

# 5. Verteilte und parallele Query-Bearbeitung

- Einführung
- verteilte Anfragebearbeitung: Teilschritte
  - Anfragetransformation
  - Daten-Lokalisierung
  - globale Optimierung
- verteilte / parallele Selektion, Projektion, Aggregationen
- verteilte Join-Verarbeitung
  - einfache Strategien: Ship Whole, Fetch as Needed
  - Semi-Join
  - Bitvektor-Join
- parallele Join-Verarbeitung
  - dynamische Replikation (Fragment and Replicate)
  - dynamische/Statische Partitionierung der Eingaberelationen
  - paralleler Hash-Join
  - Mehrwege-Joins
- parallele Sortierung



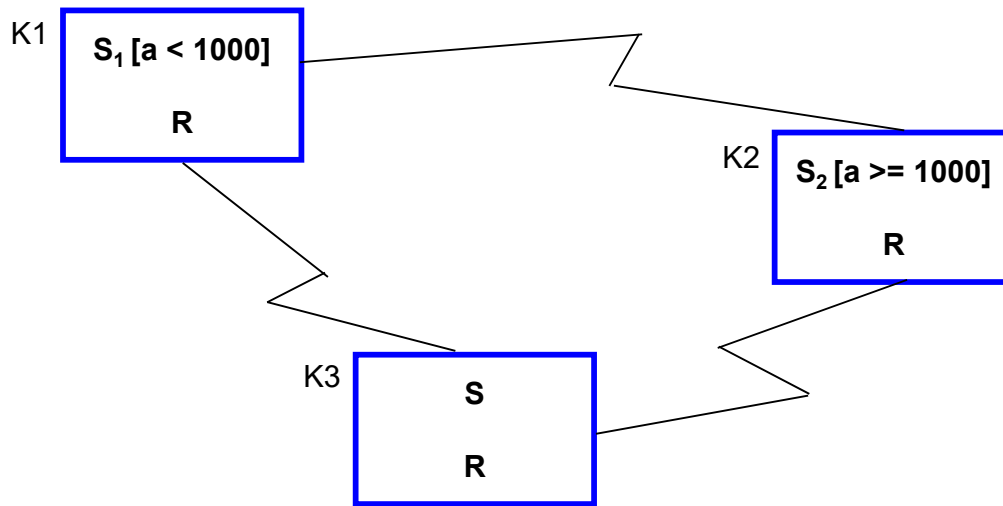
## Problemstellung

Bestimmung eines Ausführungsplanes in Abhängigkeit zur Datenverteilung, so dass eine Zielfunktion optimiert wird (z. B. Antwortzeit, Overhead)

- Kostenfaktoren:
  - Nachrichtenanzahl, Nachrichtengröße
  - E/A, CPU-Bedarf, Hauptspeicherbedarf
- Optimierungsentscheidungen:
  - Query-Zerlegung in lokal ausführbare Teilanfragen
  - Ausführungsreihenfolge für Restriktion, Projektion und Join
  - Parallelisierung von Teilanfragen
  - Auswahl der globalen und lokalen Join-Strategie
  - Rechnerauswahl, z.B. zur Join-Berechnung
  - Auswahl der Replikate
- Berücksichtigung des aktuellen Systemzustandes zur Laufzeit wünschenswert



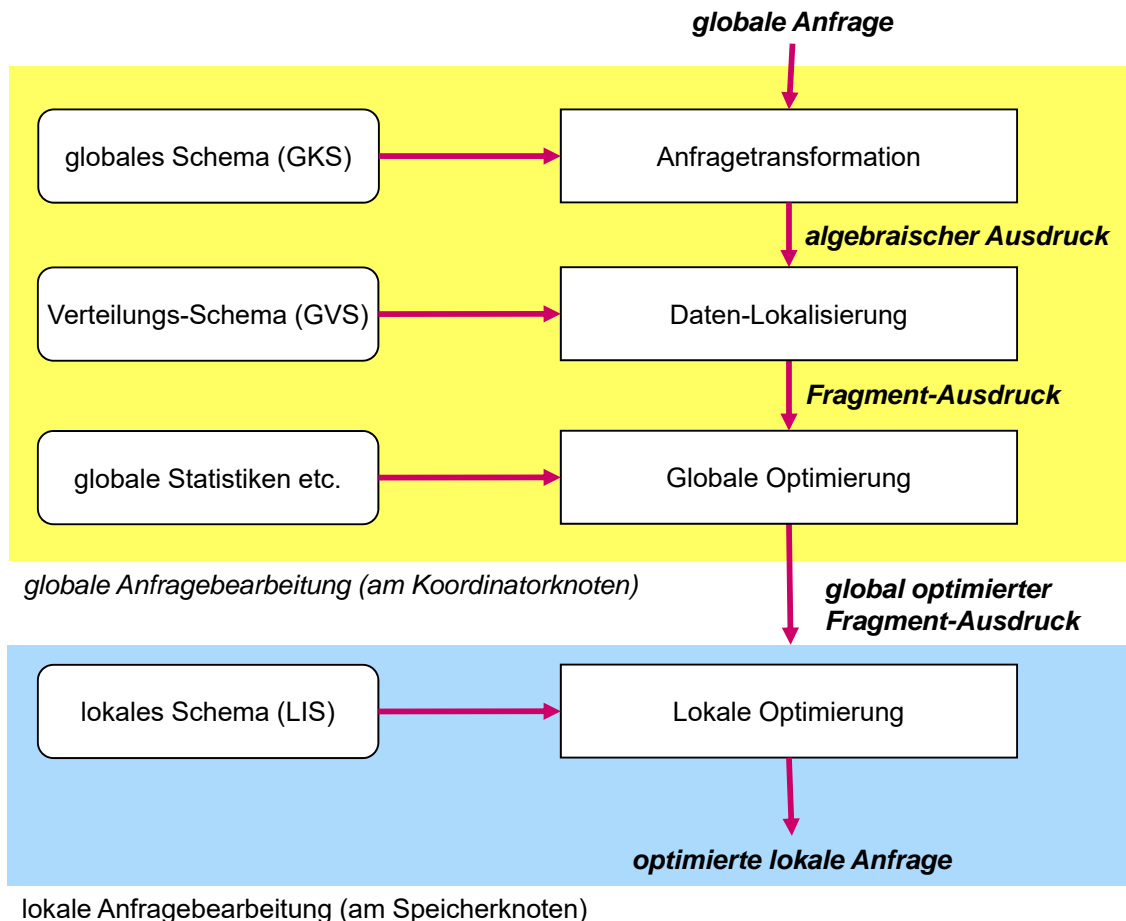
# Query-Optimierung: Beispiel



- Query in K3: *SELECT \* FROM S WHERE a IN (482, 517, 763);*
  - lokale Ausführung in K3 oder
  - Ausführung in K1 mit (kleinerem) Fragment S1
- Query in K2: *SELECT x, y, z FROM R, S WHERE R.j = S.k;*
  - sende Query zur Ausführung nach K3 oder
  - sende Fragment S2 zur Join-Berechnung zu K1



## Phasen der verteilten Anfragebearbeitung



# Anfragetransformation

## ■ Einzelschritte

- Syntaxanalyse (Parsing)
- Namensauflösung
- semantische Analyse
- Normalisierung
- algebraische Vereinfachungen und Restrukturierung

## ■ Erzeugung einer Interndarstellung für die Anfrage

- normalisierter Ausdruck der Relationenalgebra
- Überführung in **Operatorbaum**
  - Blätter: Tabellen
  - innere Knoten: Operatoren der Relationenalgebra
  - Wurzel liefert Anfrageergebnis



## Anfragetransformation (2)

### ■ Umstrukturierungen zur effizienteren Auswertbarkeit durch Nutzung von Äquivalenzbeziehungen für relationale Operatoren

$$\sigma_{P_1}(\sigma_{P_2}(R)) \Leftrightarrow \sigma_{P_1 \wedge P_2}(R)$$

$$\pi_A(\pi_{A,B}(R)) \Leftrightarrow \pi_A(R)$$

$$\pi_A(\sigma_P(R)) \Leftrightarrow \sigma_P(\pi_A(R)) \quad \text{falls } P \text{ nur Attribute aus } A \text{ umfasst}$$

$$\pi_A(\sigma_P(R)) \Leftrightarrow \pi_A(\sigma_P(\pi_{A,B}(R))) \quad \text{falls } P \text{ auch Attribute aus } B \text{ umfasst}$$

$$\sigma_{P(R_1)}(R_1 \bowtie R_2) \Leftrightarrow \sigma_{P(R_1)}(R_1) \bowtie R_2 \quad (P(R_1) \text{ sei Prädikat auf } R_1)$$

$$\pi_{A,B}(R_1 \bowtie R_2) \Leftrightarrow \pi_{A,V}(R_1) \bowtie \pi_{B,V}(R_2) \\ (A/B \text{ seien Attributmengen aus } R_1/R_2, V \text{ Join-Attribut})$$

