# Anfrage-Optimierung und -Bearbeitung in Verteilten DBMS

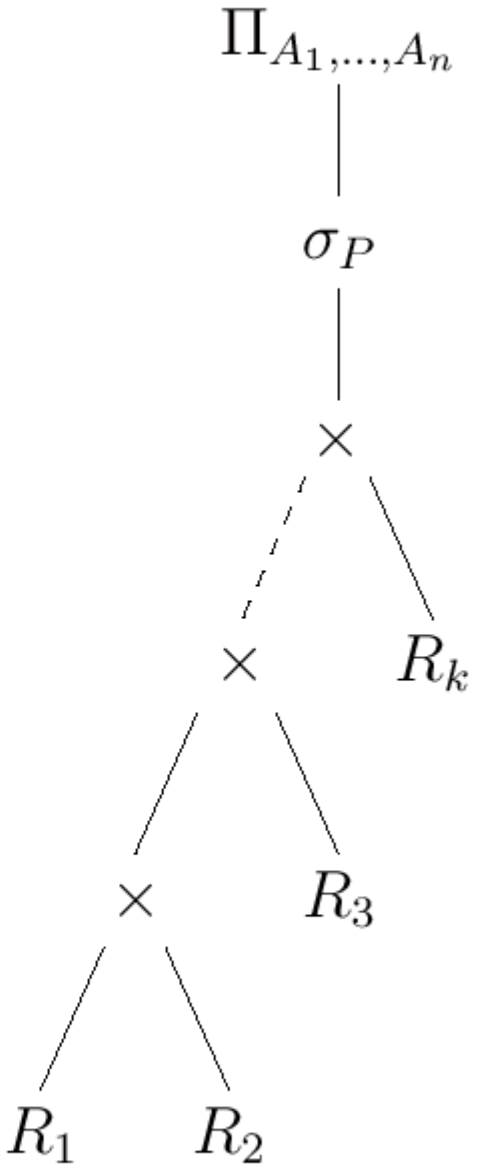
**Relationenalgebra,**
**Transformationen, Vereinfachungen,**
**verteilte Joins,**
**Kostenmodell**

# Formale Grundlagen

- Relationenalgebra
  - Selektion $\sigma$
  - Projektion $\Pi$
  - Kreuzprodukt $\times$
  - Vereinigung $\cup$
  - Differenz -
  - Umbenennung: $\rho_V(R)$ $\rho_{A \leftarrow B}(R)$
  - Join/Verbund, Semijoin, Outer Joins $\otimes$
  - Durchschnitt $\cap$
  - Division $\div$

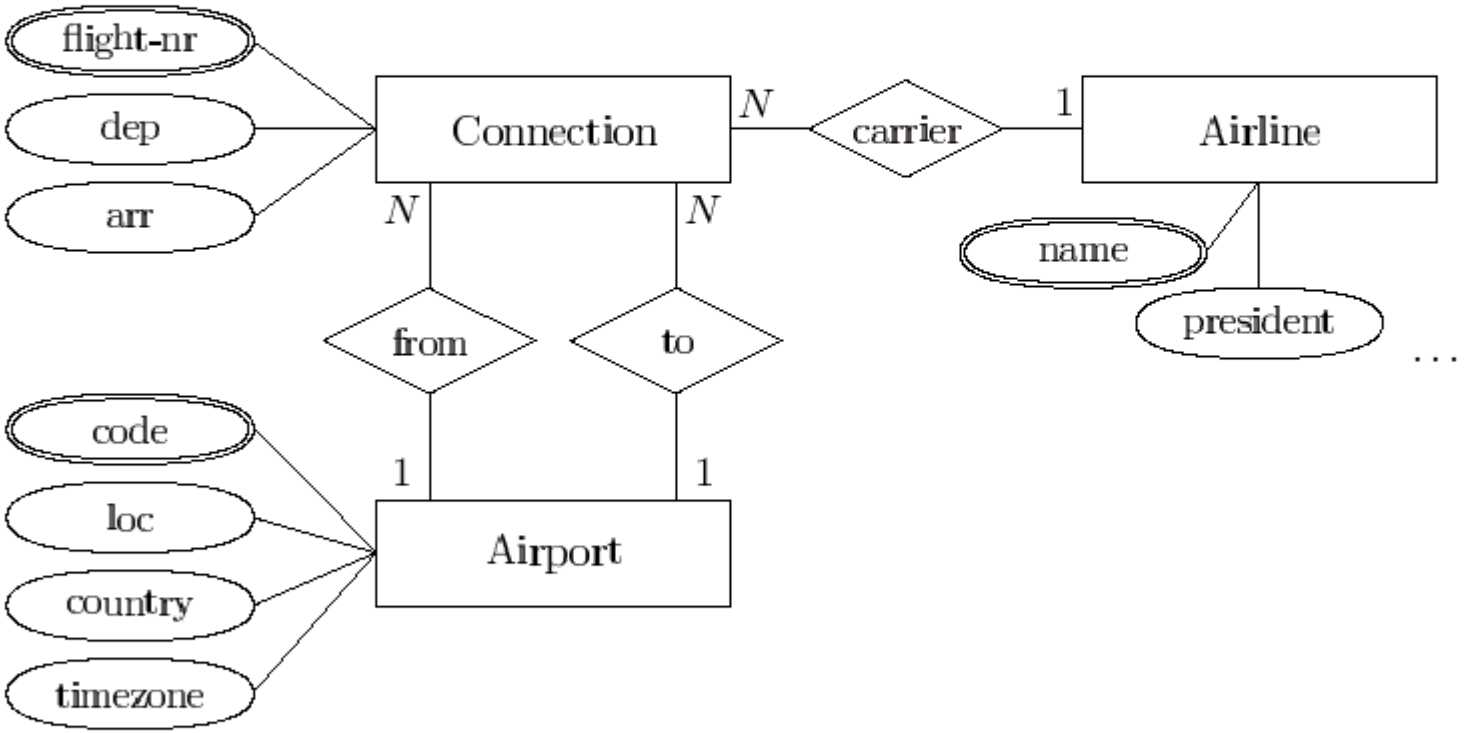
**select** $A_1, \ldots, A_n$
**from** $R_1, \ldots, R_k$
**where** $P$;

kanonische
$\Rightarrow$
Übersetzung



# Ein weiteres Beispiel: Flugreservierung

**Entity-Relationship-Schema**



## Relationales Schema

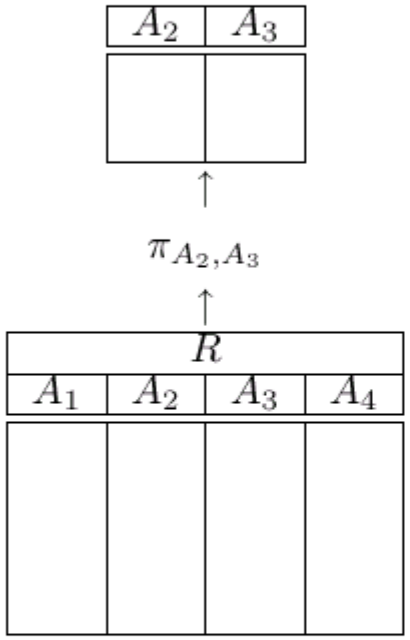
| Connection | | | | | |
|---|---|---|---|---|---|
| flight-nr | from | to | dep | arr | carrier |
| 4711 | FRA | JFK | 1500 | 1730 | UA |
| 4567 | FRA | LGA | 1230 | 1415 | AA |
| 3412 | JFK | LAX | 1830 | 2115 | DL |
| 6734 | MUC | FRA | 1310 | 1400 | LH |
| ... | ... | ... | ... | ... | ... |

| Airline | |
|---|---|
| name | ... |
| AA | ... |
| DL | ... |
| LH | ... |
| UA | ... |
| ... | ... |

| Airport | | | |
|---|---|---|---|
| code | loc | country | timezone |
| FRA | Frankfurt | Germany | MET |
| MUC | München | Germany | MET |
| LAX | Los Angeles | USA | PST |
| JFK | New York | USA | EST |
| LGA | New York | USA | EST |
| ... | ... | ... | ... |

## Relationenalgebra: $\pi, \sigma, \times, \bowtie, \cup, -, \cap, \div, \rtimes, \ltimes, \overline{\bowtie}, \overline{\rtimes}, \overline{\ltimes}, \ldots$

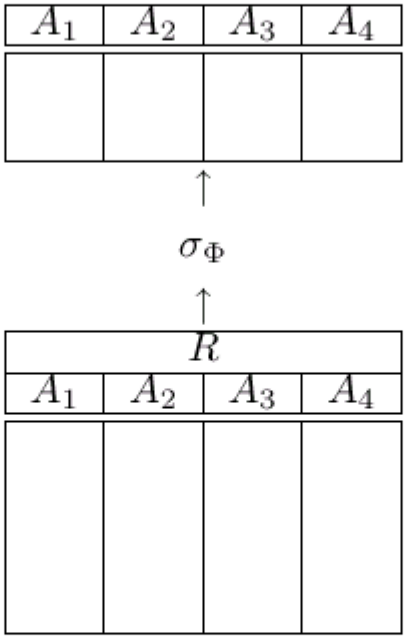
### Projektion $\pi$




**Beispiel**

$\pi_{\text{loc}}(\text{Airport})$

### Selektion $\sigma$




**Beispiel**

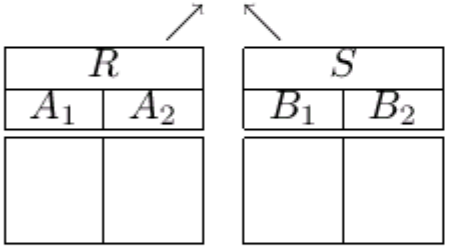
$\sigma_{\text{to=from}}(\text{Connection})$

**Kreuzprodukt** $\times$

| $R.A_1$ | $R.A_2$ | $S.B_1$ | $S.B_2$ |
|---|---|---|---|
| | | | |

$\uparrow$

$\times$




**Beispiel**

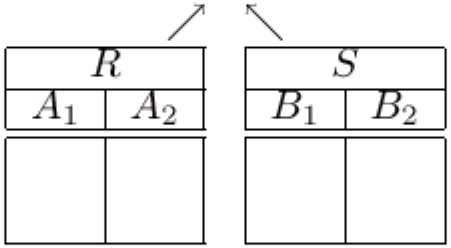
Airport $\times$ Connection

**Verbund, Join** $\bowtie_{\ldots}$

| $R.A_1$ | $R.A_2$ | $S.B_1$ | $S.B_2$ |
|---|---|---|---|
| | | | |

$\uparrow$

$\bowtie_{R.A_2=S.B_1}$




**Beispiel**

Airport $\bowtie_{\text{code=from}}$ Connection

## SQL-Anfrage: Von München direkt nach NY?

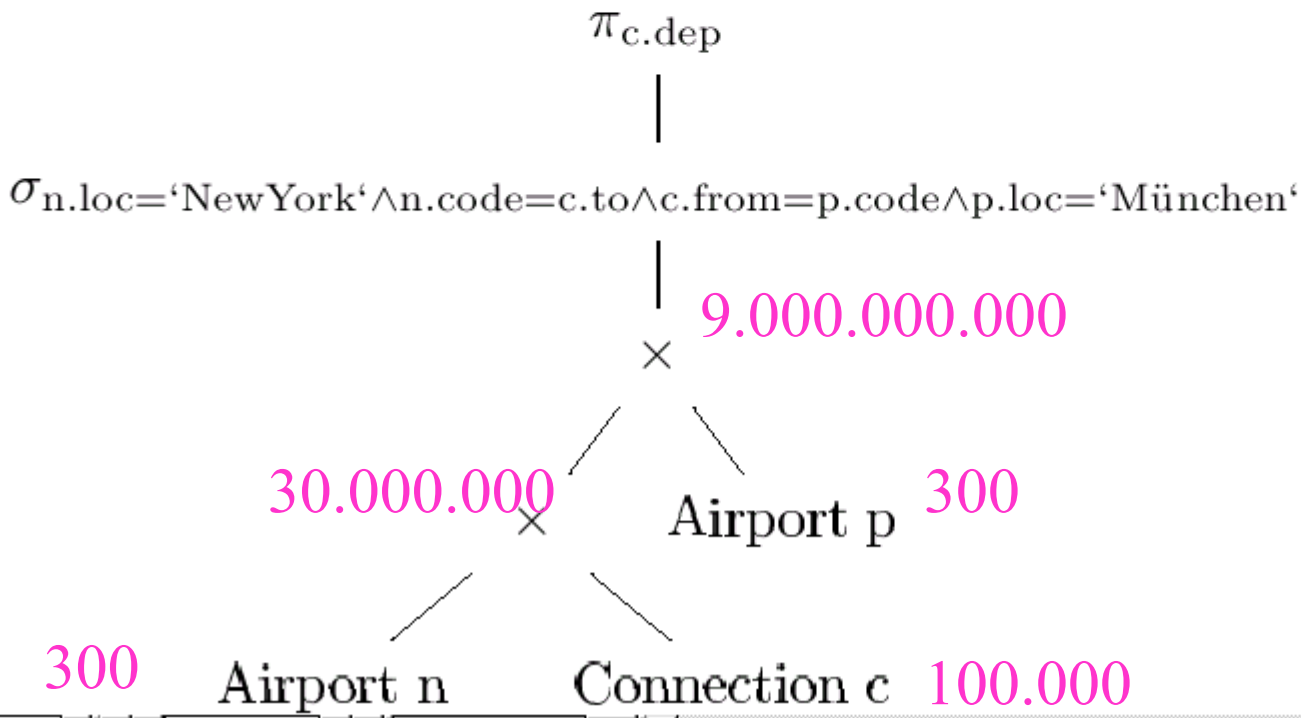
**select** c.dep
**from** Airport n, Connection c, Airport p
**where** n.loc = 'New York' **and**
n.code = c.to **and**
c.from = p.code **and**
p.loc = 'München'

| Airport p | | |
|---|---|---|
| p.code | p.loc | ... |
| ... | ... | ... |
| MUC | München | ... |
| ... | ... | ... |

| Connection c | | | |
|---|---|---|---|
| ... | c.from | c.to | ... |
| | | | |

| Airport n | | |
|---|---|---|
| n.code | n.loc | ... |
| ... | ... | ... |
| JFK | New York | ... |
| LGA | New York | ... |
| ... | ... | ... |

```
select c.dep
from Airport n, Connection c, Airport p
where n.loc = 'New York' and
        n.code = c.to and
        c.from = p.code and
        p.loc = 'München'
```



# Äquivalenzregeln der Relationenalgebra

1. Join, Vereinigung, Schnitt und Kreuzprodukt sind kommutativ, also:

$$\begin{aligned} R_1 \bowtie R_2 &= R_2 \bowtie R_1 \\ R_1 \cup R_2 &= R_2 \cup R_1 \\ R_1 \cap R_2 &= R_2 \cap R_1 \\ R_1 \times R_2 &= R_2 \times R_1 \end{aligned}$$

2. Selektionen sind untereinander vertauschbar.

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

3. Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ, also:

$$\begin{aligned} R_1 \bowtie (R_2 \bowtie R_3) &= (R_1 \bowtie R_2) \bowtie R_3 \\ R_1 \cup (R_2 \cup R_3) &= (R_1 \cup R_2) \cup R_3 \\ R_1 \cap (R_2 \cap R_3) &= (R_1 \cap R_2) \cap R_3 \\ R_1 \times (R_2 \times R_3) &= (R_1 \times R_2) \times R_3 \end{aligned}$$

4. Konjunktionen in einer Selektionsbedingung können in mehrere Selektionen aufgebrochen, bzw. nacheinander ausgeführte Selektionen können durch Konjunktionen zusammengefügt werden.

$$\sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots))$$

5. Geschachtelte Projektionen können eliminiert werden.

$$\Pi_{l_1}(\Pi_{l_2}(\ldots(\Pi_{l_n}(R))\ldots)) = \Pi_{l_1}(R)$$

Damit eine solche Schachtelung überhaupt sinnvoll ist, muß gelten:

$$l_1 \subseteq l_2 \subseteq \ldots \subseteq l_n \subseteq \mathcal{R} = sch(R)$$

# Aquivalenzen in der relationalen Algebra

6. Eine Selektion kann an einer Projektion „vorbeigeschoben“ werden, falls die Projektion keine Attribute aus der Selektionsbedingung entfernt. Es gilt also

$$\Pi_l(\sigma_p(R)) = \sigma_p(\Pi_l(R)), \text{ falls } attr(p) \subseteq l$$

7. Selektionen können an Joinoperationen (oder Kreuzprodukten) vorbeigeschoben werden, falls sie nur Attribute *eines* der beiden Join-Argumente verwenden. Enthält die Bedingung $p$ beispielsweise nur Attribute aus $\mathcal{R}_1$, dann gilt

$$\sigma_p(R_1 \bowtie R_2) = \sigma_p(R_1) \bowtie R_2$$

8. Auf ähnliche Weise können auch Projektionen verschoben werden. Hier muß allerdings beachtet werden, daß die Join-Attribute bis zum Join erhalten bleiben.

8. Auf ähnliche Weise können auch Projektionen verschoben werden. Hier muß allerdings beachtet werden, daß die Join-Attribute bis zum Join erhalten bleiben.

$$\Pi_l(R_1 \bowtie_p R_2) = \Pi_l(\Pi_{l_1}(R_1) \bowtie_p \Pi_{l_2}(R_2)) \text{ mit}$$
$$l_1 = \{A|A \in \mathcal{R}_1 \cap l\} \cup \{A|A \in \mathcal{R}_1 \cap attr(p)\} \text{ und}$$
$$l_2 = \{A|A \in \mathcal{R}_2 \cap l\} \cup \{A|A \in \mathcal{R}_2 \cap attr(p)\}$$

9. Selektionen können mit Mengenoperationen wie Vereinigung, Schnitt und Differenz vertauscht werden, also:

$$\begin{aligned} \sigma_p(R \cup S) &= \sigma_p(R) \cup \sigma_p(S) \\ \sigma_p(R \cap S) &= \sigma_p(R) \cap \sigma_p(S) \\ \sigma_p(R - S) &= \sigma_p(R) - \sigma_p(S) \end{aligned}$$

10. Der Projektions-Operator kann mit der Vereinigung vertauscht werden.

$$\Pi_l(R_1 \cup R_2) = \Pi_l(R_1) \cup \Pi_l(R_2)$$

Eine Vertauschung der Projektion mit Durchschnitt und Differenz ist allerdings nicht zulässig.

11. Eine Selektion und ein Kreuzprodukt können zu einem Join zusammengefaßt werden, wenn die Selektionsbedingung eine Joinbedingung ist. Für Equijoins gilt beispielsweise

$$\sigma_{R_1.A_1=R_2.A_2}(R_1 \times R_2) = R_1 \bowtie_{R_1.A_1=R_2.A_2} R_2$$

12. Auch an Bedingungen können Veränderungen vorgenommen werden. Beispielsweise kann eine Disjunktion mit Hilfe von DeMorgan's Gesetz in eine Konjunktion umgewandelt werden, um vielleicht später die Anwendung von Regel 4 zu ermöglichen:

$$\neg(p_1 \vee p_2) = \neg p_1 \wedge \neg p_2$$

# Besonders wichtige Äquivalenzen in Verteilten DBMS

$$(R1 \cup R2) \otimes_p (S1 \cup S2) = (R1 \otimes_p S1) \cup (R1 \otimes_p S2) \cup (R2 \otimes_p S1) \cup (R2 \otimes_p S2)$$

$$(\cup_{1 \le i \le n} R_i) \otimes_p (\cup_{1 \le j \le m} S_j) = \cup_{1 \le i \le n} \cup_{1 \le j \le m} (R_i \otimes_p S_j)$$

# Semijoin/Abgeleitete Partitionierung: Join-Optimierung

## Algebraische Äquivalenzen

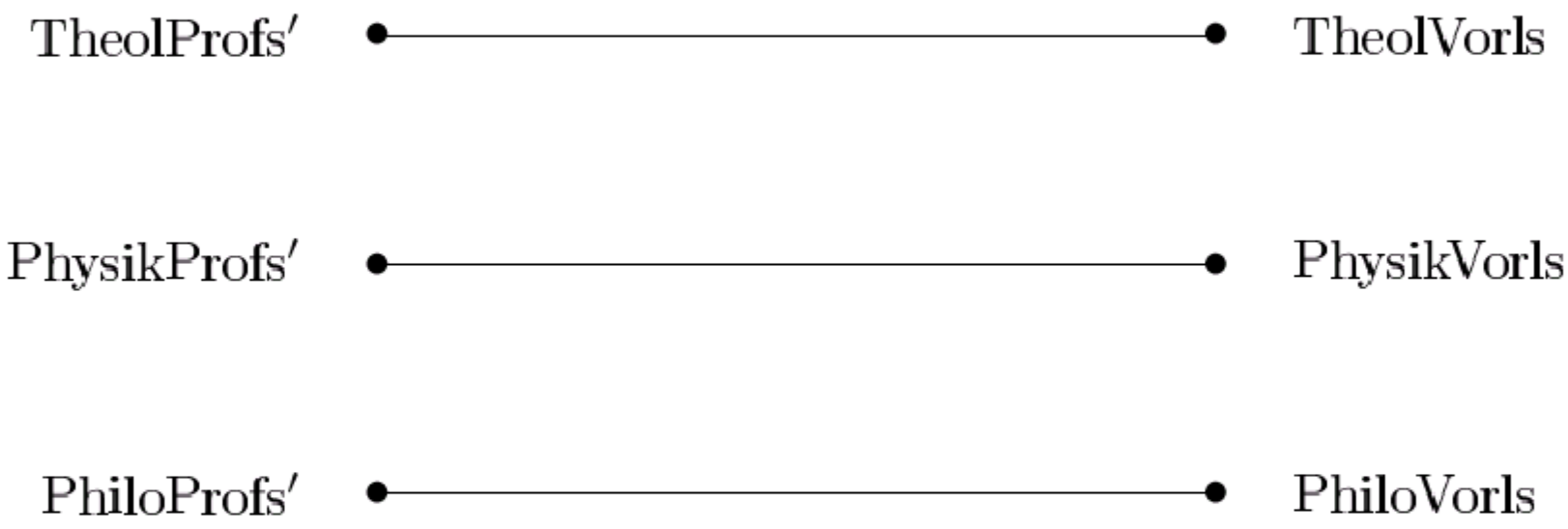
$$S_i = S \ltimes_p R_i \quad \text{mit} \quad S = S_1 \cup \cdots \cup S_n$$

$$R_i \bowtie_p S_j = \emptyset \quad \text{für} \quad i \neq j.$$

$$(R_1 \cup \cdots \cup R_n) \bowtie_p (S_1 \cup \cdots \cup S_n) = (R_1 \bowtie_p S_1) \cup (R_2 \bowtie_p S_2) \cup \cdots \cup (R_n \bowtie_p S_n)$$

$$(\text{TheolVorls} \cup \text{PhysikVorls} \cup \text{PhiloVorls}) \bowtie_{\ldots} (\text{TheolProfs} \cup \text{PhysikProfs} \cup \text{PhiloProfs})$$

**es reichen folgende Joins**




$$\sigma_p(R_1 \cup R_2) = \sigma_p(R_1) \cup \sigma_p(R_2)$$
$$\Pi_L(R_1 \cup R_2) = \Pi_L(R_1) \cup \Pi_L(R_2)$$

# Selektionsprädikate „aufbrechen“

$$\sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots))$$

$\pi_{\text{c.dep}}$

|

$\sigma_{\text{p.loc='München'}}$

|

$\sigma_{\text{c.from=p.code}}$

|

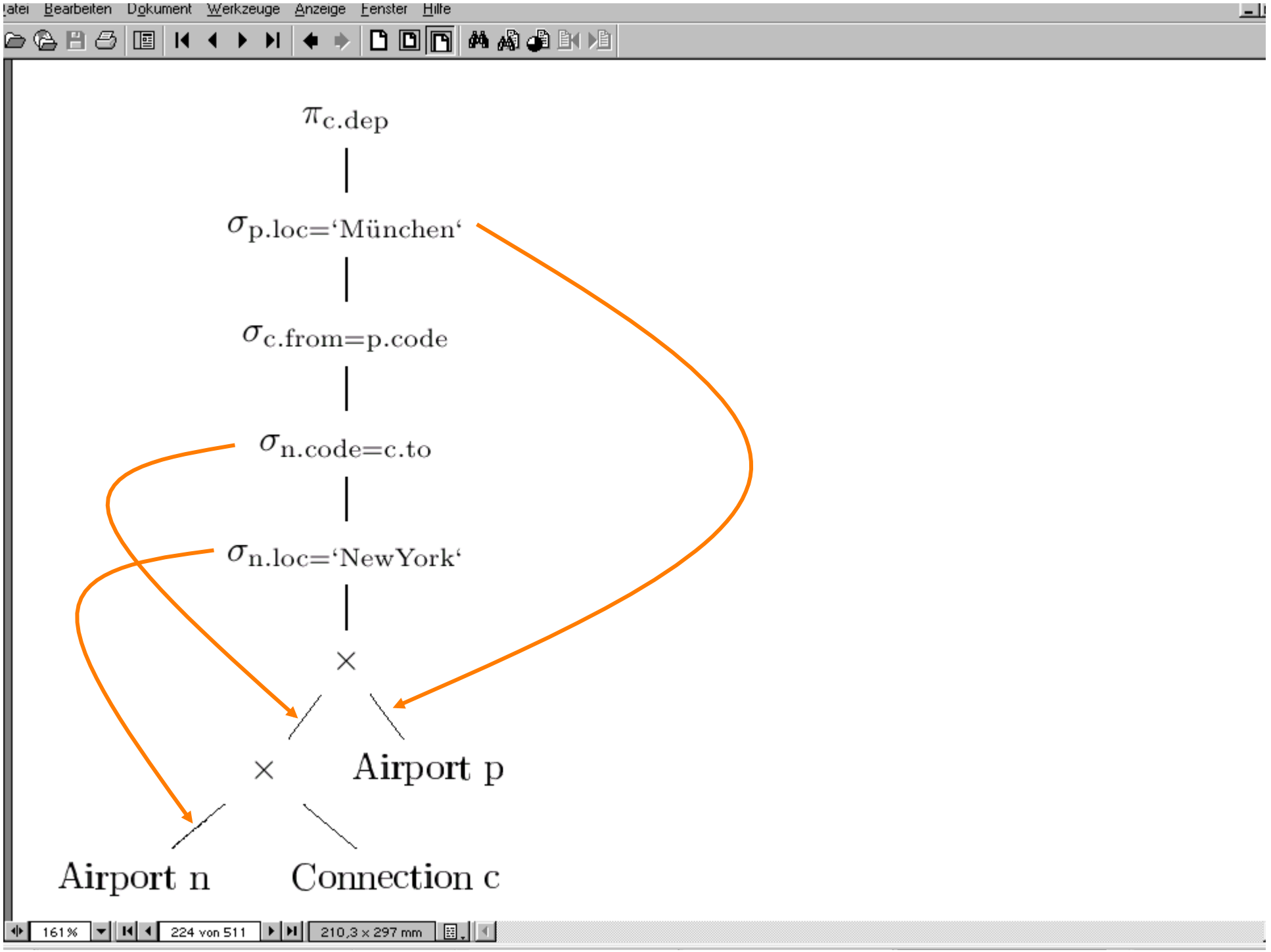
$\sigma_{\text{n.code=c.to}}$

|

$\sigma_{\text{n.loc='NewYork'}}$

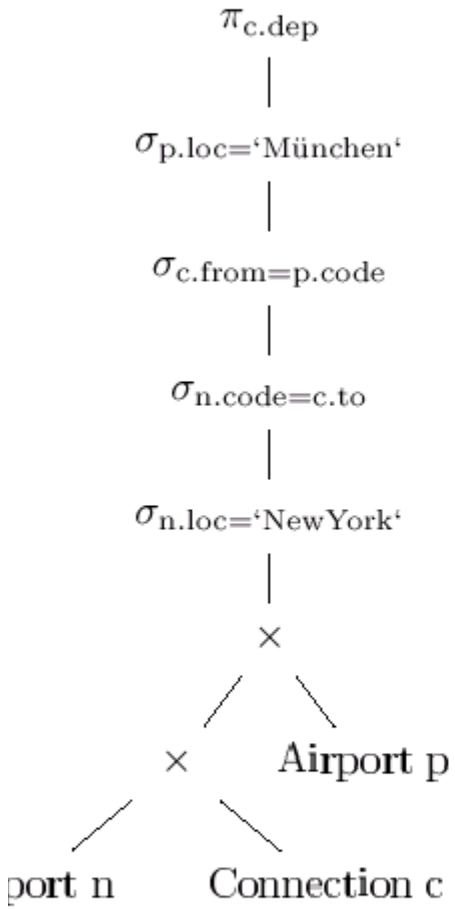
|

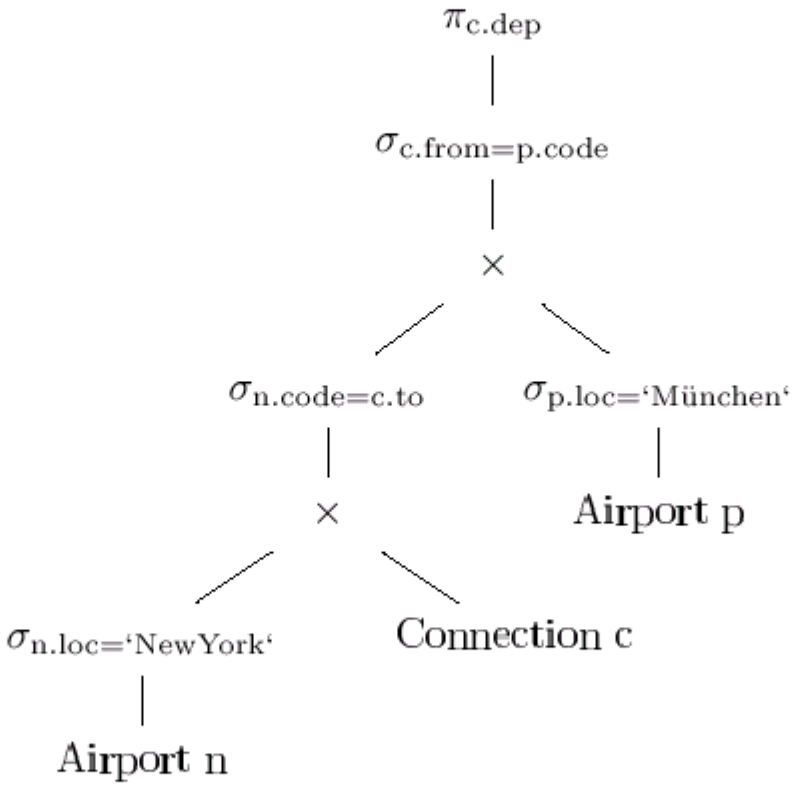
$\times$


Datei Bearbeiten Dokument Werkzeuge Anzeige Fenster Hilfe
$\pi_{c.dep}$
$\sigma_{p.loc='München'}$
$\sigma_{c.from=p.code}$
$\sigma_{n.code=c.to}$
$\sigma_{n.loc='NewYork'}$
×
×
Airport p
Airport n
Connection c
161 %
224 von 511
210,3 x 297 mm

## „Pushing Selections“

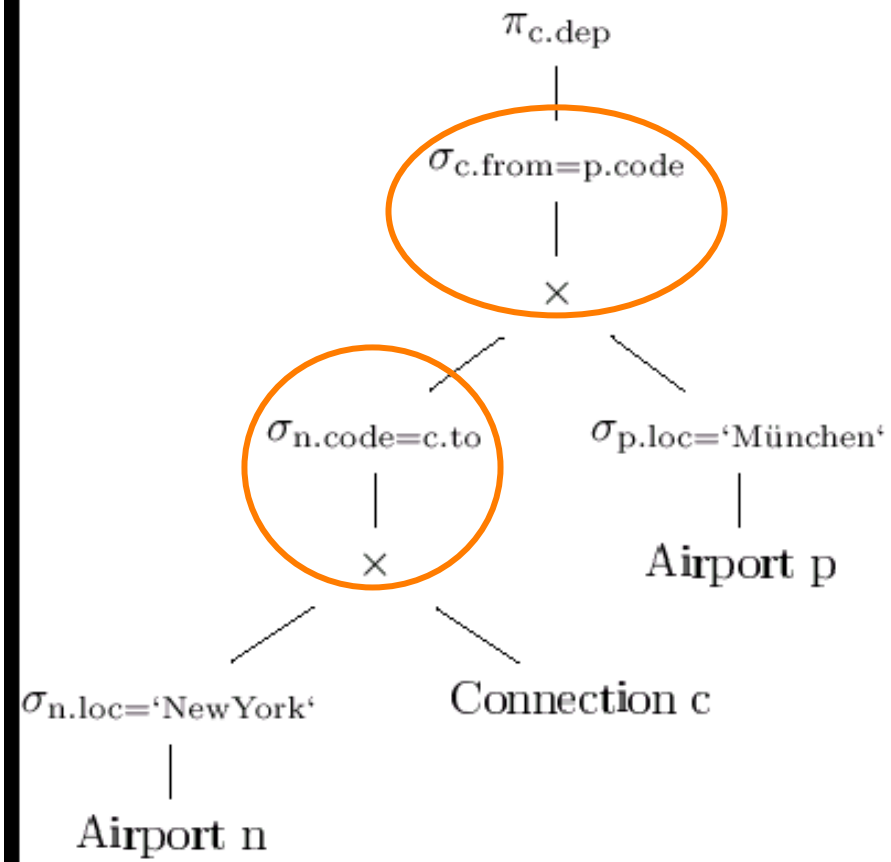
$$
\begin{aligned}
\sigma_p(\sigma_q(R)) &= \sigma_q(\sigma_p(R)) \\
\sigma_p(R_1 \bowtie R_2) &= \sigma_p(R_1) \bowtie R_2 \\
\sigma_p(R_1 \times R_2) &= \sigma_p(R_1) \times R_2
\end{aligned}
$$

