universität wien

Kapitel 11

Query Optimierung

Inhalt

11.1 Kosten-basierte Optimierung

Alternativen um eine gegebene Query abzuarbeiten

- **Äquivalente Ausdrücke**
- **Verschiedene Algorithmen** für jede Operation (siehe letztes Kapitel)

Unterschied zwischen Kosten einer guten und einer schlechten Auswertungsstrategie für eine Query kann enorm sein

Beispiel: Auswertung von $r \times s$ mit anschließender Selektion mit $r.A = s.B$ ist deutlich langsamer als ein Join $r \bowtie_{r.A=s.B} s$ mit derselben Bedingung

Schätzungen für die Kosten einer Operation werden benötigt

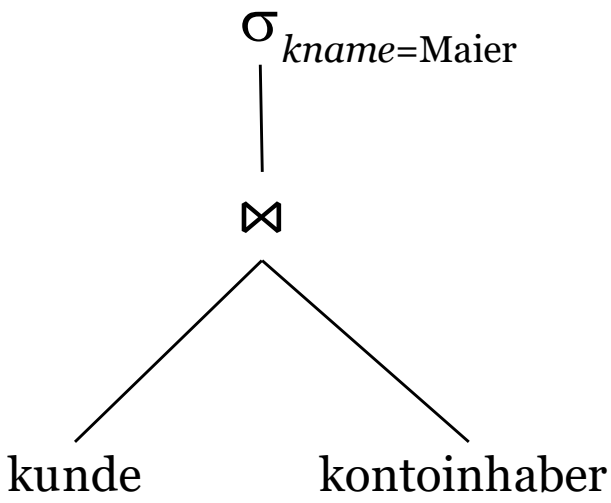
Stark abhängig von verfügbarer statistischer Information über die Relationen, die die Datenbank verwalten muss

z.B. Anzahl der Tupel, Anzahl verschiedener Werte für Join Attribute, etc.

Statistiken für Zwischenresultate müssen geschätzt werden, um die Kosten komplexer Operationen berechnen zu können

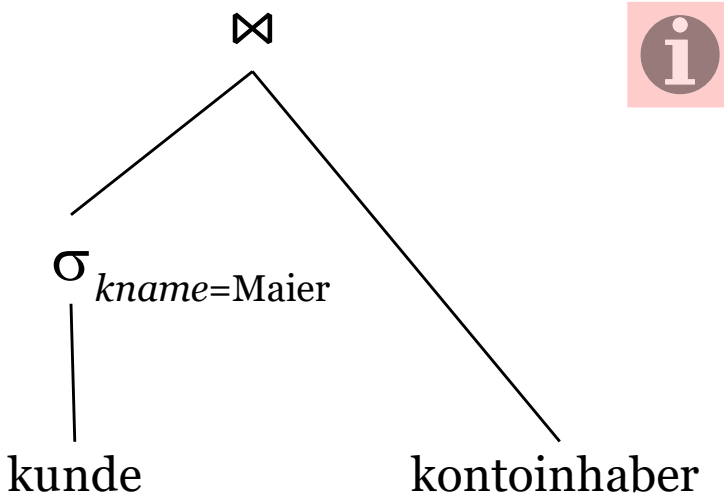
Äquivalente Ausdrücke

Relationen, die mittels **äquivalenter Ausdrücke** berechnet werden, haben dieselbe Menge von Attributen und enthalten die gleichen Tupel, die Attribute können aber unterschiedlich angeordnet sein.



Ursprünglicher Ausdruck

$\sigma_{kname=\text{Maier}}(\text{kunde} \bowtie \text{kontoinhaber})$



(Ein) äquivalenter Ausdruck

$(\sigma_{kname=\text{Maier}}\,\text{kunde}) \bowtie \text{kontoinhaber}$

Generierung Query Ausführungsplan

Generierung eines Query Ausführungsplans für einen Ausdruck erfolgt in mehreren Schritten:

1. Generierung logisch **äquivalenter Ausdrücke**

   Verwende **Äquivalenzumformungen** um einen Ausdruck in äquivalente Ausdrücke umzuformen.

2. Versehe resultierende Ausdrücke mit unterschiedlichen Annotationen (**alternativen Algorithmen**), um alternative Query Pläne zu erhalten
3. Wähle den **günstigsten Plan** basierend auf den geschätzten Kosten

Der gesamte Prozess heißt **Kosten-basierte Optimierung**.

