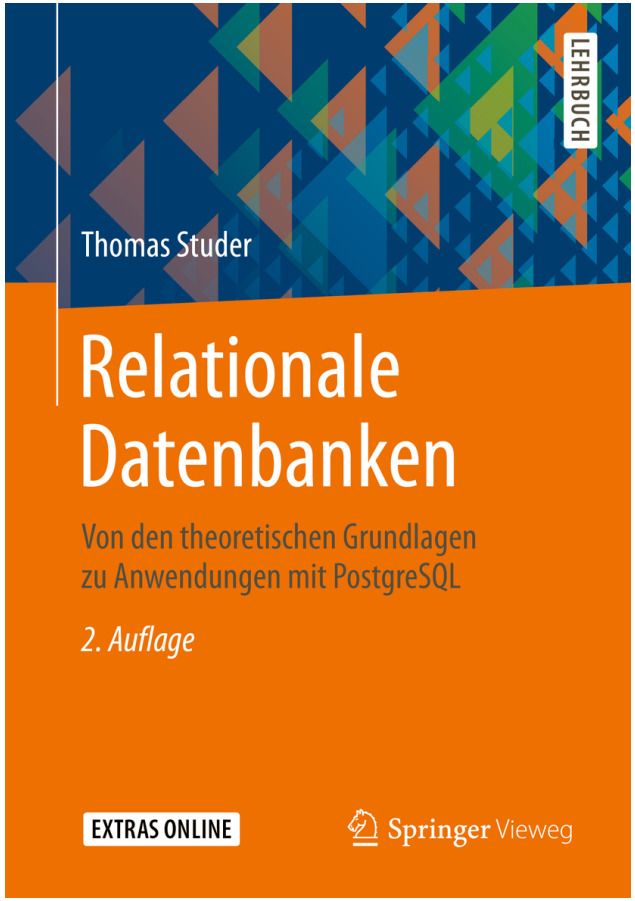


Thomas Studer
Relationale Datenbanken:
Von den theoretischen Grundlagen
zu Anwendungen mit PostgreSQL
2. Auflage
Springer Vieweg, 2019
ISBN 978-3-662-58975-5

Datenbanken

—

Anfrageoptimierung

Thomas Studer

Institut für Informatik
Universität Bern

Techniken

Indizes
Hilfsobjekte, welche die Suche nach bestimmten Daten vereinfachen

Logische Optimierung
eine gegebene Abfrage so umformulieren, dass sie dasselbe Resultat liefert aber effizienter berechnet werden kann, beispielsweise weil kleinere Zwischenresultate erzeugt werden

Physische Optimierung
effiziente Algorithmen auswählen, um die Operationen der relationalen Algebra zu implementieren

Sequentielles Abarbeiten

```
SELECT *
FROM Filme
WHERE Jahr = 2010
```

Filme

FId	Jahr	Dauer
1	2014	110
2	2012	90
3	2012	120
4	2010	100
5	2013	120
6	2011	95
7	2008	12
8	2012	105
9	2010	97
10	2009	89
11	2014	102
12	2007	89
13	2008	130

