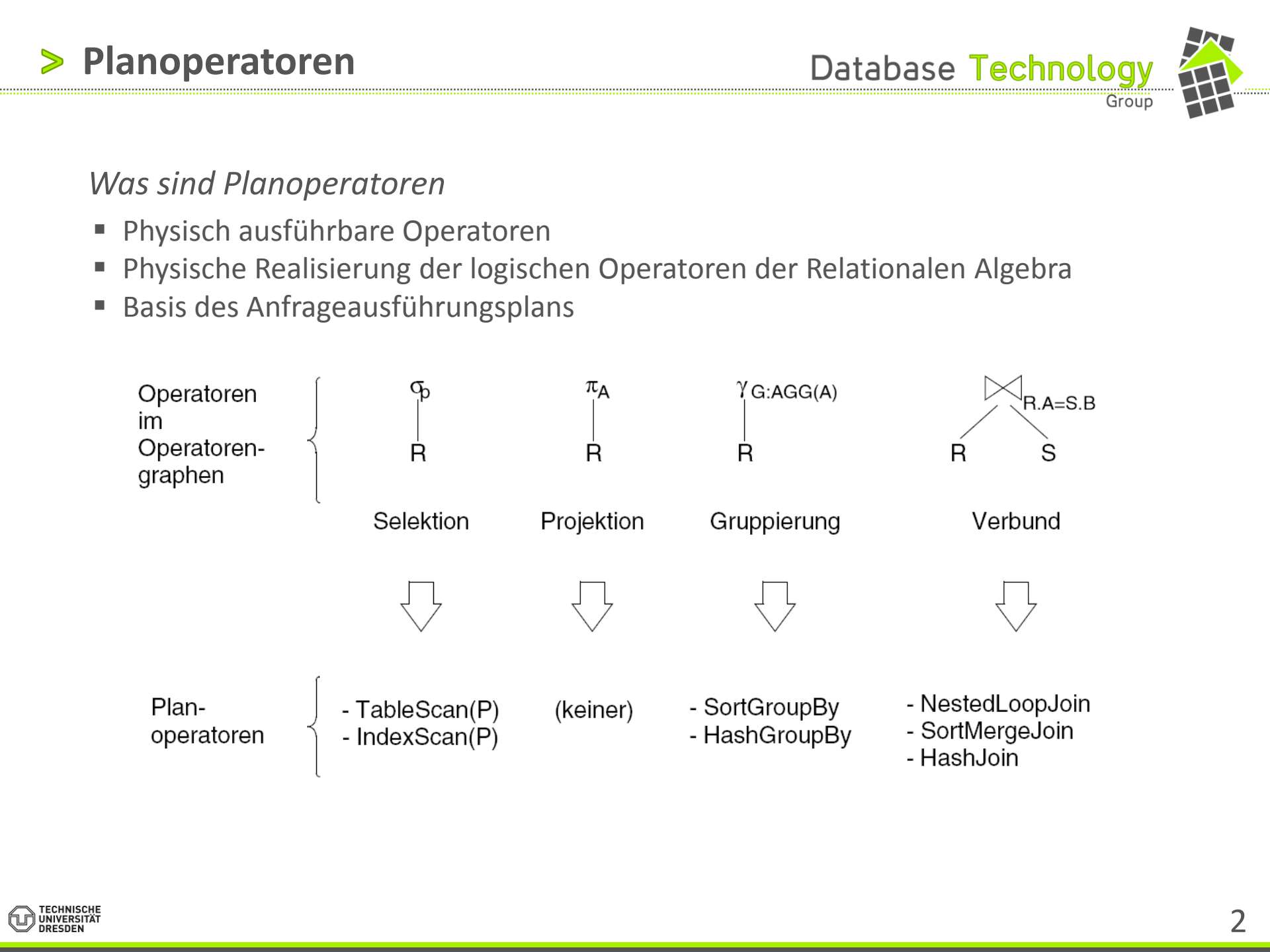
6 Relational Plan Operators



Was sind Planoperatoren

- Physisch ausführbare Operatoren
- Physische Realisierung der logischen Operatoren der Relationalen Algebra
- Basis des Anfrageausführungsplans

- Unäre Operatoren
  - Selektion und Projektion
  - Gruppierung
  - Aggregation
  - Sortieren
- Verbundoperatoren (Binäre Operatoren)
  - Verbundoperatoren
  - Nested-Loop Verbund
  - Sort-Merge Verbund
  - Hash-Verbund
  - Vergleich
  - Paralleler Verbund
  - Data-Skew
  - Verbund in verteilten Systemen

Unäre Operatoren

Planoperatoren für die Projektion

- Spalteneliminierung ist trivial
  - wird typischerweise in Kombination mit Sortierung, Selektion oder Verbund durchgeführt
- Duplikateliminierung wird durch Gruppieren auf allen distinkten Attributen ohne zusätzliche Aggregation realisiert

Planoperatoren zur Selektion

- Nutzung des Scan-Operators
  - Definition von Start- und Stopp-Bedingung
  - Definition von einfachen Suchargumenten
- Relationen-Scan
- Index-Scan
- Auswahl des kostengünstigsten Index

Besonderheit im parallelen Umfeld

- Rekonstruktion bei horizontaler Partitionierung R = ∪ (R1, R2, ..., Rn)
- Möglichkeit der parallelen Berechnung lokaler Operationen
  - Mischen der Teilergebnisse und ggf. Duplikateliminierung bei lokaler Projektion Parallele
- Projektion: PROJ(R) => ∪ (PROJ(R1), PROJ(R2), ... , PROJ(Rn))
- Parallele Selektion: SEL(R) => ∪ (SEL(R1), SEL(R2), ... , SEL(Rn))

Berücksichtigung der Architektur: Shared Nothing

- Ausführen auf den Datenknoten
- Rechner und Parallelitätsgrad (n) durch Datenverteilung bestimmt (Ausnahme: bestimmte Anfragen auf dem Verteilattribut)

Berücksichtigung der Architektur: Shared Disk / Shared Everything

- Datenverteilung auf Platte bestimmt maximalen Parallelitätsgrad
- Selektive Anfragen können auf einen Prozessor beschränkt werden (=> minimaler Kommunikationsaufwand).
- Relationen-Scans können von n Prozessoren bearbeitet werden.
- Parallelitätsgrad kann nicht nur vom Anfragetyp, sondern auch von der aktuellen Auslastung abhängig gemacht werden.

Beispiel

```
SELECT *
  FROM Umsätze
 WHERE Monat BETWEEN 1 AND 6
```

- Umsetzung durch Relationen-Scan

```
aktuellerScanID := open-rel-scan(Umsätze-RelationID);
aktuellerTID := next-TID(aktuellerScanID);
while (not end-of-scan(aktuellerScanID))
   aktuellesTupel := fetch-tuple(Personen-RelationID, aktuellerTID);
   if aktuellesTupel.Monat >= 1 and aktuellesTupel.Monat <= 6
      return(aktuellesTupel);
   aktuellerTID := next-TID(aktuellerScanID);
close-scan (aktuellerScanID);
```

- Umsetzung durch Index-Scan

```
aktuellerScanID := open-index-scan(Umsätze-Monat-IndexID, 1, 6);
aktuellerTID := next-TID(aktuellerScanID);
while (not end-of-scan(aktuellerScanID))
   aktuellesTupel := fetch-tuple(Umsätze-RelationID, aktuellerTID);
   return(aktuellesTupel);
   aktuellerTID := next-TID(aktuellerScanID);
close-scan (aktuellerScanID);
```

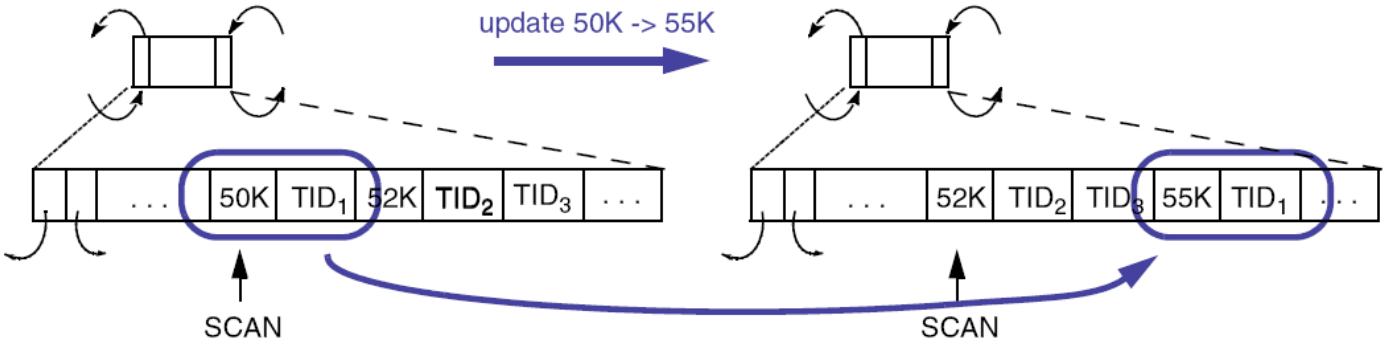
Situation

- deklarative Anweisung (SQL-Anweisung) wird satzorientiert ausgewertet
- Konflikt, falls ein zu aktualisierendes Objekt von Scan benutzt wird

Beispiel

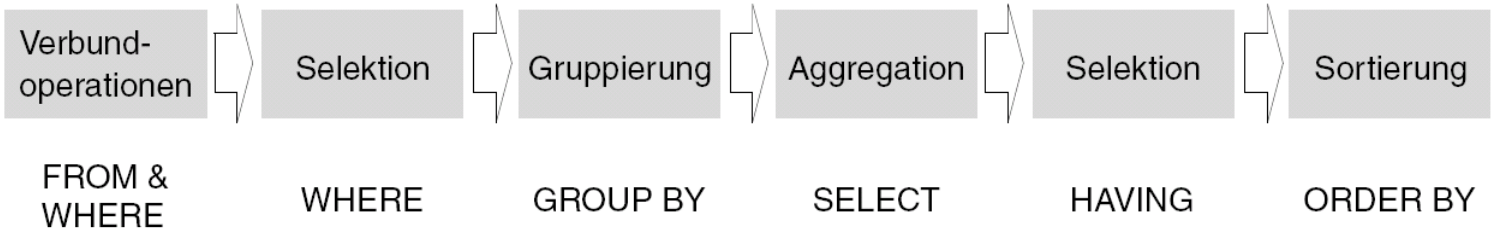
```
UPDATE PERS SET GEHALT = GEHALT * 1.1
```

- Anfrageoptimierung entscheidet, dass ein existierender Index über Gehalt zur Ausführung dieser Anweisung verwendet wird



HALLOWEEN-Problem !!! P.S.: Woher stammt der Name?

Logische Verarbeitungsreihenfolge



Hash-basierter Ansatz

- Anwendung einer Hash-Funktion auf die Werte der Gruppierungsattribute
- Hash-Tabelle hält Ergebnisse der Aggregationsfunktionen pro Kombination

Sortierungsbasierter Ansatz

- Sortierung auf Gruppierungsattributen
- Im sortiertem Datenstrom liegen Tupel mit gleichwertiger Gruppenattributen, also alle Tupel einer Gruppe, hintereinander
- Datenstrom wird eingelesen und Aggregationsspalten werden gebildet bis zum jeweilig nächsten Wertewechsel auf Gruppenattributen

Algorithmus: Sortierungsbasierter Ansatz

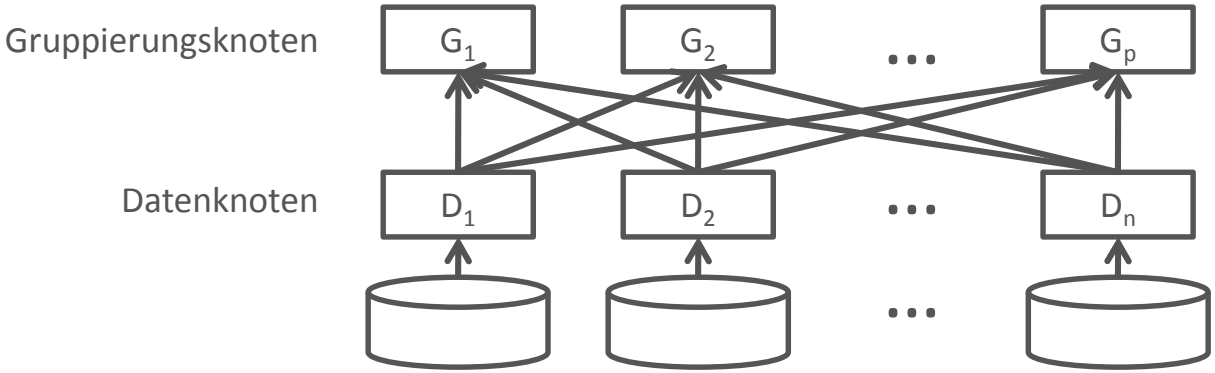
```
Input: G1,...,Gn // Gruppierungsattribute aus der GROUP BY-Klausel
       AGG(), A // Aggregationsfunktion und zu aggregierendes Attribut A
Begin
   // Sortierung des Datenstroms nach den Gruppierungsattributen
   SORT(G1,...,Gn)

   // Abarbeiten des gesamten Eingabestroms
   While (Eingabestrom noch nicht verarbeitet)
      ($g1,...,$gn,$val) := LeseNächstesTupel(Eingabestrom)

      // Innerhalb der gleichen Gruppe werden die Einträge hinzugefügt.
      If (Aktuelles Tupel hat gleiche Werte in G1,...,Gn wie letztes Tupel)
         $aggrset := $aggrset ∪ {$val};
      Else
         // Beim Wechsel einer Gruppe (oder am Ende) wird ein Ausgabetupel
         // durch die Aggregationsfunktion über die Wertemenge erzeugt
         $aggrval = AGG($aggrset)
         SchreibeNeuesTupel(Ausgabestrom, ($g1,...,$gn, $aggrval))
         $aggrset := {};
      End If
   End While
End
```

