Datenbanksysteme

Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Tuning des Datenbankschemas

- Neben dem Tuning mit Hilfe von Indexen kann auch das Schema einer Datenbank benutzt werden, um die Performance der Anfrageausführung zu erhöhen.
- Auch hierfür braucht man Annahmen bzw. Erfahrungswerte über Datenbank Workloads, d.h. welche Anfragen werden wie oft ausgeführt.

Wiederholung: Normalisierung von Relationen

Um Qualitätsprobleme im ursprünglichen Entwuft zu beheben, wird das bestehende Relationenschema $\mathcal{R}$ wird in mehrere Relationenschemata $\mathcal{R}_1, ..., \mathcal{R}_n$ zerlegt, die dann "besser" sind.

- Die Güte einer Zerlegung wird mit **Normalformen** beschrieben.
- Normalformen: 1NF, 2NF, 3NF, BCNF, 4NF, ...

Korrektheitskriterien für Zerlegung:

- **Verlustlosigkeit:** Die in der ursprünglichen Relationenausprägung R des Schemas $\mathcal{R}$ enthaltenen Daten müssen aus den Ausprägungen $R_1, ..., R_n$ der neuen Relationenschemata $\mathcal{R}_1, ..., \mathcal{R}_n$ rekonstruierbar sein.
- **Abhängigkeitsbewahrung:** Alle FDs in F_R sollten in den $F_{\mathcal{R}_1}, F_{\mathcal{R}_2}, ...F_{\mathcal{R}_n}$ bewahrt bleiben.

Wiederholung: Dritte Normalform

- Intuition: Nicht-Schlüssel Attribut darf kein anderes Nicht-Schlüssel Attribut bestimmen.

Ein Relationenschema $\mathcal{R}$ ist in dritter Normalform, wenn für jede für $\mathcal{R}$ geltende FD der Form $\alpha \rightarrow B$ mit Attribut $B \in \mathcal{R}$ mindestens eine von drei Bedingungen gilt:

1. $B \in \alpha$, d.h. die FD ist **trivial**
2. α is **Superschlüssel** von R
3. B ist **prim**

Eigenschaften:

- 3NF verhindert **partielle und transitive** Abhängigkeiten
- 3NF $\Rightarrow$ 2NF

Wiederholung: Boyce-Codd-Normalform

Die Boyce-Codd-Normalform (BCNF) ist nochmals eine Verschärfung der 3NF. **Intuition:** Jedes Attribut darf **nur** den gesamten Schlüssel beschreiben und nichts anderes.

Ein Relationenschema $\mathcal{R}$ mit FDs F ist in BCNF, wenn für jede für $\mathcal{R}$ geltende funktionale Abhängigkeit der Form $\alpha \rightarrow B$ mit Attribut $B \in \mathcal{R}$ mindestens **eine** von zwei Bedingungen gilt:

1. $B \in \alpha$, d.h. die FD ist **trivial**
2. α ist Superschlüssel von $\mathcal{R}$

- Unterschied zu 3NF: 3. Bedingung fällt weg (B ist prim).
- Man kann jede Relation **verlustlos** in BCNF-Relationen zerlegen
- **Aber:** manchmal lässt sich dabei die **Abhängigkeitserhaltung nicht** erziehlen!

Tuning des Datenbankschemas: Beispiel

Contracts(cid: integer, supplierId: integer, projectId: integer,
deptid: integer, partid: integer, qty: integer, value: real)
Departments(did: integer, budget: real, annualreport: varchar)
Parts(pid: integer, cost)
Projects(jid: integer, mgr: char(20))
Supplier(sid: integer, address: char(50))

Ein Tupel in der Relation Contract, gegeben durch cid C, ist ein Vertrag, dass Anbieter S (sid) eine Anzahl von Q (qty) Items des Typs P (pid) zu Project J (jid), assoziiert mit Abteilung D (did) liefert, der Wert dieses Vertrags ist *value* V.

