while;
  r.Close();
}
GetNext( ) {
  if (cnt < tmp.size())
    return tmp[cnt++];
  else return NotFound;
}
Close() {
  discard(tmp);
  Close();
}
}
```

Folie: Ulf Leser, HU Berlin

Iterator – Beispiele




```
p = projection.Open();
while (t <> NotFound)
  t = p.GetNext()
  return t;
p.Close();

Class projection {
Open() {
  j = join.Open();
}
GetNext() {
  t =  j.GetNext();
  return t.title
}
close() {
  j.Close();
}
}
```

```
Class join {
Open() {
  l = table.Open();
  r = projection.Open()
  tl = l.GetNext();
}
GetNext() {
  tr = r.GetNext();
  if (tr <> NotFound)
    if tl.starname=tr.name
      return tl⋈tr;
    else
    tl = l.GetNext()
    if (tl <> NotFound)
       return GetNext();
    else
      return NotFound;
}
Close() {
  l.Close;
  r.Close();
}
}
```

Folie: Ulf Leser, HU Berlin

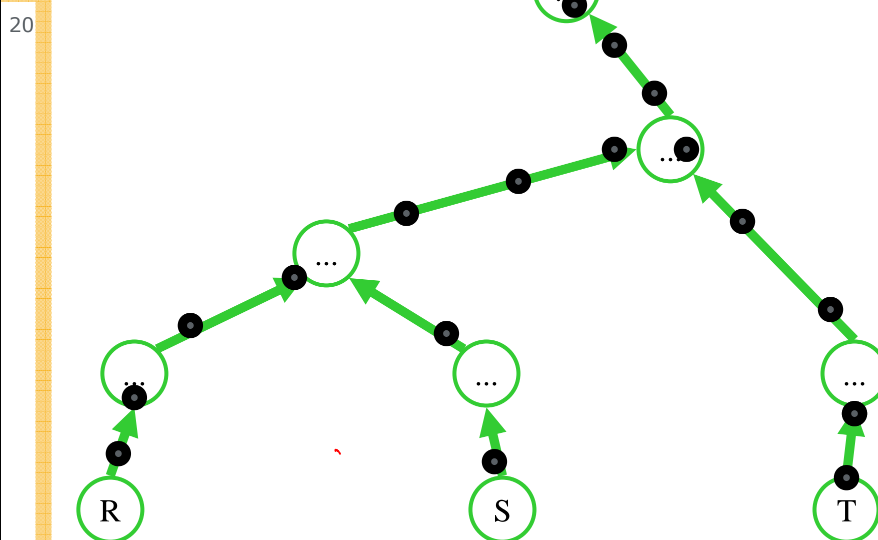
Pipelined versus Blocked

19

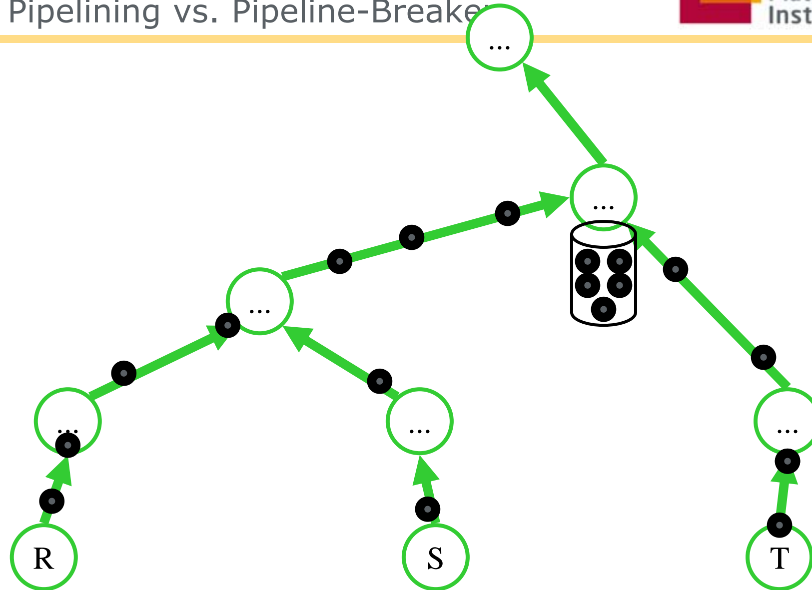
- Pipelining ist im allgemeinen sehr vorteilhaft.
  - Kein Puffern großer Zwischenergebnisse auf Festplatte
  - Operationen können auf Threads und CPUs verteilt werden
- Pipeline breaker
  - Sortierung:
    - next() kann erst ausgeführt werden wenn gesamte Relation gesehen wurde.