Paralleles Gruppieren

- Verteilen und Gruppieren
  - Verteilen aller Tupel jedes Datenknoten auf ein oder mehrere Gruppierungsknoten mittels Hashing
  - Lokales Gruppieren auf jedem Gruppierungsknoten
  - Lokales Gruppierungsverfahren freiwählbar
  - Anzahl von Daten- und Gruppierungsknoten kann unterschiedlich sein



- Mit Vorgruppieren
  - Lokales Vorgruppieren auf Datenknoten
  - Verteilen aller Tupel auf ein oder mehrere Gruppierungsknoten mittels Hashing
  - Zusammenführen auf Gruppierungsknoten durch erneutes Gruppieren
  - Aggregatbildung muss entsprechend angepasst werden

z.B.: AVG(x) AS xavg =>	SUM(x) AS xsum, COUNT(x) AS xcnt	auf Datenknoten
	SUM(xsum)/SUM(xcnt) AS xavg	auf Gruppierungsknoten

Prinzip

- ähnlich zur Projektion: Gruppierung entspricht Projektion mit Aggregation Q(R) sei Attribut von R, auf das eine Aggregationsfunktion angewendet werden soll

MIN, MAX

- Parallele Berechnung immer möglich
- MIN(Q(R)) => MIN (MIN ($Q(R_1)$), ... , MIN ($Q(R_n)$))
- MAX(Q(R)) => MAX (MAX ($Q(R_1)$), ... , MAX ($Q(R_n)$))
- Parallele Berechnung der lokalen Minima/Maxima

SUM, COUNT, AVG

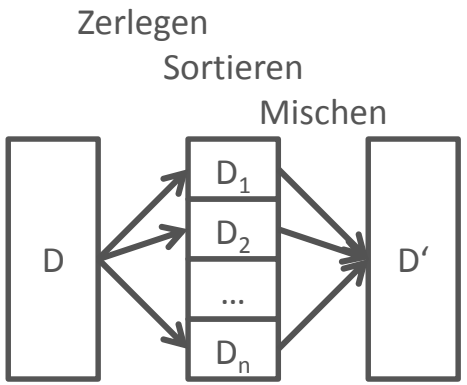
- nur anwendbar, wenn keine Duplikateliminierung erforderlich ist
- SUM (Q(R)) => Σ SUM ($Q(R_i)$)
- COUNT (Q(R)) => Σ COUNT ($Q(R_i)$)
- AVG (Q(R)) => SUM (Q(R)) / COUNT (Q(R))

Wenn möglich Sortieren durch Index-Scan

- Index auf dem Sortierattribut notwendig
- Einfaches Auslesen des Index
- Kein explizites Sortieren notwendig
- Eventuell müssen weitere Attribute noch geladen werden (FETCH)

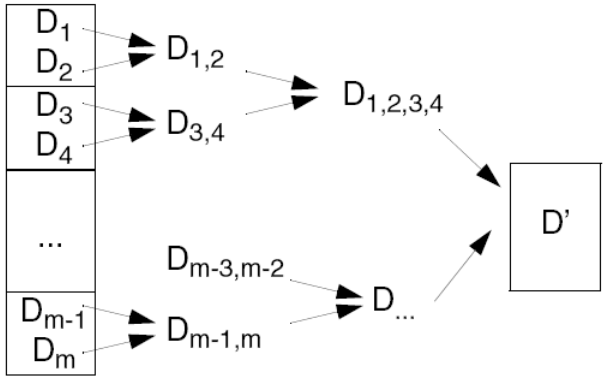
Allgemein: Externes Sortieren

- Zerlegung der Eingabe in mehrere Läufe (runs)
- Sortieren und Zwischenspeichern der sortierten Läufe
- sukzessives Mischen, bis ein sortierter Lauf entsteht
- Blockgröße der Größe des zur Verfügung stehende Arbeitsspeichers ab
- Passen Daten in den Arbeitsspeicher, entfällt das Mischen

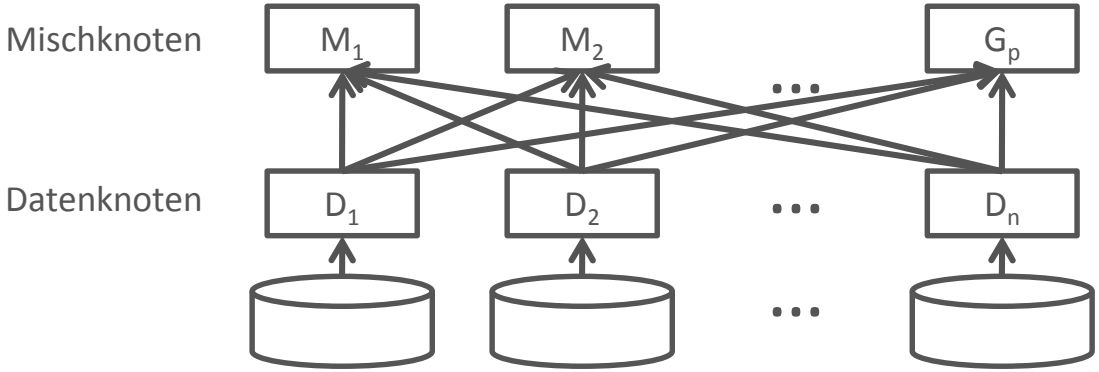


Paralleles Sortieren

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



- lokale Sortierung der Partitionen in den Datenknoten
- Umverteilung der sortierten Läufe unter p Mischknoten, über eine dynamische Bereichsfragmentierung auf dem Sortierattribut
- paralleles Mischen in den p Mischknoten
- partitionierte Ausgabe

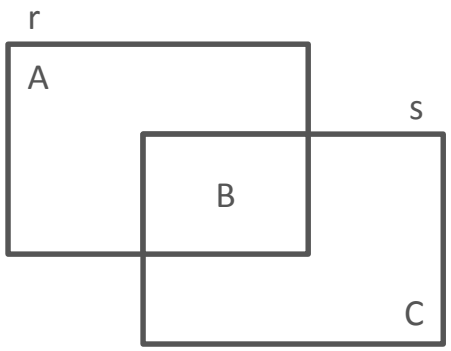


Verbundoperatoren (Binäre Operatoren)

Binäre Operatoren

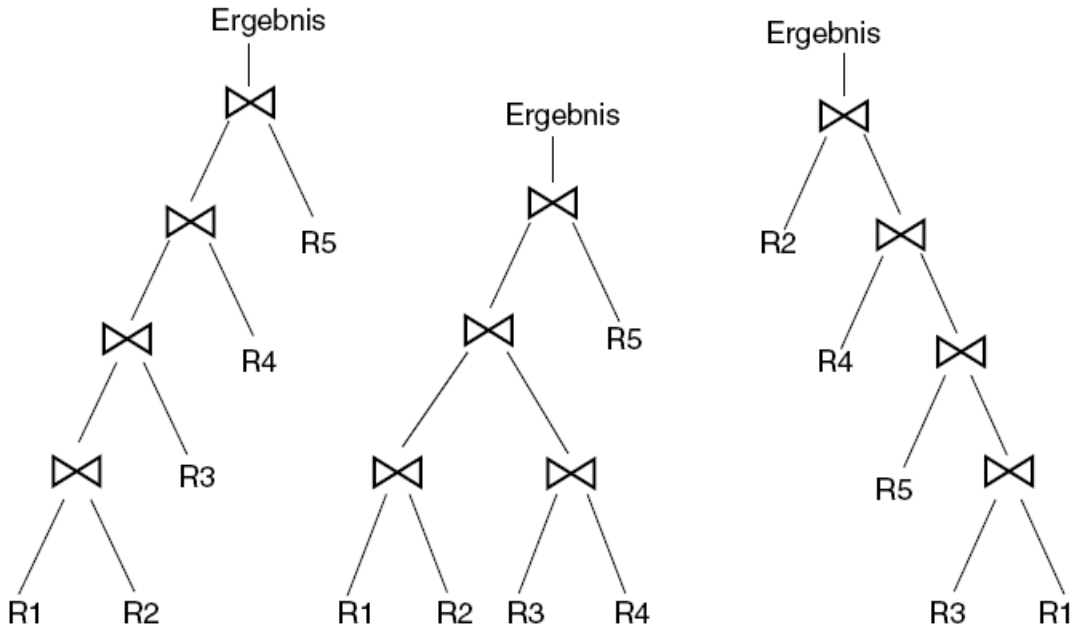
- Übereinstimmung in allen Attributen
  - A: Differenz r−s
  - B: Schnitt r ∩ s
  - C: Differenz s−r
  - A ∪ C: symmetrische Differenz (r−s)∪(s−r)
  - A ∪ B ∪ C: Vereinigung r ∪ s

- Übereinstimmung auf einigen Attributen
  - A: linksseitig Anti-Semi-Verbund
  - B: Verbund
  - C: rechtsseitig Anti-Semi-Verbund
  - A ∪ B: linksseitig äußerer Verbund (Left-Outer-Join)
  - A ∪ C: Anti-Verbund
  - B ∪ C: rechtsseitig äußerer Verbund (Right-Outer-Join)
  - A ∪ B ∪ C: vollständig äußerer Verbund (Full-Outer-Join)



Verbund über mehrere Relationen (n-Wege-Verbund)

- Zerlegung in n-1 Zwei-Wege-Verbunde
- Anzahl der Verbundreihenfolgen ist abhängig von den gewählten Verbundattributen
- n! verschiedene Reihenfolgen möglich
- optimale Auswertungsreihenfolge abhängig von
  - Planoperatoren
  - „passende“ Sortierordnungen für Verbundattribute
  - Größe der Operanden usw.
- verschiedene Verbund-reihenfolgen mit Zwei-Wege-Verbunden (n=5)



Eigenschaften der Verbundoperation

- teuer und häufig -> Optimierungskandidat !!!
- typisch: Gleichheitsverbund; allgemeines Verbundprädikat eher selten
- Standardszenario

```
SELECT *
FROM R, S
WHERE R.VA Θ S.VA // Verbundprädikat
AND P(R.SA) // lokale Selektionen
AND P(S.SA)
```

Mögliche Zugriffspfade

- DB-Scan über R uns S
- Scans über IR(R.VA) und IS(S.VA)
  - Sortierreihenfolge nach R.VA und S.VA !!!
- Scans über IR(R.SA) und/oder IS(S.SA)
  - schnelle Selektion für R.SA und S.SA !!!
- ... beliebige andere Kombinationen

Annahmen

- Sätze in R und S sind nicht nach den Verbundattributen geordnet
- es sind keine Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden

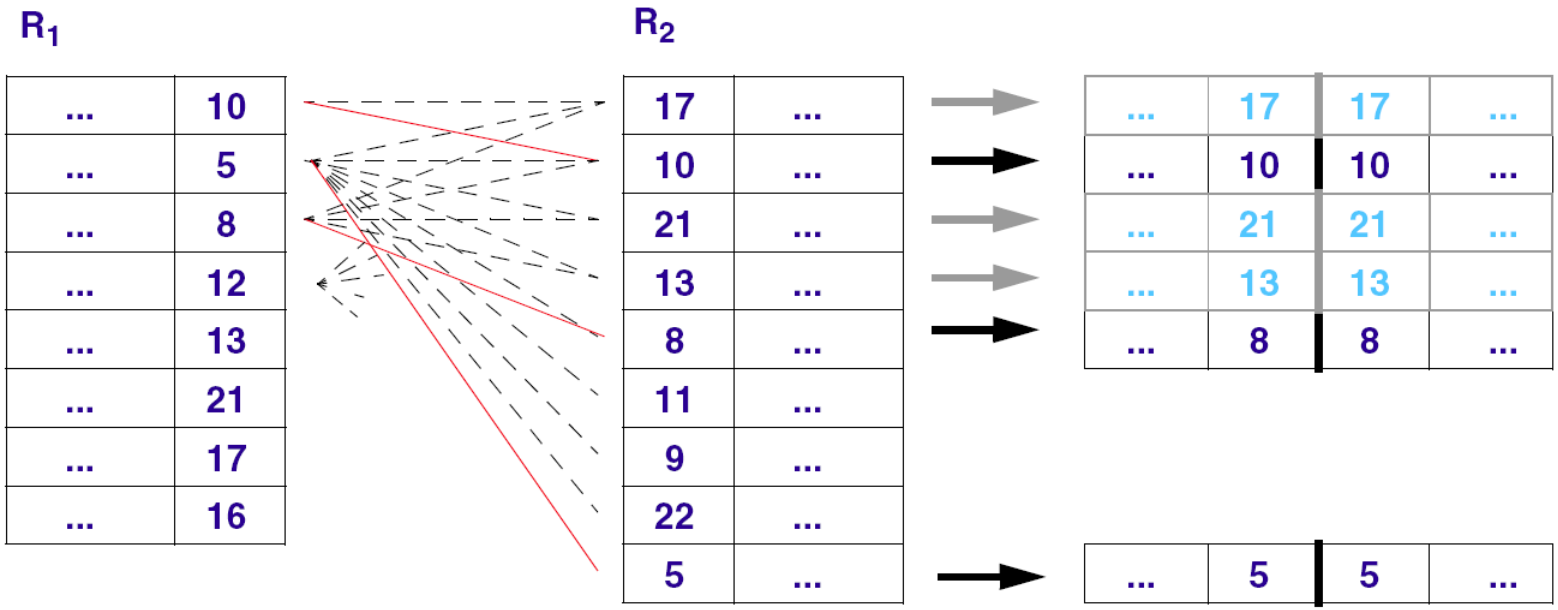
Algorithmus für Θ-Verbund

```
Scan über S
für jeden Satz s, falls P_S gilt:
   Scan über R
   für jeden Satz r, falls P_R AND (r.VA Θ s.VA) gilt:
      übernehme kombinierten Satz (r, s) in das Ergebnis
```

Komplexität

- $O(N^2)$

Beispiel



- Annahme: Fremdschlüssel-Primärschlüssel-Beziehung!