Tuning des Datenbankschemas: Beispiel (Cont'd)

Contracts(cid: integer, supplierId: integer, projectId: integer,
deptid: integer, partid: integer, qty: integer, value: real)
Departments(did: integer, budget: real, annualreport: varchar)
Parts(pid: integer, cost)
Projects(jid: integer, mgr: char(20))
Supplier(sid: integer, address: char(50))

Wir haben folgende funktionale Abhängigkeiten (FDs), für Contracts:

- **Ein Projekt kauft ein Produkt mit einem einzelnen Vertrag**.
  D.h. es gibt keine zwei verschiedene Verträge eines Projekts über das selbe Produkt. $JP \rightarrow C$
- **Eine Abteilung kauft nicht mehr als ein Produkt von einem bestimmten Anbieter**: $SD \rightarrow P$
- Natürlich ist C (cid) Schlüssel

Tuning des Datenbankschemas: Beispiel (Cont'd)

- Betrachten wir die funktionale Abhängigkeit $SD \rightarrow P$
- P ist prim, da Teil des Kandidatenschlüssels JP, also ist **Contracts in 3NF**
- Aber sie ist **nicht in BCNF.**

Zerlegung in BCNF

- Anhand von $SD \rightarrow P$
- D.h. wir bekommen zwei Relationen SDP und $CSJDQV$.
- Leider nicht abhängigkeitsbewahrend: $JP \rightarrow C$ kann nicht mehr überprüft werden; müssten also Relation JPC noch dazu anlegen.

Was ist nun falls folgende Frage häufig auftritt: Finde die Anzahl (Q) der bestellten Teile für P in Vertrag C?

Denormalisierung

Es kann Sinn machen aus Gründen der Performance auf eine mögliche starke Normalform zu verzichten.

- Die durch die bei Anfragen evtl. wegfallenden Joins wird die Performance erhöht.
- **Jedoch werden Daten redundant gespeichert.**
- D.h. es besteht ein **nicht trivialer Tradeoff zwischen Performance und Redundanz.**

Denormalisierung (2)

- Wie wir gesehen haben ist die **Relation Contracts in 3NF**.
- Was ist wenn eine **häufig auftretende Anfrage** überprüfen soll, dass der Wert eines Vertrags das Budget der Abteilung nicht übersteigt?

Zur Erinnerung:

Contracts(cid: integer, supplierId: integer, projectId: integer,
deptid: integer, partid: integer, qty: integer, value: real)
Departments(did: integer, budget: real, annualreport: varchar)
...

Idee

- **Wir könnten budget aus der Relation Departments in die Relation Contracts aufnehmen.**
- Was bedeutet dies?

Denormalisierung (3)

- **Die Anfrage könnte nun sehr effizient ausgeführt werden.**
- Aber mit dieser Änderung ist Contracts **nicht mehr in 3NF**, da wir nun die funktionale Abhängigkeit $D \rightarrow B$ haben.

- Trotzdem könnte man mit dieser Redundanz evtl. leben, je nachdem wie wichtig die Performance der Anfrage ist (**Tradeoff!**)

Vertikale Partitionierung

- Nehmen wir die Zerlegung der Relation Contracts in SDP und CSJDQV an, welche nun in BCNF sind.
- Weiter nehmen wir an, dass die folgenden beiden Anfragen häufig auftreten:
  - Finde alle Verträge (Contracts) vom Anbieter S
  - Finde alle Verträge von Abteilung D

Zerlegung von CSJDQV

- Wir könnten die Relation CSJDQV zerlegen in: CS, CD und CJQV.
- Was ist der Vorteil solch einer Zerlegung?
- Was ist wenn die folgende Anfrage häufig auftritt: Finde die Summe aller Values der Verträge eines bestimmten Anbieters?

Zusammenfassung

- Wir haben gesehen was Indexe sind: Clustered oder nicht clustered, sekundär und primär.
- Guidelines wann Indexe angelegt werden sollten.
- Wann ein Index ausgenutzt werden kann oder nicht: Indexe müssen auch instand gehalten werden ...
- B+ Index vs. Hash-Index
- Neben Indexen kann auch das DB Schema so verändert werden, dass Anfragen effizienter ausführbar sind:
- Denormalisierung und Partitionierung.

Implementierung von Join Operatoren

Bislang nur Bedeutung (Ergebnis) eines Joins

- Aber nicht wie dieser berechnet wird.

