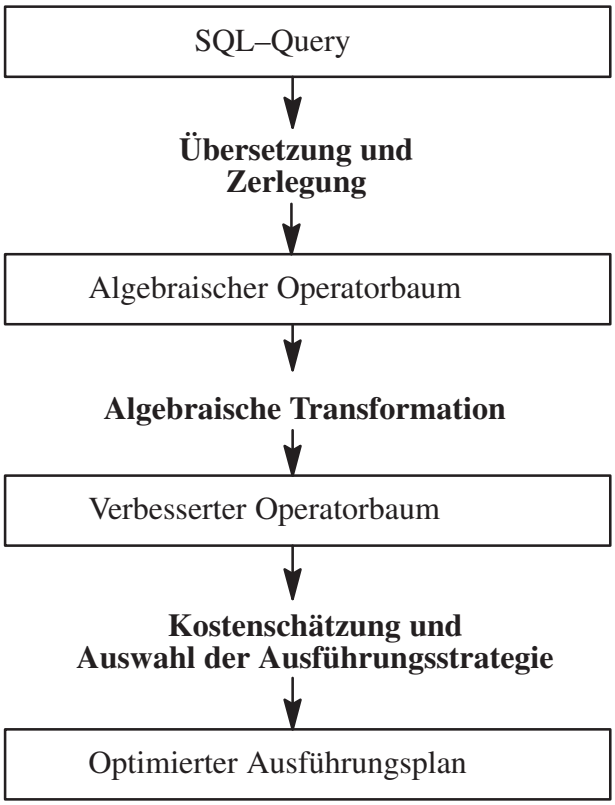
# Kapitel 9: Query–Optimierung und Query–Ausführung

## 9.1 Arbeitsweise

**Architektur des Query–Optimierers:**




Der vom Optimierer erzeugte Ausführungsplan ist ein Operatorbaum, dessen Knoten den Ausführungsprimitiven des DBS–Laufzeitsystems entsprechen.
Man kann in den meisten DBS die generierten Ausführungspläne mit dem EXPLAIN–Kommando anschauen und analysieren.

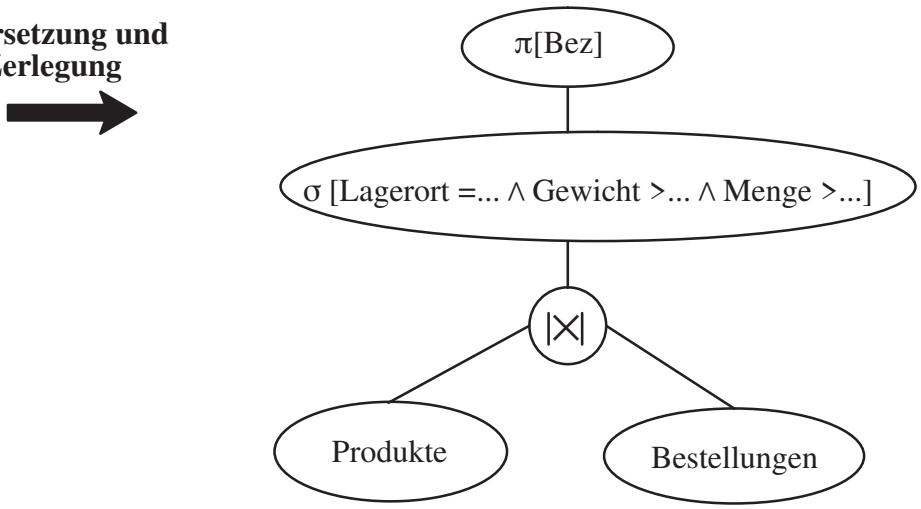
Die Optimierung findet in fast allen Produkten zur Compile–Zeit eines ESQL–Programms statt; wiederholt ausgeführte parametrisierte Queries werden nur einmal optimiert.

Wenn die Güte eines Ausführungsplans einer parametrisierten Query stark von den aktuellen Parametern oder den zur Laufzeit zur Verfügung stehenden Ressourcen abhängt, kann eine dynamische Optimierung u.U. vorteilhafter sein.

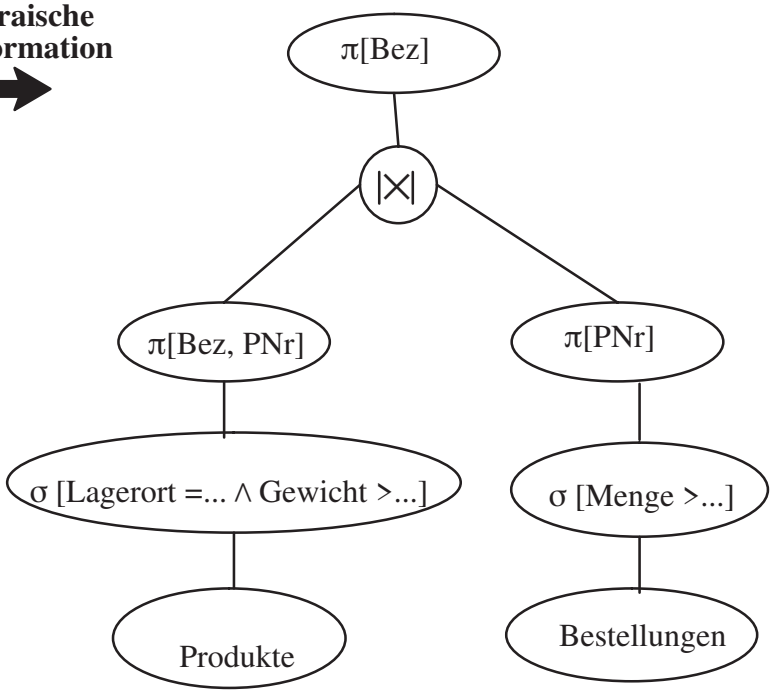
**Beispiel:**

```
SELECT Bez
FROM Produkte, Bestellungen
WHERE Lagerort = :lort AND Gewicht > :gew AND Menge > :menge
AND Produkte.PNr=Bestellungen.PNR
```

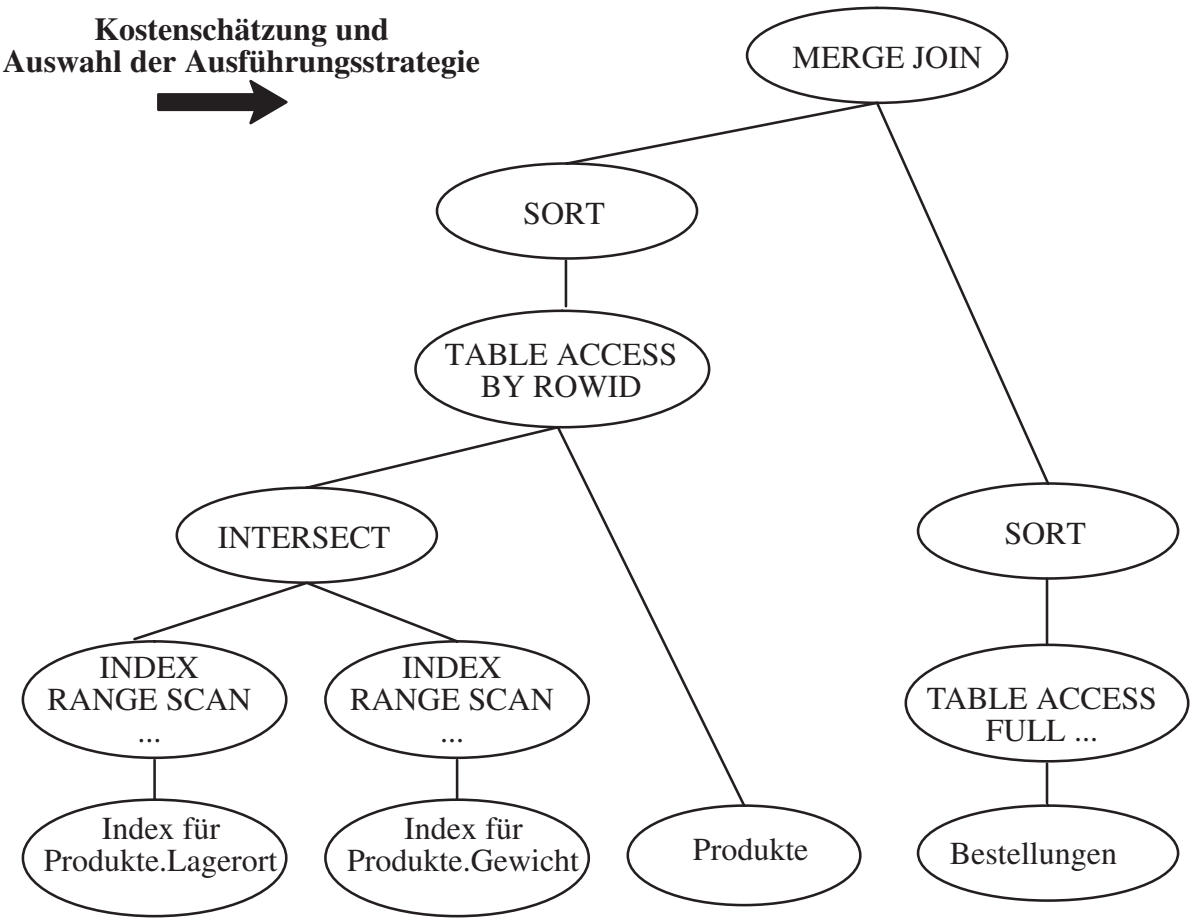
**Übersetzung und Zerlegung**




**Algebraische Transformation**



**Kostenschätzung und Auswahl der Ausführungsstrategie**



## 9.2 Wichtige Ausführungsprimitive (Operatoren der Ausführungsalgebra)

Das physische Datenbankschema ist zum Zeitpunkt der Query−Optimierung fest, so daß nicht immer alle Ausführungsprimitive anwendbar sind.

**Relationen−Scan** (TABLE ACCESS FULL):
Sequentielles Lesen aller Tupel einer Relation,
wobei auf jedem Tupel ein "Single−Scan"−Filterprädikat überprüft und
eine Attributprojektion (ohne Duplikateliminierung) vorgenommen werden kann.

**TID−Zugriff** (TABLE ACCESS BY ROWID):
Direkter Zugriff auf alle Tupel einer Liste von TIDs,
wobei auf jedem Tupel ein "Single−Scan"−Filterprädikat überprüft und
eine Attributprojektion (ohne Duplikateliminierung) vorgenommen werden kann.
Die TID−Liste sollte nach Seitennummern sortiert sein; Seiten mit kleinem "Abstand"
auf derselben Platte (z.B. im selben Zylinder) sollten mit einem einzigen Plattenzugriff
gelesen werden (multi−block I/O, list prefetch).

**Index−Scan** (INDEX RANGE SCAN oder INDEX UNIQUE SCAN):
Zugriff auf alle TIDs von Tupeln, die ein Index−Suchprädikat der Form $wert1 \leq KEY \leq wert2$
erfüllen, wobei KEY der Suchschlüssel des Index ist (ggf. eine Attributkombination).
Realisiert ist dies durch eine Indexsuche nach wert1 mit einem nachfolgenden "Blatt−Scan"
bis zum Erreichen des ersten Schlüssels, der größer als wert2 ist.

**Sortieren** (SORT):
Sortieren einer Tupelmenge nach einem Attribut bzw. einer Attributkombination.
Dabei werden Sortieralgorithmen für Externspeicher verwendet (z.B. Mehrwegemischen).

**Durchschnitt, Vereinigung, Differenz** (INTERSECTION, UNION, MINUS):
Anwenden von Mengenoperationen auf Tupelmengen oder TID−Listen.
Dies wird in der Regel erst nach vorhergehendem Sortieren angewandt.

**Selektion bzw. Filterung** (FILTER):
Überprüfen einer Booleschen Bedingung auf den Tupeln einer Tupelliste

**Projektion für Mengen** (PROJ−SET):
Projektion von Attributen einer nach diesen Attributen sortierten Tupelliste mit Duplikat-
eliminierung

**Projektion für Multimengen** (PROJ−MULTI):
Projektion von Attributen einer Tupelliste

**Nested−Loop−Join** (NESTED LOOPS):
Berechnung eines Joins zwischen zwei Tupelmengen (mit Angabe einer expliziten Join−bedingung) mittels einer Doppelschleife.

**Merge−Join** (MERGE JOIN):
Berechnung eines Equi−Joins zwischen zwei nach dem Join−Attribut sortierten Tupellisten mittels Mischen der Listen.
Dieses Join−Berechnungs−Verfahren heißt häufig auch Sort−Merge−Join, weil u.U. vor dem Mischen noch Sortierschritte stattfinden müssen.

**Hash−Join** (HASH JOIN):
Berechnung eines Equi−Joins zwischen zwei Tupelmengen durch Abbilden beider Mengen auf eine Hash−Tabelle (mit einer für beide Tupelmengen identischen Hash−Funktion über dem Join−Attribut), so daß Tupel mit gleichem Attributwert auf denselben Eintrag (bzw. Bucket) der Hash−Tabelle abgebildet werden.

**Gruppierung mit Aggregation** (GROUP):
Berechnung eines Aggregationsfunktionswerts pro Gruppe auf einer nach dem Gruppierungsattribut sortierten Tupelliste durch sequentielles Lesen aller Tupel

**Hash−Gruppierung mit Aggregation** (HASH GROUP):
Berechnung eines Aggegationsfunktionswerts pro Gruppe durch Abbilden aller Tupel auf eine Hashtabelle mit einer Hashfunktion auf dem Gruppierungsattribut und − falls erforderlich − anschließendem SORT und GROUP für jede Hashklasse

## Externes Sortieren

Schnelle Algorithmen zum Sortieren von Relationen oder Anfragezwischenergebnissen arbeiten in mehreren Phasen. Jeder Phase steht ein Hauptspeicherbereich der Größe M Blöcke und ein temporärer Arbeitsbereich auf Magnetplatte zur Verfügung.

1) *Bilden von sortierten Teilsequenzen, sogenannten Runs:*
   Es werden jeweils M Blöcke der unsortierten Eingabedaten in den Hauptspeicher gelesen. Diese werden mit einem leistungsfähigen Algorithmus für internes Sortieren (z.B. Quicksort− oder Heapsort−Varianten) sortiert und als Run in einem temporären Arbeitsbereich auf Magnetplatte geschrieben.

2) *Mischen der Runs:*
   Es werden jeweils M Runs blockweise in den Hauptspeicher gelesen und zu einem einzigen Run gemischt. Dieser Resultat−Run wird blockweise in den temporären Arbeitsbereich auf Magnetplatte zurückgeschrieben.