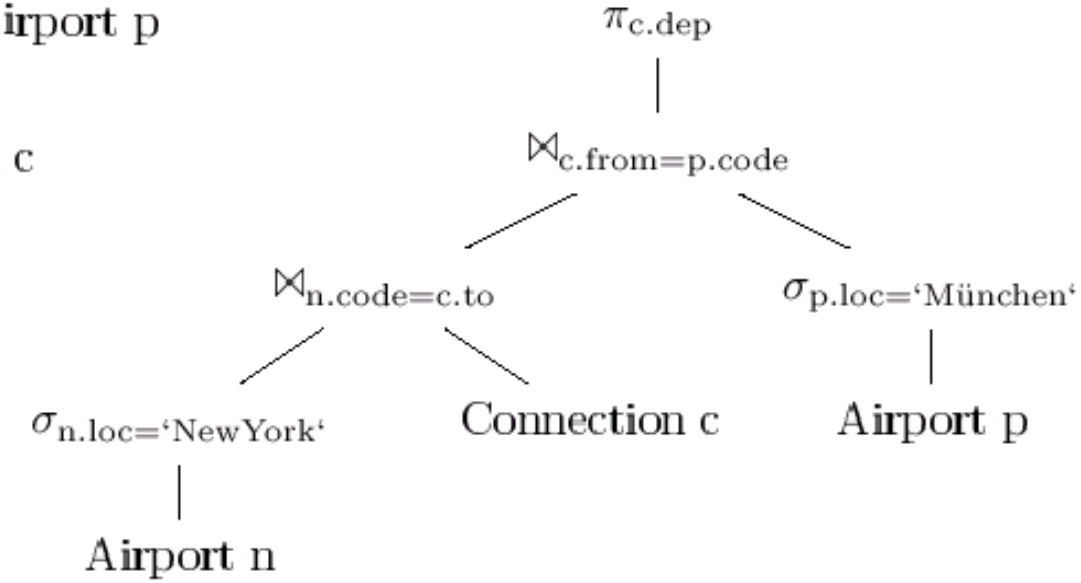


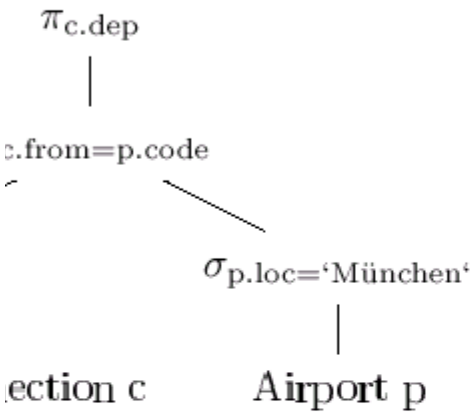
# Zusammenfassung von $\sigma\times$ zu $\bowtie$

$$\sigma_{R.A=S.B}(R \times S) = R \bowtie_{R.A=S.B} S$$

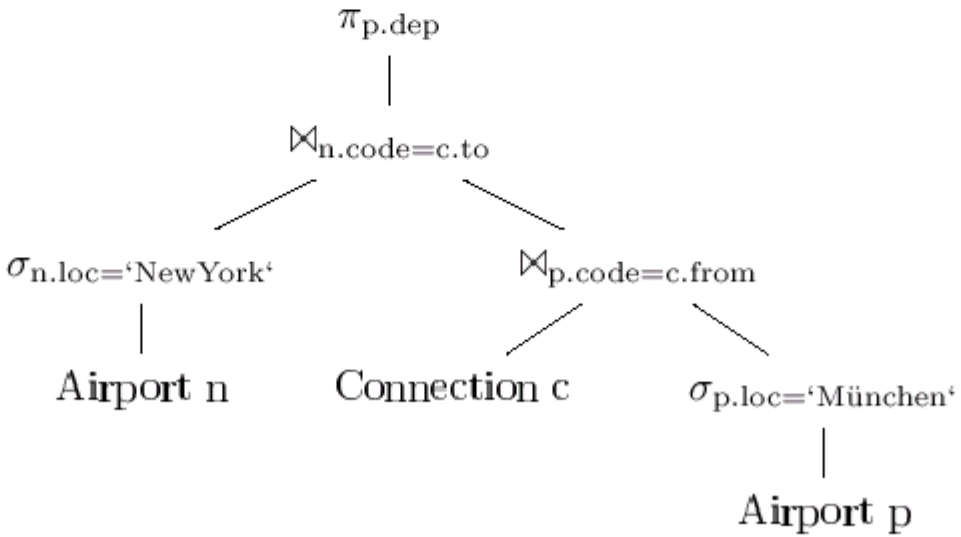


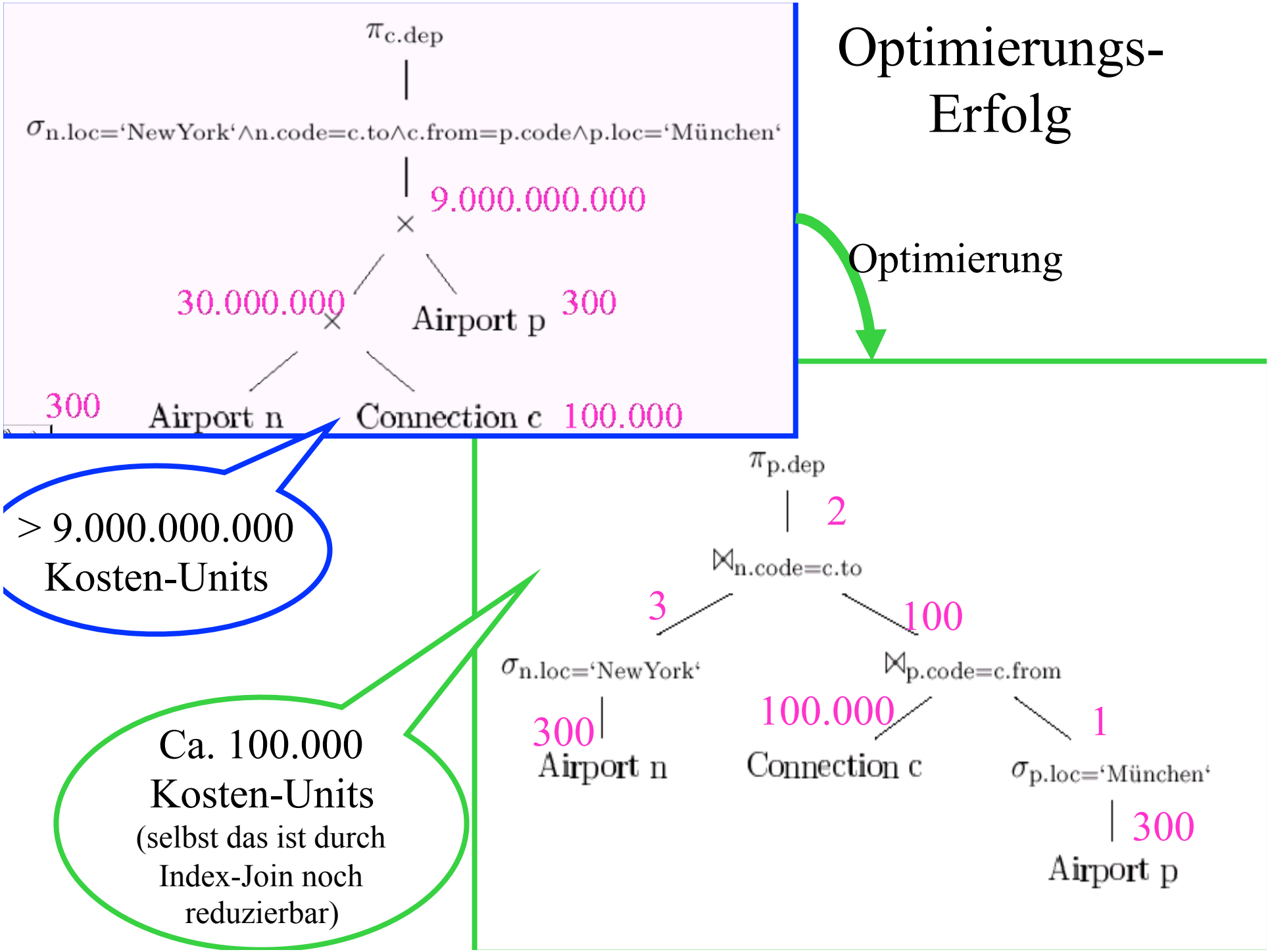



# Optimierung der Join-Reihenfolge




$$\boxed{(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)}$$

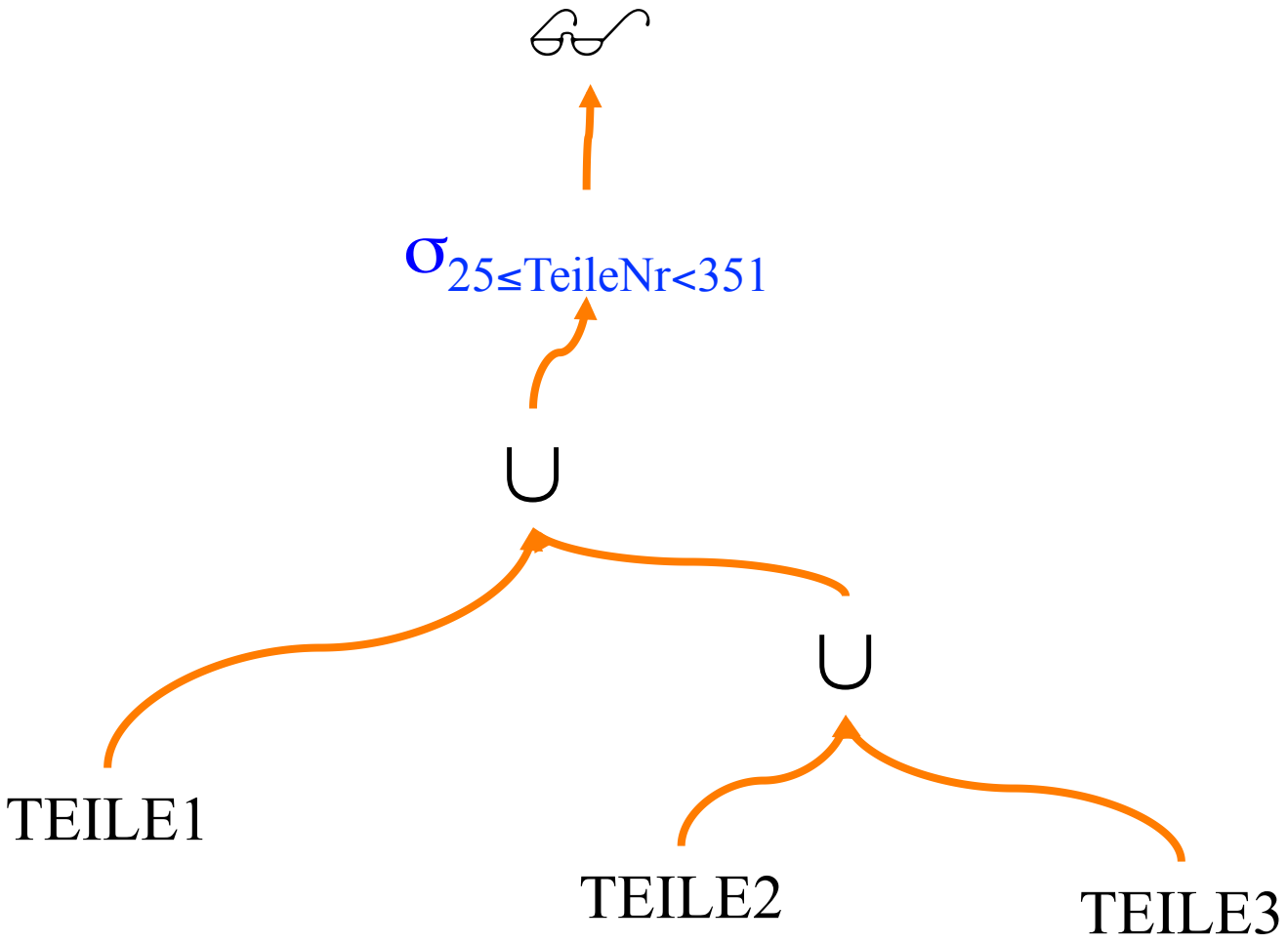




$\pi_{c.dep}$
$\sigma_{n.loc='NewYork' \wedge n.code=c.to \wedge c.from=p.code \wedge p.loc='München'}$
9.000.000.000
×
30.000.000
×
Airport p
300
300
Airport n
Connection c
100.000
Optimierungs-
Erfolg
Optimierung
> 9.000.000.000
Kosten-Units
Ca. 100.000
Kosten-Units
(selbst das ist durch
Index-Join noch
reduzierbar)
$\pi_{p.dep}$
2
$\bowtie_{n.code=c.to}$
3
100
$\sigma_{n.loc='NewYork'}$
$\bowtie_{p.code=c.from}$
300
Airport n
100.000
Connection c
1
$\sigma_{p.loc='München'}$
300
Airport p

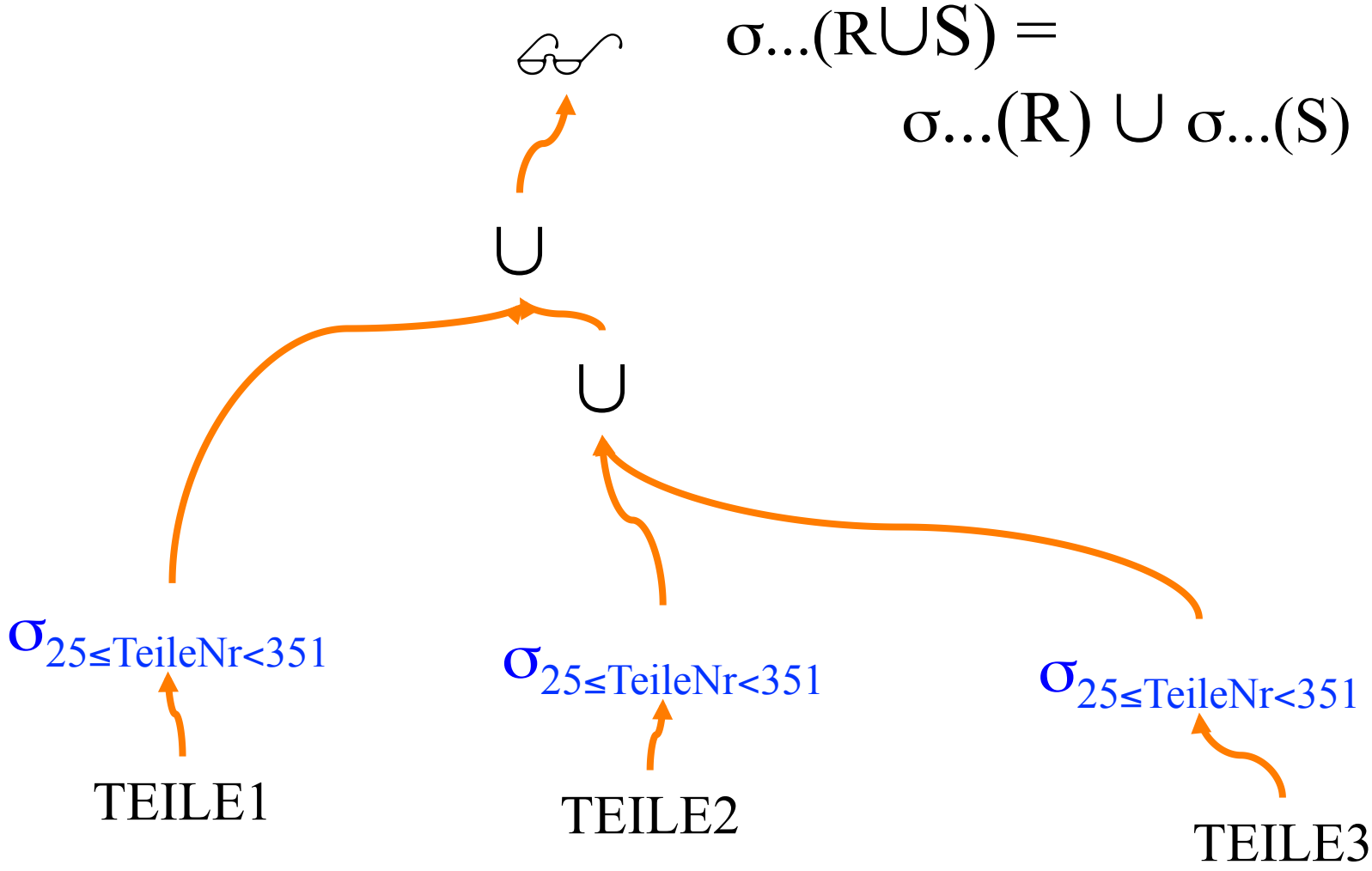
# Transformation globaler Anfragen in lokale Anfragen

- TEILE: {[TeileNr,LiefNr,Preis,...]}
  - TEILE1 := $\sigma_{0 \le TeileNr < 300}$ TEILE
  - TEILE2 := $\sigma_{300 \le TeileNr < 500}$ TEILE
  - TEILE3 := $\sigma_{500 \le TeileNr < 999}$ TEILE
  - TEILE = TEILE1 $\cup$ TEILE2 $\cup$ TEILE3
- Anfrage: $\sigma_{25 \le TeileNr < 351}$ TEILE
- Transformation durch Einsetzen der Partitionierungs-Definition
  - $\sigma_{25 \le TeileNr < 351}$ (TEILE1 $\cup$ TEILE2 $\cup$ TEILE3)

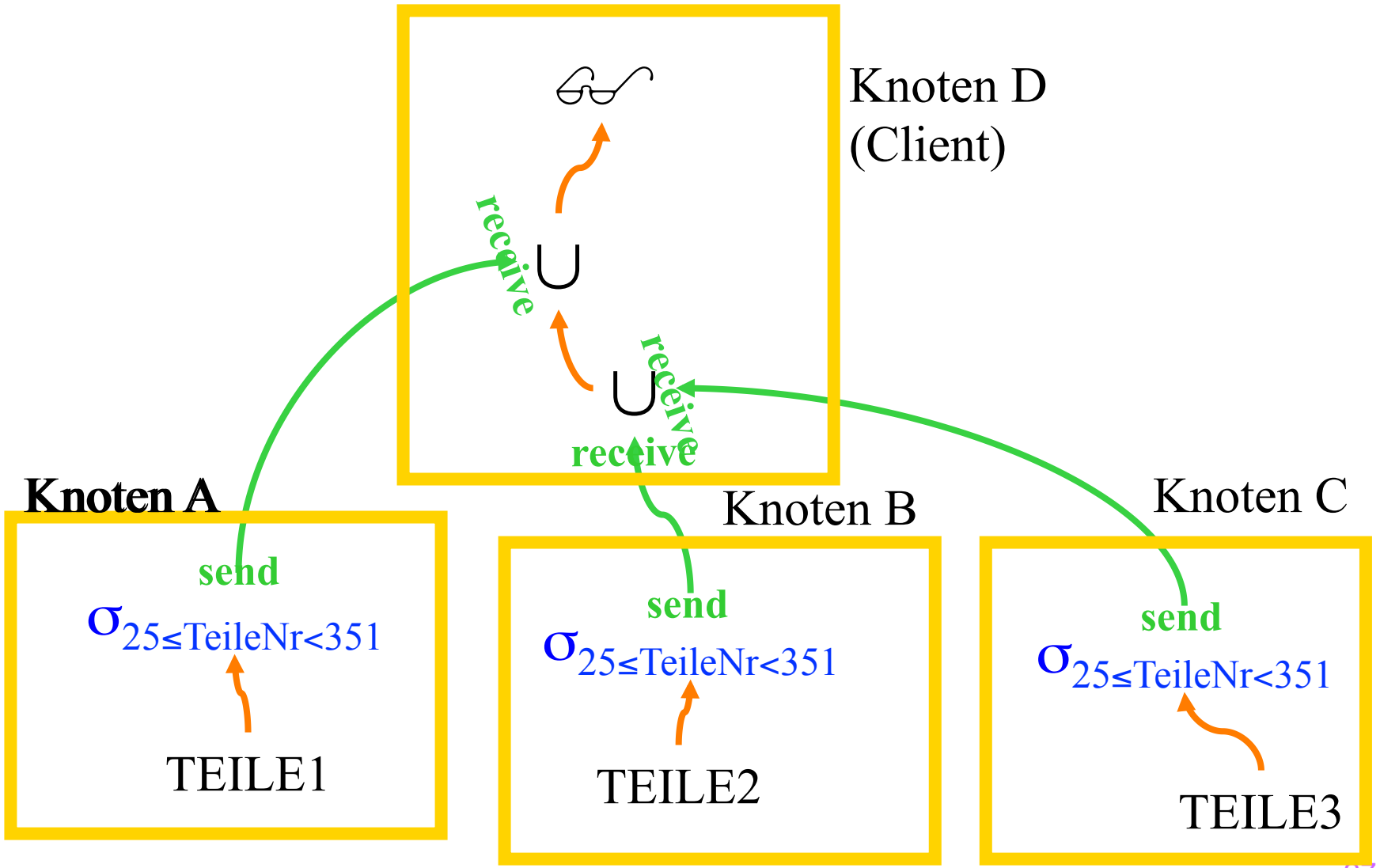
# Operator-Baum



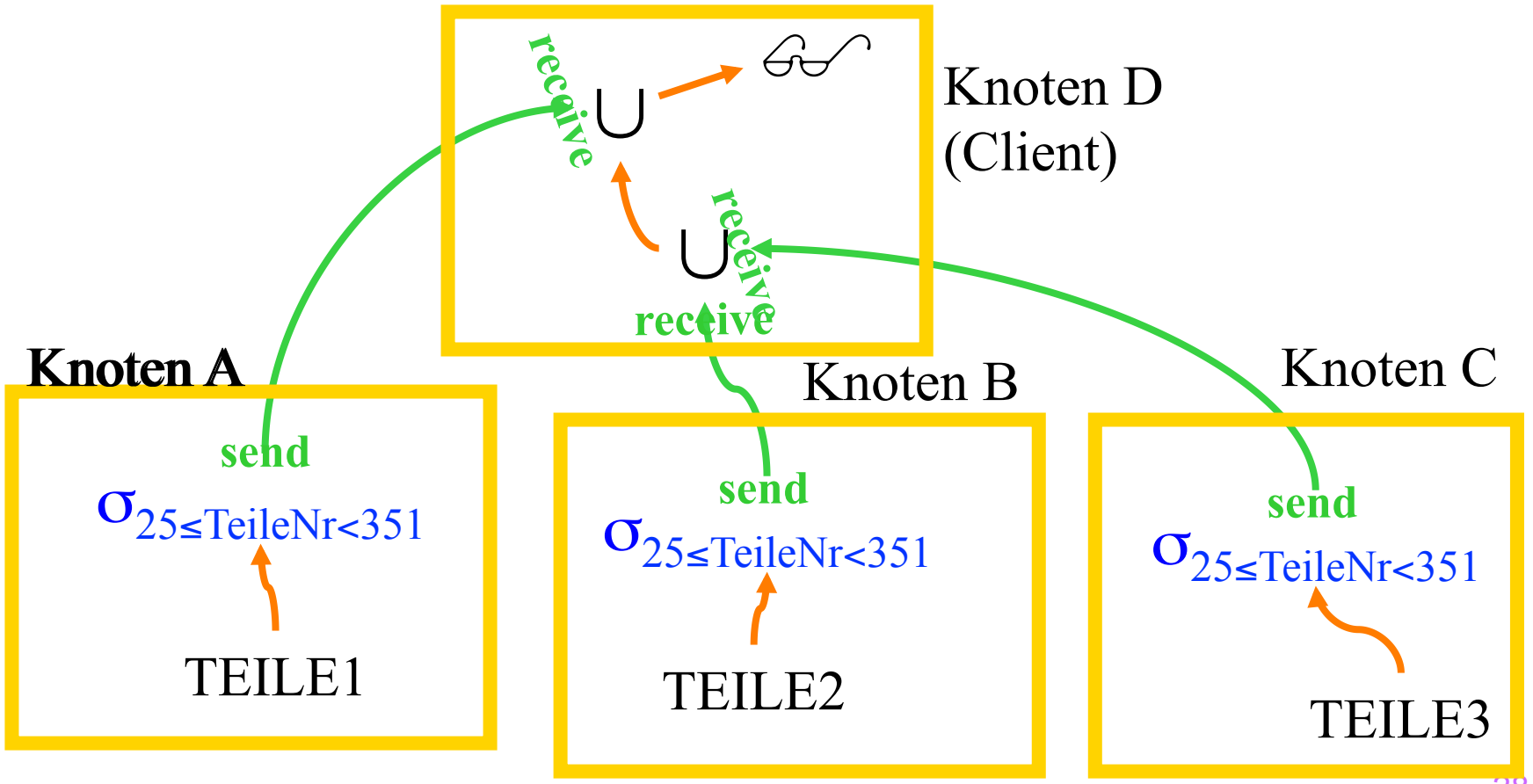
# Operator-Baum: „Pushing Selections“



# Operator-Baum: inklusive Knoten-Annotation

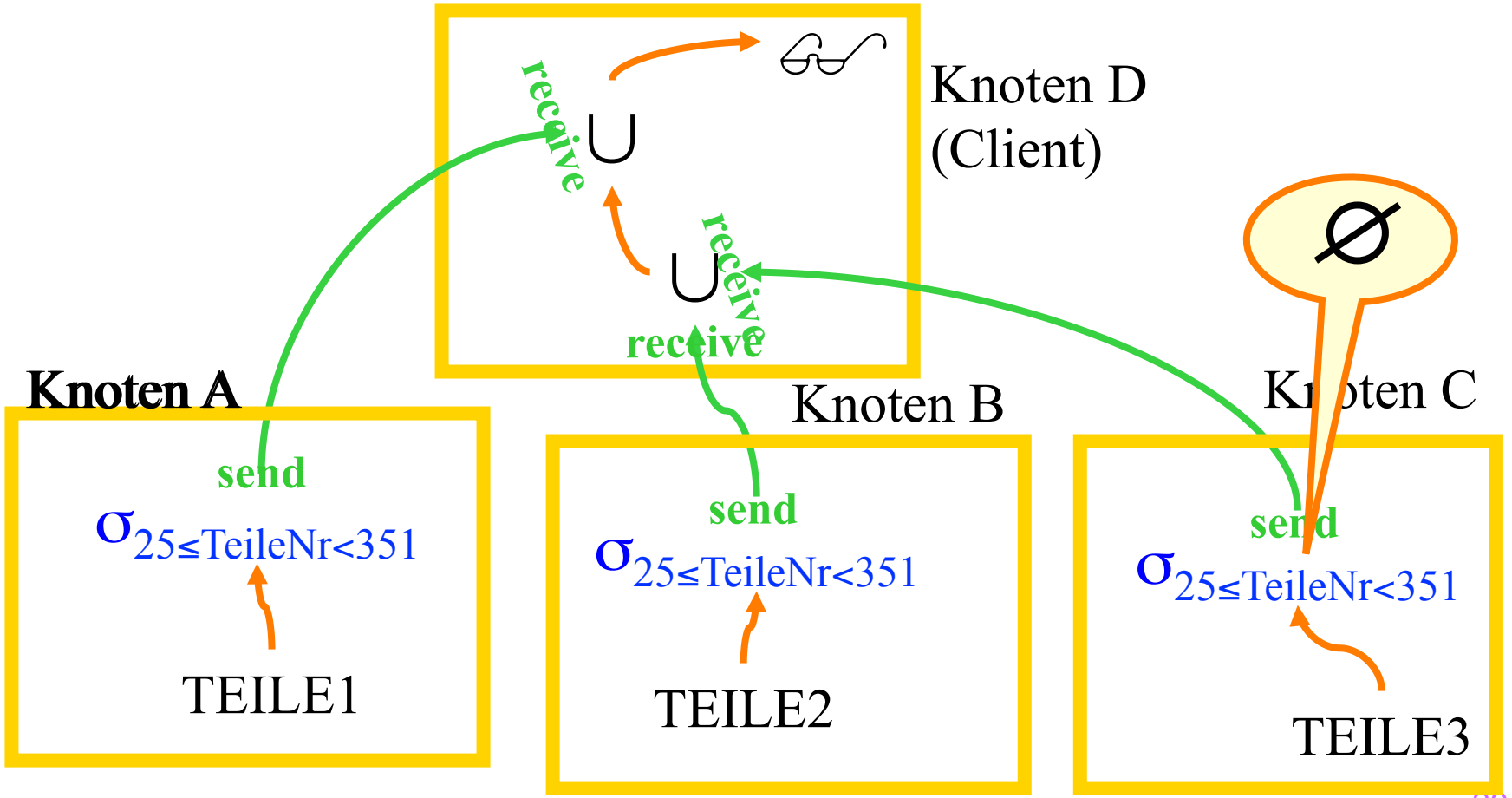


# Operator-Baum: inklusive Knoten-Annotation



# Erkennung und Entfernung überflüssiger Teilausdrücke

$TEILE3 := \sigma_{500 \leq TeileNr<999}\ TEILE$



## Formalisierung: Qualifizierungsprädikate

$(R : q_R)$ Relation R, Qualifizierungsprädikat $q_R$

Erweiterung der Relationenalgebra

$$\sigma_F(R : q_R) \Rightarrow (\sigma_F(R) : q_R \wedge F)$$

$$\Pi_{Attr}(R : q_R) \Rightarrow (\Pi_{Attr} R : q_R)$$

$$(R : q_R) \times (S : q_S) \Rightarrow (R \times S : q_R \wedge q_S)$$

$$(R : q_R) - (S : q_S) \Rightarrow (R - S : q_R)$$

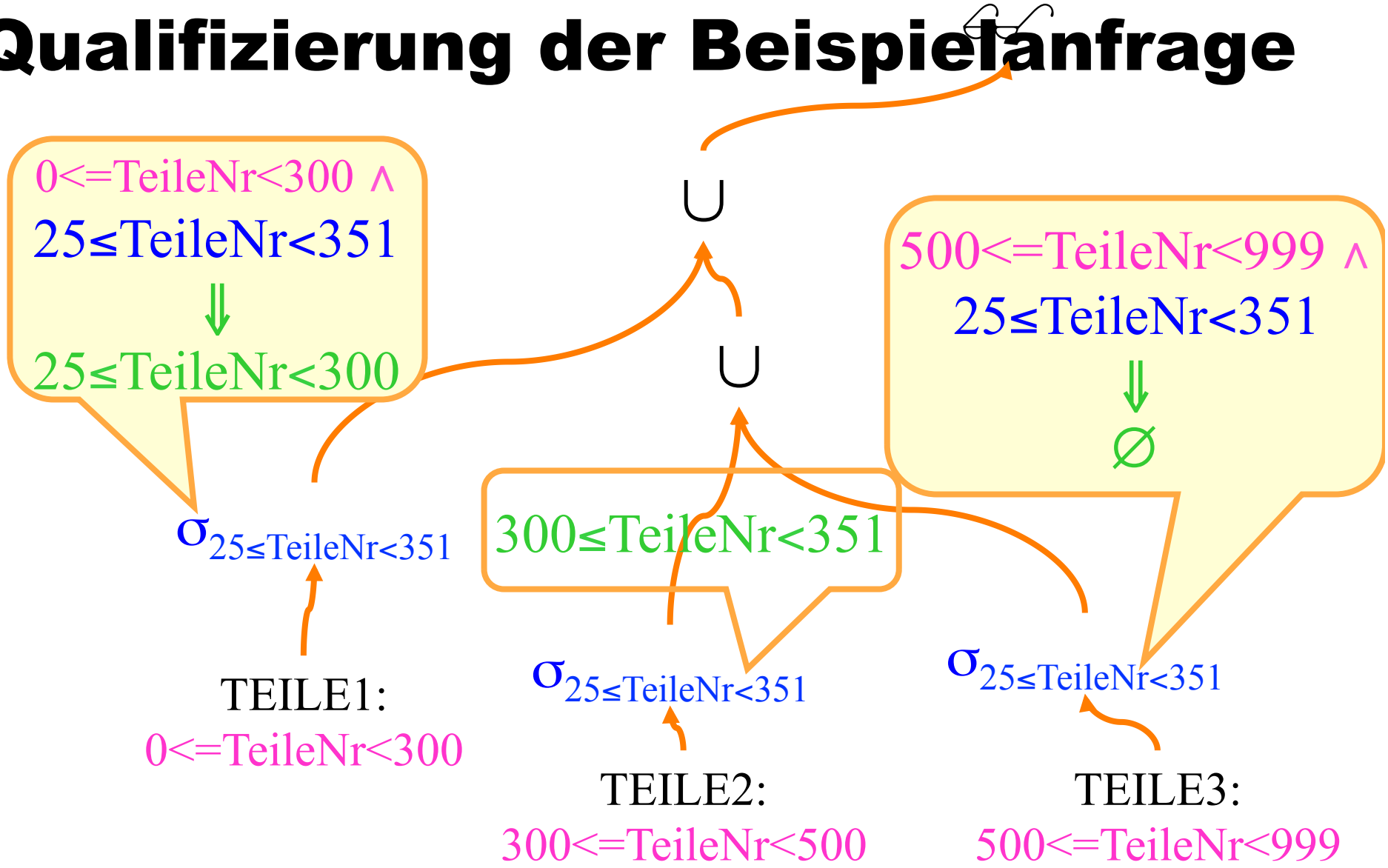
$$(R : q_R) \cup (S : q_S) \Rightarrow (R \cup S : q_R \vee q_S)$$

$$(R : q_R) \cap (S : q_S) \Rightarrow (R \cap S : q_R \wedge q_S)$$

$$(R : q_R) \otimes_F (S : q_S) \Rightarrow (R \otimes_F S : q_R \wedge q_S \wedge F)$$

$$(R : q_R) \otimes_F^{sj} (S : q_S) \Rightarrow \left(R \otimes_F^{sj} S : q_R \wedge q_S \wedge F\right)$$

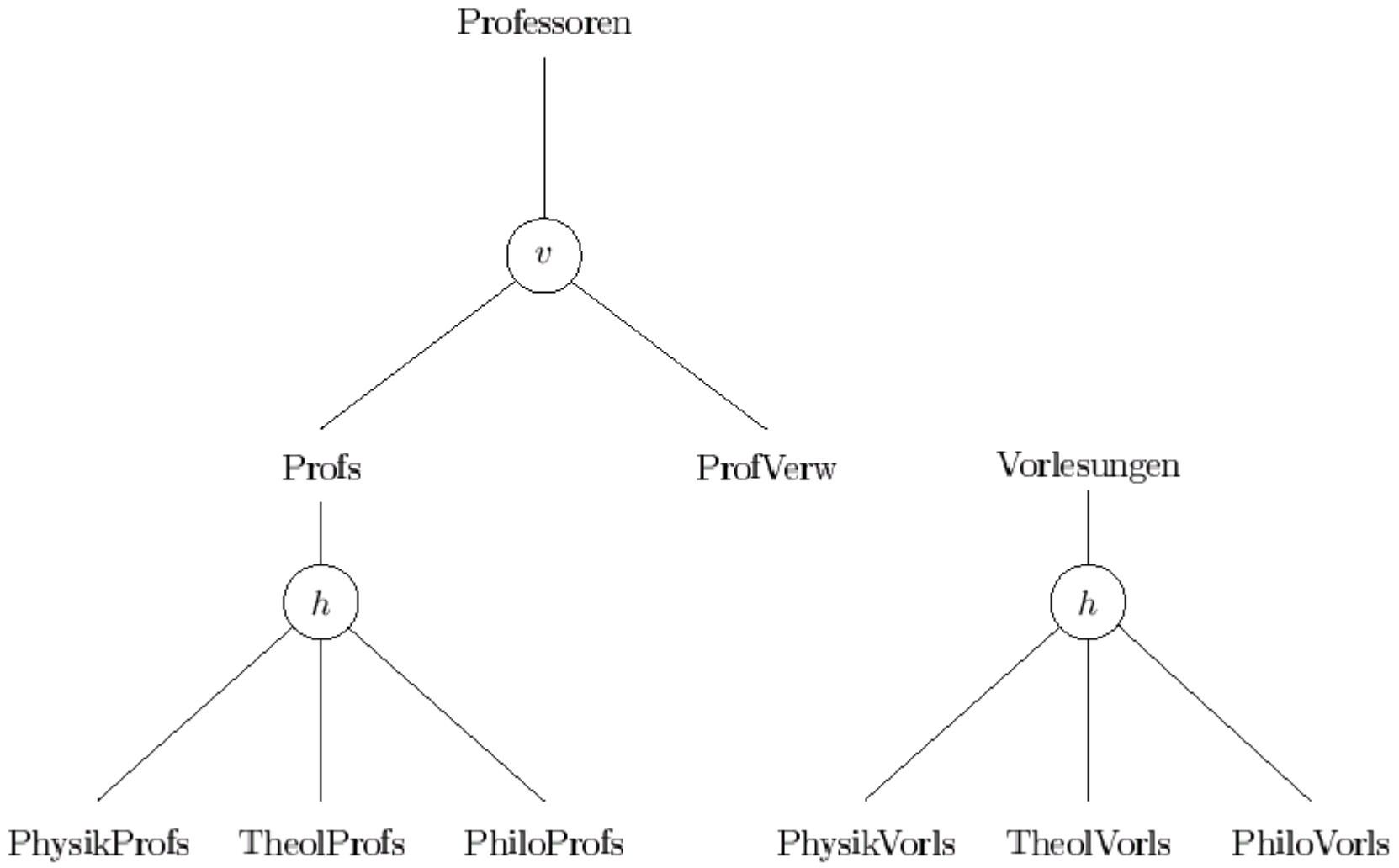
# Qualifizierung der Beispielanfrage



# Beispielausprägung der Relation *Professoren*

| Professoren | | | | | | |
|---|---|---|---|---|---|---|
| PersNr | Name | Rang | Raum | Fakultät | Gehalt | Steuerklasse |
| 2125 | Sokrates | C4 | 226 | Philosophie | 85000 | 1 |
| 2126 | Russel | C4 | 232 | Philosophie | 80000 | 3 |
| 2127 | Kopernikus | C3 | 310 | Physik | 65000 | 5 |
| 2133 | Popper | C3 | 52 | Philosophie | 68000 | 1 |
| 2134 | Augustinus | C3 | 309 | Theologie | 55000 | 5 |
| 2136 | Curie | C4 | 36 | Physik | 95000 | 3 |
| 2137 | Kant | C4 | 7 | Philosophie | 98000 | 1 |

# Baumdarstellung der Fragmentierungen unserer Beispielanwendung



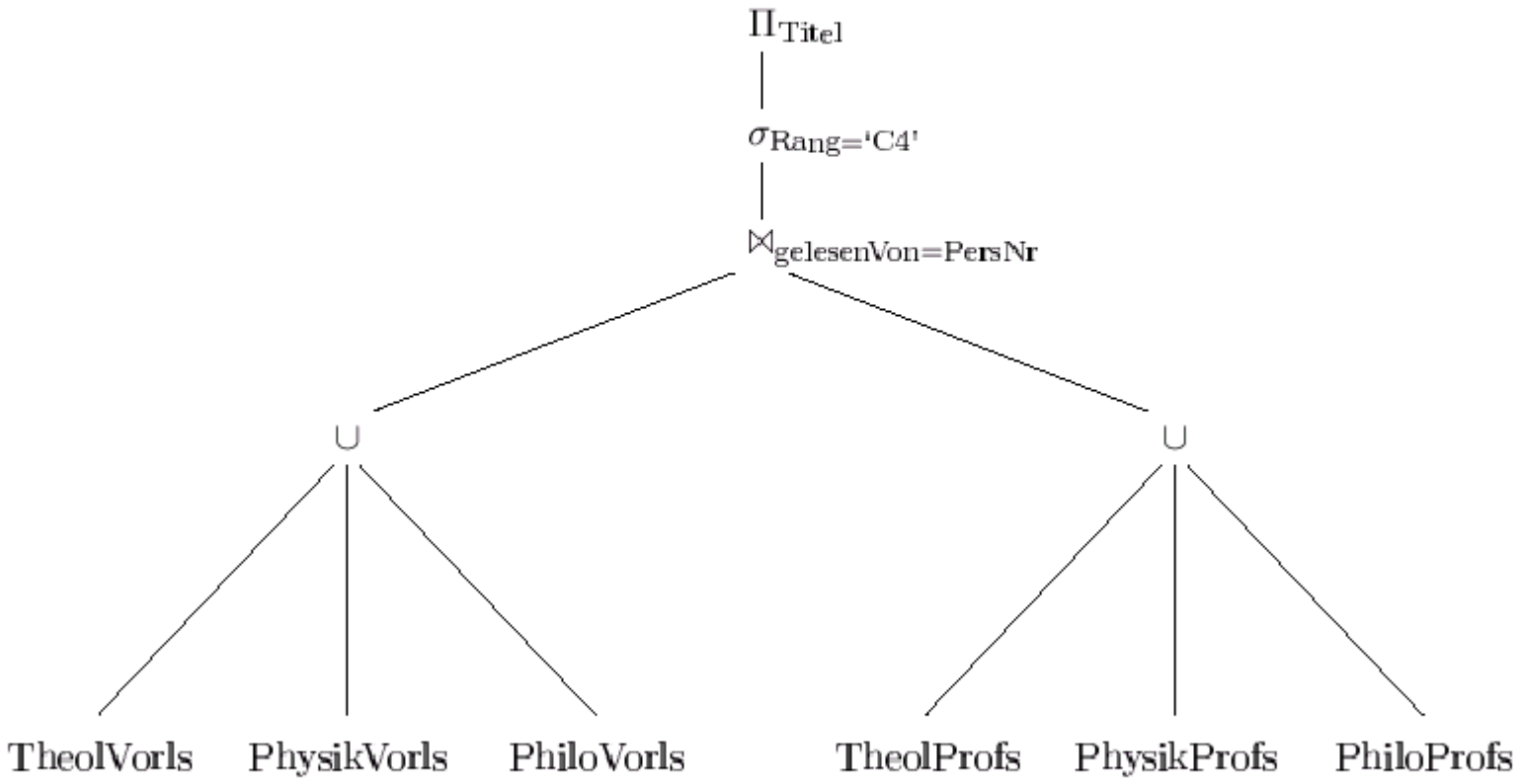
# Anfragebearbeitung bei horizontaler Fragmentierung

**select** Titel
**from** Vorlesungen, Profs
**where** gelesenVon = PersNr **and**
Rang = 'C4';

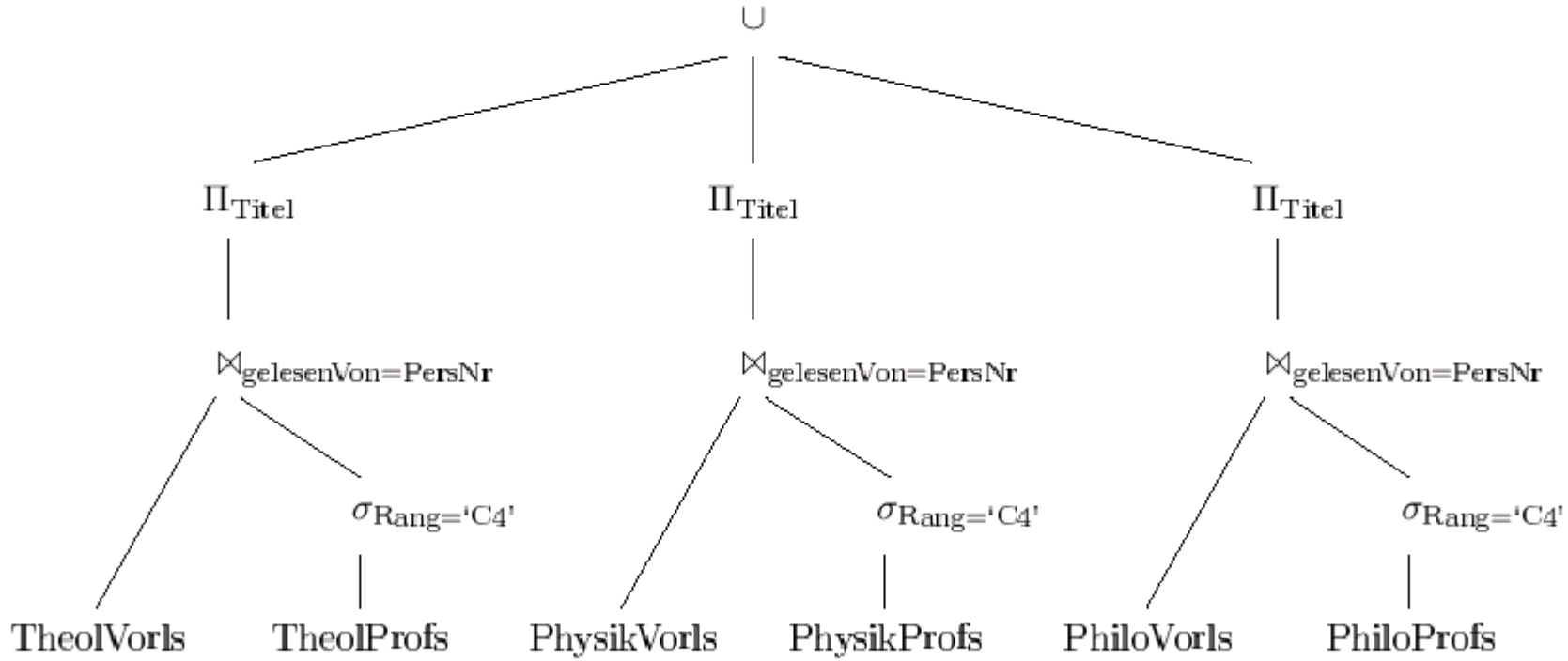
1. Rekonstruiere alle in der Anfrage vorkommenden globalen Relationen aus den Fragmenten, in die sie während der Fragmentierungsphase zerlegt wurden. Hierfür erhält man einen algebraischen Ausdruck.