Annahmen

- es sind Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden

Algorithmus für Θ-Verbund mit Indexzugriff

```
Scan über S
für jeden Satz s, falls PS gilt:
   ermittle mittels IR(VA) alle TIDs für Sätze mit r.VA = s.VA
   für jedes TID:
   hole Satz r, falls PR:
      übernehme kombinierten Satz (r, s) in das Ergebnis
```

Merke

- eigentlich wird block- bzw. seitenweise vorgegangen
- dieses Vorgehen widerspricht jedoch dem Gedanken der Schichtenarchitektur
- gleiches Prinzip wird benutzt um Mengenoperationen zu realisieren

Annahmen

- es sind Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden

Algorithmus zum Ausnützen von Indexstrukturen $I_R(R.VA)$ und $I_S(S.VA)$

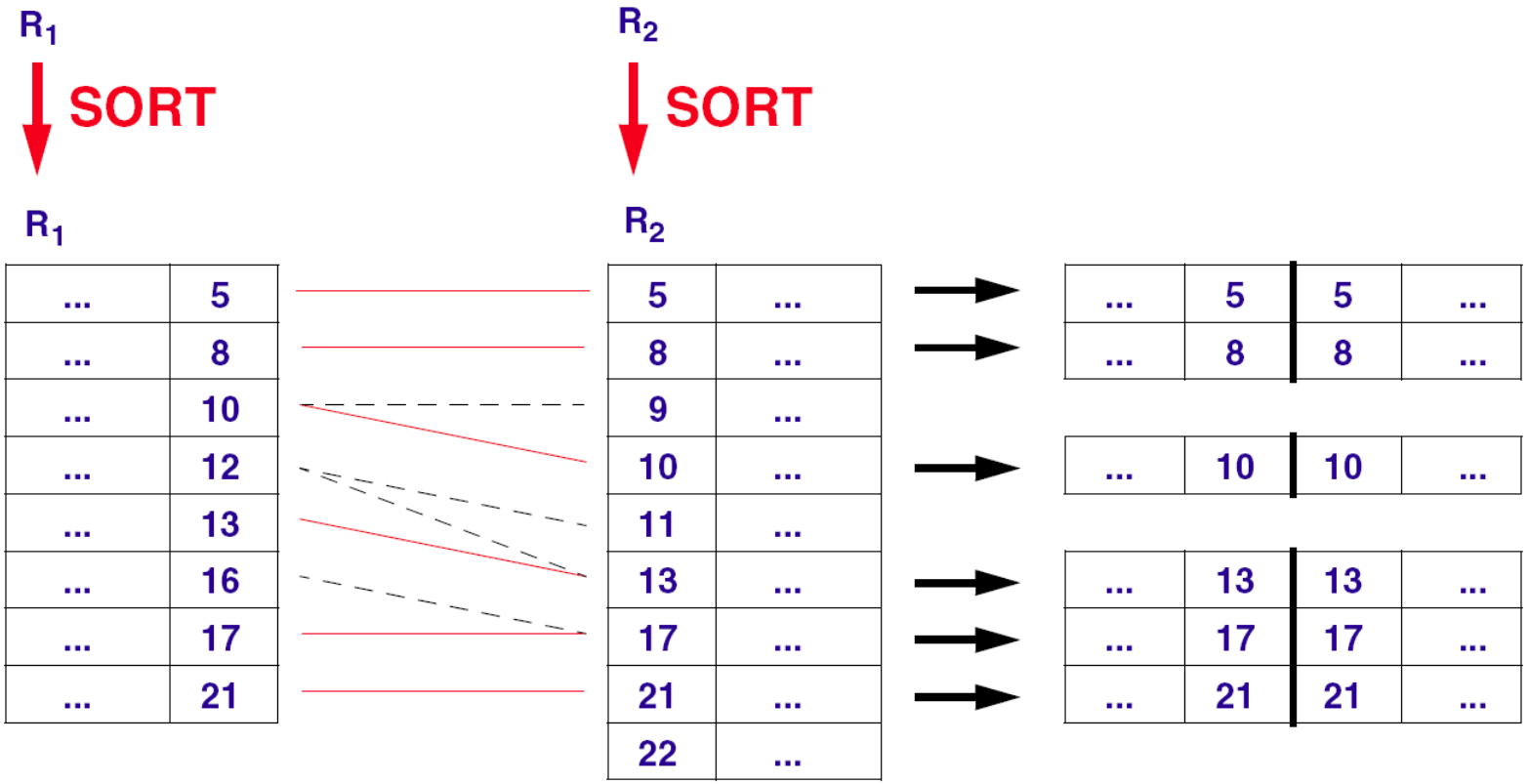
- Phase 1:
  Sortierung von R und S nach R.VA und S.VA (falls nicht bereits vorhanden), dabei frühzeitige Eliminierung nicht benötigter Tupel (durch Überprüfung von PR, PS)
- Phase 2:
  Schritthaltende Scans über sortierte R- und S-Relationen mit Durchführung des Verbundes bei r.VA = s.VA
- Pseudocode:

```
Schritthaltende Scans über I_R(VA) und I_S(VA):
für jeweils zwei Schlüssel aus I_R(VA) und I_S(VA), falls r.VA = s.VA:
   hole mit den zugehörigen TIDs die Tupel, falls P_R und P_S:
      übernehme kombinierten Satz (r, s) in das Ergebnis
```

Komplexität

- O(N log N)

Beispiel


R1
SORT
R2
SORT
R1
R2
... 5
... 8
... 10
... 12
... 13
... 16
... 17
... 21
5 ...
8 ...
9 ...
10 ...
11 ...
13 ...
17 ...
21 ...
22 ...
... 5 5 ...
... 8 8 ...
... 10 10 ...
... 13 13 ...
... 17 17 ...
... 21 21 ...

Schritt 1

- Abschnittsweises Lesen der (kleineren) Relation R
- Aufbau einer Hash-Tabelle mit $h_A(r(VA))$ nach Werten von R(VA)
- Aufteilen in p Abschnitte R_i $(1 \leq i \leq p)$ derart, dass
  - jeder der p Abschnitte in den verfügbaren Hauptspeicher passt
  - jeder Satz, der gehasht wird, P_R erfüllt

Schritt 2

- Überprüfung (Probing) für jeden Satz von S mit P_S
- im Erfolgsfall Durchführung des Verbundes

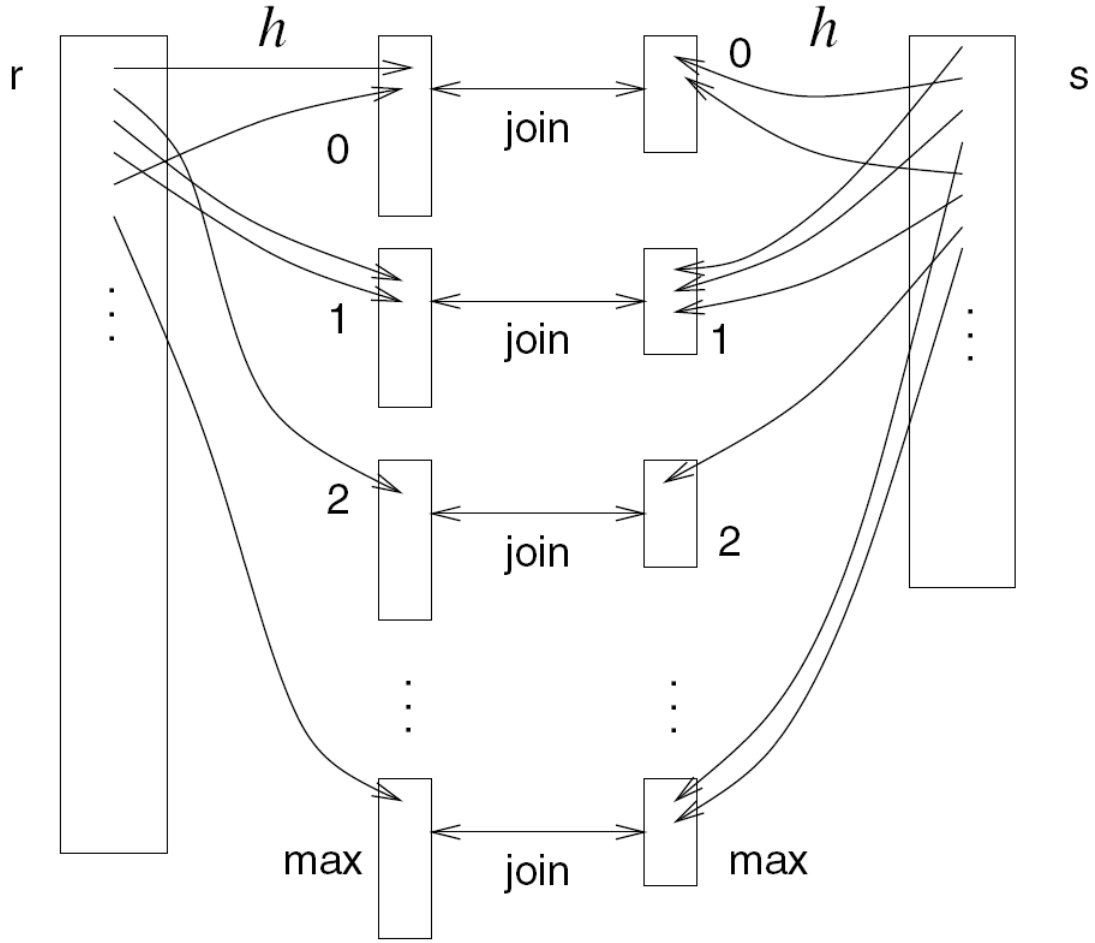
Schritt 3

- Wiederhole Schritt 1 und 2 solange, bis alle p Abschnitte bearbeitet

Komplexität

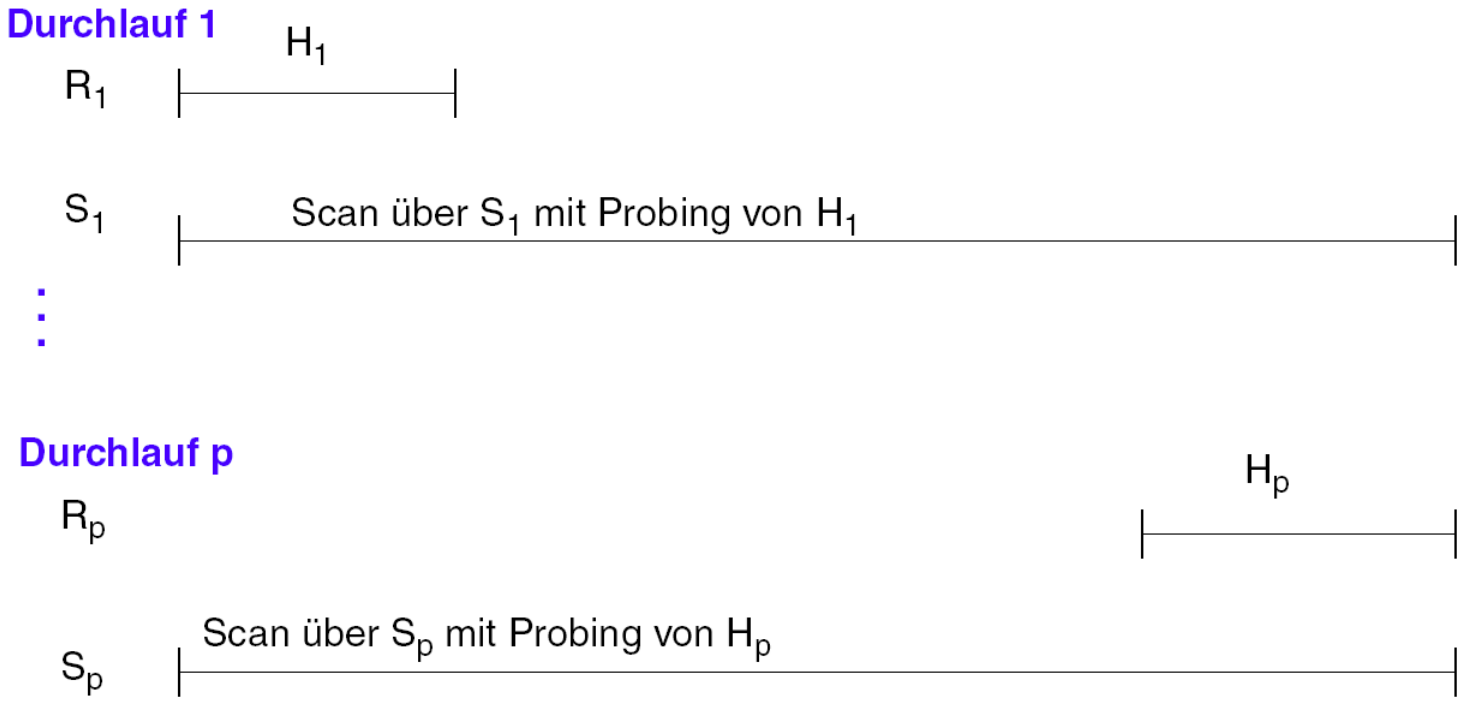
- O(p * N)
- Idealfall: R passt in den Hauptspeicher, d.h. p=1