    - Ausnahme: Input ist bereits sortiert
  - Gruppierung und Aggregation
    - Implementiert durch Sortierung oder Hashing
    - Dann führt next() die Aggregation für eine Gruppe aus
  - Minus, Durchschnitt
- Projection mit Duplikateliminierung
  - Nicht unbedingt pipeline breaker
  - next() kann früh Ergebnisse weiterreichen (Sortierung nicht nötig)
  - Aber: Man muss sich alle bereits gelieferten Ergebnisse merken (großer Zwischenspeicher)

Pipelining vs. Pipeline-Breaker

20



Folie: Alfons Kemper, TU München

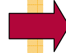


Überblick über das weitere

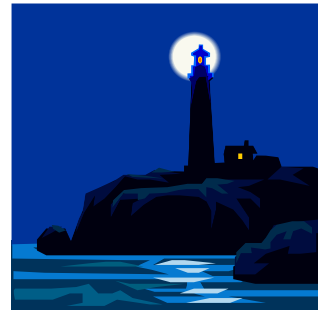
- Drei Klassen von Algorithmen
  - Sort-basierte Algorithmen
  - Hash-basierte Algorithmen
  - Index-basierte Algorithmen
- Drei Schwierigkeitsgrade von Algorithmen
  - One-Pass Algorithmen
    - Daten nur einmal von Disk lesen
    - Mindestens ein Argument passt in Hauptspeicher
      - außer Selektion und Projektion
  - Two-Pass Algorithmen
    - Meist einmal lesen, einmal schreiben, nochmal lesen
    - TPMMS
    - Gewisse Größenbeschränkung auf Input
  - Multipass Algorithmen
    - Unbeschränkt in Inputgröße
    - Rekursive Erweiterungen von Two-Pass Algorithmen
  - U.a. abhängig vom Operator

Überblick

23



- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement



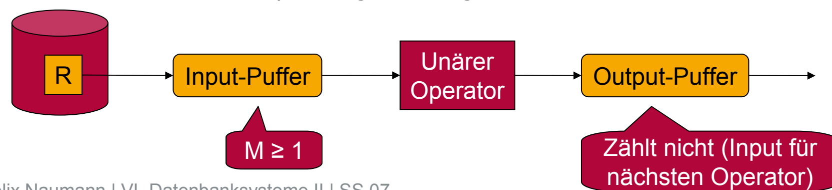
Operatorklassen

24

- Tupel-basierte unäre Operatoren
  - Benötigen jeweils nur sehr kleinen Teil des Input gleichzeitig im Hauptspeicher
  - Projektion, Selektion, Multimengen-Vereinigung
- Relationen-basierte unäre Operatoren
  - Benötigen gesamte Relation im Hauptspeicher
  - Deshalb Beschränkung der Inputgröße auf Hauptspeichergröße
  - Gruppierung, Duplikateliminierung
- Relationen-basierte binäre Operatoren
  - Benötigen mindesten eine gesamte Relation im Hauptspeicher
    - – Falls sie one-pass sein sollen
  - Alle Mengeoperatoren (außer Multimengen-Vereinigung)

Tupel-basierte unäre Operatoren

- Algorithmus für Selektion und Projektion offensichtlich
  - Unabhängig von Hauptspeichergröße
- Speicherkosten: 1
- I/O Kosten: Wie table-scan oder index-scan
  - B, falls geclustert
  - T, falls nicht geclustert
  - Weniger, falls Selektion auf Suchschlüssel eines Index
- Puffer > 1 nützlich. Wieso?
  - „Daten gemäß Zylinder organisieren"
  - Alle Blocks eines Zylinder gleichzeitig lesen.



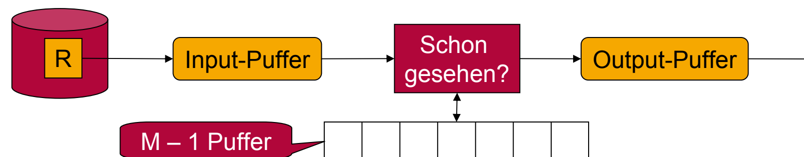
Relationen-basierte unäre Operatoren

- Operatoren: Duplikateliminierung und Gruppierung
  - Ganze Relation muss in den Hauptspeicher passen
- Genereller „Trick": Bewahre nur „Repräsentanten" im Hauptspeicher
  - Duplikateliminierung: Eindeutige Repreäsentation schon gesehener Tupel
  - Gruppierung: Gruppierungsattribute und aggregierte Teilergebnisse

Duplikateliminierung

27

- Tupel für Tupel einlesen
  - Erstes Mal dieses Tupel gesehen -> Ausgabe
  - Schon mal gesehen -> nix tun
- Puffer merkt sich welche Tupel bereits gesehen wurden
  - Datenstruktur wichtig (trotz I/O Dominanz)
    - Einfügen eines Tupels und Finden eines Tupel in fast konstanter Zeit
    - Z.B. Hashtabelle, balancierter Binärbaum
    - Geringer Speicher-overhead
- Wahl von M: $B(\delta(R)) \leq M$



Gruppierung

28

Idee: Erzeuge im Hauptspeicher einen Eintrag pro Gruppe

- Also ein Eintrag pro Gruppierungswert
- Dazu: Kumulierte Werte für aggregierte Attribute
  - Einfach: MIN/MAX, COUNT, SUM
  - Schwerer: AVG (Warum?)
    - AVG ist nicht assoziativ.
    - Merke COUNT und SUM
    - AVG erst am Ende berechnen
- Wieder: Datenstruktur im Hauptspeicher ist wichtig.
- Output: Ein Tupel pro Eintrag
  - Output erst nachdem letzter Input gesehen wurde (Blockierend)
- Hautspeicherkosten: Schwer abzuschätzen
  - Einträge selbst können größer oder kleiner als Tupel sein
  - Anzahl der Einträge höchstens so groß wie T
  - Meistens M << B

Relationen-basierte binäre Operatoren

- Vereinigung, Schnittmenge, Differenz, Kreuzprodukt, Join
  - Annahme: Eine Seite passt in Hauptspeicher
    - Außer $\cup_B$
    - Wieder: Effiziente Datenstruktur sinnvoll
    - Hauptspeicherbedarf: $M \geq \min(B(R), B(S))$
      - Hier: $B(S) < B(R)$
  - Unterscheidung: Multimengensemenatik (z.B. $\cup_B$) vs. Mengensemantik ($\cup_S$)
- $R \cup_B S$ trivial
  - I/O-Kosten: $B(R) + B(S)$
  - Hauptspeicherbedarf: 1

Relationen-basierte binäre Operatoren

- $R \cup_S S$
  - Lese alle Tupel aus S und baue Datenstruktur auf
    - Schlüssel ist gesamtes Tupel
  - Gebe alle diese Tupel aus
  - Lese R ein
    - Falls schon vorhanden: Nix tun
    - Fall nicht: Ausgeben
- $R \cap_S S$
  - Zunächst wie $R \cup_S S$ aber keine Tupel ausgeben
  - Lese R ein
    - Falls vorhanden: Ausgabe
    - Falls nicht vorhanden: Nix tun
  - Annahme: R und S sind Mengen

Relationen-basierte binäre Operatoren

31

- Mengen-Differenz
  - Nicht kommutativ!
