## Vorlesung Implementierung von Datenbanksystemen

# 11. Relationale Operatoren

**Prof. Dr. Klaus Meyer-Wegener**

**Wintersemester 2019/20**

**Lehrstuhl für Informatik 6 (Datenmanagement)**
Friedrich-Alexander-Universität Erlangen-Nürnberg

# 11.1 Transformation



- **Grundsätzliche Aufgabe:**
  - Ersetzen der Logischen Operatoren (SEL(), PROJ(), JOIN(), … ) durch (ausführbare) **Planoperatoren**
- **Teilprobleme**
  - Gruppierung von direkt benachbarten Operatoren zur Auswertung durch einen einzigen Planoperator
    - Beispiel: Verbund mit Selektionen und/oder Projektionen auf den beteiligten Relationen durch einen speziellen Planoperator gemeinsam ausführen.
  - Bestimmung der Reihenfolge bei Verbundoperationen
    - Ziel: minimale Kosten für die Operationsfolge
    - Heuristik: Minimierung der Größe der Zwischenergebnisse, d.h. die kleinsten (Zwischen-)Relationen immer zuerst verknüpfen
  - Erkennen gemeinsamer Teilbäume
    - Einmalige Berechnung
    - Dafür nötig: Zwischenspeicherung der Ergebnisrelation

# Relationale Planoperatoren



- **Ausführbar**
  - Unterprogramm / Komponente des DBMS
- **Parameter u.a.:**
  - Eingabe-Relationen
  - Zu verwendende Indexstrukturen
  - Bedingungen (Prädikate)
- **Voraussetzungen:**
  - Vorhandensein bestimmter Speicherungsstrukturen (vor allem: Indexe)
- **Ergebnis (Ausgabe):**
  - Ganze Relation oder
  - Nächstes Tupel (Pipelining) oder
  - Nächste *n* Tupel
- **Kosten der Verarbeitung**
  - Zeit, Speicher, CPU, …

# Planoperatoren (2)



- **SQL erlaubt komplexe Anfragen über k Relationen.**

- **Ein-Variablen-Ausdrücke**
  - beschreiben Bedingungen für die Auswahl von Elementen aus einer Relation.
- **Zwei-Variablen-Ausdrücke**
  - beschreiben Bedingungen für die Kombination von Elementen aus zwei Relationen.
- **k-Variablen-Ausdrücke**
  - werden typischerweise in Ein- und Zwei-Variablen-Ausdrücke zerlegt und durch entsprechende Planoperatoren ausgewertet.

# Planoperatoren (3)



- **Selektion**
  - Mehrere Planoperatoren zur Auswahl
  - Nutzung eines **Scan**-Operators
    - Wahlweise mit Definition von Start- und Stopp-Bedingung und von einfachen Suchargumenten (z.B. Attribut = Wert)
  - Relationen-Scan
    - Sequenzielles Lesen aller Tupel einer Relation
  - Index-Scan
    - Direkt auf erstes passendes Tupel springen, dann ggf. sequenziell die weiteren
    - Nach Auswahl des kostengünstigsten Index
- **Projektion**
  - Typischerweise mit im Planoperator von Sortierung, Selektion oder Verbund durchgeführt
    - Denn die bearbeiten sowieso jedes Tupel.
  - Zusätzlich auch noch als eigener Planoperator

# Planoperatoren (4)

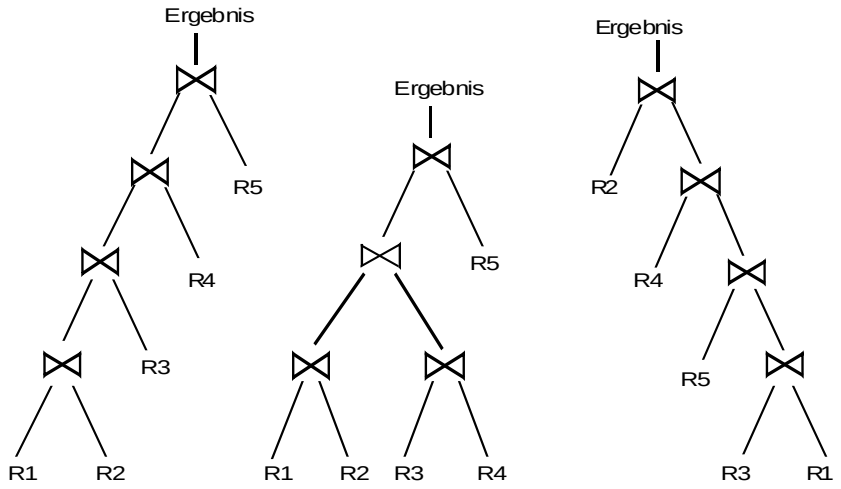


- **Sortierung**
  - Erforderlich bei ORDER BY
    - Aber auch zur Beschleunigung von Joins
      - Siehe unten
    - Und für Gruppierung
    - Und zur Duplikateliminierung (DISTINCT, Mengenoperationen)
  - Deshalb eigener Planoperator
  - Leider "blockierend":
    - Muss das letzte Eingabetupel abwarten,
      bevor die erste Ausgabe erzeugt werden kann
    - Kein Pipelining möglich
  - I.Allg. extern, d.h. sortierte Teilergebnisse müssen auf Hintergrundspeicher ausgelagert werden (sog. Runs)
    - Am Schluss zum Gesamtergebnis zusammenmischen (Merge)

# Planoperatoren (5)



- **Join über mehreren Relationen (n-Wege-Verbunde)**
  - Zerlegung in *n* – 1 Zwei-Wege-Verbunde
  - Anzahl der Verbundreihenfolgen abhängig von den Verbundattributen
    - *n*! verschiedene Reihenfolgen möglich
  - Optimale Auswertungsreihenfolge abhängig von
    - Planoperatoren,
    - "passenden" Sortierordnungen für Verbundattribute,
    - Größe der Operanden usw.
  - Verschiedene Verbundreihenfolgen mit Zwei-Wege-Verbunden (*n* = 5):



# Planoperatoren für den Verbund



- **Eigenschaften der Verbundoperation:**
  - Teuer und häufig → Optimierungskandidat !!!
  - Typisch: Gleichverbund; allgemeines Verbundprädikat eher selten
  - Standardszenario:

```
SELECT * FROM R, S
WHERE R.VA Θ S.VA          // Verbundprädikat,
                           // Θ ∈ {=, >, <, ≠, ≤, ≥}
AND P(R.SA)                // lokale Selektionen
AND P(S.SA);               // VA = Verbundattribut,
                           // SA = Selektionsattribut
```

- **Mögliche Zugriffspfade:**
  - Relationen-Scan über R und S
  - Scans über Indexe $I_R(VA)$ und $I_S(VA)$, falls vorhanden
    - Dabei besonders interessant: Sortierreihenfolge nach R.VA und S.VA !!!
  - Scans über Indexe $I_R(SA)$ und/oder $I_S(SA)$, falls vorhanden
    - Schnelle Selektion für R.SA und S.SA !!!
  - Beliebige andere Kombinationen

