Datenbanksysteme

Wintersemester 2016/17

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Implementierung/Realisierung von Operatoren

Bislang nur Bedeutung (Ergebnis) eines Operators

- Relationale Algebra: σ, π, $\bowtie$, γ, δ, ...
- Wir haben betrachtet wie Zwischenergebnisgrößen abgeschätzt werden können
- und wie logische Operatorbäume/Pläne optimiert werden können (Vgl. Äquivalenzregeln, Selektivitätsschätzung)

Nun: Realisierung von Operatoren

- Welche Alternativen in der Implementierung eines Operators gibt es?
- Welche Kosten fallen an?

Physische Optimierung

- Aus all diesen Möglichkeiten kann dann in der (physischen) Anfrageoptimierung nach der günstigsten Möglichkeit geschaut werden.
- Ebenso, gibt es Indexe, die benutzt werden können?
- Aber Achtung, wie zuvor gesehen, manchmal ist die Verwendung von Indexen teurer als Alternativen, die Relationen komplett betrachten.
- Resultat ist ein physischer Operatorbaum, in dem jedem Operator die Implementierung zugeordnet ist. Also für $\bowtie_\theta$ z.B. $\bowtie_\theta^{NL}$ für Nested-Loop-Join mit ggf. weiteren Details.

Join Implementierungen

- Nested Loop Join
- Index-based Nested Loop Join
- Block-based (Index-Based) Nested Loop Join
- Sort-merge Join
- Hash Join

Größe der Eingabe-Relationen sowie Selektivität beeinflusst Kosten eines Joins

- **Selektivität:** $sel = \frac{\#\text{Ergebnistupel}}{\#\text{Kandidaten}}$
- Für Join, #Kandidaten ist die Größe des kartesischen Produkts

Nested Loop Join

"Brute Force" algorithm

for each $r \in R$
  for each $s \in S$
    if $s.B = r.A$
      then $Res := Res \cup (r \circ s)$

- Idee: bilde Kreuzprodukt
- Suche Tupel heraus, die das Joinprädikat erfüllen
- **Problem: quadratischer Aufwand**
- **Vorteil: funktioniert für jedes Prädikat**

Notation: $r \circ s$ bedeutet Konkatenation der Tupel

Nested Loop Join (NLJ)

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

result

Als Iterator Implementierung

iterator NestedLoop

open

Öffne die linke Eingabe

next

Rechte Eingabe geschlossen?