2. Kombiniere den Rekonstruktionsausdruck mit dem algebraischen Anfrageausdruck, der sich aus der Übersetzung der SQL-Anfrage ergibt.

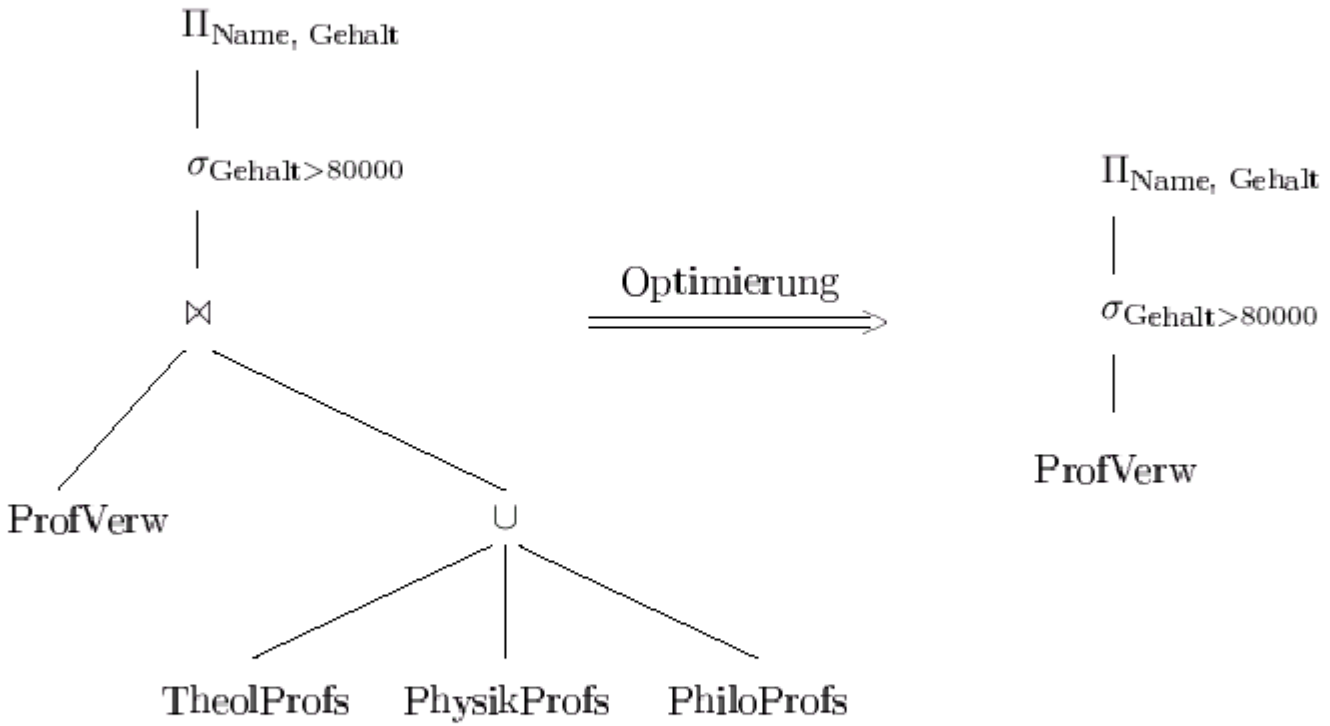
# Kanonische Form der Anfrage

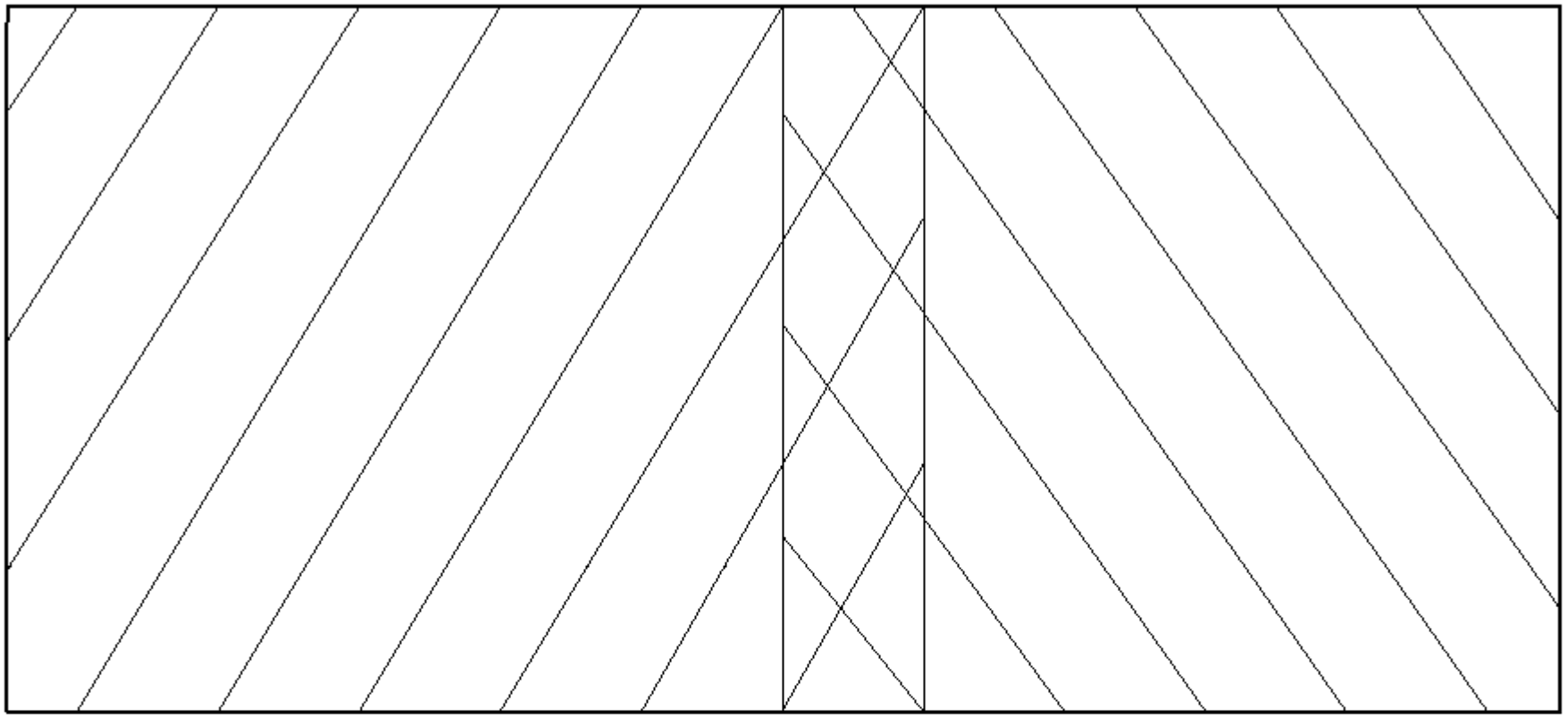


# Optimale Form der Anfrage



# Optimierung bei vertikaler Fragmentierung





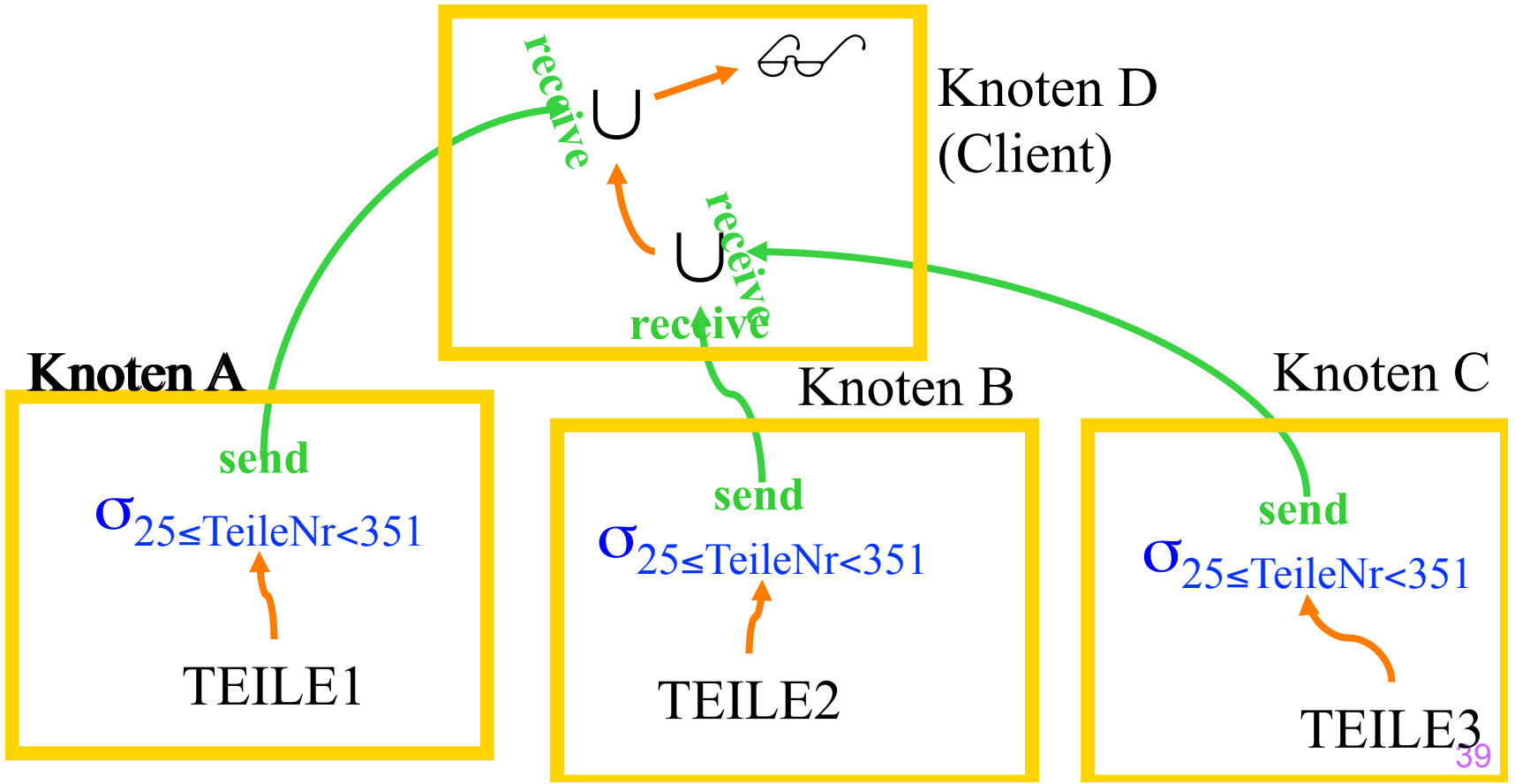

$$\text{ProfVerw} := \Pi_{\textbf{PersNr, Name, Gehalt, Steuerklasse}}(\text{Professoren})$$

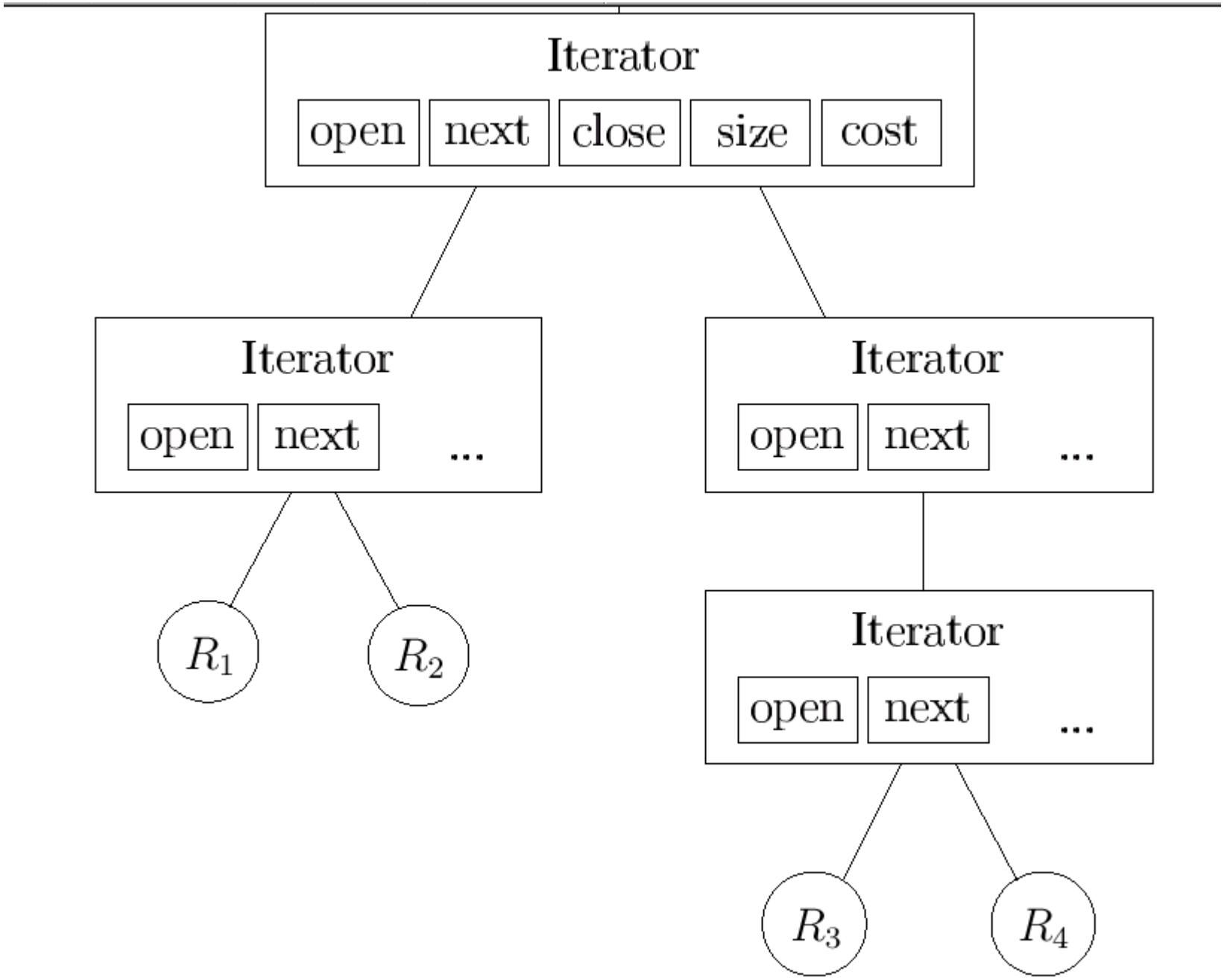
$$\text{Profs} := \Pi_{\textbf{PersNr, Name, Rang, Raum, Fakultät}}(\text{Professoren})$$

$$\text{Professoren} = \text{ProfVerw} \bowtie_{\textbf{ProfVerw.PersNr=Profs.PersNr}} \text{Profs}$$

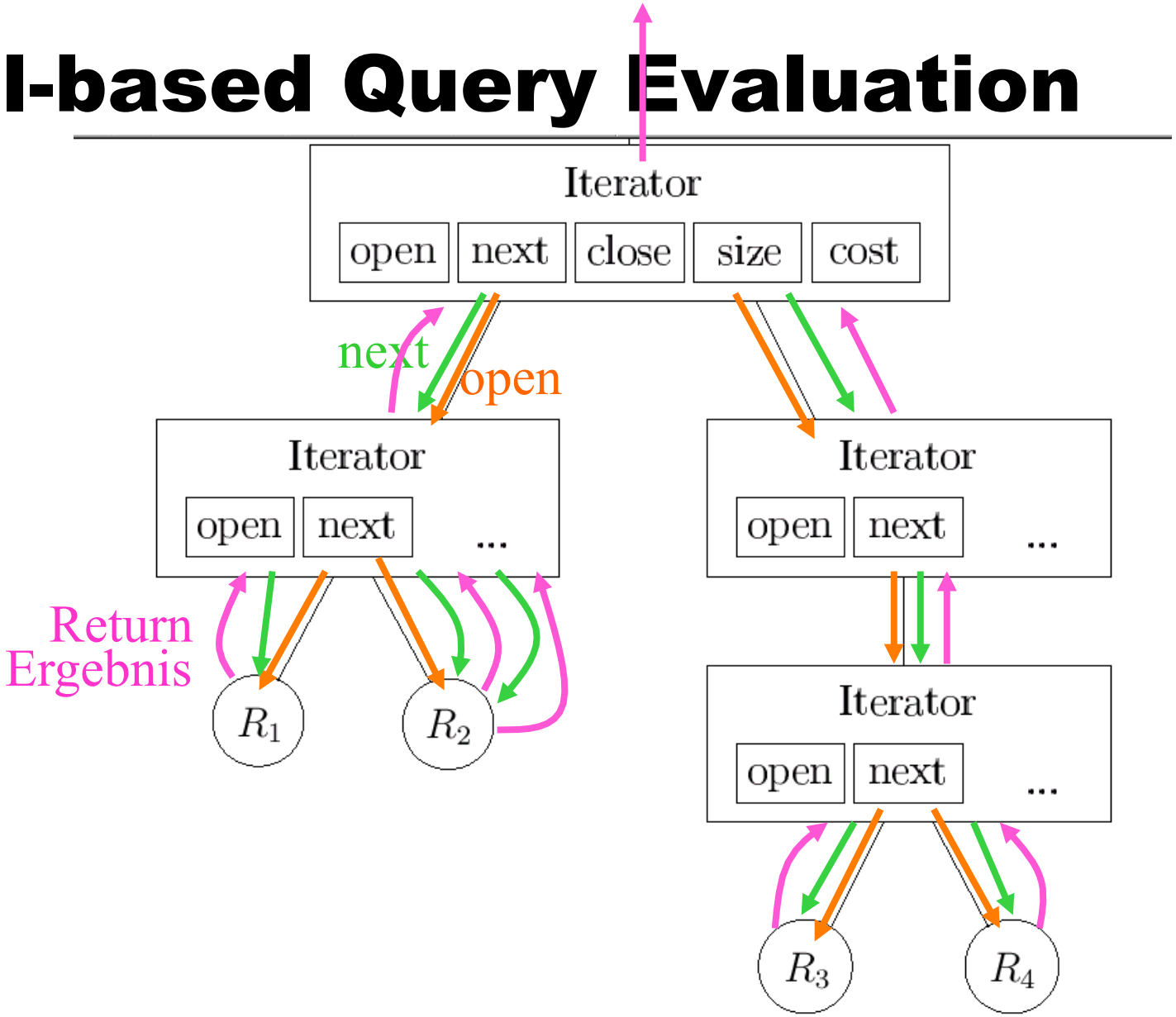
# Parallelausführung einer verteilten Anfrage

- Voraussetzung: Asynchrone Kommunikation
  - send/receive-Operatoren mit entsprechend großem Puffer

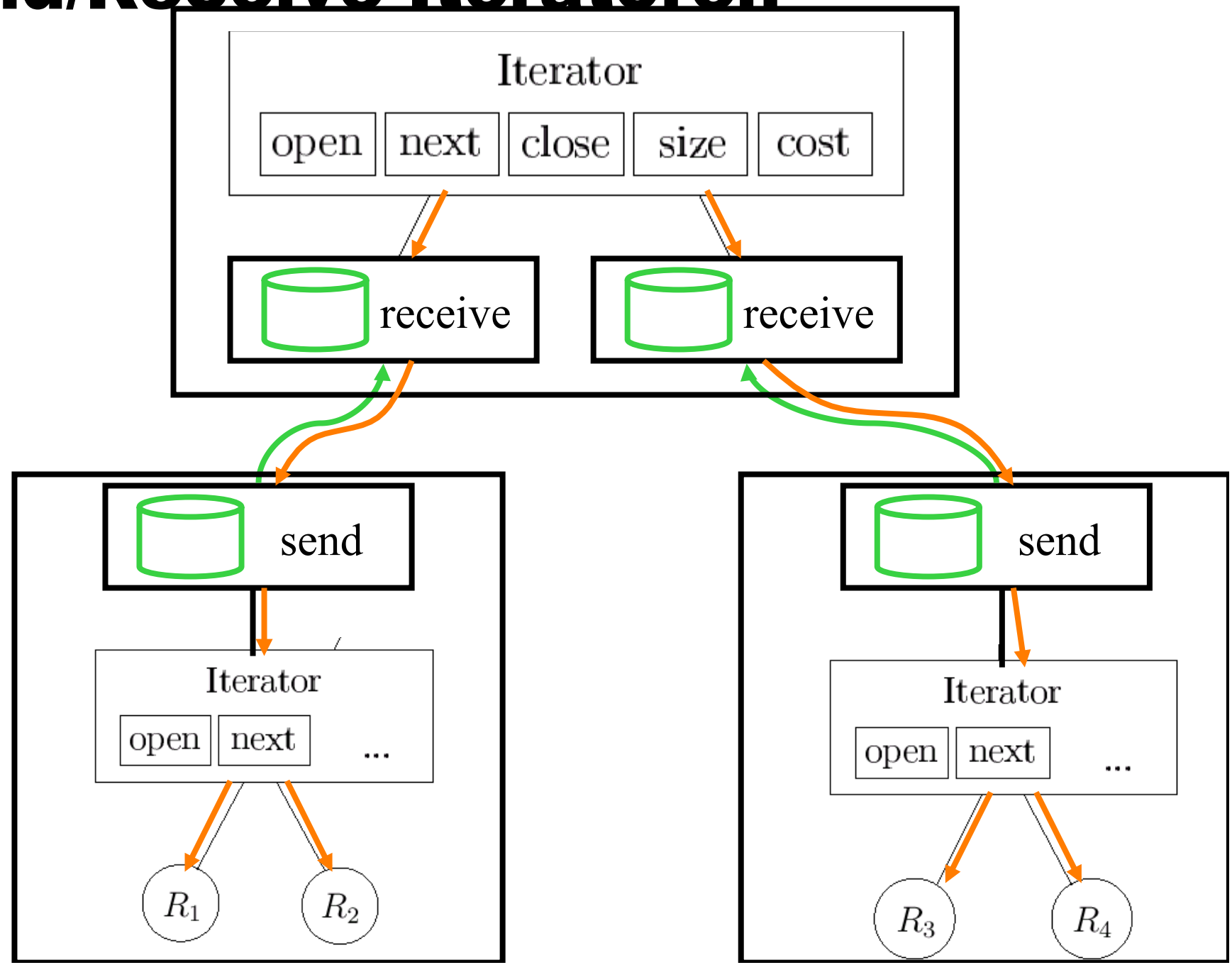



Iterator
open
next
close
size
cost
Iterator
open
next
...
Iterator
open
next
...
$R_1$
$R_2$
Iterator
open
next
...
$R_3$
$R_4$

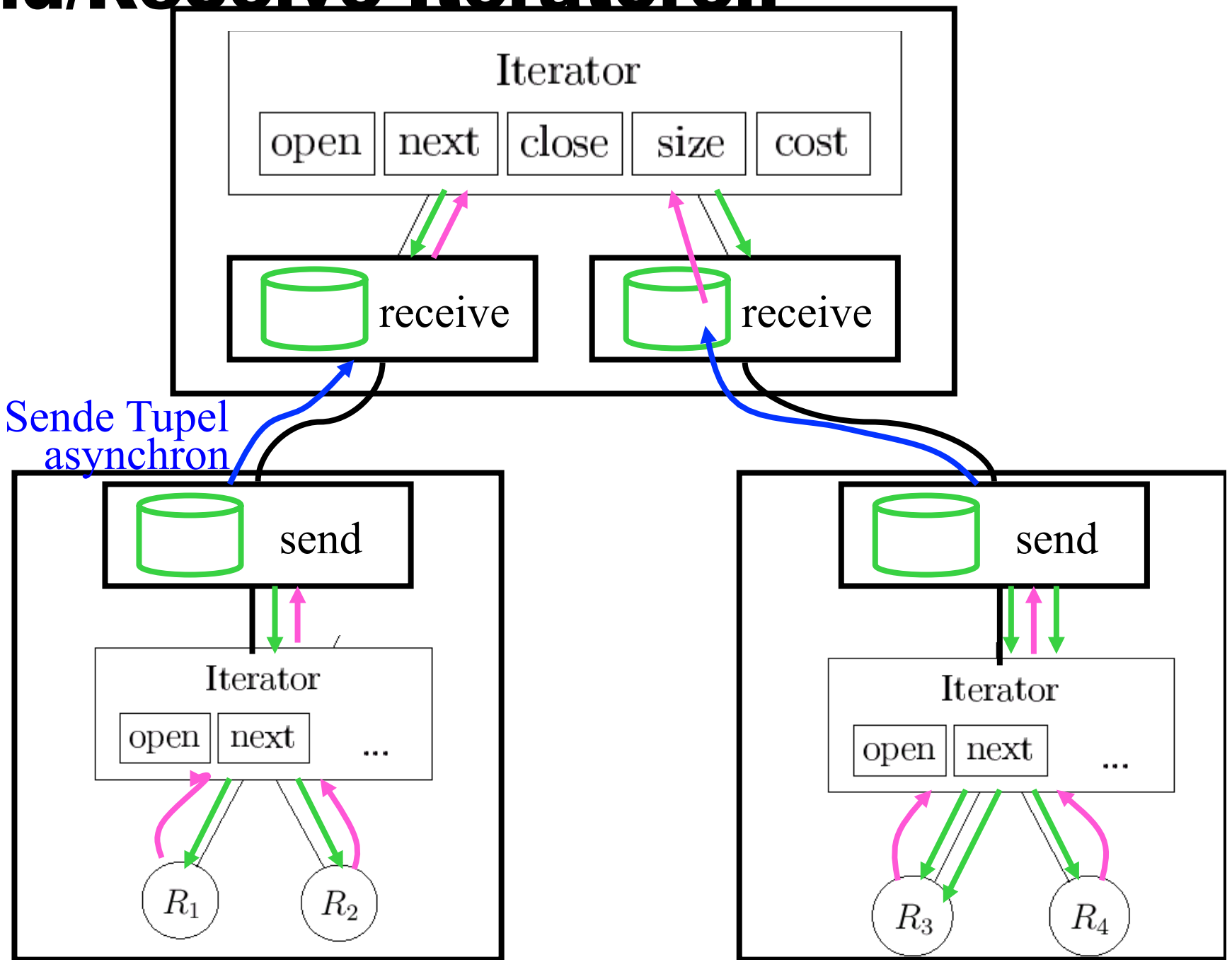
# Pull-based Query Evaluation



# Send/Receive-Iteratoren

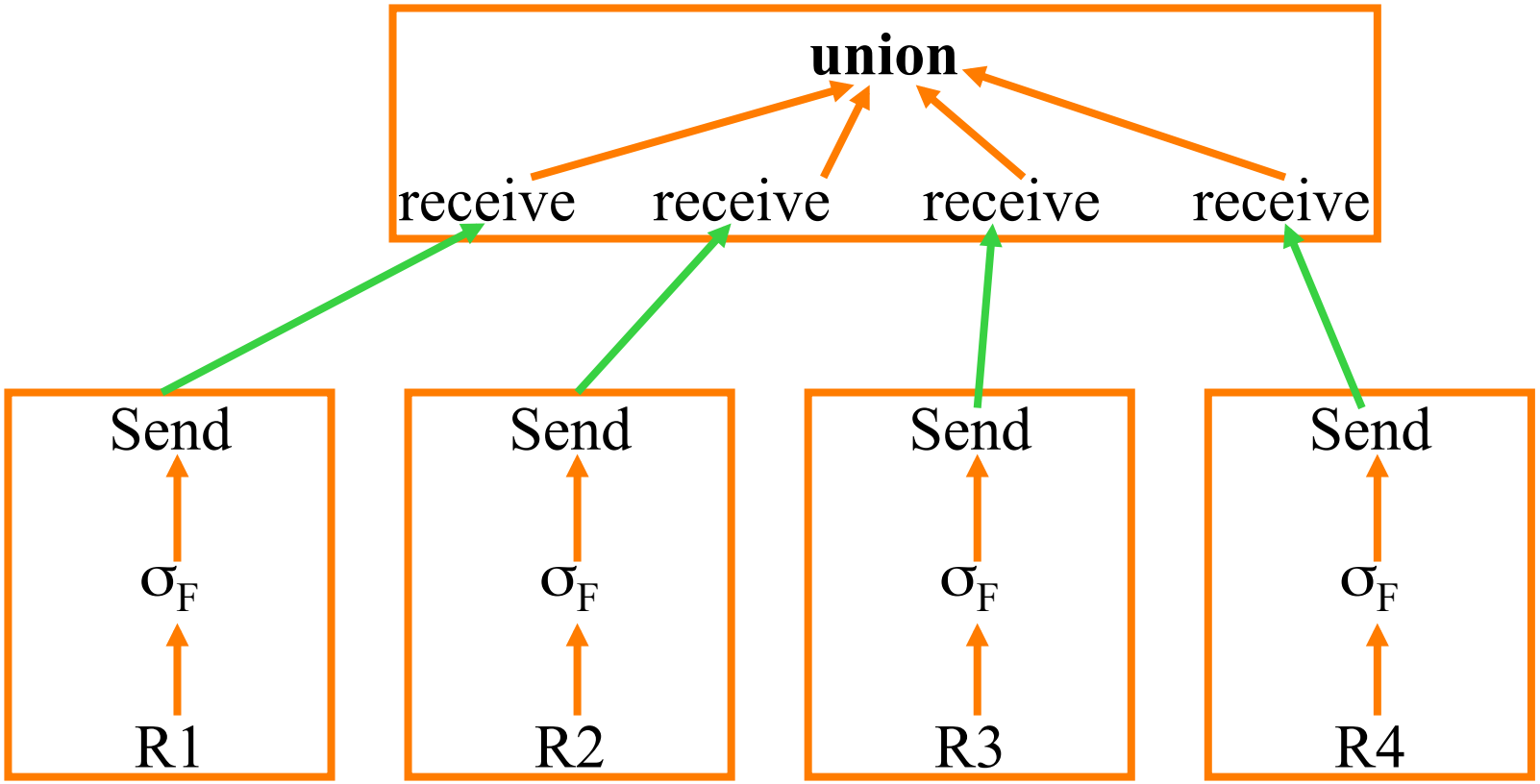


# Send/Receive-Iteratoren



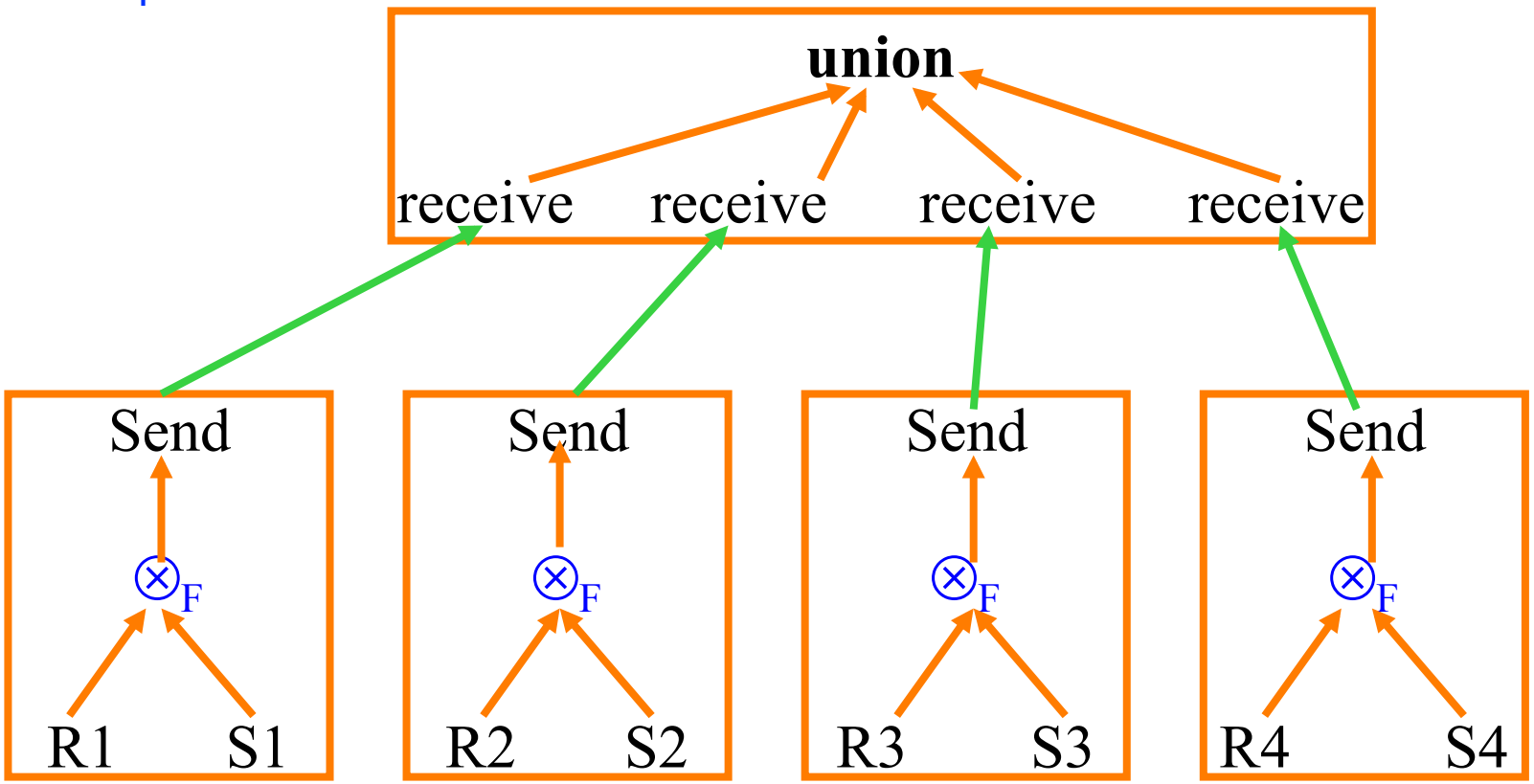
# Parallelausführung bei horizontaler Partitionierung

- $R = R1 \cup R2 \cup R3 \cup R4$
- $\sigma_F R = \sigma_F R1 \cup \sigma_F R2 \cup \sigma_F R3 \cup \sigma_F R4$



# Parallelausführung bei abgeleiteter horizontaler Partitionierung

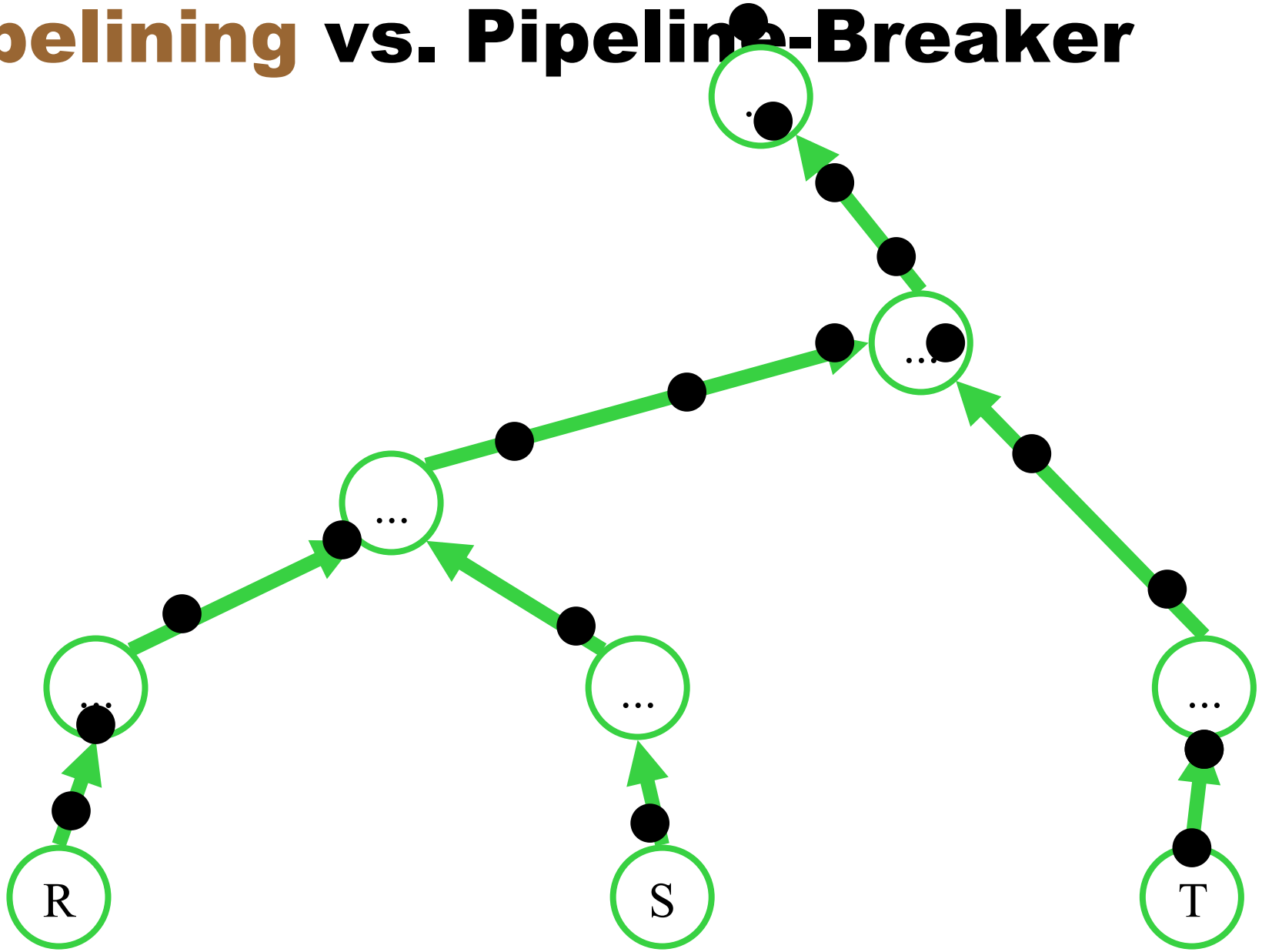
- R = R1∪R2∪R3∪R4; S = S1∪S2∪S3∪S4
- Si = S $\otimes^{lsj}_{F}$ Ri
- R $\otimes_{F}$ S



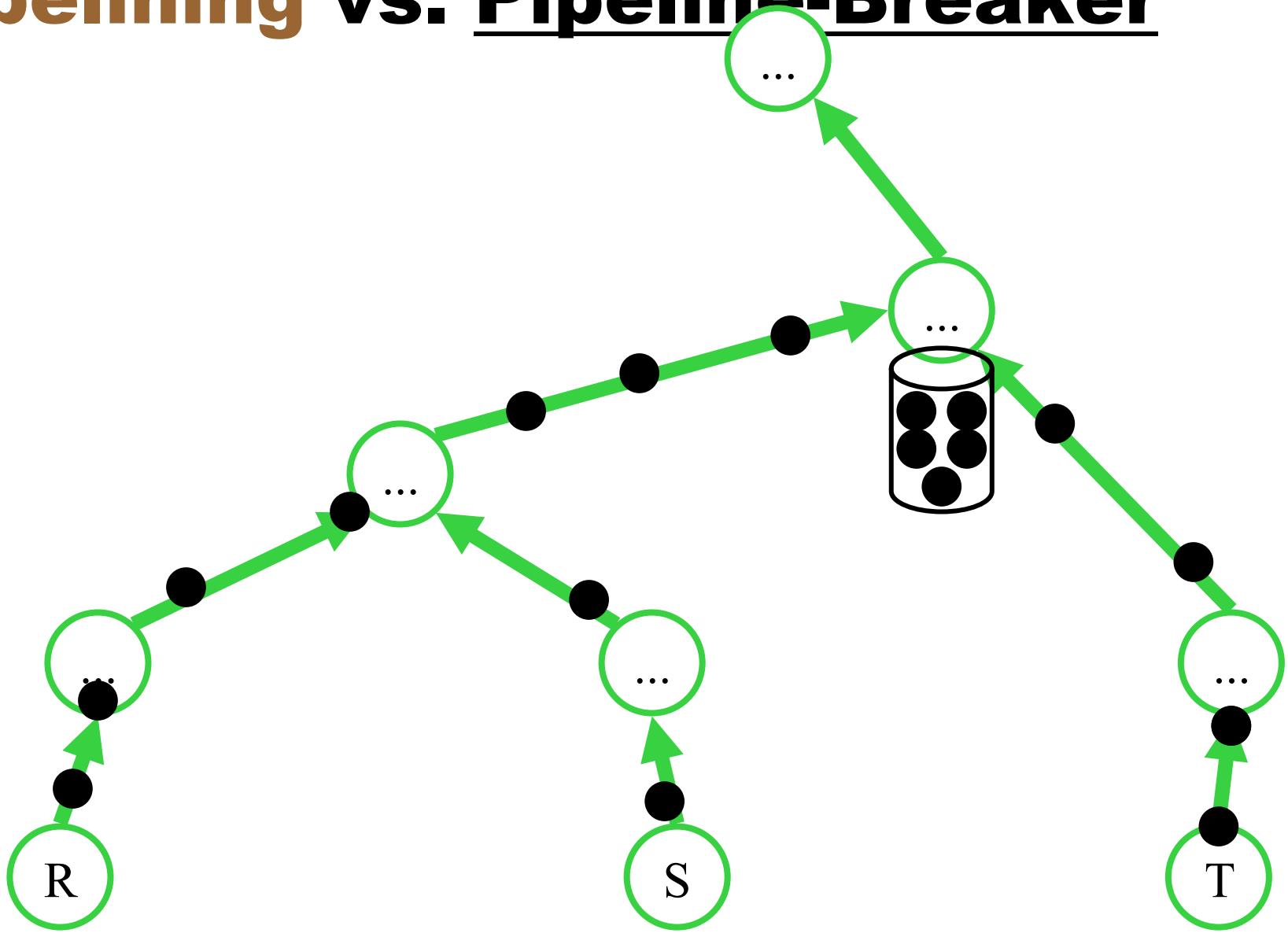
# Prallelausführung von Aggregat-Operationen

- **Min:** Min(R.A) = Min ( Min(R1.A), ... , Min(Rn.A) )
- **Max:** analog
- **Sum:** Sum(R.A) = Sum ( Sum(R1.a), ..., Sum(Rn.A) )
- **Count:** analog
- **Avg:** man muß die Summe und die Kardinalitäten der Teilrelationen kennen; aber vorsicht bei Null-Werten!
- Avg(R.A) = Sum(R.A) / Count(R) gilt nur wenn A keine Nullwerte enthält.