3) *Iterieren der Mischphasen:*
   Solange in einer Mischphase nicht alle Runs komplett zu einem einzigen Resultat−Run gemischt werden konnten, wird die Mischphase mit den bisher erzeugten − immer länger werdenden − Resultat−Runs iteriert.

### I/O−Kosten des externen Sortieralgorithmus

Annahmen:
S(R) − Anzahl der Blöcke der Eingaberelation R
M − Anzahl der Blöcke des zur Verfügung stehenden Hauptspeichers

1) Bilden von Runs:
   Es werden S(R) / M (bzw. ceil (S(R) / M) Runs der Größe M (bzw. $\leq$ M) gebildet.
   Dazu werden alle Blöcke von R je einmal gelesen und geschrieben. Es sind also
   2 * S(R)/M Seitenzugriffe
   notwendig.

2) Mischen von jeweils M Runs der Größe M:
   Es werden S(R) / $M^2$ Runs der Größe $M^2$ gebildet. Dies erfordert wiederum
   2 * S(R)/M Seitenzugriffe.

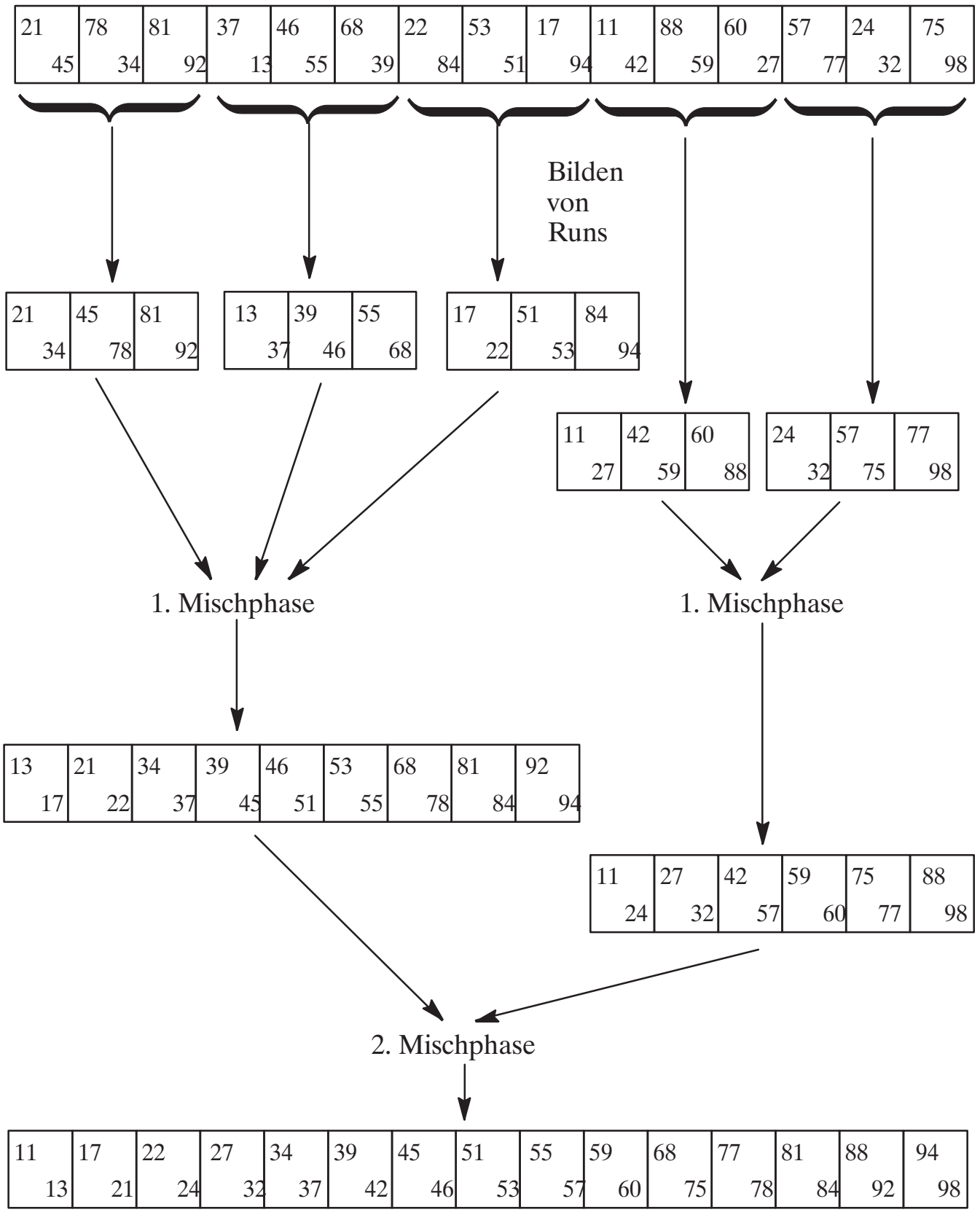
3) Iterieren der Mischphasen, bis Anzahl der Runs gleich 1 ist, also
   S(R) / $M^i \leq 1$ ist, also $i \geq \log_M S(R)$ gilt.
   Bis dahin sind i−1 Mischphasen notwendig (inkl. der letzten).

Damit betragen die Gesamtkosten:
( ceil ( $\log_M$ S(R) ) −1 ) * 2 * S(R) + 2 * S(R) Seitenzugriffe.

Man kommt garantiert mit einer einzigen Mischphase aus (hat also dann lineare I/O−Komplexität), wenn gilt:
$\log_M S(R) \leq 2$, was äquivalent ist zu $M \geq \sqrt{S(R)}$

Bei Verwendung mehrerer Platten sind weitergehendere Optimierungen möglich.

**Beispiel (M=3 Blöcke mit je 2 Tupeln):**



## Verfahren zur Berechnung von Equi−Joins

**Nested−Loop−Join:**

Festlegung der äußeren (”outer”) und der inneren (”inner”) Relation
for each Tupel r ∈ outer:
  /* Finde alle Tupel s ∈ inner mit s.Joinattribut = r.Joinattribut */
  for each Tupel s ∈ inner:
    if r.Joinattribut = s.Joinattribut
    then Füge das kombinierte Tupel (r,s) in die Resultatmenge ein

**Nested−Loop−Join mit Indexzugriff auf die innere Relation:**

Festlegung der äußeren (”outer”) und der inneren (”inner”) Relation
for each Tupel r ∈ outer:
  Finde mittels Indexzugriff die TIDs aller Tupel s mit s.Joinattribut = r.Joinattribut
  for each TID:
    Hole das Tupel s und füge das kombinierte Tupel (r,s) in die Resultatmenge ein

**Verbesserte Variante mit Indexzugriff und sequentiellem Zugriff auf die innere Relation:**

Festlegung der äußeren (”outer”) und der inneren (”inner”) Relation
for each Tupel r ∈ outer:
  Finde mittels Indexzugriff die TIDs aller Tupel s mit s.Joinattribut = r.Joinattribut
  for each TIDt:
    Schreibe (r,t) in eine temporäre Relation (”temp”)
Sortiere temp nach den TIDs der entsprechenden inneren Tupel
for each Tupel (r,t) ∈ temp:
  Hole das Tupel s mit s.TID=t und füge das kombinierte Tupel (r,s) in die Resultatmenge ein.

**Verbesserte Variante für den Fall ohne Indexunterstützung: Nested−Block−Join:**

Festlegung der äußeren (”outer”) und der inneren (”inner”) Relation
for each Seite (bzw. Menge benachbarter Seiten) von outer:
  for each Seite (bzw. Menge benachbarter Seiten) von inner:
    for each Tupel r in der outer−Seite
      for each Tupel s in der inner−Seite:
        if r.Joinattribut = s.Joinattribut
        then Füge das kombinierte Tupel (r,s) in die Resultatmenge ein

**Merge−Join:**

```
Festlegung der äußeren ("outer") und der inneren ("inner") Relation
r := erstes Tupel von outer (in bzgl. des Joinattributs sortierter Reihenfolge)
s := erstes Tupel von inner (in bzgl. des Joinattributs sortierter Reihenfolge)
while nicht alle Tupel von outer bearbeitet or nicht alle Tupel von s bearbeitet do
   if r.Joinattribut = s.Joinattribut then
      s1 := s
      while r.Joinattribut = s.Joinattribut do
         Füge das kombinierte Tupel (r,s) in die Resultatmenge ein
         s := nächstes Tupel von inner
      s := s1 /* Gehe zurück zum ersten inner−Tupel mit diesem Joinattributwert */
      r := nächstes Tupel von outer
   else  if r.Joinattribut < s.Joinattribut
         then r := nächstes Tupel von outer
         else s := nächstes Tupel von inner
```

Bemerkung: Die Schleife über die beiden Relationen kann als Relationen−Scan realisiert sein, sofern die Relationen sortiert sind, oder als Index−Scan über dem Joinattribut.

**Hash−Join:**

```
Festlegung der äußeren ("outer") und der inneren ("inner") Relation
Allokation der Hashtabelle (organisiert in Buckets, so daß ein Bucket in den
Hauptspeicher paßt)
for each Tupel r ∈ outer:
   Speichere r in Bucket h(r.Joinattribut) (wobei h die Hashfunktion ist)
for each Tupel s ∈ inner:
   Speichere s in Bucket h(s.Joinattribut)
for each Bucket b:
   for each Tupel r ∈ outer∩ b:
      for each Tupel s ∈ inner∩ b:
            if r.Joinattribut = s.Joinattribut
            then Füge das kombinierte Tupel (r,s) in die Resultatmenge ein
```

## Beispiele zur Join−Berechnung:

Sei R die äußere Relation mit 4 Tupeln r1, r2, r3, r4, die wie folgt auf zwei Blöcke R1 und R2 verteilt sind und die jeweils angegebenen Werte bzgl. des Joinattributs J haben:
R1: r1, r2 mit r1.J=a, r2.J=c
R2: r3, r4 mit r3.J=b, r4.J=a

Sei S die innere Relation mit 4 Tupeln s1, s2, s3, s4, die wie folgt auf zwei Blöcke S1 und S2 verteilt sind und die jeweils angegebenen Werte bzgl. des Joinattributs J haben:
S1: s1, s2 mit s1.J=a, s2.J=b
S2: s3, s4 mit s3.J=c, s4.J=b

**Nested−Loop−Join:**

Reihenfolge der Block− und Tupelzugriffe:
R1 r1 S1 s1 s2 S2 s3 s4 r2 S1 s1 s2 S2 s3 s4 R2 r3 S1 s1 s2 S2 s3 s4 r4 S1 s1 s2 S2 s3 s4

**Nested−Loop−Join mit Indexzugriff auf die innere Relation:**

Reihenfolge der Block−, Tupel− und Indexzugriffe (auf Werte a, b, c, d):
R1 r1 a S1 s1 r2 c S2 s3 R2 r3 b S1 s2 S2 s4 r4 a S1 s1

**Verbesserte Variante mit Indexzugriff und sequentiellem Zugriff auf die innere Relation:**

Reihenfolge der Block−, Tupel−, Index− und Temp−Zugriffe (T):
R1 r1 a T r2 c T R2 r3 b T r4 a T
T hat jetzt den Inhalt:
(r1, &s1), (r2, &s3), (r3, &s2), (r3, &s4), (r4, &s1)
Nach Sortierung nach den S−TIDs hat T den Inhalt:
(r1, &s1), (r4, &s1), (r3, &s2), (r2, &s3), (r3, &s4)
Anschließende Zugriffe auf S:
S1 s1 s1 s2 S2 s3 s4

**Verbesserte Variante für den Fall ohne Indexunterstützung: Nested−Block−Join:**

Reihenfolge der Block− und Tupelzugriffe:
R1 S1 r1 s1 s2 r2 s1 s2 S2 r1 s3 s4 r2 s3 s4 R2 S1 r3 s1 s2 r4 s1 s2 S2 r3 s3 s4 r4 s3 s4

**Merge−Join:**

R sortiert nach J: r1, r4, r3, r2
S sortiert nach J: s1, s2, s4, s3
Mischen: r1 s1 r4 s1 r3 s2 s4 r2 s3

**Hash−Join:**

Sei h die Hashfunktion mit h(a)=0, h(b)=h(c)=1
Inhalt von Hash−Bucket 0: r1, r4, s1
Inhalt von Hash−Bucket 1: r3, r2, s2, s4, s3

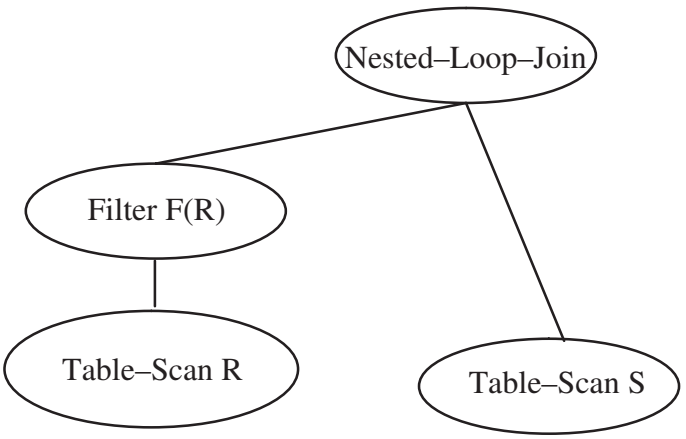
**Bedarfsgesteuerte, verschachtelte Auswertung von Operatorbäumen (Pipelining)**