Illustration der Partitionierung



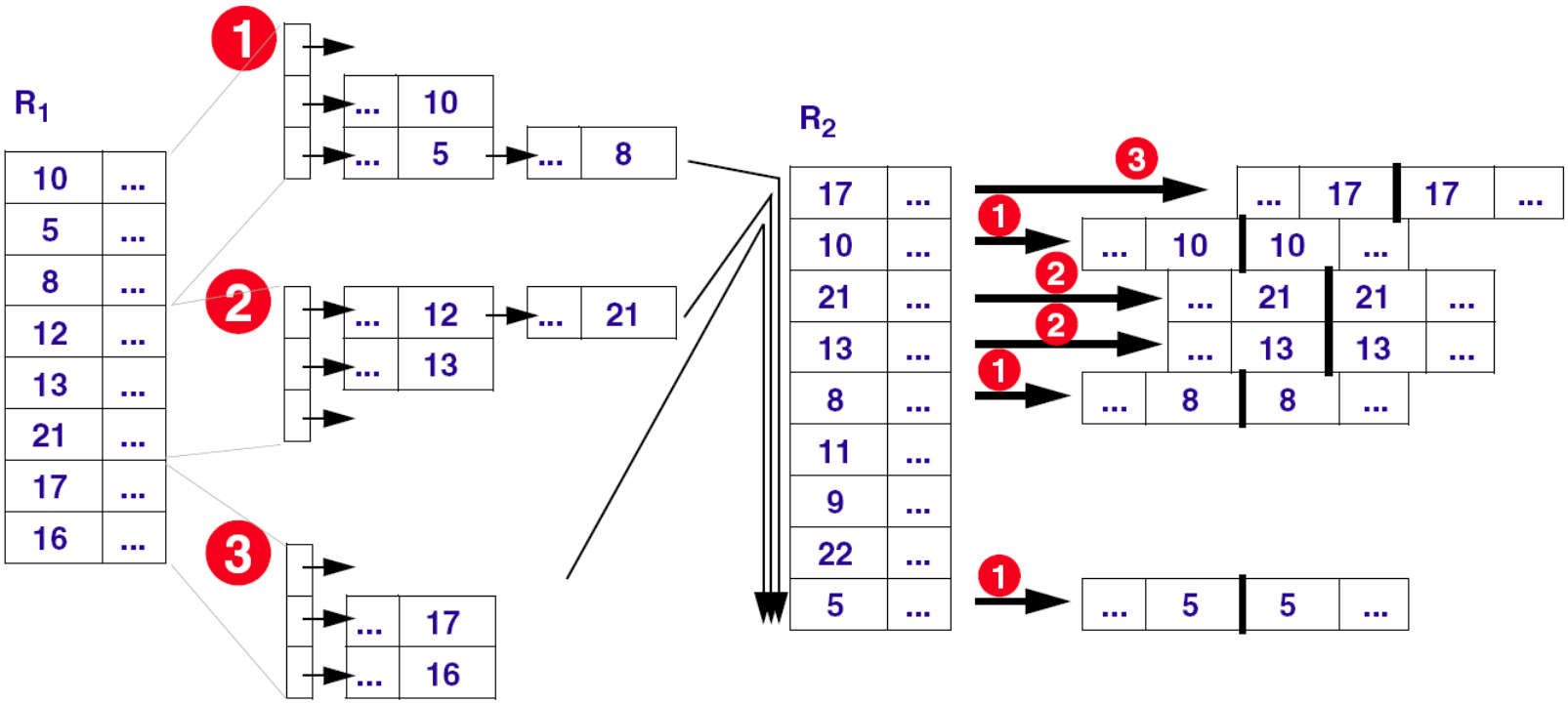
Aufbau der Hash-Tabelle und Probing

- Hash-Tabellen $H_i (1 \leq i \leq p)$ werden schrittweise im Hauptspeicher aufgebaut
- nach jedem Durchlauf von S wird die aktuelle Hash-Tabelle wieder gelöscht



Beispiel

- Voraussetzung: Hauptspeicherkapazität = 3 Tupel
- Hashing von R_1 mit h(x) = x mod 3



Nachteil des Classic Hashing

- der Verbundpartner S muss p-mal gelesen werden
- warum ist nicht auch S (analog zu R) partitionierbar?

Verbesserung: Simple-Hashing

- Hashing von R nicht nach der Reihenfolge der Tupel, sondern wertemäßig durchführen
  -> Partitionierung von R nach Werten von R.VA
- Aber
  - das ist nicht einfach, da üblicherweise keine Gleichverteilung der Werte vorliegt
  - Heranziehen von Statistiken (ins. Histogramme !)
- Zum Probing auch S mit den gleichen Kriterien partitionieren

Vielzahl unterschiedlicher Hash-Verfahren

- Simple-Hashing ist ein Beispiel

Schritt 1

- führe Scan auf kleinerer Relation R aus
- überprüfe PR und wende auf jedes qualifizierte Tupel r die Hash-Funktion h_P an
- Fällt h_P(r.VA) in den gewählten Bereich, trage es in Hi ein
- Anderenfalls speichere r in eine temporäre Zwischendatei für „übergangene“ r-Tupel

Schritt 2

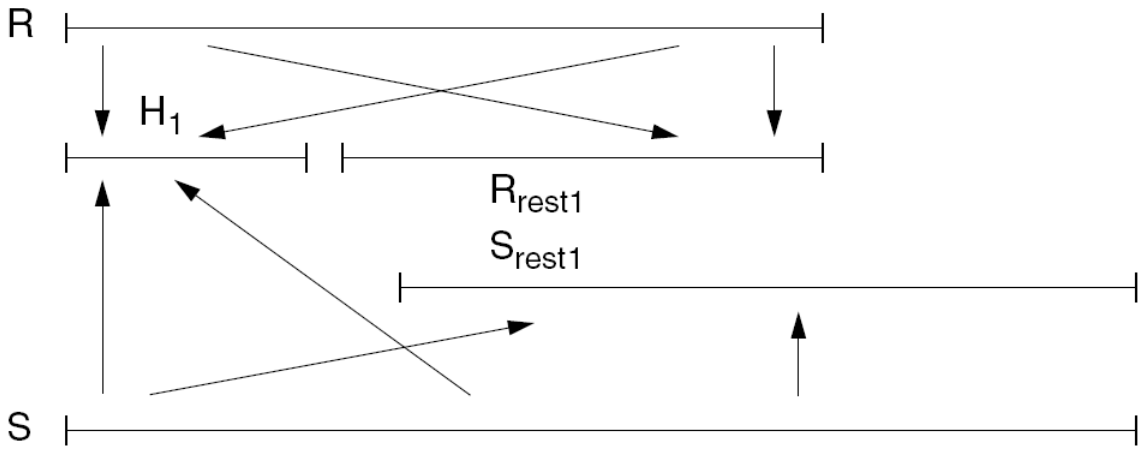
- führe Scan auf S aus
- überprüfe P_S und wende auf jedes qualifizierte Tupel s die Hash-Funktion h_P an
- Fällt h_P(s.VA) in den gewählten Bereich, suche in Hi einen Verbundpartner (Probing)
- Falls erfolgreich, bilde ein Verbundtupel und ordne es dem Ergebnis zu
- Anderenfalls speichere s in eine temporäre Zwischendatei für „übergangene“ s-Tupel

Schritt 3

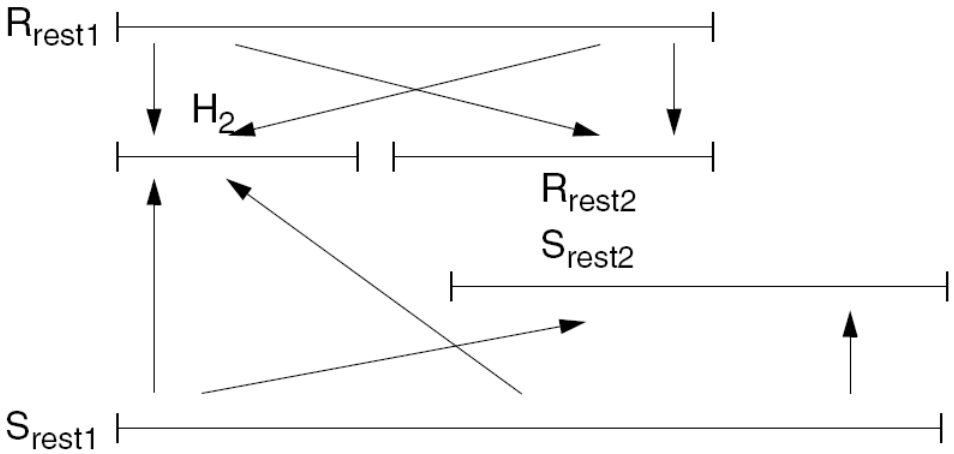
- Wiederhole Schritt 1 und 2 mit den bisher übergangenen Tupeln solange, bis R erschöpft ist (die Überprüfung von PR und PS nicht mehr erforderlich)