# Pipelining vs. Pipeline-Breaker



# Pipelining vs. Pipeline-Breaker



# Pipeline-Breaker

- Unäre Operationen
  - sort
  - Duplikatelimination (unique,distinct)
  - Aggregatoperationen (min,max,sum,...)
- Binäre Operationen
  - Mengendifferenz
- Je nach Implementierung
  - Join
  - Union

# Implementierung der Joinoperation

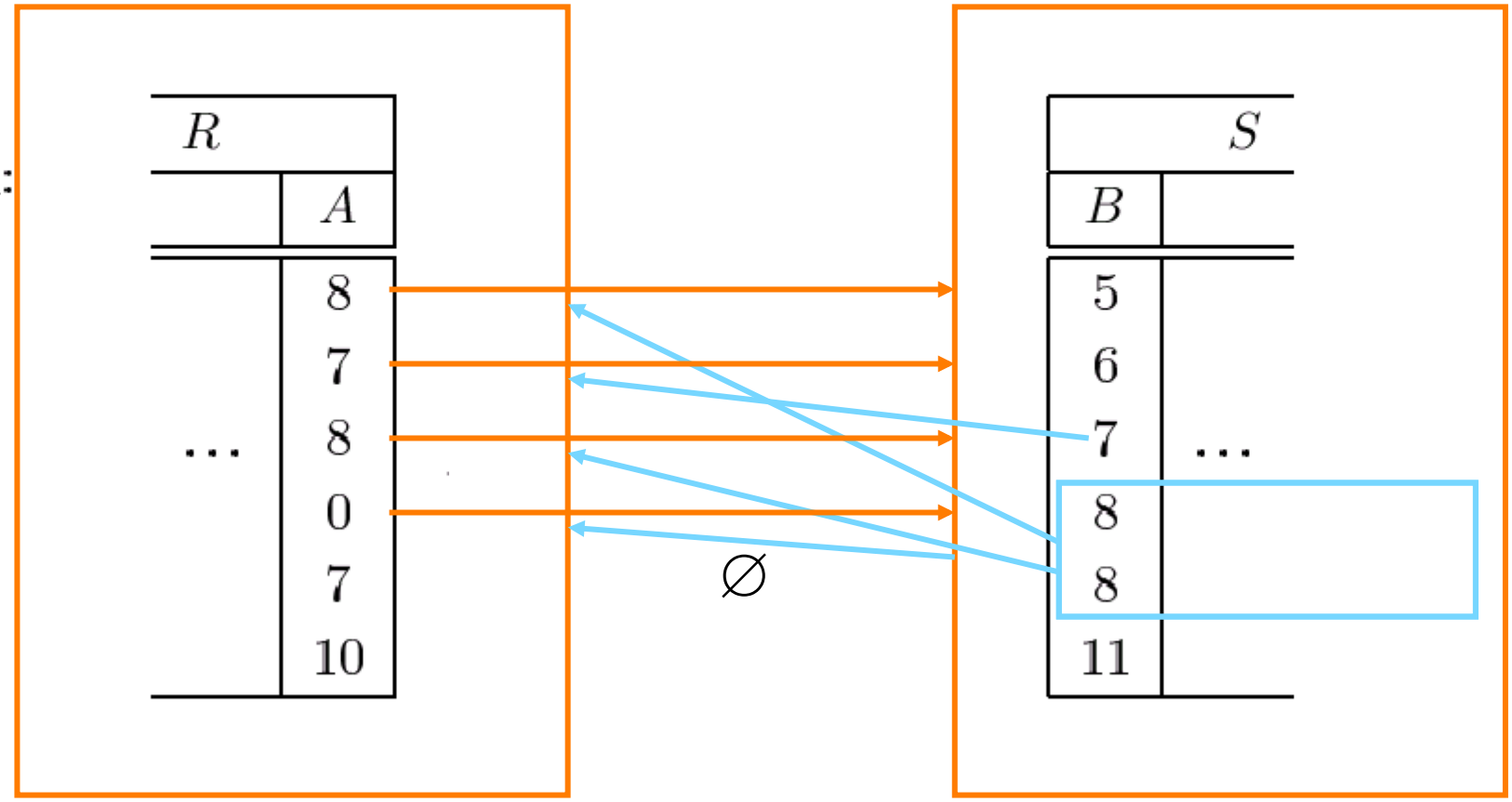
- Mengendifferenz und -durchschnitt können analog zum Join implementiert werden
- hier nur Equi-Joins betrachtet

Nested-Loop-Join:

**for each** $r \in R$
  **for each** $s \in S$
    **if** $r.A = s.B$ **then**
      $res := res \cup (r \times s)$

# Nested Loop Join in Verteilten Datenbanken

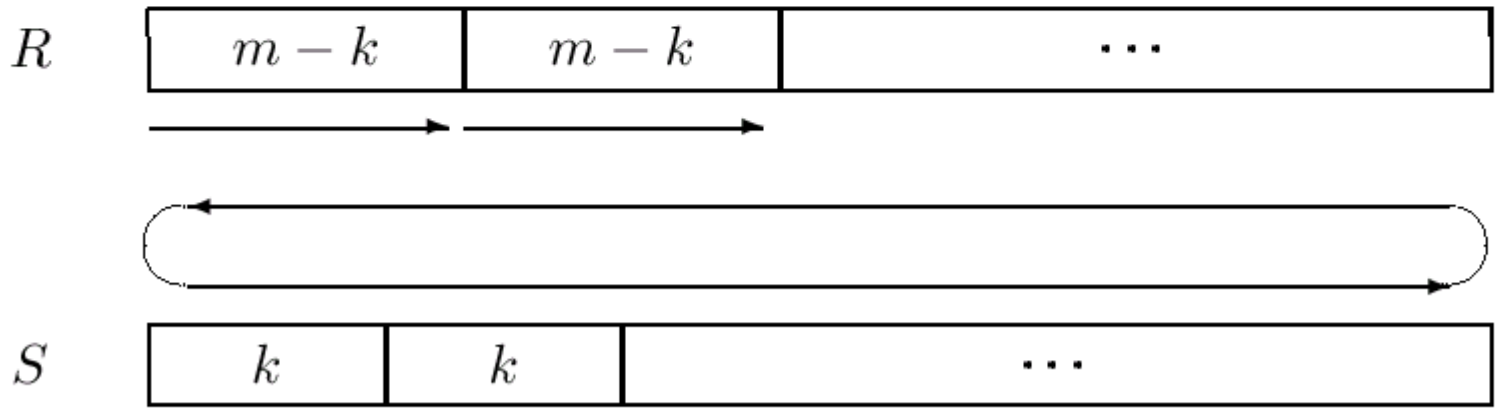
Beispiel:



# Block Nested Loop Join: zentrale Datenbank

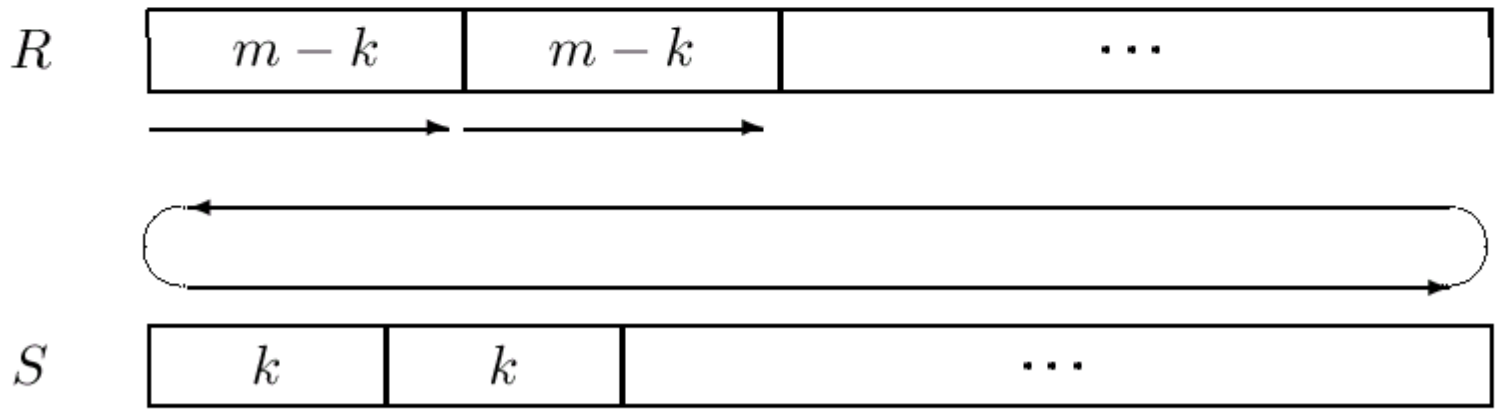
- Relationen sind *seitenweise* abgespeichert
- Es stehen $m$ Pufferrahmen im Hauptspeicher zur Verfügung:
  - $k$ für die innere Schleife des Nested Loop
  - $m - k$ für die äußere

Join von $R$ und $S$:



# Block Nested Loop Join: verteilte Datenbank

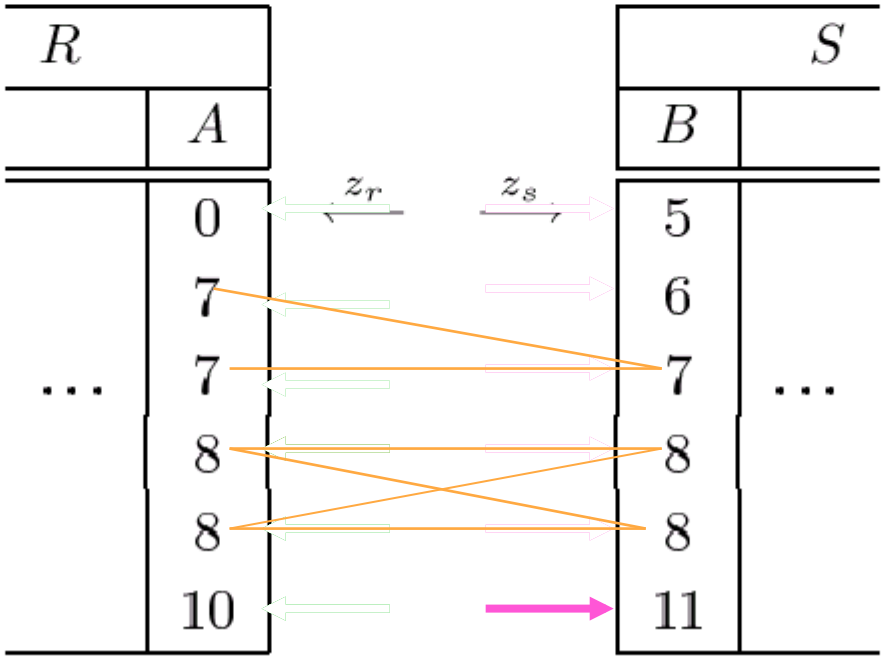
- R wird an die Heimatstation von S geschickt
- Sobald ein „hinreichend" großer Block von R-Tupeln angekommen ist, wird durch S iteriert
- Der Block von R-Tupeln sollte tunlichst in eine Hash-Tabelle geladen werden



# Der Merge-Join

- Voraussetzung: $R$ und $S$ sind sortiert (notfalls vorher sortieren)

Beispiel:

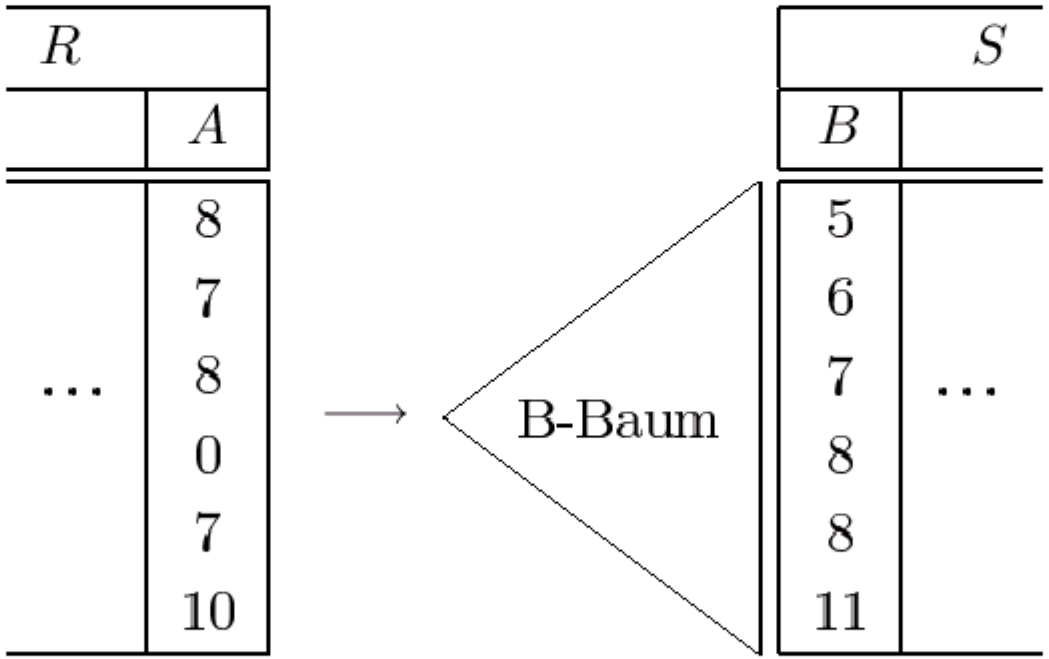


# Merge Join in Verteilten Datenbanken

- R und S sollten möglichst an ihren Heimatknoten sortiert werden
  - in (unkooperativen) Multi-Datenbanken nicht immer möglich
- Noch besser, die Heimatknoten lesen R und S sortiert „von der Platte“ (Pipelining)
  - Cluster-Index
  - Sekundär-Index
- Merge-Join wird dann z.B. dort ausgeführt, wo das Ergebnis gebraucht wird

# Index-Join

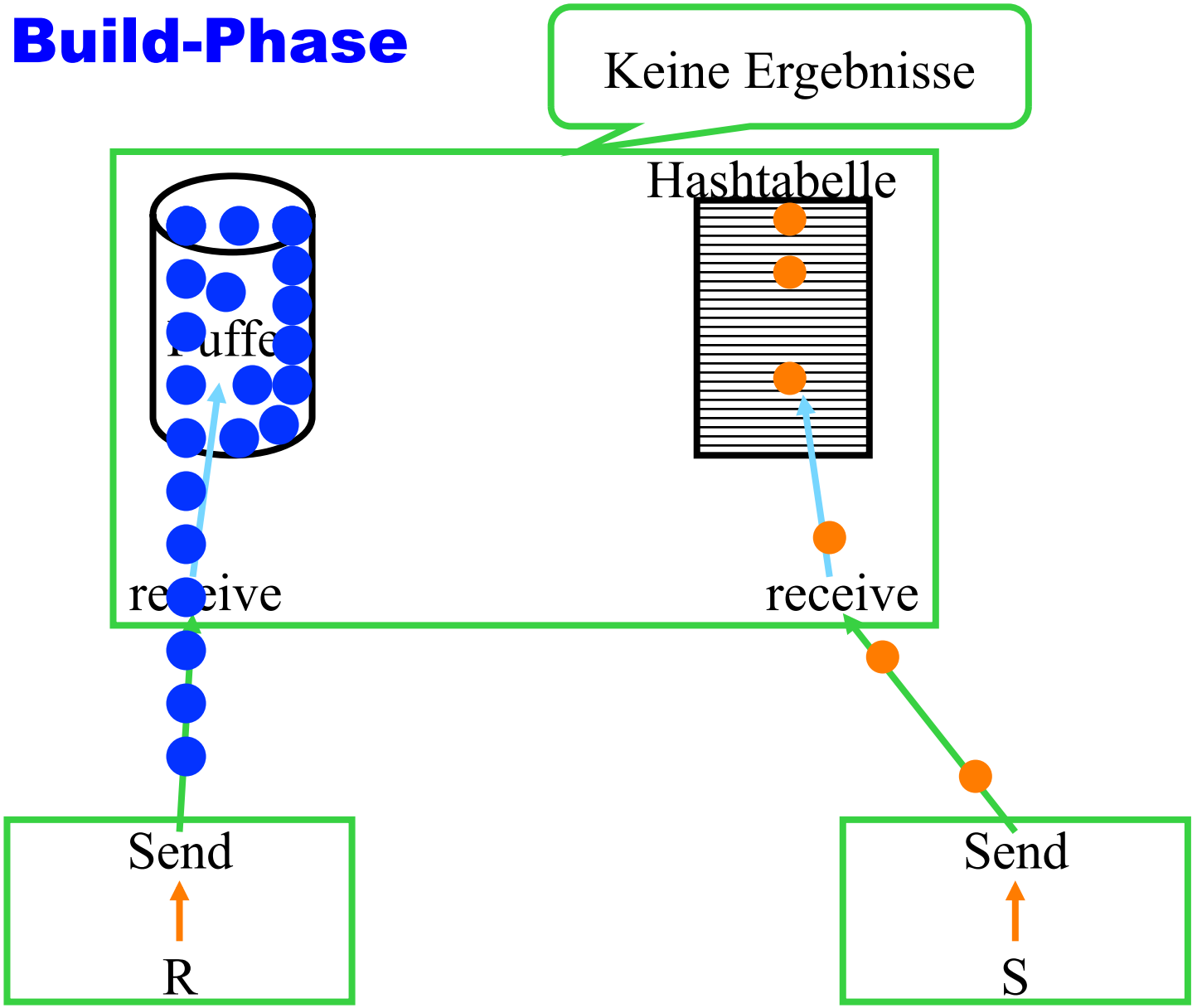
Beispiel:



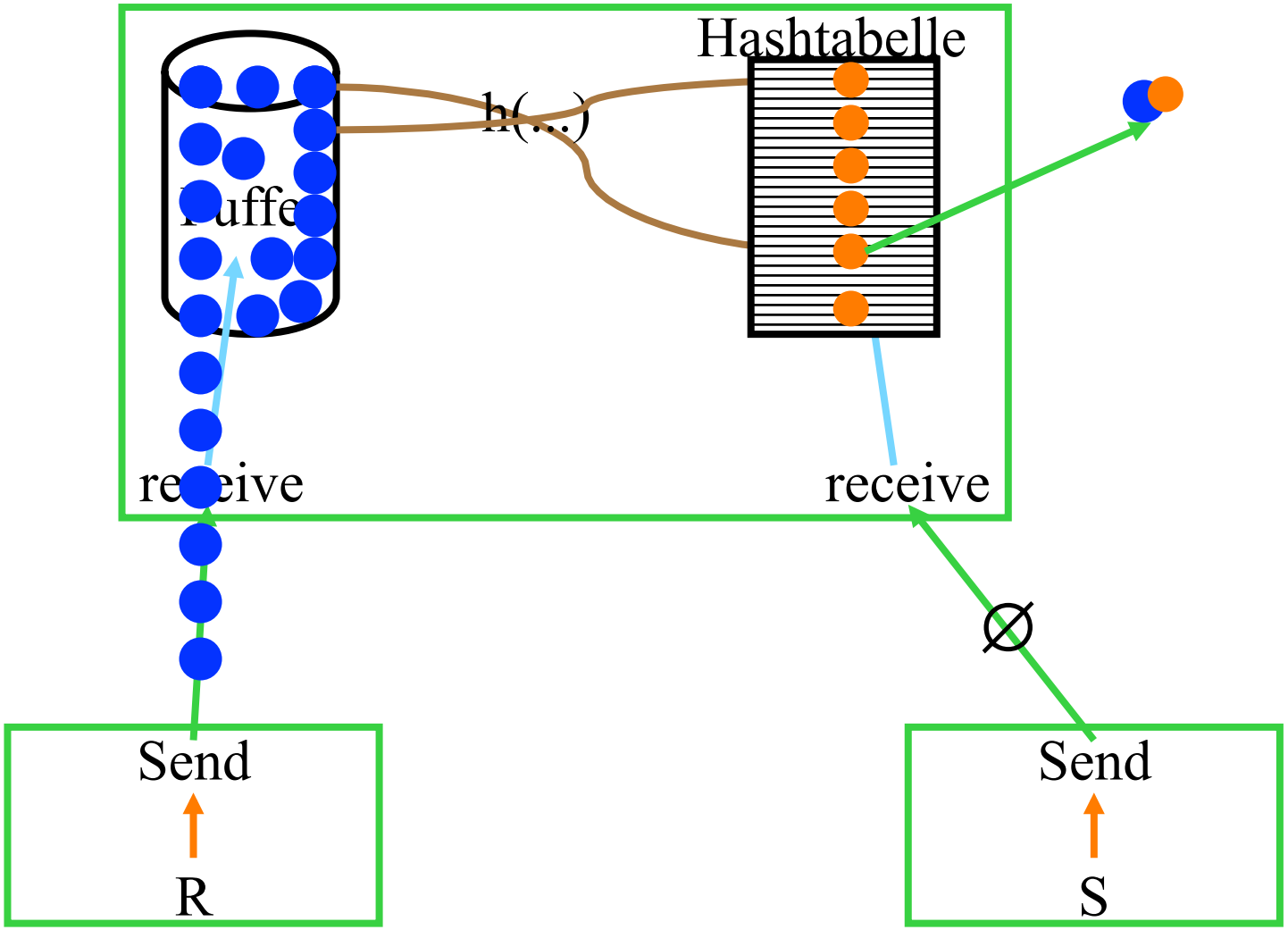
# Index Join in Verteilten Datenbanken

- Sollte dort dort ausgeführt werden, wo der Index liegt
  - hier Heimatstation von S
  - R muss dorthin transferiert werden
- Alternative: einen temporären Index aufbauen
  - Hash Join

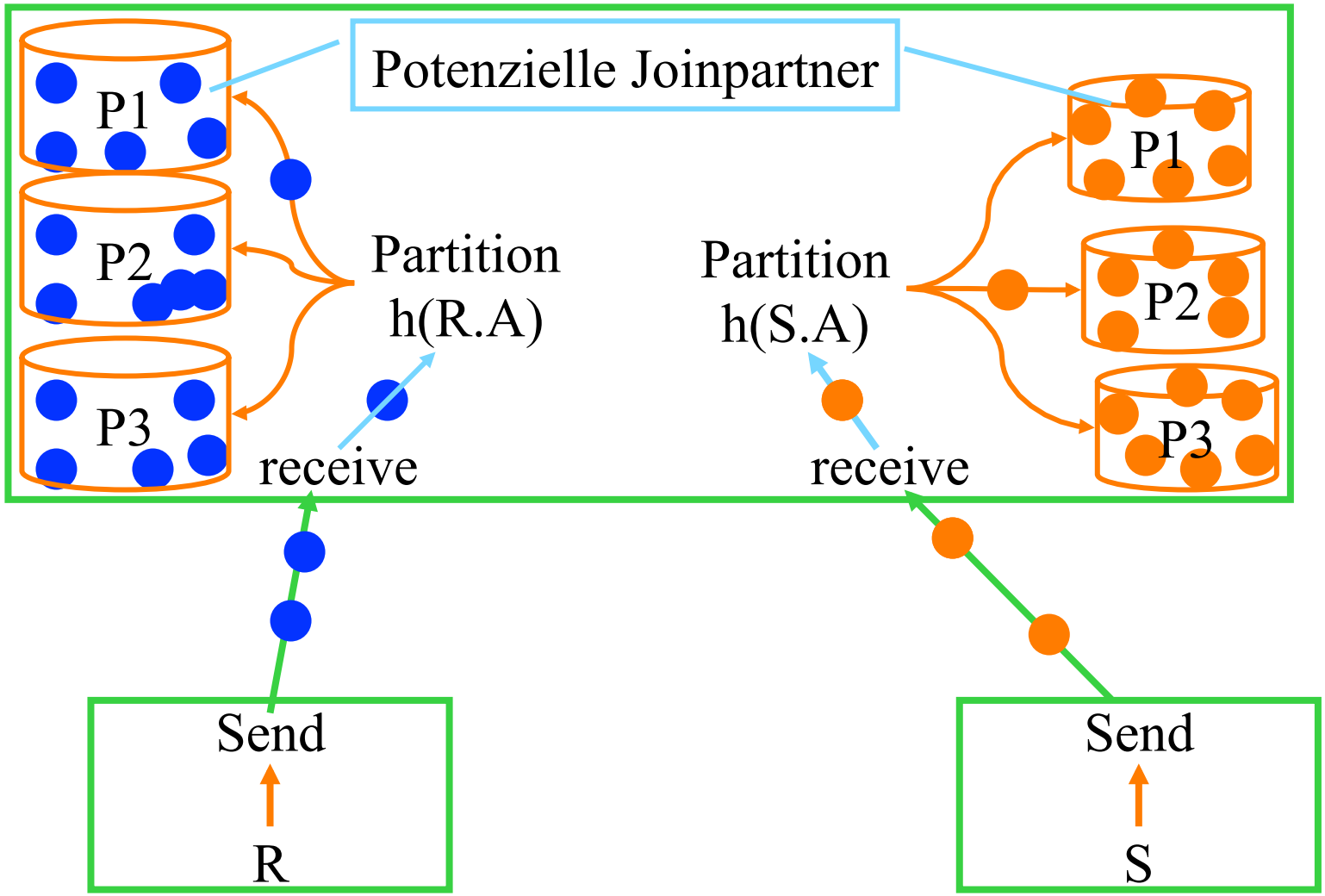
# „Normaler“ blockierender Hash-Join: Build-Phase



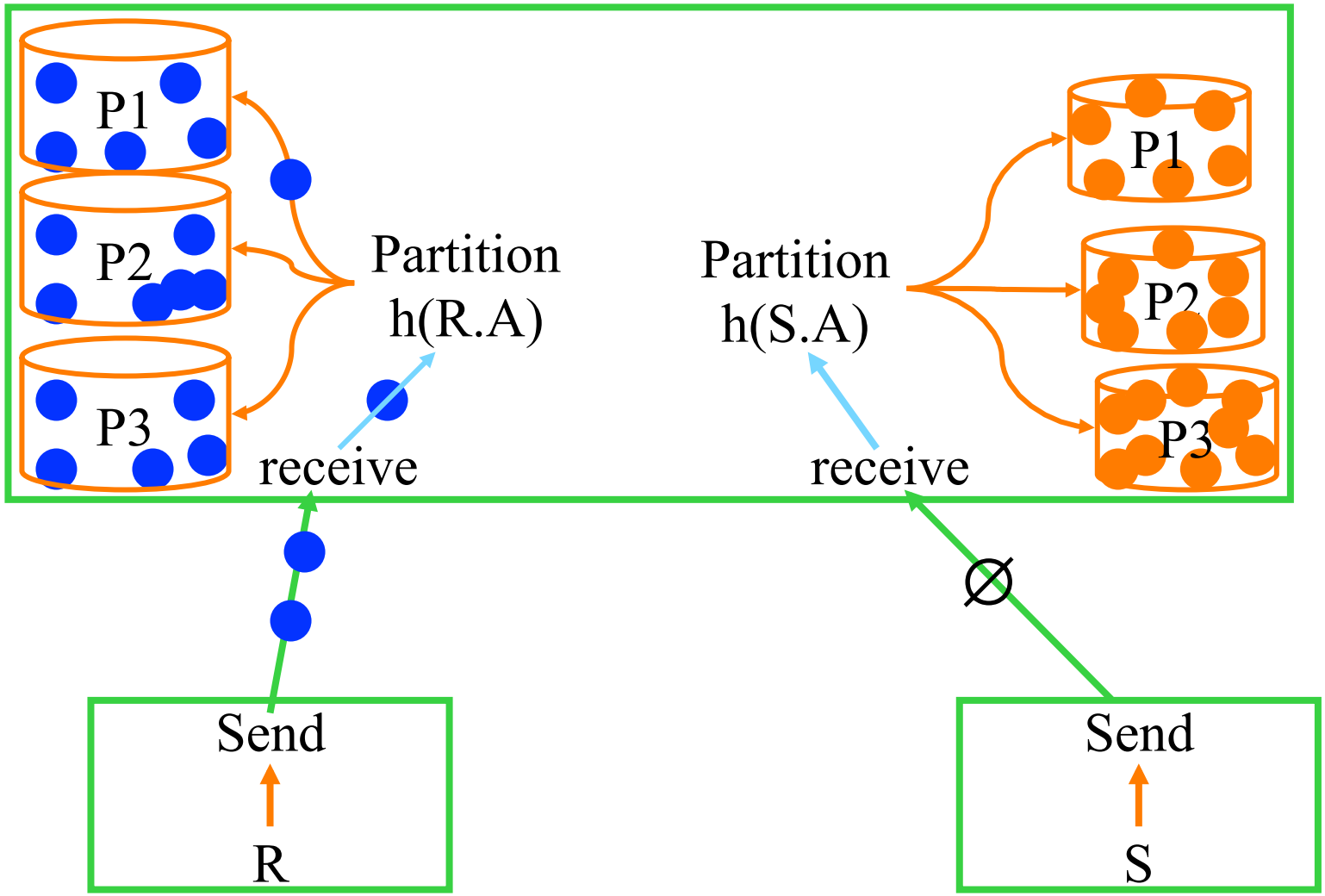
# „Normaler“ blockierender Hash-Join: Probe-Phase



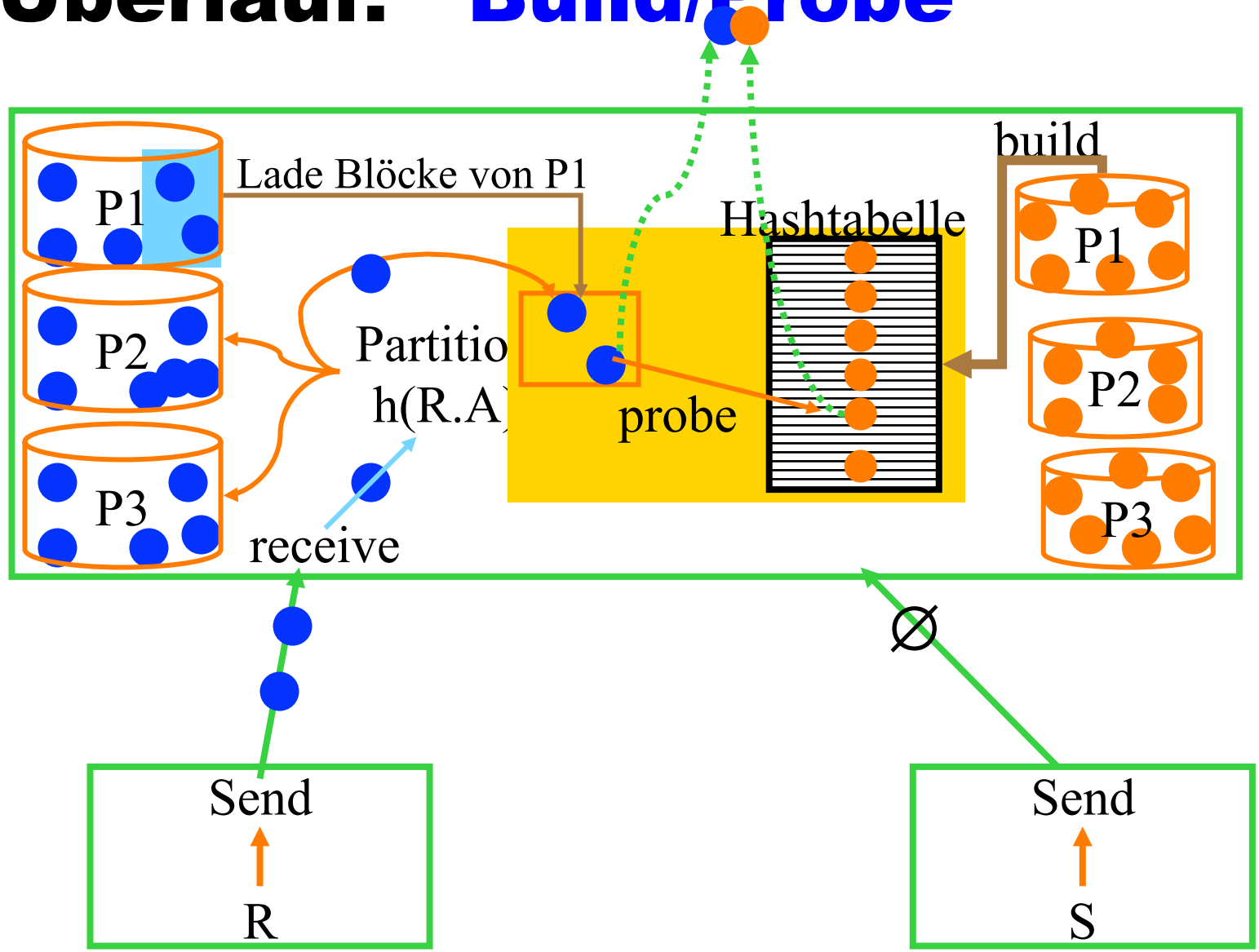
# „Normaler“ blockierender Hash-Join mit Überlauf: Partitionieren



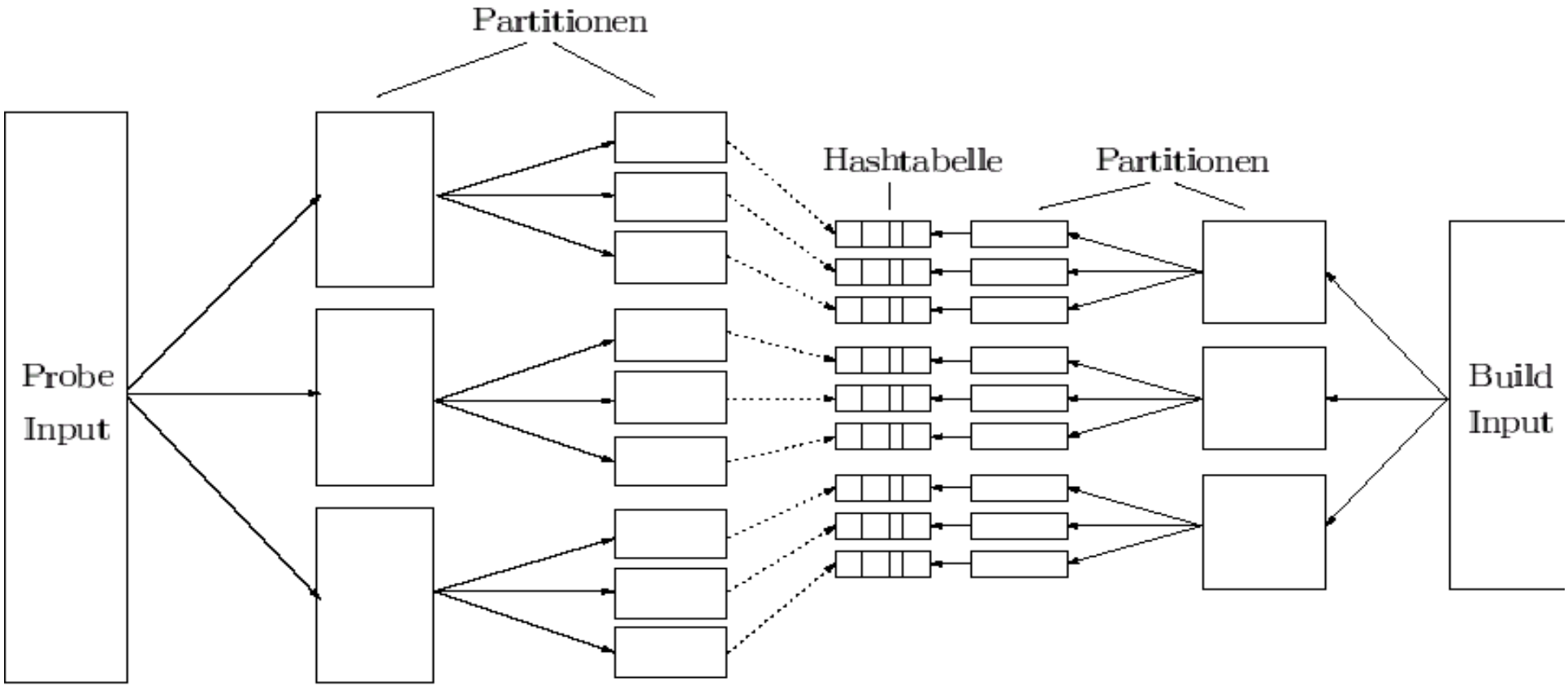
# „Normaler“ blockierender Hash-Join mit Überlauf: Partitionieren