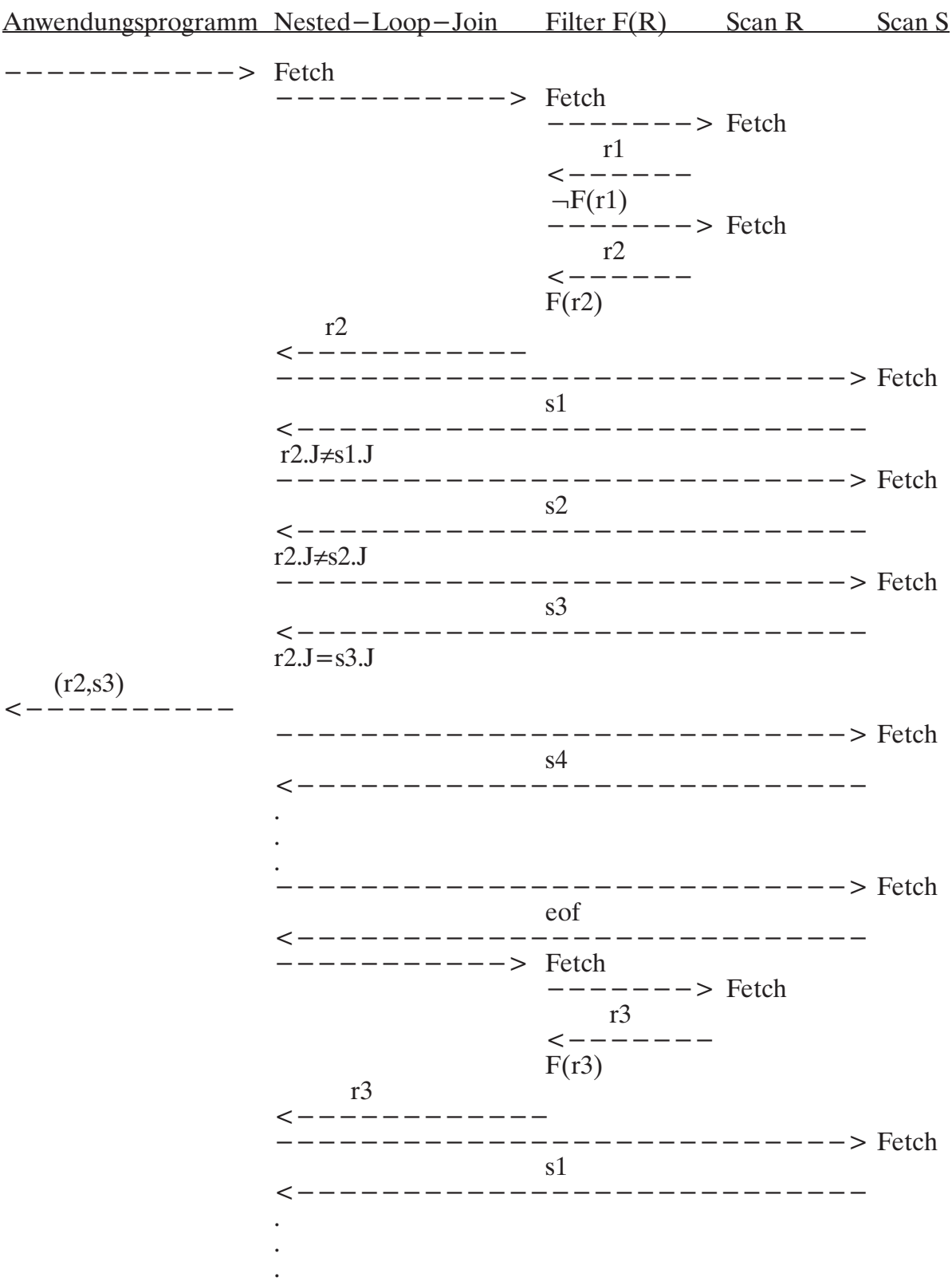
Zwischenresultate eines Teilbaums werden immer nur so weit berechnet, wie es nötig ist, damit der Vateroperator seine Berechnung fortsetzen kann.

Vorteile: – Zwischenergebnisse müssen nicht komplett materialisiert (und damit zwischengespeichert) werden, sondern werden inkrementell nach Bedarf berechnet (”demand–driven evaluation”, “lazy evaluation”)

– Anfragen, deren Resultatsausgabe abgebrochen wird, verursachen nicht die vollen Auswertungskosten

**Beispiel:**




Anwendungsprogramm
Nested−Loop−Join
Filter F(R)
Scan R
Scan S
Fetch
Fetch
Fetch
Fetch
r1
¬F(r1)
Fetch
r2
F(r2)
r2
Fetch
s1
r2.J≠s1.J
Fetch
s2
r2.J≠s2.J
Fetch
s3
r2.J=s3.J
(r2,s3)
Fetch
s4
Fetch
eof
Fetch
Fetch
r3
F(r3)
r3
Fetch
s1

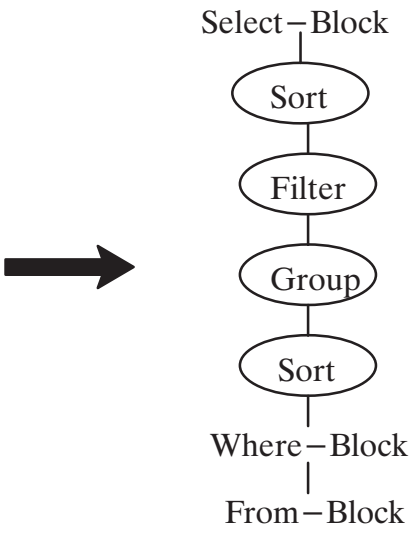
## 9.3 Übersetzung von SQL−Anfragen

Zielsprache der Übersetzung sind Ausdrücke bzw. Operatorbäume einer an der Relationenalgebra orientierten “Ausführungsalgebra”. Diese stellt Operatoren auf Mengen, Multimengen und (sortierten) Listen zur Verfügung. Dazu gehören insbesondere auch Operatoren, die in der eigentlichen Relationenalgebra überhaupt nicht auftauchen, z.B. SORT und GROUP. Andere Operatoren müssen in mehreren Varianten − entsprechend ihrer Anwendung auf Mengen, Multimengen oder Listen − spezialisiert werden; beispielsweise gibt es eine Projektion für Mengen (mit Duplikateliminierung) und eine für Multimengen (ohne Duplikateliminierung).

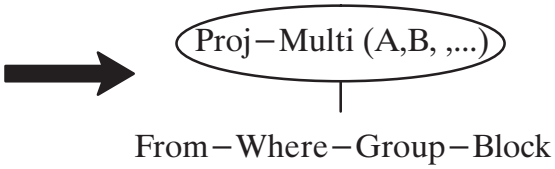
Einige Operatoren, insbesondere der Join−Operator, werden entsprechend verschiedener Implementierungsvarianten spezialisiert. Generelle Operatoren dieser Art werden in der späteren Optimierung durch spezialisierte Operatoren oder auch durch ganze Operatorbäume ersetzt.

**“Übersetzungsregeln”:**

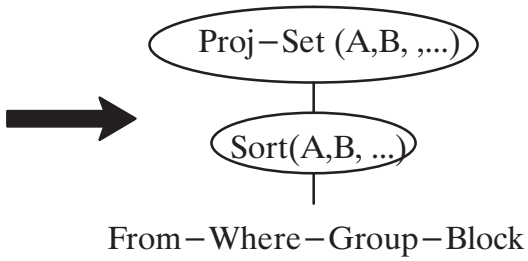
1) Select ... From ... Where ...
   GROUP BY ... HAVING ... ORDER BY ...



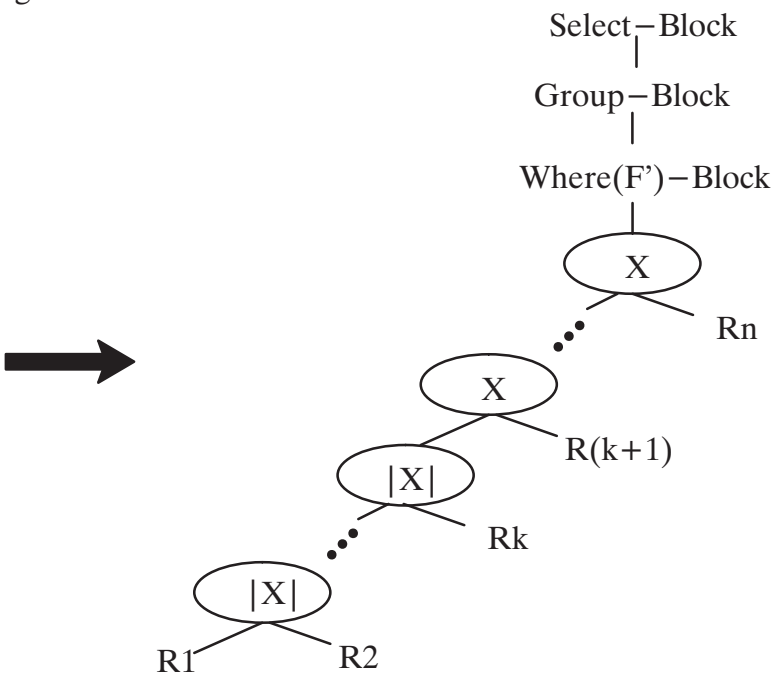

2) SELECT A, B, ...
   From ... Where ... Group By ...



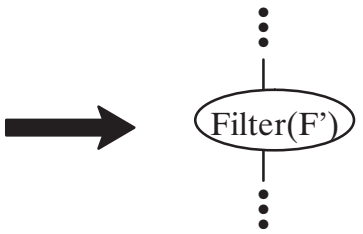

3) SELECT DISTINCT A, B, ...
   From ... Where ... Group By ...



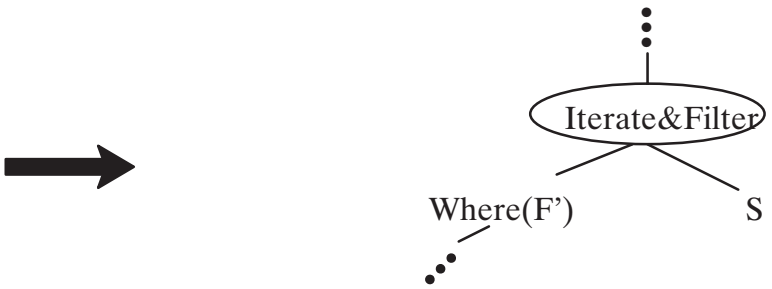
4) Select ... FROM R1, ..., Rk, Rk+1, ... , Rn WHERE F Group By ...
mit F in konjunktiver Normalform und konjunktiven Termen Ri.A=Rj.B ($1 \le i \le k, 1 \le j \le k$)
und einer “Restbedingung” F’ ohne diese Terme




5) ... WHERE F’ ohne Subqueries

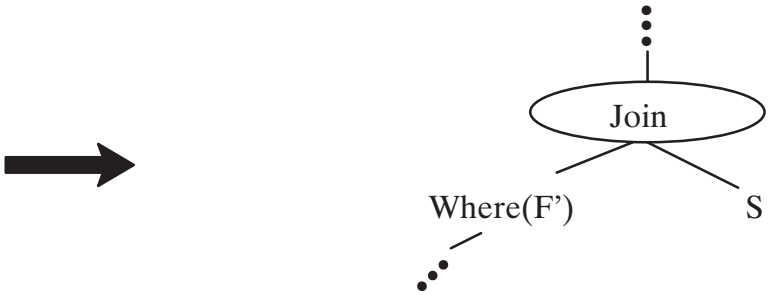


6) ... WHERE F' AND t.A Θ Subquery S
mit in S verwendeten Korrelationsvariablen x, y, ... äußerer Blöcke

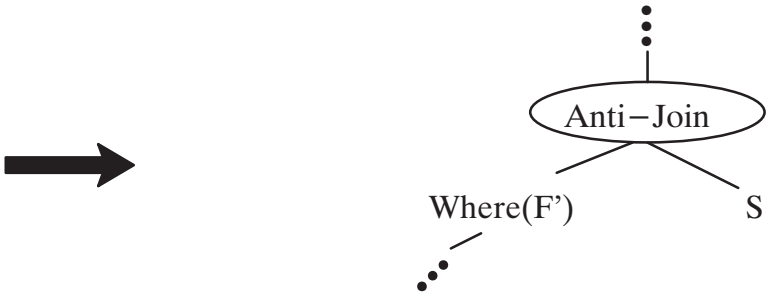



wobei Iterate&Filter für jeden in Where(F') gefundenen Wert der Tupelvariablen t.A, x, y, ... die Tupelvariablen in S durch die Werte substituiert und S wiederholt auswertet

Spezialfall: ... WHERE F' AND t.A IN Subquery S
ohne Korrelationsvariable äußerer Blöcke in S

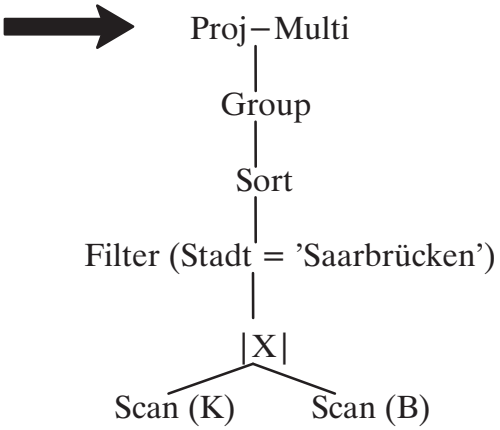



Spezialfall: ... WHERE F' AND t.A NOT IN Subquery S
ohne Korrelationsvariable äußerer Blöcke in S

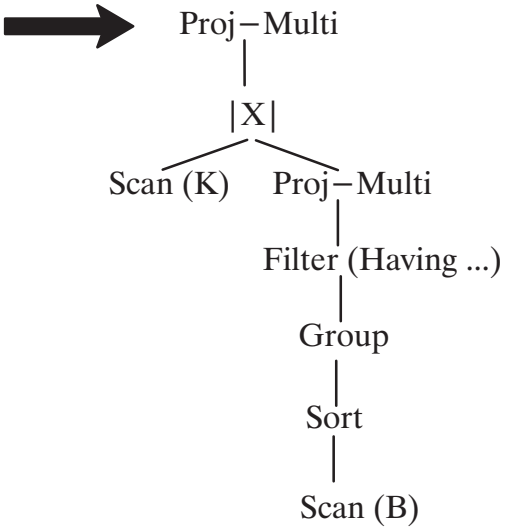


**Beispiele:** auf den Tabellen Produkte, Bestellungen, Kunden

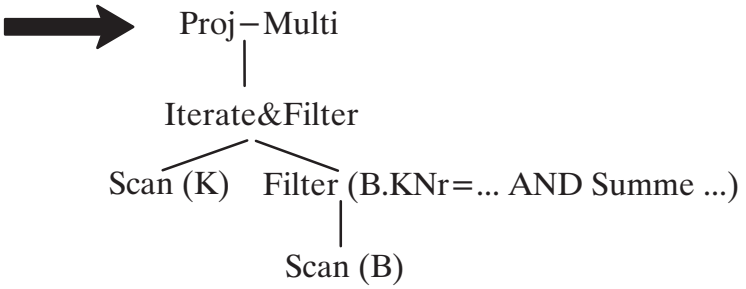
1)
```
SELECT AVG(B.Summe) FROM Kunden K, Bestellungen B
WHERE K.KNr = B.KNr AND K.Stadt = 'Saarbrücken'
GROUP BY PNr
```




2)
```
SELECT * FROM Kunden
WHERE KNr IN    (SELECT KNr FROM Bestellungen
                 GROUP BY KNr HAVING COUNT(*) > 10)
```