$$\sigma_P(R_1 \cup R_2) \Leftrightarrow \sigma_P(R_1) \cup \sigma_P(R_2)$$

$$\pi_A(R_1 \cup R_2) \Leftrightarrow \pi_A(R_1) \cup \pi_A(R_2)$$



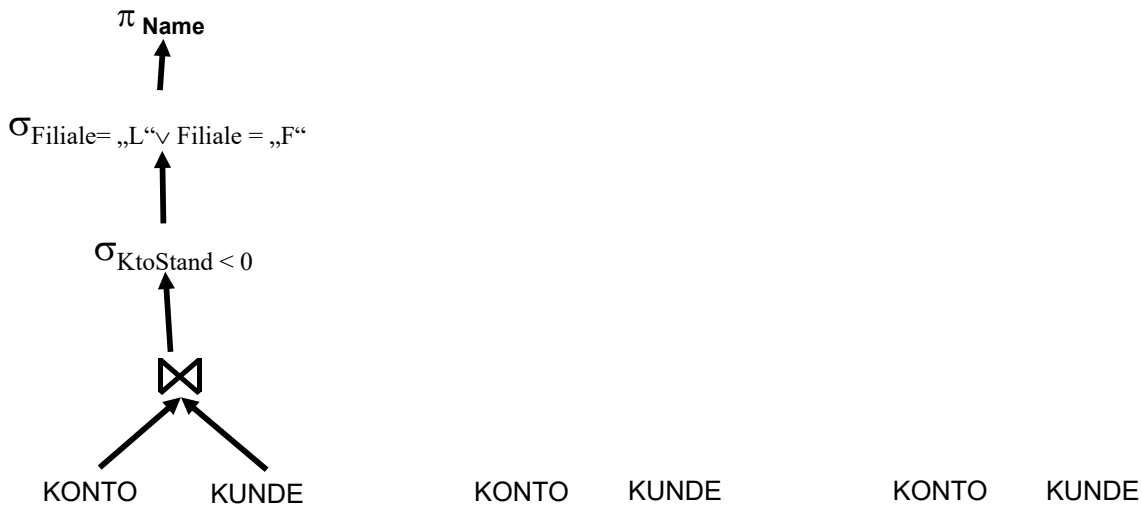
# Anfragetransformation (3)

## ■ Heuristiken

- frühzeitige Ausführung von Selektionsoperationen
- frühzeitige Durchführung von Projektionen (ohne Duplikateliminierung)
- Zusammenfassung mehrerer Selektionen und Projektionen auf demselben Objekt
- Bestimmung gemeinsamer Teilausdrücke

## ■ Beispiel

- KUNDE (KNR, Name, Filiale), KONTO (Ktonr, KNR, KtoStand)



## Erzeugung von Fragment-Queries (Daten-Lokalisierung)

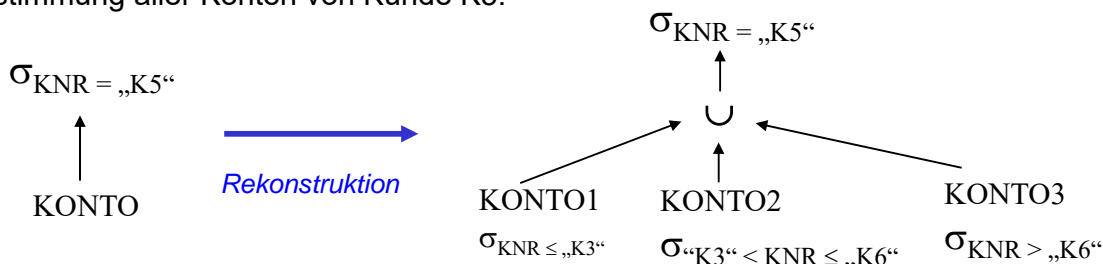
### ■ Relationen im Operatorbaum müssen auf Fragmente abgebildet werden

- ersetze Relationen durch Rekonstruktionsanweisungen auf Fragmenten
- führe algebraische Vereinfachungen durch

### ■ Fall 1: horizontale Fragmentierung

KONTO sei horizontal in drei Fragmente zerlegt (Fragmentierungsattribut KNR)

Bestimmung aller Konten von Kunde K5:



*algebraische Optimierung:*

**reduzierter Fragment-Ausdruck**

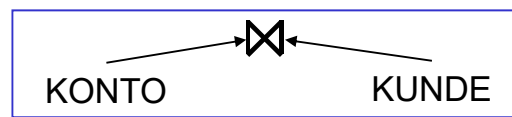


## Erzeugung von Fragment-Queries (2)

### Join-Berechnung bei horizontaler Fragmentierung

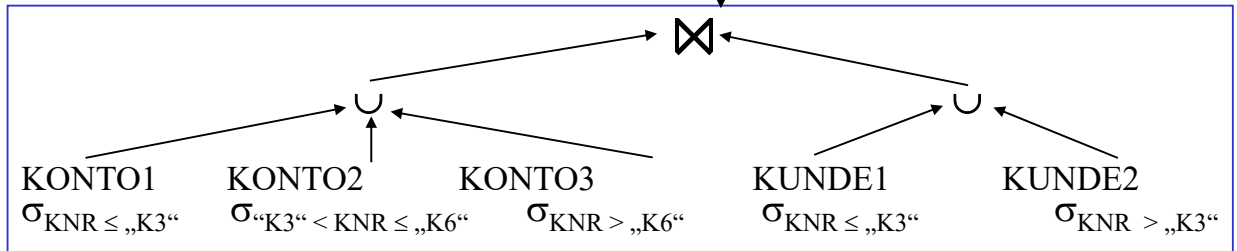
- reduzierter Kommunikations- und Verarbeitungsumfang für Joins auf Fragmentierungsattribut
- Unterstützung paralleler Join-Berechnung

*KONTO und KUNDE seien horizontal über KNR zerlegt (unterschiedl. Bereiche)*



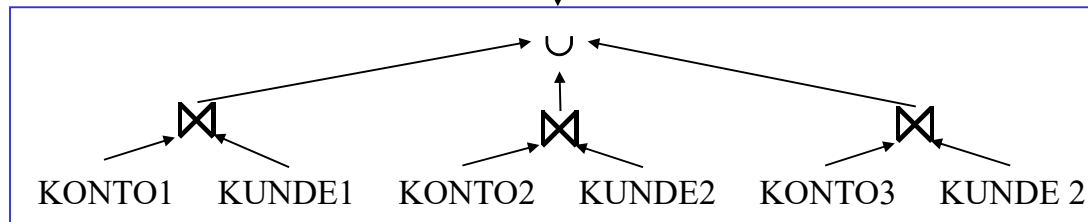
*Rekonstruktion*

**initialer Fragment-Ausdruck**



*algebraische Optimierung*

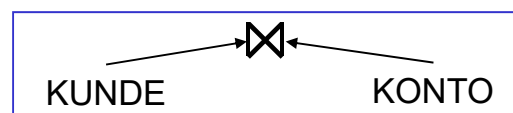
**reduzierter Fragment-Ausdruck**



## Erzeugung von Fragment-Queries (3)

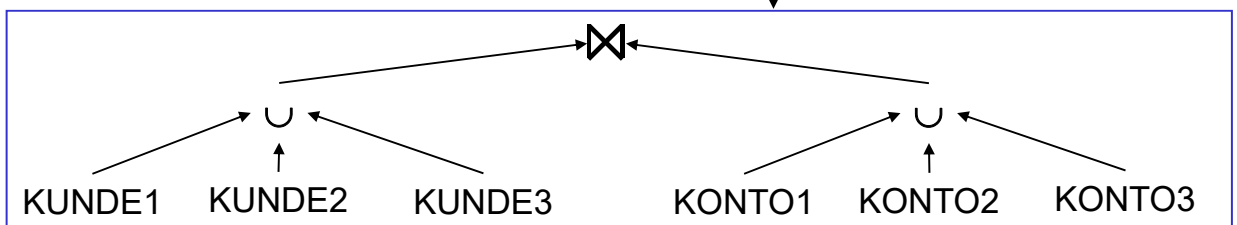
### Fall 2: Abhängige horizontale Fragmentierung

- horizontale Fragmentierung von KUNDE über Filiale,
- abhängige Fragmentierung von KONTO



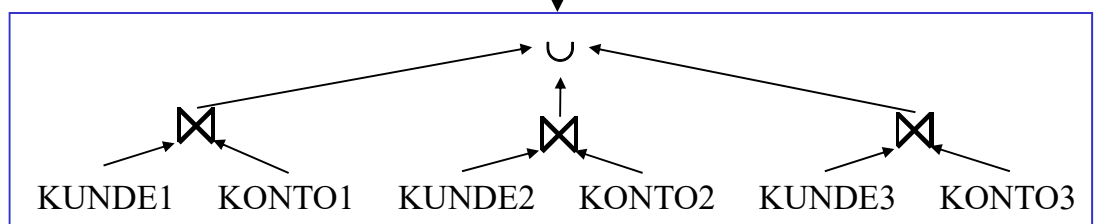
*Rekonstruktion*

**initialer Fragment-Ausdruck**



*algebraische Optimierung*

**reduzierter Fragment-Ausdruck**



- vollkommen lokale und parallele Join-Berechnung möglich

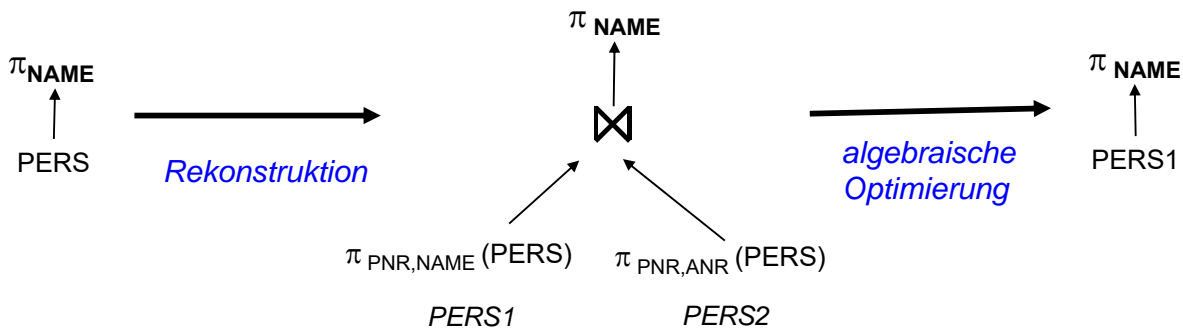


# Erzeugung von Fragment-Queries (4)

## ■ Fall 3: vertikale Fragmentierung

PERS1 (PNR, NAME);      PERS 2 (PNR, ANR)