Öffne sie

Fordere rechts solange Tupel an, bis Bedingung p erfüllt ist

Sollte zwischendurch rechte Eingabe erschöpft sein

Schließe rechte Eingabe

Fordere nächtes Tupel der linken Eingabe an

Starte **next** neu

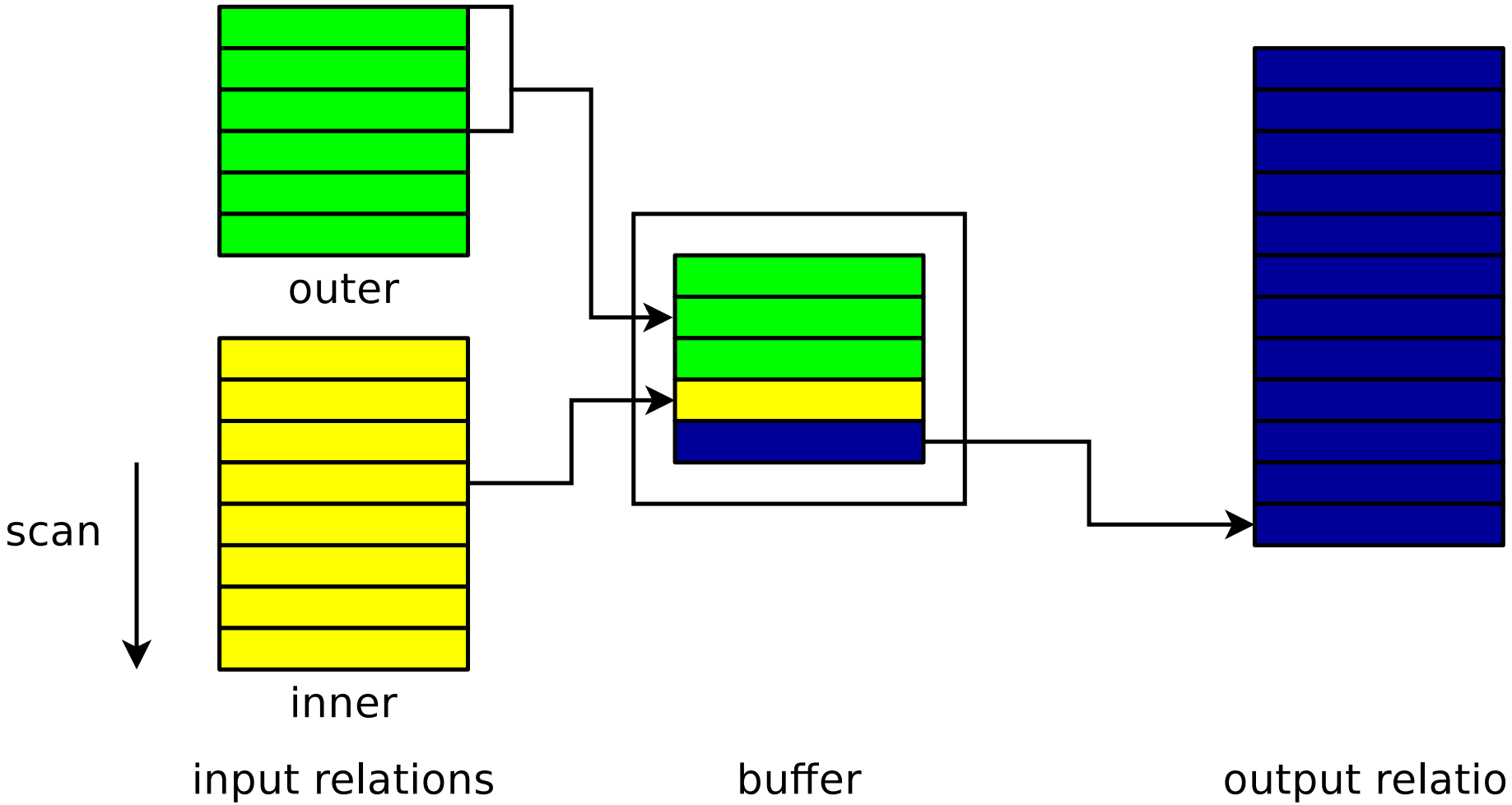
Return Verbund von aktuellem linken und aktuellem rechten Tupel

close

Schließe beide Eingabequellen

Block Nested Loop Join

Idee: Lese eine Relation (outer) ganz (falls möglich) oder einige Blöcke davon, dann gehe blockweise (mehrfach) über zweite Relation (inner).



Block Nested Loop Join

Nicht alle Blöcke passen in Hauptspeicher

repeat
  lese $n_B - 2$ Blöcke aus der äußeren Relation
  repeat
    lese 1 Block der inneren Relation
    vergleiche Tupel
  until Komplette innere Rel. gelesen
until Komplette äußere Rel. gelesen

Parameter:

- N_{inner}, N_{outer}: Anzahl Blöcke
- n_B: Größe des Puffers im Hauptspeicher

Kostenschätzung (als Anzahl Zugriffe auf Blöcke)

$$N_{outer} + (\lceil N_{outer}/(n_B - 2)\rceil) * N_{inner}$$

In-memory Matching z.B. durch Aufbau Hashtabelle auf Teil von R und Check der S Tupel dagegen.

Block Nested Loop Join

Example ($reserved \bowtie customer$)

- Größe der Relationen in Anzahl Blöcke
  $N_{reserved} = 2000$, $N_{customer} = 10$
- Größe des Puffers im Hauptspeicher
  $N_B = 6$
- Kostenschätzung (Anzahl transferierte Blöcke)
  $N_{outer} + (\lceil N_{outer}/(n_B - 2) \rceil) * N_{inner}$

Kosten

- **Relation `reserved` als äußere Relation:**
  $2000 + \lceil (2000/4) \rceil * 10 = 7000$
- **Relation `customer` als äußere Relation:**
  $10 + \lceil (10/4) \rceil * 2000 = 6010$