Beispiel für Kosten-basierte Optimierung

Finde alle Konten des Kunden „Maier“

Phase 1: Äquivalente Ausdrücke

I:
$\sigma_{kname=\text{Maier}}$ (kunde ⋈ kontoinhaber)

II:
kunde ⋈ ($\sigma_{kname=\text{Maier}}$ kontoinhaber))

III:
($\sigma_{kname=\text{Maier}}$ kunde) ⋈ kontoinhaber

…
(weitere Möglichkeiten durch kommutative Vertauschungen)

Phase 2: Alternative Algorithmen

Join (⋈)
- Nested loop (a)
- Merge join (b)
- Hash join (c)
- …

Selektion (σ):
- Linear scan (1)
- Index scan (2)
- …

(Annahme: es existiert ein Index über *kname* in kunde)

Phase 3: Wahl günstigster Plan

Berechne Kosten dieser (simplen) Möglichkeiten:

I_{a1}, I_{b1}, I_{c1},
II_{a1}, II_{b1}, II_{c1},
III_{a1}, III_{a2}, III_{b1},
III_{b2}, III_{c1}, III_{c2},
…

11.2 Kostenabschätzung

Der DBMS Katalog speichert eine Reihe **statistischer Informationen** über die DB Relationen

Beispiele sind

- n_r: Anzahl der Tupel in einer Relation r.
- b_r: Anzahl der Blöcke, die Tupel aus r enthalten.
- s_r: Größe eines Tupels aus r.
- f_r: Blocking Faktor von r,

  d.h. die Anzahl der Tupel aus r, die in einen Block passen.

  Falls die Tupel aus r zusammen in einem physikalischen File gespeichert werden, gilt: $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$

- V(A,r): Anzahl verschiedener Werte, die in r für das Attribut A auftreten; gleich der Mächtigkeit von $\prod_A(r)$.

Verzeichnisinformationen über Indizes

F_i: durchschnittlicher fan-out eines internen Knoten des Index *i*,

für baumartige Indizes, wie B+-Bäume.

HT_i: Anzahl der Ebenen im Index *i,* also Höhe von *i.*

Für einen balancierten Baum (wie B+-Baum) Index über das Attribut *A* der Relation *r*, $HT_i = \lceil \log_{Fi}(V(A,r)) \rceil$.

Für einen Hashindex ist HT_i gleich 1.

LB_i: Anzahl der Index Blöcke in *i auf der niedrigsten Baumebene*

d.h. die Anzahl der Blöcke in der Blatt-Ebene des Index.

Maße für Query Kosten

Wir nehmen an, dass

- typischerweise die **Plattenzugriffe**, die relativ leicht abgeschätzt werden können, die **Kosten dominieren**
- die **Anzahl der Block-Transfers von der Disk** als Maß für die tatsächlichen Kosten der Abarbeitung verwendet werden.
- angenommen wird, dass alle **Block-Transfers** die **gleichen Kosten** verursachen.
  - In der Praxis eingesetzte Optimierer machen diese Annahme nicht und unterscheiden zwischen sequentiellem und wahlfreiem Zugriff auf die Disk

Wir berücksichtigen nicht die Kosten um die finalen Ergebnisse auf die Disk zu schreiben.

Wir bezeichnen die Kostenabschätzung für Algorithmus *A* mit E_A

Abschätzung der Datengrößen von Ergebnissen

Die Kostenabschätzung von DB Operationen basiert in den meisten Fällen auf einer **Schätzung der Größe der (Teil-) Ergebnisse von Einzeloperationen**, wie

- Selektion, Projektion, Join, Vereinigung, Durchschnitt, etc.

Ansätze meist simpel, basieren auf statistischen Kennzahlen, wie

- Größe der Operandenrelationen (Anzahl Tupel und Blöcke),
- Selektivitäten für Werte,
- Verteilungsannahmen
- …

und Berücksichtigung von Schema-Eigenschaften, wie

- Schlüssel,
- Fremdschlüssel,
- Attributüberlappungen,
- …

Motivationsbeispiel: Kostenberechnung

Kataloginformation für Join Beispiele:

n_{kunde} = 10000

V(kname, kunde) = 10000 (*kname* ist Schlüssel in *kunde*)

f_{kunde} = 25, daraus folgt, dass

b_{kunde} = 10000/25 = 400

$F_{kname(kunde)}$ = 100, d.h. $HT_{kname(kunde)} = \log_{100} 10000 = 2$

$n_{kontoinhaber}$ = 5000

$f_{kontoinhaber}$ = 50, was bedeutet, dass

$b_{kontoinhaber}$ = 5000/50 = 100

$E_{nested\ loop} = n_r * b_s + b_r$

Zwei Möglichkeiten (aus obigem Beispiel)

E_{IIa1} ($\sigma_{kname=Maier}$ (kunde $\bowtie$ kontoinhaber), nested loop, sequential) =
10000 * 100 + 400 = ***1000400***