3)
```
SELECT * FROM Kunden K
WHERE EXISTS    (SELECT * FROM Bestellungen
                 WHERE B.KNr = K.KNr AND Summe > 1000)
```



## Beispiele für von Oracle erzeugte Ausführungspläne:

auf den Tabellen Produkte, Bestellungen, Kunden
mit Indizes auf allen Primär− und Fremdschlüsseln

1) SELECT AVG(B.Summe) FROM Kunden K, Bestellungen B
   WHERE K.KNr=B.KNR
   AND K.Stadt='Saarbrücken'
   GROUP BY PNr

   wird übersetzt in:

```
SELECT STATEMENT
   SORT                                GROUP BY
      NESTED LOOP
         TABLE ACCESS                  FULL          Bestellungen
         FILTER (Stadt='Saarbrücken')
            TABLE ACCESS               BY ROWID      Kunden
               INDEX RANGE SCAN        UNIQUE        Kunden−KNr−Index
```

2) SELECT * FROM Kunden
   WHERE KundenNr IN ( SELECT KNr FROM Bestellungen
                       GROUP BY KNr HAVING COUNT(*) > 10 )

   wird übersetzt in:

```
SELECT STATEMENT
   HASH JOIN
      FILTER (Having Count(*)>10)
         SORT                          GROUP BY
            TABLE ACCESS               FULL          Bestellungen
      TABLE ACCESS                     FULL          Kunden
```

3) SELECT * FROM Kunden K
   WHERE EXISTS ( SELECT * FROM Bestellungen B
                  WHERE B.KNr=K.KNr AND Summe > 1000 )

   wird übersetzt in:

```
SELECT STATEMENT
   FILTER (d.h. Iterate&Filter)
      TABLE ACCESS                     FULL          Kunden
      FILTER (Summe>1000)
         TABLE ACCESS                  BY ROWID      Bestellungen
            INDEX RANGE SCAN           NON−UNIQUE    Bestellungen−KNr−Index
```

# 9.4 Algebraische Transformation

Ziele:
- Vereinfachung der Query
- Elimination unnötiger Operationen
- Anwendung einfacher Heuristiken zur Effizienzverbesserung, insbesondere die frühe Ausführung von Selektionen zur Reduktion der Größe von Zwischenresultaten

Methode:
Anwendung relationenalgebraischer Äquivalenzregeln (siehe Kapitel 2.2)

Beispiele:

1) σ[Lagerort='Homburg' ∧ Menge>100] (Produkte |x| Bestellungen)
= σ[Lagerort='Homburg'] (σ[Menge>100] (Produkte |x| Bestellungen)) nach Regel 5
= σ[Lagerort='Homburg'] (σ[Menge>100] (Bestellungen |x| Produkte)) nach Regel 2
= σ[Lagerort='Homburg'] (σ[Menge>100] (Bestellungen) |x| Produkte) nach Regel 8
= σ[Lagerort='Homburg'] (Produkte |x| σ[Menge>100] (Bestellungen)) nach Regel 2
= σ[Lagerort='Homburg'] (Produkte) |x| σ[Menge>100] (Bestellungen) nach Regel 8

2) Es seien die folgenden Views gegeben:
Preise (PNr, Preis) := π[PNr, Preis] (Produkte)
Lager (PNr, Lagerort, Vorrat) := π[PNr, Lagerort, Vorrat] (Produkte)
Query−Transformation:
σ[Lagerort='Homburg' ∧ Preis>100] (Preise |x| Lager)
= σ[Lagerort='Homburg' ∧ Preis>100]
(π[PNr, Preis] (Produkte) |x| π[PNr, Lagerort, Vorrat] (Produkte))
aufgrund der Viewdefinition
= σ[Lagerort='Homburg' ∧ Preis>100]
(π[PNr, Preis, Lagerort, Vorrat] (Produkte |x| Produkte))
nach Regel 9 und weil PNr der Primärschlüssel von Produkte ist
= σ[Lagerort='Homburg' ∧ Preis>100] (π[PNr, Preis, Lagerort, Vorrat] (Produkte))
nach Regel 12

3) Es sei folgende View gegeben:
BP := Bestellungen ‖ *| Produkte
Query−Transformation:
π[Status] (σ[BestNr=111](BP))
= π[Status] (π[B−Attr.] (σ[BestNr=111](BP)))
nach Regel 4
= π[Status] (σ[BestNr=111] (π[B−Attr.](BP)))
nach Regel 1
= π[Status] (σ[BestNr=111] (π[B−Attr.](Bestellungen ‖ *| Produkte)))
aufgrund der Viewdefinition
= π[Status] (σ[BestNr=111] (Bestellungen))
nach Regel 11

*”Semantische” Transformation unter Einbeziehung von Integritätsbedingungen:*

1) Integritätsbedingung für Produkte:
   Preis < 1000
   Query:
   σ[Preis > 2000] (Produkte)
   = σ[Preis > 2000] ( σ[Preis < 1000] (Produkte) )
   = σ[Preis > 2000 ∧ Preis < 1000] (Produkte)
   = ∅

2) Integritätsbedingung für Produkte:
   Vorrat < 10000 ∨ Lagerort=’Homburg’
   Query:
   σ[Vorrat > 20000] (Produkte)
   = σ[Vorrat > 20000 ∧ (Vorrat < 10000 ∨ Lagerort=’Homburg’)] (Produkte)
   = σ[Vorrat > 20000 ∧ Lagerort=’Homburg’] (Produkte)
   → Ausführung ggf. mit Index−Scan über Lagerort !

## Äquivalenzregeln der Relationenalgebra

Seien R, S, T Relationen, P, P1, P2 Prädikate, R1, R2, S1 Teilmengen der Attribute von R bzw. S. Es gelten die folgenden Äquivalenzregeln:

**Kommutativitätsregeln:**

1) $\pi[R1]\,\sigma[P]\,(R) = \sigma[P]\,\pi[R1]\,(R)$
   falls P nur R1−Attribute enthält

2) $R \,|x|[P]\, S = S \,|x|[P]\, R$

**Assoziativitätsregeln:**

3) $R \,|x|[P1]\, (\, S \,|x|[P2]\, T\,) = (\, R \,|x|[P1]\, S\,) \,|x|[P2]\, T$
   falls P1 nur R− und S−Attribute und P2 nur S− und T−Attribute enthält

**Idempotenzregeln:**

4) $\pi[R1]\,(\,\pi[R2]\,(R)\,) = \pi[R1]\,(R)$
   falls $R1 \subseteq R2$

5) $\sigma[P1]\,(\,\sigma[P2]\,(R)\,) = \sigma[P1 \wedge P2]\,(R)$

**Distributivitätsregeln:**

6) $\pi[R1]\,(\,R \cup S\,) = \pi[R1](R) \cup \pi[R1](S)$

7) $\sigma[P]\,(\,R \cup S\,) = \sigma[P](R) \cup \sigma[P](S)$

8) $\sigma[P1]\,(\,R \,|x|[P2]\, S\,) = \sigma[P1](R) \,|x|[P2]\, S$
   falls P1 nur R−Attribute enthält

9) $\pi[R1,S1]\,(\,R \,|x|[P]\, S\,) = \pi[R1](R) \,|x|[P]\, \pi[S1](S)$
   falls P−Attribute $\subseteq R1 \cup S1$

10) $R \,|x|[P]\, (\, S \cup T\,) = (R \,|x|[P]\, S) \cup (R \,|x|[P]\, T)$

**Invertierungsregeln:**

11) $\pi[sch(R)]\,(R \,\|\,{*}|\, S) = R$

**Weitere Regeln:**

12) $R \,|x|\, R = R$
13) $R \,|x|[sch(R){=}sch(S)]\, S = R \,|x|[K{=}M]\, S$
    falls K und M Schlüsselkandidaten von R bzw. S sind

...

## Transformation von SQL−Anfragen mit Subqueries

*Typ1:*

```
SELECT R.A FROM R
WHERE R.B θANY ( SELECT S.C FROM S
                 WHERE S.D ... )
```

ist äquivalent zu

```
SELECT R.A FROM R, S
WHERE R.B θ S.C AND S.D ...
```

*Typ2:*

```
SELECT R.A FROM R
WHERE R.B θANY ( SELECT AGGR (S.D) FROM S
                 WHERE S.E ...
                 AND S.F = R.C )
```

*Typ3:*

```
... WHERE NOT EXISTS ( SELECT S.C FROM S
                       WHERE S.D ... )
```

ist äquivalent zu

```
... WHERE 0 =    ( SELECT COUNT(S.C) FROM S
                   WHERE S.D ... )

... WHERE EXISTS ( SELECT S.C FROM S
                   WHERE S.D ... )
```

ist äquivalent zu

```
... WHERE 0 <    ( SELECT COUNT(S.C) FROM S
                   WHERE S.D ... )
```

...
weitere Transformationen

Lösungsansatz zu Typ2: Transformation in

```
1) T (X, Y) :=   SELECT S.F, AGGR (S.D) FROM S
                 WHERE S.E ...
                 GROUP BY S.F
2) SELECT R.A FROM R
   WHERE R.B θANY (   SELECT T.Y FROM T
                      WHERE T.X = R.C )
```

Diese Transformation ist aber inkorrekt!
Gegenbeispiel:

**Produkte**

| PNr | ... | Vorrat |
|---|---|---|
| 1 | | 100 |
| 2 | | 200 |

**Bestellungen**

| ... | Monat | ... | PNr | Menge | ... |
|---|---|---|---|---|---|
| | 12 | | 2 | 50 | |
| | 12 | | 2 | 40 | |

Resultat von

```
SELECT PNr FROM Produkte
WHERE Vorrat >= (   SELECT SUM(Menge) FROM Bestellungen
                    WHERE Monat=12
                    AND Bestellungen.PNr=Produkte.PNr )
```

| PNr |
|---|
| 1 |
| 2 |

Resultat von

```
T (X, Y) :=   SELECT PNr, SUM(Menge) FROM Bestellungen
              WHERE Monat=12
              GROUP BY PNr
```

**T**

| X | Y |
|---|---|
| 2 | 90 |

Resultat von

```
SELECT PNr FROM Produkte
WHERE Vorrat >=ANY  (   SELECT Y FROM T WHERE X=PNr)
```

| PNr |
|---|
| 2 |

Korrekt ist die Transformation in:

```
1) Q :=          SELECT * FROM S
                 WHERE S.E ...
2) T (X, Y) :=   SELECT R.C, AGGR(Q.D)
                 FROM Q RIGHT OUTER JOIN R ON (Q.F=R.C)
                 GROUP BY R.C
3) SELECT R.A FROM R
   WHERE R.B θANY (   SELECT T.Y FROM T WHERE T.X=R.C )
```

# 9.5 Zugriffskostenschätzung und Strategieauswahl

**Kostenfaktoren:**

| | |
|---|---|
| I/O−Kosten | Anzahl der Plattenzugriffe<br>ohne Berücksichtigung von Puffertreffern (aus Komplexitätsgründen) |
| CPU−Kosten | Annahme: proportional zur Anzahl der Tupel im (Zwischen−) Ergebnis<br>Zusatzannahme: ungefähr proportional zur Anzahl der Plattenzugriffe |
| Speicherkosten | Zeit−Platz−Produkt für den benutzten Hauptspeicher<br>(wird typischerweise vernachlässigt) |

[ Kommunikationskosten: nur in verteilten DBS relevant ]

**Optimierungsziel:**

Finde denjenigen Ausführungsplan, für den die gewichteten Kosten

w1 * I/O−Kosten + w2 * CPU−Kosten + w3 * Speicherkosten

minimal sind, wobei w1, w2, w3 vom Hersteller technologieabhängig gewählte Gewichte sind (die u.U. vom DBA verändert werden können).
Das Gewicht w3 ist typischerweise 0; das Gewicht w2 kann häufig vernachlässigt werden.

**Optimierungsmethode:**

Konzeptionell werden alle überhaupt möglichen Ausführungspläne enumeriert, die jeweiligen Kosten geschätzt und der insgesamt beste Ausführungsplan ausgewählt.
Wenn es sehr viele mögliche Ausführungspläne gibt (bei komplexen Queries), kann der Suchraum des Optimierers durch geeignete Heuristiken für kombinatorische Optimierungsprobleme reduziert werden (z.B. Branch−and−bound, Simulated Annealing). In diesem Fall ist allerdings die Optimalität des ausgewählten Ausführungsplans nicht mehr garantiert; solche Heuristiken führen jedoch in aller Regel zu einem ”guten” Ausführungsplan.

**Kostenschätzung:**

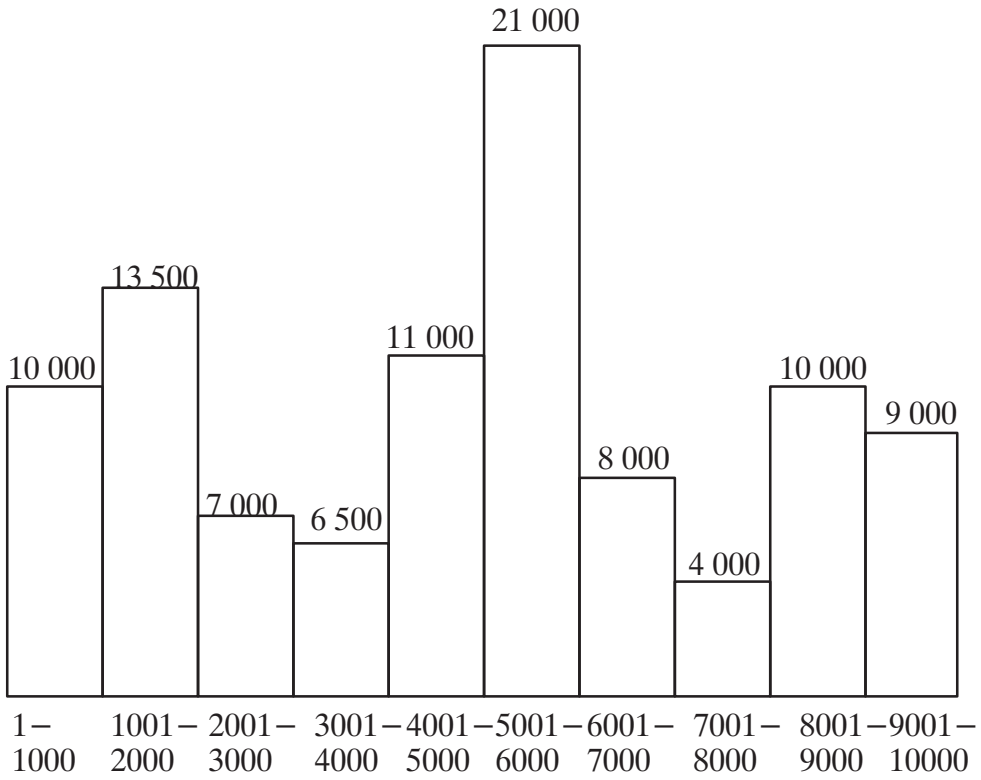
Kostenschätzungen basieren auf *Statistiken* über die Daten.
Damit die Aktualisierung der Statistiken den laufenden Betrieb nicht behindert, werden die Statistiken entweder nur auf expliziten Wunsch des DBA aktualisiert (in Oracle mit dem Kommando ANALYZE) oder vom DBS selbst wiederum geschätzt (z.B. aufgrund von Stichproben).