Index-based Nested Loop Join

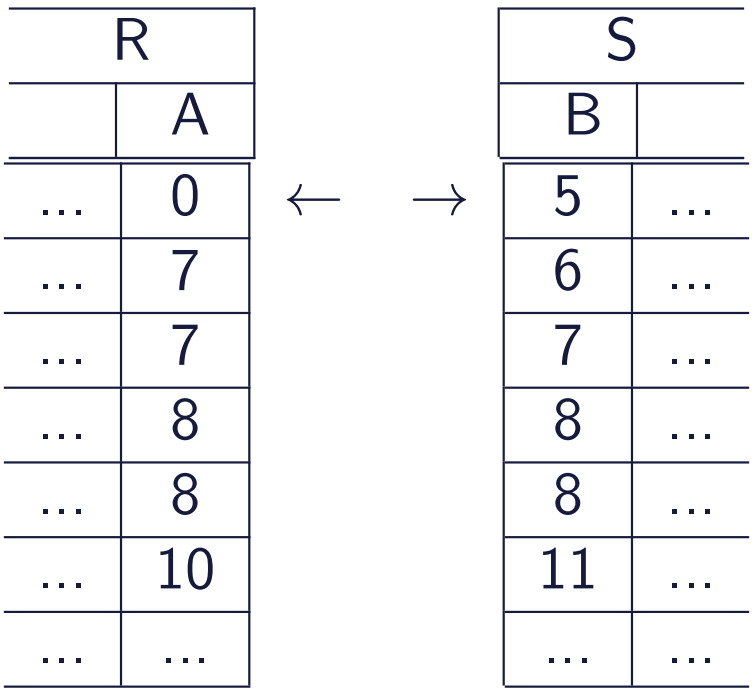
for each $r \in R$
  for each $s \in S$ **where** $[s.B = r.A]$
    then $Res := Res \cup (r \circ s)$

Gleiches Prinzip wie Nested Loop Join

- Äußere Relation
- Innere Relation
- **Suche anhand Index kann (teure) naive Suche auf innerer Relation ersetzen**
- Zugriffskosten für Index typischerweise 2–4 I/Os für B+ Baum und 1-2 I/Os für Hash-Index
- Evtl. noch ein weiterer Zugriff, um Tupel zu laden, falls Index nur Zeiger auf Tupel-IDs hat.

Merge Join

Ausnutzen von sortierten Relationen (sonst muss sortiert werden: **Sort-Merge Join**)



- Beide Eingaben sind sortiert (oder werden vorher sortiert)
- Dann genügt ein einfacher Merge über die Eingaben.
- **Achtung! Bei "Zeilen" mit gleichen Werten muss zurückgesprungen werden.**

Merge Join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

$\bowtie$

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

$=$

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

Sort-Merge-Join zwischen Relationen R und S, anhand Prädikat $R_i = S_j$

if R not sorted on attribute i, sort it
if S not sorted on attribute j, sort it

Tr = first tuple in R
Ts = first tuple in S
Gs = first tuple in S

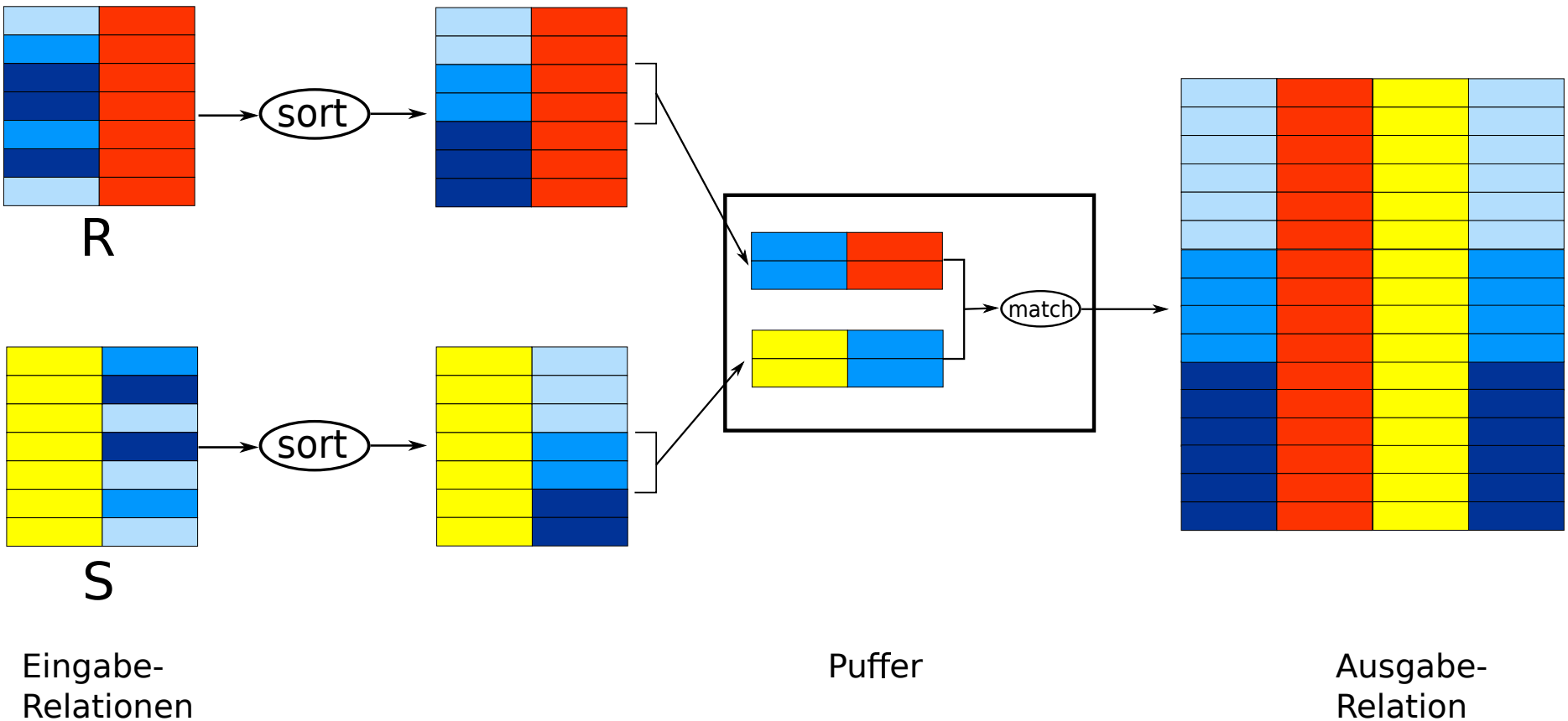
while ($Tr \neq$ eof and $Gs \neq$ eof) **do** {