# Nested-Loop-Verbund



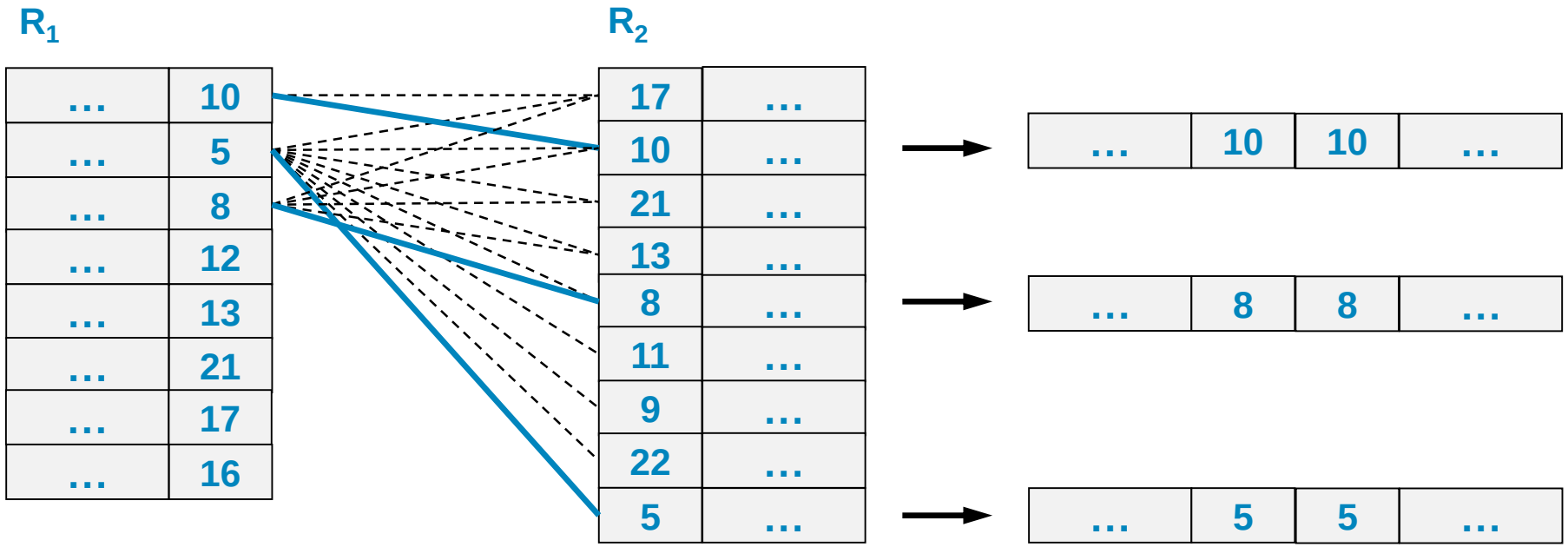
- **Annahmen:**
  - Sätze in R und S sind nicht nach den Verbundattributen geordnet.
  - Es sind keine Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden.
- **Algorithmus für Θ-Verbund:**

```
Scan über S;                    // äußere Schleife
für jeden Satz s, für den P(s.SA) gilt:
    Scan über R;                // innere Schleife
    für jeden Satz r,
    für den P(r.SA) AND (r.VA Θ s.VA) gilt:
        übernimm kombinierten Satz (r, s)
        in das Ergebnis;
```

- **Komplexität:**
  - $O(N^2)$

# Beispiel zum Nested-Loop-Verbund






- **Merke:**
  - Falls das Verbundattribut der zweiten Relation die Eigenschaft UNIQUE aufweist, kann die Suche abgebrochen werden, nachdem der erste Verbundpartner gefunden wurde.

# Nested-Loop-Verbund mit Indexzugriff



- **Annahme:**
  - Es sind Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden.
- **Algorithmus für Gleichverbund mit Indexzugriff:**

```
Scan über S;
für jeden Satz s, für den P(s.SA) gilt:
   ermittle über I_R(R.VA) alle (TIDs der) Sätze mit r.VA = s.VA;
   ggf. für jedes TID: hol Satz r;
   für jeden Satz r, für den P(r.SA) gilt:
        übernimm kombinierten Satz (r, s) in das Ergebnis;
```

- **Merke:**
  - Eigentlich wird seitenweise vorgegangen.
    - Mehrfachen Zugriff auf dieselbe Seite vermeiden

# Sort-Merge-Verbund



- **Zweiphasiger Algorithmus**
  - 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 P(R.SA) bzw. P(S.SA))
  - Phase 2:
    - Schritthaltende Scans über sortierte Relationen R und S mit Durchführung des Verbunds bei r.VA = s.VA (d.h. auch Gleichverbund)
  - Komplexität: O(N log N)
- **Ausnutzen von Indexstrukturen $I_R$(VA) und $I_S$(VA) mit Sortierordnung:**

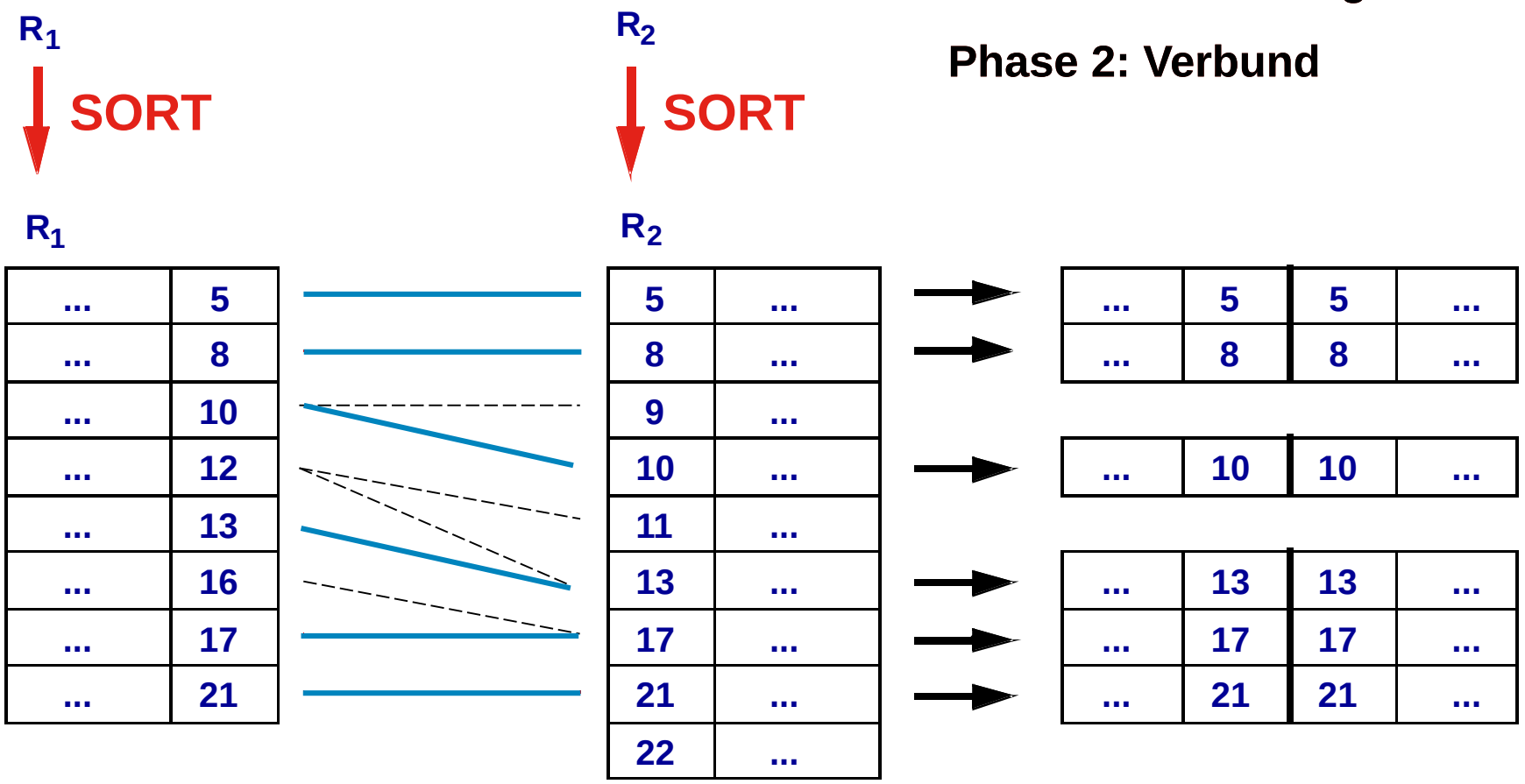
```
Schritthaltende Scans über I_R(R.VA) und I_S(S.VA);
für jeweils zwei Schlüssel aus I_R(R.VA) und I_S(S.VA),
für die r.VA = s.VA gilt:
   hol mit den zugehörigen TIDs die Tupel;
   für jedes Paar (r, s), für das P(r.SA) und P(s.SA) gelten:
       übernimm kombinierten Satz (r, s) in das Ergebnis;
```

