

Datenbanken 2

Anfragebearbeitung

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
FB Computerwissenschaften
Universität Salzburg



WS 2019/20
Version 29. Januar 2020

Inhalt

- 1 Einführung
- 2 Anfragekosten abschätzen
- 3 Sortieren
- 4 Selektion
- 5 Join

Literatur und Quellen

Lektüre zum Thema "Anfragebearbeitung":

- Kapitel 8 aus Kemper und Eickler: Datenbanksysteme: Eine Einführung. Oldenbourg Verlag, 2013.
- Chapter 12 in Silberschatz, Korth, and Sudarshan: Database System Concepts. McGraw Hill, 2011.

Danksagung Die Vorlage zu diesen Folien wurde entwickelt von:

- Michael Böhlen, Universität Zürich, Schweiz
- Johann Gamper, Freie Universität Bozen, Italien

Inhalt

- 1 Einführung
- 2 Anfragekosten abschätzen
- 3 Sortieren
- 4 Selektion
- 5 Join

PostgreSQL Beispiel/1

```

SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 7000000;

```

Output pane

QUERY PLAN	
text	
1	Aggregate (cost=1224.35..1224.36 rows=1 width=0)
2	-> Nested Loop (cost=5.14..1224.10 rows=100 width=0)
3	-> Bitmap Heap Scan on r1 r (cost=5.14..388.89 rows=100 width=4)
4	Recheck Cond: (unique1 > 7000000)
5	-> Bitmap Index Scan on i1 (cost=0.00..5.11 rows=100 width=0)
6	Index Cond: (unique1 > 7000000)
7	-> Index Scan using i3 on r2 s (cost=0.00..8.34 rows=1 width=4)
8	Index Cond: (s.unique1 = r.unique1)

Unix Ln 4 Col 23 Ch 85 8 rows. 8 ms

PostgreSQL Beispiel/2

```

SELECT COUNT(*)
FROM r1 r, r2 s
WHERE r.unique1 = s.unique1
AND r.unique1 > 7000000;

```

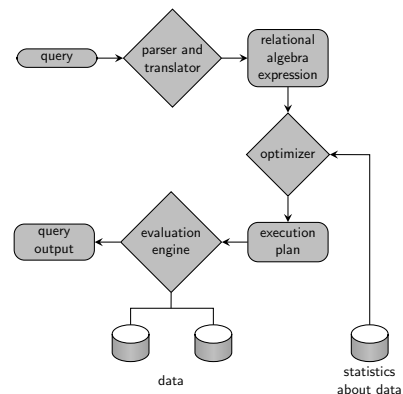
Output pane

Unix Ln 4 Col 23 Ch 85 8 rows. 8 ms

Anfragebearbeitung

- Effizienter Auswertungsplan gehört zu den wichtigsten Aufgaben eines DBMS.
- 3 Schritte der Anfragebearbeitung:

1. **Parsen und übersetzen**
(von SQL in Rel. Alg.)
2. **Optimieren**
(Auswertungsplan erstellen)
3. **Auswerten**
(Auswertungsplan ausführen)



Inhalt

- 1 Einführung
- 2 Anfragekosten abschätzen
- 3 Sortieren
- 4 Selektion
- 5 Join

Anfragekosten/1

- Anfragekosten werden als **gesamte benötigte Zeit** verstanden.
- Mehrere Faktoren tragen zu den Anfragekosten bei:
 - CPU
 - Netzwerk Kommunikation
 - Plattenzugriff
 - sequentielles I/O
 - random I/O
 - Puffergröße
- Puffergröße:
 - mehr Puffer-Speicher (RAM) reduziert Anzahl der Plattenzugriffe
 - verfügbarer Puffer-Speicher hängt von anderen OS Prozessen ab und ist schwierig von vornherein festzulegen
 - wir verwenden oft worst-case Anschätzung mit der Annahme, dass nur der mindest nötige Speicher vorhanden ist

Anfragekosten/2

- Plattenzugriff macht **größten Teil der Kosten** einer Anfrage aus.
- Kosten für Plattenzugriff **relativ einfach abzuschätzen** als Summe von:
 - Anzahl der Spurwechsel * mittlere Spurwechselzeit (avg. seek time)
 - Anzahl der Block-Lese-Operationen * mittlere Block-lese-Zeit
 - Anzahl der Block-Schreib-Operationen * mittlere Block-schreib-Zeit
 - Block schreiben ist teurer als lesen, weil geschriebener Block zur Kontrolle nochmal gelesen wird.
- Zur Vereinfachung
 - zählen wir nur die **Anzahl der Schreib-/Lese-Operationen**
 - berücksichtigen wir nicht die Kosten zum **Schreiben des Ergebnisses** auf die Platte