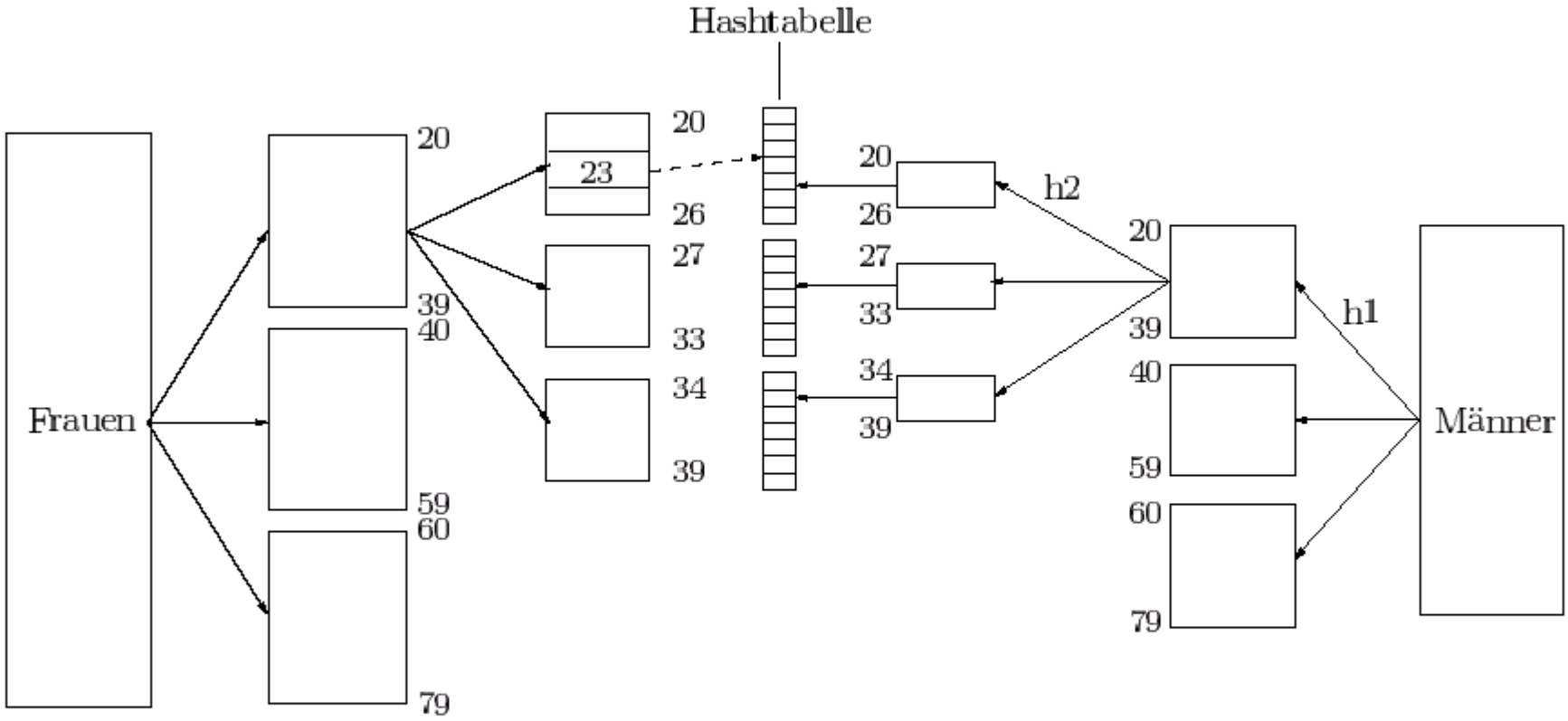
# „Normaler“ blockierender Hash-Join mit Überlauf: Build/Probe



# Partitionierung von Relationen



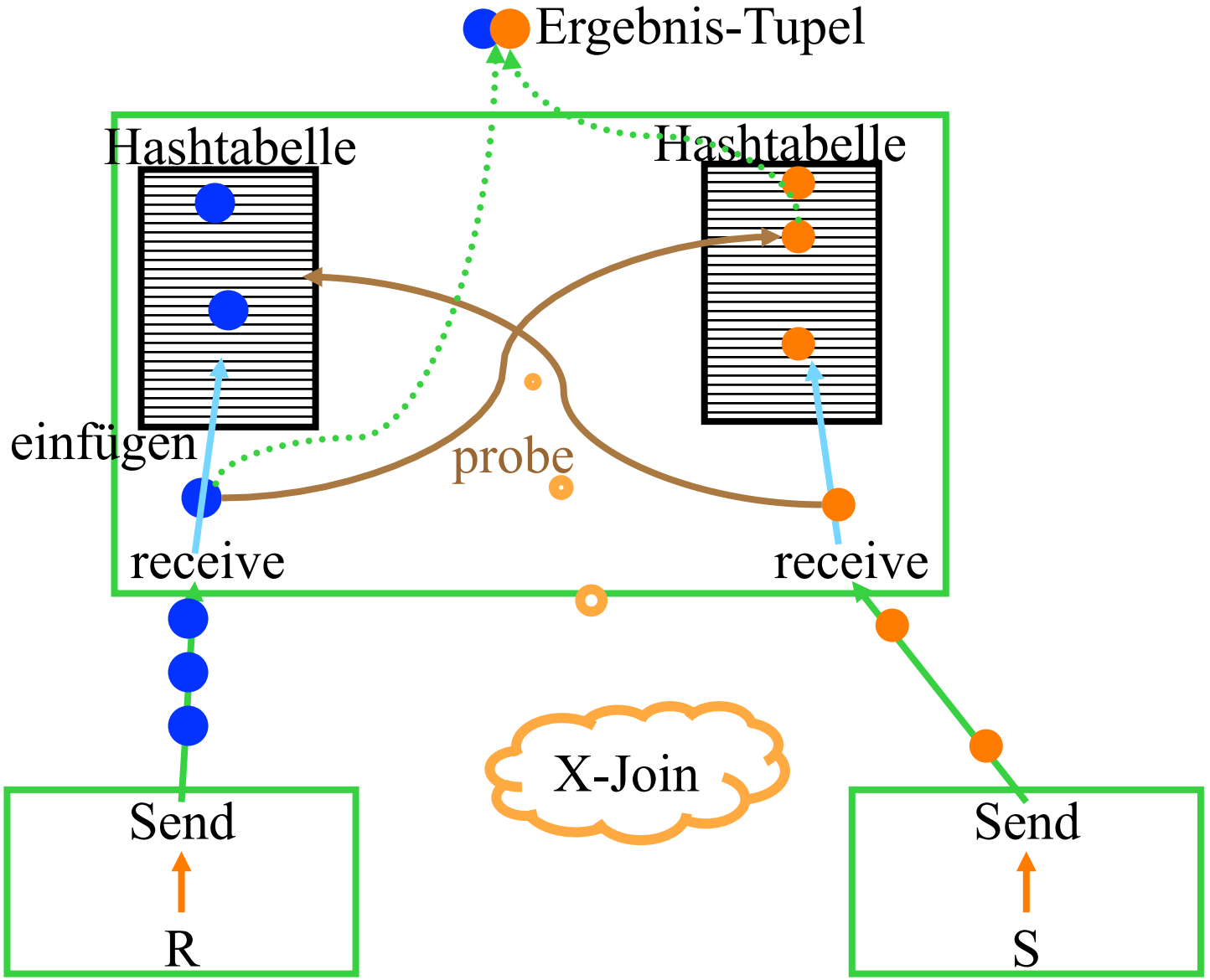
# Demonstration der Partitionierung



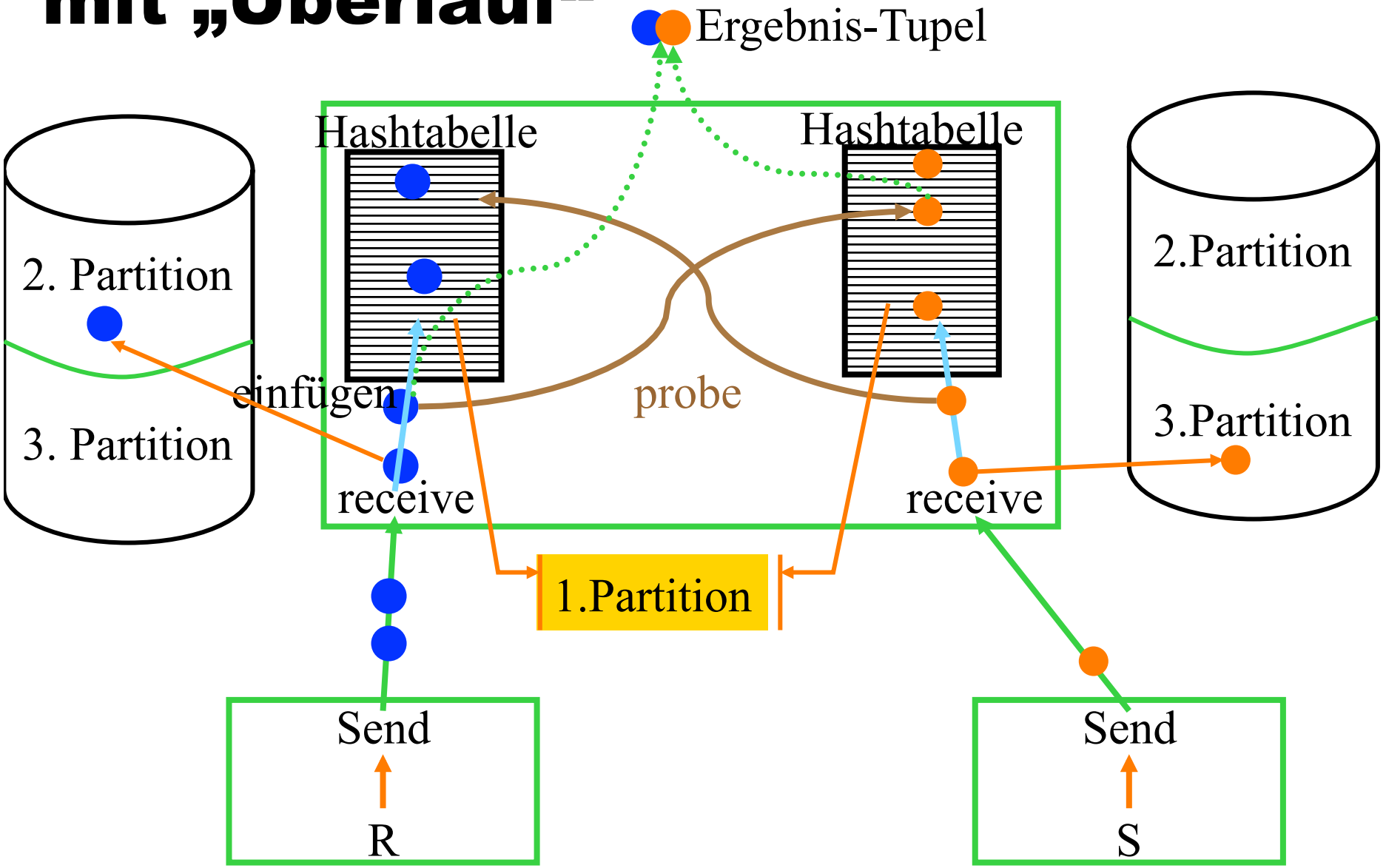
# Resümee: Hash Join (in verteilten Datenbanken)

- Der Build-Input (hier S) muß erst vollständig transferiert sein, bevor das erste Join-Tupel generiert werden kann
- Man kann die erste Partition schon „join-en" während der Probe-Input (hier R) noch angeliefert wird
  - Die 2. Partition kann aber erst bearbeitet werden, sobald der Probe-Input vollständig empfangen wurde
- Normalerweise nimmt man die kleinere Relation als Build-Input
- In verteilten Systemen muß man die Kommunikationskosten mit berücksichtigen
  - möglicherweise nimmt man die größere Relation wenn sie schneller transferiert werden kann
  - dynamisches Umschalten zwischen Build- und Probe-Input, falls der Build-Input zu langsam geliefert wird

# Double-Pipelined Hash-Join



# Double-Pipelined Hash-Join mit „Überlauf“



# Spezielle Join-Pläne für Verteilte DBMS

- Semi-Join zur Reduktion des Datentransfer-Volumens
  - $R \otimes_F S = (R \otimes^{lsj}_F S) \otimes_F S$
  - $R \otimes_F S = (R \otimes^{lsj}_F S) \otimes_F (R \otimes^{rsj}_F S)$
- Hash-Filter-Join
  - anstatt des Semijoin-Ergebnisses wird ein Bitvektor, der das Semijoin-Ergebnis approximiert, generiert.

# Der natürliche Verbund zweier Relationen $R$ und $S$

| $R$ | | |
|---|---|---|
| $A$ | $B$ | $C$ |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_3$ | $b_3$ | $c_1$ |
| $a_4$ | $b_4$ | $c_2$ |
| $a_5$ | $b_5$ | $c_3$ |
| $a_6$ | $b_6$ | $c_2$ |
| $a_7$ | $b_7$ | $c_6$ |

$\bowtie$

| $S$ | | |
|---|---|---|
| $C$ | $D$ | $E$ |
| $c_1$ | $d_1$ | $e_1$ |
| $c_3$ | $d_2$ | $e_2$ |
| $c_4$ | $d_3$ | $e_3$ |
| $c_5$ | $d_4$ | $e_4$ |
| $c_7$ | $d_5$ | $e_5$ |
| $c_8$ | $d_6$ | $e_6$ |
| $c_5$ | $d_7$ | $e_7$ |

$=$

| $R \bowtie S$ | | | | |
|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $E$ |
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ |
| $a_3$ | $b_3$ | $c_1$ | $d_1$ | $e_1$ |
| $a_5$ | $b_5$ | $c_3$ | $d_2$ | $e_2$ |

# Join-Auswertung mit Filterung ...

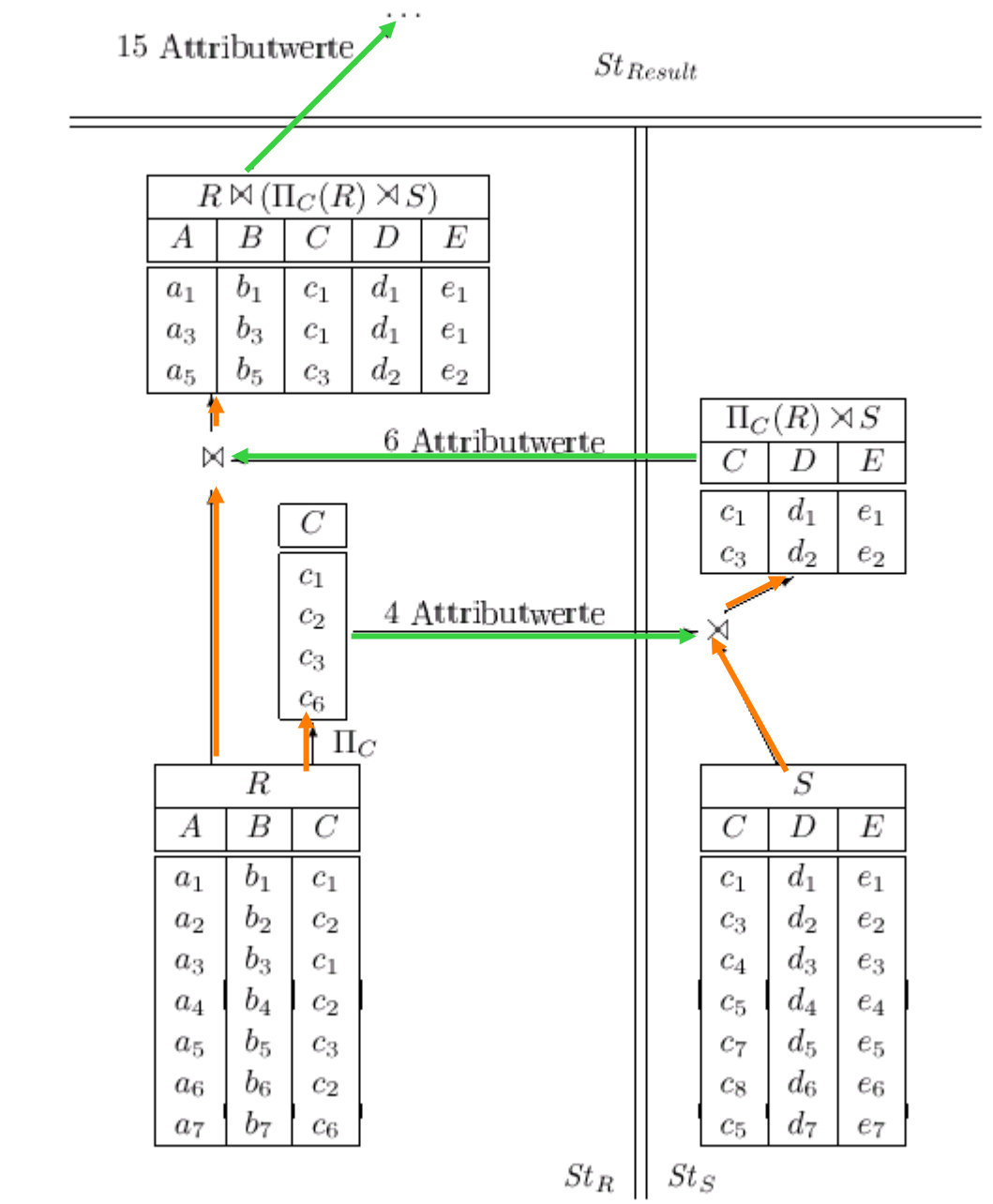
## ...einer Argumentrelation – hier $S$

$$R \bowtie S \quad = \quad R \bowtie (R \rtimes S)$$

$$R \rtimes S \quad = \quad \Pi_C(R) \rtimes S$$

$$R \rtimes S = \Pi_C(R) \rtimes S$$

$$|| \Pi_C(R) || + || R \rtimes S || \quad < \quad || S ||$$




| $R \bowtie (\Pi_C(R) \rtimes S)$ | | | | |
|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $E$ |
| $a_1$ | $b_1$ | $c_1$ | $d_1$ | $e_1$ |
| $a_3$ | $b_3$ | $c_1$ | $d_1$ | $e_1$ |
| $a_5$ | $b_5$ | $c_3$ | $d_2$ | $e_2$ |

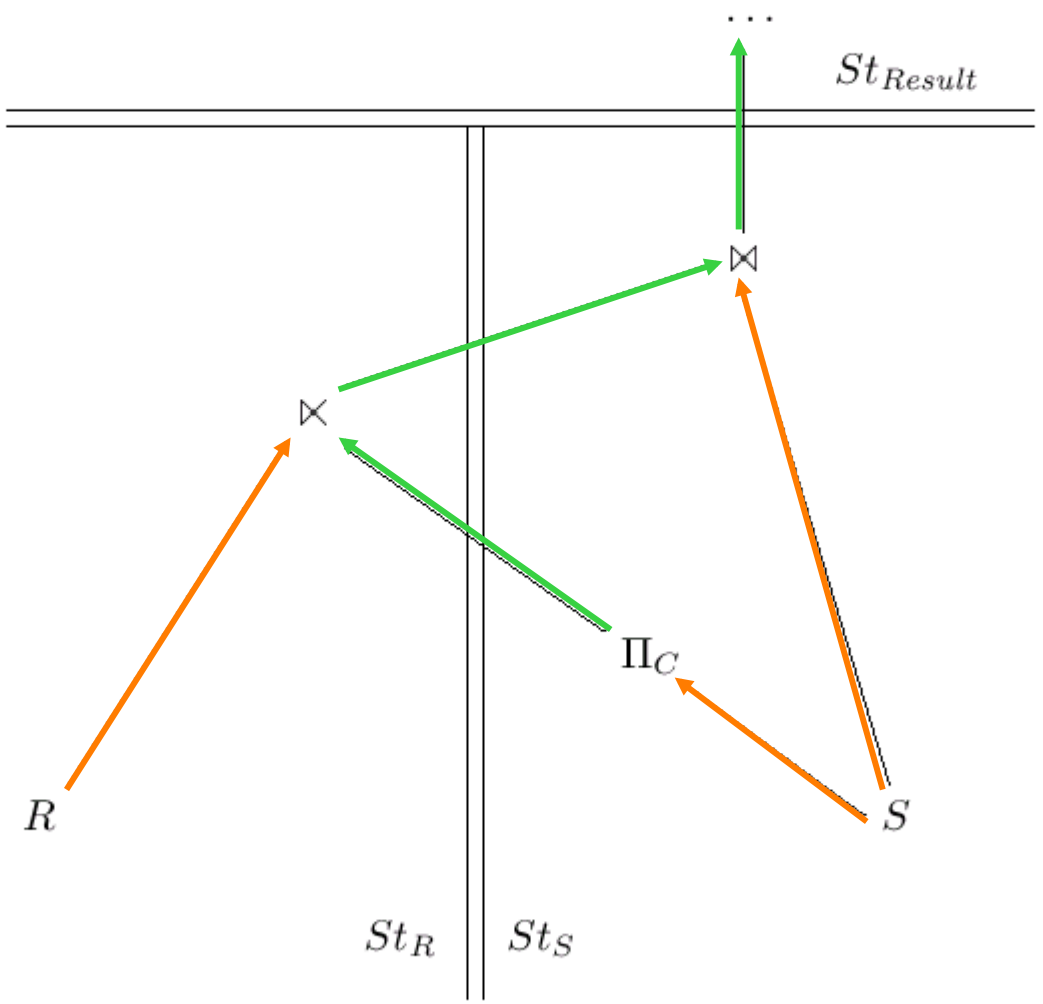
| $\Pi_C(R) \rtimes S$ | | |
|---|---|---|
| $C$ | $D$ | $E$ |
| $c_1$ | $d_1$ | $e_1$ |
| $c_3$ | $d_2$ | $e_2$ |

| $C$ |
|---|
| $c_1$ |
| $c_2$ |
| $c_3$ |
| $c_6$ |

| $R$ | | |
|---|---|---|
| $A$ | $B$ | $C$ |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_3$ | $b_3$ | $c_1$ |
| $a_4$ | $b_4$ | $c_2$ |
| $a_5$ | $b_5$ | $c_3$ |
| $a_6$ | $b_6$ | $c_2$ |
| $a_7$ | $b_7$ | $c_6$ |

| $S$ | | |
|---|---|---|
| $C$ | $D$ | $E$ |
| $c_1$ | $d_1$ | $e_1$ |
| $c_3$ | $d_2$ | $e_2$ |
| $c_4$ | $d_3$ | $e_3$ |
| $c_5$ | $d_4$ | $e_4$ |
| $c_7$ | $d_5$ | $e_5$ |
| $c_8$ | $d_6$ | $e_6$ |
| $c_5$ | $d_7$ | $e_7$ |

# Alternativer Auswertungsplan mit Semi-Join-Filterung

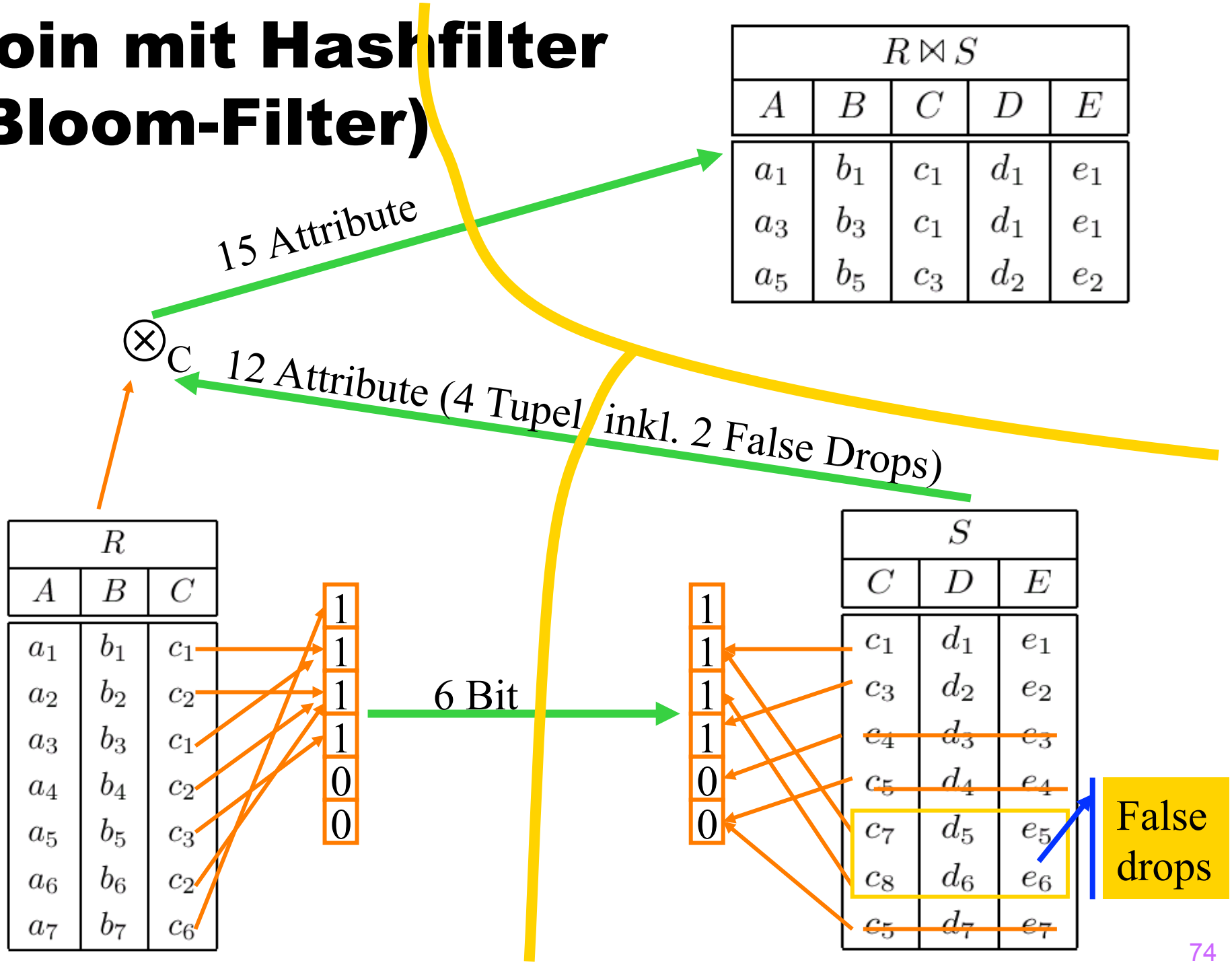


# Semi-Join-Pläne beim Mehrwege-Join

- R(A,B) ⊗ S(B,C) ⊗ T(C,D)
  - Join-Graph:

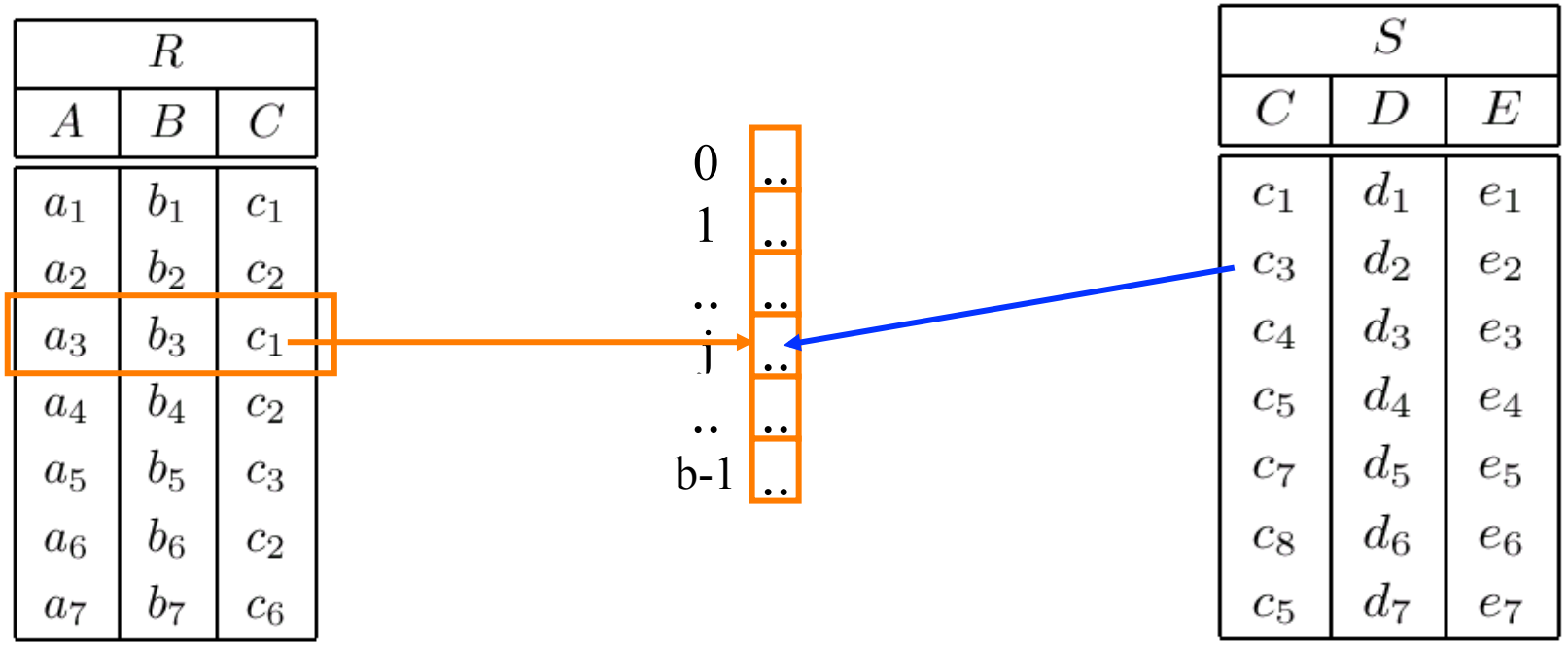

  - [ R $\otimes^{lsj}$ (S $\otimes^{lsj}$ T) ] ⊗ S ⊗ T
  - [ R $\otimes^{lsj}$ (S $\otimes^{lsj}$ T) ] ⊗ [ S $\otimes^{lsj}$ T ] ⊗ T
  - [ R $\otimes^{lsj}$ S ] ⊗ [ S $\otimes^{lsj}$ T ] ⊗ T
  - [ R $\otimes^{lsj}$ S ] ⊗ [ R $\otimes^{rsj}$ S ] ⊗ T
  - [ R $\otimes^{lsj}$ S ] ⊗ [ R $\otimes^{rsj}$ S $\otimes^{lsj}$ T ] ⊗ T
  - [ R $\otimes^{lsj}$ S ] ⊗ [ R $\otimes^{rsj}$ S $\otimes^{lsj}$ T ] ⊗ [ T $\otimes^{lsj}$ S ]
  - [[ R ⊗ S ] $\otimes^{lsj}$ T ] ⊗ T
  - .....
  - Finde den besten Plan: full reducer
    - nur bei azyklischen Join-Graphen möglich

# Join mit Hashfilter (Bloom-Filter)



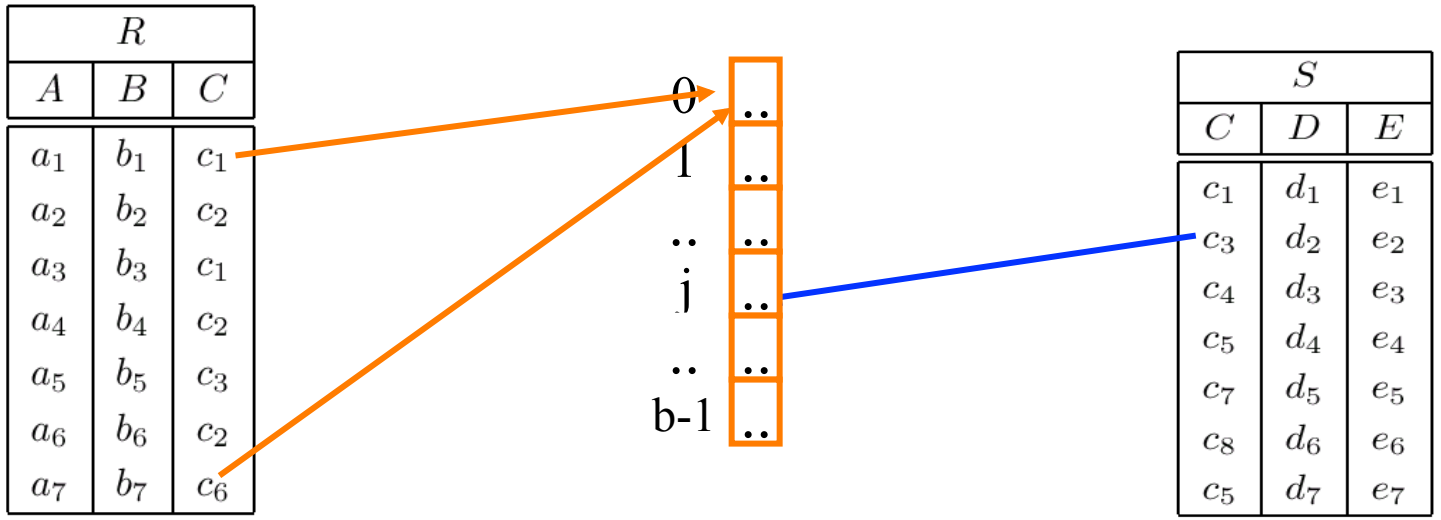
# Join mit Hashfilter (False Drop Abschätzung)

- Wahrscheinlichkeit, dass ein bestimmtes Bit j gesetzt ist
  - W. dass ein bestimmtes r∈R das Bit setzt: 1/b
  - W. dasss kein r∈R das Bit setzt: $(1-1/b)^{|R|}$
  - W. dass ein r∈R das Bit gesetzt hat: $1- (1-1/b)^{|R|}$



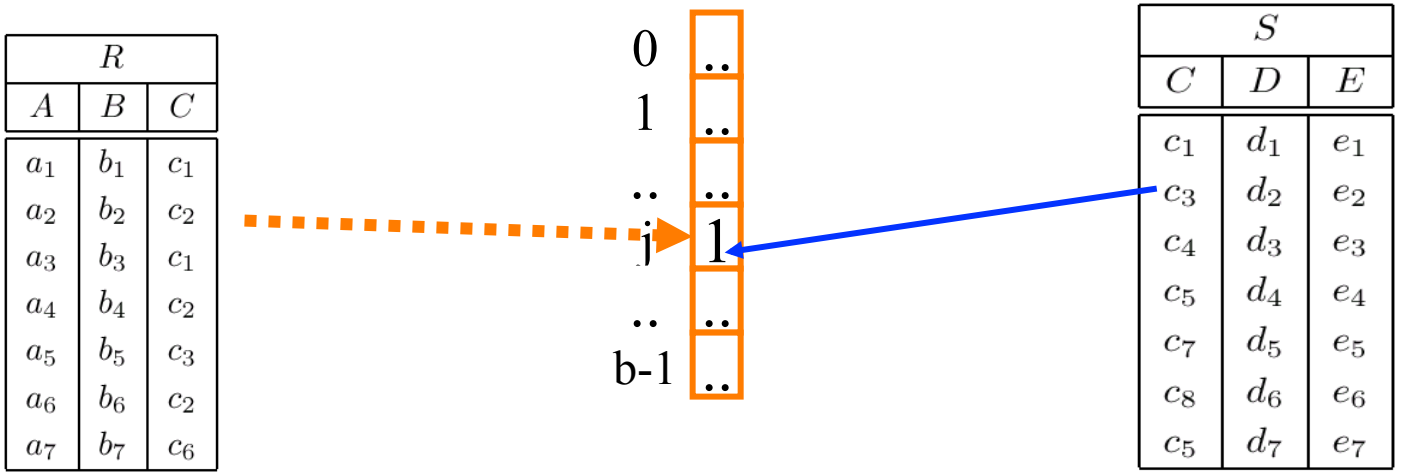
# Join mit Hashfilter (False Drop Abschätzung)

- W. dass irgendein r∈R ein bestimmtes Bit gesetzt hat: $1-(1-1/b)^{|R|}$
- Wieviele Bits sind gesetzt?
  - $b * [1- (1-1/b)^{|R|}]$
- Mehrere r∈R können dasselbe Bit setzen
- Approximation: alle r∈R setzen unterschiedliche Bits
  - W. dass ein bestimmtes Bit j gesetzt ist: |R| / b
  - b >> |R|



# Join mit Hashfilter (False Drop Abschätzung)

- W. dass irgendein r∈R ein bestimmtes Bit gesetzt hat: $1-(1-1/b)^{|R|}$
- W. dass ein bestimmtes s∈S ausgewählt wird:
  - $1-(1-1/b)^{|R|}$
- Wieviele s∈S werden ausgewählt?
  - $|S| * [1-(1-1/b)^{|R|}]$
- Approximation: alle r setzen unterschiedliche Bits
  - W. dass ein bestimmtes Bit j gesetzt ist: |R| / b
  - |S|*(|R|/b) Elemente aus S werden ausgewählt