Annahme: Pipelining

E_{IIIa2} (($\sigma_{kname=Maier}$ kunde) $\bowtie$ kontoinhaber , nested loop, index scan) =
2 + 1 * 100 + 1 = ***103*** (nested loop, d.h. Indexzugriff + $E_{nested\ loop}$)

11.3 Transformation Relationaler Ausdrücke

Zwei Ausdrücke in relationaler Algebra werden als **äquivalent** bezeichnet, falls beide Ausdrücke in jeder gültigen Datenbankinstanz die gleiche Menge von Tupeln generieren

- Achtung: Die Anordnung der Tupel ist irrelevant

In SQL sind Inputs und Outputs Multimengen aus Tupeln

- Multimengen sind Mengen, in denen Elemente mehrfach auftreten können (eng. „bag“)
  - Die Reihenfolge wird ignoriert, aber die Multiplizität von Elementen ist signifikant, z.B. Multimengen {1,2,3} und {2,1,3} sind äquivalent, aber {1,1,2,3} und {1,2,3} unterscheiden sich
- Zwei Ausdrücke werden in der Multimengen-Version der relationalen Algebra als äquivalent bezeichnet, falls beide Ausdrücke in jeder gültigen Datenbankinstanz die gleiche Multimenge von Tupeln generieren

Eine **Äquivalenzumformung** gibt zwei äquivalente Formen eines Ausdrucks an

- die erste Form kann durch die zweite ersetzt werden und vice versa

Äquivalenzumformungen

1. Konjunktive Selektion kann zerlegt werden in eine Abfolge einzelner Selektionen.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selektionsoperationen sind kommutativ.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. In einer Sequenz von Projektionen ist nur die letzte notwendig, die anderen können vernachlässigt werden.

$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(E))\ldots)) = \Pi_{t_1}(E)$$

4. Selektionen können kombiniert werden mit kartesischen Produkten und Theta Joins.

$$\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$$
$$\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$$

Äquivalenzumformungen (2)

5. Theta Join Operationen (und natural Joins) sind kommutativ.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural Join Operationen sind assoziativ:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta Joins sind folgendermaßen assoziativ:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

wobei θ_2 nur Attribute aus E_2 und E_3 enthält

jede der Bedingungen kann leer sein

Äquivalenzumformungen (3)

7. Die Selektionsoperation ist distributiv mit einem Theta Join unter den beiden folgenden Bedingungen:

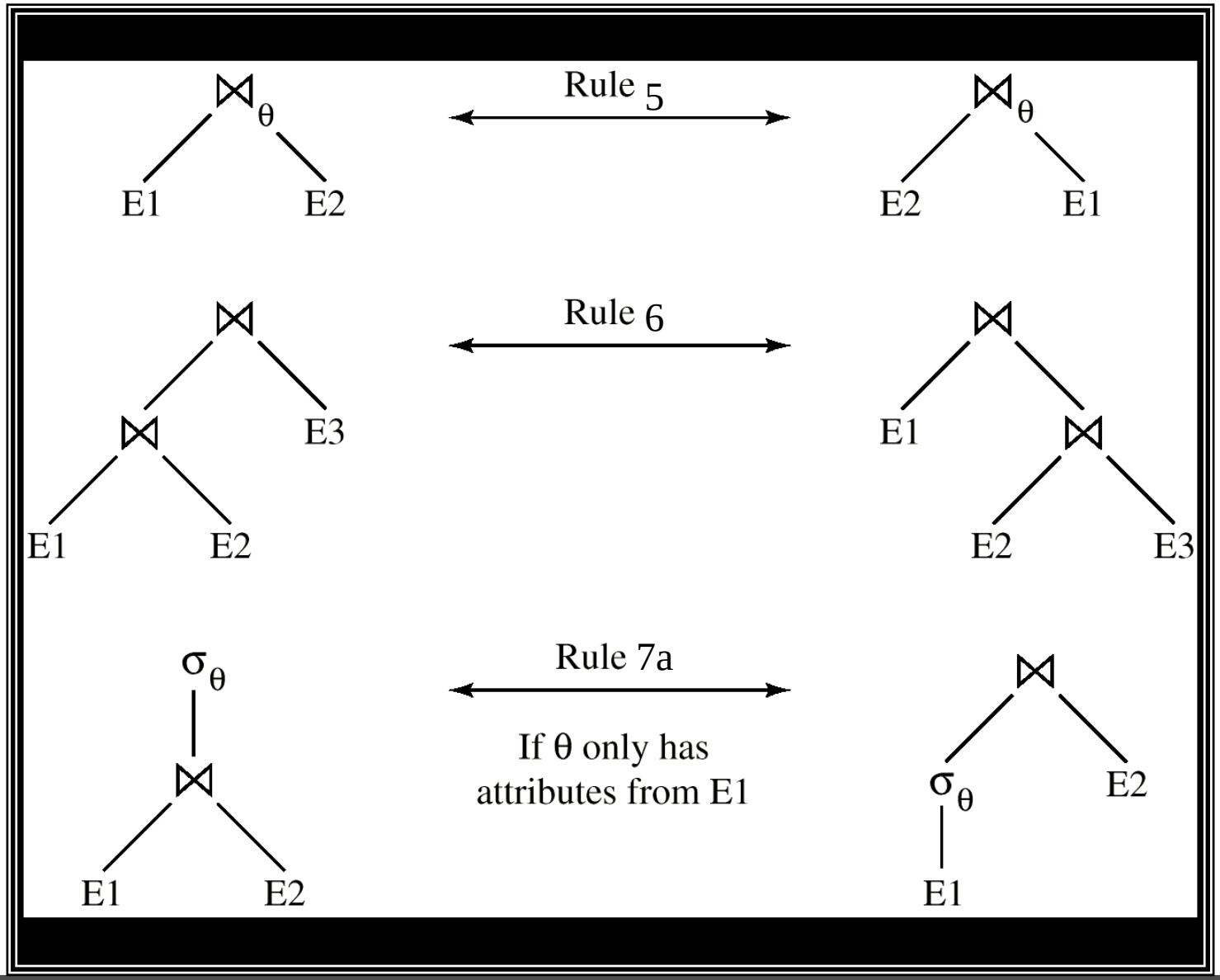
(a) Wenn alle Attribute in θ_0 nur aus einem der Ausdrücke (E_i) stammen, die mittels Join verbunden werden.