Inhalt

- 1 Einführung
- 2 Anfragekosten abschätzen
- 3 **Sortieren**
- 4 Selektion
- 5 Join

Sorting

- **Sortieren** ist eine wichtige Operation:
 - SQL-Anfragen können explizit eine sortierte Ausgabe verlangen
 - mehrere Operatoren (z.B. Joins) können effizient implementiert werden, wenn die Relationen sortiert sind
 - oft ist Sortierung der entscheidende erste Schritt für einen effizienten Algorithmus
- **Sekundärindex für Sortierung verwenden?**
 - Index sortiert Datensätze nur logisch, nicht physisch.
 - Datensätze müssen über Pointer im Index zugegriffen werden.
 - Für jeden Pointer (Datensatz) muss möglicherweise ein eigener Block von der Platte gelesen werden.
- **Algorithmen je nach verfügbarer Puffergröße:**
 - Relation kleiner als Puffer: Hauptspeicher-Algorithmen wie **Quicksort**
 - Relation größer als Puffer: Platten-Algorithmen wie **Mergesort**

Externes Merge-Sort/1

- Grundidee:
 - **teile** Relation in Stücke (Läufe, *runs*) die in den Puffer passen
 - **sortiere** jeden Lauf im Puffer und schreibe ihn auf die Platte
 - **mische** sortierte Läufe so lange, bis nur mehr ein Lauf übrig ist
- Notation:
 - b : Anzahl der Plattenblöcke der Relation
 - M : Anzahl der Blöcke im Puffer (Hauptspeicher)
 - $N = \lceil b/M \rceil$: Anzahl der Läufe

Externes Merge-Sort/2

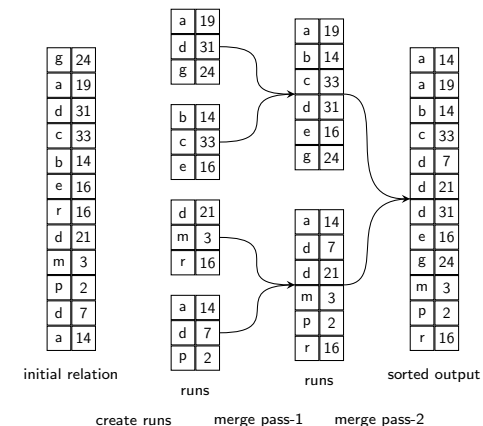
- Schritt 1: erzeuge N Läufe
 1. starte mit $i = 0$
 2. wiederhole folgende Schritte bis Relation leer ist:
 - a. lies M Blöcke der Relation (oder den Rest) in Puffer
 - b. sortiere Tupel im Puffer
 - c. schreibe sortierte Daten in Lauf-Datei L_i
 - d. erhöhe i
- Schritt 2: mische Läufe (N -Wege-Mischen) (Annahme $N < M$) (N Blöcke im Puffer für Input, 1 Block für Output)
 1. lies ersten Block jeden Laufs L_i in Puffer Input Block i
 2. wiederhole bis alle Input Blöcke im Puffer leer sind:
 - a. wähle erstes Tupel in Sortierordnung aus allen nicht-leeren Input Blöcken
 - b. schreibe Tupel auf Output Block
 - falls der Block voll ist, schreibe ihn auf die Platte
 - c. lösche Tupel vom Input Block
 - falls Block i leer ist, lies nächsten Block des Laufs L_i

Externes Merge-Sort/3

- Falls $N \geq M$, werden **mehrere Misch-Schritte** (Schritt 2) benötigt.
- Pro Durchlauf...
 - werden jeweils $M - 1$ Läufe gemischt
 - wird die Anzahl der Läufe um Faktor $M - 1$ reduziert
 - werden die Läufe um den Faktor $M - 1$ größer
- Durchläufe werden **wiederholt** bis nur mehr ein Lauf übrig ist.
- Beispiel: Puffergröße $M = 11$, Anzahl Blocks $b = 1100$
 - $N = \lceil b/M \rceil = 100$ Läufe à 11 Blocks werden erzeugt
 - nach erstem Durchlauf: 10 Läufe à 110 Blocks
 - nach zweitem Durchlauf: 1 Lauf à 1100 Blocks

Externes Merge-Sort/4

- Beispiel: $M = 3$, 1 Block = 1 Tupel



Externes Merge-Sort/5

- **Kostenanalyse:**
 - b : Anzahl der Blocks in Relation R
 - anfängliche Anzahl der Läufe: b/M
 - gesamte Anzahl der Misch-Durchläufe: $\lceil \log_{M-1}(b/M) \rceil$
 - die Anzahl der Läufe sinkt um den Faktor $M - 1$ pro Misch-Durchlauf
 - Plattenzugriffe für Erzeugen der Läufe und für jeden Durchlauf: $2 * b$
 - Ausnahme: letzter Durchlauf hat keine Schreibkosten