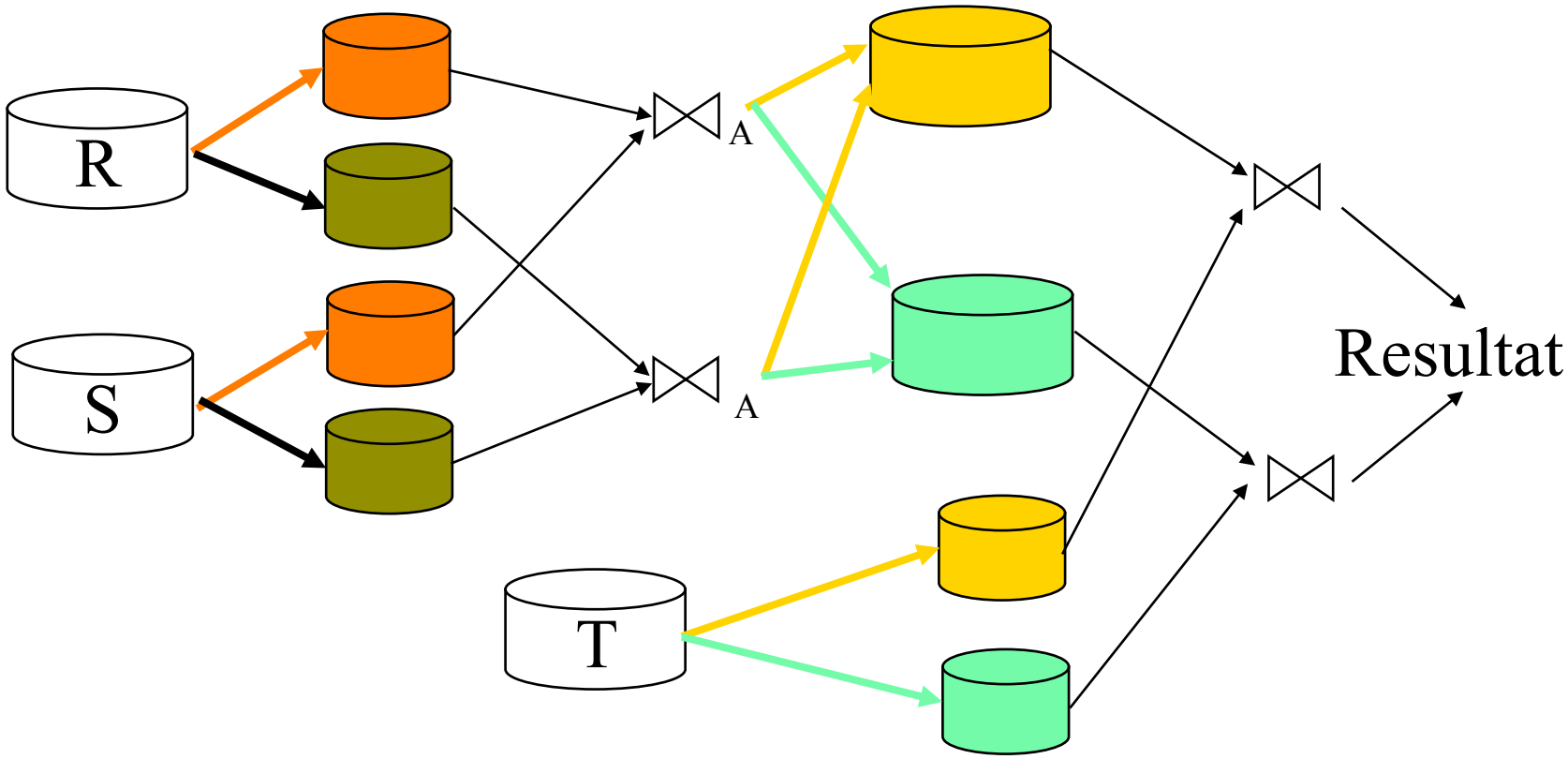


# Weitere Einsatzmöglichkeiten für Hash-Filter

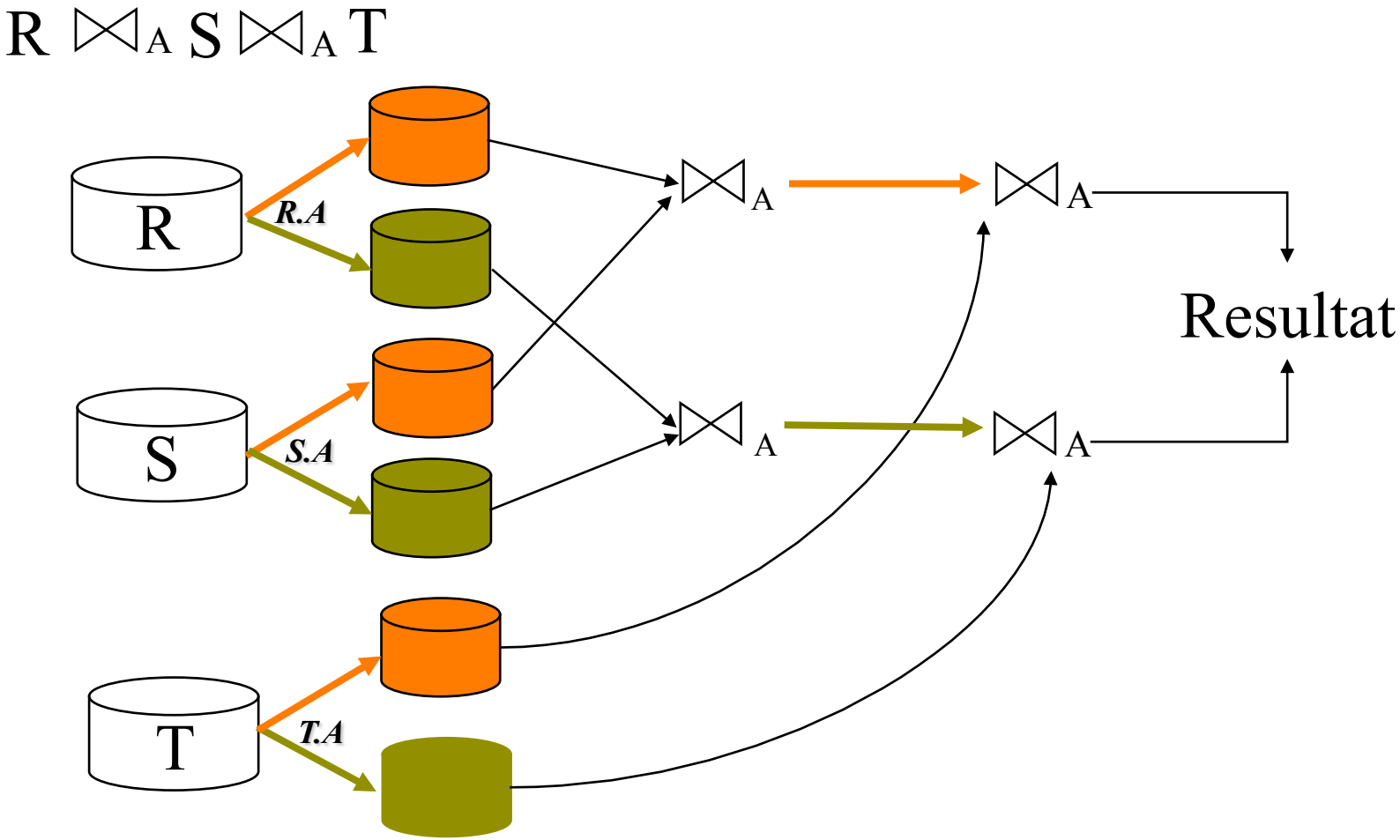
- Für Signatur-Files (~Indexe für die Filterung von Daten)
- Beim „normalen" Hash-Join
  - beim Partitionieren der einen Relation wird eine Bitmap (mgl. pro Partition) gebaut
  - beim Partitionieren der anderen Relation wird/werden diese Bitmaps zum Filtern verwendet
  - z.B. in MS SQL-Server eingebaut
- Zur abgeleiteten Partitionierung hierarchischer Datenstrukturen
  - early partitioning

# Traditioneller Join Plan

$R \bowtie S \bowtie T$

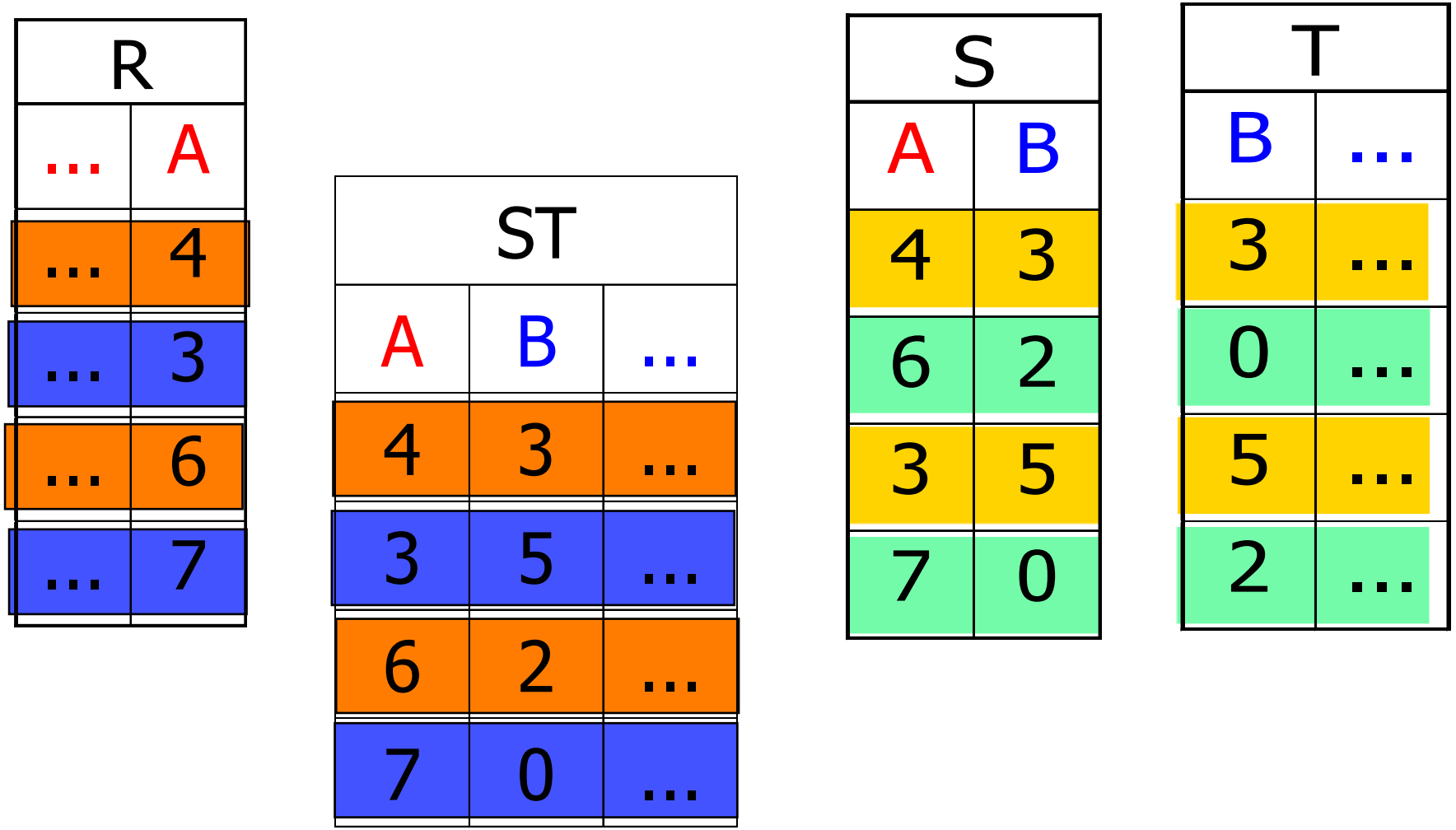


# Traditioneller Hash Team Join Plan
## [Graefe, Bunker, Cooper: VLDB 98, MS SQL Server]



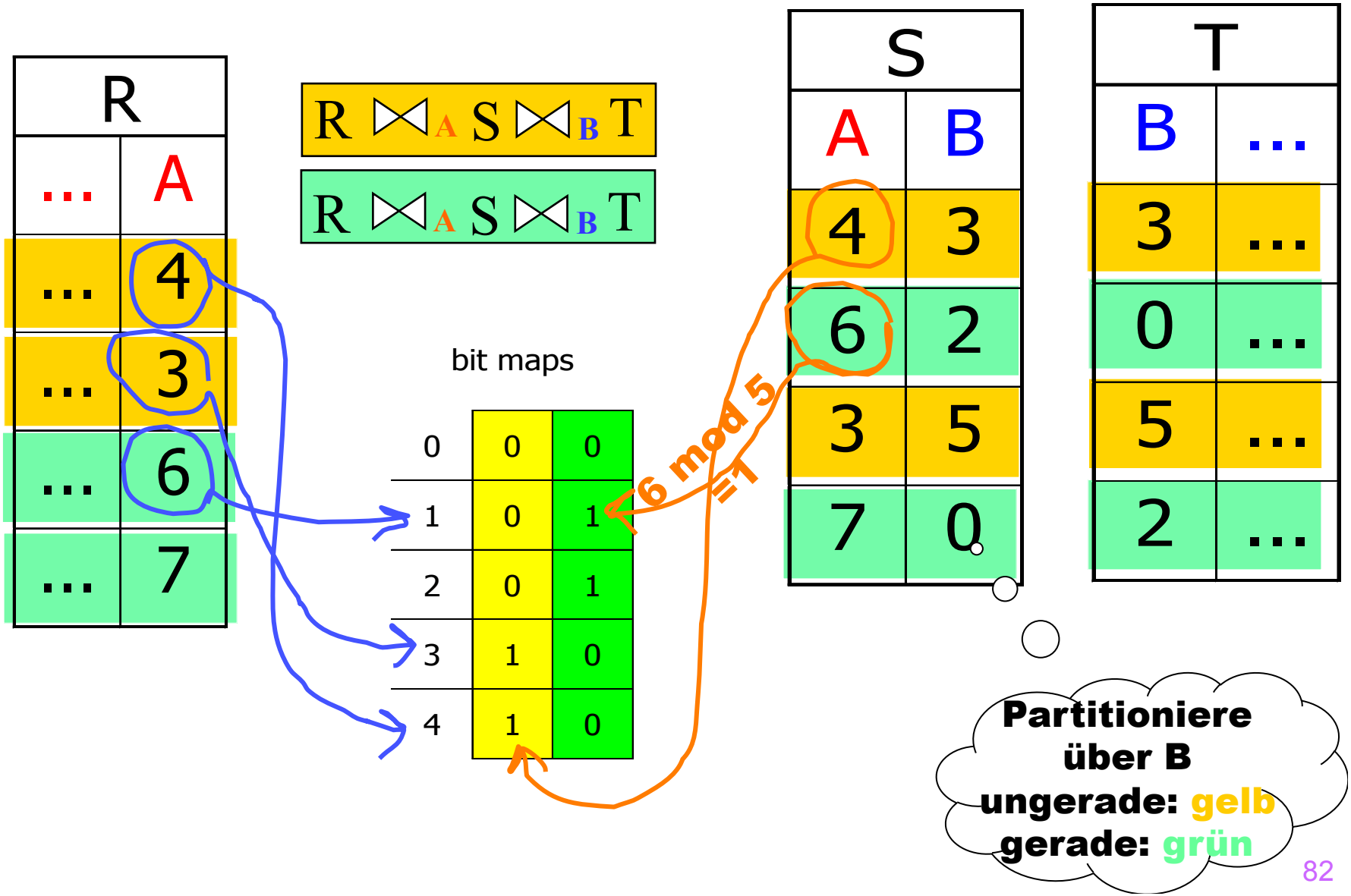
# Generalized Hash Teams

$R \bowtie_A S \bowtie_B T$



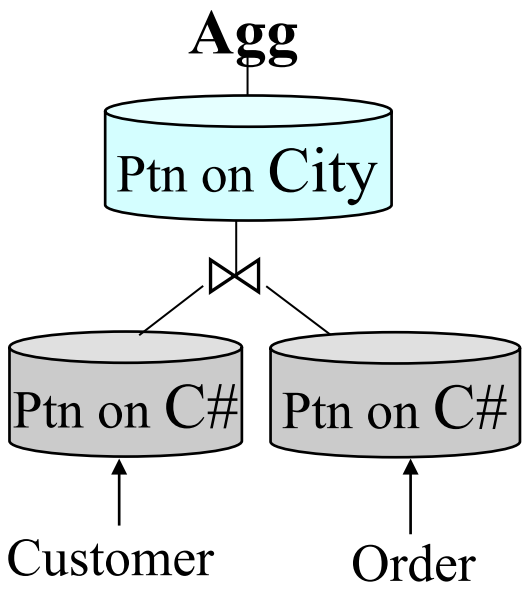
# Generalisierte Hash Teams

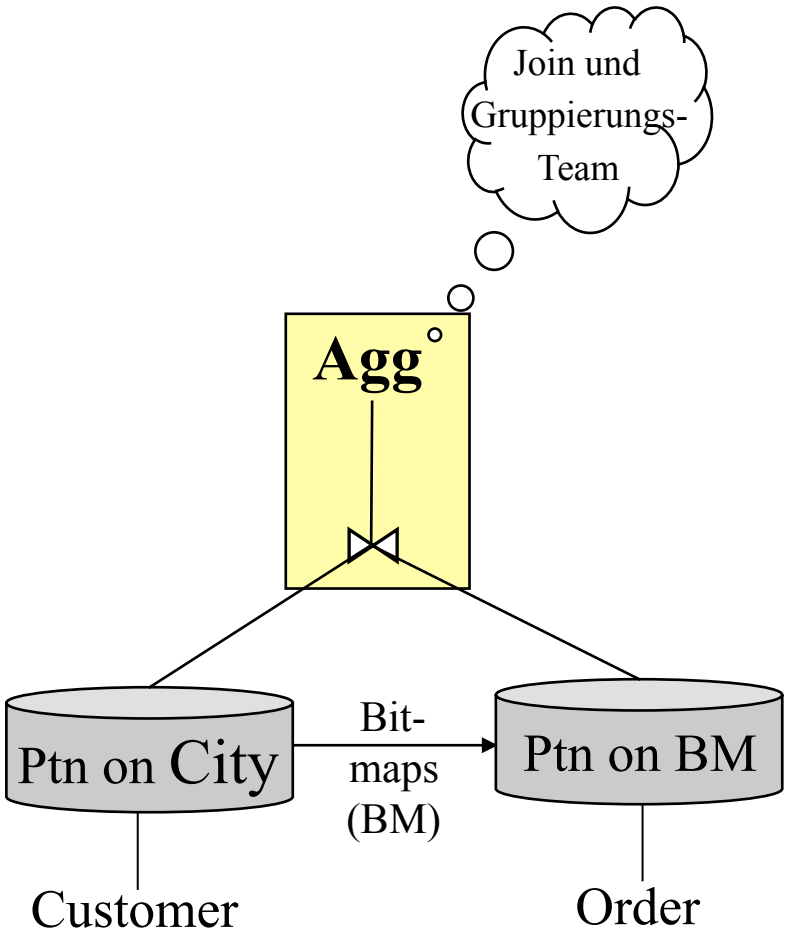
## [Kemper et al., VLDB 1999 und VLDBJ 2000]



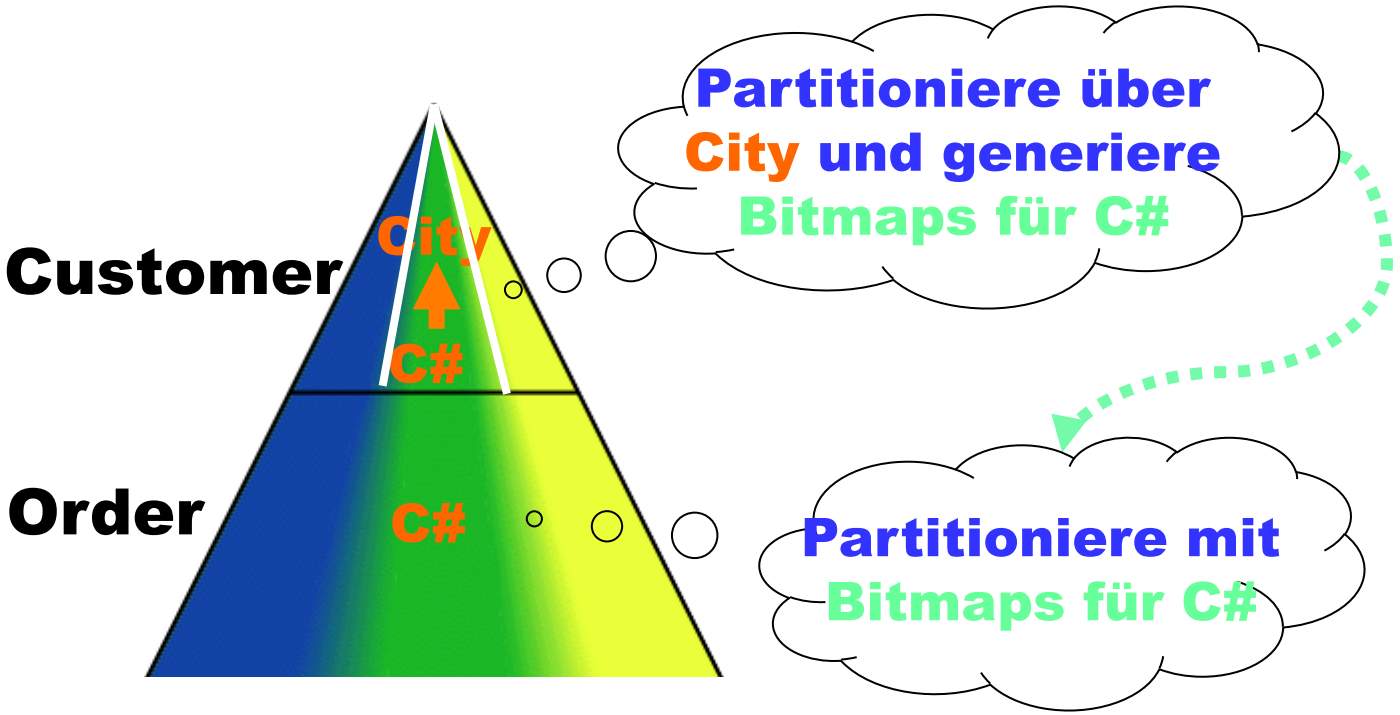
# Generalisierte Hash Team für Gruppierung/Aggregation

- **select** c.City, **sum**(o.Value)
  **from** Customer c, Order o
  **where** c.C# = o.C#
  **group** by c.City



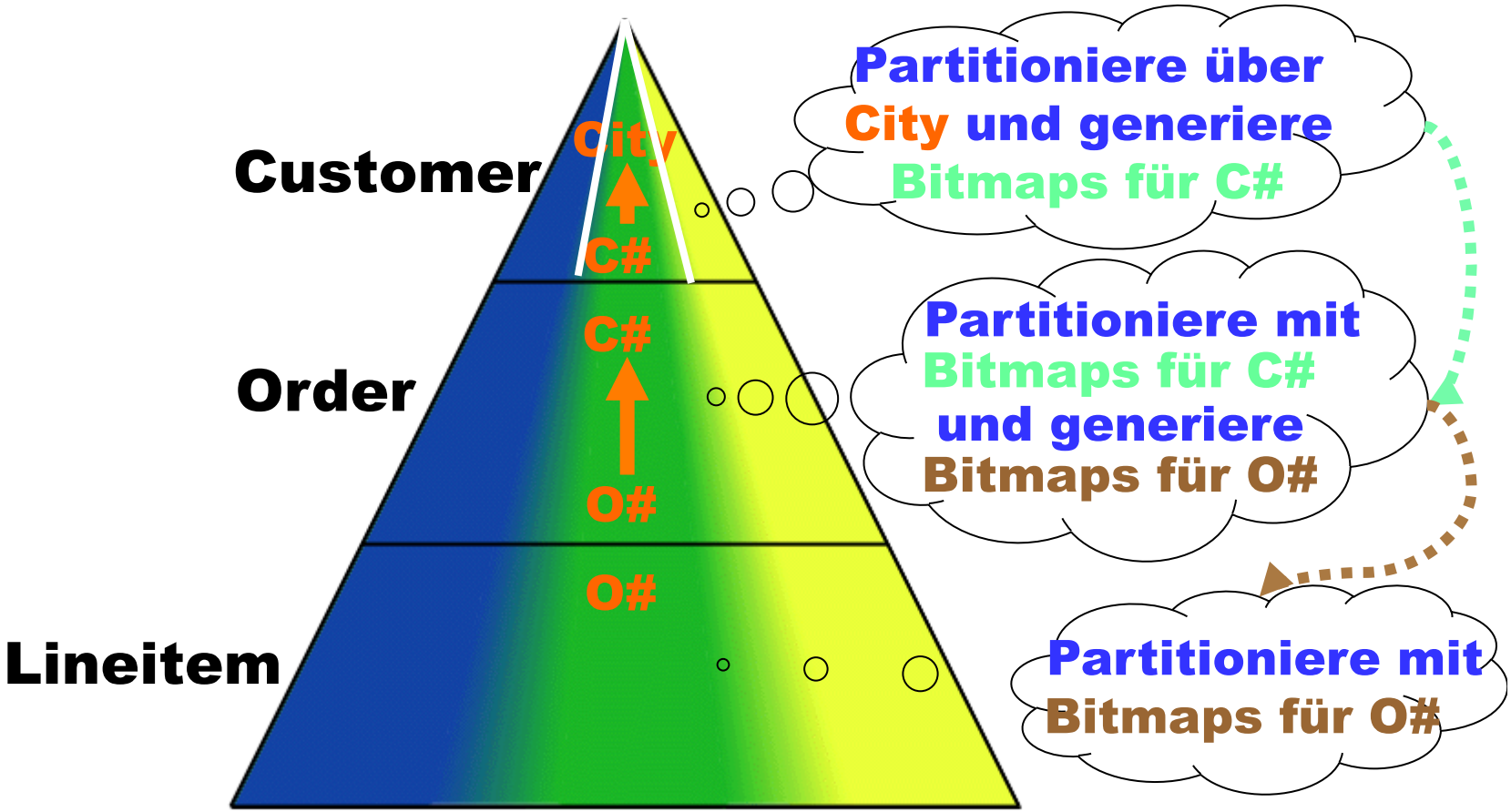



# $\text{Group}_{\text{City}}(\text{Customer} \bowtie_{\text{C\#}} \text{Order})$

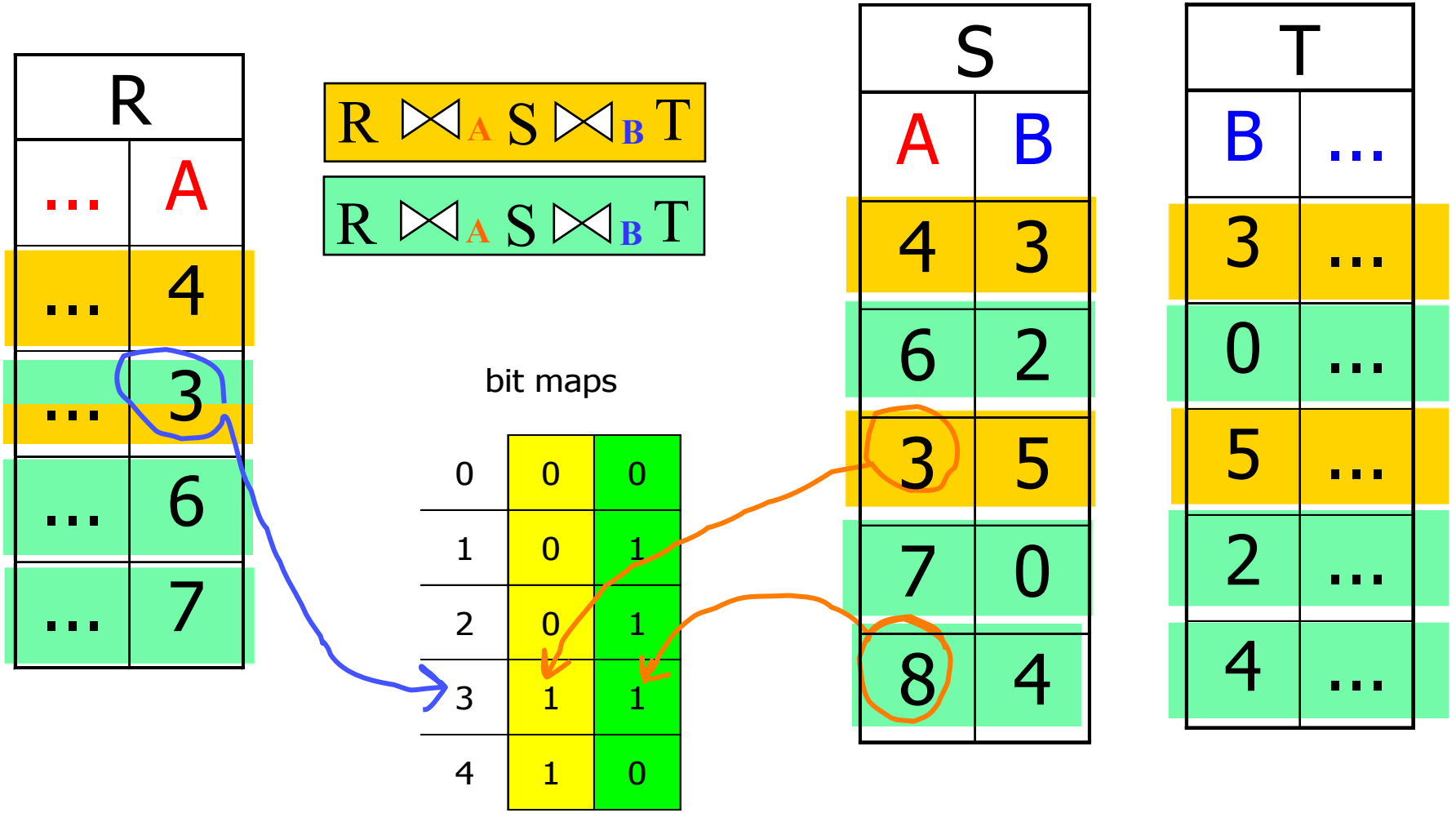



Customer: {[C#, Name, City, ...]}
Order: {[O#, C#, Wert, ...]}
Lineitem: {[O#, L#, Anzahl, Preis, ...]}

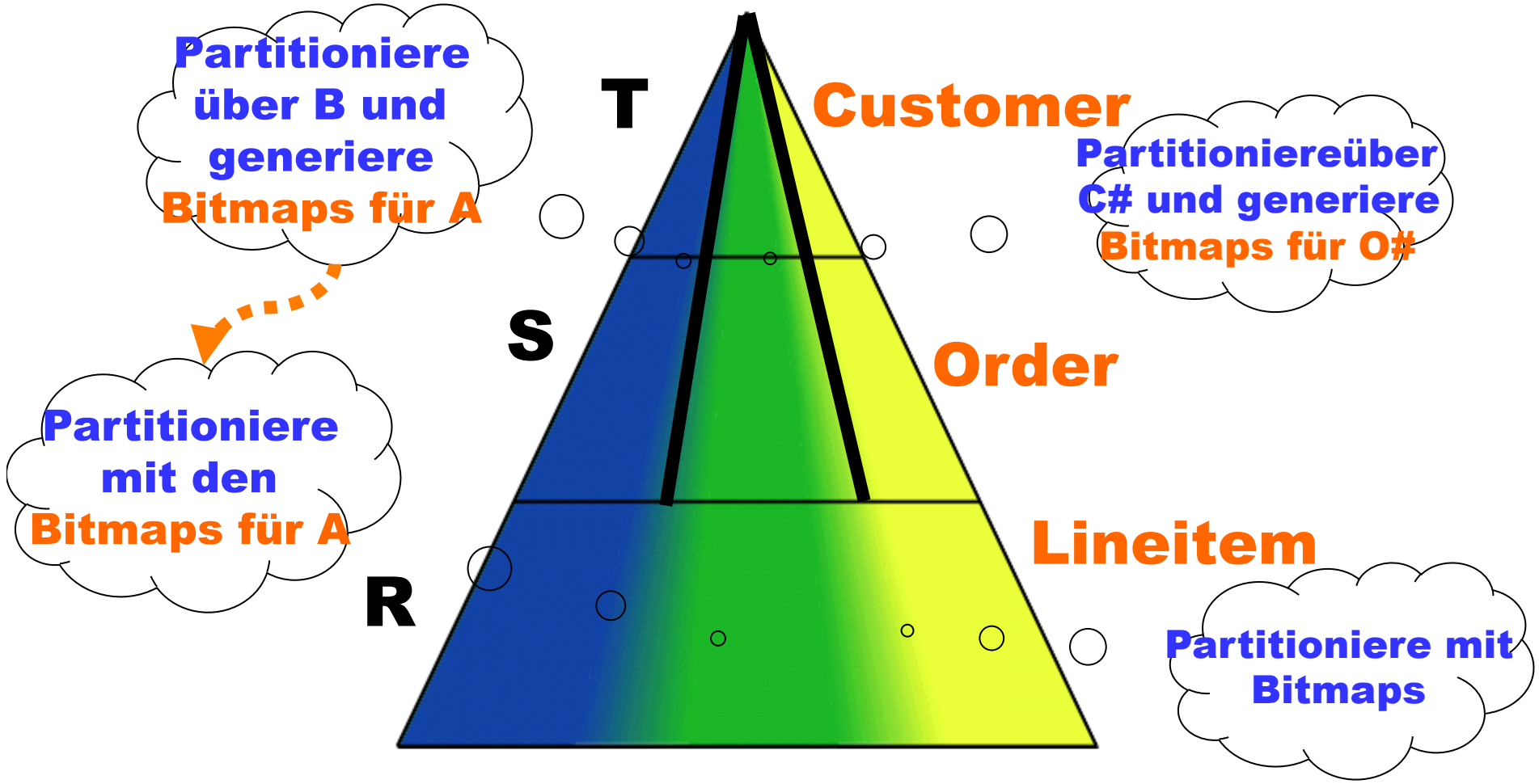
$$\text{Group}_{\text{City}}(\text{Customer} \bowtie_{\text{C\#}} \text{Order} \bowtie_{\text{O\#}} \text{Lineitem})$$



# False Drops


R
... A
... 4
... 3
... 6
... 7
R ⋈A S ⋈B T
R ⋈A S ⋈B T
bit maps
0 0 0
1 0 1
2 0 1
3 1 1
4 1 0
S
A B
4 3
6 2
3 5
7 0
8 4
T
B ...
3 ...
0 ...
5 ...
2 ...
4 ...

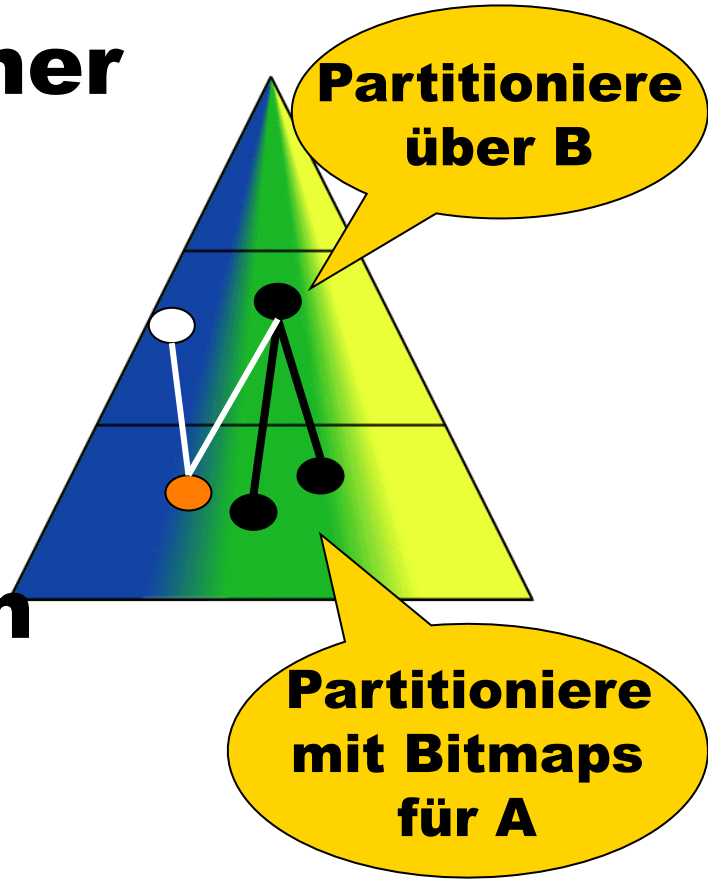
# Überlappende Partitionen




$$(\text{Customer} \bowtie_{C\#} \text{Order} \bowtie_{O\#} \text{Lineitem})$$

# Anwendbarkeit der Generalisierten Hash Teams

- **für die Partitinierung partitioning hierarchischer Strukturen A → B**




- **aber auch korrekt für nicht-strikte Hierarchien A ↛ B (aber Leistungs-Degradierung zu befürchten)**

# Nicht-strikte Hierarchie

A ⇸ B


| R | |
|---|---|
| ... | A |
| ... | 4 |
| ... | 3 |
| ... | 6 |
| ... | 7 |

$R \bowtie_A S \bowtie_B T$

$R \bowtie_A S \bowtie_B T$

bit maps

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 0 | 1 |
| 3 | 1 | 1 |
| 4 | 1 | 0 |

| S | |
|---|---|
| A | B |
| 4 | 3 |
| 6 | 2 |
| 3 | 5 |
| 7 | 0 |
| 3 | 2 |

| T | |
|---|---|
| B | ... |
| 3 | ... |
| 0 | ... |
| 5 | ... |
| 2 | ... |

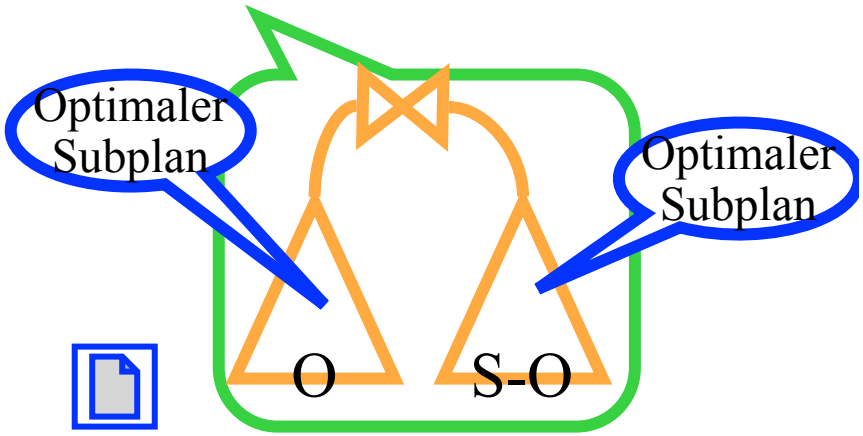
# Optimierung zentralisierter Anfragen

## Grundsätze:

- Sehr hohes Abstraktionsniveau der mengenorientierten Schnittstelle (SQL).
- Sie ist deklarativ, nicht-prozedural, d.h. es wird spezifiziert, was man finden möchte, aber nicht wie.
- Das wie bestimmt sich aus der Abbildung der mengenorientierten Operatoren auf Schnittstellen-Operatoren der internen Ebene (Zugriff auf Datensätze in Dateien, Einfügen/Entfernen interner Datensätze, Modifizieren interner Datensätze).
- Zu einem was kann es zahlreiche wie‘s geben: effiziente Anfrageauswertung durch Anfrageoptimierung.
- i.Allg. wird aber nicht die optimale Auswertungsstrategie gesucht (bzw. gefunden) sondern eine einigermaßen effiziente Variante
  - Ziel: „avoiding the worst case“

# Optimierung durch Dynamische Programmierung

- Standardverfahren in heutigen relationalen Datenbanksystemen
- Voraussetzung ist ein Kostenmodell als Zielfunktion
  - I/O-Kosten
  - CPU-Kosten
- DP basiert auf dem Optimalitätskriterium von Bellman
- Literatur zu DP:
  - D. Kossmann und K. Stocker: Iterative Dynamic Programming, TODS, 2000 (online)



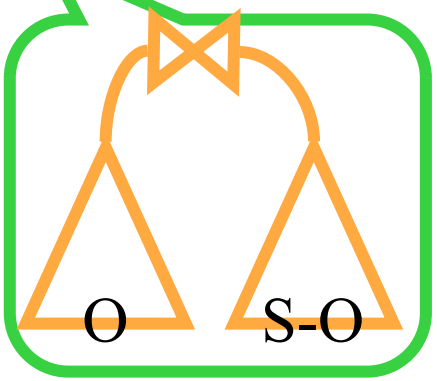
# „Klassische“ Optimierung durch Dynamisches Programmieren

**Input:** SPJ query $q$ on relations $R_1, \ldots, R_n$
**Output:** A query plan for $q$

1: **for** $i = 1$ **to** $n$ **do** {
2: optPlan($\{R_i\}$) = accessPlans($R_i$)
3: prunePlans(optPlan($\{R_i\}$))
4: }
5: **for** $i = 2$ **to** $n$ **do** {
6: **for all** $S \subseteq \{R_1, \ldots, R_n\}$ **such that** $|S| = i$ **do** {
7: optPlan($S$) = $\emptyset$
8: **for all** $O \subset S$ **do** {
9: optPlan($S$) = optPlan($S$) $\cup$ joinPlans(optPlan($O$), optPlan($S - O$))
10: prunePlans(optPlan($S$))
11: }
12: }
13: }
14: finalizePlans(optPlan($\{R_1, \ldots, R_n\}$))
15: prunePlans(optPlan($\{R_1, \ldots, R_n\}$))
16: **return** optPlan($\{R_1, \ldots, R_n\}$)