  - Zunächst: Lese S in effiziente Datenstruktur ein
    - Gesamtes Tupel ist Schlüssel
  - Annahme: R und S sind Mengen
- $R -_S S$
  - Lese R ein
    - Falls Tupel schon vorhanden: Nix tun
    - Falls nicht vorhanden: Ausgabe
- $S -_S R$
  - Lese R ein
    - Falls Tupel schon vorhanden: Lösche aus Datenstruktur
    - Falls nicht vorhanden: Nix tun
  - Gebe übrig gebliebenen Tupel aus.

Relationen-basierte binäre Operatoren

32

- $R \cap_B S$
  - Lese S ein
    - Merke einen COUNT-Wert pro Tupel
    - Kann etwas mehr Speicher kosten (i.d.R weniger)
  - Lese R ein
    - Falls nicht bereits vorhanden: Nix tun
    - Falls vorhanden und COUNT > 0: Ausgabe und COUNT reduzieren
    - Sonst: Nix tun
- Multimengendifferenz
  - $S -_B R$
    - Lese S ein und speichere ein COUNT-Wert
    - Lese R ein
      - » Falls Tupel schon vorhanden: Veringere COUNT
      - » Falls nicht vorhanden: Nix tun
    - Gebe Tupel mit COUNT > 0 entsprechend oft aus.
  - $R -_B S$
    - Lese S ein und speichere ein COUNT-Wert (*c*)
      - » *c* Gründe ein Tupel aus R nicht auszugeben
    - Lese R ein
      - » Falls Tupel schon vorhanden und COUNT > 0: COUNT verringern
      - » Falls Tupel schon vorhanden und COUNT = 0: Ausgabe
      - » Falls nicht vorhanden: Ausgabe

Relationen-basierte binäre Operatoren

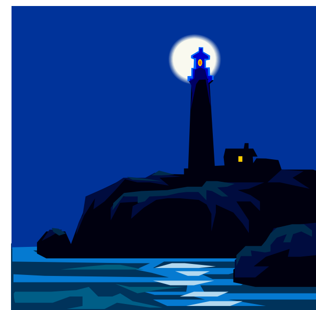
33

- R × S
  - Lese S in Hauptspeicher ein
    - Datenstruktur egal
  - Lese R ein
    - Konkateniere mit jedem Tupel aus S
    - Ausgabe
  - Rechenzeit pro Tupel lang: Ausgabe ist eben groß
- R(X,Y) ⋈ S(Y,Z) (natural join)
  - Lese S in Hauptspeicher ein
    - Y als Suchschlüssel
  - Lese R ein
    - Für jedes Tupel, suche passende Tupel aus S und gebe aus
  - I/O Kosten: B(R) + B(S)
  - Annahme: B(S) ≤ M-1 bzw. vereinfacht: B(S) ≤ M
  - Equi-join analog
  - Theta-join: Kreuzprodukt + Selektion

Überblick

34

- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement



Nested-Loop-Join-Algorithmen (NLJ)

35

- 1,5-pass Algorithmen
  - Eine Relation nur einmal einlesen
  - Die andere Relation mehrfach einlesen
- Größe beider Relationen beliebig
- Tupel-basierte Variante
  - Naiv
  - ```
    FOR EACH TUPLE s IN S DO
        FOR EACH TUPLE r IN R DO
            IF (r.Y = s.Y) THEN OUTPUT (r ⋈ s)
    ```
  - I/O-Kosten: T(R)·T(S)
  - Verbesserungen
    - Index auf Joinattribut in R (später)
    - Aufteilung der Tupel auf Blöcke berücksichtigen (gleich)

Iterator für Tupel-basierten NLJ

36

```
Open(R,S) {
        R.Open();
        S.Open();
        s := S.GetNext();
}

Close(R,S) {
        R.Close();
        S.Close();
}
```

```
GetNext(R,S) {
  REPEAT {
        r := R.GetNext();
        IF (NOT Found ) {
        /* R is exhausted for the current s */
          R.Close();
          s := S.GetNext();
          IF (Not Found) RETURN;
        /* both R and S are exhausted */
          R.Open();
          r := R.GetNext();
          }
        }
  UNTIL( r.Y = s.Y )
  RETURN join of r and s;
  }
```

Vorteil: Pipelining
```

Block-basierter NLJ

37

Ideen

- Organisiere Tupel nach Blöcken
  - Sinnvoll für innere Schleife
- Nutze Hauptspeicher um so viel wie möglich von S (äußere Schleife) zu halten
  - Ein R-Tupel wird nicht nur mit einem, sondern mit vielen S-Tupeln verjoint.
- Annahmen
  - B(S) ≤ B(R) (wie bisher)
  - B(S) > M (schwieriger als bisher)
  - Effiziente Datenstruktur für S im Hauptspeicher

Block-basierter NLJ

38

```
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;
```

Drei Schleifen?

Block-basierter NLJ – Kosten

39

- B(R) = 1.000
- B(S) = 500
- M = 101
- => 5x äußere Schleife á 100 I/O
- => jeweils 1.000 I/O für R
- = 5.500 I/O

- Nun: R in äußerer Schleife
  - => 10x äußere Schleife á 100 I/O
  - => jeweils 500 I/O für S
  - = 6.000 I/O

- => Kleinere Relation sollte außen sein.