- **Kosten für externes Merge-Sort:** Anzahl der gelesenen oder geschriebenen Blöcke

$$\text{Kosten} = b(2\lceil \log_{M-1}(b/M) \rceil + 1)$$

- **Beispiel:** Kostenanalyse für voriges Beispiel:
 - $M = 3, b = 12$
 - $12 * (2 * \lceil \log_2(12/3) \rceil + 1) = 60$ Schreib-/Lese-/Operationen

Inhalt

- 1 Einführung
- 2 Anfragekosten abschätzen
- 3 Sortieren
- 4 **Selektion**
- 5 Join

Auswertung der Selektion/1

- Der **Selektionsoperator:**
 - **select * from R where θ**
 - $\sigma_{\theta}(R)$

berechnet die Tupel von R welche das Selektionsprädikat (=Selektionsbedingung) θ erfüllen.
- **Selektionsprädikat θ** ist aus folgenden Elementen aufgebaut:
 - Attributnamen der Argumentrelation R oder Konstanten als Operanden
 - arithmetische Vergleichsoperatoren ($=, <, >$)
 - logische Operatoren: \wedge (**and**), \vee (**or**), \neg (**not**)
- **Strategie zur Auswertung** der Selektion hängt ab
 - von der Art des Selektionsprädikats
 - von den verfügbaren Indexstrukturen

Auswertung der Selektion/2

Grundstrategien für die Auswertung der Selektion:

- **Sequentielles Lesen der Datei (file scan):**
 - Klasse von Algorithmen welche eine Datei Tupel für Tupel lesen um jene Tupel zu finden, welche die Selektionsbedingung erfüllen
 - grundlegendste Art der Selektion
- **Index Suche (index scan):**
 - Klasse von Algorithmen welche einen Index benutzen um eine Vorauswahl von Tupeln zu treffen
 - Beispiel: B^+ -Baum Index auf A und Gleichheitsbedingung: $\sigma_{A=5}(R)$

Auswertung der Selektion/3

Arten von Prädikaten:

- Gleichheitsanfrage: $\sigma_{A=V}(R)$
- Bereichsanfrage: $\sigma_{A<V}(R)$ oder $\sigma_{A>V}(R)$
- Konjunktive Selektion: $\sigma_{\theta_1 \wedge \theta_2 \dots \wedge \theta_n}(R)$
- Disjunktive Selektion: $\sigma_{\theta_1 \vee \theta_2 \dots \vee \theta_n}(R)$

Auswertung der Selektion/4

A1 Lineare Suche: Lies jeden einzelnen Block der Datei und überprüfe jeden Datensatz ob er die Selektionsbedingung erfüllt.

- **Ziemlich teuer, aber immer anwendbar**, unabhängig von:
 - (nicht) vorhandenen Indexstrukturen
 - Sortierung der Daten
 - Art der Selektionsbedingung
- **Hintereinanderliegende Blöcke lesen** wurde von den Plattenherstellern optimiert und ist schnell hinsichtlich Spurwechsel und Latenz (pre-fetching)
- **Kostenabschätzung** (b = Anzahl der Blöcke in der Datei):
 - Worst case: $Cost = b$
 - Selektion auf Kandidatenschlüssel: *Mittlere Kosten* = $b/2$ (Suche beenden, sobald erster Datensatz gefunden wurde)

Auswertung der Selektion/5

A2 Binäre Suche: verwende binäre Suche auf Blöcken um Tupel zu finden, welche Bedingung erfüllen.

- **Anwendbar** falls
 - die Datensätze der Tabelle physisch sortiert sind
 - die Selektionsbedingung auf dem Sortierschlüssel formuliert ist
- **Kostenabschätzung** für $\sigma_{A=C}(R)$:
 - $\lfloor \log_2(b) \rfloor + 1$: Kosten zum Auffinden des ersten Tupels
 - plus Anzahl der weiteren Blöcke mit Datensätzen, welche Bedingung erfüllen (diese liegen alle nebeneinander in der Datei)

Auswertung der Selektion/6

Annahme: Index ist B^+ -Baum mit H Ebenen¹

A3 Primärindex + Gleichheitsbedingung auf Suchschlüssel

- gibt **einen einzigen Datensatz** zurück
- **Kosten** = $H + 1$ (Knoten im B^+ -Baum + 1 Datenblock)