Tablescan
Indexscan
Clusterindex
Sekundärindex

Wenn zwei Pläne vergleichbar sind, behalte nur den billigsten



# DP - Beispiel



## 1. Phase: Zugriffspläne ermitteln

| Index | Pläne |
|---|---|
| {ABC} | |
| {BC} | |
| {AC} | |
| {AB} | |
| {C} | |
| {B} | |
| {A} | |

# DP - Beispiel



## 1. Phase: Zugriffspläne ermitteln

| Index | Pläne |
|---|---|
| {ABC} | |
| {BC} | |
| {AC} | |
| {AB} | |
| {C} | scan(C) |
| {B} | scan(B), iscan(B) |
| {A} | scan(A) |

# DP - Beispiel



## 2. Phase: Join-Pläne ermitteln (2-fach,...,n-fach)

| Index | Pläne |
|---|---|
| {ABC} | |
| {BC} | ... |
| {AC} | s(A)⋈s(C), s(C)⋈s(A) |
| {AB} | s(A)⋈s(B), s(A)⋈is(B), is(B)⋈s(A),... |
| {C} | scan(C) |
| {B} | scan(B), iscan(B) |
| {A} | scan(A) |

**Pruning**

# DP - Beispiel



## 3. Phase: Finalisierung

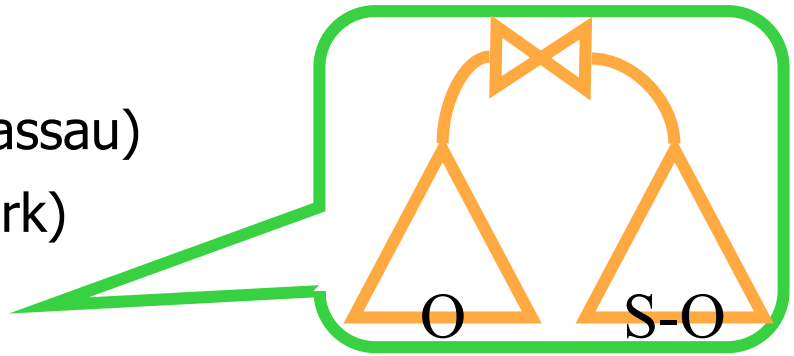
| Index | Pläne |
|---|---|
| {ABC} | (is(B) ⋈ s(A)) ⋈ s(C) |
| {BC} | ... |
| {AC} | s(A) ⋈ s(C) |
| {AB} | s(A) ⋈ is(B), is(B) ⋈ s(A) |
| {C} | scan(C) |
| {B} | scan(B), iscan(B) |
| {A} | scan(A) |

# „Fine Points“ der DP-Optimierung

- Komplexität $O(3^n)$
- Unvergleichbarkeit von zwei Plänen
  - R sort-merge-join S
  - R hash-join S
  - ersterer ist vermutlich teurer, generiert aber ein gemäß Join-Attribut sortiertes Ergebnis
    - unvergleichbare Pläne
  - führt möglicherweise später zu einem billigeren (merge-)Join (mit bspw. T)
  - Man spricht von „interesting physical properties“
- Also, haben wir nur eine partielle Ordnung zwischen den semantisch äquivalenten Plänen (Zeile 3 und 4)
  - Widerspricht eigentlich dem Optimalitätskriterium, das für die Anwendbarkeit von DP erforderlich ist

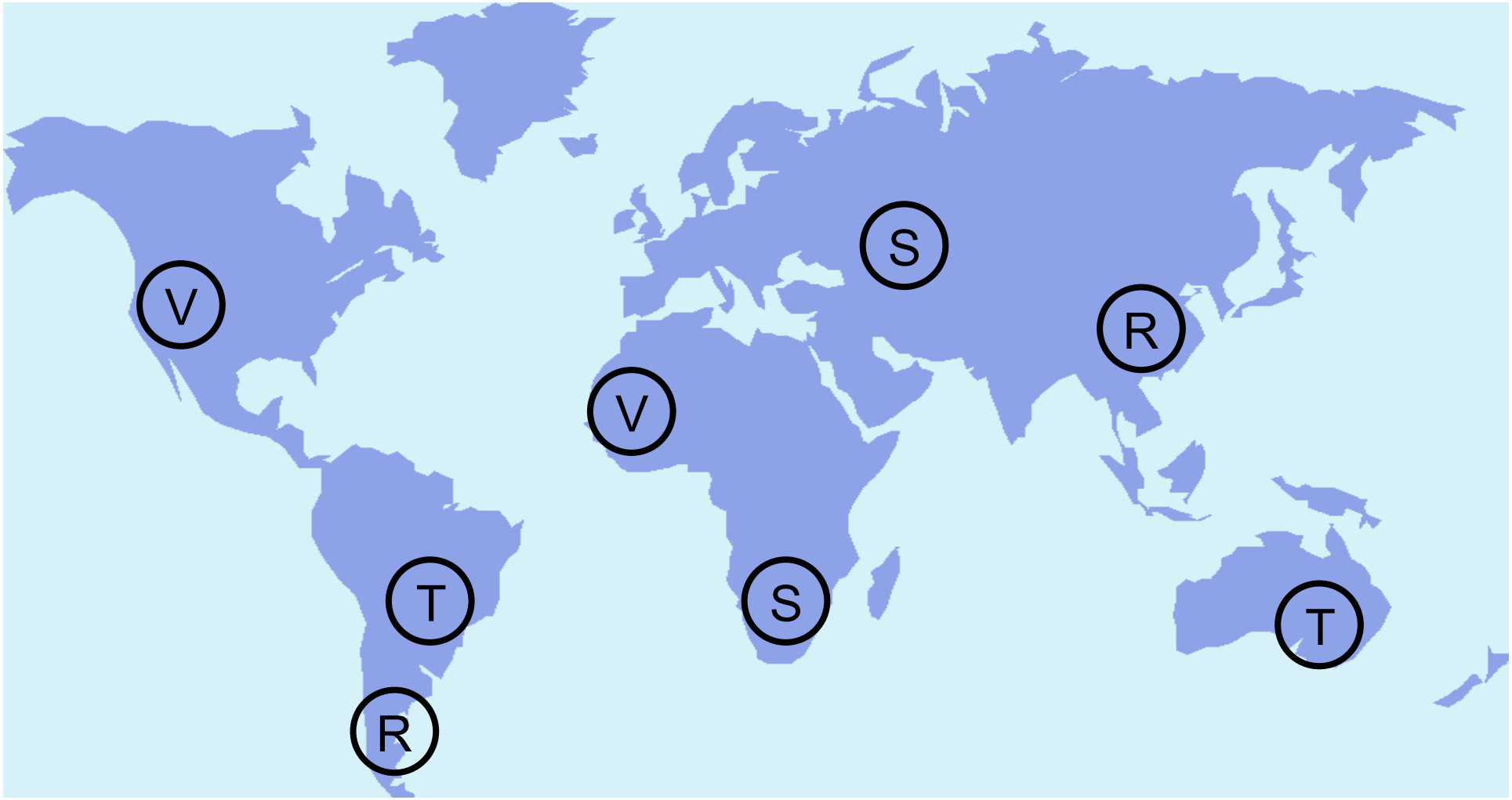
# Erweiterungen für Verteilte Datenbanken

- Replizierte Relationen: generiere accessPlans für alle Möglichkeiten
  - table_scan(Angestellte,Pasau)
  - idx_scan(Angestellte.Gehalt,Passau)
  - table_scan(Angestellte,NewYork)
- Ausführung des Joins (Zeile 9)

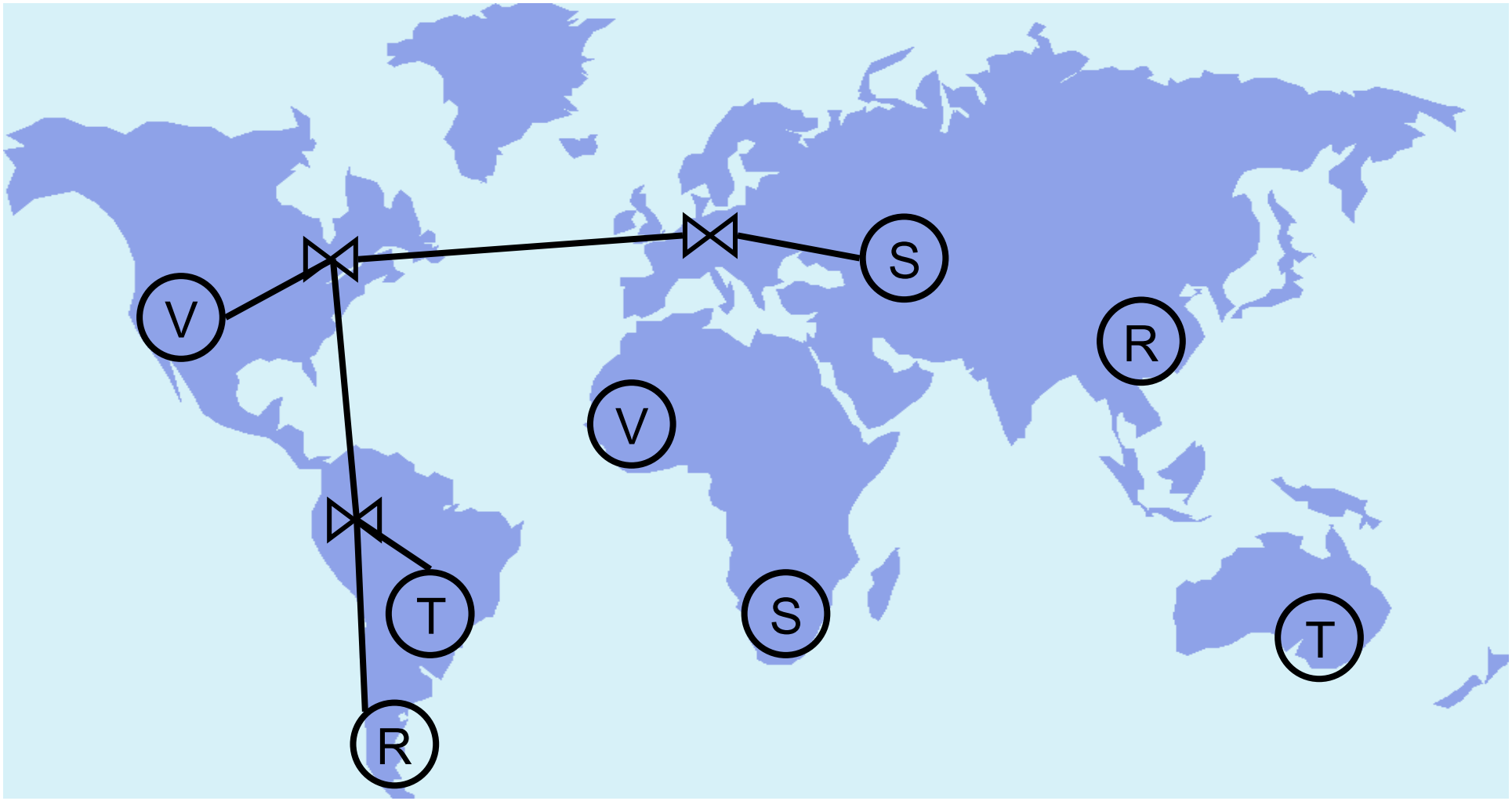



  - am Knoten wo das äußere/linke Join-Argument generiert wird
  - am Knoten wo das innere/rechte Join-Argument generiert wird
  - an allen weiteren „interessanten" Knoten
  - Für S={Ri1,...,Rik} sind alle Heimatknoten von {R1,...,Rn}\ {Ri1,...,Rik} und der Knoten, an den das Ergebnis kommen muss, interessant

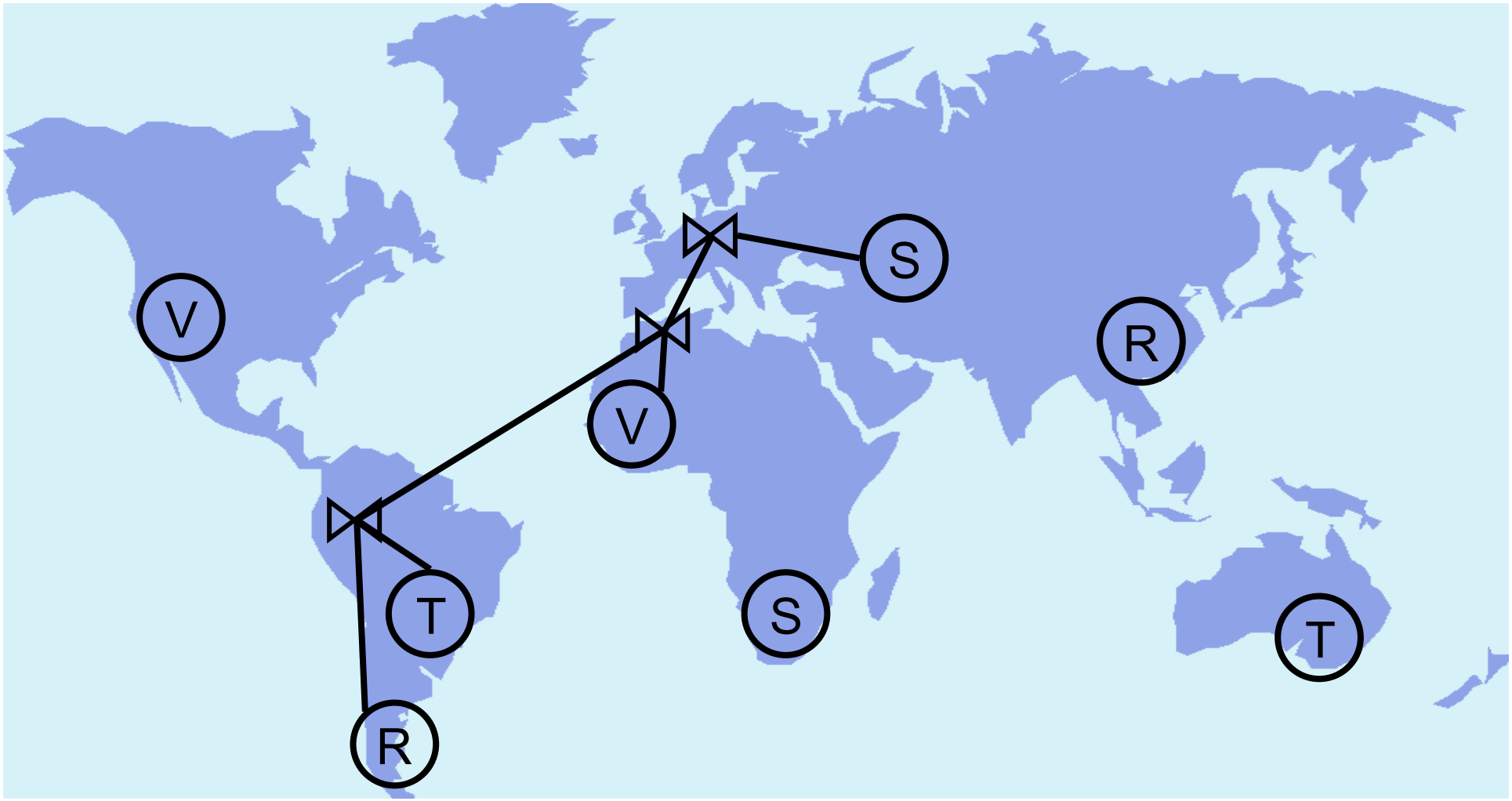
# Optimierung in VDBMSs: Replikation



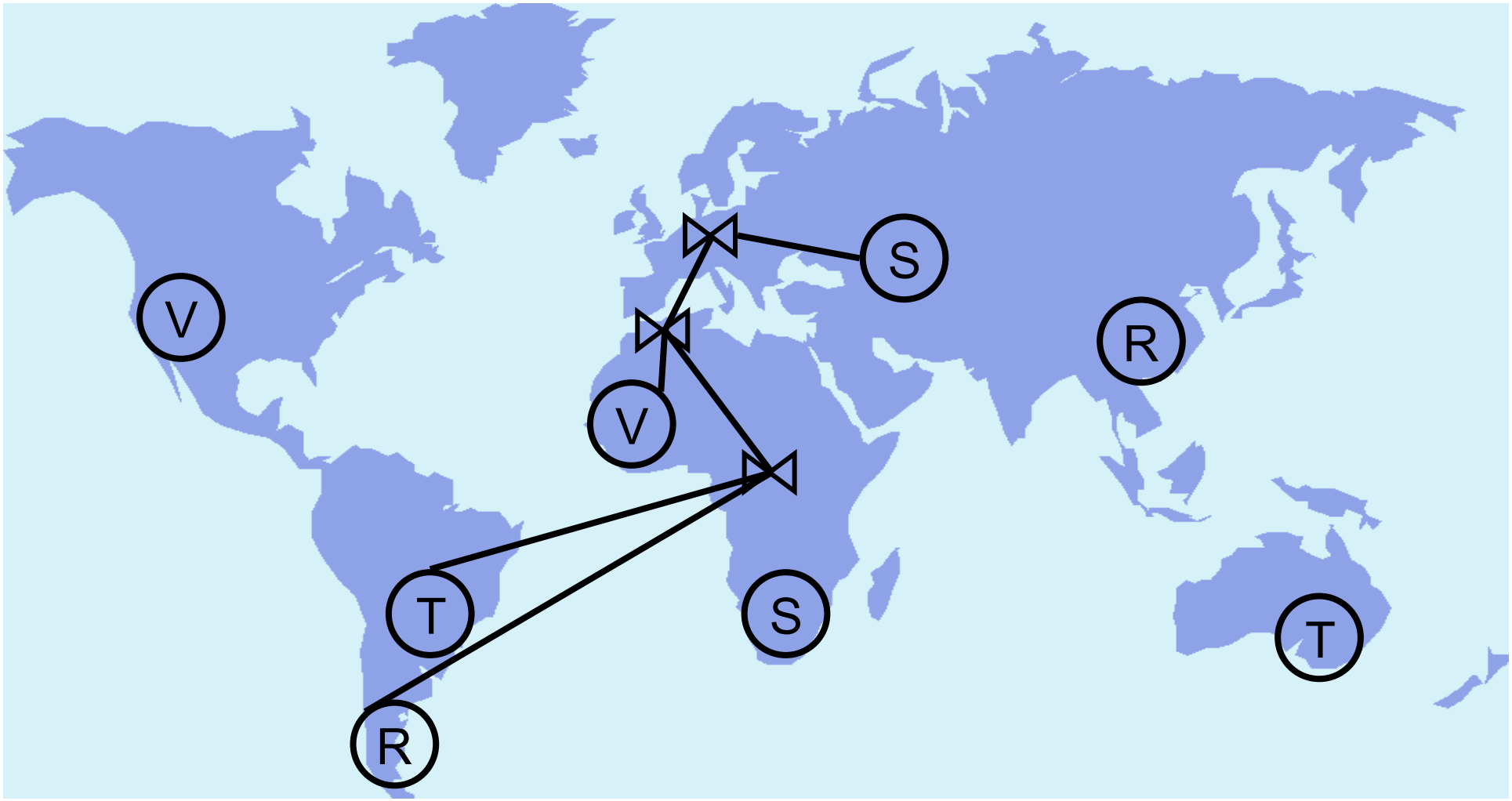
# Optimierung in VDBMSs: Join-Reihenfolge und Ausführungsort



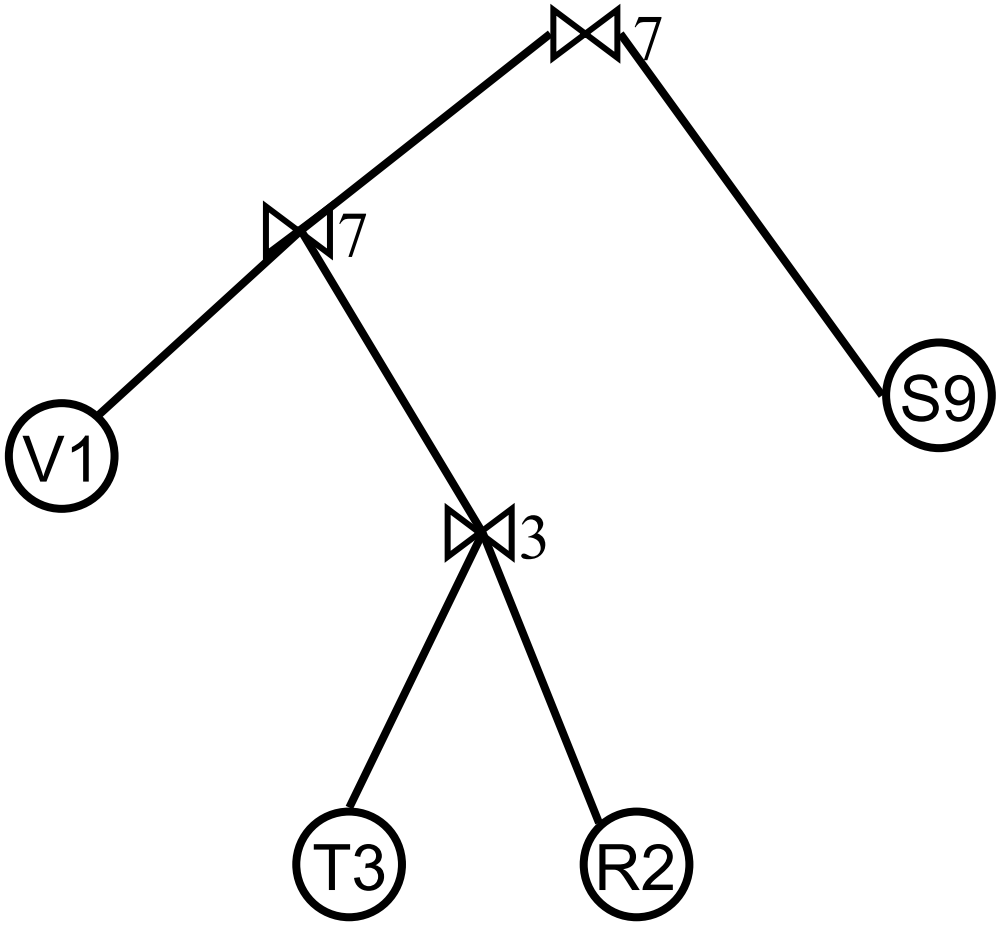
# Optimierung in VDBMSs: Join-Reihenfolge und Ausführungsort



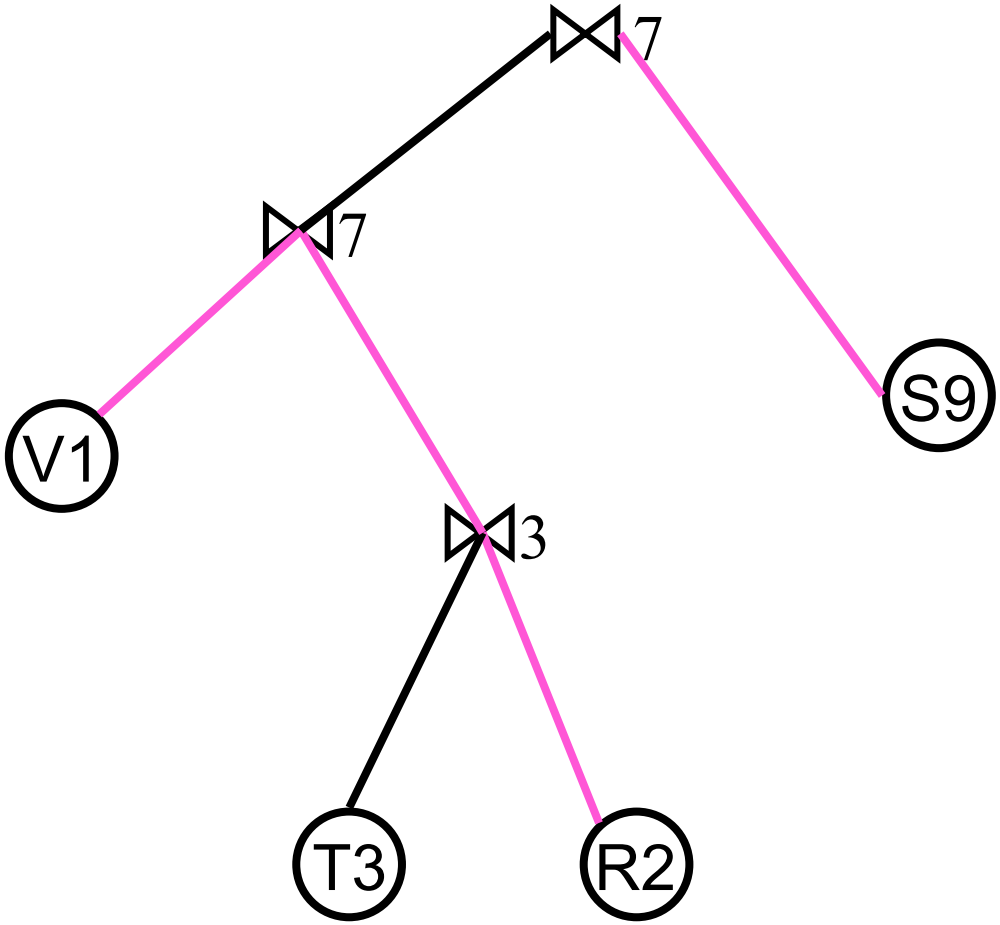
# Optimierung in VDBMSs: Join-Reihenfolge und Ausführungsort



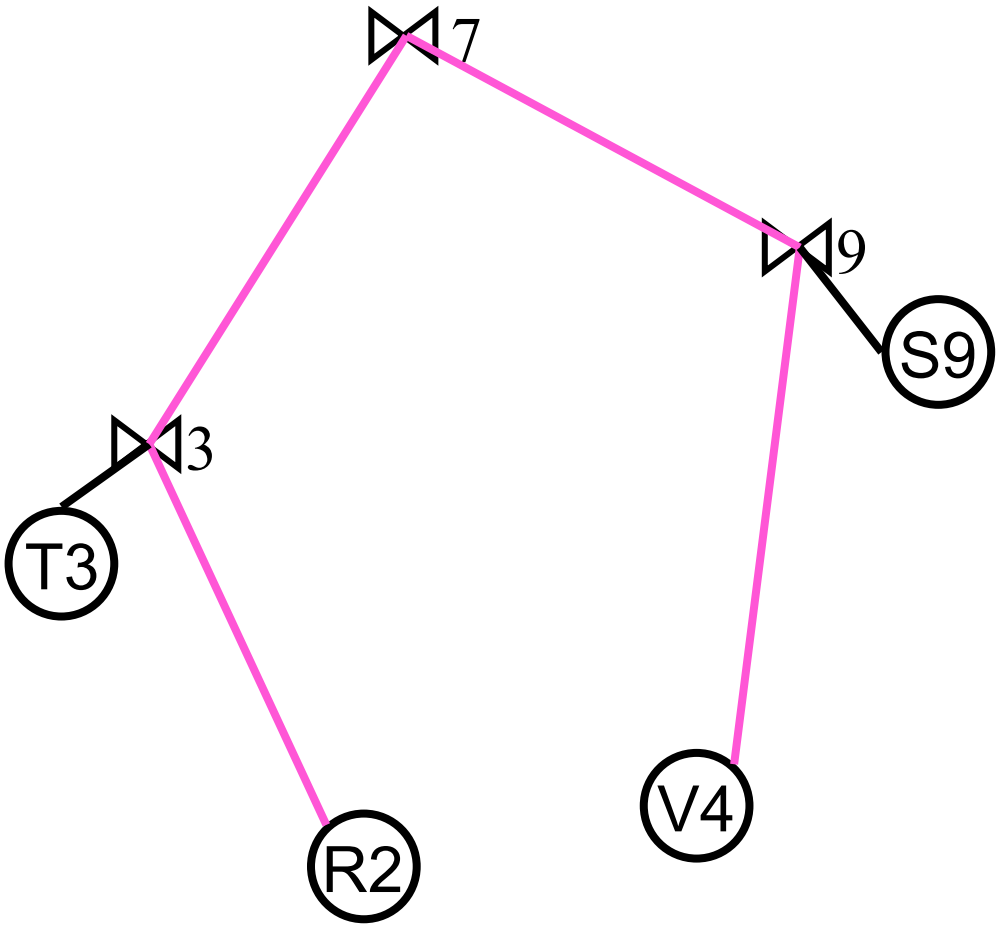
# Optimierung in VDBMSs



# Optimierung in VDBMSs



# Optimierung in VDBMSs



# Erweiterungen für Verteilte Datenbanken (2)

- Pruning eines Plans P1 wenn es einen semantisch äquivalenten Plan P2 gibt mit:

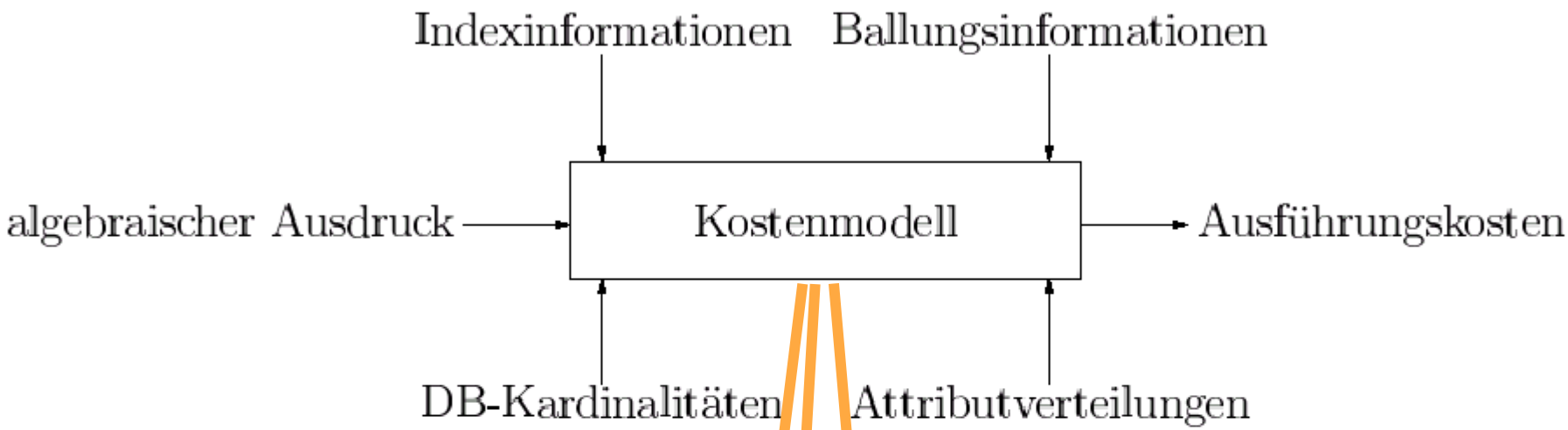
$$\forall i \in interesting\_sites(P_1) : \mathbf{cost}(ship(P_1, i)) \geq \mathbf{cost}(ship(P_2, i))$$

- Was kostet der Datentransfer zwischen den Knoten?
  - In einem homogenen Netzwerk kann man von gleichen Kosten zwischen allen Knoten ausgehen.
  - Plan P1 kann auf jeden Fall schon eliminiert werden, wenn man das Ergebnis von P2 billiger nach x schicken kann, wobei x der Knoten ist, an dem P1 sein Ergebnis generiert.

$$\mathbf{cost}(P_1) \geq \mathbf{cost}(ship(P_2, x))$$

- Zeitkomplexität: $O(s^3 * 3^n)$

# Kostenmodelle




Replikationsinformation

Allokationsinformation

Kommunikationskosten:
Bandbreite, Latenz zwischen
den Stationen

# Kostenmodelle

- In herkömmlichen Datenbanksystemen: Durchsatzoptimierung (throughput)
  - Aufwands-Kostenmodell W
- In verteilten Anwendungen: Antwortzeit-Optimierung (response time)
  - Antwortzeit-Kostenmodell T
- Beispielanfrage q mit optimalem Wo bzw To
  - Betrachte einen Auswertungsplan p für q
    - Wp bzw Tp
    - man sollte aber nicht beliebig viel Aufwand treiben, um Tp zu optimieren

# Zunächst: Überblick über Aufwandsabschätzung

- Selektivitätsabschätzung
- Kostenmodellierung der wichtigsten Operatoren
  - Nested Loops Join
  - Merge Join
  - Index Join
  - Hash Join

# Danach: Überblick über Antwortzeit-Kostenmodell

- Parallelverarbeitung (pipelining)
- Sequenzielle Ausführung (pipeline-breaker)

# Beispiel-Datenbankschema

- Attribute, die Teil des Primärschlüssels sind, sind unterstrichen.

Employee

| Fname | Lname | SSN | BDate | Adress | Sex | Salary | SuperSSN | DNO |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

Department

| Dname | Dnumber | MgrSSN | MgrStartDate |
|---|---|---|---|
| | | | |

DeptLocations

| Dnumber | Dlocation |
|---|---|
| | |

Project

| Pname | Pnumber | Plocation | Dnum |
|---|---|---|---|
| | | | |

WorksOn

| ESSN | PNO | Hours |
|---|---|---|
| | | |

# Selektivität

- Die Selektivität eines Suchprädikats schätzt die Anzahl der qualifizierenden Tupel relativ zur Gesamtanzahl der Tupel in der Relation.
- Beispiele:
  - die Selektivität einer Anfrage, die das Schlüsselattribut einer Relation *R* spezifiziert, ist 1/ #*R*, wobei #*R* die Kardinalität der Relation *R* angibt.
  - Wenn ein Attribut *A* spezifiziert wird, für das *i* verschiedene Werte existieren, so kann die Selektivität als

    $$(\#R/i) \,/\, \#R \quad \text{oder} \quad 1/i$$

    abgeschätzt werden.

# Join-Selektivität

- Join von $R$ mit $S$:

$$sel_{RS} := \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$

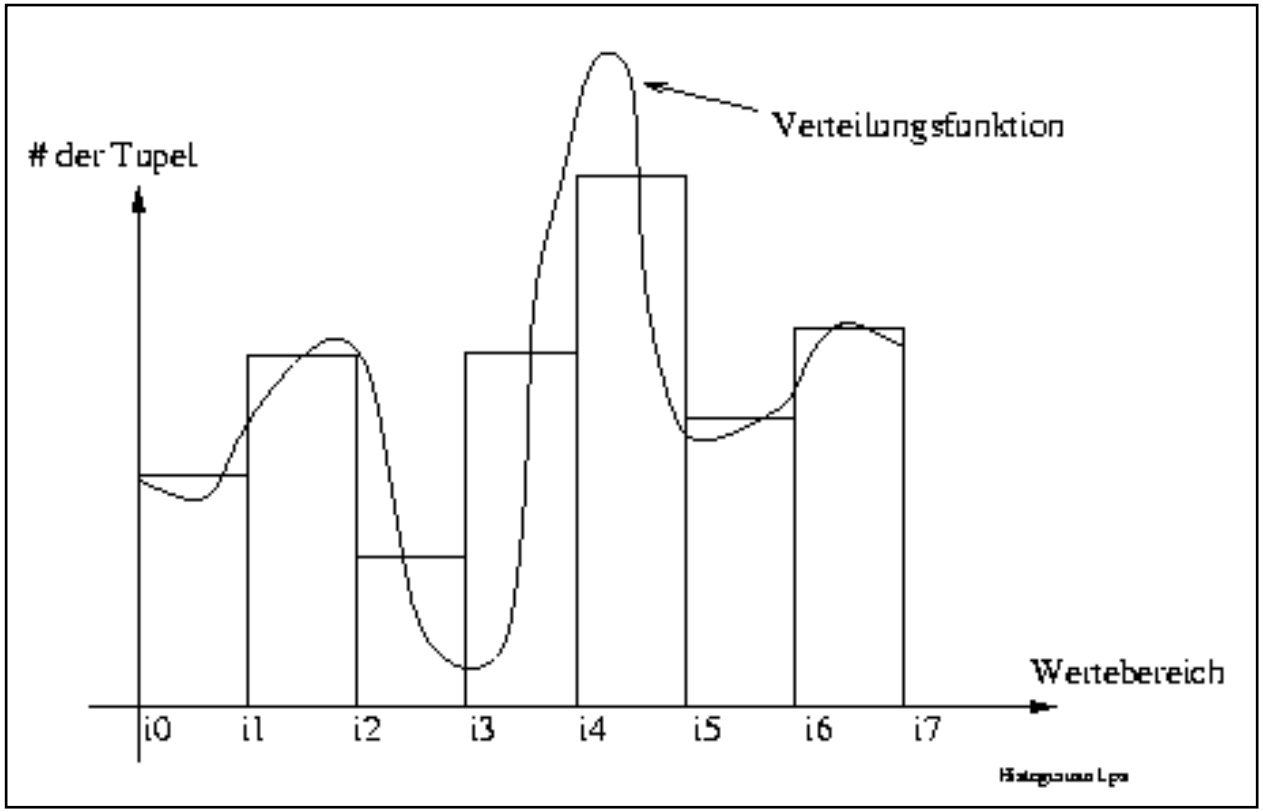
Abschätzung der Selektivität:

- $sel_{R.A=C} = \frac{1}{|R|}$
  falls $A$ Schlüssel von $R$
- $sel_{R.A=C} = \frac{1}{i}$
  falls $i$ die Anzahl der Attributwerte von $R.A$ ist (Gleichverteilung)
- $sel_{R.A=S.B} = \frac{1}{|R|}$
  bei Equijoin von $R$ mit $S$ über Fremdschlüssel in $S$

Ansonsten z.B. Stichprobenverfahren

# Systematische Kostenabschätzung

Histogrammverfahren: Beispiel:




- In den Intervallen [*i*3,*i*4] und [*i*4,*i*5] relative schlechte Abschätzung (große Abweichung Verteilung zu Histogramm).