Relevante statistische Information sind u.a.:

- für jede Relation R:
  - T (R) die Anzahl der Tupel von R
  - S (R) die Anzahl der Seiten, in denen R gespeichert ist
- für jedes Attribut A einer Relation:
  - W (A) die Anzahl der verschiedenen vorkommenden Attributwerte
  - K (A) der kleinste vorkommende Attributwert
  - G (A) der größte vorkommende Attributwert
  - L (A) die (durchschnittliche) Länge eines Attributwerts
- für jeden Index I:
  - H (I) die Höhe des Index ($\geq 1$)
  - B (I) die Anzahl der Blattknoten des Index

Sehr leistungsfähige, moderne DBS verwenden darüber hinaus Statistiken über die Verteilung der Attributwerthäufigkeiten in Form von *Histogrammen*. Dabei wird der Wertebereich eines Attributes in eine relativ kleine Anzahl von – im einfachsten Fall – äquidistanten Werteintervallen aufgeteilt, und für jedes Intervall wird die Anzahl der in der Relation vorkommenden Tupel mit Attributwerten aus diesem Intervall registriert (sowie u.U. zusätzlich die Anzahl der vorkommenden verschiedenen Attributwerte in einem Intervall). Solche Histogramme werden typischerweise für alle Primärschlüssel und alle Fremdschlüssel geführt.

Beispielillustration eines Histogramms für Bestellungen.PNr
(mit 10 "Bins"=Werteintervallen) bei insgesamt 100 000 Bestellungs–Tupeln:



## Schätzung der I/O−Kosten für einige Ausführungsprimitive (auf der Basis einfacher statistischer Informationen)

Generelle Differenzierung (nicht in allen DBS):
- Kosten eines wahlfreien Seitenzugriffs: 1 RIO (random I/O)
- Kosten eines sequentiellen Seitenzugriffs: 1 SIO (sequential I/O)

Das Kostenverhältnis (bzgl. der Antwortzeit) zwischen 1 RIO und 1 SIO ist ungefähr 10.
(1 RIO entspricht ungefähr 15 msec, 1 SIO ungefähr 1.5 msec.)

**Relationen−Scan über Relation R:**

I/O−Kosten: S(R) * SIO

**Index−Scan über den mittelbaren Index $I_A$ für das Attribut A der Relation R:**

Fall 1: Filterprädikat A = wert

I/O−Kosten: $H(I_A) * RIO + \left(\frac{1}{W(A)} * B(I_A) - 1\right) * RIO$

Fall 2: Filterprädikat wert1 ≤ A ≤ wert2

I/O−Kosten: $H(I_A) * RIO + \left(\frac{wert2 - wert1}{G(A) - K(A)} * B(I_A) - 1\right) * RIO$

Die Terme $\frac{1}{W(A)}$ bzw. $\frac{wert2 - wert1}{G(A) - K(A)}$ werden als die *Selektivität* des Filterprädikats bezeichnet.
Notation: ϱ(F) ist die Selektivität eines Filterprädikats F.

Der Schätzung dieser Selektivitäten liegen die folgenden Annahmen zugrunde:
- Alle Werte des Suchschlüssels kommen in der Relation gleichhäufig vor.
- Die vorkommenden Werte sind gleichmäßig über das Intervall [K(A), G(A)] verteilt.

Bessere Schätzungen sind möglich auf der Basis von Histogrammen über Werteverteilungen; dies ist bei Scans aber nur bei dynamischer Optimierung (mit Kennnis der aktuellen Eingabeparameter einer Query) von Nutzen.

**TID−Zugriff nach Index−Scan(s) mit Filterprädikat F:**

Fall 1: der vorhergehende Index−Scan bezieht sich auf einen Clustered−Index
I/O−Kosten:

$$\rho(F) * S(R) * SIO = \begin{cases} \frac{1}{W(A)} * S(R) * SIO & \textit{falls F die Form A = wert hat} \\ \frac{w2 - w1}{G(A) - K(A)} * S(R) * SIO & \textit{falls F die Form } w1 \le A \le w2 \textit{ hat mit } w1 \ne w2 \end{cases}$$

Falls ϱ(F) * S(R) sehr klein ist (z.B. 1), dann sollte man statt mit SIO eigentlich mit RIO gewichten.

Fall 2: der vorhergehende Index−Scan bezieht sich auf einen Unclustered−Index
*Pessimistische Schätzung* der I/O−Kosten
(unter der Annahme, daß alle Tupel der TID−Liste auf verschiedenen Seiten liegen):

$$\rho(F) * T(R) * RIO = \begin{cases} \frac{1}{W(A)} * T(R) * RIO & \textit{falls F die Form A = wert hat} \\ \frac{wert2 - wert1}{G(A) - K(A)} * T(R) * RIO & \textit{falls F die Form } wert1 \le A \le wert2 \textit{ hat} \end{cases}$$

*Bessere Schätzung der Anzahl verschiedener Seiten, die von einer TID−Liste referenziert werden ("Ball−Color"−Problem):*

Gegeben:
− n = T(R): Anzahl der Tupel in R (Bälle)
− m = S(R): Anzahl der Seiten von R (verschiedene, gleichhäufige Farben der Bälle)
− t = ϱ(F)*T(R): Anzahl zufällig ausgewählter Tupel (zufällig ausgewählte Bälle)

Gesucht:
− k: Erwartungswert für die Anzahl verschiedener Seiten, auf denen die t ausgewählten Tupel liegen (verschiedener Farben der t ausgewählten Bälle)

Ansatz 1 (Cardenas, 1975):

Prob [eine Seite bzw. Farbe kommt nie vor] = $(\frac{m-1}{m})^t$
(Achtung: gilt eigentlich nur mit Zurücklegen der Seiten bzw. Bälle!)

⇒ Prob [eine Seite bzw. Farbe kommt mindestens einmal vor] = $1 - (\frac{m-1}{m})^t =: p$

⇒ k = Erwartungswert [Anzahl der Seiten bzw. Farben unter den t Tupeln bzw. Bällen]

$= mp = m\,(1 - (\frac{m-1}{m})^t)$ (Binomialverteilung)

$$\approx \begin{cases} m & falls\ t \geq 2m \\ (t+m)/3 & falls\ 2m > t \geq m/2 \\ t & falls\ m/2 > t \end{cases}$$

Beispiel:
n = 32, m = 4, t = 5 ⇒ k = 3
n = 300, m = 30, t = 50 ⇒ k = 25 nach der exakten Formel bzw. k ≈27 approximiert

Ansatz 2 (Yao, 1977):

Prob [Seite bzw. Farbe j kommt nie vor] = $\binom{n-b}{t} / \binom{n}{t}$ mit $b = \lfloor \frac{n}{m} \rfloor$

⇒ Prob [Seite bzw. Farbe j kommt mindestens einmal vor] = $1 - \binom{n-b}{t} / \binom{n}{t} =: p$

⇒ k = Erwartungswert [Anzahl der Seiten bzw. Farben unter den t Tupeln bzw. Bällen]

$$= mp = m * \left(1 - \binom{n-b}{t} / \binom{n}{t}\right) = m * \left[1 - \prod_{i=0}^{b-1} \frac{n-k-i}{n-i}\right]$$

Achtung:
Bei beiden Ansätzen wird − in approximativer, also i.a. eigentlich unzulässiger Weise − angenommen, daß die Wahrscheinlichkeiten für das Vorkommen verschiedener Seiten bzw. Farben voneinander unabhängig sind.

*Konsequenz für die Kostenschätzung:*
Wenn die TIDs in einer TID−Liste nach Seitennummer sortiert sind bzw. sortiert werden, kann man in der Kostenschätzung den Term ϱ(F) * T(R) durch k ersetzen.

## Kostenschätzung für komplexe Selektionen

**Index−Scans mit nachfolgendem TID−Listen−Durchschnitt und TID−Zugriff für konjunktive Prädikate der Form F1 ∧ F2:**

Selektivitätsschätzung: $\varrho(F) = \varrho(F1) * \varrho(F2)$
unter der Annahme, daß die Wahrscheinlichkeiten, daß ein Tupel F1 bzw. F2 erfüllt, unabhängig voneinander sind.

I/O−Kosten (mit pessimistischer Schätzung der TID−Zugriffskosten) für ein Prädikat der Form
A1=wert1 ∧ A2=wert2 ∧ ... ∧ An=wertn,
so daß für jedes Ai ein separater (mittelbarer, unclustered) Index benutzt wird:

$$\sum_{i=1}^{n}\left(H(I_{Ai}) + \frac{1}{W(Ai)} * B(I_{Ai}) - 1\right) * RIO \quad + \quad \prod_{i=1}^{n}\left(\frac{1}{W(Ai)}\right) * T(R) * RIO$$

Annahme: Der TID−Listen−Durchschnitt selbst findet im Hauptspeicher statt.

Konsequenz für den Optimierer:
Man sollte die Benutzung von Indizes in aufsteigender Reihenfolge ihrer Selektivität erwägen und einen Index dann nicht benutzen, wenn er nur noch einen marginalen Grenznutzen bringen würde.

**Index−Scans mit nachfolgender TID−Listen−Vereinigung und TID−Zugriff (für disjunktive Prädikate der Form F1 ∨ F2):**

Selektivitätsschätzung: $\varrho(F) = \varrho(F1) + \varrho(F2) - \varrho(F1) * \varrho(F2)$
unter der Annahme, daß die Wahrscheinlichkeiten, daß ein Tupel F1 bzw. F2 erfüllt, unabhängig voneinander sind.

I/O−Kosten (mit pessimistischer Schätzung der TID−Zugriffskosten) für ein Prädikat der Form
A1=wert1 ∨ A2=wert2,
so daß für A1 und A2 jeweils ein separater (mittelbarer, unclustered) Index ausgenutzt wird:

$$\sum_{i=1}^{2}\left(H(I_{Ai}) + \frac{1}{W(Ai)} * B(I_{Ai}) - 1\right) * RIO \quad + \quad \left(\frac{1}{W(A1)} + \frac{1}{W(A2)} - \frac{1}{W(A1) * W(A2)}\right) * T(R) * RIO$$

Annahme: Die TID−Listen−Vereinigung selbst findet im Hauptspeicher statt.

**Beispiele für die Kostenschätzung** (alle Indizes mittelbar und unclustered):

Statistiken des DBS:

| | **T**upel | **S**eiten | **W**erte | **K**leinstes | **G**rößtes | **L**änge | **B**lätter | **H**öhe |
|---|---|---|---|---|---|---|---|---|
| Bestellungen | 10 Mio. | 250 000 | | | | | | |
| KNr | | | 100 000 | 1 | 100 000 | 24 | 25 000 | 3 |
| PNr | | | 10 000 | 1 | 10 000 | 24 | 20 000 | 3 |
| Monat \|\| Tag | | | 300 | 1 \|\| 3 | 12 \|\| 30 | 24 | 21 600 | 3 |

I/O−Kosten der Query
σ [KNr = :knr] (Bestellungen) mit Index−Scan und TID−Zugriff:

( 3 + ⌈ 1/100000 * 25000⌉ − 1 + ⌈ 10 000 000 / 100 000⌉ ) * RIO = 103 RIO

I/O−Kosten der Query
σ [Monat=11] (Bestellungen) mit Index−Scan und TID−Zugriff
(Filterprädikat: Monat=11 ∧ 1 ≤ Tag ≤ 30):

( 3 + ⌈ 30/360 * 21600⌉ − 1 + ⌈ 30/360 * 10 000 000⌉ ) * RIO = (3 + 1800 − 1 + 833334) RIO

I/O−Kosten der Query
σ [Monat=11 ∧ Tag=11 ] (Bestellungen)

mit Index−Scan und TID−Zugriff:

( 3 + ⌈ 1/300 * 21600⌉ − 1 + ⌈ 1/300 * 10 000 000⌉ ) * RIO = (3 + 72 − 1 + 33334) RIO

mit Relationen−Scan:

250000 SIO (ca. 375 sec gegenüber ca. 500 sec beim Index−Scan)

I/O−Kosten der Query
σ [Tag=30] (Bestellungen) mit Index−Scan und TID−Zugriff:

( 3 + 21600 − 1 + ⌈ 12/360 * 10 000 000⌉ ) * RIO = (3 + 21600 − 1 + 333333) RIO
bzw. 12*(3 + 21600/300 − 1) + 333333 RIO bei verbesserter Traversierung der Indexblätter
bzw. 3 − 1 + 300 − 12 + 12 * 21600/300 + 400000 RIO

I/O−Kosten der Query
σ [KNr=:knr ∧ PNr=:pnr ∧ Monat=:mon ∧ Tag=:tag] (Bestellungen)
mit 3 Index−Scans, TID−Listen−Durchschnitt und TID−Zugriff:

( 3 + ⌈ 1/100000 * 25000⌉ − 1 + 3 + ⌈ 1/10000 * 20000⌉ − 1 + 3 + ⌈ 1/300 * 21600⌉ − 1
+ ⌈ 1/100000 * 1/10000 * 1/300 * 10000000⌉ ) * RIO
= (3 + 4 + 74 + 1) RIO = 82 RIO

I/O−Kosten der Query
σ [KNr=:knr ∧ PNr=:pnr ∧ Monat=:mon ∧ Tag=:tag] (Bestellungen)
mit 2 Index−Scans für die beiden selektivsten Indizes,
TID−Listen−Durchschnitt und TID−Zugriff:

( 3 + ⌈ 1/100000 * 25000⌉ − 1 + 3 + ⌈ 1/10000 * 20000⌉ − 1
+ ⌈ 1/100000 * 1/10000 * 10000000⌉ ) * RIO
= (3 + 4 + 1) RIO = 8 RIO
(ca. 120 msec gegenüber 1.23 sec bei 3 Index−Scans:
signifikanter Gewinn im Hinblick auf Durchsatz und Mehrbenutzer−Antwortzeit)

## Optimierung der Join−Reihenfolge bei Mehrwege−Joins

1) Zerlegung von Mehrwege−Joins (zwischen mehr als 2 Relationen) in mehrere binäre Joins
2) Festlegung der Reihenfolge der Join−Berechnungen (Wahlfreiheit wegen Assoziativität und Kommutativität des Joins)
3) Festlegung der Join−Berechnungsstrategie für die einzelnen Joins

Beispiel für das Problem der Join−Reihenfolge:

σ[Stadt='Homburg'] (Kunden) |x|
σ[Monat=11] (Bestellungen) |x|
σ[Lagerort='Merzig'] (Produkte)

**σ[...] (Kunden)**

| KNr | ... |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |

**σ[...] Bestellungen**

| KNr | PNr | ... |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 5 | 1 | |
| 6 | 2 | |
| 7 | 1 | |
| 8 | 2 | |
| 1 | 3 | |
| 2 | 3 | |
| 5 | 3 | |
| 6 | 3 | |
| 7 | 2 | |
| 8 | 1 | |

**σ[...] (Produkte)**

| PNr | ... |
|---|---|
| 1 | |
| 2 | |

Mögliche Join−Reihenfolgen (unter Vermeidung kartesischer Produkte):

1) ( σ[...] (Kunden) |x| σ[...] Bestellungen ) |x| σ[...] (Produkte)
   → 4 Resultattupel beim ersten Join
   → 2 Resultattupel beim zweiten Join
2) σ[...] (Kunden) |x| ( σ[...] Bestellungen |x| σ[...] (Produkte) )
   → 8 Resultattupel beim ersten Join
   → 2 Resultattupel beim zweiten Join

⇒ Reihenfolge 1 ist günstiger

Bemerkung:

Bei Join−Queries über (Join−) Views tritt dieses Problem in der Praxis u.U. für Joins zwischen 5 bis 10 Relationen auf.