A3 Clustered Index + Gleichheitsbedingung auf Suchschlüssel

- gibt **mehrere Datensätze** zurück
- alle Ergebnisdatensätze **liegen hintereinander** in der Datei
- **Kosten** = $H + \#$ Blöcke mit *Ergebnisdatensätzen*

¹ $H \leq \lceil \log_{\lfloor m/2 \rfloor}(K) \rceil + 1$ für B^+ -Baum mit K Suchschlüsseln, Knotengrad m

Auswertung der Selektion/7

A5 Sekundärindex + Gleichheitsbedingung auf Suchschlüssel

- **Suchschlüssel ist Kandidatenschlüssel**
 - gibt einen einzigen Datensatz zurück
 - $Kosten = H + 1$
- **Suchschlüssel ist nicht Kandidatenschlüssel²**
 - mehrere Datensätze werden zurückgeliefert
 - $Kosten = (H - 1) + \# \text{Blattknoten mit Suchschlüssel} + \# \text{Ergebnisdatsätze}$
 - kann sehr teuer sein, da jeder Ergebnisdatsatz möglicherweise auf einem anderen Block liegt
 - sequentielles Lesen der gesamten Datei möglicherweise billiger

²Annahme: TIDs werden an Suchschlüssel angehängt, um diese im B⁺-Baum eindeutig zu machen; die Erweiterung um TIDs ist für Benutzer nicht sichtbar.

Auswertung der Selektion/8

A6 Primärindex auf A + Bereichsanfrage

- $\sigma_{A>V}(R)$: verwende Index um ersten Datensatz $> V$ zu finden, dann sequentielles Lesen
- $\sigma_{A<V}(R)$: lies sequentiell bis erstes Tupel $\geq V$ gefunden; Index wird nicht verwendet

A7 Sekundärindex auf A + Bereichsanfrage

- $\sigma_{A>V}(R)$: finde ersten Datensatz $> V$ mit Index; Index sequentiell lesen um alle Pointer zu den entsprechenden Datensätzen zu finden; Pointer verfolgen und Datensätze holen
- $\sigma_{A<V}(R)$: Blätter des Index sequentiell lesen und Pointer verfolgen bis Suchschlüssel $\geq V$
- Pointer verfolgen braucht im schlimmsten Fall eine Lese-/Schreib-Operation pro Datensatz; sequentielles Lesen der gesamten Datei möglicherweise schneller

Auswertung der Selektion/9

- **Pointer verfolgen in Sekundärindex:**
 - jeder Datensatz liegt möglicherweise auf einem anderen Block
 - Pointer sind nicht nach Block-Nummern sortiert
 - das führt zu Random-Zugriffen quer durch die Datei
 - derselbe Block wird möglicherweise sogar öfters gelesen
 - falls Anzahl der Ergebnisdatsätze $\geq b$, dann wird im Worst Case jeder Block der Relation gelesen
- **Bitmap Index Scan:** hilft bei großer Anzahl von Pointern
 - Block i wird durch i -tes Bit in Bit Array der Länge b repräsentiert
 - statt Pointer im Index zu verfolgen, wird nur das Bit des entsprechenden Blocks gesetzt
 - dann werden alle Blöcke gelesen, deren Bit gesetzt ist
 - ermöglicht teilweise sequentielles Lesen
 - gut geeignet, falls Suchschlüssel kein Kandidatenschlüssel ist

Integrierte Übung 3.1

Was ist die beste Auswertungsstrategie für folgende Selektion, wenn es einen B⁺-Baum Sekundärindex auf (*BrName*, *BrCity*) auf der Relation *Branch*(*BrName*, *BrCity*, *Assets*) gibt?

$$\sigma_{BrCity < 'Brighton' \wedge Assets < 5000 \wedge BrName = 'Downtown'}(Branch)$$

Inhalt

- 1 Einführung
- 2 Anfragekosten abschätzen
- 3 Sortieren
- 4 Selektion
- 5 Join

Join Operator/1

- **Theta-Join:** $r \bowtie_{\theta} s$
 - für jedes Paar von Tupeln $t_r \in r$, $t_s \in s$ wird Join-Prädikat θ überprüft
 - falls Prädikat erfüllt, ist $t_r \circ t_s$ im Join-Ergebnis
 - Beispiel: Relationen $r(a, b, c)$, $s(d, e, f)$
Join-Prädikat: $(a < d) \wedge (b = d)$
Schema des Join-Ergebnisses: (a, b, c, d, e, f)
- **Equi-Join:** Prädikat enthält “=” als einzigen Operator
- **Natürlicher Join:** $r \bowtie s$
 - Equi-Join, bei dem alle Attribute gleichgesetzt werden die gleich heißen
 - im Ergebnis kommt jedes Attribut nur einmal vor
 - Beispiel: Relationen $r(a, b, c)$, $s(c, d, e)$
Natürlicher Join $r \bowtie s$ entspricht θ -Equi-Join $\pi_{a,b,c,d,e}(r \bowtie_{r.c=s.c} s)$
Schema des Ergebnisses: (a, b, c, d, e)