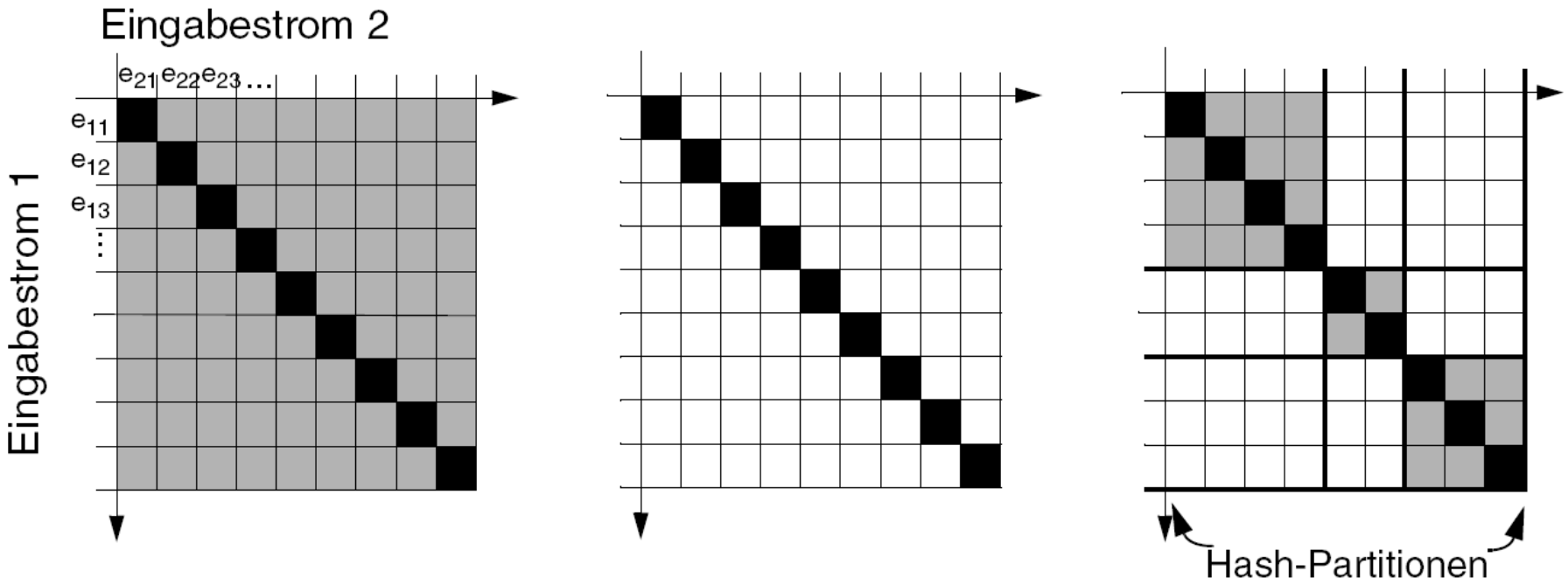
Illustration

Durchlauf 1



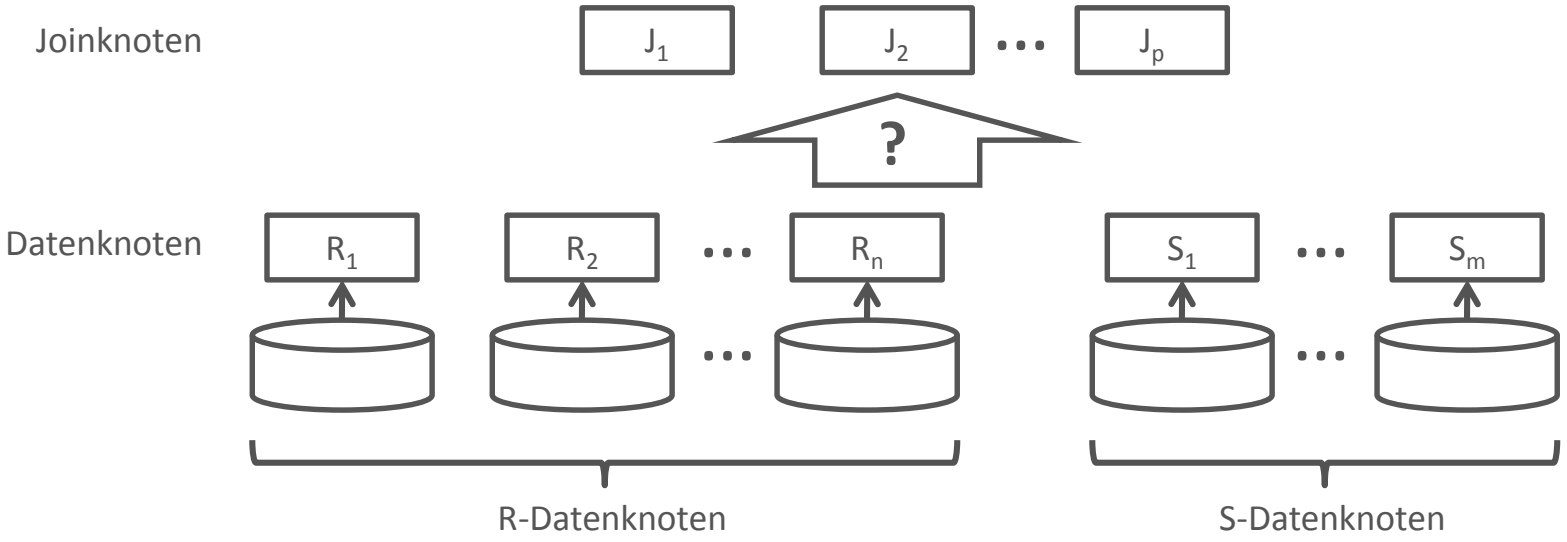
Durchlauf 2




Eingabestrom 2
e21 e22 e23 ...
e11
e12
e13
...
Eingabestrom 1
Hash-Partitionen
Schleifeniteration
Mischverbund
Hash-Verbund
Elementvergleich
Elementvergleich, der zu einem Tupelverbund führt

Szenario

- Gleichheitsverbund zwischen R und S
  - $R = \cup (R_1, R_2, \ldots, R_n)$
  - $S = \cup (S_1, S_2, \ldots, S_m)$
- S sei kleiner als R
- Verbundberechnung auf p Joinprozessoren



Prinzip des parallelen Hash-Verbunds

- Umverteilung der kleineren Relation S über Hash-Funktion h() auf Verbundattribut
- In Join-Prozessoren kommen eingehende Tupel in Hauptspeicher-Hash-Tabelle
- Umverteilung der zweiten Relation R auf die Join-Prozessoren unter Anwendung der Hash-Funktion
- Probing: für eingehende Tupel Verbundpartner in Hash-Tabelle ermitteln

Merkmale

- Sequenzialisierung der Scan-Phasen
- Vorteil: Reduzierung des Umverteilungsaufwands für R durch Anwendung von Bitvektor-Filterung möglich
- Pipeline-Parallelität in Building- und Probing-Phase möglich
- Überlaufbehandlung erforderlich, falls S-Partitionen nicht vollständig im Hauptspeicher Platz finden (=> dreistufige Partitionierung)

Replizierter Verbund „broadcast join“

- Partitionierung bei kleinen Relationen lohnt sich nicht....
- Zuteilung einer Kopie des kleineren Verbundpartners zu den Partitionen des größeren Verbundpartners
  - Vorteil: keine Relation muss nach dem Verbundattribut partitioniert sein

Einseitig umverteilender Verbund „directed join“

- Einer der beiden Verbundpartner ist nach dem Verbundattribut partitioniert
- Partitionen des anderen Verbundpartners werden zur Laufzeit neu nach dem Verbundattribut partitioniert
- Beispiel
  - Auftragsrelation sei partitioniert nach dem Kundenschlüssel
  - Repartitionierung nach dem Attribut O_ORDERKEY

Vollständig umverteilender Verbund „repartitioned join“

- beide Verbundpartner werden nach dem Verbundattribute neu partitioniert
- hohe Kommunikationskosten -> Vermeiden!

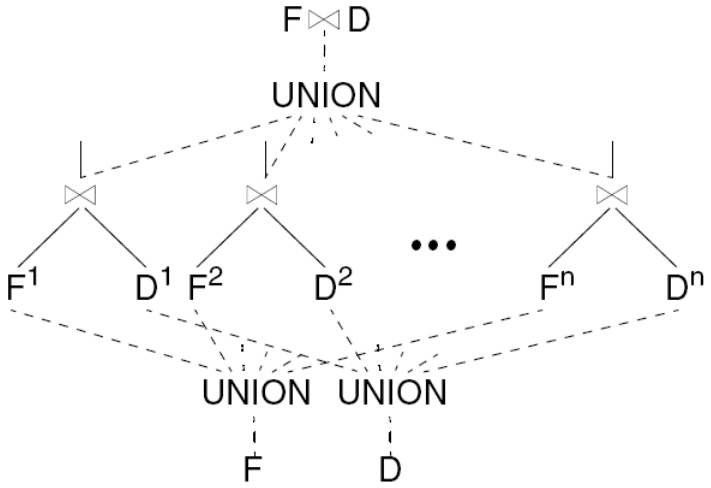
Partitionslokaler Verbund (›co-located joins‹)

- Verbundattribut bei beiden Verbundpartnern ist gleichzeitig das Verbundattribut
- maximale Parallelität bei minimalem Kommunikationsaufwand zwischen den parallel ablaufenden Verbundoperatoren
- Beispiel
  - Faktentabelle und Auftragsrelation (ORDERS) partitioniert nach L_ORDERKEY bzw. O_ORDERKEY
- partitionslokaler Verbund bei folgender Anfrage

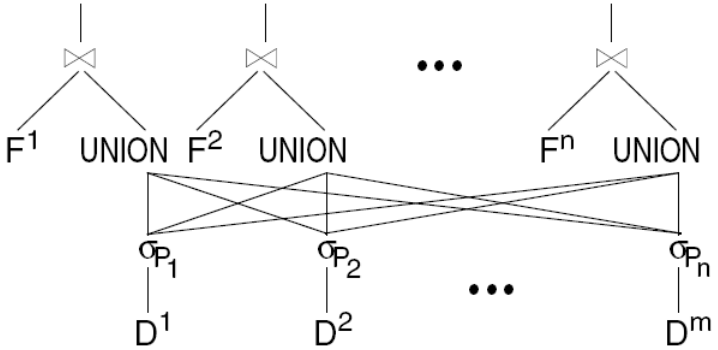
```
SELECT O_ORDERPRIORITY, SUM(L_QUANTITY) AS SUM_QUAN
  FROM TPCD.LINEITEM, TPCD.ORDERS
 WHERE L_ORDERKEY = P_ORDERKEY
 GROUP BY O_ORDERPRIORITY;
```

Beispiel

- F Faktentabelle
- F^k Partitionen der Faktentabelle ($1 \le k \le n$)
- P_k Partitionierungsprädikate der Faktentabelle ($1 \le k \le n$)
- D Dimensionstabelle
- D^k Partitionen der Dimensionstabelle ($1 \le k \le n/m$)



a) Partitionslokaler Verbund



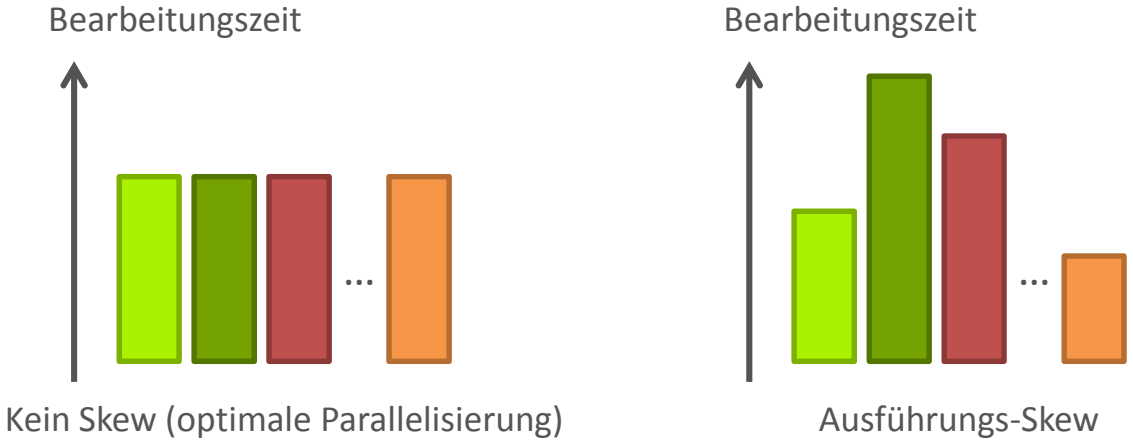
b) Einseitig umverteilender Verbund

Beobachtung

- ungleiche Bearbeitungszeit von Teiloperationen (Ausführungs-Skew) beeinträchtigt Parallelisierung

Ursache

- Ausführungs-Skew geht häufig auf Daten-Skew zurück:
  unterschiedlich große Datenmengen pro Teiloperation aufgrund ungleichförmiger Verteilung von Attributwerten und Tupeln



Datenverteilungs-Skew (tuple placement skew)

- unterschiedliche Partitionsgrößen
- ungleichmäßige Dauer von Scan-Operationen
- Behandlung:
  möglichst gute Kenntnis der Werteverteilung für Verteilattribut
  - Histogramme
  - stichprobenartig über Sampling-Verfahren
  - Bestimmung während des Sortierens bei Sort-Merge-Joins

Umverteilungs-Skew (redistribution skew)

- Verteilungsfunktion führt zu unterschiedlichen Fragmentgrößen
- Behandlung:
  wie Datenverteilungs-Skew

Selektivitäts-Skew

- unterschiedliche Trefferhäufigkeiten pro Rechner
  (z.B. Bereichsanfragen bezüglich Verteilattribut bei Bereichspartitionierung)
- Behandlung:
  kaum behandelbar, da durch Anfrage und Datenverteilung bestimmt

Join-Produkt-Skew

- unterschiedliche Join-Selektivität pro Knoten
- Behandlung:
  - Abschätzung der Gesamtgröße des Join-Ergebnisses sowie der dabei entstehenden Werteverteilung für das Join-Attribut
  - Bereichpartitionierung festlegen, die für jeden der p Join-Prozessoren ein etwa gleich großes Teilergebnis liefert

Star-Join

Voraussetzungen für einen Star-Join

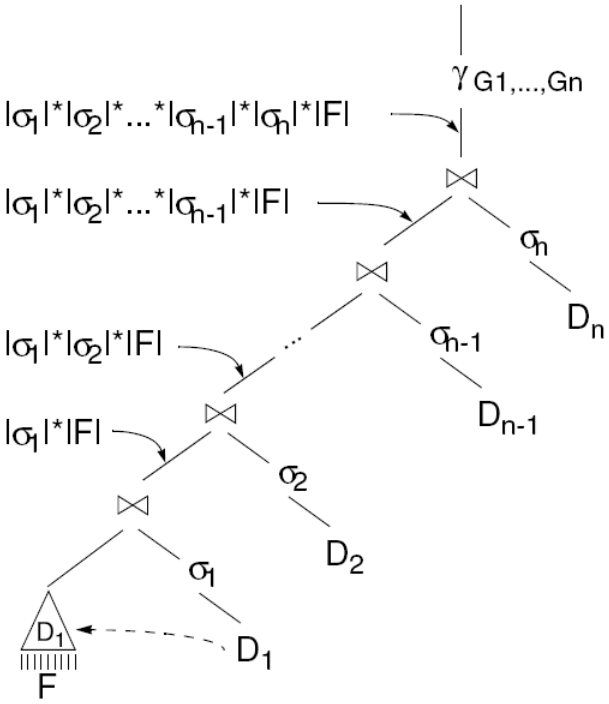
- Faktentabelle ist immer Bestandteil einer Star-Query
- Verbund mit Dimensionstabelle reflektiert eine indirekte Selektion über Einschränkungen auf der Dimensionsrelation
  (“Erweiterung der Faktentabelle um dimensionale Attribute on the fly...”)