    while $Tr_i < Gs_j$ **do** {Tr = next tuple in R after Tr}
    while $Tr_i > Gs_j$ **do** {Gs = next tuple in S after Gs}

    $Ts = Gs$
    while $Tr_i == Gs_j$ **do** {
        $Ts = Gs$
        while ($Ts_j == Tr_i$ **do** {
            add $Tr \circ Ts$ to result
            Ts = next tuple in S after Ts
        }
        Tr = next tuple in R after Tr
    }
    $Gs = Ts$
}

Aus dem Buch von Ramakrishnan und Gehrke.

Merge Join: Veranschaulichung mit Sortier-Phase



Merge Join – Kosten

Parameter

- b_1, b_2: Anzahl Blöcke

Kostenschätzung (Anzahl Blöcke-Transfers)

$$b_1 + b_2$$

Für den Fall, dass eine Relation nicht mehrfach gescannt werden muss.

Was passiert, wenn die Tupel in beiden Seiten (d.h. alle) den gleichen Wert für das Join Attribut haben?

Erweiterung

- Kombination mit Sortieren falls Eingabe-Relationen nicht bereits sortiert.
- Was ist wenn Daten zu groß für Hauptspeicher sind?

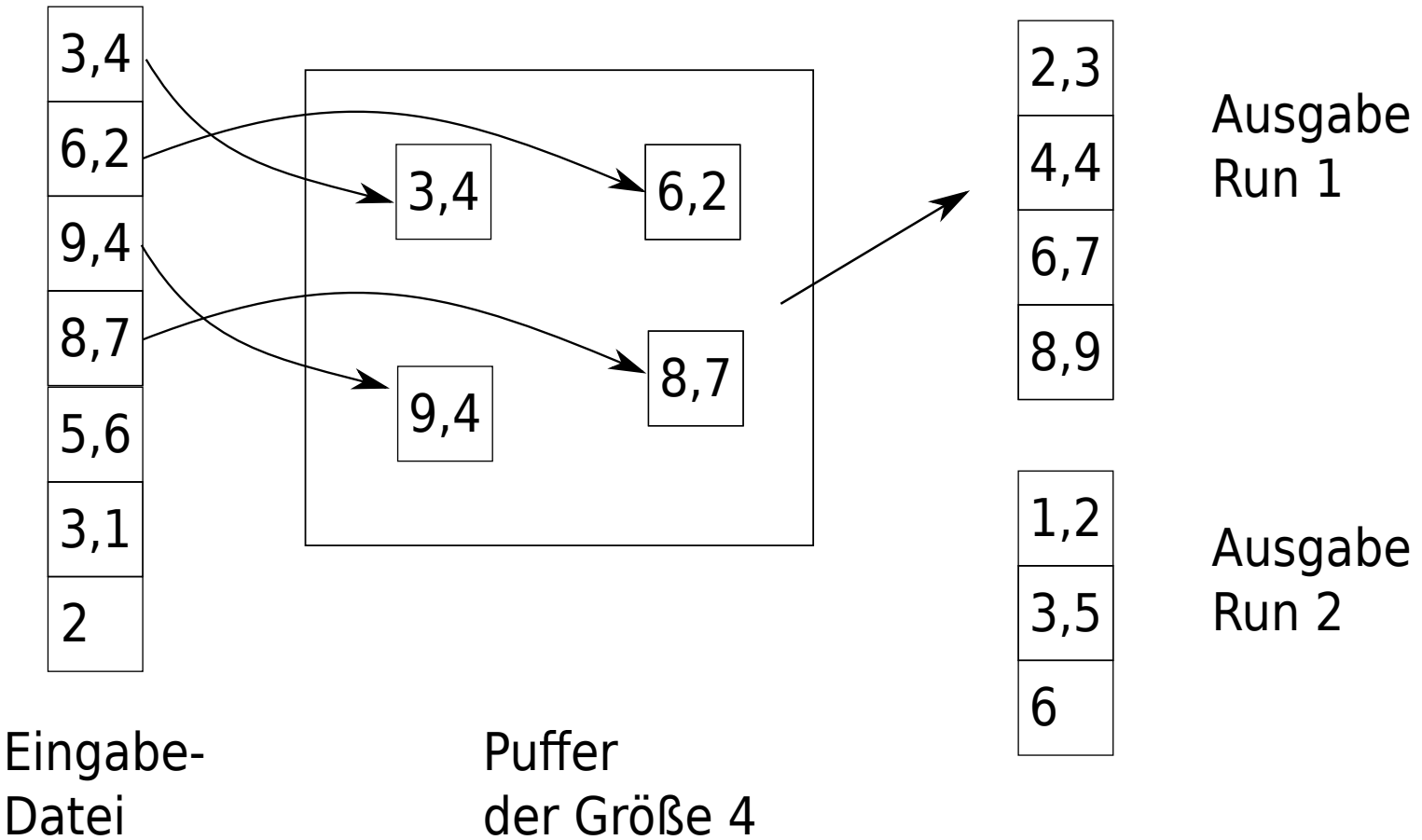
Externes Sortieren

Wenn eine Datei (Relation) nicht als Ganzes in den Hauptspeicher passt kann sie dennoch sortiert werden:

Idee

- Spalte Datei in (viele) kleinere Teile (Runs)
- Sortiere jedes Teil.
- Merge diese inkrementell.
- Anzahl dieser Schritte hängt von Größe des zur Verfügung stehenden Hauptspeicher Puffers ab

Externes Sortieren: Erzeugen von Runs Illustration



Erzeugen von Runs

Wie viele solcher Runs gibt es?

- **Anzahl von Blöcken der Datei:** N (d.h. Größe der Datei in Anzahl Blöcke)
- Größe des zum sortieren nutzbaren Hauptspeichers (Puffer) in Blöcke: n_B
- **Anzahl Runs** gegeben durch

$$n_R = \lceil \frac{N}{n_B} \rceil$$

- **Beispiel:** $n_B = 5$ Blöcke, $N = 1024$ Blöcke, dann sind 205 (initiale) Runs erzeugt worden
  - Jeder dieser Runs ist 5 Blöcke groß, bis auf den letzten, der 4 Blöcke belegt.
  - D.h. nach diesem Schritt liegen 205 sortierte Runs als temporäre Dateien auf der Festplatte.