# Beispiel zum Sort-Merge-Verbund



**Phase 1: Sortierung**

**Phase 2: Verbund**



# Hash-Verbund



- **Immer größere Hauptspeicher verfügbar**
  - Ausnutzen für Zwischenergebnisse
- **Gleichverbund**
  - Sehr häufiger Fall
- **Idee:**
  - Tupel der einen Relation so im Hauptspeicher ablegen,
    dass sie über Verbundattribut schnell gefunden werden können
  - Tupel der anderen Relation sequenziell durchlaufen
    und mit Wert des Verbundattributs
    die passenden Verbundpartner im Hauptspeicher aufsuchen
- **Organisation der Tupel im Hauptspeicher?**
  - Naheliegend: über Hashing

# Hash-Verbund (2)



- **Einfachster Fall ("Classic Hashing")**
  - Äußere Schleife:
    - Abschnittweises Lesen der (kleineren) Relation **R**
    - 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.SA) erfüllt.
    - Aufbau einer Hash-Tabelle mit $\mathbf{h_A}$(r.VA) nach Werten von R(VA)
  - Innere Schleife (für jeden Abschnitt $R_i$):
    - Überprüfung ("Probing") für jeden Satz von **S**, der P(S.SA) erfüllt:
      - Ebenfalls Hashing $h_A$(s.VA)
        - Verbundpartner (falls vorhanden) muss sich an dieser Adresse befinden
    - Im Erfolgsfall Durchführung des Verbunds
- **Merke:**
  - Komplexität: $O(p \times N)$
  - Idealfall: R passt ganz in den Hauptspeicher, d.h. $p = 1$

# Hash-Verbund (3)



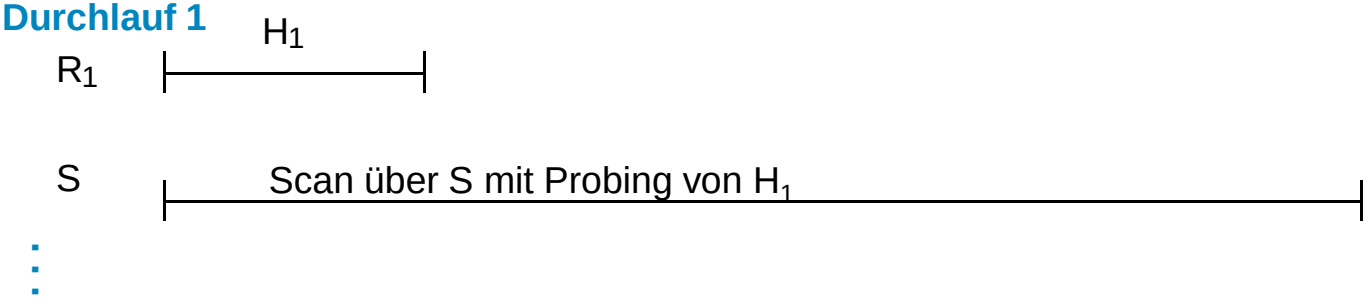
- **Pseudocode:**

```
Berechne Anzahl p der Abschnitte der kleineren Relation R,
so dass jeder Abschnitt in den Hauptspeicher passt;
Scan über R;
für jeden Abschnitt R_i, 1 <= i <= p:
    für jeden Satz r, der P(r.SA) erfüllt:
            Hash über r.VA, Ablage im Hauptspeicher;
    Scan über S;
    für jeden Satz s, der P(s.SA) erfüllt:
            Hash über s.VA, "Probing" im Hauptspeicher;
            bei Erfolg (Satz r mit r.VA = s.VA):
                    übernimm kombinierten Satz (r, s)
                    in das Ergebnis;
    Hauptspeicherinhalt (Hash-Tabelle) leeren;
```

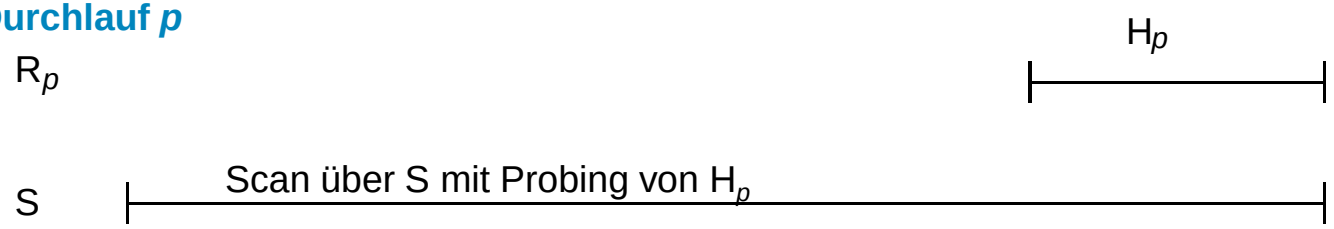
# Aufbau der Hash-Tabelle

- **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 Hash-Tabelle wieder gelöscht.