Probleme bei einer klassischen Verbundoperation

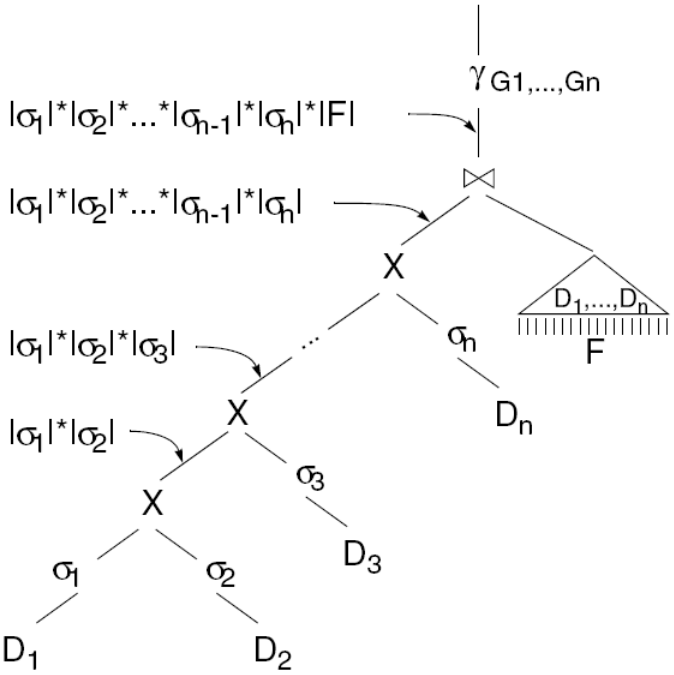
- Faktentabelle ist in der ersten Verbundoperation bereits enthalten
- Dimensionstabellen werden sukzessive verknüpft
- Größe des Datenstromes startet mit |F| und nimmt erst schrittweise ab
- Nur ein einziger ein-dimensionaler Index kann nur in der ersten Verbundoperation verwendet werden

Star-Join mit Restrukturierung

- Bildung des kartesischen Produktes der Dimensionstabellen
- Zugriff auf Faktentabelle mit n-fachen Index
- Nachteil: kartesisches Produkt kann sehr groß werden!



a) Operatorengraph einer Star-Query



b) Operatorengraph einer Star-Query nach Bildung des kartesischen Produktes

Idee

- Beibehaltung der regulären Verbundstrategie
- Reduktion der Größe der Faktentabelle in einer Vorstufe (Vorselektion)
- Benötigt: Jeweils ein Index auf den Fremdschlüsselattributen von F

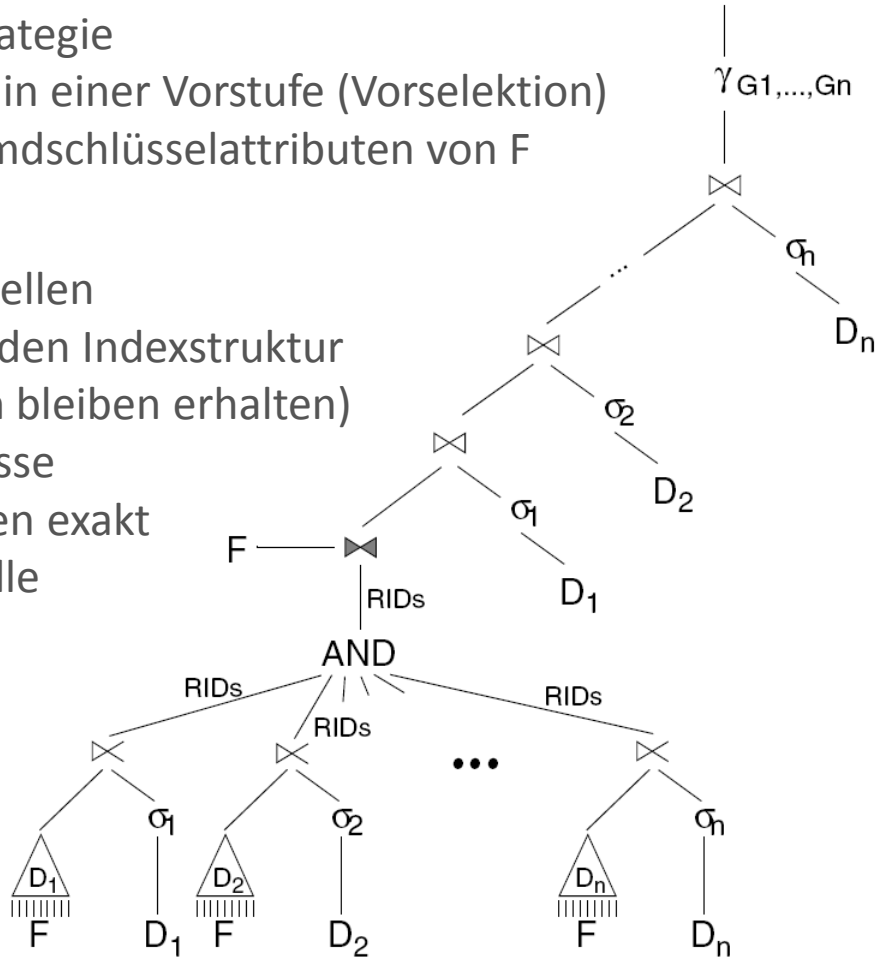
Realisierung

- lokale Selektion auf den Dimensionstabellen
- Semi-Verbund mit der korrespondierenden Indexstruktur der Faktentabelle (interne Satzadressen bleiben erhalten)
- Schnittmengenbildung aller Teilergebnisse
- Resultierende Satzadressen identifizieren exakt die benötigten Einträge der Faktentabelle

⋈ wertebasierter Gleichheitsverbund

⋉ Semi-Verbund

⋈ RID-basierter ›Verbund‹ (FETCH-Operator)

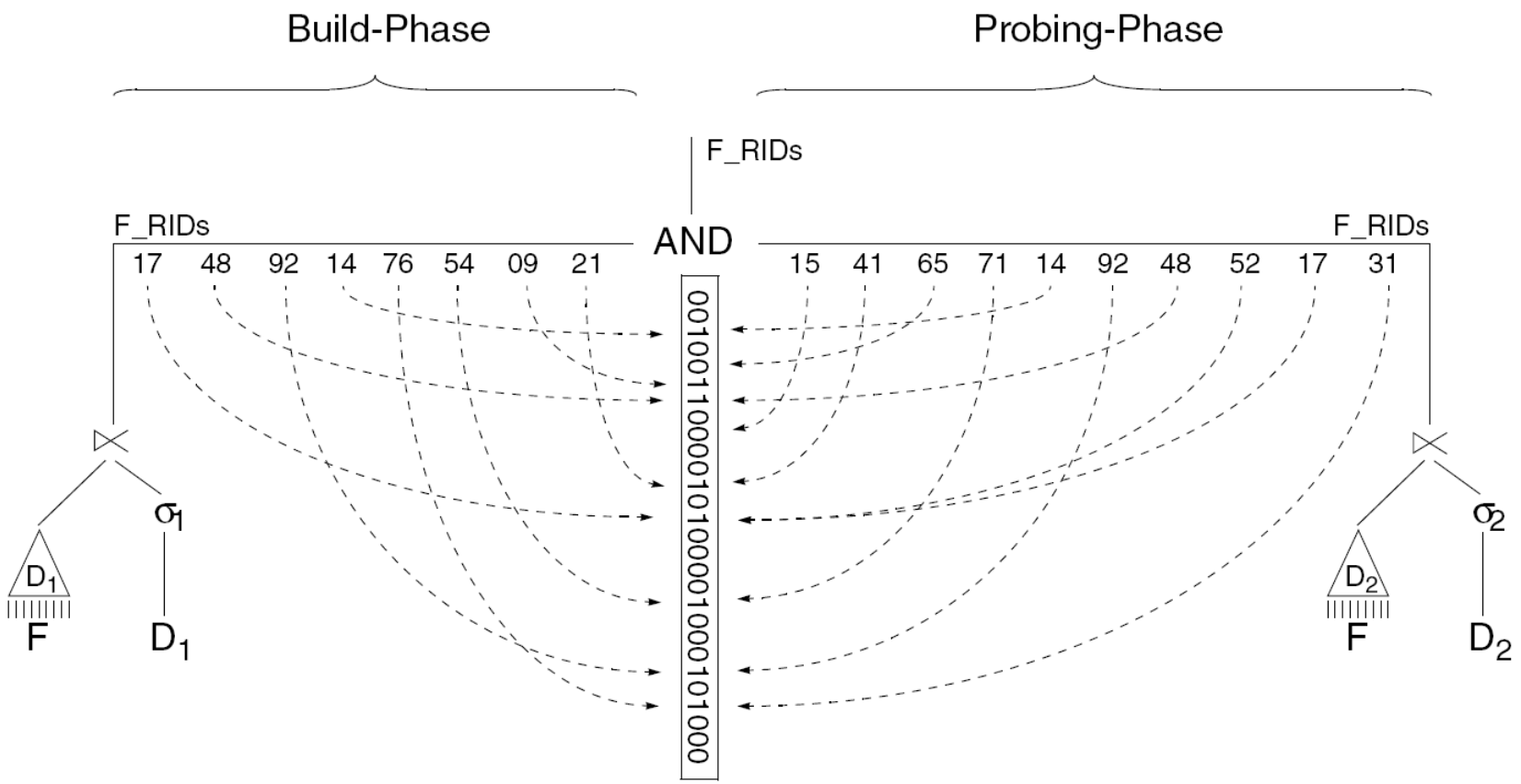


Bewertung der Vorselektion

- Vorteil: Faktentabelle besitzt vor der Verbundoperation bereits endgültige Größe
- Nachteil: extrem aufwändige Schnittmengenbildung der RID-Listen

Einführung einer Unschärfe: Bloom-Filtertechnik

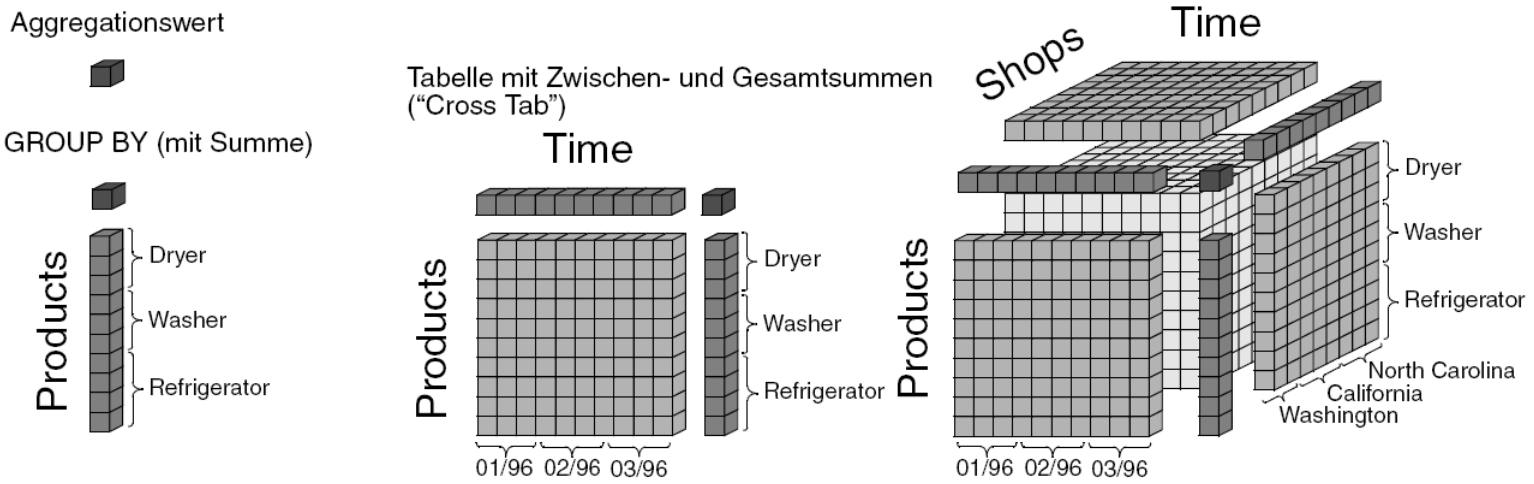
- Erzeugung einer mit 0 initialisierten Bitliste im Hauptspeicher
- Aufbauphase (build phase)
  - Hashfunktion bestimmt Position der Bitliste für jeden RID-Eintrag: auf 1 setzen
- Testphase (probing phase)
  - Verwurf eines RID, falls korrespondierender Bitlisteneintrag 0 ist
  - anderfalls: Aufnahme in das Ergebnis (unscharf!)
- Beispiel
  - Build Phase:
    h(RID 14)=3, h(RID 21)=12
  - Probing Phase:
    h(RID 14)=3 --> korrektes Ergebnis, h(RID 65)=4 --> korrekter Verwurf,
    h(RID 31)=25 --> fälschlicherweise Übernahme in Ergebnis
  - Güte abhängig von der Größe der Bitliste; Korrektur durch echte Verbundoperationen