Join Operator/2

- Join ist **kommutativ** (bis auf Ordnung der Attribute):

$$r \bowtie s = \pi(s \bowtie r)$$

- Ordnung der Attribute wird durch (logisches) Vertauschen der Spalten (Projektion π) wiederhergestellt und ist praktisch kostenlos

- Join ist **assoziativ**:

$$(r \bowtie s) \bowtie t = r \bowtie (s \bowtie t)$$

- **Effizienz der Auswertung:**

- vertauschen der Join-Reihenfolge ändert zwar das Join-Ergebnis nicht
- die Effizienz kann jedoch massiv beeinflusst werden!

- **Benennung der Relationen:** $r \bowtie s$

- r die **äußere Relation**
- s die **innere Relation**

Join Selektivität

- **Kardinalität:** absolute Größe des Join Ergebnisses $r \bowtie_{\theta} s$

$$|r \bowtie_{\theta} s|$$

- **Selektivität:** relative Größe des Join Ergebnisses $r \bowtie_{\theta} s$

$$sel_{\theta} = \frac{|r \bowtie_{\theta} s|}{|r \times s|}$$

- **schwache Selektivität:** Werte nahe bei 1 (viele Tupel im Ergebnis)
- **starke Selektivität:** Werte nahe bei 0 (wenig Tupel im Ergebnis)

Integrierte Übung 3.2

Gegeben Relationen $R1(\underline{A}, B, C)$, $R2(\underline{C}, D, E)$, $R3(\underline{E}, F)$, Schlüssel unterstrichen, mit Kardinalitäten $|R1| = 1000$, $|R2| = 1500$, $|R3| = 750$.

- Schätzen Sie die Kardinalität des Joins $R1 \bowtie R2 \bowtie R3$ ab (die Relationen enthalten keine Nullwerte).
- Geben Sie eine Join-Reihenfolge an, welche möglichst kleine Joins erfordert.
- Wie könnte der Join effizient berechnet werden?

Join Operator/3

- Es gibt **verschiedene Algorithmen** um einen Join auszuwerten:
 - Nested Loop Join
 - Block Nested Loop Join
 - Indexed Nested Loop Join
 - Merge Join
 - Hash Join
- Auswahl aufgrund einer **Kostenschätzung**.
- Wir verwenden folgende **Relationen in den Beispielen**:
 - Anleger = (AName, Stadt, Strasse)
 - Anzahl der Datensätze: $n_a = 10'000$
 - Anzahl der Blöcke: $b_a = 400$
 - Konten = (AName, KontoNummer, Kontostand)
 - Anzahl der Datensätze: $n_k = 5'000$
 - Anzahl der Blöcke: $b_k = 100$

Nested Loop Join/1

- **Nested Loop Join Algorithms**: berechne Theta-Join $r \bowtie_{\theta} s$

```

for each tuple  $t_r$  in  $r$  do
  for each tuple  $t_s$  in  $s$  do
    if  $(t_r, t_s)$  erfüllt Join-Bedingung  $\theta$  then
      gib  $t_r \circ t_s$  aus
    end
  end
end

```
- **Immer anwendbar**:
 - für jede Art von Join-Bedingung θ anwendbar
 - kein Index erforderlich
- **Teuer** da jedes Tupel des Kreuzproduktes ausgewertet wird

Nested Loop Join/2

- **Ordnung der Join Argumente relevant**:
 - r wird 1x gelesen, s wird bis zu n_r mal gelesen
- **Worst case**: $M = 2$, nur 1 Block von jeder Relation passt in Puffer
 $Kosten = b_r + n_r * b_s$
- **Best case**: $M > b_s$, innere Relation passt vollständig in Puffer (+1 Block der äußeren Relation)
 $Kosten = b_r + b_s$
- **Beispiel**:
 - Konten \bowtie Anleger: $M = 2$
 $b_k + n_k * b_a = 100 + 5'000 * 400 = 2'000'100$ Block Zugriffe
 - Anleger \bowtie Konten: $M = 2$
 $b_a + n_a * b_k = 400 + 10'000 * 100 = 1'000'400$ Block Zugriffe
 - Kleinere Relation (*Konten*) passt in Puffer: $M > b_k$
 $b_a + b_k = 400 + 100 = 500$ Block Zugriffe
- Einfacher Nested Loop Algorithms **wird nicht verwendet** da er nicht Block-basiert arbeitet.