**Durchlauf 1**




**Durchlauf *p***

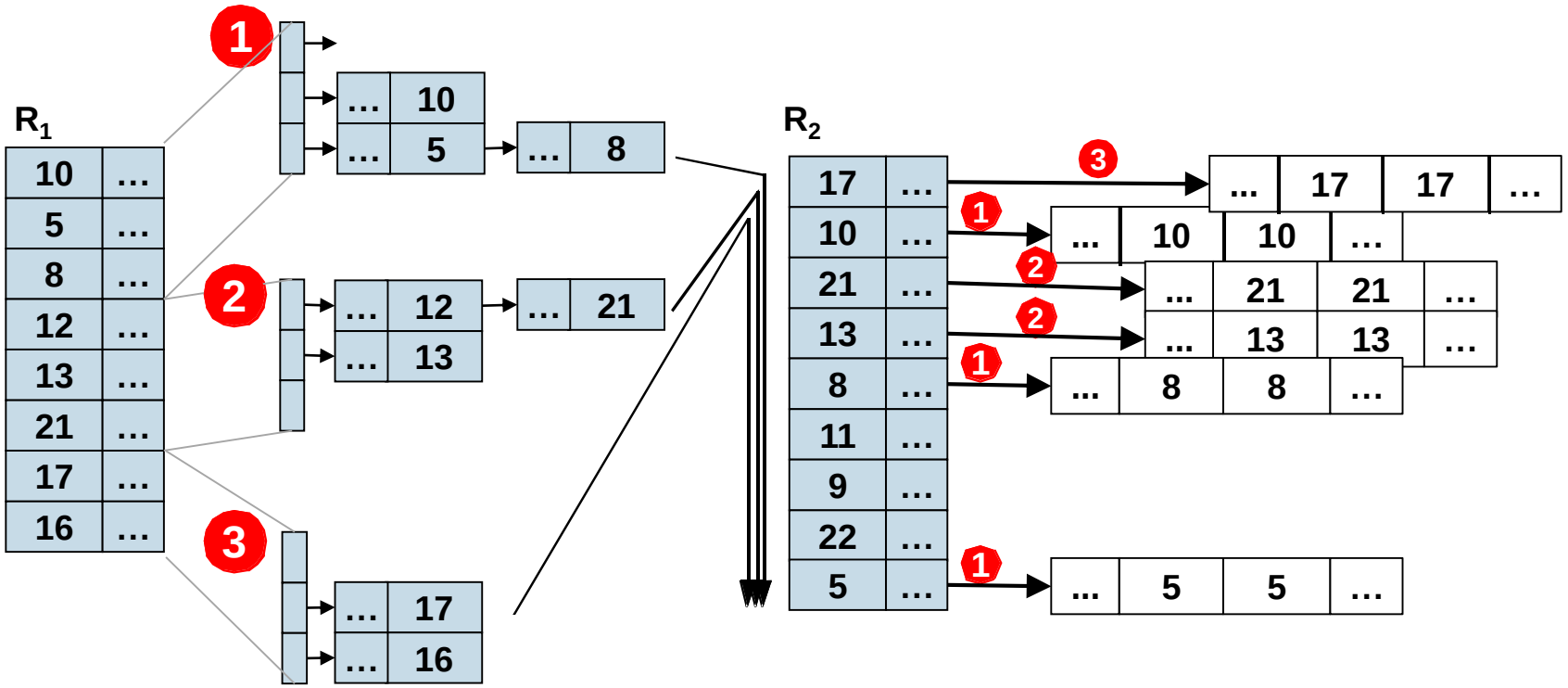


# Beispiel zum Hash-Verbund



**Annahme: Hauptspeicherkapazität = 3 Tupel**

**Hashing von $R_1$ mit h(x) = x mod 3**



# Verbesserungen am Verfahren



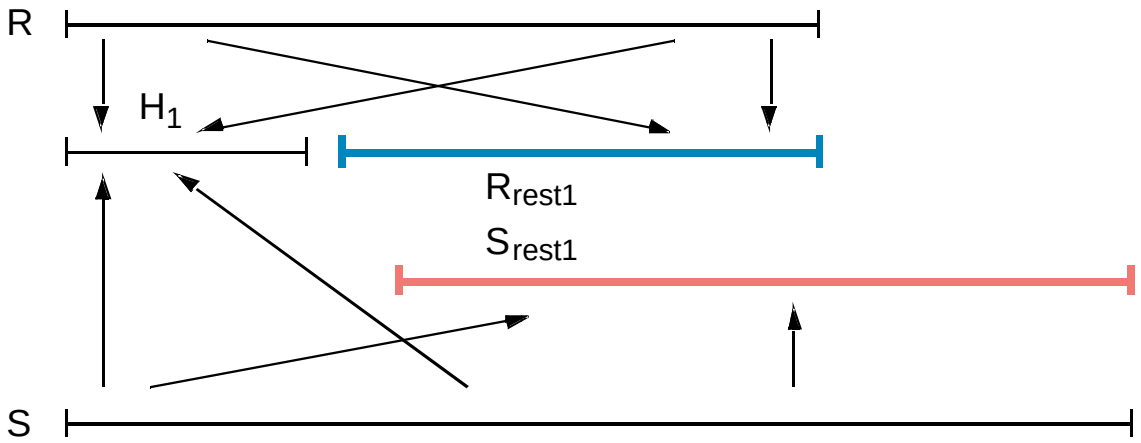
- **Nachteil des Hashing:**
  - Verbundpartner S muss *p*-mal gelesen werden.
- **Idee:**
  - Warum nicht auch S (analog zu R) partitionieren?
  - Hashing von R nicht nach der Reihenfolge der Tupel, sondern **wertemäßig** durchführen

    $\rightarrow$ Partitionierung von R nach Werten von R.VA
  - Aber:
    - Nicht einfach, da üblicherweise keine Gleichverteilung der Werte vorliegt
    - Heranziehen von Statistiken (insbes. Histogramme !)
  - Zum Probing auch S nach den gleichen Kriterien partitionieren
- **Vielzahl unterschiedlicher Hash-Verfahren**
  - "Simple Hashing"

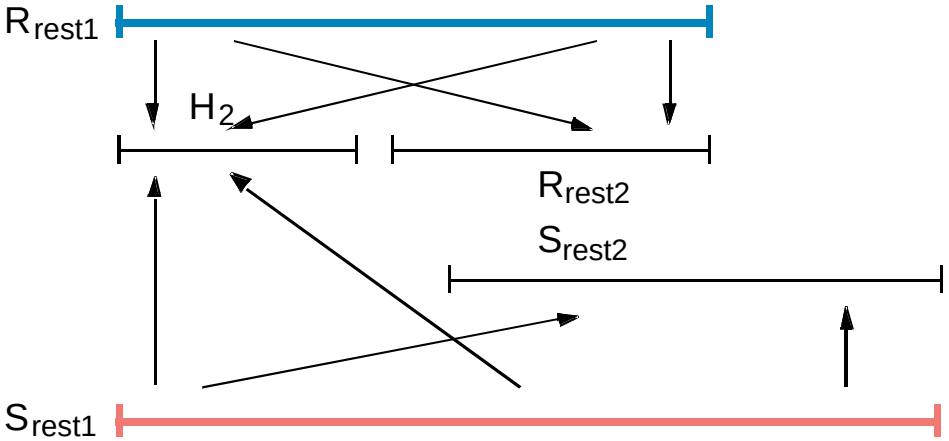
# Partitionierungsprinzip beim Simple-Hash-Verbund