```
FOR EACH chunk of M-1 blocks of S DO BEGIN
  read blocks into main memory;
  organize tuples into 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 memory that join;
       output those joined tuples;
    END;
  END;
END;
```

- Extremfall 1
  - B(S) = 100
  - B(R)= 1.000.000
  - 10.000x äußere Schleife á 100 + 100 I/O
  - = 10.000 x 200 = 2.000.000 I/O
- Extremfall 2
  - 1x äußere Schleife á 100 + 1.000.000 I/O
  - = 1x 1.000.100 I/O

Block-basierter NLJ – Kosten

40

Allgemeinere Berechnung

- Äußere Schleife: B(S)/(M-1)-fach
- Jeweils
  - M-1 Blöcke von S
  - B(R) Blöcke von R
- Zusammen

$$\frac{B(S)}{M-1}(M-1+B(R))$$

$$= B(S) + \frac{B(S)B(R)}{M-1}$$

$$\approx B(S)B(R)/M$$

```
FOR EACH chunk of M-1 blocks of S DO BEGIN
  read blocks into main memory;
  organize tuples into 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 memory that join;
       output those joined tuples;
    END;
  END;
END;
```

Zusammenfassung bisheriger Algorithmen

41

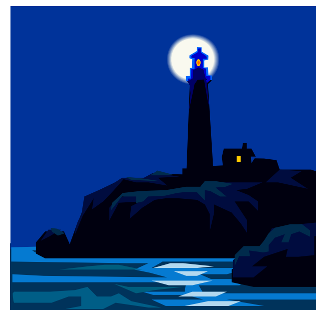
Operator	Nötiger Hauptspeicher M	I/O	Algorithmus
σ, π	1	B	1-pass, tupel-basiert
γ, δ	≈ B	B	1-pass, relationen-basiert
∪, ∩, −, ×, ⋈	min(B(S), B(R))	B(R) + B(S)	1-pass, relationen-basiert binär
⋈	M ≥ 2	B(R)B(S)/M	Block-basierter NLJ

Felix Naumann | VL Datenbanksysteme II | SS 07

Überblick

42

- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement



Felix Naumann | VL Datenbanksysteme II | SS 07

1-, 2-, Mehr-Phasen

43

- Bisher: One-Pass Algorithmen; eine Relation passt in Hauptspeicher
- Nun: Two-Pass Algorithmen; keine Relation passt in Hauptspeicher
- Zwei Phasen
  - Einlesen der Daten
  - Verarbeitung der Daten (hier: Sortierung von Teillisten)
  - Schreiben der Daten
  - Wiedereinlesen der Daten (hier: Merging der Teillisten)
    - Hier unterscheiden sich die Algorithmen
- Mehr-Phasen?
  - Zwei Phasen reichen meist
  - Verallgemeinerung zu Mehr-Phasen einfach

Duplikateliminierung

44

Idee: Ähnlich wie TPMMS

- …
- Ein Block pro sortierter Teilliste
  - Betrachte jeweils erstes Tupel
  - Suche kleinstes Tupel
  - Gib ein dieses Tupel aus; verwerfe alle anderen identischen Tupel
- Beispiel
  - M = 3; 2 Tupel pro Block
  - 17 Tupel: 2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3
  - Phase 1: 3 sortierte Teillisten
  - Phase 2: s.o.

Duplikateliminierung – Kosten

45

- Wie TPMMS
  - B(R) für Einlesen in Phase 1
  - B(R) für Schreiben der Teillisten
  - B(R)für Lesen der Teillisten
  - Zusammen: 3·B(R)
- One-pass Algorithmus: 1·B(R)
- Aber größerer Input möglich
  - One-pass: $B \leq M$
  - Two-pass: $B \leq M^2$

Gruppierung und Aggregation

46

- Phase 1
  - Lese R ein (jeweils M Blöcke)
  - Sortiere M Blöcke nach Gruppierungsattribut(en)
  - Schreibe sortierte Teillisten
- Phase 2
  - Lade jeweils einen Block jeder Teilliste
  - Suche kleinste Schlüssel (neue Gruppe)
  - Aggregiere alle Tupel mit diesem Schlüssel
    - Gegebenenfalls Blöcke nachladen
  - Gebe ein Tupel mit aggregierten Werten (und gegebenenfalls Gruppierungsattribut) aus.
  - Suche nächsten kleinsten Schlüssel
- I/O-Kosten: 3B(R)   Maximale Größe: $B(R) \leq M^2$

Vereinigung

47

- Lese R ein und schreibe sortierte Teillisten
  - Sortierschlüssel ist gesamtes Tupel
- Lese S ein und schreibe sortierte Teillisten
  - Sortierschlüssel ist gesamtes Tupel
- Lese jeweils einen Block aus beiden Mengen sortierter Teillisten
- Suche kleinste Tupel aus allen Blöcken
  - -> Ausgabe
  - Entfernung aus allen anderen Teillisten
    - Zur Not Blöcke nachladen
- Funktioniert für Mengen und Multimengen
  - Bei Multimengen ist one-pass Algorithmus besser
- I/O-Kosten: 3(B(R) + B(S))
- Maximale Größe: $B(R) + B(S) \leq M^2$

Schnittmenge und Differenz

48

- Sortierung und Laden der Teillisten wie bei Vereinigung
- Suche kleinstes Tupel *t*
- count(R, *t*) = Anzahl der Vorkommen von *t* in R
- count(S, *t*) analog
  - Gegebenenfalls nachladen
- $\cap_S$: Ausgabe von *t* falls count(R, *t*) > 0 und count(S, *t*)>0
- $\cap_B$: Ausgabe von *t* min(count(R, *t*), count(S, *t*)) mal
  - Gegebenenfalls nicht ausgeben (wenn ein count = 0)
- $R -_S S$: Ausgabe von *t* falls count(R, *t*) > 0 und count(S, *t*) = 0
- $R -_B S$: Ausgabe von *t* count(R, *t*) – count(S, t) mal
- I/O-Kosten: 3(B(R) + B(S))
- Maximale Größe: $B(R) + B(S) \leq M^2$

Einfacher Sort-basierter Join Algorithmus

49

- Neues Problem: Alle Tupel mit gleichem Joinattributwert müssen gleichzeitig im Hauptspeicher sein.