Block Nested Loop Join/1

- **Block Nested Loop Join** vergleicht jeden Block von r mit jedem Block von s .
- Algorithmus für $r \bowtie_{\theta} s$

```

for each Block  $B_r$  of  $r$  do
  for each Block  $B_s$  of  $s$  do
    for each Tuple  $t_r$  in  $B_r$  do
      for each Tuple  $t_s$  in  $B_s$  do
        if  $(t_r, t_s)$  erfüllt Join-Bedingung  $\theta$  then
          gib  $t_r \circ t_s$  aus

```

Block Nested Loop Join/2

- **Worst case:** $M = 2$, $Kosten = b_r + b_r * b_s$
 - Jeder Block der inneren Relation s wird für jeden Block der äußeren Relation einmal gelesen (statt für jedes Tupel der äußeren Relation)
- **Best case:** $M > b_s$, $Kosten = b_r + b_s$
- **Beispiel:**
 - $Konten \bowtie$ $Anleger$: $M = 2$
 $b_k + b_k * b_a = 100 + 100 * 400 = 40'100$ Block Zugriffe
 - $Anleger \bowtie$ $Konten$: $M = 2$
 $b_a + b_a * b_k = 400 + 400 * 100 = 40'400$ Block Zugriffe
 - Kleinere Relation ($Konten$) passt in Puffer: $M > b_k$
 $b_a + b_k = 400 + 100 = 500$ Block Zugriffe

Block Nested Loop Join/3

- **Zick-Zack Modus:** $R \bowtie_{\theta} S$
 - reserviere $M - k$ Blöcke für R und k Blöcke für S
 - innere Relation wird abwechselnd vorwärts und rückwärts durchlaufen
 - dadurch sind die letzten k Seiten schon im Puffer (LRU Puffer Strategie) und müssen nicht erneut gelesen werden
- **Kosten:** $k \leq b_s, 0 < k < M$

$$b_r + k + \lceil b_r / (M - k) \rceil (b_s - k)$$
 - r muss einmal vollständig gelesen werden
 - innere Schleife wird $\lceil b_r / (M - k) \rceil$ mal durchlaufen
 - erster Durchlauf erfordert b_s Block Zugriffe
 - jeder weitere Durchlauf erfordert $b_s - k$ Block Zugriffe
- **Optimale Ausnutzung des Puffers:**
 - $b_r \leq b_s$: kleinere Relation außen (Heuristik)
 - $k = 1$: $M - 1$ Blöcke für äußere Relation, 1 Block für innere

Integrierte Übung 3.3

Berechne die Anzahl der Block Zugriffe für folgende Join Alternativen, jeweils mit Block Nested Loop Join, Puffergröße $M = 20$.

Konto: $n_k = 5'000$, $b_k = 100$. Anleger: $n_a = 10'000$, $b_a = 400$

- $Konto \bowtie$ $Anleger$, $k = 19$
- $Konto \bowtie$ $Anleger$, $k = 10$
- $Konto \bowtie$ $Anleger$, $k = 1$
- $Anleger \bowtie$ $Konto$, $k = 1$

Indexed Nested Loop Join/1

- **Index Suche** kann Scannen der inneren Relation ersetzen
 - auf innerer Relation muss Index verfügbar sein
 - Index muss für Join-Prädikat geeignet sein (z.B. Equi-Join)
- **Algorithmus:** Für jedes Tupel t_r der äußeren Relation r verwende den Index um die Tupel der inneren Relation zu finden, welche die Bedingung θ erfüllen.
- **Worst case:** für jedes Tupel der äußeren Relation wird eine Index Suche auf die innere Relation gemacht.
 $Kosten = b_r + n_r * c$
 - c sind die Kosten, den Index zu durchlaufen und alle passenden Datensätze aus der Relation s zu lesen
 - c kann durch die Kosten einer einzelnen Selektion mithilfe des Index abgeschätzt werden
- **Index auf beiden Relationen:** kleinere Relation außen

Indexed Nested Loop Join/2

- **Beispiel:** Berechne $Konten \bowtie Anleger$ (Konten als äußere Relation), B^+ -Baum mit $m = 20$ auf Relation $Anleger$.
- **Lösung:**
 - $Anleger$ hat $n_a = 10'000$ Datensätze.
 - Kosten für 1 Datensatz von Relation $Anleger$ mit Index lesen:

$$c = \lceil \log_{\lceil m/2 \rceil}(L) \rceil + 2 = \lceil \log_{\lceil m/2 \rceil}(\lceil \frac{n_a}{\lceil m-1 \rceil} \rceil) \rceil + 2 = 5$$
 - B^+ -Baum durchlaufen: maximale Pfadlänge + 1
 - 1 Zugriff auf Datensatz (Schlüssel)
 - $Konten$ hat $n_k = 5'000$ Datensätze und $b_k = 100$ Blöcke.
 - Indexed Nested Loop Join:
 $Kosten = b_k + n_k * c = 100 + 5'000 * 5 = 25'100$ Blockzugriffe