Eingabe/Ausgabe

- Eingabe: Tabelle R und S
- Ausgabe: Alle Tupel in $R \bowtie S$
- Mit bekannter Semantik

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

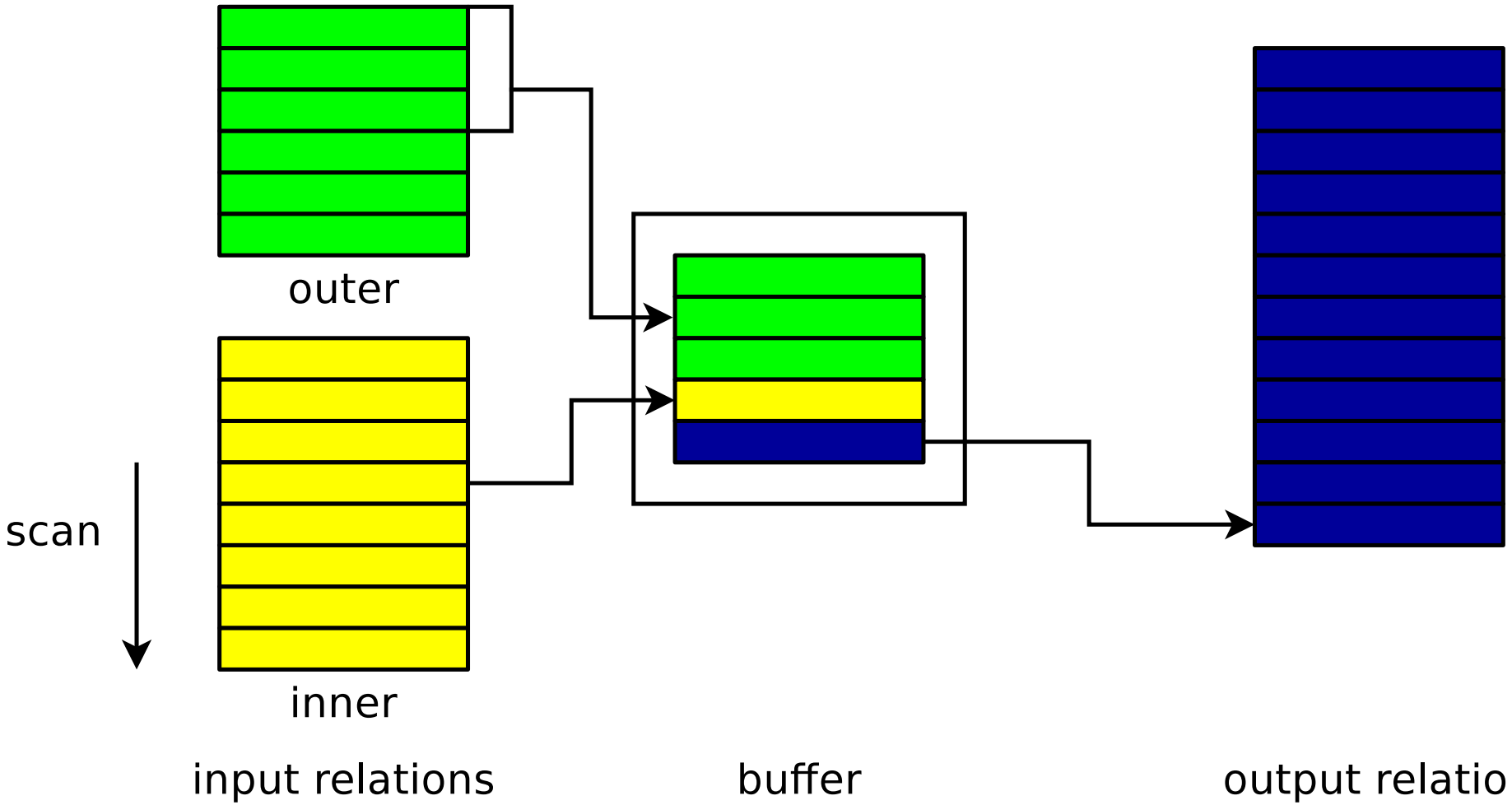
Return Verbund von aktuellem linken und aktuellem rechten Tupel

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:

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

Kostenschätzung (als Anzahl Zugriffe auf Blöcke)