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

(b) Wenn θ_1 nur Attribute aus E_1 und θ_2 nur Attribute aus E_2 enthält.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

Grafische Darstellung der Äquivalenzumformungen



Äquivalenzumformungen (5)

9. Die Mengenoperationen Vereinigung und Schnitt sind kommutativ

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

(Mengendifferenz ist nicht kommutativ).

10. Mengenvereinigung und -schnitt sind assoziativ.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. Die Selektion ist distributiv bezüglich $\cup$, $\cap$ und –.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

und analog für $\cup$ und $\cap$ statt –

Auch: $$\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

und analog für $\cap$ statt –, aber nicht für $\cup$

12. Die Projektion ist distributiv bezüglich Vereinigung

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Beispiel zur Transformation

Query: Finde die Namen aller Kunden, die ein Konto in einer Filiale in Brooklyn haben.

$$\Pi_{kname}(\sigma_{stadt = \text{“Brooklyn”}}(filiale \bowtie (konto \bowtie kontoinhaber)))$$

Transformation nach Regel 7a:

$$\Pi_{kname}((\sigma_{stadt = \text{“Brooklyn”}}(filiale)) \bowtie (konto \bowtie kontoinhaber))$$

Ausführen der Selektion so früh wie möglich reduziert die Größe der Relationen, die mittels Join verbunden werden müssen.

Beispiel mit mehreren Transformationen

Query: Finde die Namen aller Kunden mit einem Konto in einer Filiale in Brooklyn, deren Kontostand mehr als $1000 beträgt.

$$\Pi_{kname}(\sigma_{stadt = \text{“Brooklyn”} \wedge kontostand > 1000}(filiale \bowtie (konto \bowtie kontoinhaber)))$$

Transformation mittels Assoziativität von Join (Regel 6a):