Index für SELECT * FROM Filme WHERE Jahr = 2010

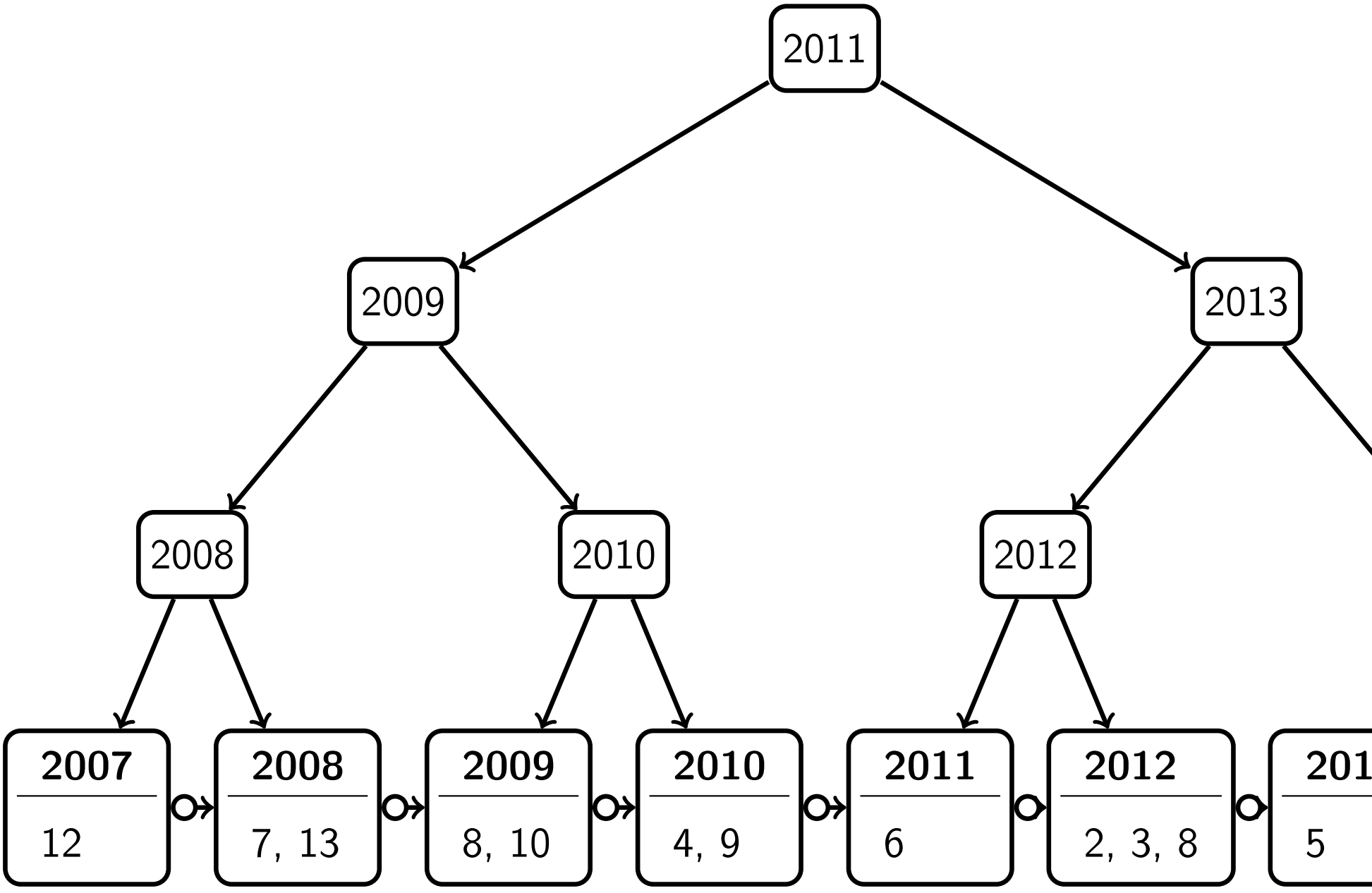
Filme

FId	Jahr	Dauer
1	2014	110
2	2012	90
3	2012	120
4	2010	100
5	2013	120
6	2011	95
7	2008	12
8	2012	105
9	2010	97
10	2009	89
11	2014	102
12	2007	89
13	2008	130

Index

Jahr	FId
2007	12
2008	7, 13
2009	8, 10
2010	4, 9
2011	6
2012	2, 3, 8
2013	5
2014	1, 11

B^++-Baum


2011
2009
2013
2008
2010
2012
2007
12
2008
7, 13
2009
8, 10
2010
4, 9
2011
6
2012
2, 3, 8
201
5

Verkettung der Blattknoten

Dank der Verkettung der Blattknoten kann der Index auch für Queries verwendet werden, welche Vergleichsoperatoren verwenden. Betrachten wir folgende SQL Abfrage:

```
SELECT *
FROM Filme
WHERE Jahr >= 2010
```

Um das Resultat dieser Abfrage zu berechnen, suchen wir zuerst wie oben den Blattknoten für das Jahr 2010. Nun können wir einfach durch die verkettete Liste iterieren, um die Knoten für die Jahre grösser als 2010 zu finden.