- Lösungsidee: Reserviere so viel Speicher wie möglich für aktuelle Jointupel
  - Reduziere Speicherbedarf anderer Algorithmusteile
- $R(X, Y) \bowtie S(Y, Z)$
- Sortiere R und S gemäß Y mit TPMMS
  - Inkl. letzter Phase (Schreiben des sortierten Ergebnisses)
- Merge R und S
  1. Jeweils ein Block
  2. Suche kleinstes Y in beiden Blocks
  3. Falls nicht in anderem Block vorhanden: Entferne alle Tupel mit diesem Y
  4. Falls vorhanden: Identifiziere alle Tupel mit diesem Y
     - Gegebenenfalls nachladen
  5. Gebe alle Kombinationen aus
  6. Lade gegebenenfalls Tupel nach

Einfacher Sort-basierter Join Algorithmus – Kosten

50

- R: 1000 Blocks; S: 500 Blocks; M = 101
- TPMMS: $4(B(R) + B(S)) = 4 \cdot 1500 = 6000$ I/O
- Merging: Nochmals R und S lesen: 1500 I/O
  - Nur 2 Blocks werden benötigt
  - Aber: Alle Tupel mit einem bestimmten Y-Wert müssen in 101 Blöcke passen
- I/O: $5(B(R) + B(S))$; Hauptspeicher: $B(R) \leq M^2$ und $B(S) \leq M^2$
- Vergleich zu nested loops: 5500 I/O
  - Aber nested loops ist quadratisch
  - Sort-based join ist linear
  - Gleich noch Verbesserung auf $3(B(R) + B(S))$

Felix Naumann | VL Datenbanksysteme II | SS 07

Einfacher Sort-basierter Join Algorithmus – Erweiterung

- Falls alle Tupel mit einem bestimmten Y-Wert nicht in Hauptspeicher passen
  - Falls alle solche Tupel einer Relation in M-1 Blöcke passen
    - One-pass join
  - Falls nicht
    - Nested loop join
- Fallunterscheidung kann überflüssiges I/O kosten.
- Analyse
  - Y ist oft in einer Relation ein Schlüssel ist => leicht
  - Oft sind viele Speicherblöcke übrig, da $B(R)+B(S) << M^2$

Sort-basierter Join Algorithmus – Verbesserung

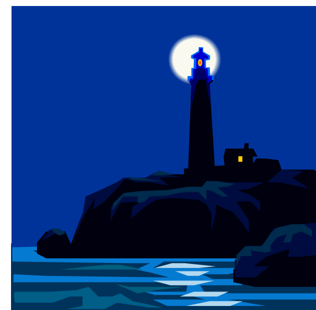
- Idee: Kombiniere 2te Phase des TPMMS mit dem Joinen
  - => „sort-join", „merge-join", „sort-merge-join"
- Annahmen
  - Anzahl Teillisten ≤ M
  - Tupel mit gemeinsamen Y-Wert passen zusammen in verbleibenden Hauptspeicher
- R: 1000 Blocks; S: 500 Blocks; M = 101
  - Phase 1: 10 Teillisten für R, 5 Teillisten für S
  - Phase 2: 15 Blöcke gleichzeitig im Hauptspeicher
    - => 86 freie Blöcke für aktuelle Join-Tupel
  - Zusammen 3(B(R)+ B(S)) = 4500 I/O

Zusammenfassung – sortbasierte, two-pass Algorithmen

Operators	Approximate M required	Disk I/O
γ, δ	$\sqrt{B}$	3B
∪,∩, -	$\sqrt{B(R)+B(S)}$	3(B(R) + B(S))
$\bowtie$	$\sqrt{\max(B(R),B(S))}$	5(B(R) + B(S))
$\bowtie$	$\sqrt{B(R)+B(S)}$	3(B(R) + B(S))

Überblick

- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement



Grundidee

55

- Input passt nicht in Hauptspeicher
- Hashe alle Inputargument
  - Tupel, die gemeinsam betrachtet werden müssen, erhalten gleichen Hashwert
  - Landen also in einem Bucket
- Unäre Operatoren: Bearbeite einen Bucket nach dem anderen
- Binäre Operatoren: Bearbeite Paare von Buckets

- Reduktion des Speicherbedarfs um Faktor M
  - Verwende M Buckets

Partitionierung mittels Hashing

56

- Grundalgorithmus
- Gegeben M Puffer, verteile R auf M-1 Buckets
  - Möglichst gleicher Größe
- Ein Bucket pro Puffer
- Letzter Puffer für Tupel aus R
- Idee
  - Für jedes Tupel aus R berechne h(t) und kopiere Tupel in entsprechenden Bucket
  - Falls voll: Schreibe auf Overflowblock auf Disk
  - Am Ende: Schreibe auch alle Buckets auf Disk

Partitionierung mittels Hashing

57

```
Initialize M-1 buckets using M-1 empty buffers;
FOR each block b of R DO BEGIN
  Read block b into M-th buffer
  FOR each tuple t in b DO BEGIN
        IF buffer for bucket h(t) has no room for t THEN
                BEGIN
                  Copy the buffer to disk;
                  Initialize a new empty block in that buffer;
                END;
        Copy t to buffer for bucket h(t);
  END;
END;
FOR each bucket DO
  IF the buffer for this bucket is not empty THEN
        Write the buffer to disk;
```


Duplikateliminierung

58

- Algorithmus wie eben
- Duplikate landen im gleichen Bucket
- Betrachte jeden Bucket einzeln
  - Duplikateliminierung innerhalb des Buckets
  - Danach Vereinigung aller Buckets
- Annahme: Alle Blöcke eines Buckets passen in Hauptspeicher
  - => One-pass Algorithmus funktioniert
  - Bei Gleichverteilung durch *h*: Bucket hat B(R)/(M-1) Blöcke
  - => R darf bis zu M(M-1) viele Blöcke umfassen
  - Vermutlich noch besser (wie zuvor): Es müssen nur distinct Tupel in Hauptspeicher passen
- I/O-Kosten: 3·B(R)

Gruppierung und Aggregation $\gamma_L(R)$

- Grundalgorithmus wie zuvor
- Aber: Hashfunktion hängt nur von Gruppierungsattributen ab
- Dann: One-pass Algorithmus für Gruppierung auf jedem Bucket
- Hauptspeicherbedarf: $B(R) \leq M^2$
  - Vermutlich besser: Nur ein Tupel pro Gruppe im Hauptspeicher
- I/O-Kosten: $3 \cdot B(R)$

Mengenoperationen

- Bei binären Operationen: Gleiche Hashfunktion für beide Inputs!