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

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

Kosten

- **Relation `reserved` als äußere Relation:**
  $2000 + \lceil (2.000/4) \rceil * 10 = 7000$
- **Relation `customer` als äußere Relation:**
  $10 + \lceil (10/4) \rceil * 2000 = 6010$

Index-based Nested Loop Join

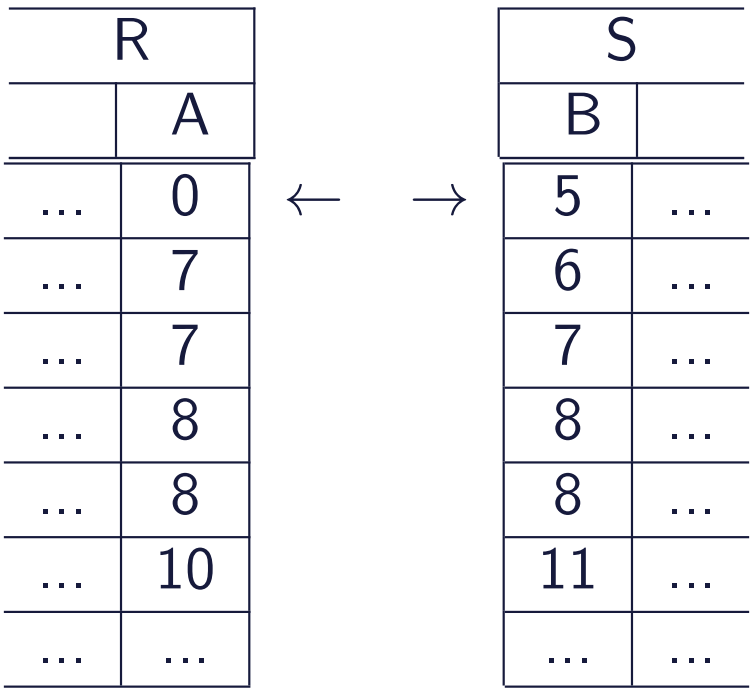
for each $r \in R$
  for each $s \in S$ **where** $[s.B = r.A]$
    then $Res := Res \cup (r \circ s)$