**Durchlauf 1**

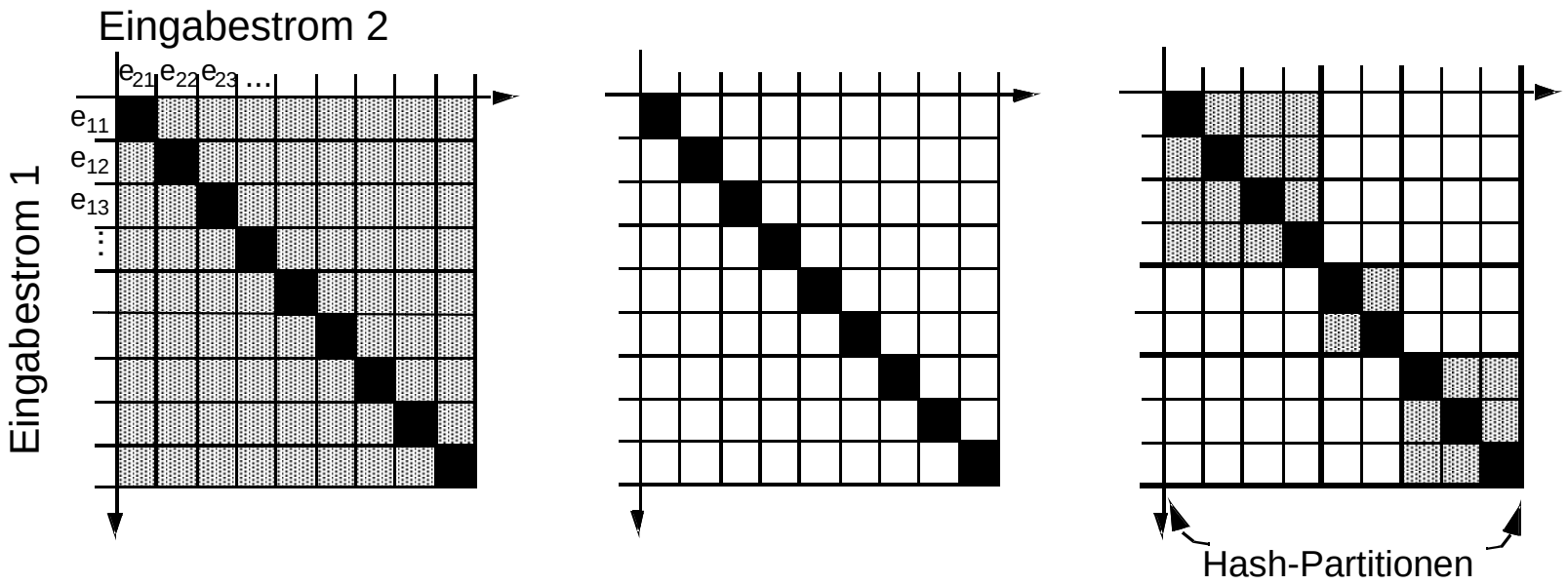



**Durchlauf 2**



# Vergleich der Verbundalgorithmen






**Schleifeniteration** **Mischverbund** **Hash-Verbund**

Elementvergleich

Elementvergleich, der zu einem Tupelverbund führt

# Weitere Planoperatoren



- **Duplikat-Eliminierung:**
  - Klassisch durch Sortierung
    und "Gruppierung" im anschließenden Scan
  - Möglich aber auch Hashing:
    - Duplikate werden zwangsläufig auf gleiche Adresse abgebildet.
- **Gruppierung:**
  - Klassisch durch Sortierung
    und Scan mit Aggregation pro Gruppe
  - Aber ebenfalls über Hashing möglich:
    - Abbildung auf Zähler, bisherige Summe, bisheriges Minimum oder Maximum
    - Für Durchschnitt Kombination von Zähler und Summe

# 11.2 Anfrageoptimierung

- **Ziel:**
  - Von der Anfrage (WAS?) zur Auswertung (WIE?)
  - Ermittlung des *kostengünstigsten* Auswertungswegs
- **Zentrales Problem:**
  - Globale Optimierung ist im allgemeinen zu aufwändig.
    - Fehlen von exakten statistischen Informationen
    - Sehr großer Suchraum
- **Deshalb Einsatz von Heuristiken**
- **Optimierungsziel**
  - Entweder Maximierung des Outputs bei gegebenen Ressourcen
    - Durchsatzmaximierung (in Transaktionen pro Sekunde)
  - Oder Minimierung der Ressourcennutzung für gegebenen Output
    - Antwortzeitminimierung für eine gegebene Anfragesprache, einen Mix von Anfragen verschiedenen Typs und eine gegebene Systemumgebung

# Kostenarten bei der Anfrageoptimierung



- **Berücksichtigung unterschiedlicher Kostenarten**
  - **Berechnungskosten**
    - CPU-Kosten
    - Pfadlängen (Anzahl Maschineninstruktionen)
  - **E/A-Kosten**
    - Anzahl der physischen Referenzen
  - **Speicherungskosten**
    - Temporäre Speicherbelegung im DB-Puffer und auf Externspeichern
  - Kommunikationskosten (bei verteilten DBS)
    - Anzahl der Nachrichten
    - Menge der zu übertragenden Daten
- **Merke:**
    - Kostenarten sind nicht unabhängig voneinander.
    - In DBVS oft gewichtete Funktion von Berechnungs- und E/A-Kosten

# Erstellung und Auswahl von Ausführungsplänen



- **Eingabe:**
  - Algebraisch optimierter Anfragebaum
  - Existierende Speicherungsstrukturen und Zugriffspfade
  - Kostenmodell
- **Ausgabe:**
  - Optimaler (eher: guter) Ausführungsplan
- **Zwei oft notwendige Annahmen:**
  - Alle Datenelemente und alle Attributwerte sind gleichverteilt.
  - Suchprädikate in Anfragen sind unabhängig.
- **Beide Annahmen sind (im Allgemeinen) falsch!**
  - Beispiel:
    (Gehalt ≥ 100.000) AND (Alter BETWEEN 20 AND 30)
    mit Gehalt: [10K .. 1M] und Alter: [20 .. 65]
    - Lineare Interpolation und Multiplikation von Wahrscheinlichkeiten eher nicht angemessen …

# Erstellung und Auswahl von Ausführungsplänen (2)



- **Vorgehensweise (vereinfacht):**
  - Generieren aller "vernünftigen" logischen Ausführungspläne zur Auswertung der vorliegenden Anfrage
  - Vervollständigen der Ausführungspläne mit Einzelheiten aus der physischen Datenrepräsentation (Sortierreihenfolge, Zugriffspfadmerkmale, statistische Information)
  - Bewertung der generierten Alternativen und Auswahl des billigsten Ausführungsplans gemäß dem vorgegebenen Kostenmodell
- **Merke:**
  - Wie entstehen alternative Ausführungspläne ???
    - Planoperatoren liegen in verschiedenen Implementierungen vor.
    - Operationsreihenfolgen (z.B. bei Mehrfachverbunden) können variiert werden.
  - Entstehung sehr großer Suchräume bei komplexen Anfragen mit allen Alternativen
    - Beispiel: ca. $10^{23}$ mögliche Ausführungspläne bei einer Anfrage mit 15 Verbunden
      - 15! (Reihenfolge) * $3^{15}$ (versch. Planoperatoren) * $2^{15}$ (Index- o. Table-Scan)