Lösung:

Schätzung der Zwischenresultsgrößen der verschiedenen Join−Reihenfolgen und Auswahl derjenigen Join−Reihenfolge mit den kleinsten Zwischenresultaten

Sei A das Joinattribut für den Join R |x| S, seien alle in R bzw. S vorkommenden A−Werte in R bzw. S gleichhäufig und gelte $\pi[A](R) \subseteq \pi[A](S)$.
Dann gilt für die Kardinalität card (R |x| S):

$$\text{card}(R \mid x \mid S) = T(R) * \frac{T(S)}{W(S.A)}$$

# 9.6 Optimierung verteilter Queries

Bei verteilten Datenbanken mit einer relativ langsamen WAN−Verbindung zwischen den verschiedenen Rechnern ist das Primärziel der Anfrageoptimierung die Minimierung der Kommunikationskosten. Die Optimierung muß für jeden Knoten des Operatorbaums (inkl. einer übergeordneten Wurzel zur Resultatsausgabe) den Rechner festlegen, auf dem der Operator ausgeführt werden soll.

Beispiel−Query in Merzig:
σ[Gewicht>100](Produkte) |x| Bestellungen
mit der Tabelle “Produkte” in Saarlouis und “Bestellungen” in Homburg

Mögliche verteilte Ausführungspläne sind:

1) Sende ”σ Produkte” nach Merzig
Sende ”Bestellungen” nach Merzig
Berechne den Join in Merzig

2) Sende ”σ Produkte” nach Homburg
Berechne den Join in Homburg
Sende das Joinresultat nach Merzig

3) Sende ”Bestellungen” nach Saarlouis
Berechne den Join in Saarlouis
Sende das Joinresultat nach Merzig

Annahmen:
− card(Produkte) = 10000
− card (σ[Gewicht>100](Produkte) = 1000
− card(Bestellungen) = 20000
− Das Attribut “Bez” ist 50 Bytes lang; alle anderen Attribute sind 4 Bytes lang.

Kommunikationskosten:

1) SLS −> MZG: 1000 Tupel * 58 Bytes
HOM −> MZG: 20000 Tupel * 12 Bytes
insgesamt: 298000 Bytes
2) SLS −> HOM: 1000 Tupel * 58 Bytes
Joinberechnung liefert 2000 Tupel
HOM −> MZG: 2000 Tupel * 66 Bytes
insgesamt: 190000 Bytes
3) HOM −> SLS: 20000 Tupel * 12 Bytes
Joinberechnung liefert 2000 Tupel
SLS −> MZG: 2000 Tupel * 66 Bytes
insgesamt: 372000 Bytes

**Semi−Joins**

Unter gewissen Bedingungen lassen sich die Kommunikationskosten für Joinoperanden weiter reduzieren, indem der Operatorbaum um zusätzliche Semi−Joins erweitert wird. Die Idee dabei ist, solche Tupel aus einer zu übertragenden Relation vorab zu eliminieren, die keine Partnertupel in dem jeweils anderen Joinoperanden am Zielort haben.

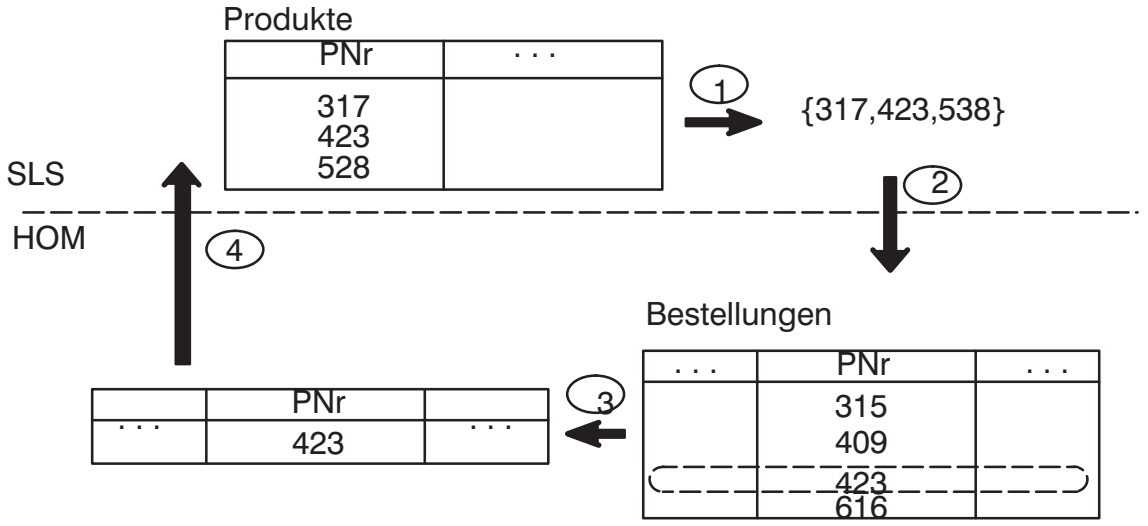
Der Einsatz von Semi−Joins beruht auf der folgenden Äquivalenz

$$R \ltimes S = (R \ltimes S) \ltimes S \qquad (\text{mit } R \ltimes S := \pi[\text{sch}(R)] (R \ltimes S) = R \ltimes \pi[\text{Joinattribute}] (S) ).$$

Der Semi−Join $R \ltimes S$ ist potentiell vorteilhaft bei einer Berechnung des Joins am Ort von S. Der Semi−Join selbst wird dabei wie folgt implementiert:

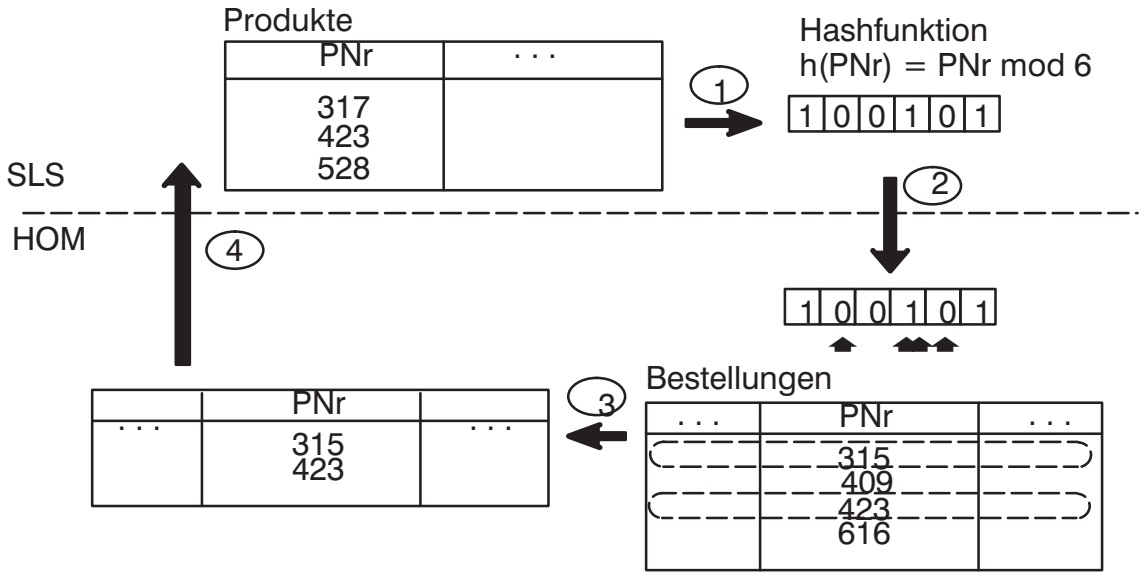
1) Berechne die Projektion π[Joinattribute] (S) am Ort von S
2) Sende das Resultat der Projektion an den Ort von R
3) Berechne den Semi−Join $R \ltimes \pi[\text{Joinattribute}](S)$ am Ort von R
4) Sende das Resultat des Semi−Joins (also R−Tupel) an den Ort von S
5) Berechne den Join am Ort von S

Beispiel: Semi−Join bei Join−Berechnung in SLS




Approximation des Semi−Join−Resultats R |x S durch Hashfilter:

1) Bilde π[Joinattribut](S) per Hashfunktion auf einen Bitvektor fester Länge ab
2) Sende den Bitvektor an den Ort von R
3) Bilde π[Joinattribut](R) per Hashfunktion auf den Bitvektor ab und sende die R−Tupel, die ein gesetztes Bit vorfinden, an den Ort von S




**Schätzung der Selektivität von Semi−Joins:**

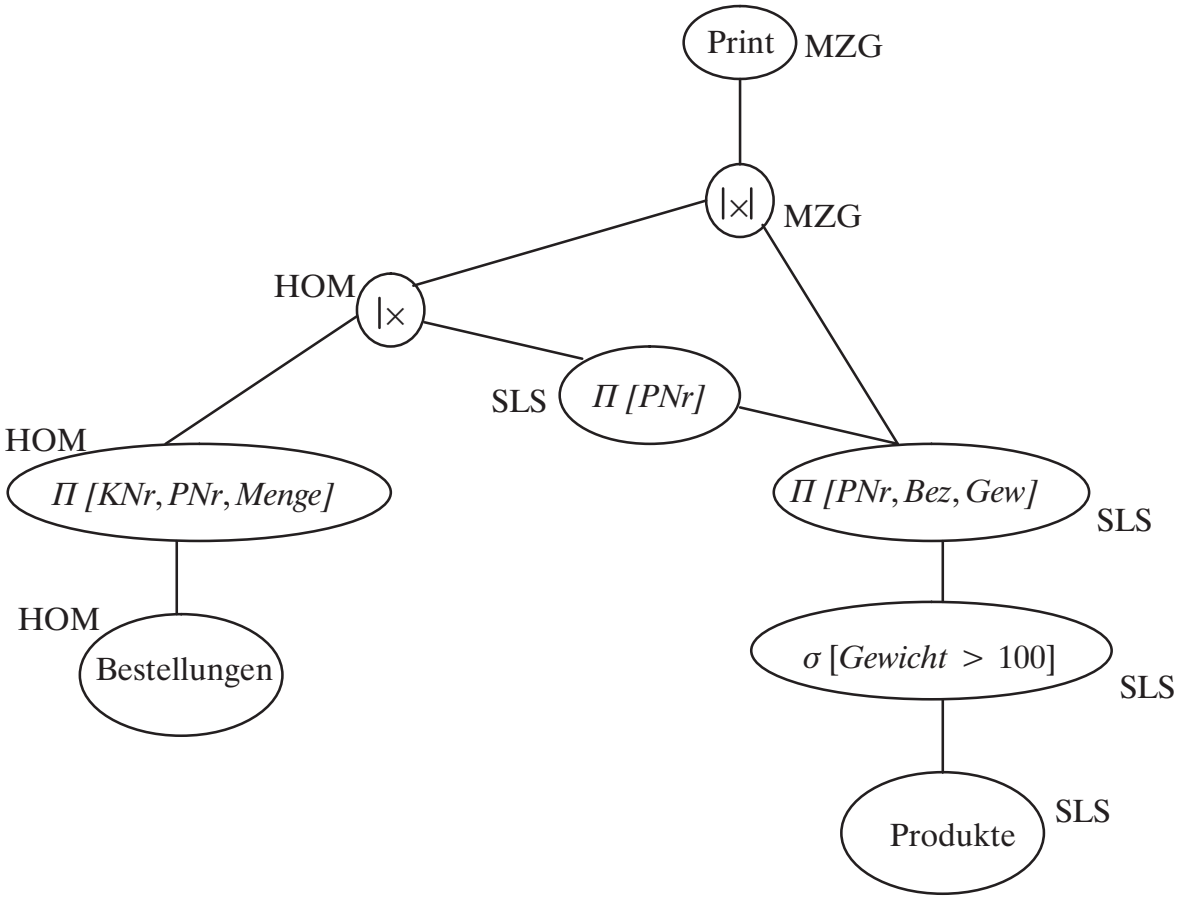
Größe des Resultats von R |x S und S |x R mit Joinattibut. A:

$$card(R \mid \times\ S) = card(R) * \frac{card(\pi_{distinct}[A](S))}{card(\pi_{distinct}[A](R))}$$

$$card(S \mid \times\ R) = card(S)$$

unter den Annahmen:
1) $\pi[A](S) \subseteq \pi[A](R)$
2) Gleichverteilung der Attributwerte

Verbesserung des Ausführungsplans 1 durch einen Semi−Join:




Annahmen:
- card(Produkte) = 10000
- card (σ[Gewicht>100](Produkte) = 1000
- card(Bestellungen) = 20000
- Das Attribut “Bez” ist 50 Bytes lang; alle anderen Attribute sind 4 Bytes lang.