Merge Join/1

- **Merge Join:** Verwende zwei Pointer pr und ps die zu Beginn auf den ersten Datensatz der sortierten Relationen r bzw. s zeigen und bewege die Zeiger synchron, ähnlich wie beim Mischen, nach unten.
- **Algorithmus:** $r \bowtie s$ (Annahme: keine Duplikate in Join-Attributen)
 1. sortiere Relationen nach Join-Attributen (falls nicht schon richtig sortiert)
 2. starte mit Pointern bei jeweils 1. Tupel
 3. aktuelles Tupel-Paar ausgeben falls es Join-Bedingung erfüllen
 4. bewege den Pointer der Relation mit dem kleineren Wert; falls die Werte gleich sind, bewege den Pointer der äußeren Relation
- **Duplikate** in den Join-Attributen: bei gleichen Werten muss jede Kopie der äußeren mit jeder Kopie der inneren Relation gepaart werden

	a_1	a_2
pr →	a	3
	b	1
	d	8
	d	13
	f	7
	m	5
	q	6

r

	a_1	a_3
ps →	a	A
	b	G
	c	L
	d	N
	m	B

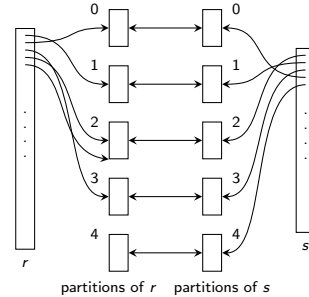
s

Merge Join/2

- Anwendbar **nur für Equi- und Natürliche Joins**
- **Kosten:** Falls alle Tupel zu einem bestimmten Join-Wert im Puffer Platz haben:
 - r und s 1x sequentiell lesen
 - $Kosten = b_r + b_s$ (+ Sortierkosten, falls Relationen noch nicht sortiert)
- Andernfalls muss ein **Block Nested Loop Join** zwischen den Tupeln mit identischen Werten in den Join-Attributen gemacht werden.
- **Sort-Merge Join:** Falls Relationen noch nicht sortiert sind, muss zuerst sortiert werden.

Hash Join/1

- Nur für **Equi- und Natürliche Joins**.
- Partitioniere Tupel von r und s mit derselben **Hash Funktion h** , welche die Join-Attribute (*JoinAttrs*) auf die Menge $\{0, 1, \dots, n\}$ abbildet.
- Alle Tupel einer Relation mit demselben Hash-Wert bilden eine **Partition** (=Bucket):
 - Partition r_i enthält alle Tupel $t_r \in r$ mit $h(t_r[\text{JoinAttrs}]) = i$
 - Partition s_i enthält alle Tupel $t_s \in s$ mit $h(t_s[\text{JoinAttrs}]) = i$
- Partitionsweise joinen**: Tupel in r_i brauchen nur mit Tupel in s_i verglichen werden
 - ein r -Tupel und ein s -Tupel welche die Join-Kondition erfüllen, haben denselben Hash-Wert i und werden in die Partitionen r_i bzw. s_i gelegt



Hash Join/2

- Algorithmus** für Hash Join $r \bowtie s$.
 - Partitioniere r und s mit derselben Hash Funktion h ; jede Partition wird zusammenhängend auf die Platte geschrieben
 - Für jedes Paar (r_i, s_i) von Partitionen:
 - build**: lade s_i in den Hauptspeicher und baue einen Hauptspeicher-Hash-Index mit neuer Hash-Funktion $h' \neq h$.
 - probe**: für jedes Tupel $t_r \in r_i$ suche zugehörige Join-Tupel $t_s \in s_i$ mit Hauptspeicher-Hash-Index.
- Relation s wird **Build Input** genannt; r wird **Probe Input** genannt.
- Kleinere Relation** (in Anzahl der Blöcke) wird als Build Input verwendet, damit weniger Partitionen benötigt werden.
 - Hash-Index für jede Partition des Build Input muss in Hauptspeicher passen ($M - 1$ Blöcke für Puffergröße M)
 - von Probe Input brauchen wir jeweils nur 1 Block im Speicher

Hash Join/3

- Kosten** für Hash Join:
 - Partitionieren der beiden Relationen: $2 * (b_r + b_s)$
 - jeweils gesamte Relation einlesen und zurück auf Platte schreiben
 - Build- und Probe-Phase lesen jede Relation genau einmal: $b_r + b_s$
 - $Kosten = 3 * (b_r + b_s)$
 - Kosten von nur **teilweise beschriebenen Partitionen** werden nicht berücksichtigt.