# Suchstrategien für einen guten Ausführungsplan

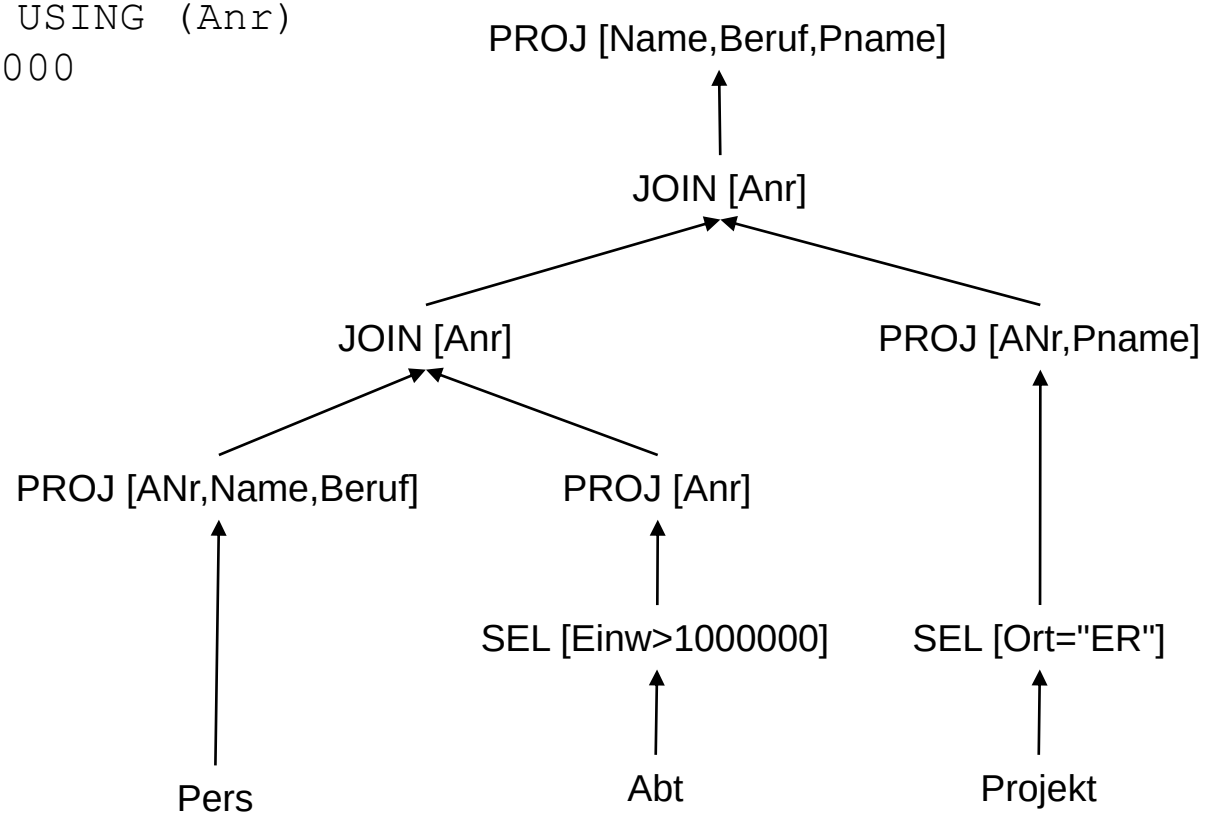


- **Ziel der Plangenerierung**
  - Auffinden eines guten Plans
    - Gelingt immer und geht schnell
  - Mit einer möglichst kleinen Anzahl generierter Pläne auskommen
- **Unterschiedliche Strategieklassen:**
  - voll-enumerativ
  - beschränkt-enumerativ
  - zufallsgesteuert
    - genetische Algorithmen und die Strategien des "simulated annealing"

# Beispiel zur Auswahl eines Ausführungsplans



```
SELECT P.Name, P.Beruf, J.Pname
FROM (Pers P JOIN Abt A USING (Anr))
     JOIN Projekt J USING (Anr)
WHERE A.Einw > 1000000
AND J.Ort = "ER";
```



# Beispiel (2)

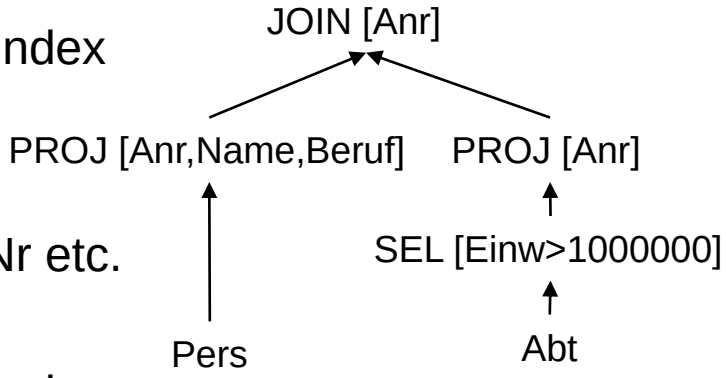


- **Mögliche Zugriffspfade für die einzelnen Relationen**
  - Relation **Pers**:
    - Index $I_P$(ANr) – nicht eindeutig, für den Join mit Abt und Proj
    - Scan(P)
  - Relation **Abt**:
    - Index $I_A$(ANr) – eindeutig, für den Join mit Pers und Proj
    - Index $I_A$(Einw) – für die Selektion
    - Scan(A)
  - Relation **Proj**:
    - Index $I_J$(ANr) – nicht eindeutig, für den Join mit Pers und Abt
    - Index $I_J$(Ort) – für die Selektion
    - Scan(J)

# Beispiel (3)



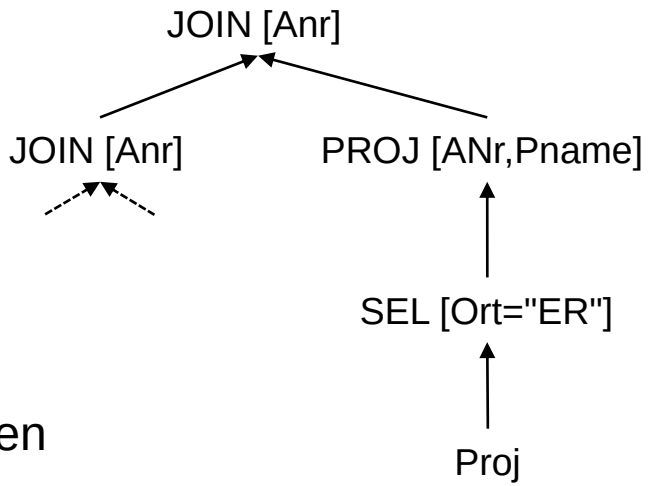
- **Nested-Loop-Join von Pers und Abt**
  - Mit **Abt** anfangen
    - $I_A$(Einw) benutzen für Selektion
      - Bei jedem Treffer gleich Projektion auf ANr
    - $I_P$(ANr) benutzen für Join-Partner
      - Direktzugriff über ANr, kann jeder Index
      - Evtl. mehrere Join-Partner
  - Mit **Pers** anfangen
    - Scan(P) benutzen mit Projektion auf ANr etc.
    - $I_A$(ANr) benutzen für Join-Partner
      - Direktzugriff über ANr, kann jeder Index
      - Immer nur ein Join-Partner
      - Selektionsbedingung dann noch prüfen



# Beispiel (4)



- **Zwischenergebnisse haben keinen Index.**
  - Nur Scan möglich
- **Nested-Loop-Join von (Pers, Abt) mit Proj**
  - Scan(P,A) benutzen
  - $I_J$(ANr) benutzen für Join-Partner
    - Direktzugriff über ANr, kann jeder Index
    - Mehrere Join-Partner
    - Selektionsbedingung prüfen
  - Alternativ $I_J$(Ort) benutzen – hohe Selektivität?
    - Dann das (kleine) Ergebnis
      für jedes Tupel von Scan(P,A)
      sequenziell nach Join-Partnern durchsuchen



# Beispiel (5)



- **Weitere Möglichkeiten:**
  - Falls $I_A$(ANr) und $I_P$(ANr) Sortierung nach Schlüssel bieten: Sort-Merge-Join
  - Erst Join (A,J), dann Join des Ergebnisses mit P
    - Also andere Join-Reihenfolge
  - …

# Berechnung der Zugriffskosten



- **Kostenvoranschlag**
  - für jeden Ausführungsplan (möglicher Lösungsweg)
- **Kostenformel**
  - Gewichtetes Maß für E/A- und CPU-Belastung:
    Kosten: #physische-Seitenzugriffe + W × #Aufrufe-des-Zugriffssystems
    - **W** ist das Verhältnis des Aufwands für einen Aufruf des Zugriffssystems (interne Satzschnittstelle) zum Aufwand für einen Seitenzugriff
  - Ziel der Gewichtung: Minimierung der Kosten abhängig vom Systemtyp
    - **CPU-bound**: höherer E/A-, geringerer CPU-Aufwand
      - Beispiel für W:
        $W_{CPU}$ = #Instr-pro-Aufruf-des-Zugriffssystems / #Instr-pro-E/A-Vorgang
    - **I/O-bound**: geringerer E/A-, höherer CPU-Aufwand
      - Beispiel für W:
        $W_{IO}$ = #Instr-pro-Aufruf-des-Zugriffssystems / (#Instr-pro-E/A-Vorgang + Zugriffszeit × MIPS-Rate)

# Ermittlung statistischer Kenngrößen



- **Statistische Größen für Segmente**
  - $M_S$ – Anzahl der Datenseiten des Segments S
  - $L_S$ – Anzahl der leeren Seiten in S
- **Statistische Größen für Relationen**
  - $N_R$ – Anzahl der Tupel in Relation R (auch card(R) oder | R | )
  - $T_{R,S}$ – Anzahl der Seiten in S mit Tupeln von R
  - $C_R$ – Cluster-Faktor (Anzahl der Tupel von R pro Seite)
- **Statistische Größen pro Index I auf Attributen A einer Relation R**
  - $j_I$ – Anzahl der Attributwerte (Schlüsselwerte) im Index (= card(PROJ(R,A))
  - $B_I$ – Anzahl der Blattseiten (beim B*-Baum)
- **Merke:**
  - Statistiken müssen gesammelt und im DB-Katalog gewartet werden.