Bestimme alle Angestelltenamen



# Erzeugung von Fragment-Queries (5)

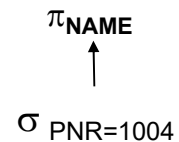
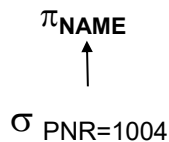
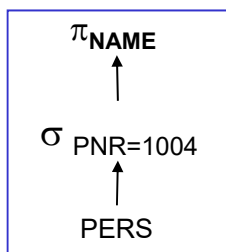
## ■ Fall 4: hybride Fragmentierung

$PERS1 = \sigma_{PNR < 1003} (\pi_{PNR, NAME} (PERS))$

$PERS2 = \sigma_{PNR \geq 1003} (\pi_{PNR, NAME} (PERS))$

$PERS3 = \pi_{PNR, ANR} (PERS)$

Bestimme den Namen von Angestelltem mit PNR 1004



# Globale Optimierung

- Bestimmung eines Ausführungsplanes mit minimalen globalen Kosten
  - Bestimmung der Ausführungsknoten
  - Festlegung der Ausführungsreihenfolge (sequentiell, parallel)
  - alternative Strategien zur Join-Berechnung bewerten
- Trennung zwischen globaler und lokaler Optimierung kann zur Auswahl suboptimaler Pläne führen
- Kostenmodell: Berücksichtigung von CPU-, E/A- und Kommunikationskosten

$$\begin{aligned} \text{Gesamtkosten} = & W_{\text{CPU}} * \#\text{Instruktionen} + \\ & W_{\text{I/O}} * \#\text{E/A} + \\ & W_{\text{Msg}} * \#\text{Nachrichten} + \\ & W_{\text{Byte}} * \#\text{Bytes} \end{aligned}$$



## Parallele Berechnung von Selektion und Projektion

- Daten-Lokalisierung führt zu Selektions- und Projektionsoperationen auf Fragmenten
- effektive Parallelisierung bei horizontaler Fragmentierung:

$$R = \cup (R_1, R_2, \dots, R_n)$$

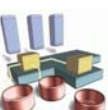
$$\text{Selektion: } \sigma(R) \Rightarrow \cup (\sigma(R_1), \sigma(R_2), \dots, \sigma(R_n))$$

$$\text{Projektion: } \pi(R) \Rightarrow \cup (\pi(R_1), \pi(R_2), \dots, \pi(R_n))$$

- Datenverteilung gestattet parallele Berechnung der lokalen Teiloperationen
- Mischen der Teilergebnisse (Vereinigung)
- Projektion: ggf. Duplikateliminierung (z.B. durch Sortierung)

### ■ Shared Nothing

- Ausführung auf den Datenknoten
- Rechner und Parallelitätsgrad (n) durch Datenverteilung bestimmt (Ausnahme: Anfragen auf dem Verteilattribut)
- auch Index-Scans i.a. auf allen n Rechnern auszuführen



# Parallele Berechnung von Aggregatfunktionen

Sei  $Q(R)$  ein Attribut von  $R$ , auf das Aggregat-Funktion angewendet werden sollen

## ■ parallele Berechnung für MIN, MAX immer möglich:

$$\text{MIN}(Q(R)) \Rightarrow \text{MIN}(\text{MIN}(Q(R_1)), \dots, \text{MIN}(Q(R_n)))$$

$$\text{MAX}(Q(R)) \Rightarrow \text{MAX}(\text{MAX}(Q(R_1)), \dots, \text{MAX}(Q(R_n)))$$

- parallele Berechnung der lokalen Minima/Maxima
- Bestimmung des globalen Extremwerts

## ■ für SUM, COUNT, AVG parallele Berechnung nur anwendbar, falls keine Duplikateliminierung erforderlich ist

$$\text{SUM}(Q(R)) \Rightarrow \Sigma \text{SUM}(Q(R_i))$$

$$\text{COUNT}(Q(R)) \Rightarrow \Sigma \text{COUNT}(Q(R_i))$$

$$\text{AVG}(Q(R)) \Rightarrow \text{SUM}(Q(R)) / \text{COUNT}(Q(R))$$



# Join-Berechnung

## ■ Join (Verbund)

- satztypübergreifende Operation: gewöhnlich sehr teuer
- häufige Nutzung: wichtiger Optimierungskandidat
- typische Anwendung: Gleichverbund; allgemeiner  $\Theta$ -Verbund selten

## ■ Implementierungsalternativen in zentralen DBS

### Nested Loop:

- jeder Satz der ersten wird mit jedem Satz der zweiten Relation abgeglichen
- für alle Join-Arten nutzbar

### Sort-Merge:

- Eingaberelationen sind auf Join-Attribut sortiert (bzw. haben Index auf Join-Attribut)
- Gleichverbund über schritthaltenden Durchgang und Abgleich der Relationen

### Hash-Join:

- kleinere Relation wird in Hauptspeicher-Hash-Tabelle gebracht (Hash-Funktion auf Join-Attribut)
- Prüfung für jeden Satz der zweiten Relation, ob Wert des Join-Attributs in Hash-Tabelle (Relation 1)
- nur für Gleichverbund

## ■ Optimierungsziele in VDBS/PDBS:

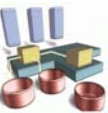
- Reduzierung der Kommunikationskosten sowie Nutzung von Parallelverarbeitung
- Mehr-Wege-Joins: Wahl der Ausführungsreihenfolge; Einsatz von Inter-Operator-Parallelität





# Join-Berechnung in Verteilten DBS

- Join-Anfrage in Knoten K zwischen R und S
  - (Teil-)Relation R an Knoten  $K_R$
  - (Teil-)Relation S an  $K_S$
- Festlegung des Ausführungsknotens: K,  $K_R$  oder  $K_S$
- Bestimmung der Auswertungsstrategie
  1. sende beteiligte Relationen vollständig an einen Knoten und führe lokale Join-Berechnung durch ("*Ship Whole*")
    - minimale Nachrichtenanzahl
    - sehr hohes Übertragungsvolumen
  2. fordere für jeden Verbundwert der ersten Relation zugehörige Tupel der zweiten an ("*Fetch as Needed*")
    - hohe Nachrichtenanzahl
    - nur relevante Tupel werden berücksichtigt
  3. Kompromisslösung: *Semi-Join* bzw. Erweiterungen (*Bitvektor-Join*)



## Kommunikationskosten der Join-Berechnung

- hier: Verwendung von zwei Kostenfaktoren
  - $K_N$  = Kommunikationskosten pro Nachricht
  - $K_A$  = Kommunikationskosten für Übertragung eines Attributwerts (AW)

- Strategie 1 (Ship Whole):
  - Übertragung von R und S nach Knoten K
  - Join-Berechnung an Knoten K

- Strategie 2 (Fetch as Needed):
  - Übertragung von R nach Knoten  $K_S$
  - Join-Berechnung an Knoten  $K_S$

- Strategie 3 (Fetch as needed):
  - Join-Berechnung an Knoten  $K_S$
  - pro S-Tupel sukzessive Anforderung von zugehörigen R-Tupeln

**Beispiel**

R		S		
A	B	B	C	D
1	7	1	6	4
2	2	2	5	1
3	7	1	4	2
4	8	5	3	3
5	6	5	2	5
6	3	6	1	8
7	9			

$14 \text{ AW}$ 
 $18 \text{ AW}$

R  $\bowtie$  S

A	B	C	D
2	2	5	1
5	6	1	8

$8 \text{ AW}$



# Semi-Join-Verfahren

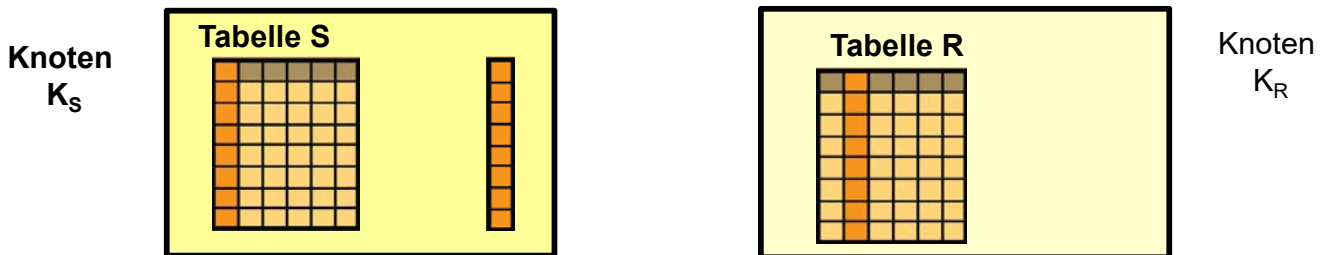
- Reduzierung des Kommunikationsaufwandes
  - alle Verbundpartner können auf einmal angefordert werden
- entspricht Anwendung des Semi-Joins als „Reducer“ (Filter)

$$\begin{aligned}
 R \bowtie S &= R \bowtie (S \ltimes R) = R \bowtie (S \bowtie \pi_V(R)) \text{ (Verbundattribut } V) \\
 &= (R \ltimes S) \bowtie S = (R \bowtie \pi_V(S)) \bowtie S \\
 &= (R \ltimes S) \bowtie (S \ltimes R) = (R \bowtie \pi_V(S)) \bowtie (S \bowtie \pi_V(R))
 \end{aligned}$$