Hash Join/4

Beispiel: $Konto \bowtie Anleger$ soll als Hash Join berechnet werden. Puffergröße $M = 20$ Blöcke, $b_k = 100$, $b_a = 400$.

- Welche Relation wird als Build Input verwendet?
Konto, da kleiner ($b_k < b_a$)
- Wieviele Partitionen müssen gebildet werden?
 $\lceil \frac{b_k}{M-1} \rceil = 6$ Partitionen, damit Partitionen von Build Input in Puffer ($M - 1 = 19$) passen. Partitionen von Probe Input müssen nicht in Puffer passen: es wird nur je ein Block eingelesen.
- Wie groß sind die Partitionen?
Build Input: $\lceil 100/6 \rceil = 17$, Probe Input: $\lceil 400/6 \rceil = 67$
- Kosten für Join?
 $3(b_k + b_a) = 1'500$ laut Formel. Da wir aber nur ganze Blöcke schreiben können, sind die realen Kosten etwas höher:
 $b_k + b_a + 2 * (6 * 17 + 6 * 67) = 1'508$

Rekursives Partitionieren

- Eine Relation kann **höchstens in $M - 1$ Partitionen** zerlegt werden:
 - 1 Input-Block
 - $M - 1$ Output Blocks (1 Block pro Partition)
- Partitionen der Build-Relation (b Blöcke) **müssen in Speicher passen**
 - Build-Partition darf maximal $M - 1$ Blöcke haben
 \Rightarrow Anzahl der Partitionen mindestens $\lceil \frac{b}{M-1} \rceil$
- Build-Relation könnte **zu groß** für maximale Partitionen-Anzahl sein:
 - falls $\lceil \frac{b}{M-1} \rceil > M - 1$ können nicht genug Partitionen erzeugt werden
- **Rekursives Partitionieren:**
 - erzeuge $M - 1$ Partitionen (r_i, s_i) , $1 \leq i < M$
 - partitioniere jedes Paar (r_i, s_i) rekursiv (mit einer neuen Hash-Funktion), bis Build-Partition in Hauptspeicher passt
 - (r_i, s_i) wird also behandelt wie zwei Relationen

Overflows/1

- **Overflow:** Build Partition passt nicht in den Hauptspeicher
 - kann auch vorkommen, wenn es sich von der Größe der Build-Relation her ausgehen müsste (d.h. $\lceil \frac{b}{M-1} \rceil \leq M - 1$)
- Overflows entstehen durch **verschieden große Partitionen:**
 - einige Werte kommen viel häufiger vor oder
 - die Hashfunktion ist nicht uniform und random
- **Fudge Factor:**
 - etwas mehr als $\lceil \frac{b}{M-1} \rceil$ Partitionen (z.B. 20% mehr) werden angelegt
 - dadurch werden kleine Unterschiede in der Partitionsgröße abgedeckt
 - hilft nur bis zu einem gewissen Grad
- **Lösungsansätze**
 - Overflow Resolution
 - Overflow Avoidance

Overflows/2

- **Overflow Resolution:** während der Build-Phase
 - falls Build-Partition s_i zu groß: partitioniere Probe- und Build-Partition (r_i, s_i) erneut bis Build-Partition in Speicher passt
 - für erneutes Partitionieren muss neue Hashfunktion verwendet werden
 - selbe Technik wie rekursives Partitionieren (es wird jedoch aufgrund unterschiedlicher Partitionsgrößen neu partitioniert, nicht wegen der Größe der Build-Relation)
- **Overflow Avoidance:** während des Partitionierens
 - viele kleine Partitionen werden erzeugt
 - während der Build-Phase werden so viele Partitionen wie möglich in den Hauptspeicher geladen
 - die entsprechenden Partitionen in der anderen Relation werden für das Probing verwendet
- **Wenn alle Stricke reißen...**
 - wenn einzelne Werte sehr häufig vorkommen versagen beide Ansätze
 - Lösung: Block-Nested Loop Join zwischen Probe- und Build-Partition

Zusammenfassung

- **Nested Loop Joins:**
 - Naive NL: ignoriert Blöcke
 - Block NL: berücksichtigt Blöcke
 - Index NL: erfordert Index auf innere Relation
- **Equi-Join Algorithmen:**
 - Merge-Join: erfordert sortierte Relationen
 - Hash-Join: keine Voraussetzung