Externes Sortieren

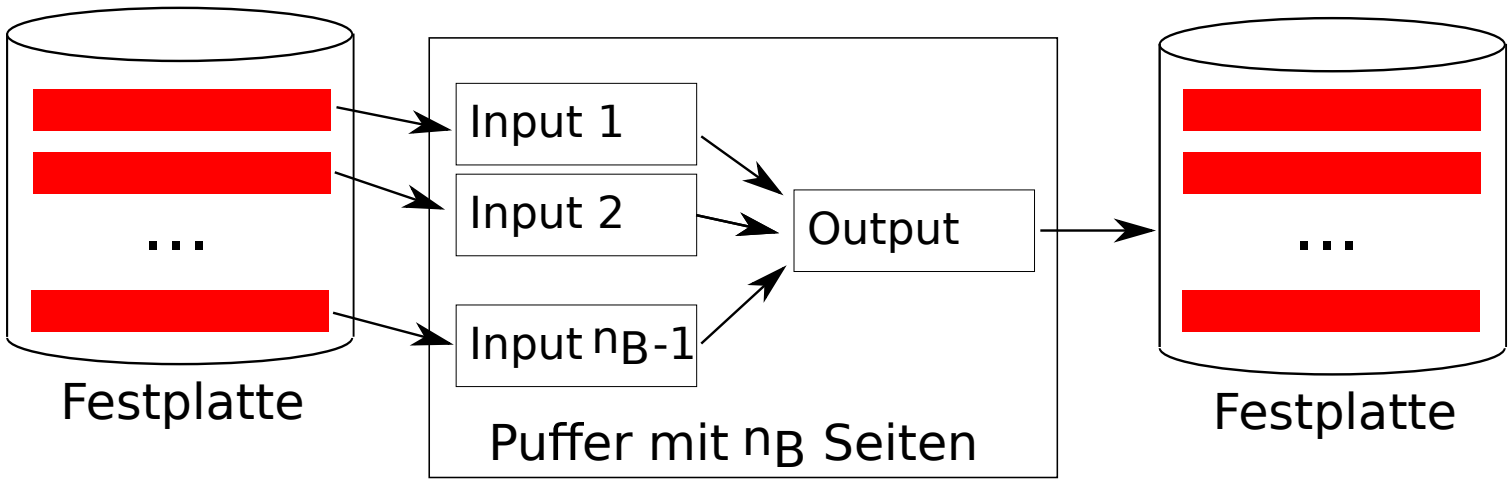
Sortier-Phase

- Lese die nächsten n_B Blöcke der Datei in den Puffer
- Sortiere die Tupel im Puffer und schreibe Ergebnis in temporäre Teil-Datei (Run)

Merge-Phase

- Anzahl Merge-Phasen ist $p = \lceil log_{n_B-1}(n_R) \rceil$
- Solange noch mehr als ein Run übrig:
  - Lese die nächsten $n_B - 1$ Runs, je ein Block nach dem anderen
  - Merge Tupel und schreibe Ergebnis (Run) auf Festplatte, ein Block nach dem anderen.
  - Hierbei werden längere Runs erzeugt

Externes Sortieren: Illustration Mischen von Runs



- $(n_B - 1)$ – way merge

Externes Sortieren: Beispiel

Angenommen wir haben einen Puffer der Größe 5 Seiten und möchten eine Datei/Relation mit 108 Seiten sortieren.

- **Durchlauf 0** erzeugt $\lceil \frac{108}{5} \rceil = 22$ Runs.
- **Durchlauf 1** macht einen 4-way merge und erzeugt so $\lceil \frac{22}{4} \rceil = 6$ sortierte Runs, die je 20 Seiten groß sind, nur der letzte dieser Runs ist 8 Seiten groß.
- **Durchlauf 2** erzeugt $\lceil \frac{6}{4} \rceil = 2$ sortierte Runs. Einen mit 80 Seiten und einen mit 28 Seiten.
- **Durchlauf 3** merged die beiden Runs aus Durchlauf 2. **Fertig.**

Beispiel

Anzahl Durchläufe (Passes), in Abhängigkeit von Größe der Datei in N Seiten (Blöcke) und Anzahl Puffer Seiten n_B.

N	n_B=3	n_B=5	n_B=9	n_B=17	n_B=129	n_B=257
100	7	4	3	2	1	1
1.000	10	5	4	3	2	2
10.000	13	7	5	4	2	2
100.000	17	9	6	5	3	3
1.000.000	20	10	7	5	3	3
10.000.000	23	12	8	6	4	3
100.000.000	26	14	9	7	4	4
1.000.000.000	30	15	10	8	5	4

Blocked I/O

- **Beobachtung: Unterschied zwischen Anzahl Seitenzugriffe und Minimierung der Zugriffskosten.**
- Bislang, so wenig Seitenzugriffe wie möglich, d.h. möglichst wenige Runs. D.h. Möglichst viele Runs gleichzeitig mischen, also $n_B - 1$, bei einem Block als Puffer.
- Aber: Lesen von mehreren Blöcken sequentiell anstelle eines einzelnen Blocks reduziert Kosten pro Block (Zur Erinnerung: Kosten von wahlfreien vs. sequentiellen Zugriffen; Latenz!)
- **Idee: Lese/Schreibe b Blöcke auf einmal.**

Blocked I/O (2)

- Die Menge b Blöcke wird Buffer-Block genannt.
- Wir brauchen einen Buffer-Block pro Eingabe-Run und einen Buffer-Block für die Ausgabe.
- D.h. $\lfloor \frac{n_B - b}{b} \rfloor$ Runs können pro Durchlauf gemischt werden.
- Beispiel: Für $n_B = 10$ können wir 9 Runs mit je einem Block als Eingabe mischen, mit einem Block für die Ausgabe.