- Mengenvereinigung:
  - Hashe R und S jeweils auf M-1 Buckets
  - Bilde Mengenvereinigung passender Bucketpaare
- Multimengenvereinigung: Voriger Algorithmus
- Wieder: Jeweils Hashen und dann jeweiligen One-pass Algorithmus anwenden
- Speicherbedarf: $\min(B(R), B(S)) \leq M^2$
  - Da bei One-pass Varianten kleinere Relation in Hauptspeicher passen muss
- I/O-Kosten: $3 \cdot (B(R) + B(S))$

Hashjoin

61

- Algorithmus wie zuvor
- Aber: Hashschlüssel sind Joinattribute
  - Tupel mit gleichen Joinattributwerten landen im gleichen Bucket.
- Danach One-pass Join Variante für jeden Bucket
- Beispiel von zuvor: B(R) = 1000, B(S) = 500, M = 101
- Hashing
  - Ca. 10 R-Blocks pro Bucket
  - Ca. 5 S-Block pro Bucket
- Min(10, 5) = 5 => One-pass Algorithmus klappt (5 < 101)
- I/O-Kosten:
  - 1500 für das Hashing + 1500 um buckets zu schreiben
  - 1500 um Buckets zu lesen
  - Zusammen: 3(B(R) + B(S)) = 4500 (wie sort-basierte Methode)

I/O Einsparungen

62

Grundidee: Nutze nicht verwendeten Speicher

- Idee 1: Verwende mehr als 1 Block pro Bucket
  - Effizienteres Schreiben (aber gleiche I/O-Kosten)
- Idee 2: Hybrid Hashjoin
  - Beim Hashen von S: Behalte *m* Buckets komplett im Speicher
    - Auch nach Ende des Hashens
    - Jeweils mit geeigneter Datenstruktur
  - Falls k Buckets insgesamt für s nötig sind: Behalte für die übrigen k-*m* Buckets nur ein Block im Hauptspeicher
  - $(m \cdot B(S)/k) + k - m \leq M$
  - Beim Hashen von R sind im Hauptspeicher: *m* Buckets für S und ein Block für die k-m Buckets von R
  - Falls t in einen der *m* Buckets gehasht wird
    - Joinpartner suchen
    - Gegebenenfalls direkte Ausgabe
  - Falls t in einen der k-*m* Buckets gehasht wird
    - Verfahre wie zuvor: Auf Disk schreiben
  - Phase 2 dann nur noch auf den k-*m* Buckets

Hybrid Hashjoin – Analyse

63

- Einsparungen: 2 I/Os für jeden Block, der im Hauptspeicher gehalten werden kann.
  - $2(m/k)$ (B(R) + B(S))
- => Maximiere (m/k), gegeben ($m \cdot B(S)/k$) + k – $m \leq M$
  - Lösung: Wähle m = 1 und minimiere k.
- Minimierung von k (gesamte Anzahl der Buckets): Wähle Bucketgröße so, dass ein Bucket gerade eben in Hauptspeicher passt
  - Bucketgröße M
  - => k = B(S) / M
  - Eigentlich etwas kleiner, damit die übrigen Buckets durch mindestens einen Block repräsentiert werden können
- => Einsparungen (2M / B(S)) · (B(R) + B(S))
- => I/O-Kosten: (3 – (2M/B(S))) · (B(R) + B(S))

Wähle wenige große Buckets statt viele kleine.

Hybrid Hashjoin – Beispiel

64

- B(R) = 1000, B(S) = 500, M = 101
- Wähle k = B(S) / M = 500 / 101 ≈ 5
  - => Ein Bucket hat ca. 100 Blocks
  - => 104 Hauptspeicher nötig
  - => Besser k = 6
- 1 Puffer für erste 5 Buckets und 96 Puffer für letzten Bucket
  - Erwartete Größe: 500/6 ≈ 83
- Phase 1
  - I/O-Kosten für S: 500x lesen und 417x schreiben
  - I/O-Kosten für R: 1000x lesen und 833x schreiben (5 der 6 Buckets)
- Phase 2
  - Alle geschriebenen Blöcke wieder lesen: 417 + 833 = 1250
- Zusammen: 500 + 1000 + 2·(417 + 833) = 4000 I/Os
  - < 4500 für einfachen Hash-Join bzw. Sort merge Join

Zusammenfassung Hash-basierter Verfahren

65

Operatoren	Hauptspeicher-bedarf	I/O-Kosten
γ, δ	$\sqrt{B}$	3B
∪, ∩, −	$\sqrt{B(S)}$	3(B(R) + B(S))
⋈	$\sqrt{B(S)}$	3(B(R) + B(S))
⋈	$\sqrt{B(S)}$	(3 − (2M/B(S))) · (B(R) + B(S))

Wdh.: **Sort**-basierte, two-pass Algorithmen

66

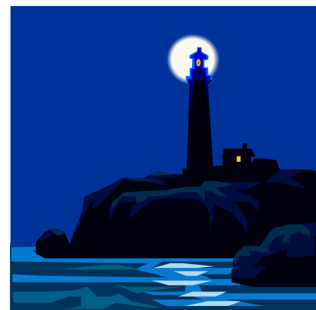
Operators	Approximate M required	Disk I/O
γ, δ	$\sqrt{B}$	3B
∪,∩, -	$\sqrt{B(R)+B(S)}$	3(B(R) + B(S))
⋈	$\sqrt{\max(B(R),B(S))}$	5(B(R) + B(S))
⋈	$\sqrt{B(R)+B(S)}$	3(B(R) + B(S))

Vergleich Hash-basierte und Sort-basierte Algorithmen

- Speicherbedarf und I/O-Kosten ähnlich
- Speicherbedarf Hash-basierter Verfahren hängt nur vom kleineren der beiden Inputs statt Summe der beiden Inputs ab.