Einige Überlegungen

- Es werden Bäume mit einem hohen Verzweigungsgrad eingesetzt, z.T. hat ein Knoten 100 Nachfolger. Damit wird die Tiefe des Baumes kleiner und die Suche geht schneller.
- Die Bäume sind balanciert, d.h. die linken und rechten Teilbäume sind jeweils etwa gleich gross. Damit dauert die Suche immer etwa gleich lange.
- Echte Implementationen berücksichtigen die Speicherstruktur. Der Zugriff auf die gesuchten Daten soll mit möglichst wenigen Page Loads erfolgen.

CREATE INDEX

```
CREATE INDEX ON Filme (Jahr)
```

PostgreSQL erzeugt automatisch einen Index für den Primärschlüssel einer Tabelle. Ausserdem werden für alle weiteren Attributmengen, auf denen ein `UNIQUE` Constraint definiert wurde, automatisch Indizes erzeugt.

Die Verwendung von Indizes kann Abfragen beschleunigen.
Es entsteht dafür ein zusätzlicher Aufwand bei `INSERT` und `UPDATE` Operationen, da nun nicht nur die Tabelle geändert wird, sondern auch der Index angepasst werden muss.

Die Zeit, welche die Suche in einem Baum benötigt, ist in der Ordnung von $\log_g(n)$, wobei g der Verzweigungsgrad des Baumes und n die Anzahl der Datensätze ist.

Hash Funktionen

Eine Hashfunktion ist eine Funktion, welche Suchschlüssel auf sogenannte Behälter (Buckets) abbildet.
Ein Behälter ist eine Speichereinheit, welche die Daten, die dem Suchschlüssel entsprechen, aufnehmen kann.
Im Falle eines Hash Index, wird so ein Behälter dann Referenzen auf die eigentlichen Tupel enthalten. Formal ist eine Hashfunktion also eine Abbildung:

$$h : S \rightarrow B \ ,$$

wobei S die Menge der möglichen Suchschlüssel und B die Menge von (oder eine Nummerierung der) Behälter ist.

Hash Index

Hashfunktion: $h(x) := x \bmod 3$

Behälter	Jahr	FId
0	2010	4
	2013	5
	2010	9
	2007	12
1	2014	1
	2011	6
	2008	7
	2014	11
	2008	13
2	2012	2
	2012	3
	2012	8
	2009	10

Überlegungen

Mit Hilfe eines Hash Indexes kann nun in *konstanter* Zeit gesucht werden. Um beispielsweise die Filme des Jahres 2012 zu suchen, berechnen wir den Hashwert von 2012 und erhalten $h(2012) = 2$. Wir können somit direkt den Behälter 2 laden und müssen nur noch bei den darin enthaltenen Filmen (maximal fünf) testen, ob `Jahr` $= 2012$ erfüllt ist.

Baum Indizes sind vielseitiger einsetzbar als Hash Indizes. Deshalb werden Bäume als Standardstruktur für Indizes verwendet. Wir können jedoch explizit angeben, dass PostgreSQL einen Hash Index für das Attribut `Jahr` der Tabelle `Filme` anlegen soll. Dazu verwenden wir die Anweisung:

```
CREATE INDEX ON Filme  USING hash (Jahr)
```

Partielle Indizes

Ein partieller Index wird nur auf einem Teil einer Tabelle erstellt, wobei dieser Teil durch ein Prädikat definiert wird. Der Index enthält dann nur Einträge für Tabellenzeilen, die das Prädikat erfüllen.

Da eine Abfrage, welche nach einem häufigen Wert sucht, sowieso nicht auf einen Index zugreifen wird, macht es auch keinen Sinn, häufige Werte in einem Index zu halten.