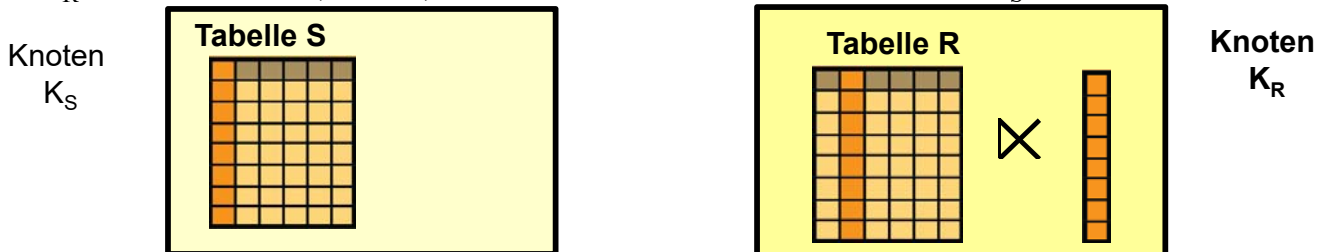


## Semi-Join-Algorithmus

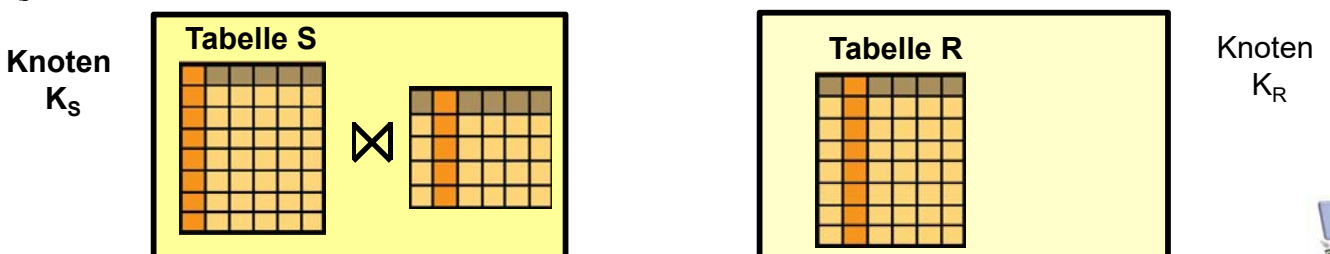
1. Bestimme  $S' = \pi_V(S)$  in  $K_S$  und sende  $S'$  an Knoten  $K_R$



2. in  $K_R$  bestimme  $R' = (R \ltimes S') = R \ltimes S' = R \ltimes S$  und sende  $R'$  an  $K_S$

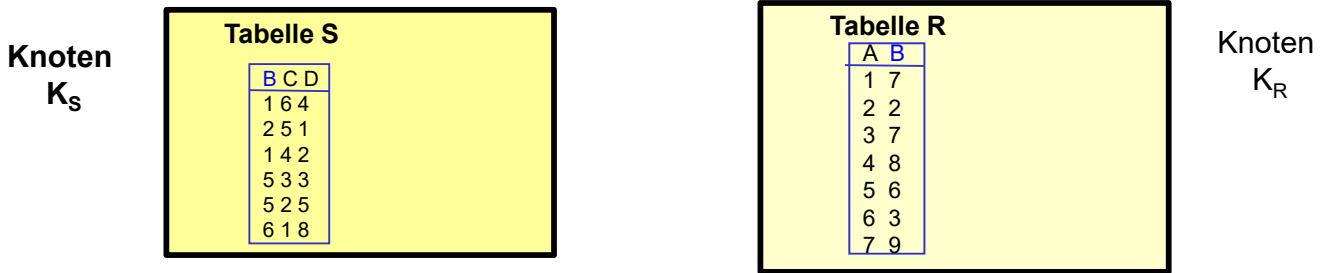


3.  $K_S$  berechnet den Join zwischen  $S$  und  $R'$  (entspricht  $R \bowtie S$ )

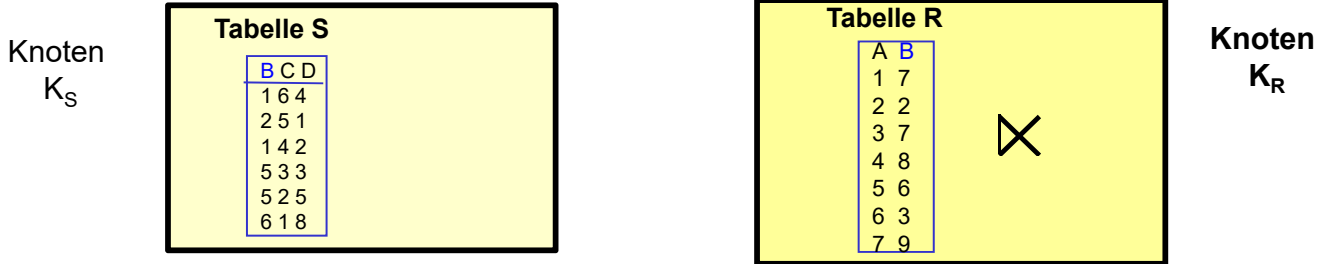


# Semi-Join-Algorithmus: Beispiel

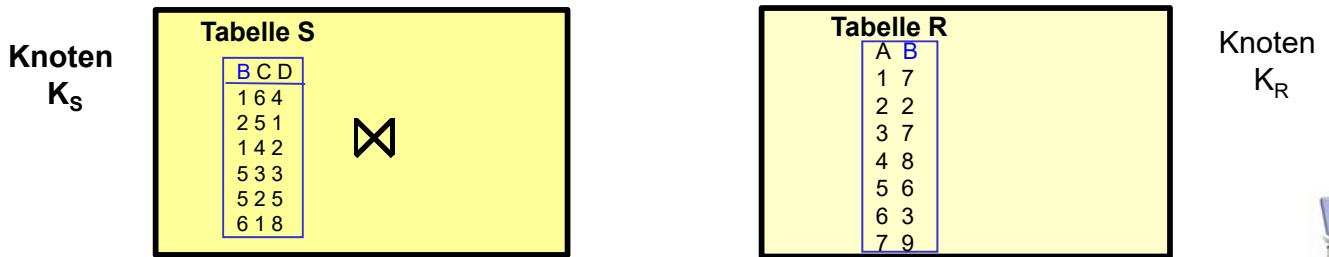
1. Bestimme  $S' = \pi_B(S)$  in  $K_S$  und sende  $S'$  an Knoten  $K_R$



2. in  $K_R$  bestimme  $R' = (R \bowtie S') = R \bowtie S$  und sende  $R'$  an  $K_S$

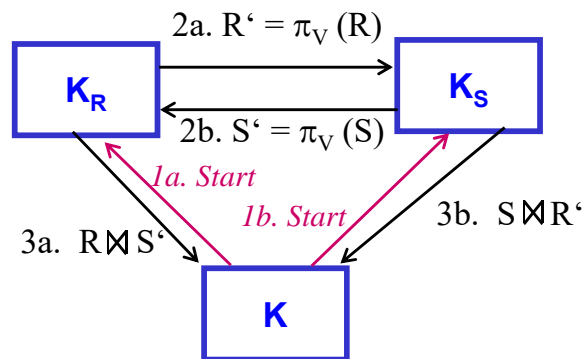


3.  $K_S$  berechnet den Join zwischen  $S$  und  $R'$  (entspricht  $R \bowtie S$ )

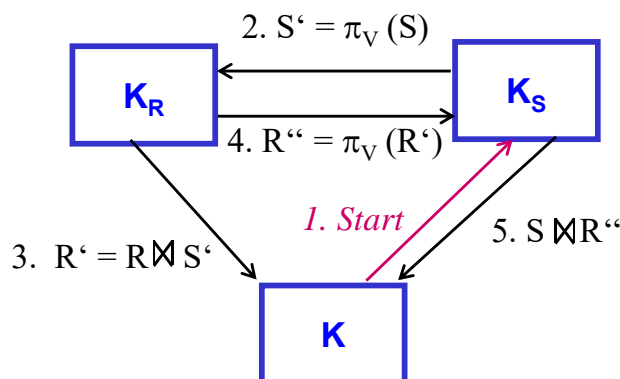


## Semi-Join: Join an drittem Knoten K

parallele Variante



sequenzielle Variante (über  $K_S$ )