Allgemeinere Schätzung der Größe des Join−Resultats R |x| S
(bei gleichverteilten Joinattributwerten in R und S):

$$card(R \mid \times \mid S) \;=\; card(R \mid \times\; S) * \frac{card(S \mid \times\; R)}{card(\pi_{\,distinct}[A](S \mid \times\; R))}$$

Kommunikationskosten:
- SLS−>HOM:
  1000 Tupel * 4 Bytes = 4’000 Bytes
- Semi−Join−Berechnung:
  20000 * 1000/10000 = 2000 Tupel
- HOM−>MZG:
  2000 Tupel * 12 Bytes = 24’000 Bytes
- SLS−>MZG:
  1000 Tupel * 58 Bytes = 58’000 Bytes

insgesamt: 86’000 Bytes
(gegenüber 58’000 + 240’000 = 298’000 Bytes ohne Semi−Join)

# 9.7 Parallelisierung der Query−Ausführung

Datenbankoperationen lassen sich parallelisieren, da sie auf großen Mengen von Tupeln arbeiten. Die Parallelisierung kann durch den Query−Übersetzer und −Optimierer automatisch erfolgen (also transparent für das Anwendungsprogramm).

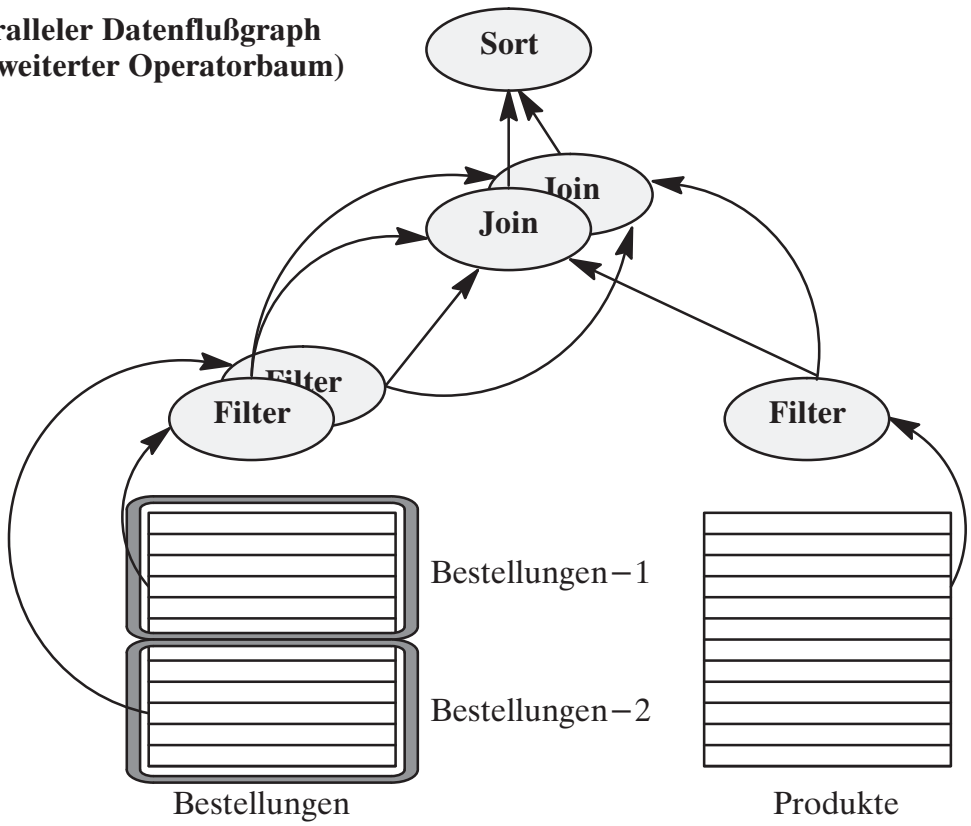
**Parallelitätsarten:**

- *Funktionsparallelität* (horizontale Interoperator−Parallelität):
  parallele Ausführung von unabhängigen Bruderknoten im Operatorbaum
- *Datenparallelität* (Intraoperator−Parallelität):
  Partitionierung einer Tupelmenge und parallele Ausführung eines Operators auf allen Partitionen
- *Pipelining* (vertikale Interoperator−Parallelität):
  überlappende Ausführung von Vater− und Sohnknoten im Operatorbaum (der Sohnknoten produziert eine Tupelmenge, die der Vaterknoten konsumiert)

  Pipelining wird − im Sinne von Koroutinen − auch bei nichtparalleler Ausführung verwendet, um Zwischenergebnisse nicht zwischenspeichern zu müssen.

Beispiel−Query:
Erstelle eine nach Produktbezeichnung sortierte Liste der Bestellungen von Produkten vom Lagerort Homburg, die nach dem 1. November erfolgt sind.

**Paralleler Datenflußgraph (erweiterter Operatorbaum)**




| | |
|---|---|
| Filter (Bestellungen) \|\| Filter (Produkte) | *Funktionsparallelität* |
| Filter (Bestellungen−1) \|\| Filter (Bestellungen−2) | *Datenparallelität* |
| Filter (Produkte) \|\| Join | *Pipelining* |

**Parallelisierung der Join−Operation**

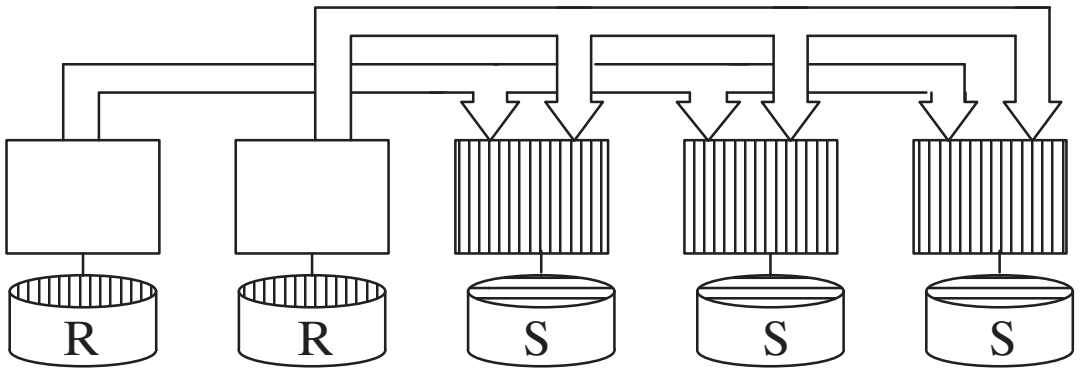
Berechnung von R |x| S durch
1) Partitionierung von R in R1, ..., Rn und/oder
   Partitionierung von S in S1, ..., Sn
2) Parallele Berechnung der Teil−Joins Ri |x| Si

Dabei wird zusätzlich angestrebt, die Partitionierungs− und die Joinberechnungsphasen durch Pipelining zu überlappen. Außerdem kann u.U. die bereits vorliegende partitionierte Speicherung von R und/oder S, die in der Regel über mehrere Platten und/oder Prozessoren verteilt sind, ausgenutzt werden.

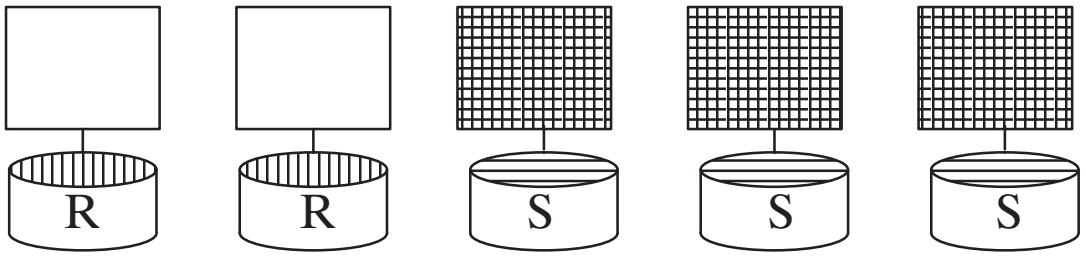
Die auf diesen Überlegungen aufbauenden parallelen Join−Algorithmen können sowohl auf Parallelrechnern mit verteiltem Speicher und schnellem Kommunikationsnetzwerk als auch auf Shared−Memory−Parallelrechnern realisiert werden, wobei im letzteren Fall zusätzliche Effizienzsteigerungen in der Implementierung möglich sind.

**Variante 1: Parallelisierter Nested−Loop−Join**
**(Broadcast Join, Fragment&Replicate Join)**

1. Phase (Broadcast R): sende alle R−Tupel zu allen Prozessoren, die S−Tupel halten

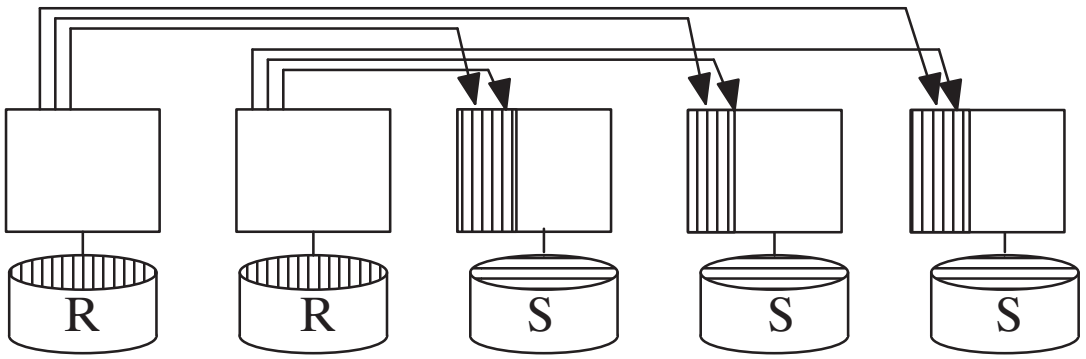



2. Phase (Join−Phase): berechne Teil−Joins in allen S−Prozessoren

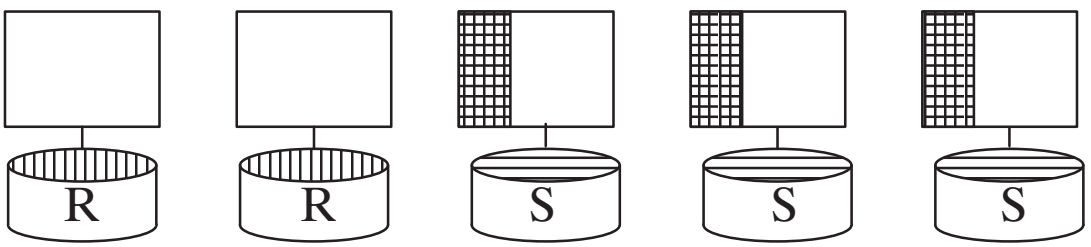



Spezialfall “Assoziativer Join”:
Falls S nach dem Join−Attribut partitioniert gespeichert ist, genügt es, in der 1. Phase die R−Tupel nur an den jeweils passenden S−Prozessor zu schicken.

1. Phase:

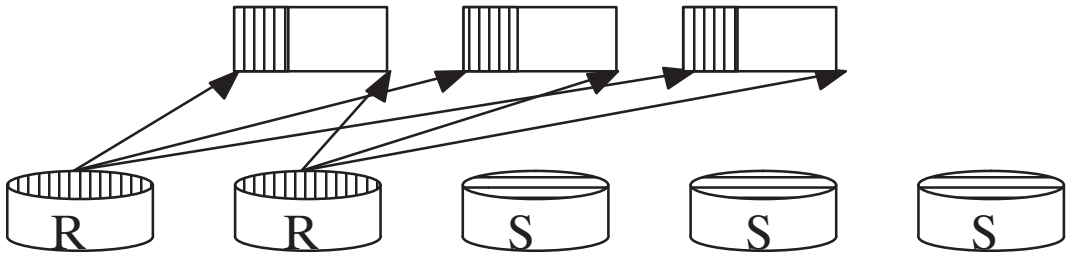



2. Phase

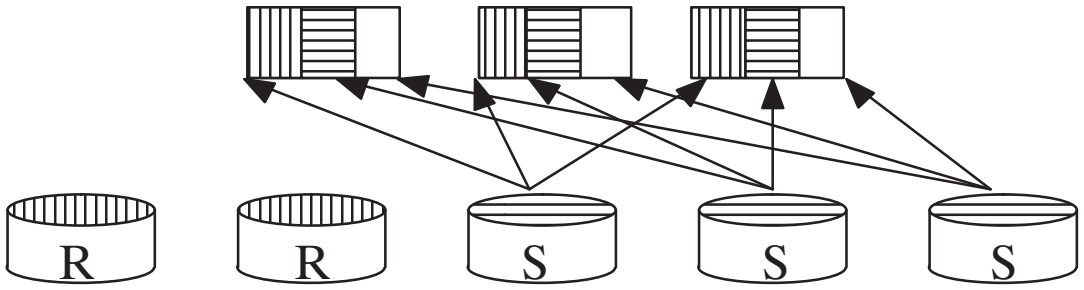


**Variante 2: Parallelisierter Hash−Join**

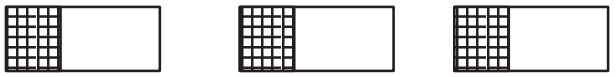
1. Phase (Partitionierung von R):
verteile R−Tupel durch Hashing auf dem Joinattribut an eine Menge von Join−Prozessoren (die sich mit den Heimatprozessoren von R oder S überlappen dürfen)




2. Phase (Partitionierung von S):
verteile S−Tupel an die Join−Prozessoren durch Hashing auf dem Joinattribut unter Verwendung derselben Hashfunktion wie in der 1. Phase



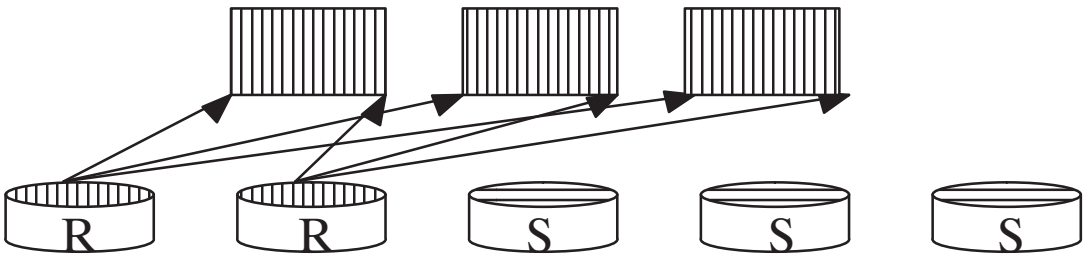

3. Phase (Joinberechnung): berechne die Teil−Joins auf den Join−Prozessoren



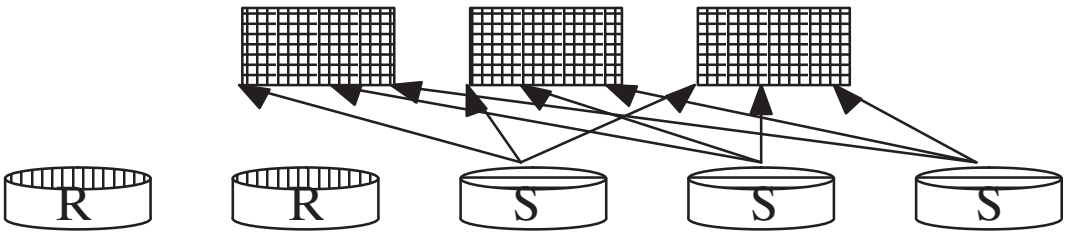
**Verbesserte Variante: Asymmetrischer parallelisierter Hash−Join**