Die Verwendung eines partiellen Indexes hat folgende Vorteile:

- Der Index wird kleiner. Dadurch werden die Operationen, welche auf den Index zugreifen, schneller.
- Updates der Tabelle werden schneller, da nicht in jedem Fall der Index aktualisiert werden muss.

Partielle Indizes: Beispiel

Tabelle welche bezahlte und unbezahlte Bestellungen enthält. Dabei machen die unbezahlten Bestellungen nur einen kleinen Bruchteil der Tabelle aus, jedoch greifen die meisten Abfragen darauf zu.

Dieser Index wird mit folgender Anweisung erstellt

```
CREATE INDEX ON Bestellungen (BestellNr)
WHERE Bezahlt is not true
```

Die folgende Abfrage kann nun diesen Index verwenden:

```
SELECT *
FROM Bestellungen
WHERE Bezahlt is not true AND BestellNr < 10000
```

Der Index kann sogar in Queries verwendet werden, welche nicht auf das Attribut `BestellNr` zugreifen, so z.B.:

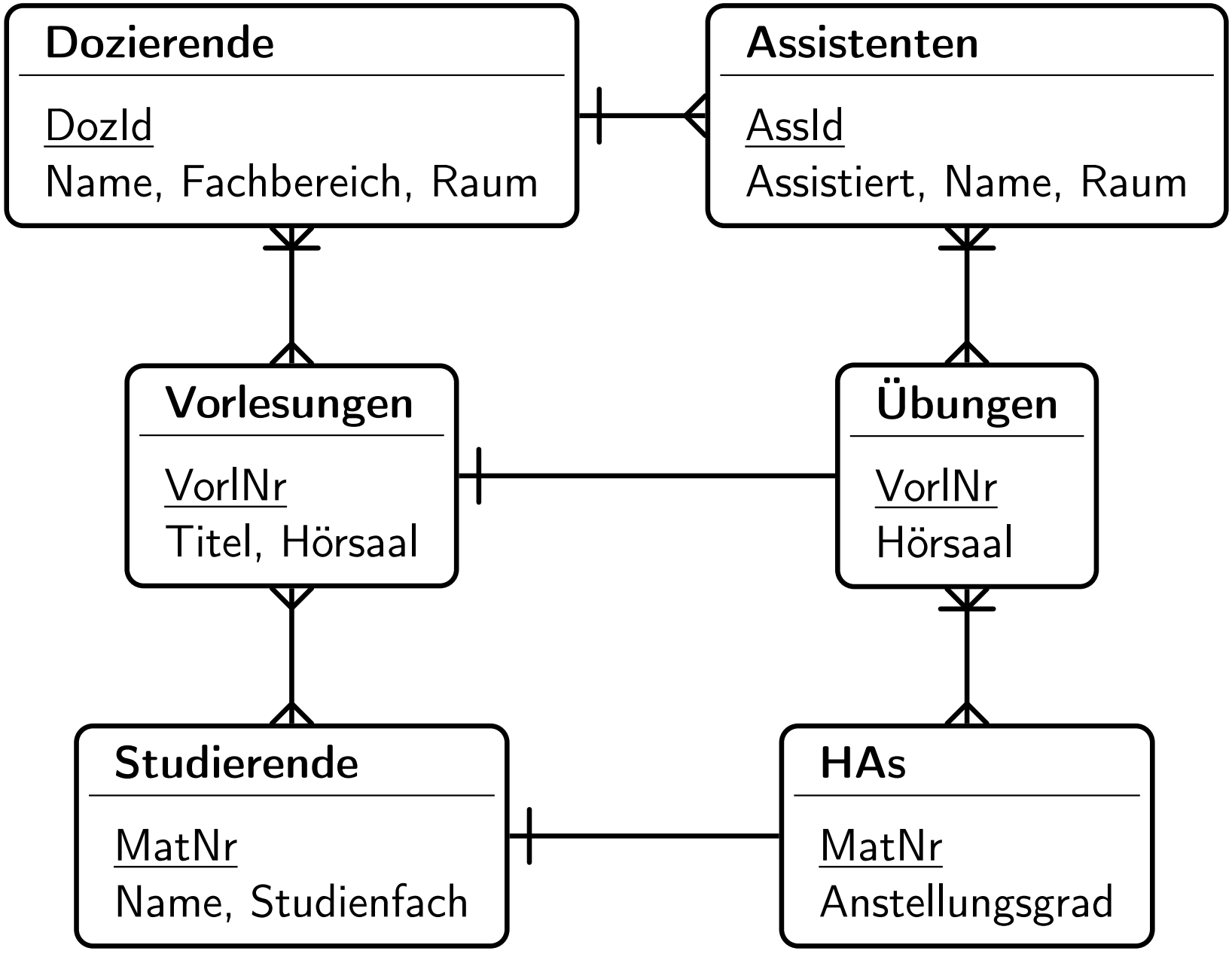
```
SELECT *
FROM Bestellungen
WHERE Bezahlt is not true AND Betrag > 5000
```