  Oder 4 Runs mit je zwei Blöcken für Eingabe pro Run und zwei Blöcken für die Ausgabe.

- **Nachteil: Anzahl der Phasen wächst. Vorteil: Weniger Kosten pro einzelnem Blockzugriff.**
- Aber: Durch normalerweise recht großen Hauptspeicher, lassen sich auch mit Blocked I/O sehr große Dateien in nur zwei Phasen sortieren.

Blocked I/O (3)

- Erster Durchlauf erzeugt $n_R = \lceil \frac{N}{n_B} \rceil$ Runs der Länge n_B
- In jedem weiteren Durchlauf können $F = \lfloor n_B/b \rfloor - 1$ Runs gemerged werden.
- Die Anzahl benötigter Durchläufe ist somit $1 + \lceil log_F\ n_R \rceil$

Sortieren: Anwendung

Wann muss ein Datenbanksystem Daten sortieren?

- Wie oben gesehen, zur **Realisierung von Sort-Merge Joins**
- Oder zur **Eliminierung von Duplikaten** (select distinct ...) (wie?)
- Falls Benutzer **Antworten in einer bestimmen Reihenfolge** haben möchten.
- Wenn z.B. ein B+ Baum über großen bestehenden Daten angelegt werden soll: Sogenanntes **Bulk-Loading**. Wieso ist das geschickt?

Hash Join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

Angestellte

$\bowtie$

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

Telefon

Wende Hashfunktion an auf Join-Attribut(e)
$\rightarrow$ **Partitioniert Tupel in Buckets**