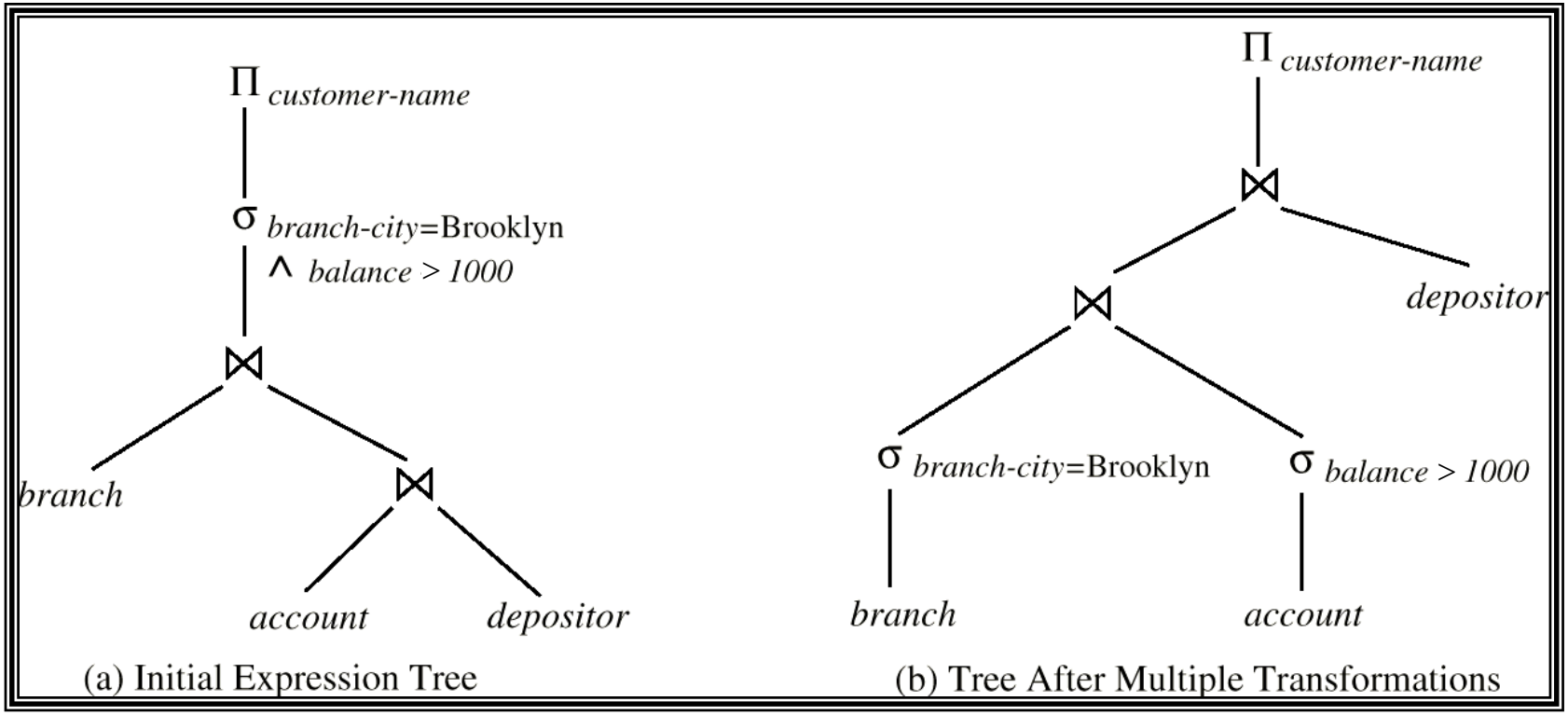
$$\Pi_{kname}(\sigma_{stadt = \text{“Brooklyn”} \wedge kontostand > 1000}((filiale \bowtie konto) \bowtie kontoinhaber))$$

Zweite Form bietet Möglichkeit für die Anwendung der “führe Selektionen früh aus” Regel (Regel 7a und 7b), und ergibt den Teilausdruck

$$\sigma_{stadt = \text{“Brooklyn”}}(filiale) \bowtie \sigma_{kontostand > 1000}(konto)$$

Eine Sequenz von Transformationen kann also sinnvoll sein

Beispiel mit mehreren Transformationen (Fortsetzung)



Beispiel für Reihenfolge von Joins

Für alle Relationen r_1, r_2, und r_3 gilt:

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

falls $r_2 \bowtie r_3$ ziemlich groß und $r_1 \bowtie r_2$ klein ist, wählen wir

$$(r_1 \bowtie r_2) \bowtie r_3$$

so wird eine kleinere temporäre Relation berechnet und gespeichert.

Beispiel für Reihenfolge von Joins (2)

Wir betrachten den Ausdruck

$\Pi_{kname}\ ((\sigma_{stadt = \text{“Brooklyn”}}\ (filiale)) \bowtie konto \bowtie kontoinhaber)$

Wir könnten $konto \bowtie kontoinhaber$ zuerst berechnen und das Ergebnis mittels Join mit

$$\sigma_{stadt = \text{“Brooklyn”}}\ (filiale)$$

verbinden, aber $konto \bowtie kontoinhaber$ ist wahrscheinlich eine große Relation.

Da es wahrscheinlicher ist, dass nur ein kleiner Teil der Bankkunden Konten in Brooklyn hat, ist es besser zuerst

$$\sigma_{stadt = \text{“Brooklyn”}}\ (filiale) \bowtie konto$$

zu berechnen.