1) *Build−Phase*: Bei Ankunft der R−Tupel auf den Join−Prozessoren wird dort jeweils eine Hash−Tabelle mit den R−Tupeln aufgebaut
2) *Probe−Phase*: Bei Ankunft eines S−Tupels auf einem Join−Prozessor wird dort jeweils direkt die Hash−Tabelle sondiert
(Pipelining zwischen der Partitionierung von S und der lokalen Joinberechnung)

Build−Phase:

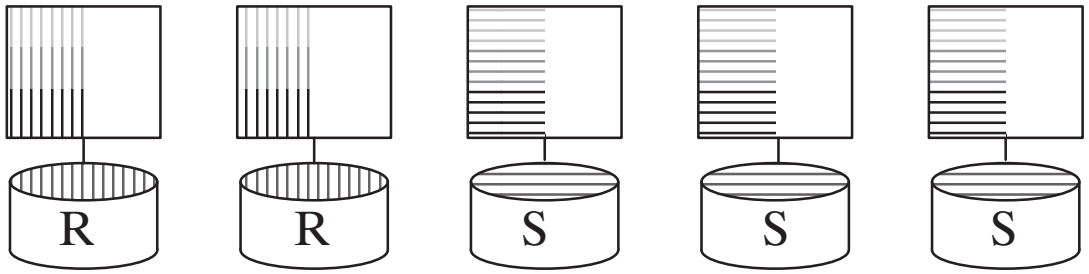



Probe−Phase:

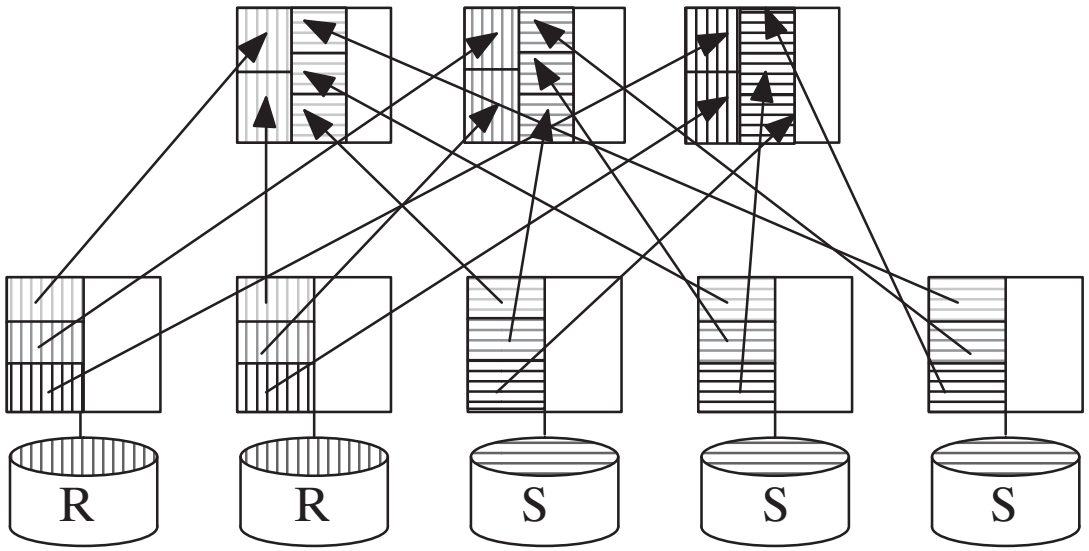


**Variante 3: Parallelisierter Sort−Merge−Join**

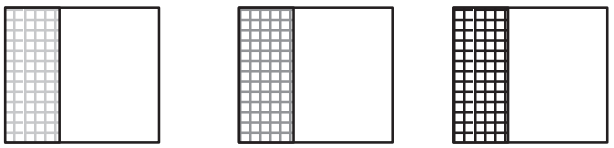
1. Phase (lokale Sortierung):
Generierung sortierter Fragmente (”Runs”) auf den Heimatprozessoren von R und S




2. Phase (Partitionierung von R und S):
Bestimmung globaler Werteintervallgrenzen für m Intervalle,
Partitionierung der R− und S−Fragmente aller Prozessoren in jeweils m Teilfragmente aufgrund dieser Werteintervallgrenzen und
Verteilung der Teilfragmente an m Join−Prozessoren, so daß
das jeweils i−te Teilfragment dem Join−Prozessor i zugeteilt wird

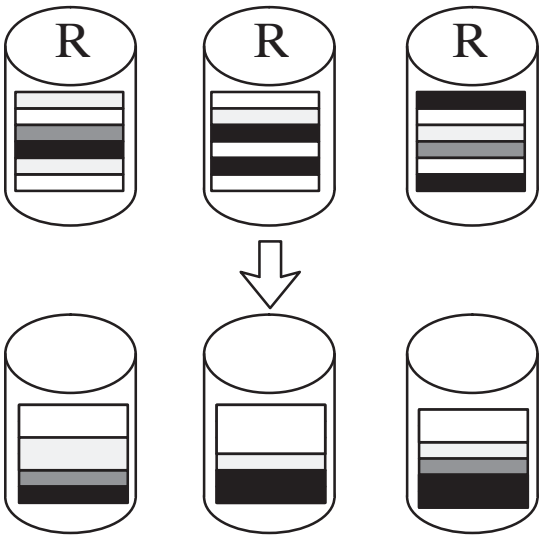



3. Phase (lokales Mischen):
Mischen der zusammengehörigen R− und S−Teilfragmente auf den Join−Prozessoren

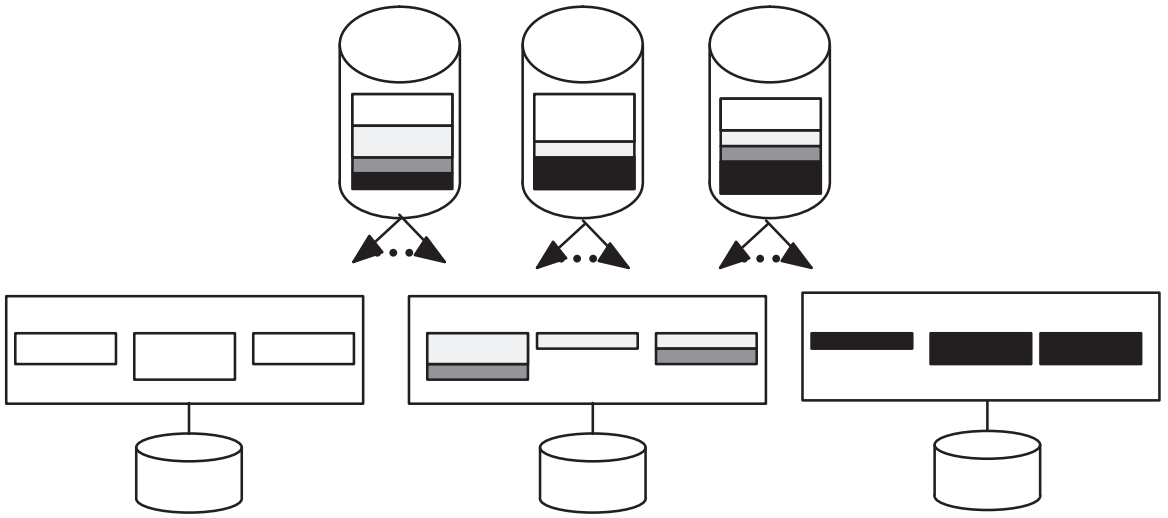


**Parallelisierter Sortier−Operator**

1. Phase (lokale Sortierung):
Generieren sortierter Fragmente (”Runs”) pro Prozessor, dem R−Tupel zugewiesen sind



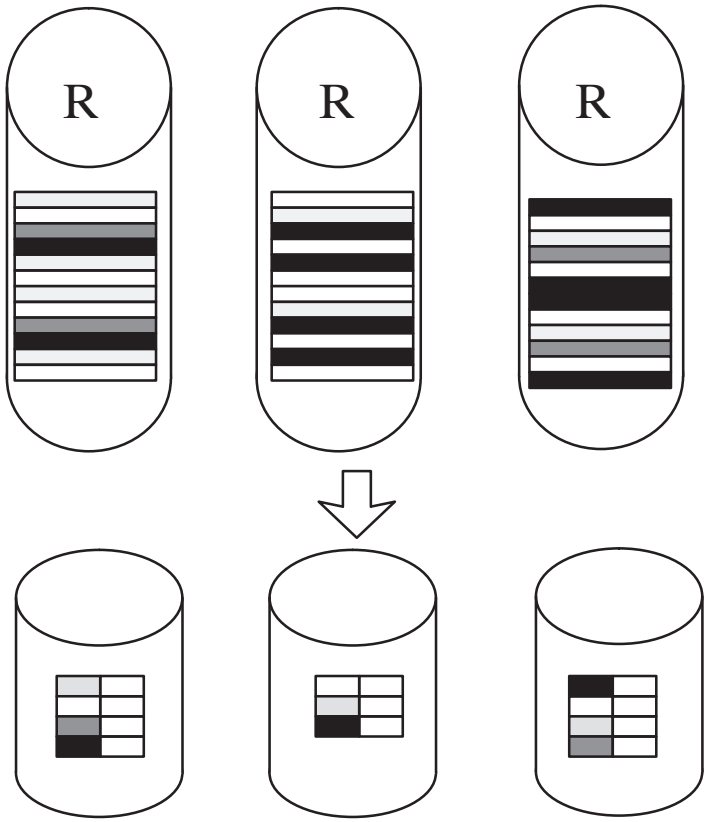

2. Phase (Repartitionierung der Daten):
Bestimmung globaler Werteintervallgrenzen für m Intervalle,
Partitionierung der Fragmente aller Prozessoren aufgrund dieser Intervallgrenzen und
Verteilung der resultierenden m Teilfragmente pro Prozessor an m Prozessoren, so daß
das jeweils i−te Teilfragment dem Prozessor i zugeteilt wird



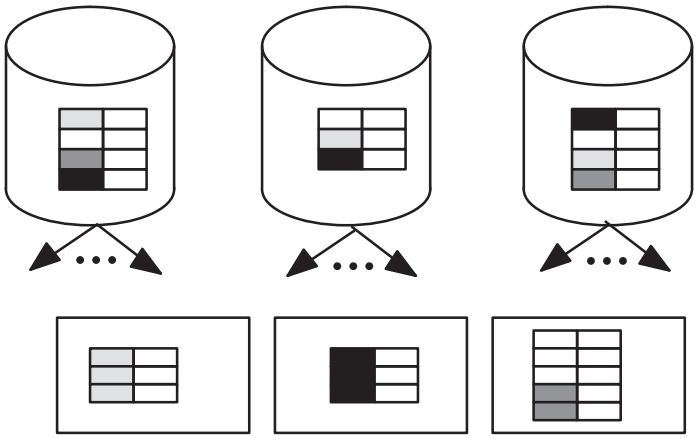
3. Phase (lokales Mischen):
Mischen aller Teilfragmente mit demselben Werteintervall, die jeweils einem der
m Prozessoren zugeteilt werden

**Parallelisierte Aggregation mit Gruppierung**

1. Phase (lokale Gruppierung und Aggregation):
Gruppierung der lokalen Daten auf jedem Prozessor mit R−Tupeln und
Berechnung lokaler Aggregationsfunktionswerte für jede Gruppe




2. Phase (Repartitionierung der Daten):
Verteilung der Gruppen mit ihren zugehörigen Aggregationsfunktionswerten durch Hashing auf dem Gruppierungsattribut



3. Phase (globale Aggregation):
Berechnung der globalen Aggregationsfunktionswerte für jede Gruppe

# Ergänzende Literatur zu Kapitel 9:

C. Yu, W. Meng, Principles of Database Query Processing for Advanced Applications, Morgan Kaufmann, 1998

G. Graefe, Query Evaluation Techniques for Large Databases, ACM Computing Surveys Vol.25 No.2, June 1993, pp. 73−170

S. Chaudhuri, An Overview of Query Optimization in Relational Systems, ACM Symposium on Principles of Database Systems, Seattle, 1998

T. Härder, E. Rahm, Datenbanksysteme − Konzepte und Techniken der Implementierung, Springer−Verlag, 1998

T. Härder, Realisierung von operationalen Schnittstellen, in: P.C. Lockemann, J.W. Schmidt (Hrsg.), Datenbank−Handbuch, Springer−Verlag, 1987

Oracle7 Server Concepts Manual, Chapter 13: The Optimizer

Bulletin of the IEEE TC on Data Engineering Vol. 16 No. 4, December 1993, Special Issue on Query Processing in Commercial Database Systems

Bulletin of the IEEE TC on Data Engineering Vol. 18 No. 3, September 1995, Special Issue on Database Query Optimization and Query Processing;
insbesondere: Y. Ioannidis, V. Poosala, Histogram−Based Solutions to Diverse Database Estimation Problems

Bulletin of the IEEE TC on Data Engineering Vol.19 No.4, December 1996, Special Issue on Query Processing for Non−Standard Data

Bulletin of the IEEE TC on Data Engineering Vol.20 No.1, March 1997, Special Issue on Supporting On−line Analytical Processing

J.C. Freytag, D. Maier, G. Vossen (Editors), Query Processing for Advanced Database Applications, Morgan Kaufmann, 1994

D. DeWitt, J. Gray, Parallel Database Systems: The Future of High Performance Database Systems, Communications of the ACM Vol.35 No.6, June 1992, pp. 85−98

E. Rahm, Mehrrechner−Datenbanksysteme, Addison−Wesley, 1994

Bulletin of the IEEE TC on Data Engineering Vol.20 No.2, June 1997, Special Issue on Commercial Parallel Systems