Hash Join

ID	name
15	Pete
21	Dave

Angestellte$_0$

$\bowtie$

number	ID
120	15
160	15
170	21

Telefon$_0$

$=$

ID	name	number
15	Pete	120
15	Pete	160
21	Dave	170

Ergebnis$_0$

ID	name
10	Jim
13	Joe

Angestellte$_1$

$\bowtie$

number	ID
110	10
150	13

Telefon$_1$

$=$

ID	name	number
10	Jim	110
13	Joe	150

Ergebnis$_1$

ID	name
14	Sue
23	Anne

Angestellte$_2$

$\bowtie$

number	ID
100	23
130	23
140	23

Telefon$_2$

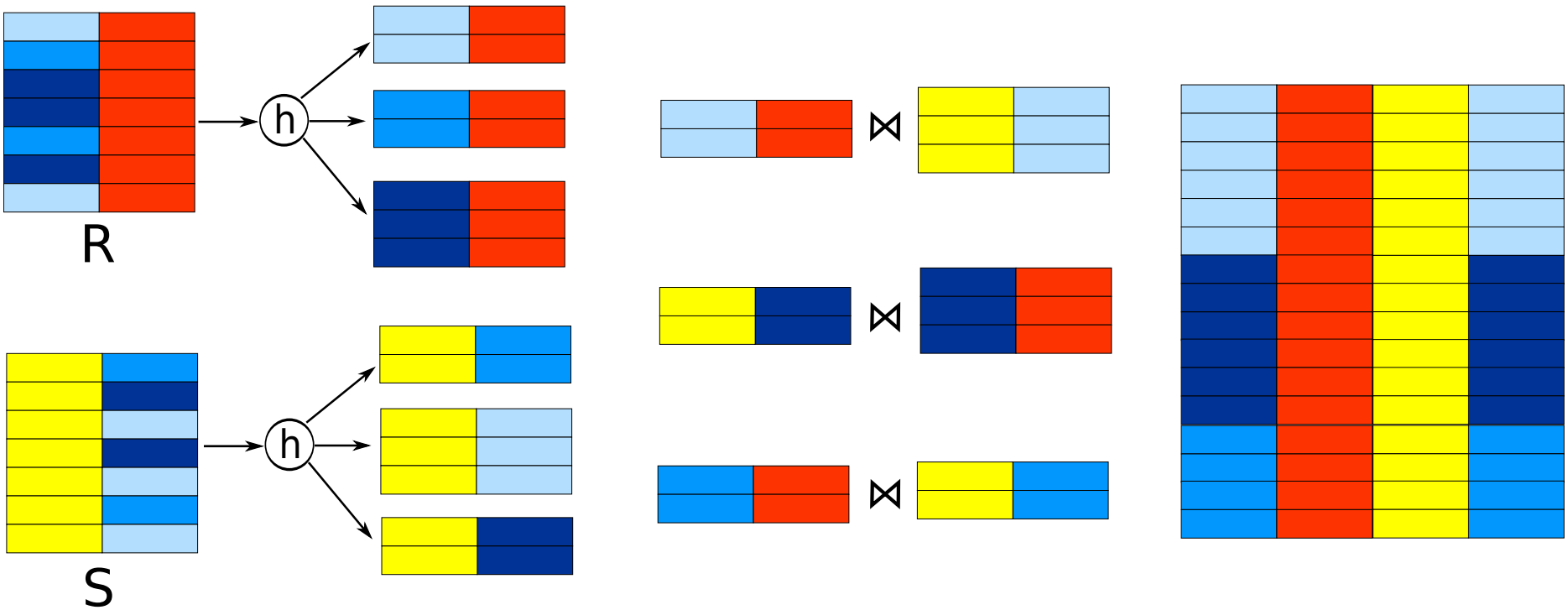
$=$

ID	name	number
23	Anne	100
23	Anne	130
23	Anne	140

Ergebnis$_2$

Hash Join: Idee

- Hashing jeder Relation basierend auf Join Attribut(en)
- Jeder Bucket muss klein genug für Hauptspeicher sein
- Tupel in korrespondierenden Buckets der beiden Relationen werden dann verbunden



Eingabe-Relationen

Ausgabe-Relation

Hash Join: Pseudocode

Aka. **Grace Hash Join** (in Literatur)

Partitioniere R in k Partitionen R_i

for each tuple $r \in R$ **do**
    lese r und füge es Puffer-Seite (Block) $h(r)$ hinzu

Partitioniere S in k Partitionen S_i

for each tuple $s \in S$ **do**
    lese s und füge es Puffer-Seite (Block) $h(s)$ hinzu

Hash Join: Pseudocode (2)

Sondierung (engl. Probing) Phase

for $l = 1, \ldots, k$ **do** $\{$

//Erzeuge im Hauptspeicher Hash-Tabelle für R_l mit Hash-Funktion h_2
for each tuple $r \in$ partition R_l **do**
lese r und füge es in Hash-Tabelle anhand $h_2(r)$ ein

//Lese S_l und teste nach passenden Tupeln aus R_l
for each tuple $s \in$ partition S_l **do** $\{$
lese s und teste Hash-Tabelle unter Verwendung von $h_2(s)$
für passende Tupel $r \in R$: $Res := Res \cup (r \circ s)$ $\}$
leere Hash-Tabelle und betrachte nächste Partition.

$\}$