Build-Phase
Probing-Phase
F_RIDs
F_RIDs
AND
F_RIDs
17 48 92 14 76 54 09 21
15 41 65 71 14 92 48 52 17 31
0010011000010100001000101000
σ1
D1
F
D1
σ2
D2
F
D2

Multidimensionale Gruppierung

CUBE-Operator: Erweiterung für die GROUP BY-Klausel:

- Abkürzung für die Aufzählung aller 2^n möglichen Gruppierungskombinationen, d.h.
  `GROUP BY CUBE(A,B) ist äquivalent zu GROUP BY GROUPINGS SETS ((A,B),(A),(B),())`
- Nulldimensionaler Datenwürfel: CUBE()
  ein einziger Aggregationswert / eine Zelle (nicht vorhandene Group-By Klausel)
- Eindimensionaler Datenwürfel: CUBE(Article)
  eine Wertezeile und eine Zelle
- Zweidimensionaler Datenwürfel: CUBE(Article, Day)
  eine Fläche, zwei Wertezeilen, eine Zelle (Kreuztabelle mit “Totals”)



Beispiel

```
SELECT L_SHIPMODE, L_SHIPINSTRUCT,
       SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,
       GROUPING(L_SHIPMODE) AS GRP_SHIPMODE,
       GROUPING(L_SHIPINSTRUCT) AS GRP_SHIPINSTRUCT
  FROM TPCD.LINEITEM  GROUP BY CUBE(L_SHIPMODE, L_SHIPINSTRUCT);
```

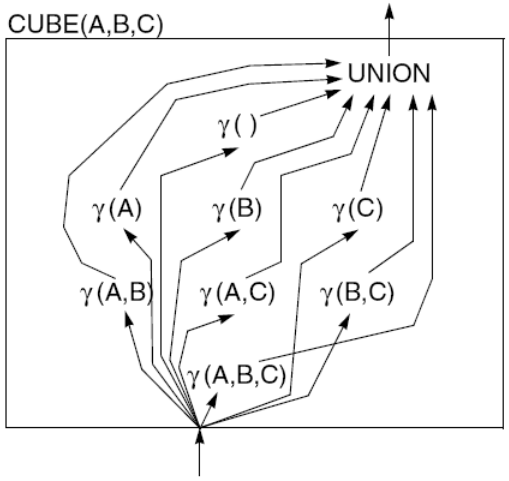
L_SHIPMODE	L_SHIPINSTRUCT	SUM_CHARGE	GRP_SHIPMODE	GRP_SHIPINSTRUCT
AIR	COLLECT COD	761624782,329	0	0
AIR	DELIVER IN PERSON	754419485,270	0	0
AIR	NONE	763523815,451	0	0
AIR	TAKE BACK RETURN	769723164,447	0	0
AIR	-	3049291247,498	0	1
MAIL	COLLECT COD	760829060,943	0	0
MAIL	DELIVER IN PERSON	765447918,010	0	0
MAIL	NONE	756568001,823	0	0
MAIL	TAKE BACK RETURN	767495428,550	0	0
MAIL	-	3050340409,327	0	1
TRUCK	COLLECT COD	760705270,142	0	0
...				
-	COLLECT COD	5334791395,658	1	0
-	NONE	5340588680,414	1	0
-	TAKE BACK RETURN	5364491060,031	1	0
-	-	21356601173,078	1	1

Problem

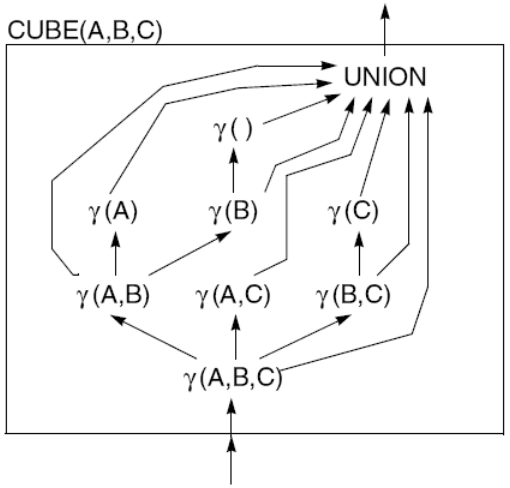
- CUBE() resultiert in 2^n Gruppierungskombinationen

Naive Variante

- 2^n-fache Ausführung der Anfrage und Bildung der Vereinigungsmenge
- Verbesserung durch Ausnützen der direkten Ableitbarkeit



a) Naiver Ansatz



b) Ausnützen der direkten Ableitbarkeit

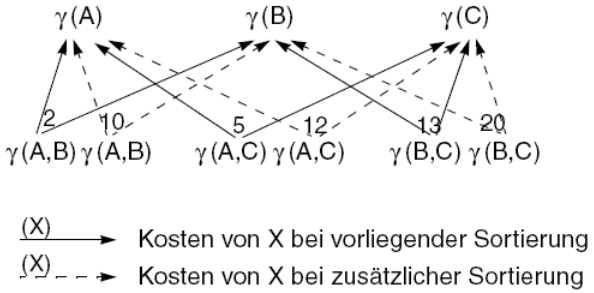
Problem

- Datenbestand muss umsortiert werden, um unterschiedliche Gruppierungskombinationen zu berechnen
- Beispiel
  - Datenbestand ist sortiert nach ABC
  - γ(AB) oder γ(AC) kann ohne Sortierung berechnet werden
  - γ(BC) erfordert Sortierung nach BCA

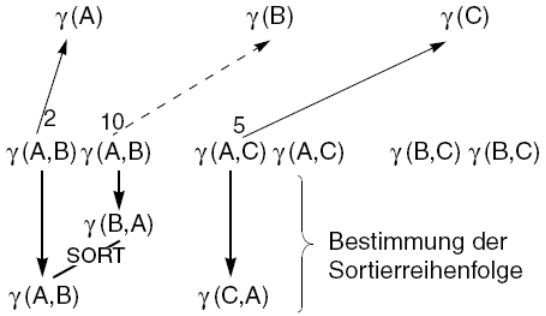
Auswahl der direkten Vorgängerknoten mit Sortierreihenfolge

- Sortierung des Datenbestands mit minimalen Kosten
  Konflikt:
  - kleinster Vorgänger („smallest parent“)
  - günstigste Sortierreihenfolge („share sort“)
- Pro Ebene im Aggregationsgitter mit n Attributen
  - ein Operator mit Kosten ohne Sortierung
  - n-1 Kopien des Operators mit Kosten für eine Sortierung
  - Finde kostengünstigste Paarbildung („minimum cost matching“) auf dem bipartiten Graph

CUBE-Implementierung mit Pipelining



a) Erweitertes Aggregationsgitter



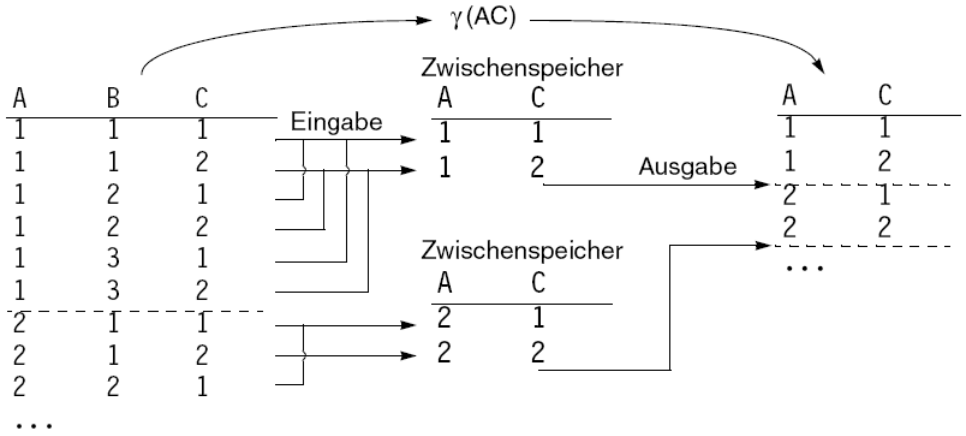
b) Ergebnis nach Paarbildung

Erweiterung

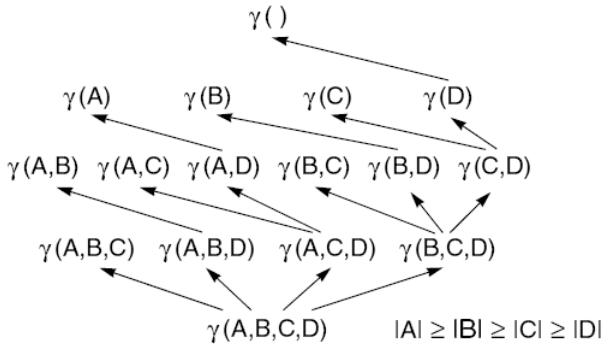
- Aufgabe des vollständigen Pipelining
- Zwischenspeicherung bei Einhaltung der partiellen Sortierreihenfolge

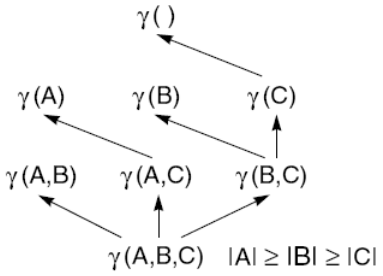
Berechnung über partielle Sortierreihenfolge

- Auswahl der Vorgänger mit partieller Sortierreihenfolge



- Ableitungsbaum bei Erhaltung der partiellen Sortierreihenfolge

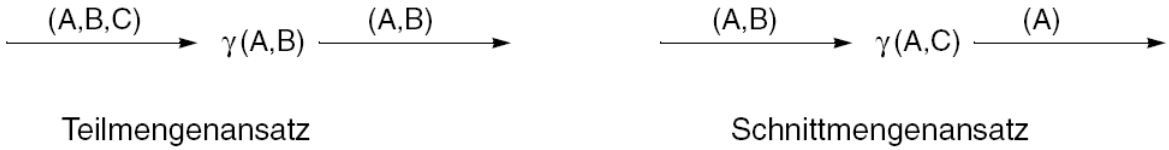




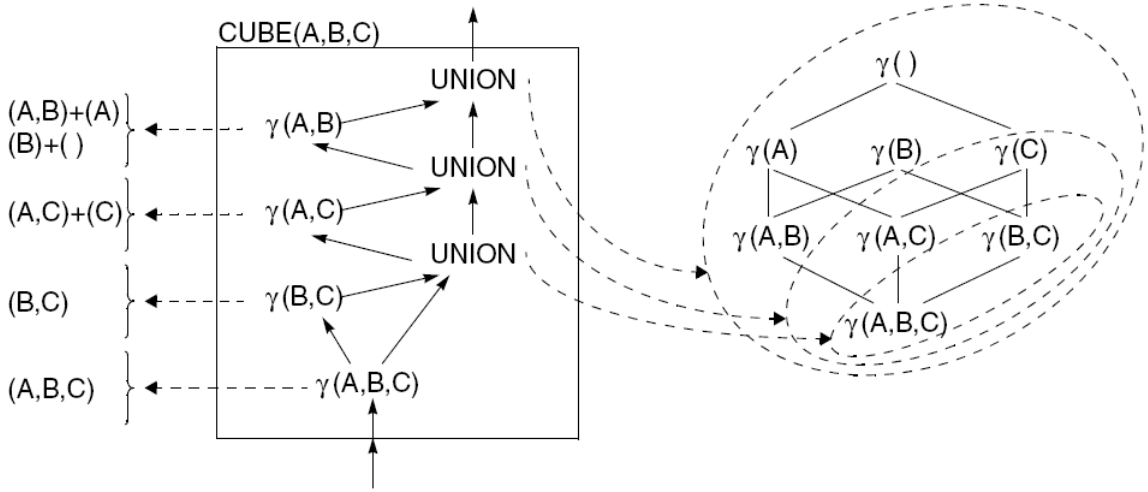
```
Algorithmus: PartitionedCube
Eingabe:R          // Relation mit zu gruppierenden Tupeln
        G1,...,Gn  // Gruppierungsattribute aus der GROUP BY-Klausel
        AGG(), A   // Aggregationsfunktion und zu aggregierendes Attribut A
Begin
  If (Size(R) < MaxMemorySize)
    // Falls R in dem Hauptspeicher passt, wird eine CUBE() basierend
    // auf der direkten Ableitbarkeit berechnet.
    C := ComputeMainMemoryCube(R, G1,...,Gn, A, AGG());
  Else
    // Wahl eines Gruppierungsattributes, nach welchem der Datenbestand
    // horizontal in m Fragemente partitioniert wird.
    k := PickSplitPosition(G1,...,Gn);
    (R1,...,Rm) := PartitionTableByAttr(R, k); // wobei gilt: m ≤ |Gk|
    // Individuelle Berechnung von m Teilwürfeln
    For i = 1 To m
      Ci := PartitionedCube(Ri, G1,...,Gn, AGG(), A);
      C:= C + Ci;
    End For
    // Berechnung der teilwürfelübergreifenden Teilsummen, indem die Menge
    // der Gruppierungsattribute um das Partitionsattribute verringert wird.
    C' := PartitionedCube(R, G1,..., Gk-1, Gk+1,...,Gn, AGG(), A);
    C := C + C';
  End If
  Return(C);
End
```