- Sortier-basierte Verfahren produzieren oft ein sortierten Output
  - Vorteile später im Plan
- Sortierbasierte Verfahren können sortierte Teilliste hintereinander auf Disk schreiben
  - Spart bei einer I/O-Operation Seektime
  - Bei großem M: Auch mehrere Blöcke einer Liste auf einmal lesen
- Gleiches auch bei Hash-basierten Verfahren möglich, falls Anzahl Buckets kleiner als M

Überblick

- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement



Grundidee

69

Existenz von Indizes ermöglichen manchmal andere Algorithmen

- Insbesondere Selektion
- Auch: Joins und andere binäre Operatoren

Clustering

- Clustered Relation
  - Tupel auf so wenig wie möglich Blöcken auf Disk
- Cluster-Index
  - Tupel mit gleichem Schlüsselwert sind auf so wenig wie möglich Blöcken
  - Voraussetzung: Relation ist clustered
- Eine clustered Relation kann auch non-Cluster-Indizes haben.

Felix Naumann | VL Datenbanksysteme II | SS 07

Index-basierte Selektion

70

- Basisalgorithmus: Lese gesamte Relation ein und prüfe Bedingung
  - Ohne Index ist dies die beste Methode
  - I/O-Kosten: B(R) bzw. T(R) falls R nicht clustered
- Besser: Selektionsbedingung a=v und a ist Suchschlüssel eines Cluster-Indexes
  - I/O-Kosten: B(R)/V(R,a)
    - V(R,L) = Anzahl distinct Werte von π_L(R)
  - Eventuell mehr
    - I/O-Kosten für Index
    - Tupel nicht perfekt auf Blöcke verteilt: 1 Block extra
    - Blöcke nicht absolut vollgepackt
    - Fremde Tupel auf Blöcken
    - Aufrunden
      - » Z.B. a ist Schlüssel => V(R,a) = T(R) >> B(R)
      - » Dennoch mindestens 1 Block

Felix Naumann | VL Datenbanksysteme II | SS 07

Index-basierte Selektion

71

- Selektionsbedingung a=v und a ist Suchschlüssel eines nicht-Cluster-Indexes
- => Jedes Tupel auf anderen Block (vermutlich)
- Anzahl Tupel: T(R) / V(R,a)
  - Wieder zusätzliche I/O-Kosten: Indizes
  - Eventuell besser: Zufällig mehr als ein Tupel auf dem Block

Index-basierte Selektion – Beispiel

72

- Beispiel: B(R) = 1000, T(R) = 20000 (=> 20 Tupel pro Block)
  - $\sigma_{a=0}$(R); Index auf a
  - R ist clustered; Index wird nicht verwendet: 1000 I/Os
  - R nicht clustered; Index wird nicht verwendet: 20000 I/Os
  - V(R,a)=100; Index ist clustering: 1000/100 = 10 I/Os
  - V(R,a) = 10; Index ist nicht clustering: 20000/10 = 2000 I/Os
    - Falls R clustered: Lieber ganz R einlesen
  - V(R,a) = 20000 (Schlüssel): 1 I/O

Joining mit Index

73

- Natural Join: R(X,Y) ⋈ S(Y,Z)
- Algorithmus
  - S habe Index auf Y
  - Lese jeden Block in R.
  - Für jedes Tupel extrahiere Y-Wert und verwende Index um entsprechendes S-Tupel zu finden
- Kosten
  - Falls R clustered: B(R)
  - Für jedes der T(R) Tupel muss man durchschnittlich T(S)/V(S,Y) Tupel lesen
    - Falls Index nicht clustering ist: T(R)T(S)/V(S,Y)
    - Falls Index clustering: T(R)B(R)/V(S,Y) bzw. T(R)·max(1, B(R)/V(S,Y))
  - Größe von S dominiert Kosten

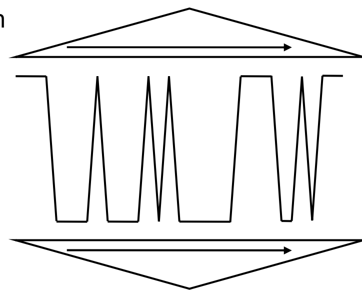
Joining mit Index – Beispiel

74

- B(R) = 1000, B(S) = 500, T(R) = 10000, T(S) = 5000
  - 10 Tupel pro Block
- V(S,Y) = 100 (also 100 distinct Y-Werte in S)
- R sei clustered; Index auf S[Y] sei clustering
- I/O-Kosten:
  - 1000 zum Lesen von R
  - 10000 · 500/100 = 50000 I/Os zum Vergleich mit S
- Diskussion
  - Klappt besser falls R sehr klein => Viele Blöcke von S werden nie angefasst
  - Hingegen bei Hash- und Sort-basierten Methoden wird immer ganz R und ganz S betrachtet

Joining mit sortiertem Index

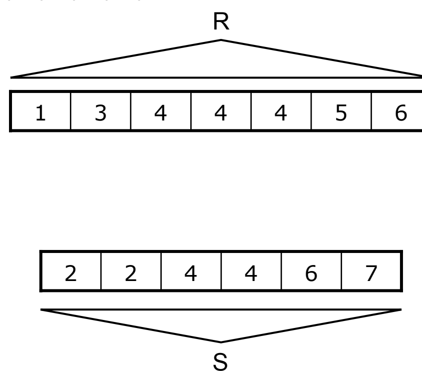
- Sortierter Index, z.B. B-Baum
- Idee 1: Sort-Merge-Join, aber nur eine Relation muss sortiert werden.