Enumeration Äquivalenter Ausdrücke

Query Optimizer **verwenden Äquivalenzumformungen systematisch** um für einen gegebenen Ausdruck äquivalente Ausdrücke zu finden

Konzeptuell können **alle äquivalenten Ausdrücke** durch Wiederholen des folgenden Schritts gefunden werden, bis keine neuen Ausdrücke mehr generiert werden:

„Für jeden bisher gefundenen Ausdruck, verwende alle anwendbaren Äquivalenzumformungen und füge neu generierte Ausdrücke zur Menge der bisher gefundenen Ausdrücke hinzu“

Dieser Ansatz braucht **viel Zeit und Speicherplatz**

Memorybedarf kann durch gemeinsame Nutzung von Teilausdrücken verringert werden:

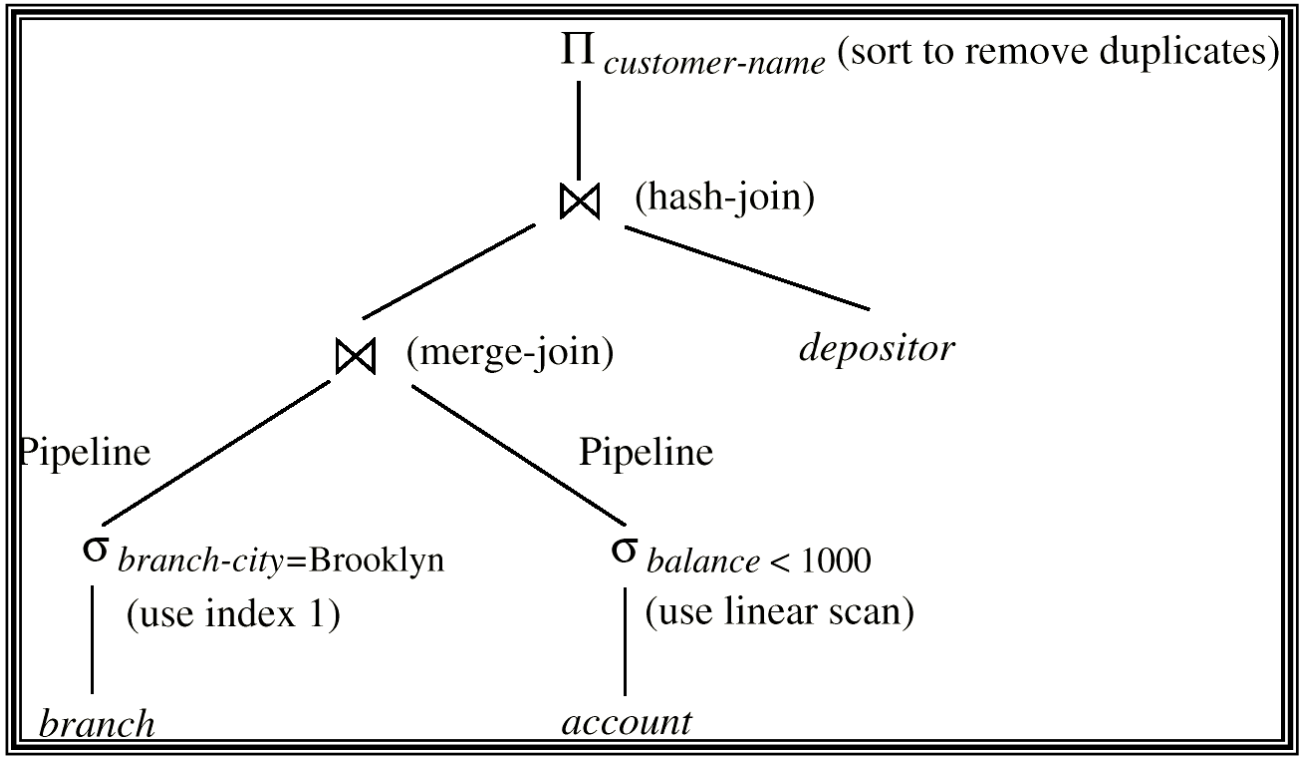
wenn E1 aus E2 durch Äquivalenzumformung entsteht, sind die beiden gewöhnlich auf der Wurzelebene unterschiedlich, darunter liegende Unterbäume sind gleich und können gemeinsam verwendet werden

z.B. Anwendung der Assoziativität bei Join

Zeitbedarf verringern indem **nicht alle äquivalenten Ausdrücke** generiert werden.

11.4 Ausführungsplan

Ein **Ausführungsplan** (**Evaluation Plan**) legt genau fest, welcher Algorithmus für jede Operation verwendet wird und koordiniert die Operationsausführung



Wahl eines Ausführungsplans

Gegenseitige Einflüsse der Auswertungstechniken müssen bei der Auswahl des Ausführungsplans berücksichtigt werden:

Die unabhängige **Wahl des billigsten Algorithmus** für jede einzelne Operation **muss nicht** den **besten Gesamtalgorithmus** bilden, z.B.