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

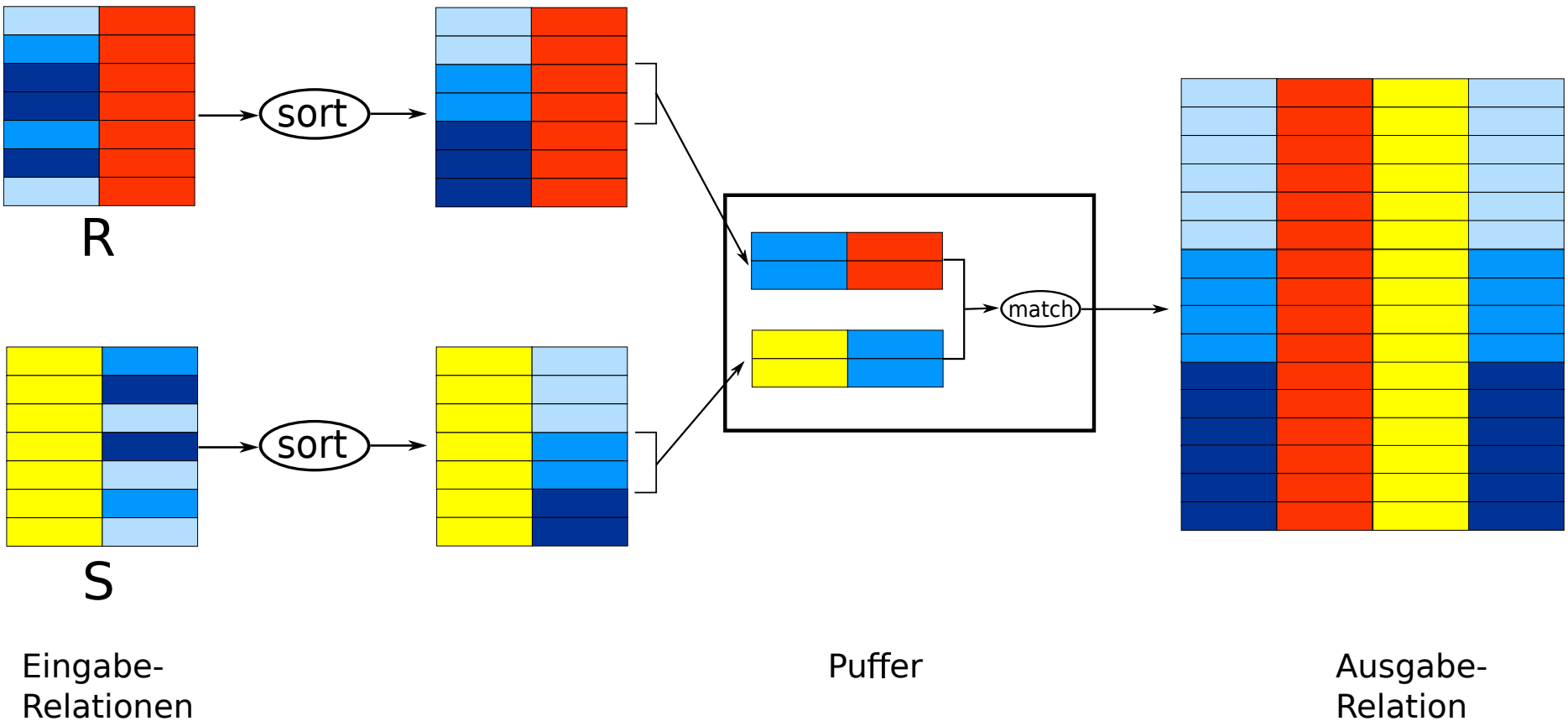
$\bowtie$

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

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?