Logische und physische Optimierung: Übersetzung einer SQL Abfrage

1. Die SQL Abfrage wird geparst und in einen entsprechenden Ausdruck der relationalen Algebra übersetzt. Dies beinhaltet auch das Auflösen von Views.
2. Der Anfrageoptimierer erzeugt nun aus dem relationalen Ausdruck einen sogenannten Auswertungsplan, das heisst, eine effiziente Implementierung zur Berechnung der Relation, welche durch den relationalen Ausdruck beschrieben wird.
3. Im letzten Schritt wird der Auswertungsplan vom Datenbanksystem entweder kompiliert oder direkt interpretiert.

Zu einer SQL Abfrage gibt es viele Möglichkeiten, wie diese implementiert werden kann. Im Allgemeinen geht es bei der Optimierung nicht darum, die beste Implementierung zu finden, sondern nur eine gute.

Hochschuldatenbank



Beispiel

Wir werden die Tabellen nur durch den Anfangsbuchstaben ihres Namens bezeichnen.

Finden den Namen derjenigen Dozierenden, der die Assistentin `Meier` zugeordnet ist.

SQL Query:

```
SELECT D.Name
FROM D, A
WHERE A.Name = 'Meier' AND D.DozId = A.Assistiert
```

Kanonische Übersetzung:

$$\pi_{\texttt{D.Name}}(\sigma_{\texttt{A.Name='Meier'} \wedge \texttt{D.DozId=A.Assistiert}}(D \times A)) \ .$$

Abschätzungen

Annahme: 10 Dozierende, 50 Assistierende

$$\pi_{\texttt{D.Name}}(\sigma_{\texttt{A.Name='Meier'} \wedge \texttt{D.DozId=A.Assistiert}}(D \times A)) \ .$$

Kartesisches Produkt: 500 Tupeln
Selektion aus diesen 500 Tupeln: 1 Tupel

Besser:

$$\pi_{\texttt{D.Name}}(\sigma_{\texttt{D.DozId=A.Assistiert}}(D \times \sigma_{\texttt{A.Name='Meier'}}(A))) \ .$$

Selektion aus 50 Tupeln: 1 Tupel
Kartesisches Produkt: 10 Tupel
Selektion aus diesen 10 Tupeln: 1 Tupel

Noch besser: Θ-Join

$$\pi_{\texttt{D.Name}}(D \bowtie_{\texttt{D.DozId=A.Assistiert}} (\sigma_{\texttt{A.Name='Meier'}}(A))) \ .$$

Abfrageplan:

1. Wie bisher wird zuerst die passende Assistentin selektiert.
2. Damit kennen wir den Wert ihres `Assistiert` Attributs und wissen, welchen Wert das `DozId` Attribut der gesuchten Dozierenden haben muss.
3. Wir können also die entsprechende Dozierende mit Hilfe des Indexes auf dem Attribut `DozId` effizient suchen.
4. Dieser Index existiert, weil `DozId` der Primärschlüssel ist.

Umformungen

$E_1 \equiv E_2$ heisst, die relationalen Ausdrücke E_1 und E_2 enthalten dieselben Attribute und sind bis auf die Reihenfolge der Spalten gleich.