# Ermittlung statistischer Kenngrößen (2)



- **Problem: Aktualisierung bei jeder Änderung sehr aufwändig**
  - Zusätzliche Schreib- und Log-Operationen
  - DB-Katalog wird zum Sperr-Engpass
- **Alternative:**
  - Initialisierung der statistischen Werte zum Lade- oder Generierungszeitpunkt von Relationen und Indexstrukturen
  - **Periodische Neubestimmung** der Statistiken durch eigenes Kommando oder Dienstprogramm
    - DB2: `runstats on table` ...
    - Oracle: `analyse table` ...

# Selektivitätsabschätzungen

- **Ermittlung eines Selektivitätsfaktors SF (0 ≤ SF ≤ 1)**
  - Erwarteter Anteil an Tupeln, die ein Prädikat p erfüllen:
    SF(p) = card(SEL(R,p)) / card(R)
  - Beispiel-Heuristiken zur Abschätzung des Selektivitätsfaktors:

| *Prädikat* | *SF* | *wenn* |
|---|---|---|
| $A_i = a_i$ | $1 / j_{Ii}$<br>1 / 10 | Index $I_i$ auf Attribut $A_i$<br>sonst |
| $A_i = A_k$ | $1 / MAX(j_{Ii}, j_{Ik})$<br>$1 / j_{Ii}$<br>$1 / j_{Ik}$<br>1 / 10 | Index auf beiden Attributen<br>Index nur auf $A_i$<br>Index nur auf $A_k$<br>sonst |
| $A_i > a_i$ | $(a_{max} – a_i) /$<br>$(a_{max} – a_{min})$<br>1 / 3 | Index auf $A_i$ und Werte interpolierbar<br><br>sonst |

(Großbuchstaben: Attributnamen; Kleinbuchstaben: Attributwerte)

# Selektivitätsabschätzungen (2)



- Beispiel-Heuristiken zur Abschätzung des Selektivitätsfaktors (Forts.):

| *Prädikat* | *SF* | *wenn* |
|---|---|---|
| $A_i$ BETWEEN $a_i$ AND $a_j$ | $(a_j - a_i) / (a_{max} - a_{min})$<br>1 / 4 | Index auf $A_i$ und Werte interpolierbar<br>sonst |
| $A_i$ IN $(a_1, a_2, \ldots, a_r)$ | $r / j_{l_i}$<br>1 / 2 | Index auf $A_i$ und SF < 0,5<br>sonst |

- **Berechnung von Verbundtermen im Qualifikationsterm**
  - SF(p(A) and p(B)) = SF(p(A)) · SF(p(B))
  - SF(p(A) or p(B)) = SF(p(A)) + SF(p(B)) – SF(p(A)) · SF(p(B))
  - SF(not p(A)) = 1 – SF(p(A))
- **Beachte:**
  - Wieder die Annahme der Unabhängigkeit von Prädikaten!

# Beispiel zur Selektivitätsabschätzung



- **Anfrage**

```
SELECT Name, Gehalt FROM Pers
WHERE Beruf = "Programmierer" AND Gehalt >= 100000;
```

- **Vorhandene Zugriffspfade**
  - Relationen-Scan im Segment von Pers
  - $I_{Pers}$(Beruf)
  - $I_{Pers}$(Gehalt)
- **Vorhandene statistische Kennwerte**
  - **N** = Anzahl der Tupel in Relation Pers (N = 5000)
  - $C_{Pers}$ = durchschn. Anzahl von Pers-Tupeln pro Seite
  - $j_i$ = Index-Kardinalität (Anzahl der Attributwerte für $A_i$)
    - Es gibt 25 unterschiedliche Berufe.
    - Die Gehälter liegen zwischen 30.000 und 120.000.

# Beispiel zur Selektivitätsabschätzung (2)



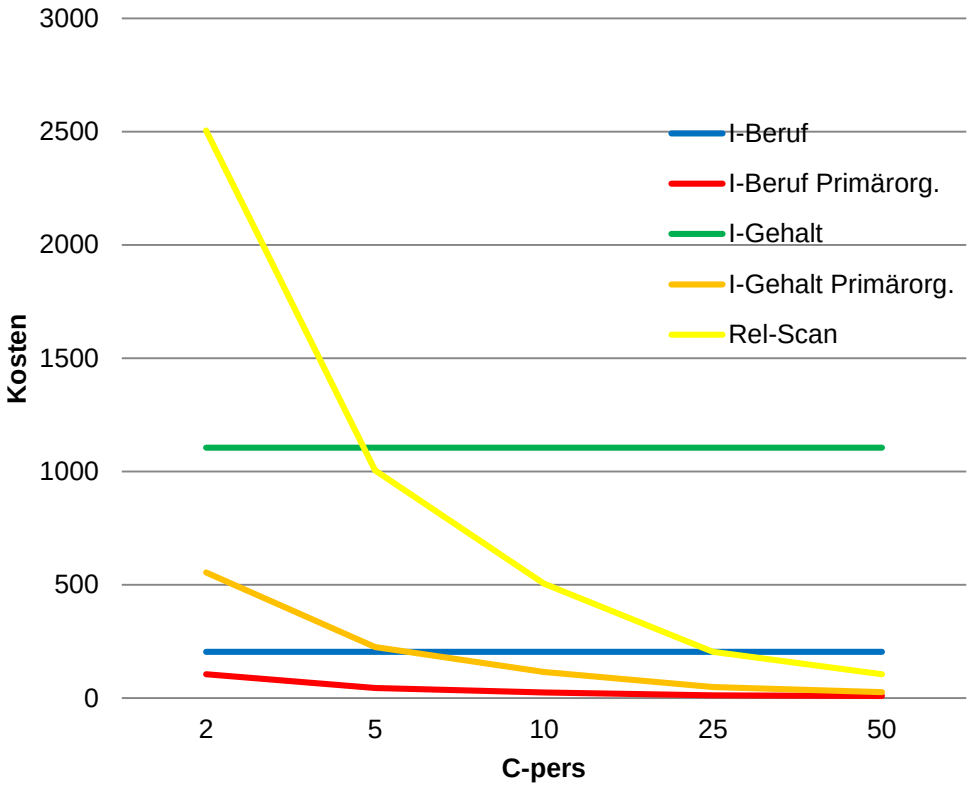
- $SF_B$ = SF(Beruf = "Programmierer") = 1 / $j_{Beruf}$ = 1 / 25
- $SF_G$ = SF(Gehalt >= 100.000) = (120.000 – 100.000) / (120.000 – 30.000)
  = 0,22