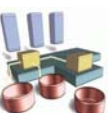
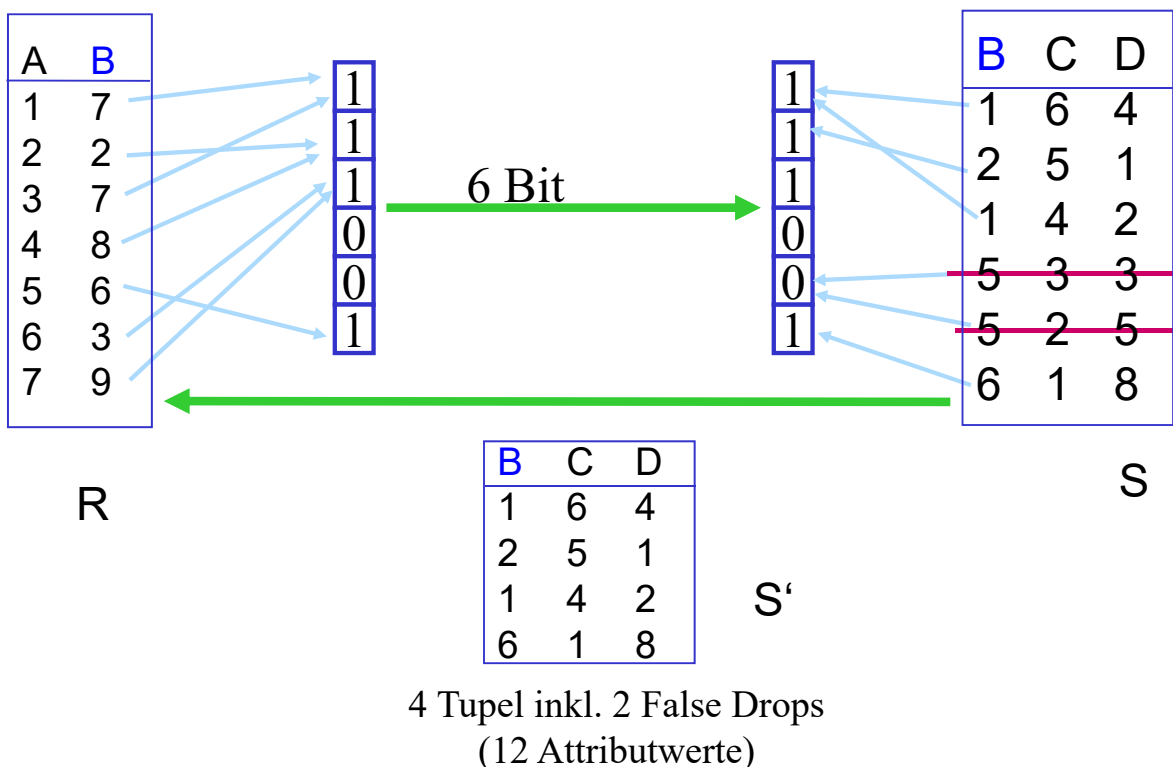


# Bitvektor-Join (Hash-Filter-Join)

- Abbildung von Werten des Join-Attributes auf Bitvektor mittels Hash-Funktion
  - Übertragung des Bitvektors anstatt von Attributwerten wie bei Semi-Join
- Nutzung zur Join-Berechnung an Knoten  $K_R$ :
  1. in  $K_R$  setze für jeden Wert  $w$  in  $\pi_v(R)$  zugehöriges Bit  $h(w)$  im Bitvektor und sende diesen an Knoten  $K_S$  (Verbundattribut  $v$ )
  2. in  $K_S$  bestimme  $S' = \{s \in S \mid \text{Bit an Position } h(s.v) \text{ im Bitvektor gesetzt}\}$  und sende  $S'$  an  $K_R$
  3.  $K_R$  berechnet den Join zwischen  $R$  und  $S'$
- Beobachtungen
  - Bit-Vektor in Schritt 1 kompakter als Menge der Attributwerte
  - Teilrelation  $S'$  in Schritt 2 i.a. größer als bei Semi-Join (aufgrund von False Drops)
  - Scan statt Join in Schritt 2



## Bitvektor-Join: Beispiel



# Parallele Join-Bearbeitung

## ■ Join zwischen R und S:

- $R = \cup (R_1, R_2, \dots, R_n)$
- $S = \cup (S_1, S_2, \dots, S_m)$
- S sei kleiner als R

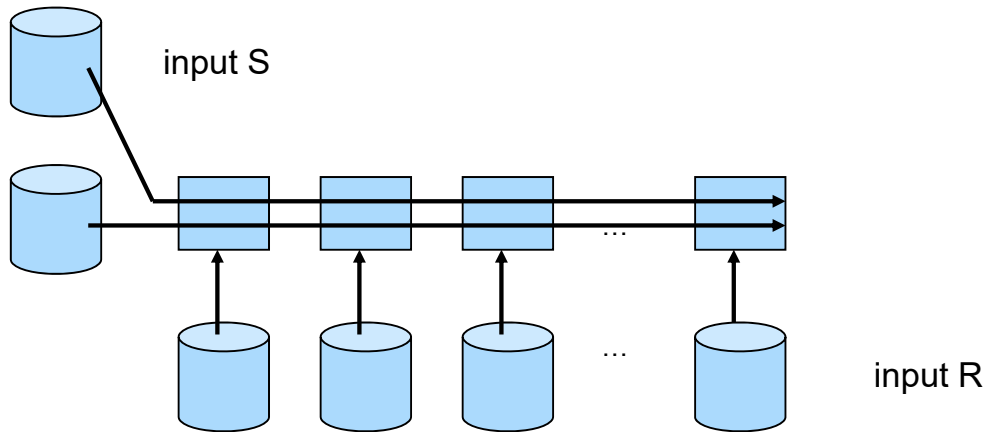
## ■ allgemeine Alternativen (für beliebige lokale Join-Verfahren)

- **dynamische Replikation (Fragment and Replicate)**
  - sende S an jeden R-Knoten
  - für alle Join-Arten nutzbar
- **dynamische Re-Partitionierung**
  - Umverteilung der Eingabedaten durch Partitionierung auf Join-Attribut
  - symmetrische vs. asymmetrische Variante
  - nur Gleichverbund
- **statische Partitionierung auf Join-Attribut**
  - abhängige horizontale Fragmentierung auf Join-Attribut
  - optimaler Fall bezüglich Kommunikationsaufwand
  - nur Gleichverbund

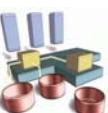
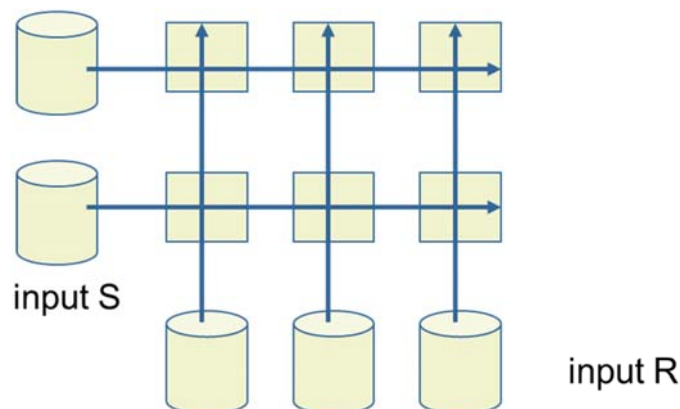
## ■ parallele Hash-Joins



## Fragment and Replicate



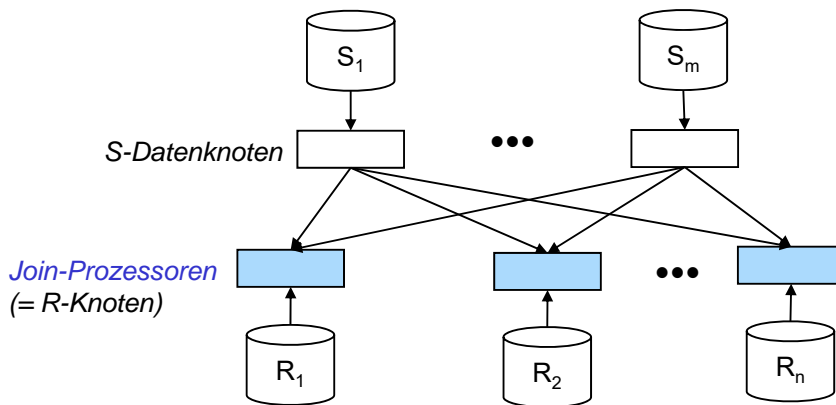
## verallgemeinertes Fragment and Replicate



# Dynamische Replikation der kleineren Relation

## ■ Algorithmus

1. Koordinator K: initiere Join auf allen  $R_i$  ( $i=1 \dots n$ ) und allen  $S_j$  ( $j=1 \dots m$ )
2. Scan-Phase: in jedem S-Knoten führe parallel durch:  
lies lokale Partition  $S_j$  und sende sie an jeden Knoten  $R_i$  ( $i=1 \dots n$ )
3. Join-Phase: in jedem R-Knoten mit Partition  $R_i$  führe parallel durch:
  - $S := \cup S_j$  ( $j=1 \dots m$ )
  - berechne  $T_i := R_i \bowtie S$  (impliziert Lesen von  $R_i$ )
  - schicke  $T_i$  an Koordinator
4. Koordinator: empfangen und mische alle  $T_i$



## ■ Eigenschaften

- für alle Join-Prädikate einsetzbar
- lokale Join-Berechnung mit beliebigem Verfahren möglich



# Dynamische (Re-) Partitionierung

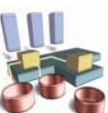
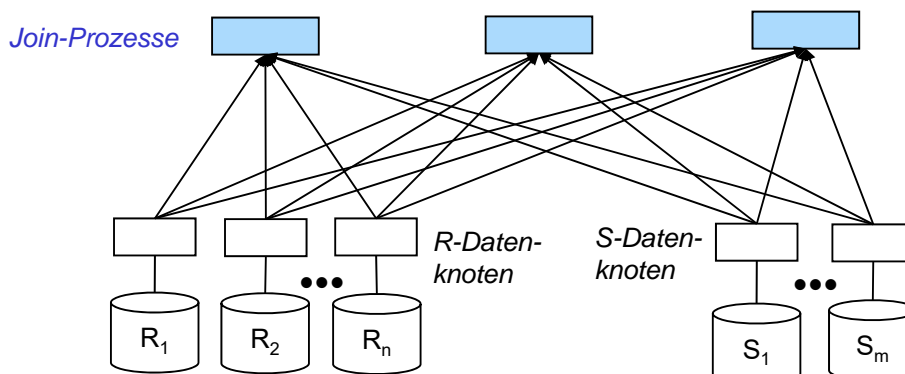
## ■ Voraussetzung: Gleichverbund