  - Falls beide Relationen sortierten Index auf Y haben: Nur noch Merge-Phase
  - Dann: Zig-Zag-Join
  - Tupel aus R ohne Joinpartner in S werden nie gelesen (und umgekehrt)



HPI Hasso Plattner Institut

Zig-Zag-Join – Beispiel

- Y-Werte in R: 1, 3, 4, 4, 4, 5, 6
- Y-Werte in S: 2, 2, 4, 4, 6, 7



Felix Naumann | VL Datenbanksysteme II | SS 07

Joining mit Indizes – Beispiel

Hasso Plattner Institut

77

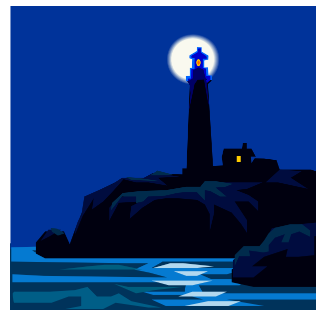
- Seien R und S clustered; S habe sortierten Index auf Y; R habe keinen Index
- 10 sortierte Teillisten für R: 2000 I/Os
- Nun 11 Puffer: Einen für jede Teilliste, einen für Blöcke aus S
  - Ganz R und ganz S werden gelesen: 1500 I/Os
- Zusammen 3500 I/O
  - Weniger als bisher
- Nun habe R auch einen Index
  - Sortierung ist egal: Zig-Zag-Join
  - Schlimmstenfalls nur ganz R und ganz S lesen: 1500 I/O
  - Bei wenigen Joinpartnern: Viel weniger I/Os

Überblick

Hasso Plattner Institut

78

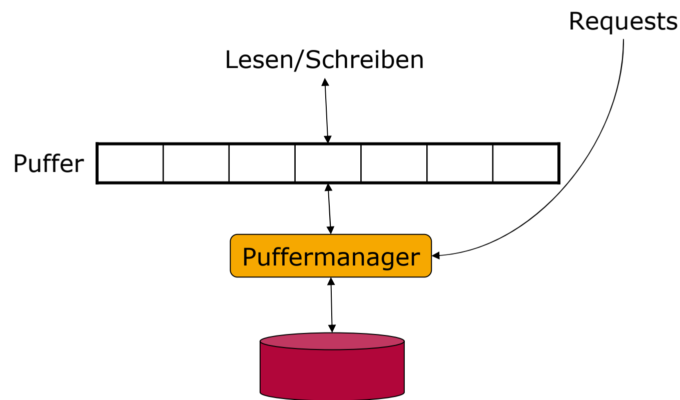
- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement



Felix Naumann | VL Datenbanksysteme II | SS 07

Puffermanager

79



Variante 1: Puffermanager verwaltet Hauptspeicher direkt.
Variante 2: Puffermanager verwaltet virtuellen Speicher und OS verwaltet physischen Hauptspeicher.

Puffermanager – Aufgaben

80

- Gesamtspeicher darf nicht überschritten werden
  - Verwaltung eines Bufferpools
  - Größe des Bufferpool bei Initialisierung festgelegt
- Gegebenenfalls Block im Speicher verwerfen
  - Löschen
  - Gegebenenfalls auf Disk schreiben
  - Verdrängungsstrategie

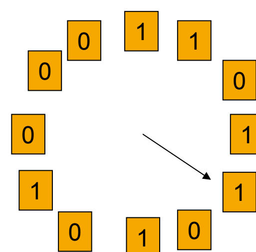
Verdrängungsstrategien

81

- LRU – Least Recently Used
  - Verwirft Block, der an längsten weder gelesen noch geschrieben wurde
  - Verwaltung der Lese- und Schreibzeiten mit jedem Zugriff nötig: Aufwändig!
- FIFO – First-In-First-Out
  - Puffer, der am längsten gleichen Block enthält wird geleert
  - Verwaltung einfacher: Nur erste Zeit muss gemerkt werden
  - Aber fehleranfällig: Wurzel eines B-Baums
- Clock Algorithmus
  - Häufig verwendet; approximiert LRU

Clock Algorithmus

82



- Auch „Second Chance"
- Puffer im „Kreis" angeordnet
- Puffer hat ein Bit als flag
  - 0: Puffer kann entleert werden
  - 1: Puffer kann nicht entleert werden
- Beim ersten Befüllen des Puffers: 1
- Bei Zugriff auf den Puffer: 1
- „Zeiger" wandert im Uhrzeigersinn, wenn Puffer benötigt wird
  - Falls flag = 1: Setzte Flag auf 0 und wandere weiter
  - Falls flag = 0: Entleere Puffer
- Variationen mit anderen Werten
  - Hohe Werte für besonders wichtige Seiten

Felix Naumann | VL Datenbanksysteme II | SS 07

Zusammenfassung

- Physische Operatoren
- One-Pass Algorithmen
- Nested Loop Join
- Sort-basierte Two-Pass Algorithmen
- Hash-basierte Two-Pass Algorithmen
- Index-basierte Algorithmen
- Puffermanagement