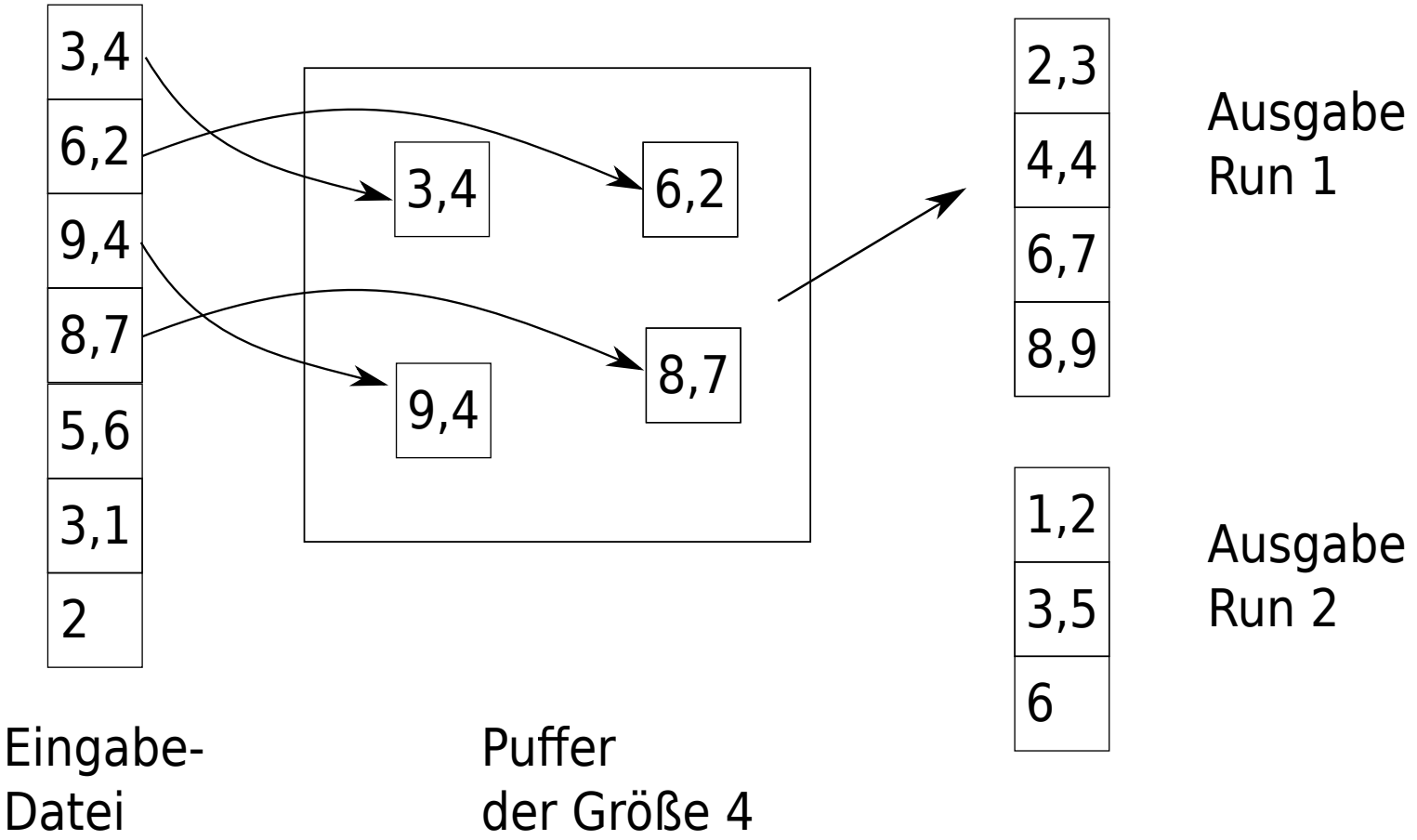
Externes Sortieren

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: b (d.h. Größe der Datein in Anzahl Blöcke)
- Größe des zum sortieren nutzbaren Hauptspeichers (Puffer) in Blöcke: n_B
- Anzahl Runs gegeben durch

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

- Beispiel: $n_B = 5$ Blöcke, $b = 1024$ 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

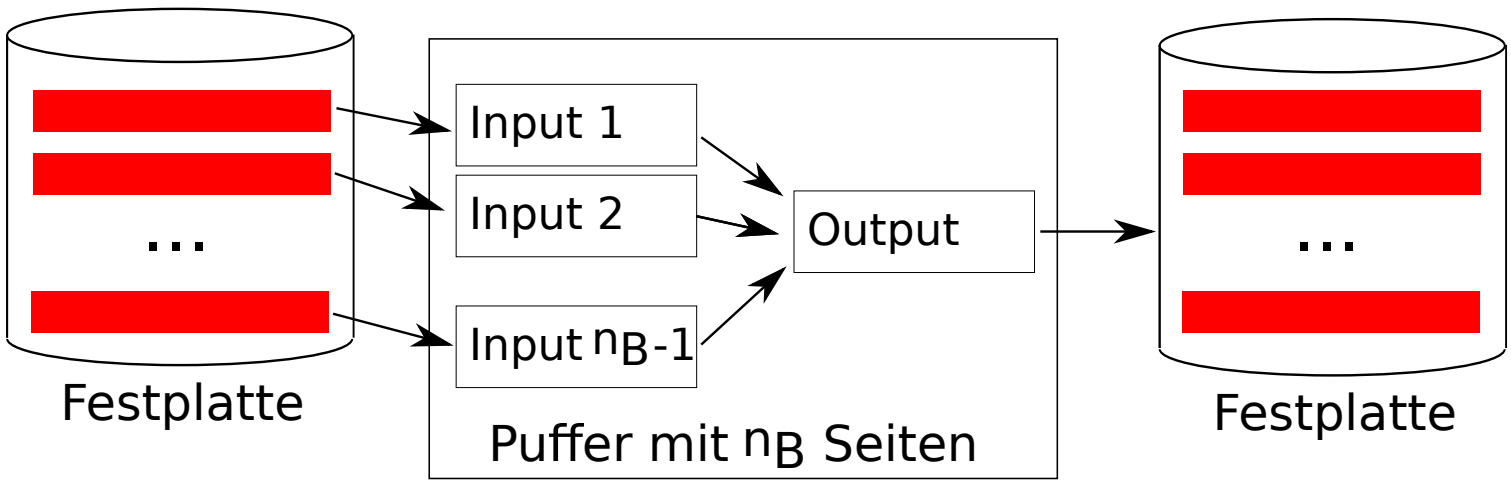
Sortier-Phase

- Lese die nächsten n_B Blocks 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 b Seiten (Blöcke) und Anzahl Puffer Seiten n_B.

b	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

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?