Schnittmengenansatz

- Prinzipieller Unterschied zum Teilmengenansatz
  - direkte Ableitbarkeit im Teilmengenansatz („subset stacking“)
  - Schnittmengenansatz („intersection stacking“)

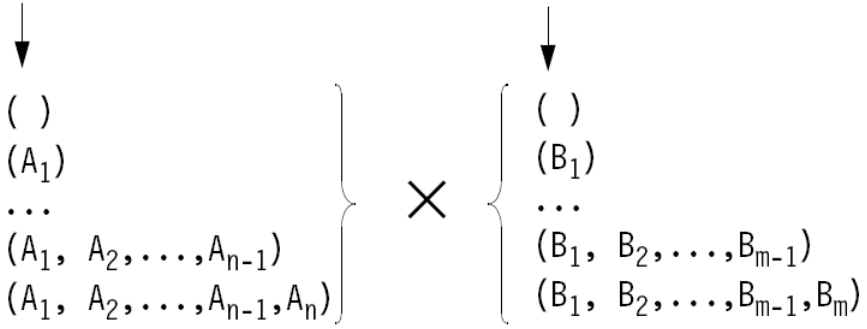


- Realisierung des CUBE()-Operators mit Schnittmengenansatz



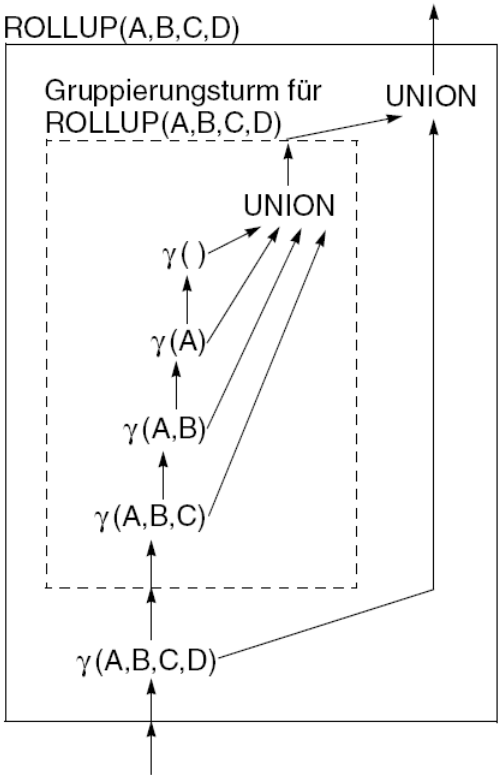
ROLLUP-Operator: Kombination von Dimensionshierarchien

GROUP BY ROLLUP$(A_1,A_2,\ldots,A_n)$, **ROLLUP**$(B_1,B_2,\ldots,B_m)$

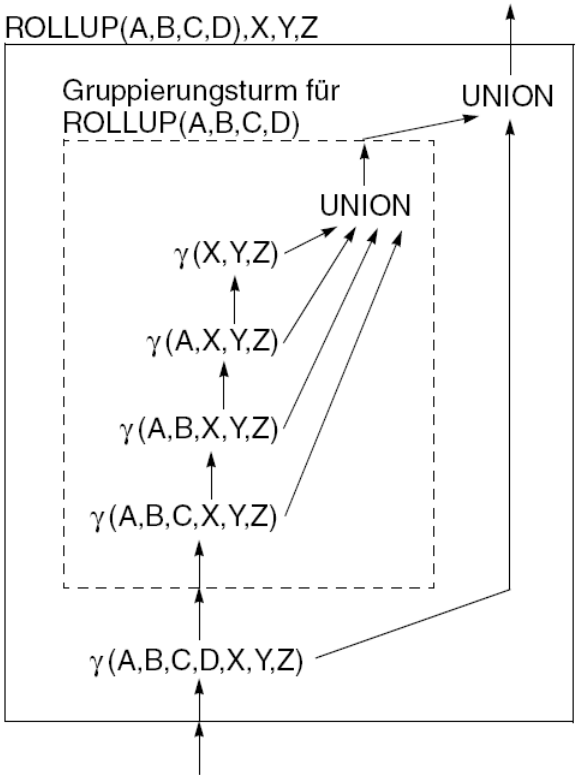


Beispiel

```
SELECT ...
  FROM TPCD.LINEITEM,                                  -- Faktentabelle
       TPCD.ORDERS, TPCD.CUSTOMER, TPCD.NATION N1      -- Auftragsdimension
       TPCD.SUPPLIER, TPCD.NATION N2                   -- Lieferantendimension
 WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY
   AND C_NATIONKEY = N1.N_NATIONKEY
   AND L_SUPPKEY = S_SUPPKEY AND S_NATIONKEY = N2.N_NATIONKEY
 GROUP BY ROLLUP(N1.N_REGIONKEY, N1.N_NATIONKEY, C_CUSTKEY, O_ORDERKEY),
          ROLLUP(N2.N_REGIONKEY, N2.N_NATIONKEY, S_SUPPKEY);
```



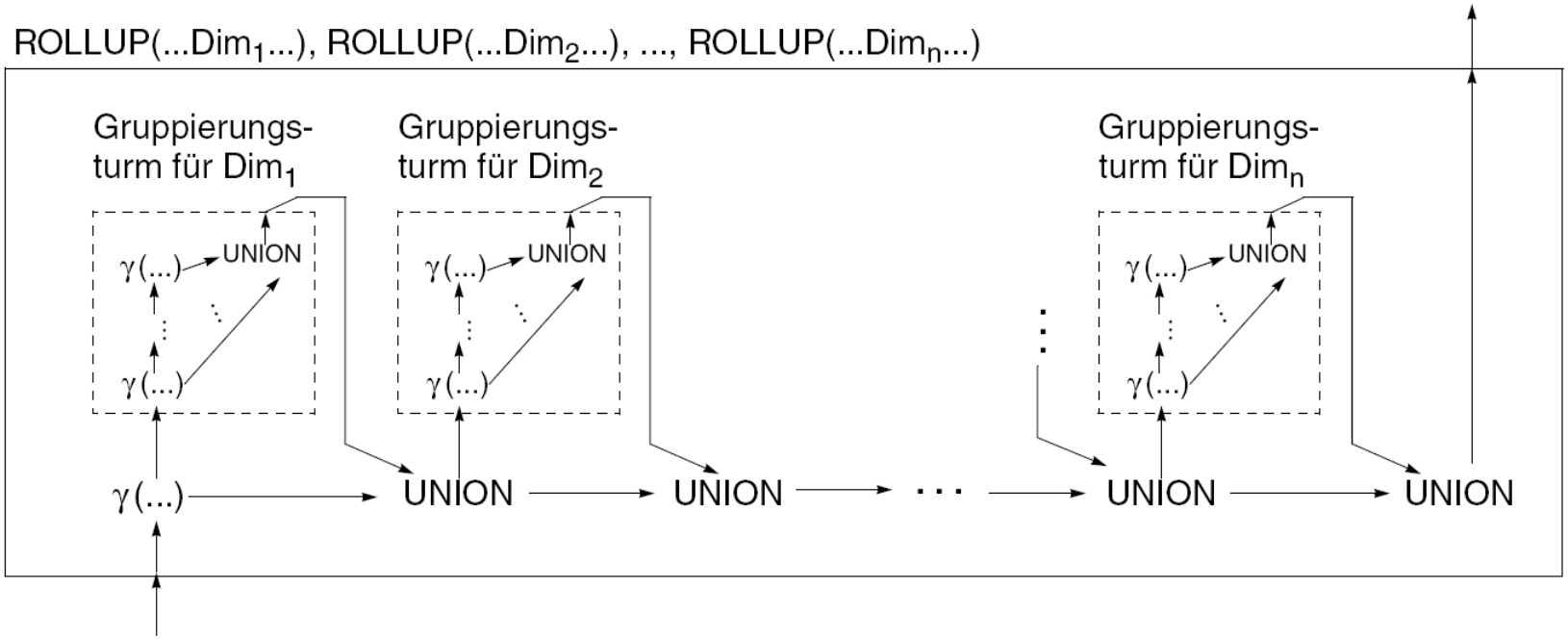
a) Einzelner Gruppierungsturm



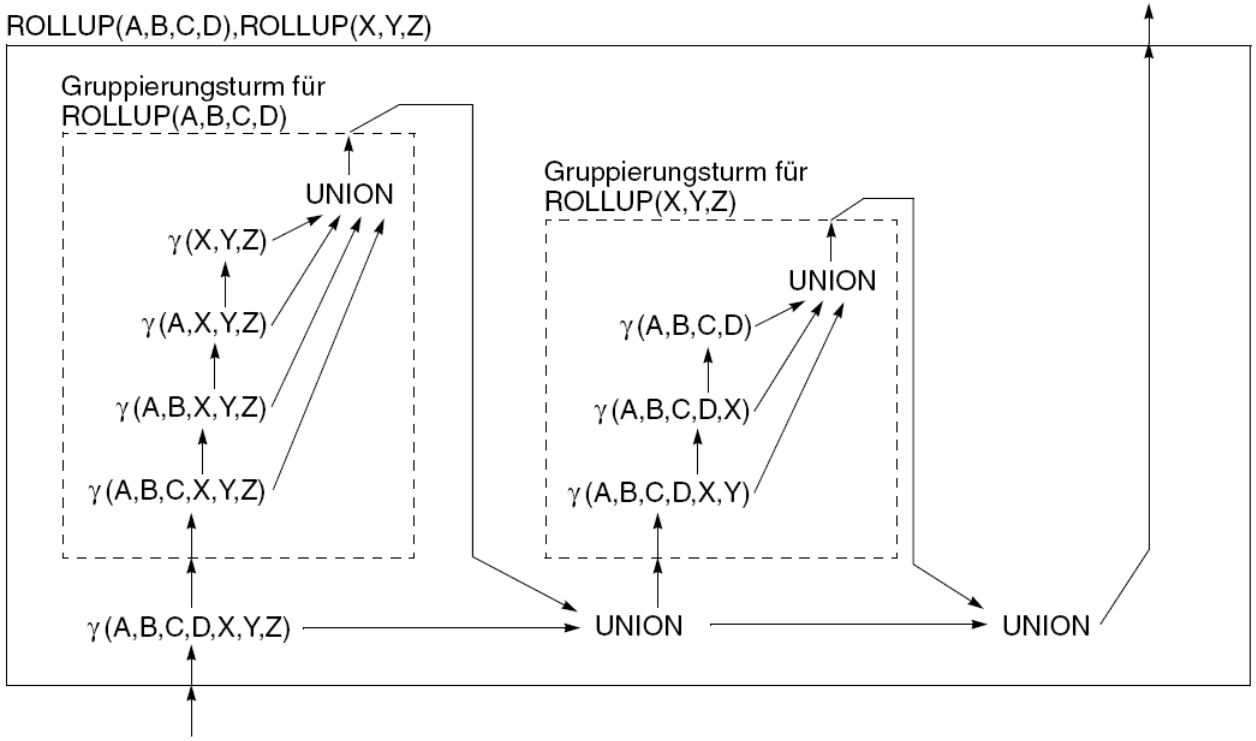
b) Gruppierungsturm mit weiteren Gruppierungsattributen

Kette von Gruppierungstürmen für hierarchische Datenwürfel

- sukzessive Auswertung von ROLLUP()-Konstrukten
- „Durchreichen“ der Partial-Ergebnisse an das Gesamtergebnis



- Beispiel



- Erweiterung
  - Partitionierung nach einem Gruppierungsattribut
  - Berechnung der Teilwürfel plus der Gruppierung über die Summen der Teilwürfel

Unäre Operatoren

- Selektion und Projektion
- Gruppierung
- Aggregation
- Sortieren

Verbundoperatoren (Binäre Operatoren)

- Verbundoperatoren
  - Nested-Loop Verbund
  - Sort-Merge Verbund
  - Hash-Verbund
  - Paralleler Verbund
- Data-Skew
- Verbund in verteilten Systemen

Literatur

- Härder, T. & Rahm, E. Datenbanksysteme: Konzepte und Techniken der Implementierung. Springer-Verlag, 1999
- Saake, G.; Heuer, A. & Sattler, K.-U. Datenbanken: Implementierungstechniken. MITP-Verlag, 2005
- Hellerstein, J. M.; Stonebraker, M. & Hamilton, J. R. Architecture of a Database System. Foundations and Trends in Databases, 2007, 1, 141-259
- Graefe, G. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 1993, 25, 73-170