Achtung: Hier wird eine andere Hashfunktion als zuvor benutzt. Warum ist dies elementar?

Kosten und Anwendbarkeit der verschiedenen Join-Strategien

Nested Loop Join

- Kann für alle Arten von Joins benutzt werden
- Kann aber sehr teuer sein

Merge Join

- Eingaben müssen bereits sortiert vorliegen
- Oder für den Join extra sortiert werden
- Kann ggf. Indexe ausnutzen

Hash Join

- Gute Hashfunktionen sind elementar
- Performance am besten, wenn kleinere Relation direkt in Hauptspeicher passt

Was ist mit Joins über mehrere Attribute? Was mit Joins, die nicht auf Gleichheit (=) testen? Welche Implementierungen sind anwendbar?

Implementierung anderer Operatoren

Selektion

- Gibt es Indexe, die ausgenutzt werden können? Clustered oder nicht clustered? Ansonsten "full-table scan".
- Wie sieht das Prädikat aus? Disjunktion, Konjunktion?
- select ... from table where (x<100 **OR** name='Joe') mit Index auf name. Index benutzen?? Brauchen sowieso full scan wegen x<100.
- was ist wenn (x<100 **AND** name='Joe') ?

Duplikate Eliminieren und Aggregation

- Verschiedene Möglichkeiten, basierend auf Sortierung oder Hashing.
- Idee: Bringe Duplikate nah zueinander: Sortiert oder in gleichen Hash-Bucket.

Projektion

- Trivial. Bis auf den Fall wo Duplikate entfernt werden sollen "select distinct", dann siehe vorherigen Punkt.

Zusammenfassung Implementierung von Operatoren

- Verschiedene Implementierungen haben in der Regel verschiedene Ausführungskosten
- Diese hängen wiederum von den Eingaben und ggf. Reihenfolge der Eingaben ab.
- Ob Implementierung überhaupt benutzt werden kann ist auch abhängig von Eingabe, bzw. ob z.B. ein geeigneter Index existiert.
- Diese Kosten müssen geschätzt werden, vor der Ausführung (natürlich)