## ■ allgemeiner Fall:

- **symmetrische Umverteilung** beider Relationen unter p Join-Prozessoren
- Verteilungsfunktion (Hash- oder Bereichspartitionierung) auf dem Join-Attribut

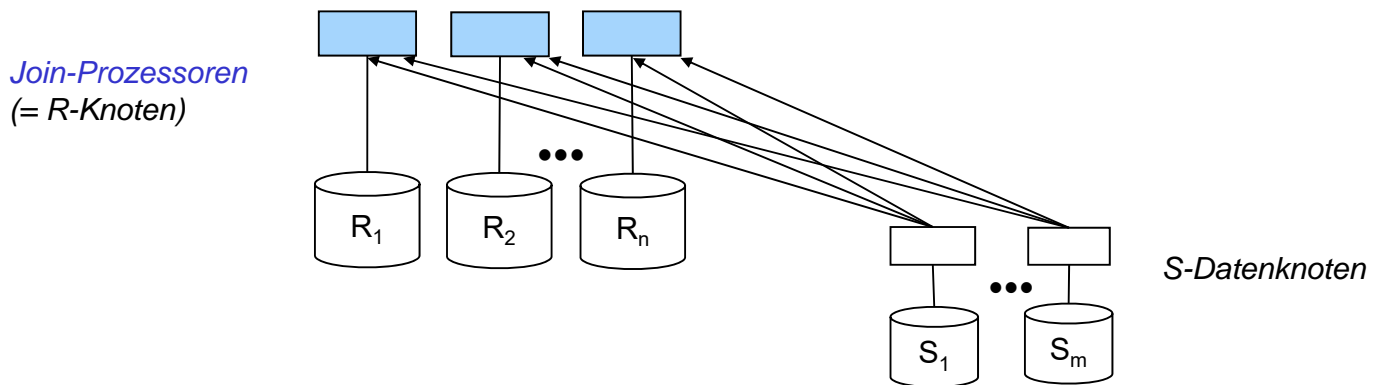
## ■ Bewertung:

- reduzierter Join-Aufwand gegenüber dynamischer Replikation
- hohe Flexibilität zur dynamischen Lastbalancierung (Wahl des Parallelitätsgrades p sowie der Join-Prozessoren)
- hoher Kommunikationsaufwand
- jeder lokale (Equi-) Join-Algorithmus anwendbar



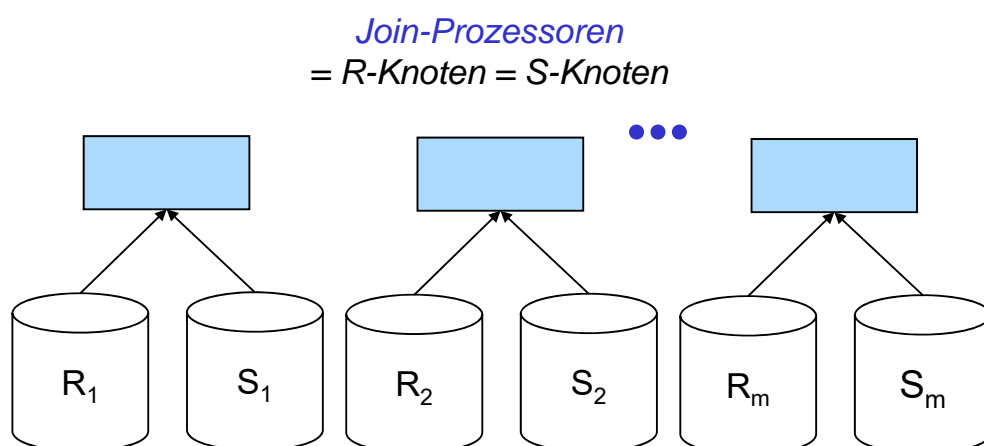
## Dynamische Re-Partitionierung (2)

- Spezialfall: asymmetrische Re-Partitionierung
  - Verteilungsattribut = Join-Attribut für eine der beiden Relationen
- nur eine Relation braucht umverteilt zu werden
- Potenzial zur dynamischen Lastbalancierung entfällt



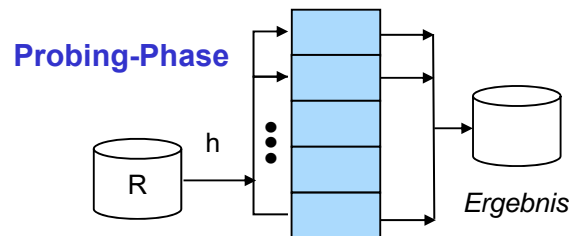
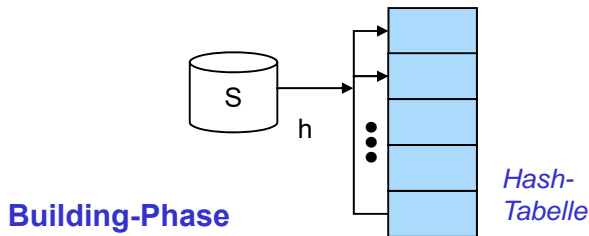
## Statische Partitionierung

- beide Relationen besitzen Join-Attribut als Verteilattribut und identische Verteilfunktion ( $m=n$ ,  $R_i$  und  $S_i$  an denselben Rechnern)
  - entspricht abhängiger horizontaler Fragmentierung
  - „co-located joins“
  - keinerlei Umverteilung erforderlich !
- minimaler Kommunikationsaufwand



# Hash-Join (zentraler Fall)

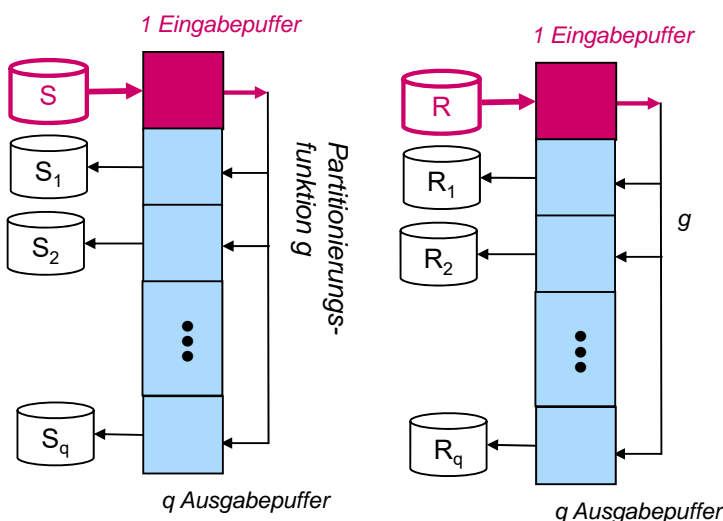
- nur für Gleichverbund
- Idealfall: kleinere (innere) Relation S passt vollständig in Hauptspeicher
  - **Building-Phase:** Einlesen von S und Speicherung in einer Hash-Tabelle unter Anwendung einer Hash-Funktion h auf dem Join-Attribut
  - **Probing-Phase:** Einlesen von R und Überprüfung für jeden Join-Attributwert, ob zugehörige S-Tupel vorliegen (wenn ja, erfolgt Übernahme ins Join-Ergebnis)
- Vorteile
  - lineare Kosten  $O(N)$
  - Partitionierung des Suchraumes: Suche nach Verbundpartnern nur innerhalb 1 Hash-Klasse
  - Nutzung großer Hauptspeicher
  - auch für Joins auf Zwischenergebnissen gut nutzbar



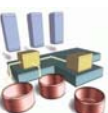
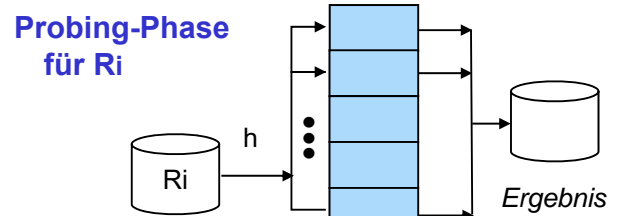
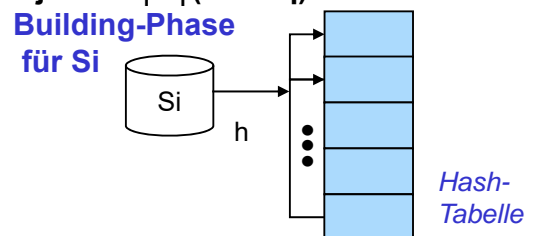
# Hash-Join (2)

- falls kleinere Relation nicht vollständig in Hauptspeicher passt => **Überlaufbehandlung** durch Partitionierung der Eingaberelationen
  - Partitionierung von S und R in q Partitionen über (Hash-)Funktion g auf dem Join-Attribut, so dass jede S-Partition in den Hauptspeicher passt
  - q-fache Anwendung des Basisalgorithmus' auf je zwei zusammengehörigen Partitionen
  - rund 3-facher E/A-Aufwand gegenüber Basisverfahren ohne Überlauf