- merge-join kann teurer sein als hash-join, aber stellt unter Umständen sortierten Output zur Verfügung, was die Kosten für eine folgende Aggregation vermindern kann.
- nested-loop join bietet unter Umständen die Möglichkeit zum Pipelining

Existierende Query Optimizer verwenden Elemente der beiden folgenden Herangehensweisen:

1. **Durchsuche alle Pläne** und wähle den besten Plan basierend auf geschätzten Kosten.
2. Verwende **Heuristiken** zur Auswahl eines Plans.

Kosten-basierte Optimierung

Auffinden der besten Reihenfolge für $r_1 \bowtie r_2 \bowtie \ldots r_n$.

- Für n = 3 gibt es 12 verschiedene Reihenfolgen:
- $r_1 \bowtie (r_2 \bowtie r_3)$, $r_1 \bowtie (r_3 \bowtie r_2)$, $(r_2 \bowtie r_3) \bowtie r_1$, $(r_3 \bowtie r_2) \bowtie r_1$ …
- Generell gibt es $(2(n-1))!/(n-1)!$ verschiedene Reihenfolgen für diesen Ausdruck. Mit $n = 7$, sind das 665280, mit $n = 10$, mehr als 176 Milliarden!
- Führt zu einem NP Problem

Es brauchen aber nicht alle möglichen Reihenfolgen generiert werden.

Mittels **dynamischer Programmierung** wird die günstigste Join Reihenfolge für jede Teilmenge von $\{r_1, r_2, \ldots r_n\}$ nur einmal bestimmt und für die weitere Verwendung gespeichert.

Heuristische Optimierung

Kosten-basierte Optimierung ist auch mit dynamischer Programmierung teuer.

Systeme können Heuristiken verwenden, um die Anzahl der Auswahlmöglichkeiten für die kosten-basierte Optimierung zu verringern.

Heuristische Optimierung transformiert den Query-Baum durch Anwendung von Regeln, die typischerweise (aber nicht in allen Fällen) die Performanz der Ausführung verbessern:

- Führe **Selektion früh** aus (reduziert die Anzahl der Tupel)
- Führe **Projektion früh** aus (reduziert Anzahl der Attribute)
- Führe die **restriktivsten Selektionen und Joins vor anderen** ähnlichen Operationen aus.

Häufig Anwendung von **A*- Algorithmen**

z.B. in Verbindung mit Blackboard Methode aus der AI

Schritte bei einer typischen Heuristischen Optimierung

1. Zerlege konjunktive Selektionen in eine Sequenz einzelner Selektionen (Regel 1).
2. Schiebe Selektionen im Query-Baum nach unten, so dass sie so früh wie möglich ausgeführt werden (Regeln 2, 7a, 7b, 11).
3. Führe zuerst die Selektionen und Joins aus, die die kleinsten Relationen produzieren (Regel 6).
4. Ersetze kartesische Produkte, denen eine Auswahlbedingung folgt, durch Join (Regel 4a).
5. Zerlege Listen von Attributen für Projektionen und schiebe sie so weit wie möglich den Baum hinunter, wenn nötig erzeuge zusätzliche Projektionen (Regeln 3, 12).
6. Finde Unterbäume, deren Operationen in einer Pipeline ausgeführt werden können, und verwende Pipelining entsprechend.

Bemerkungen zu Query Optimierer

Einige Systeme verwenden nur Heuristiken, andere kombinieren Heuristik mit Generierung von alternativen Plänen auf Basis Kosten-basierter Optimierung.

Auch unter der Verwendung von Heuristiken, bedeutet Kosten-basierte Query Optimierung einen substantiellen Aufwand

- Siehe Ballooning-Ansatz

Optimierung zahlt sich aber in der Regel aus

- Problem der langsamen Plattenzugriffe

Eigenarten von SQL verkomplizieren die Query Optimierung

- z.B. verschachtelte Abfragen

Die meisten kommerziellen Systeme enthalten ausgereifte Optimierer (z.B. Oracle)

11.5 Benutzer-Tuning von Datenbankabfragen

Statistiken (Histogramme, etc.) müssen explizit angelegt werden

- Würden sonst die Update-Operationen zu stark belasten

Anderenfalls liefern die Kostenmodelle falsche Werte

In Oracle …

```
analyze table Professoren compute statistics for
  table;
```

- Man kann sich auch auf approximative Statistiken verlassen
  - Anstatt **`compute`** verwendet man **`estimate`**