- **Methode 1: Scan über $I_{Pers}$(Beruf)**
  - Kosten: #physischer-Seitenzugriffe + W × #Aufrufe-des-Zugriffssystems
    $(N \times SF_B) + W \times (N \times (SF_B \times SF_G)) = N / 25 + W \times N \times 0{,}0088$
    - Index-Scan mit zusätzlichem einfachen Suchargument (Folie 11-5)
  - **Beachte**: Falls $I_{Pers}$(Beruf) Primärorganisation, reduziert sich die Anzahl der Seitenzugriffe um den Faktor $C_{Pers}$ auf $(N / C_{Pers}) \times SF_B$
- **Methode 2: Scan über $I_{Pers}$(Gehalt)**
  - Kosten: $(N \times SF_G) + W \times (N \times (SF_B \times SF_G)) = N \times 0{,}22 + W \times N \times 0{,}0088$
  - **Beachte**: Falls $I_{Pers}$(Gehalt) Primärorganisation, reduziert sich die Anzahl der phys. Seitenzugriffe um den Faktor $C_{Pers}$ auf $(N / C_{Pers}) \times SF_G$

# Beispiel zur Selektivitätsabschätzung (3)



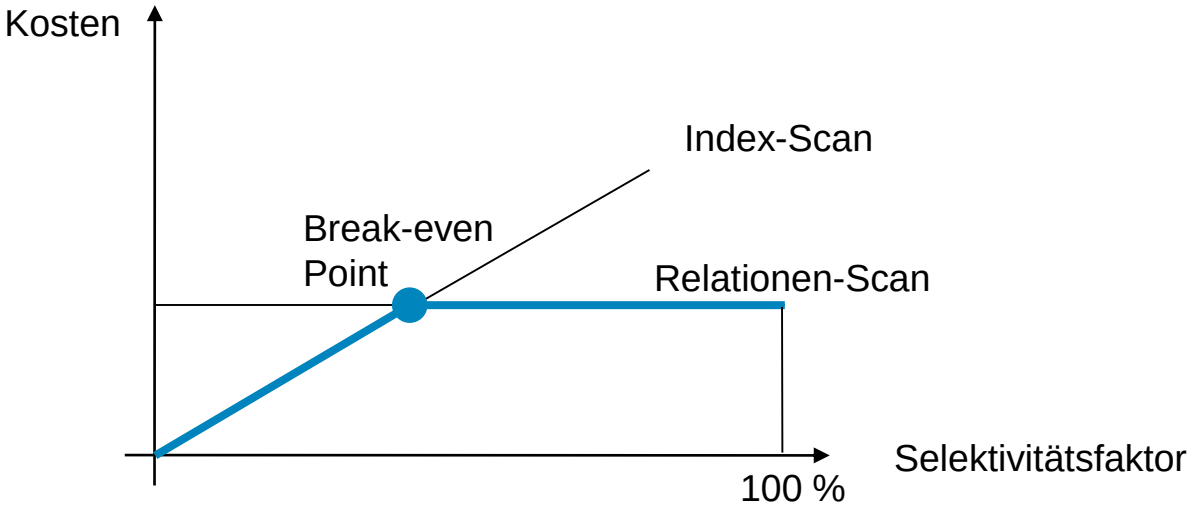
- **Methode 3: Relationen-Scan**
  - Kosten: $(N / C_{Pers}) + W \times (N \times (SF_B \times SF_G)) = (N / C_{Pers}) + W \times N \times 0{,}0088$
- **Kostenvergleich:**
  - W = 0,1
    N = 5000



# Betrachtung der Grenztrefferrate



- **Folgerung:**
  - Vorhandene Indexstrukturen können die Laufzeit dramatisch reduzieren!
  - Aber nur bei sehr geringen Trefferraten (= hoher Selektivität) lohnt sich ein Index-Scan.
  - Qualitatives Zugriffsdiagramm:

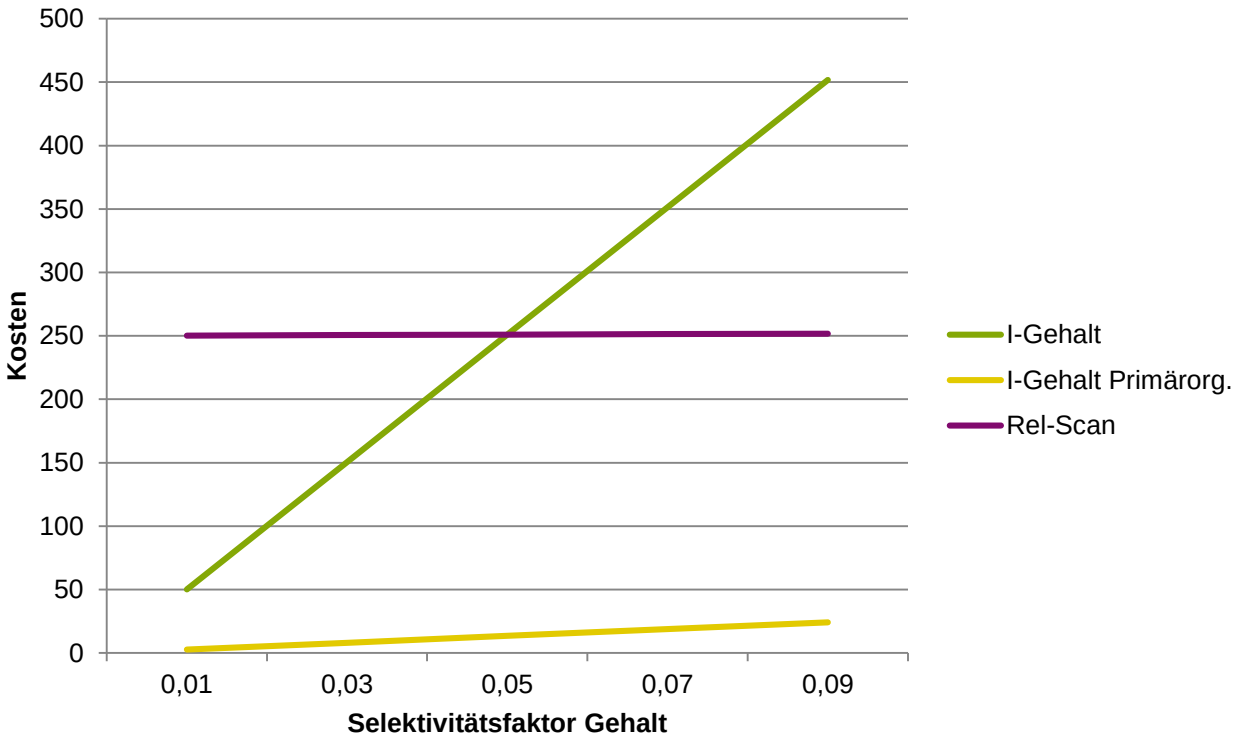



  - Grenztrefferrate liegt üblicherweise bei max. 5 %.

# Betrachtung der Grenztrefferrate (2)



- **Am Beispiel:**
  - $C_{pers}$ nun fest bei 20
  - $SF_G$ variiert



# 11.3 Zusammenfassung



- **Ziel der Anfrageoptimierung**
  - Ermittlung des kostengünstigsten Auswertungswegs
- **Standardisierung und Vereinfachung**
  - Umformungs-, Idempotenzregeln, …
- **Restrukturierung und Transformation**
  - Zuordnung von physischen Operatoren zu logischen
  - Reihenfolge der Operatorenausführung
- **Erstellung und Auswahl von Ausführungsplänen**
  - Sehr große Suchräume bei komplexen Anfragen
  - Strategieklassen: voll-enumerativ, beschränkt-enumerativ, zufallsgesteuert
  - Berechnung der Zugriffskosten:
    - Abschätzung von Selektivitäten
    - Nutzung vorhandener Zugriffspfade und statistischer Kennwerte