## Partitionierungs-Phase für S und R



## für jedes $S_i/R_i$ ( $i = 1..q$ ):

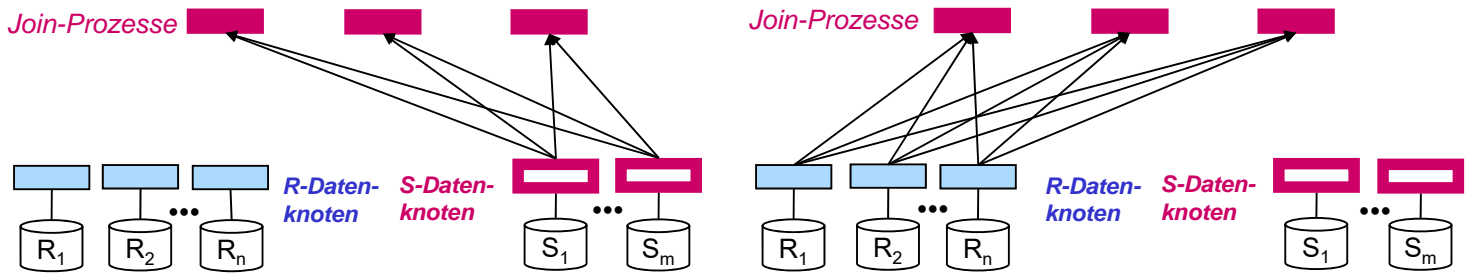




# Paralleler Hash-Join

Building-Phase:

Probing-Phase:



- dynamische Partitionierung über Hash-Funktion  $h1$  auf Join-Attribut
  - zunächst Umverteilung der kleineren Relation S unter Join-Prozessoren
  - Building: in Join-Prozessoren kommen eingehende Tupel in Hauptspeicher-Hash-Tabelle (Hash-Funktion  $h2$ )
  - Umverteilung der zweiten Relation R auf die Join-Prozessoren unter Anwendung von  $h1$
  - Probing: für eingehende Tupel werden Verbundpartner in der Hash-Tabelle ermittelt



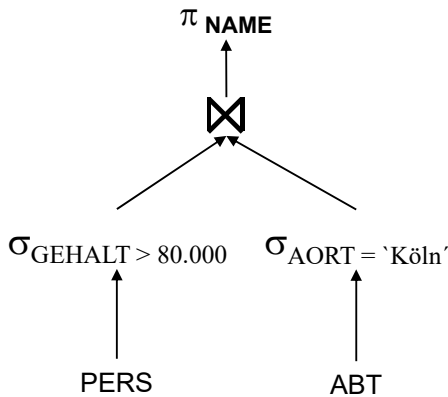
## Paralleler Hash-Join (2)

- Nacheinanderausführung der Scan-Phasen ermöglicht Pipeline-Parallelität in Building- und Probing-Phase
- Reduzierung des Umverteilungsaufwandes für R durch Anwendung von **Bitvektor-Filterung**
  1. Erstellung von Bitvektoren an S-Knoten für vorkommende Verbundattributwerte und Übermittlung an R-Knoten
  2. OR-Verknüpfung der Bitvektoren an R-Knoten
  3. Umverteilung nur von R-Sätzen, für deren Verbundattributwert Bitvektoreintrag gesetzt ist

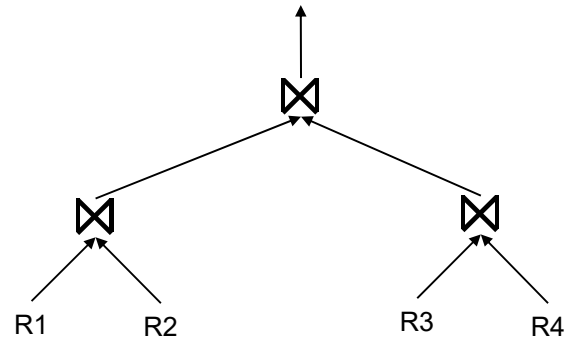


# Inter-Operator-Parallelität

- Analyse des Operatorbaumes zur parallelen Berechnung unabhängiger Knoten (Operatoren)
- Beispiele:



Parallele Berechnung verschiedener Selektionen



Mehr-Wege-Join  
(R1 ⋈ R2 ⋈ R3 ⋈ R4)



## Mehr-Wege-Joins: Nutzung von Pipeline-Parallelität

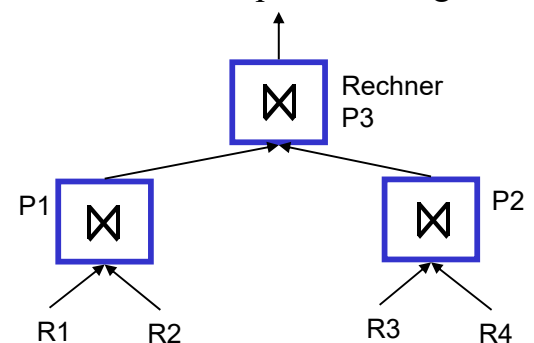
### ■ Beispiel: Pipeline-Join in Rechner P3

- P3 habe Eingangswarteschlange Q für Tupel aus P1 und P2
- spezielle Nachrichten ENDP1, ENDP2 zeigen an, dass keine weiteren Tupel mehr folgen

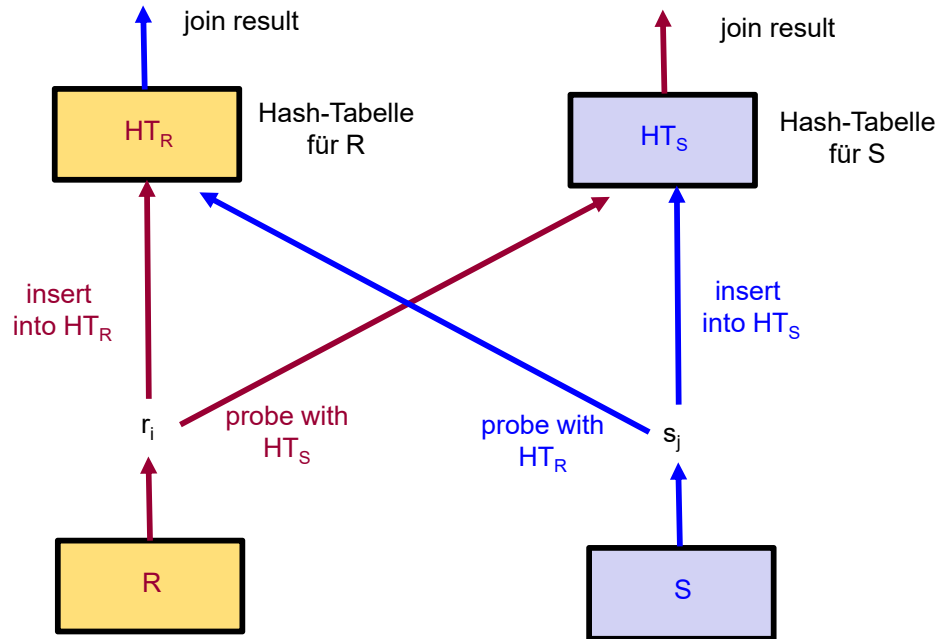
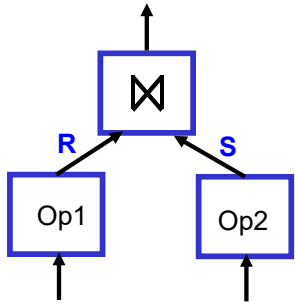
### ■ Algorithmus:

```

Fertig1, Fertig2 := FALSE;
Von1, Von2, Ergebnis := { }
WHILE NOT (Fertig1 AND Fertig2) DO BEGIN
  IF Q leer THEN warte bis neue Tupel eintreffen;
  t := erstes Tupel in Q;
  IF t = ENDP1 THEN Fertig1 := TRUE
  ELSE IF t=ENDP2 THEN Fertig2 := TRUE
  ELSE IF t von P1 THEN BEGIN
    Von1 := Von1 ∪ {t};
    Ergebnis := Ergebnis ∪ ( {t} ⋈ Von2 ) END
  ELSE BEGIN (* t von P2 *)
    Von2 := Von2 ∪ {t};
    Ergebnis := Ergebnis ∪ ( {t} ⋈ Von1 ) END
END; (* WHILE *)
    
```

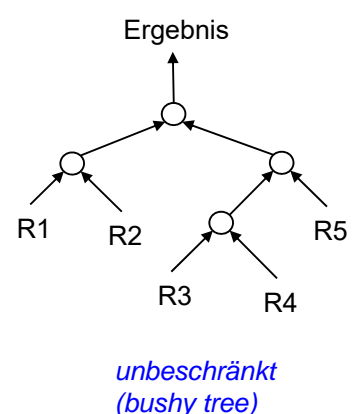
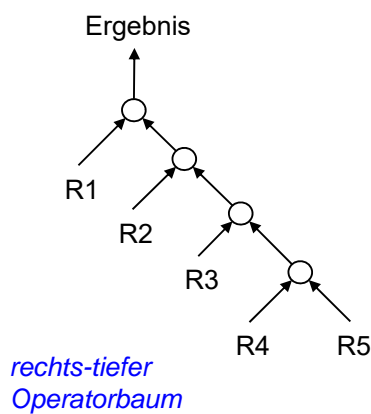
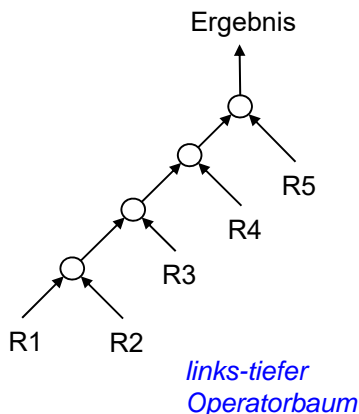