# Systematische Kostenabschätzung

## Histogrammverfahren:

- Abhilfe kann durch nicht äquidistante Unterteilung geschaffen werden: Gleiche Höhe statt gleiche Breite.
- Vorteil:
  - Fehler hängt nicht mehr von der Verteilung ab
- Nachteil:
  - Erstellung ist teurer, da die Attributwerte sortiert werden
  - Fortschreiben unter Erhalten der gleichen Höhe ist kaum möglich, stattdessen i.d.R. Neuerstellung des Histogramms in periodischen Zeitabständen

# Kostenabschätzung für Block-Nested Loop Join (BNL)

```
foreach  r ∈ R
 foreach s ∈ S
   if s.B = r.A  then Res:=Res ∪ (r ∘ s)
```

Günstigste Realisierung:

- $n_B$-1 Rahmen für die äußere Schleife ($n_B$: Zahl der Pufferrahmen)
- 1 Rahmen für die innere Schleife

```
repeat:
  lese n_B-1 Blöcke der Relation R in HashTabelle
  ein
  repeat:
     lese 1 Block der Relation S
     vergleiche jedes Hauptspeicher-Tupel aus R
      mit jedem Hauptspeicher-Tupel aus S und
      verbinde sie gegebenenfalls
  until: alle Blöcke aus S sind gelesen
until: alle Blöcke aus R sind gelesen
```

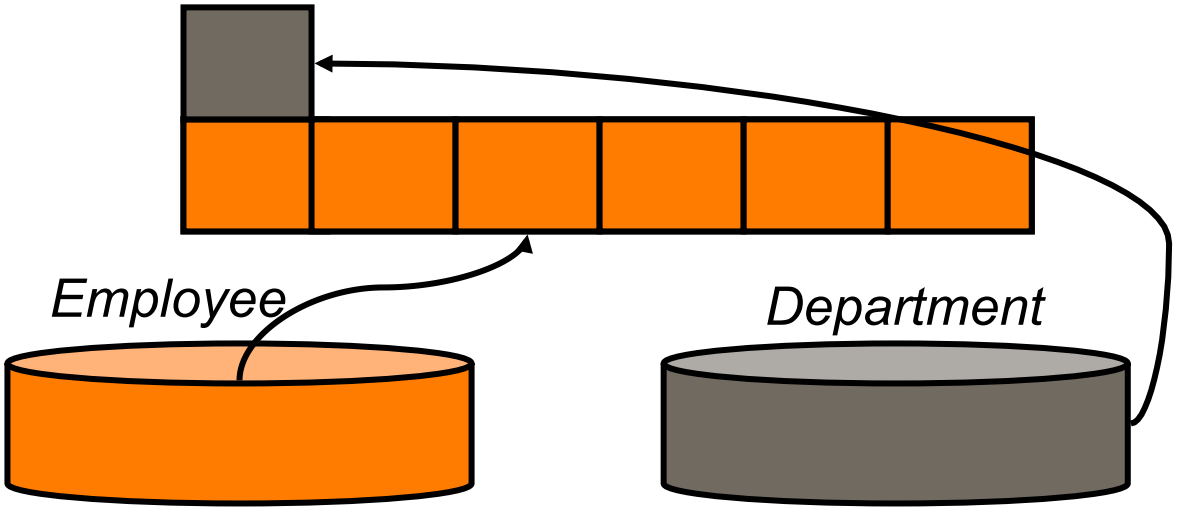
# Kostenabschätzung für BNL-Join

- Seitenzugriffe ist das „Maß der Dinge“
- Allerdings haben neuere empirische Arbeiten gezeigt, dass CPU-Kosen nicht zu vernachlässigen sind
- Auch muss man unbedingt zwischen „random IO“ und „sequential/chained IO“ differenzieren
- Beispiel:

  $Employee \otimes_{DNO=Dnumber} Department$

  - $\#E = 5000$ Tupel auf $b_E = 2000$ Seiten verteilt
  - $\#D = 50$ Tupel auf $b_D = 10$ Seiten verteilt
  - $n_B = 6$ Pufferrahmen

# Kostenabschätzung für BNL-Join




- Es werden $b_E$ Seiten in der äußeren Schleife eingelesen
- Es wird $\lceil b_E / (n_B\text{-}1) \rceil$ mal *Department* neu eingelesen
- Damit werden

$$b_D * \lceil b_E / (n_B\text{-}1) \rceil$$

  Seiten in der inneren Schleife eingelesen
- Also werden

  $b_E + \lceil b_E / (n_B\text{-}1) \rceil * b_D = 2000 + \lceil 2000 / 5 \rceil * 10 = 6000$

  Seiten gelesen

# Kostenabschätzung für BNL-Join

Vertauschung der Join-Reihenfolge:

$$Department \otimes_{DNO=Dnumber} Employee$$

Kosten:

$$b_D + \lceil b_D / (n_B\text{-}1) \rceil * b_E = 10 + \lceil 10 / 5 \rceil * 2000 = 4010$$

- Man kann also durch Vertauschen der Join-Reihenfolge erhebliche Kosten einsparen.
- Grundsätzlich bei BNL-Join: die von der Seitenzahl her kleinere Relation nach außen verlagern.
- Für die innere nur 1 Seite (müssen aber große sein)

# Kostenabschätzung für Index-NL-Join

- Beispiel: $Employee \otimes_{SSN = MgrSSN} Department$ ($V_1$)
- Annahme: Primärindex für *Employee.SSN* und Sekundärindex für *Department.MgrSSN*
- INL-Join kann also in beiden Richtungen ausgeführt werden, also auch:
  $Department \otimes_{MgrSSN = SSN} Employee$ ($V_2$)
- Zusatzannahme: Indexdatei physisch in B-Baum realisiert.
- Indexhöhe (≈ Höhe des B-Baums)
  - $x_{SSN} = 4$
  - $x_{MgrSSN} = 2$
- Selektivitäten:
  - *Employee.SSN*: 1 Satz pro SSN
  - *Department.MgrSSN*: im Mittel $n$ Sätze pro MgrSSN
- Kosten für $V_1$:
  $b_E + (\#E * (x_{MgrSSN}+n)) = 2000 + 5000 * (2+n) = 12000 + 5000n$
- Kosten für $V_2$:
  $b_D + (\#D * (x_{SSN}+1)) = 10 + 50 * 5 = 260$

# Kostenabschätzung für Merge-Join

- Beide Relationen müssen geordnet sein.
- Sonst evtl. Sortierung vorschalten (mit entsprechenden Kosten)
- Sortierung erlaubt dann im günstigsten Fall (Join-Attribut in mindestens einer der beiden Relationen „unique“) „single pass“-Merge-Lauf.
- Also werden in unserem Beispiel

  $b_E + b_D = 2000 + 10$

  Seiten gelesen.

# Kostenabschätzung für Hash Join

- Hash-Join gilt als sehr effizient.
- Schwierige analytische Kosenabschätzung (insb. Hybrid hash join).
- q ist der Anteil von R, der in den Hauptspeicher passt

$$b_{R_1} + b_{R_2} + 2 * (b_{R_1} + b_{R_2}) * (1 - q)$$

Memory size

Fudge Factor

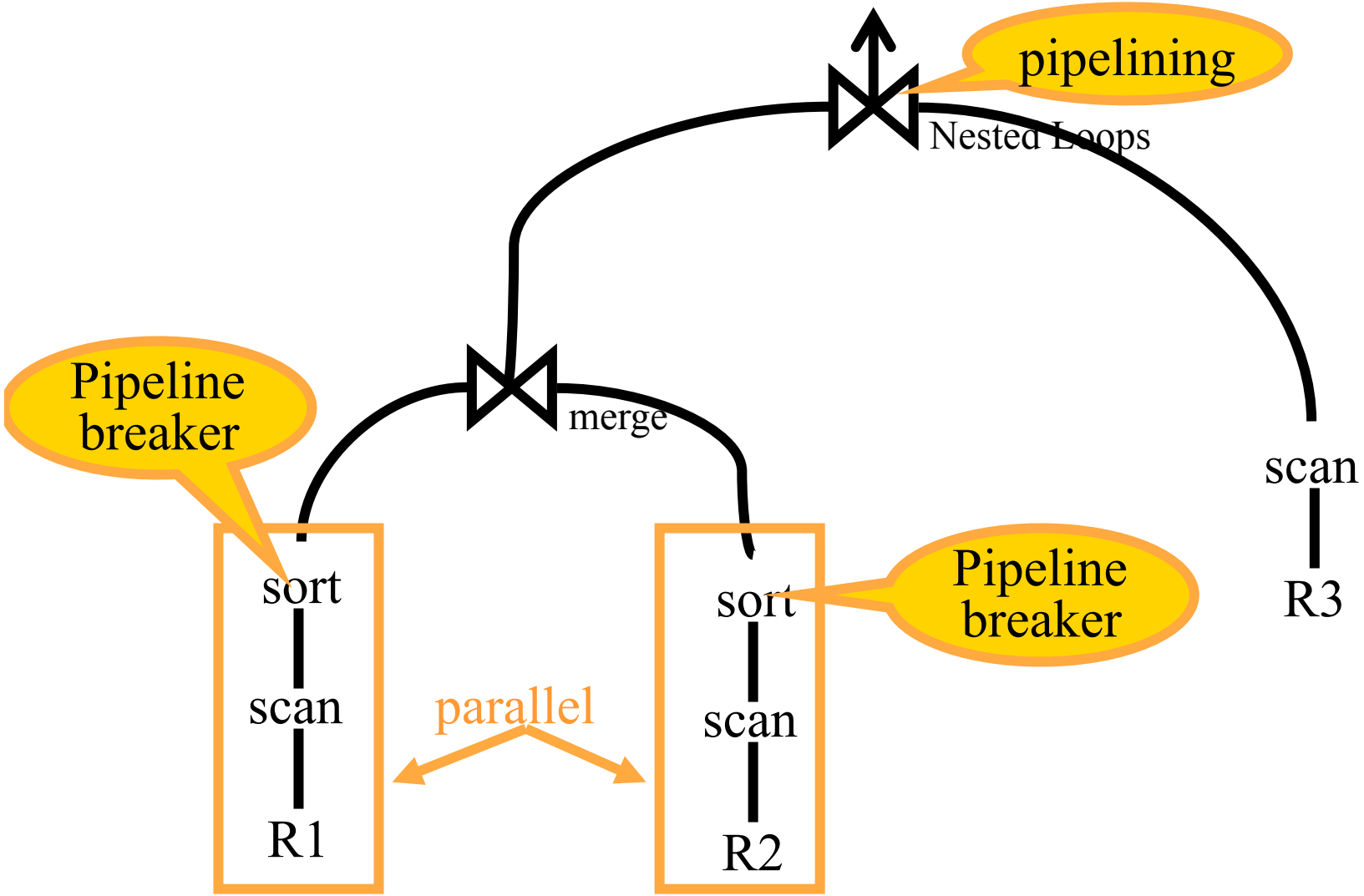
$$q = \frac{ms - \left\lceil \frac{1.4 * b_{R_1} - ms}{ms - 1} \right\rceil}{b_{R_1}}$$

- Literatur: Steinbrunn, Moerkotte, Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. VLDB Journal, Vol 6, No 3, Aug 1997. online

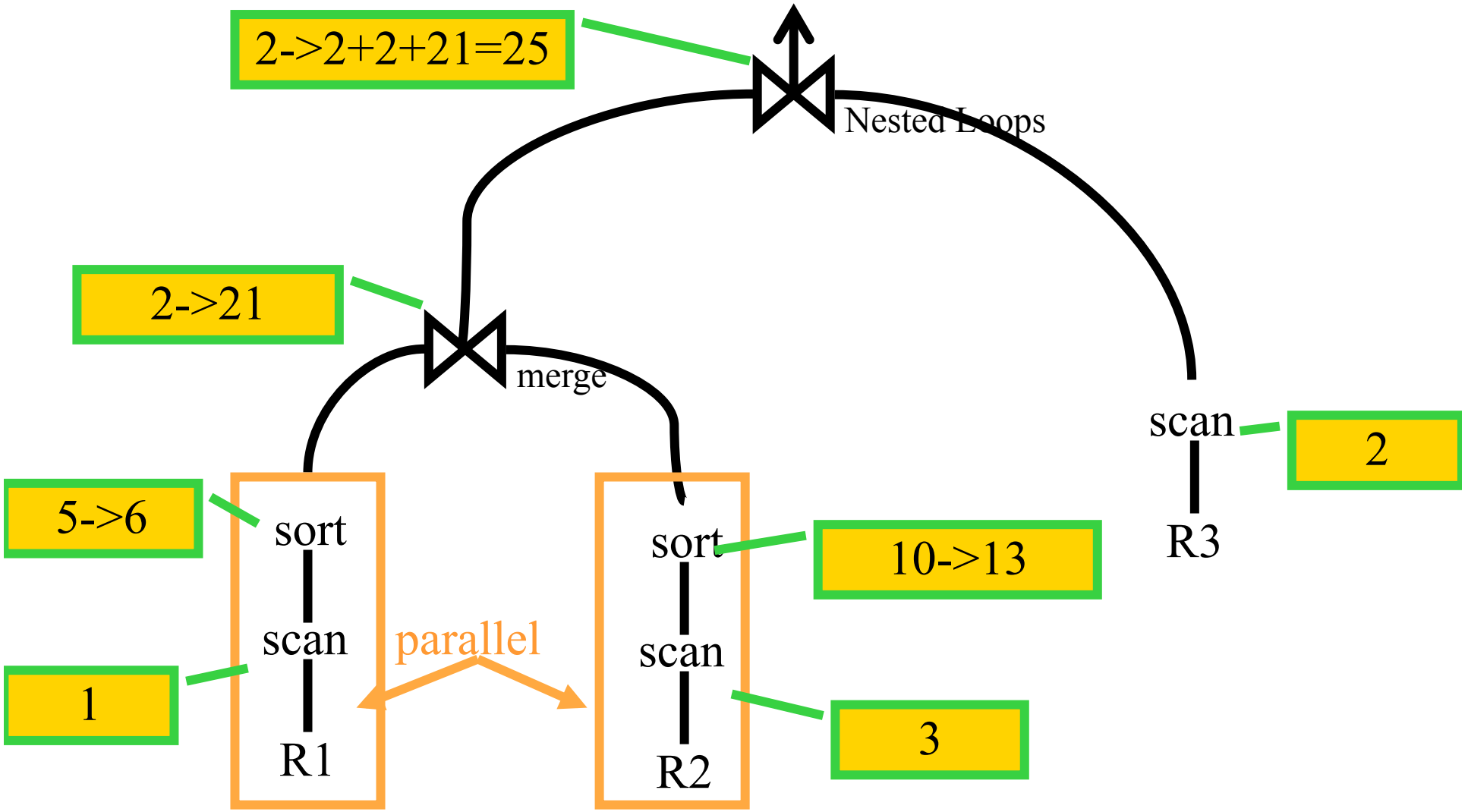
# Strategien bei der Optimierung für Parallele Ausführung

- Beispielanfrage q mit optimalem Wo bzw To
  - Betrachte einen Auswertungsplan p für q
    - Wp bzw Tp
- man sollte aber nicht beliebig viel Aufwand treiben, um Tp zu optimieren! 2 Strategien:
  - Limitiere Durchsatz-Degradierung
    - Wp <= k * Wo muss gelten
    - sonst: Tp := ∞
  - Kosten/Nutzen-Verhältnis muss stimmen
    - (To - Tp)/(Wp - Wo) <= k muss gelten
    - sonst: Tp := ∞

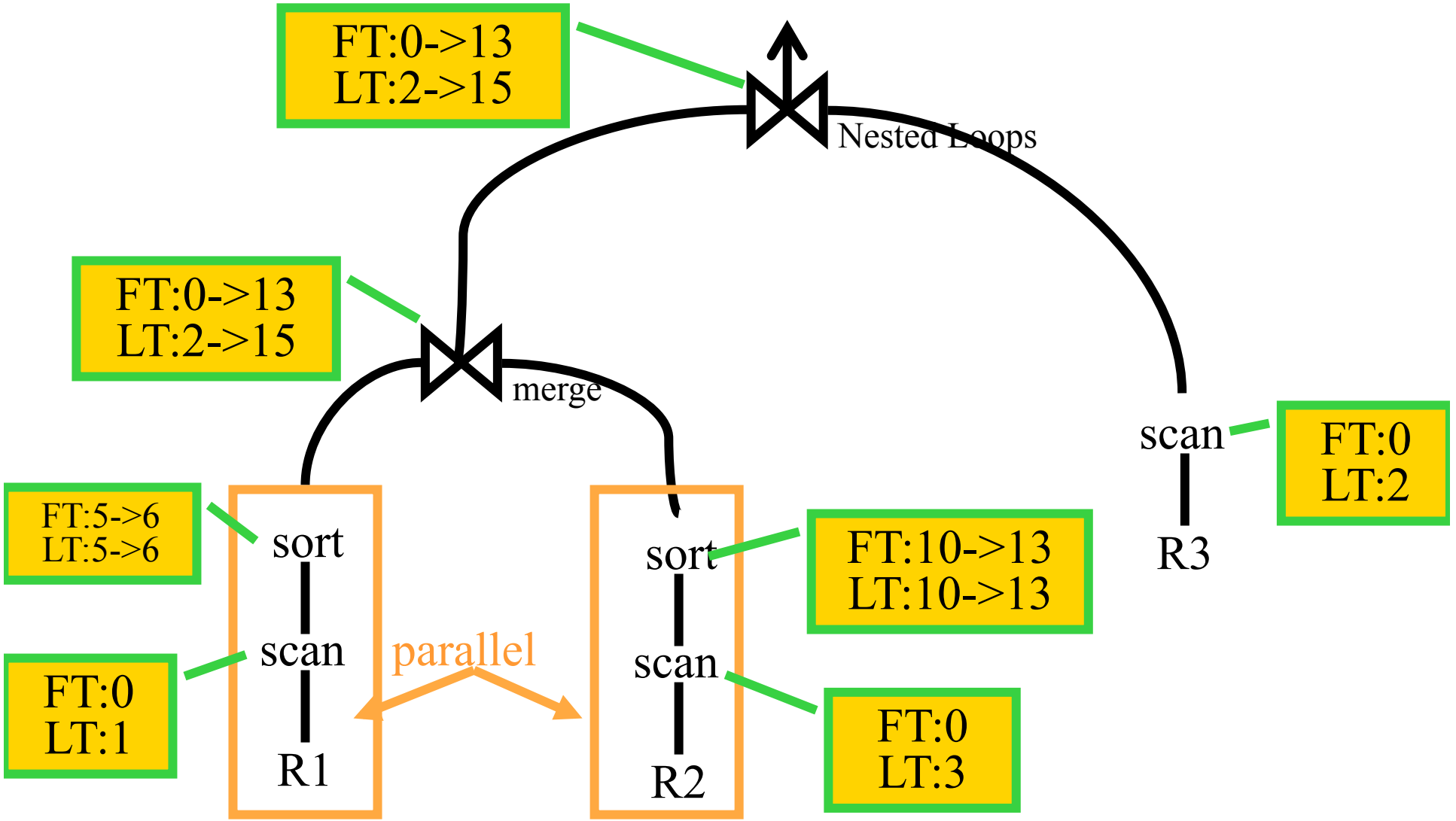
# Beispielanfrage



# Beispielanfrage: Kosten nach Aufwandsabschätzung



# Beispielanfrage: Deskriptoren First Tuple/Last Tuple



# Antwortzeit-Abschätzung eines Operatorbaums

- t1||t2 schätzt die RT (response time) zweier unabhängiger paralleler Ausführungseinheiten S1 und S2
  - t1||t2 := max(t1,t2)
- t1;t2 schätzt die RT zweier sequentieller Ausführungseinheiten
  - t1;t2 := t1+t2
- Pipeline S1 wird „gefüttert" von einer materialisierten Sub-Query S2. Die Pipeline S1⊖S2 braucht demnach
  - t1 ⊖ t2 := t1 - t2

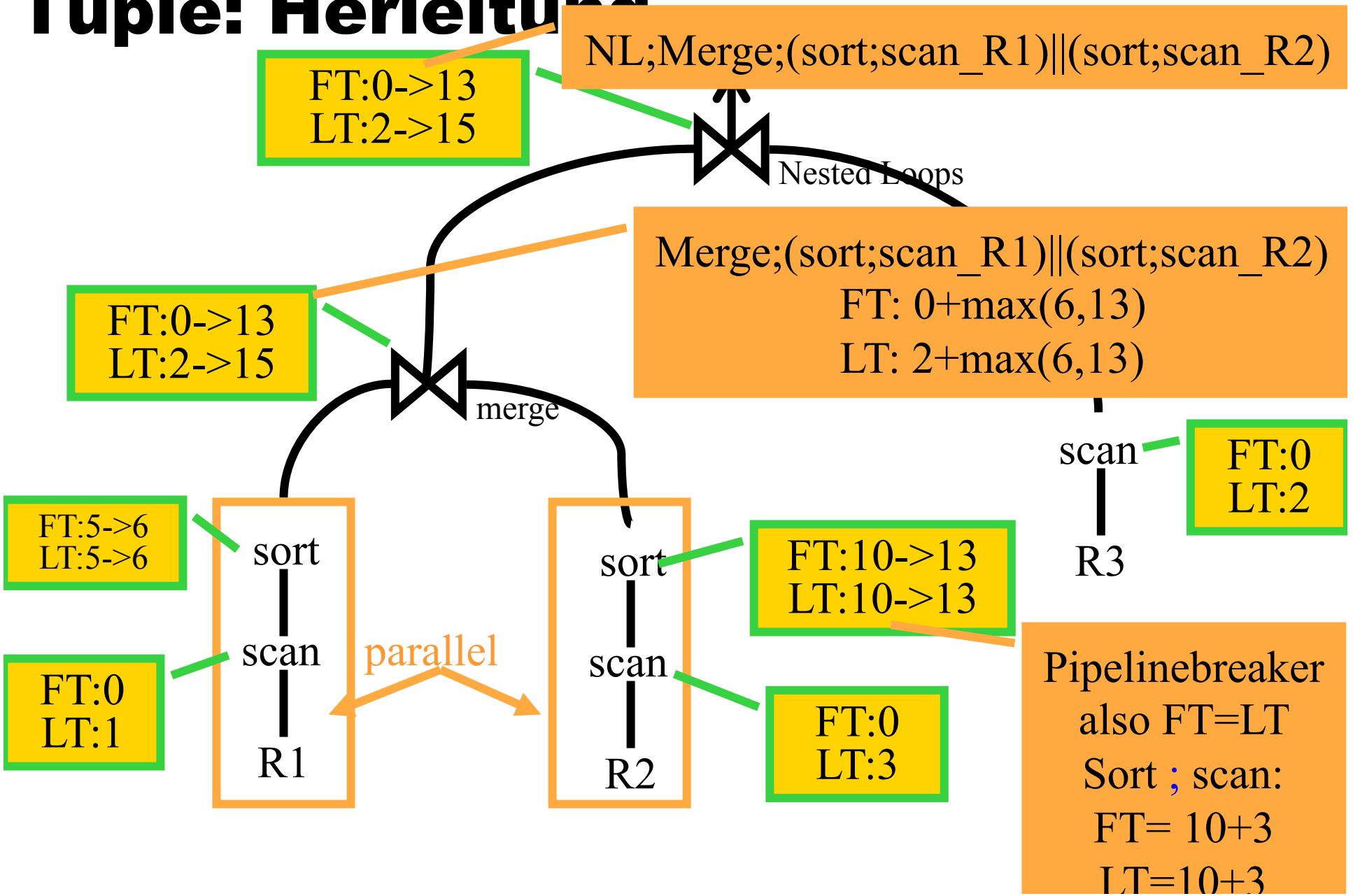
# Antwortzeit-Abschätzung einer Pipeline

- Produzent P mit Deskriptor (Pf,Pl)
- Konsument C mit Deskriptor (Cf,Cl)
- Der Antwortzeit-Deskriptor (Tf,Tl) der Pipeline T = „P ---> C" ergibt sich dann wie folgt
  - Tf = (Pf;Cf) := Pf +Cf
  - Tl = (Pf; Cf; ((Pl ⊖ Pf)||(Cl ⊖ Cf))

Max(...)

die langsamere der beiden Pipelines bestimmt die Antwortzeit

# Deskriptoren First Tuple/Last Tuple: Herleitung



# Erweiterung des Kostenmodells:
## Ressourcen haben beschränkte Bandbreite

**Idee:** Protokolliere Ressourcenverbrauch mit. Anstatt einfach nur Antwortzeit wird für jeden Operator noch in einem Ressourcevektor festgehalten, wieviel er von jeder Ressource verbraucht hat:

$\vec{r} = (\text{CPU}_{NY}, \text{CPU}_{DC}, \text{CPU}_{PA}, \text{Netz})$

Nun überlade $\|$, ;, $\ominus$ so daß sie komponentenweise auf die Ressourcevektoren $\vec{r}$ und Antwortzeiten angewendet werden können:

Sequentielle Ausführung:

$$t_1; t_2 = t_1 + t_2$$

$$\vec{r_1}; \vec{r_2} = \vec{r_1} + \vec{r_2}$$

Pipeline Ausführung:

$$t_1 \ominus t_2 = t_1 - t_2$$

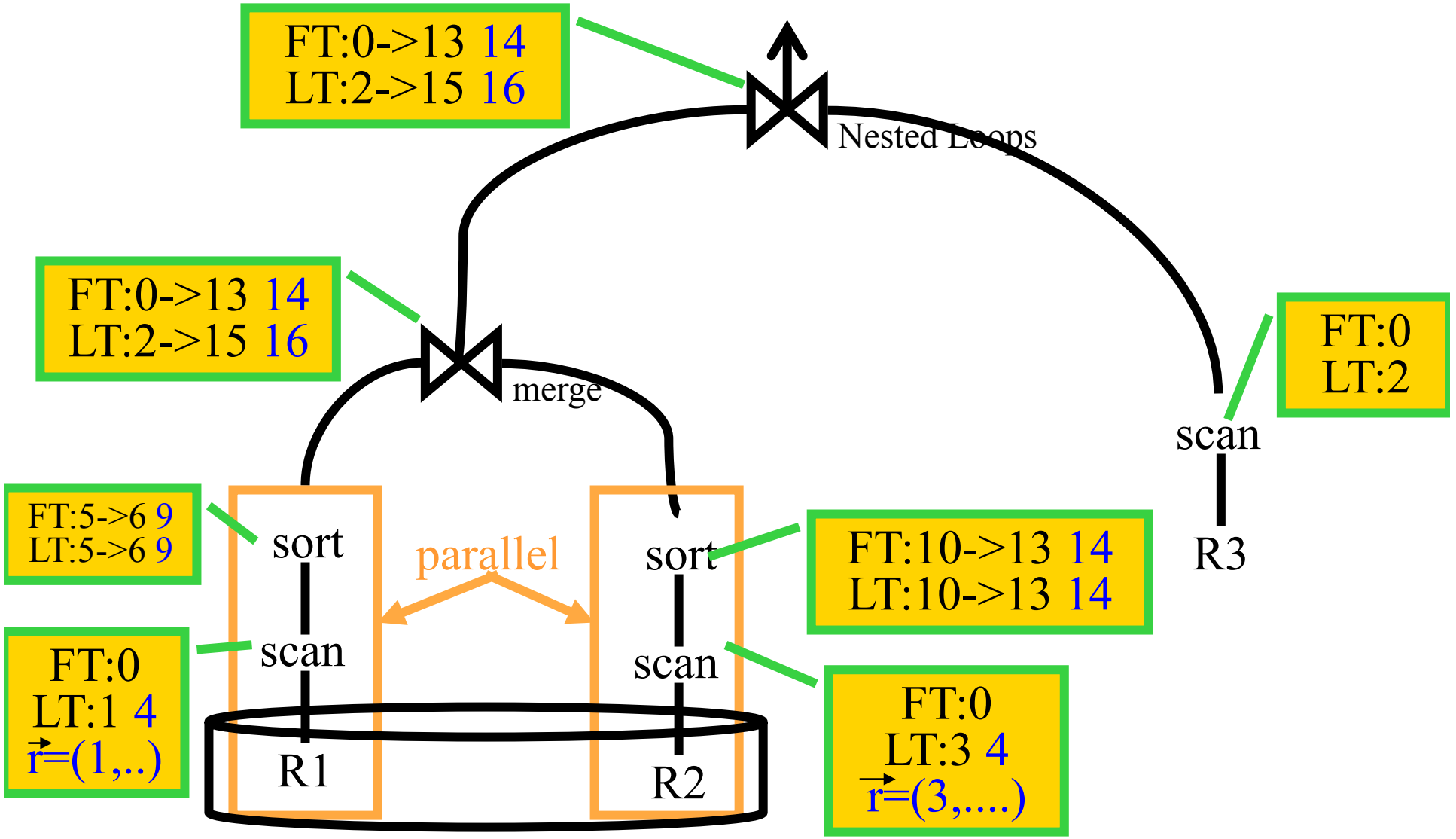
$$\vec{r_1} \ominus \vec{r_2} = \vec{r_2} - \vec{r_1}$$

Unabhängig parallele Ausführung:

$$t_1 \| t_2 = max(t_1, t_2, max_i(r_1^i + r_2^i))$$

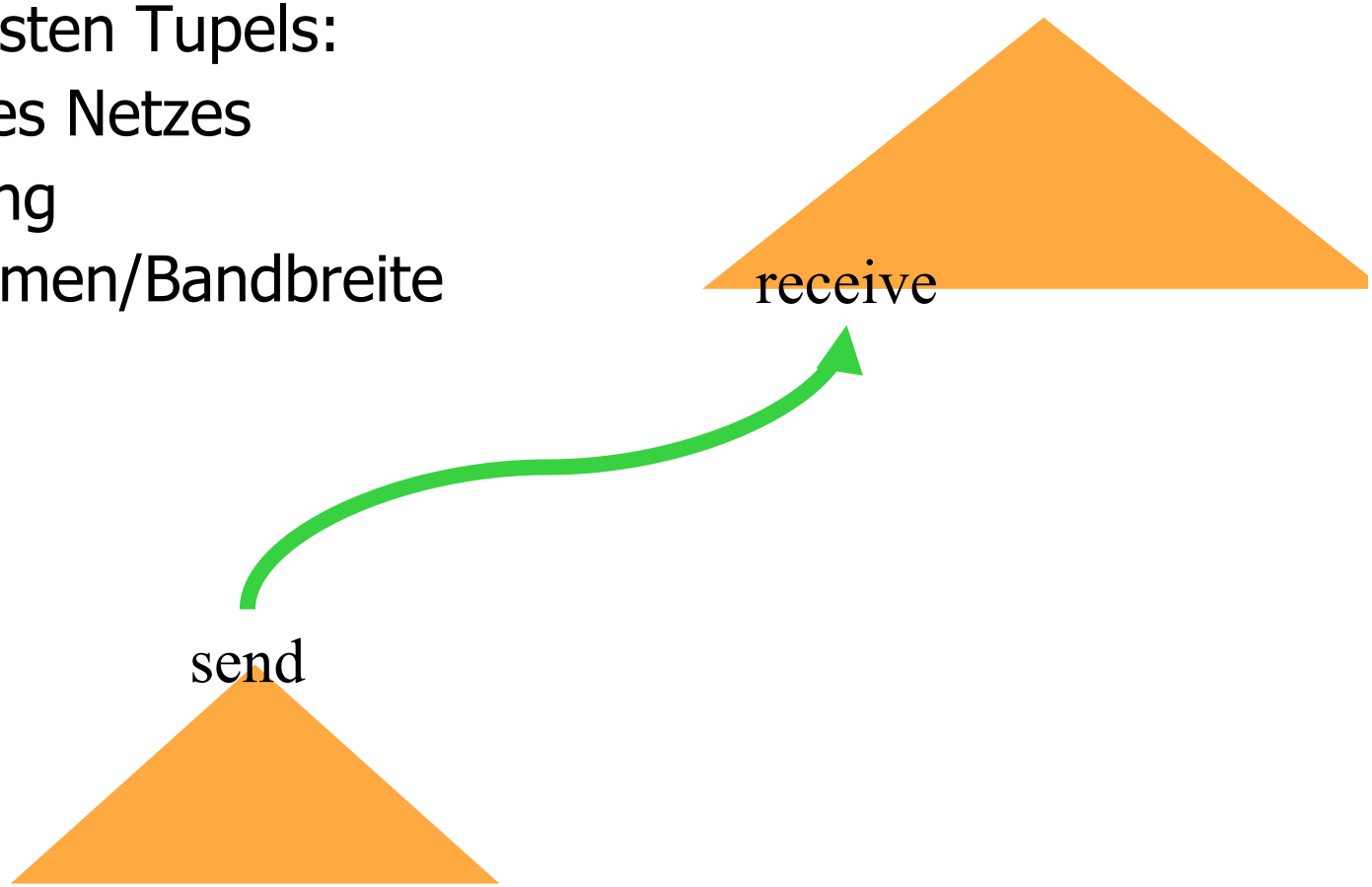
$$\vec{r_1} \| \vec{r_2} = \vec{r_1} + \vec{r_2}$$

# Resource Contention: R1 und R2 auf derselben Platte

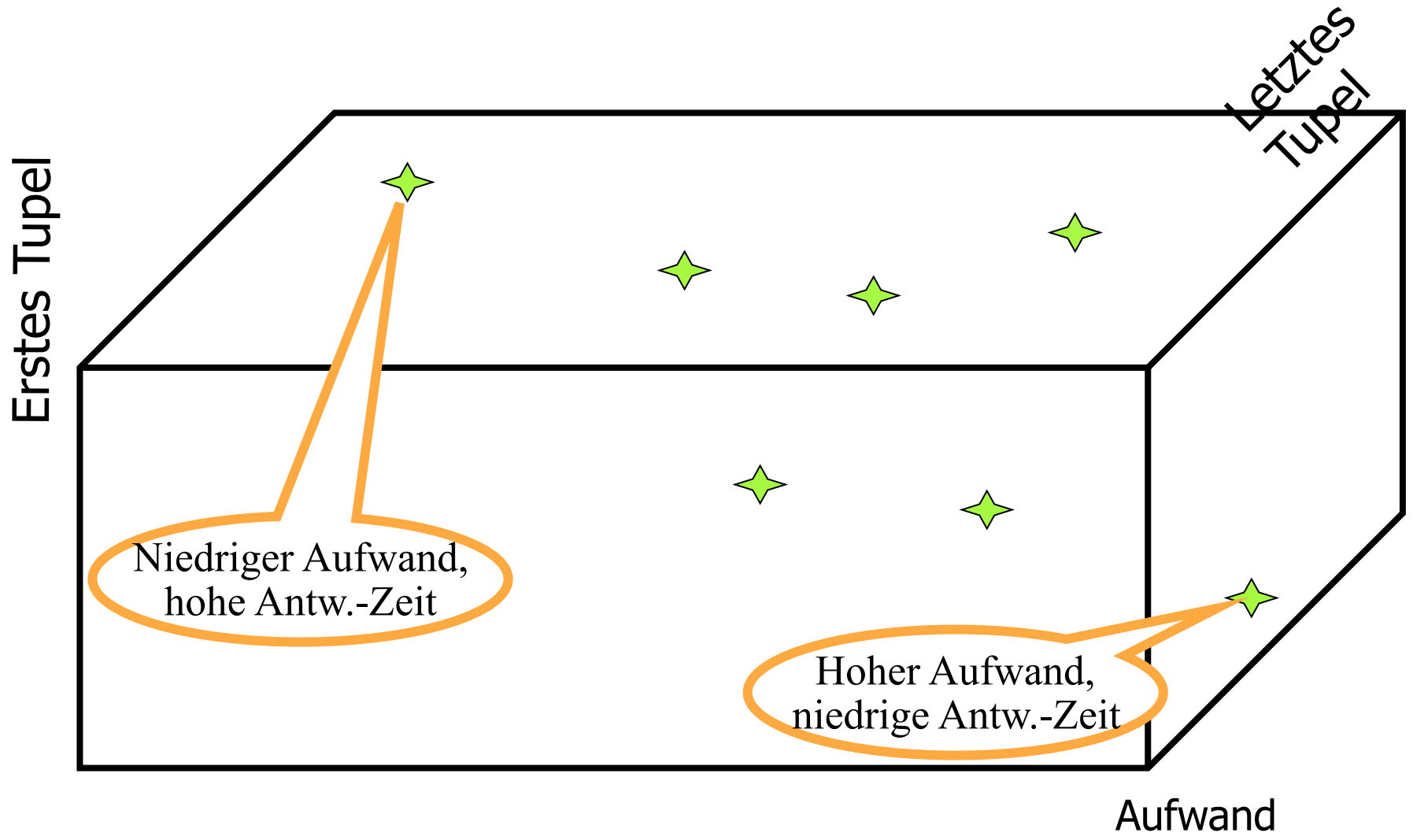


# Einbeziehung der Kommunikationskosten

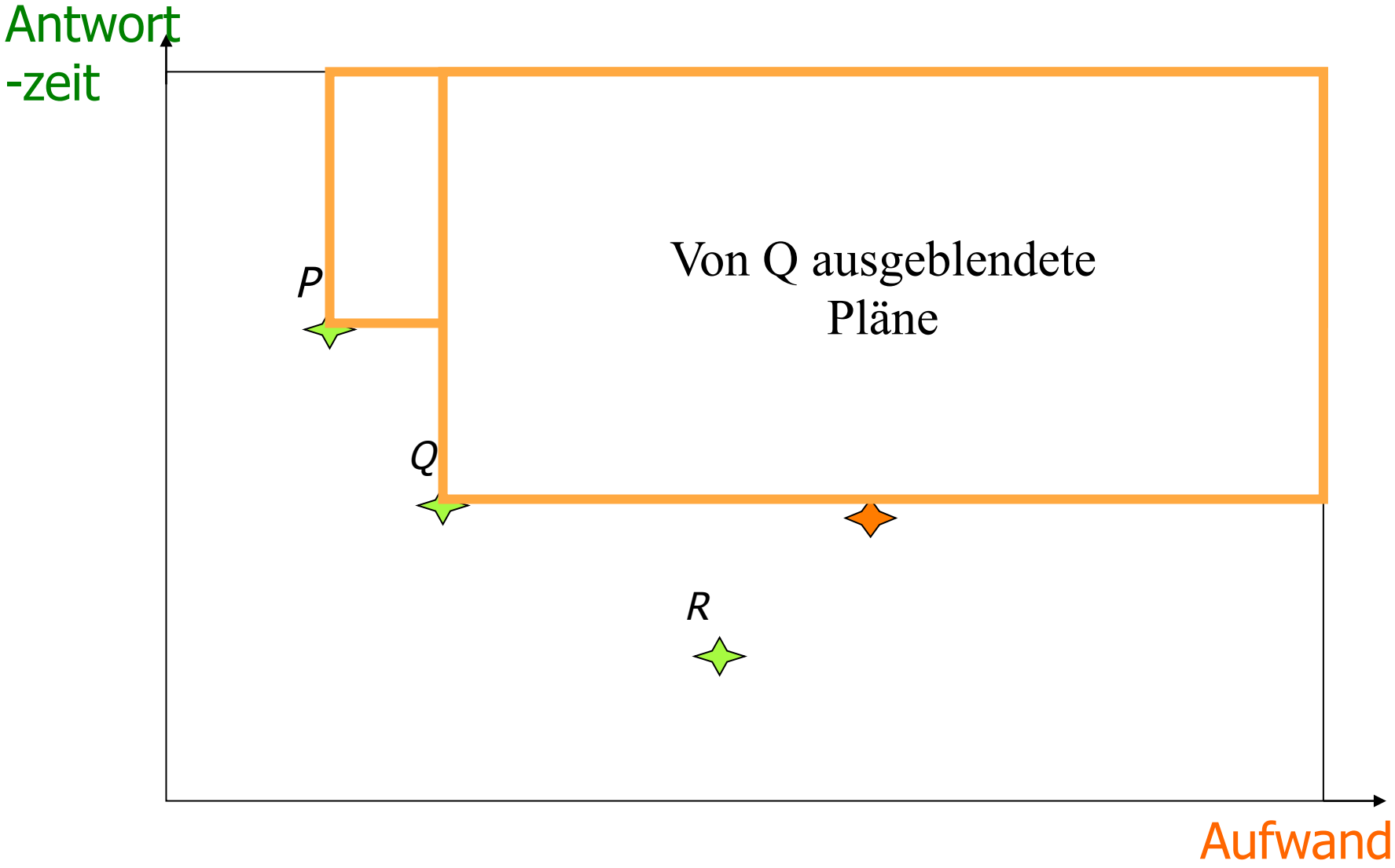
- Betrachte Send/Receive-Iteratoren als „ganz normale“ Operatoren
- Schicken des ersten Tupels:
  - Latenzzeit des Netzes
- Danach Pipelining
  - Transfervolumen/Bandbreite



# Mehrdimensionales Kostenmodell



# Anfrageoptimierung: partielle Ordnung der Pläne (QEPs)



# Anfragebearbeitung in heterogenen Multi-Datenbanken

Teil2, ab S. 96