In DB2 …

```
runstats on table …
```

Analysieren von Leistungsengpässen

In Oracle durch den **`explain plan`** Befehl

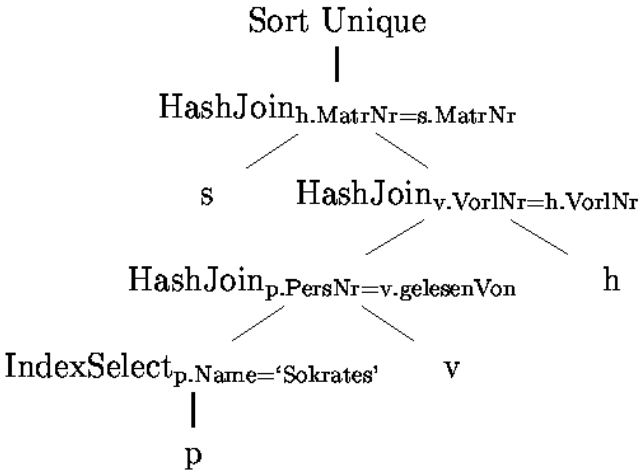
Beispiel (aus Kemper, Eickler)

```
explain plan for
   select distinct s.Semester
   from Studenten s, hören h, Vorlesungen v, Professoren p
   where p.Name = 'Sokrates' and v.gelesenVon = p.PersNr and
         v.VorlNr = h.VorlNr and h.MatrNr = s.MatrNr;
```

```
SELECT STATEMENT    Cost = 37710
  SORT UNIQUE
    HASH JOIN
      TABLE ACCESS FULL STUDENTEN
      HASH JOIN
        HASH JOIN
          TABLE ACCESS BY ROWID PROFESSOREN
            INDEX RANGE SCAN PROFNAMEINDEX
          TABLE ACCESS FULL VORLESUNGEN
        TABLE ACCESS FULL HOEREN
```

Geschätzte Kosten in Oracle

Entsp. Baumdarstellung



Query Tuning

Allgemeines Prinzip: Zuerst Tuning-Methoden versuchen, deren Effekt streng lokal ist

Umschreiben von (SQL)-Abfragen hat diese Eigenschaft

Hinweise für schlechte Abfrage-Performanz:

- zu viele Plattenzugriffe z.B. Relation wird (unnötigerweise) sequentiell durchlaufen (Scan)
- Abfrage-Plan:
  - z.B. Feststellen, ob Index verwendet wird
  - **`explain plan`**
- Trace

Tuning von Selects (1)

1. keine Operationen auf indiziertem Attribut

```
where preis*1.2 > 5000
```

besser:

```
where preis > 5000 / 1.2
```

noch besser:

```
where preis > 4166.66
```

2. keine Funktionen auf indiziertem Attribut

```
where substr(Name, 1, 4) = 'Hube'
```

häufig besser:

```
where Name LIKE 'Hube%'
```

mitunter noch besser:

```
where Name Between 'Hube' and 'Hubf'
```

Tuning von Selects (2)

3. Reihenfolge verändern

```
where geschlecht = 'M' and Note = 1
```

mitunter besser:

```
where  Note = 1 and geschlecht = 'M'
```

Annahme: es gibt mehr Männer als Personen mit Note 1

4. Abfragen auf ungleich vermeiden

```
where Geburtsland <> 'A'
```

mitunter besser:

```
where  Geburtsland > 'A' or Geburtsland < 'A'
```

5. Abfragen auf IS NOT NULL vermeiden

```
where ATP-Punkte IS NOT NULL
```

mitunter besser:

```
where  ATP-Punkte >= 0;
```

(Annahme: es gibt keine negativen Punkte)

Tuning von Selects (5)

6. Typkonversion

```
where integer_attribut > 17.4
```

meist besser:

```
where  integer_attribut  > 17
```

7. Codierung

eigentlich ein Designproblem - wirkt sich hier allerdings aus

```
where  func(attribut) = 17
```

Bsp:

```
where substr(Code, 4, 1) = 'M'
where Mod(trunc(code/100), 7) = 2;
```

sollte vermieden werden!

Tuning von Joins

Tuningziele

- Optimale Reihenfolge
- Joins vs Subqueries
  - meist besser, Query mit Join formulieren
- Join-Bedingung
  - es gilt alles, was für Selects gesagt wurde

Reihenfolge

- manche Query-Optimizer bilden Join nach vorgegebenen Reihenfolge

```
Select *
From   A, B, C
where  a.x = b.x and b.y = c.y
```

- daher ausprobieren:

```
... From      A, C, B
... From      C, B, A
```

Tuning von Joins (2)

Zusätzliche Joinbedingungen

a) zwischen 2 Relationen

```
where   a.empno = b.empno
  and   a.ssn = b.ssn
```

b) zwischen mehreren Relationen

```
where   a.empno = b.empno
  and   b.empno = c.empno
  and   a.empno = c.empno
```

c) zusätzliche Bedingungen um Index-Verwendung bzw. Reihenfolge zu beeinflussen:

```
where   a.empno >= 0
  and   a.empno = b.empno
```