# Symmetrischer Hash-Join

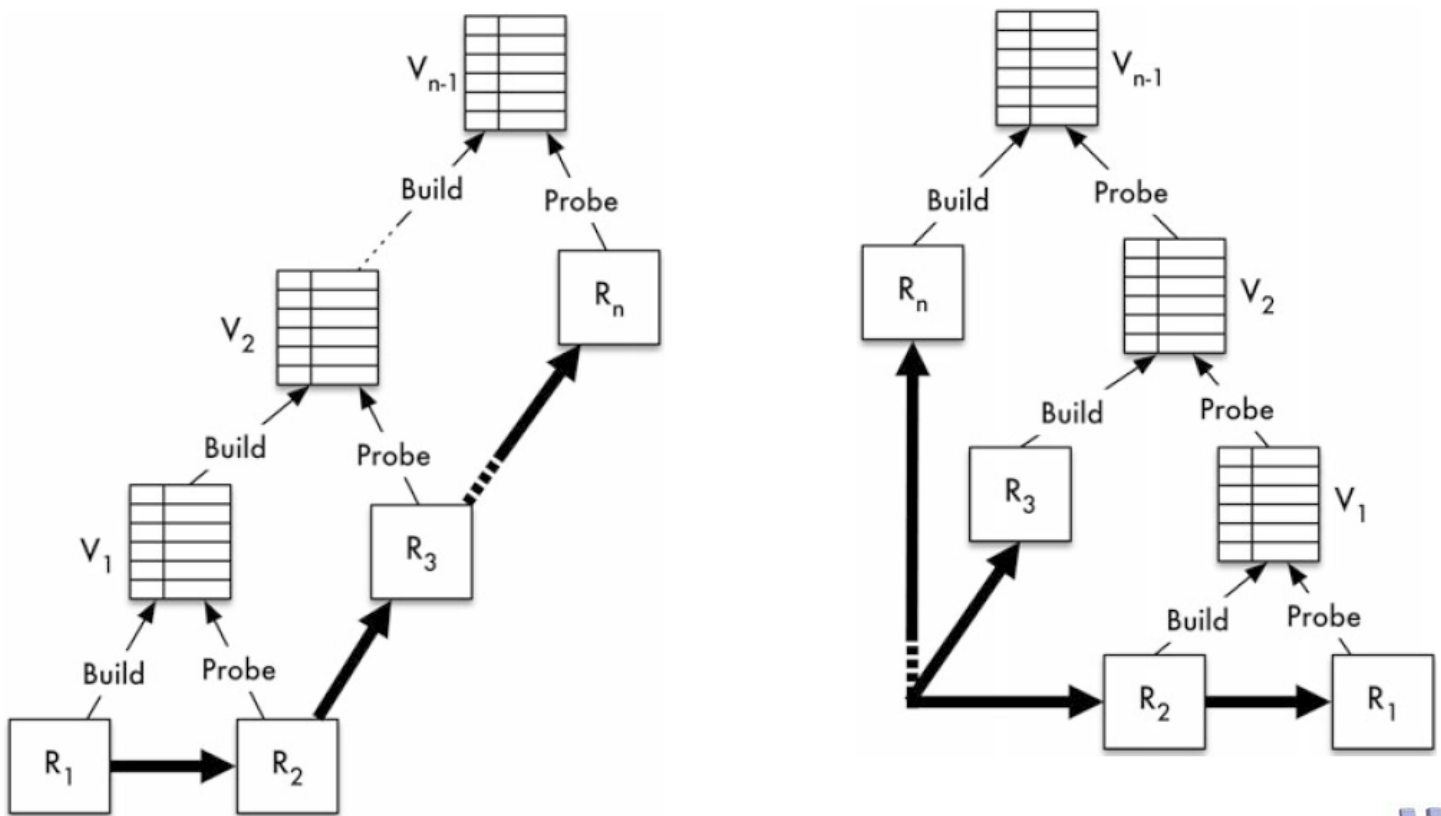


## Mehr-Wege-Joins (2)

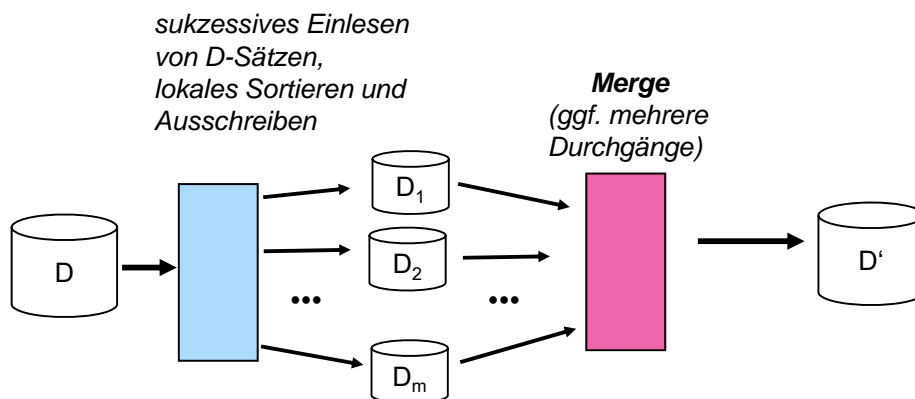
- beschränkte Berechnungsreihenfolgen für Mehrwege-Joins erleichtern Optimierungsproblem
- links-tiefe Bäume
  - Join-Berechnung in N Schritten
  - pro Schritt werden nur zwei Relationen bearbeitet
- rechts-tiefe Bäume
  - hoher Grad an Parallelität bei Verwendung von Hash-Joins
  - jedoch hoher Ressourcenbedarf



# Links- v. rechts-tiefe Mehrwege-Hash-Joins



## Parallele Sortierung



### ■ DBS: externes Sortieren (Merge-Sort)

- Zerlegung der Eingabe in mehrere Läufe (runs)
- Sortieren und Zwischenspeichern der sortierten Läufe
- sukzessives Mischen bis 1 sortierter Lauf entsteht

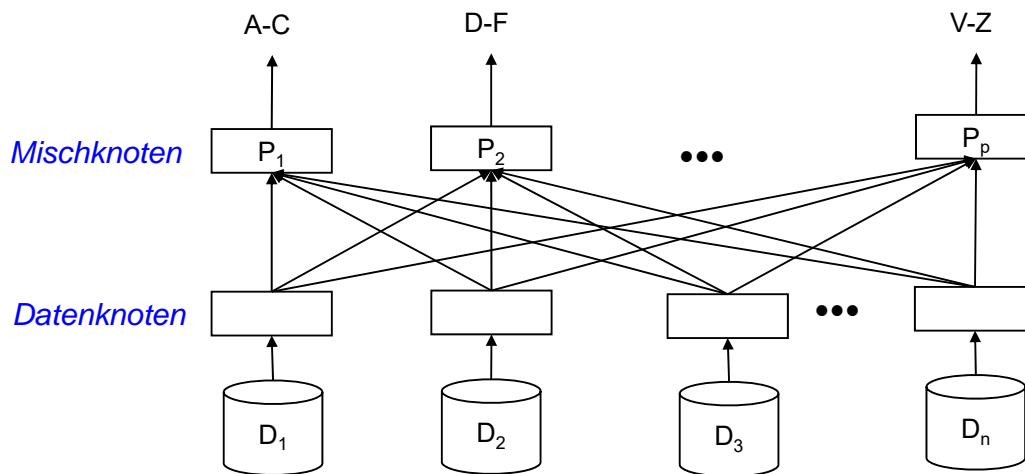
### ■ Anforderungen an parallele Sortierung

- parallele Eingabe (multiple input)
- parallele Sortierphasen
- paralleles Mischen
- Partitionierung der sortierten Ausgabe (multiple output)



## Parallele Sortierung (2)

- lokale Sortierung der Partitionen an den Datenknoten
- dynamische Umverteilung (Partitionierung) der sortierten Läufe unter  $p$  Mischknoten
  - Umverteilung über dynamische **Bereichsfragmentierung** auf dem Sortierattribut
- paralleles Mischen in den  $p$  Mischknoten
- partitionierte Ausgabe



## Zusammenfassung

- verteilte und parallele Anfragebearbeitung
  - Anfragetransformation, algebraische Optimierung
  - Daten-Lokalisierung: Abbildung von Operatoren auf Fragmente
  - globale Optimierung: Kostenbewertung unter Berücksichtigung von Kommunikationsaufwand und Parallelisierungsalternativen
- verteilte Join-Verarbeitung
  - Alternativen bezüglich Wahl des Join-Knotens und Übertragung der Daten
  - Semi-Join und Bitvektor-Join effektiv nutzbar
- Selektion, Projektion, Aggregationen
  - Parallelisierbarkeit durch horizontale Fragmentierung
  - Datenallokation bestimmt Ausführungsort (Shared Nothing)
- parallele Join-Verarbeitung, Sortierung
  - hohes Lastbalancierungspotenzial bei dynamischer Umverteilung der Relationen
  - Kommunikationseinsparungen für Join falls Join-Attribut = Verteilattribut (insbesondere bei abhängiger horizontaler Fragmentierung)
- Nutzung von Pipeline-Parallelität für Hash-Join und Mehr-Wege-Joins