1. Aufbrechen und Vertauschen von Selektionen. Es gilt

$$\sigma_{\Theta_1 \wedge \Theta_2}(E) \equiv \sigma_{\Theta_1}(\sigma_{\Theta_2}(E)) \equiv \sigma_{\Theta_2}(\sigma_{\Theta_1}(E)) \ .$$

2. Kaskade von Projektionen. Sind $A_1, \ldots, A_m$ und $B_1, \ldots, B_n$ Attribute mit

$$\{A_1, \ldots, A_m\} \subseteq \{B_1, \ldots, B_n\} \ ,$$

   so gilt

$$\pi_{A_1,\ldots,A_m}(\pi_{B_1,\ldots,B_n}(E)) \equiv \pi_{A_1,\ldots,A_m}(E) \ .$$

3. Vertauschen von Selektion und Projektion. Bezieht sich das Selektionsprädikat Θ nur auf die Attribute $A_1, \ldots, A_m$, so gilt

$$\pi_{A_1,\ldots,A_m}(\sigma_{\Theta}(E)) \equiv \sigma_{\Theta}(\pi_{A_1,\ldots,A_m}(E)) \ .$$

Umformungen 2

4. Kommutativität. Es gelten

   $E_1 \times E_2 \equiv E_2 \times E_1 \qquad E_1 \bowtie E_2 \equiv E_2 \bowtie E_1 \qquad E_1 \bowtie_\Theta E_2 \equiv E_2 \bowtie_\Theta E_1$

5. Assoziativität. Es gelten

   $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3) \qquad (E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$

   Bezieht sich die Joinbedingung Θ_1 nur auf Attribute aus E_1 sowie E_2 und die Joinbedingung Θ_2 nur auf Attribute aus E_2 sowie E_3, so gilt

   $$(E_1 \bowtie_{\Theta_1} E_2) \bowtie_{\Theta_2} E_3 \equiv E_1 \bowtie_{\Theta_1} (E_2 \bowtie_{\Theta_2} E_3) \ .$$

Umformungen 3

6. Vertauschen von Selektion und kartesischem Produkt. Bezieht sich das Selektionsprädikat Θ nur auf die Attribute aus E_1, so gilt

$$\sigma_\Theta(E_1 \times E_2) \equiv \sigma_\Theta(E_1) \times E_2 \ .$$

7. Vertauschen von Projektion und kartesischem Produkt. Sind $A_1, \ldots, A_m$ Attribute von E_1 und $B_1, \ldots, B_n$ Attribute von E_2, so gilt

$$\pi_{A_1,\ldots,A_m,B_1,\ldots,B_n}(E_1 \times E_2) \equiv \pi_{A_1,\ldots,A_m}(E_1) \times \pi_{B_1,\ldots,B_n}(E_2) \ .$$

   Dieselbe Idee funktioniert auch bei Θ-Joins. Falls sich die Join Bedingung Θ nur auf die Attribute $A_1, \ldots, A_m$ und $B_1, \ldots, B_n$ bezieht, so gilt

$$\pi_{A_1,\ldots,A_m,B_1,\ldots,B_n}(E_1 \bowtie_\Theta E_2) \equiv \pi_{A_1,\ldots,A_m}(E_1) \bowtie_\Theta \pi_{B_1,\ldots,B_n}(E_2) \ .$$

Umformungen 4

8. Selektion ist distributiv über Vereinigung und Differenz. Es gelten

$$\sigma_\Theta(E_1 \cup E_2) \equiv \sigma_\Theta(E_1) \cup \sigma_\Theta(E_2) \qquad \sigma_\Theta(E_1 \setminus E_2) \equiv \sigma_\Theta(E_1) \setminus \sigma_\Theta(E_2) \ .$$

9. Projektion ist distributiv über Vereinigung. Es gilt

$$\pi_{A_1 \ldots, A_m}(E_1 \cup E_2) \equiv \pi_{A_1 \ldots, A_m}(E_1) \cup \pi_{A_1 \ldots, A_m}(E_2) \ .$$

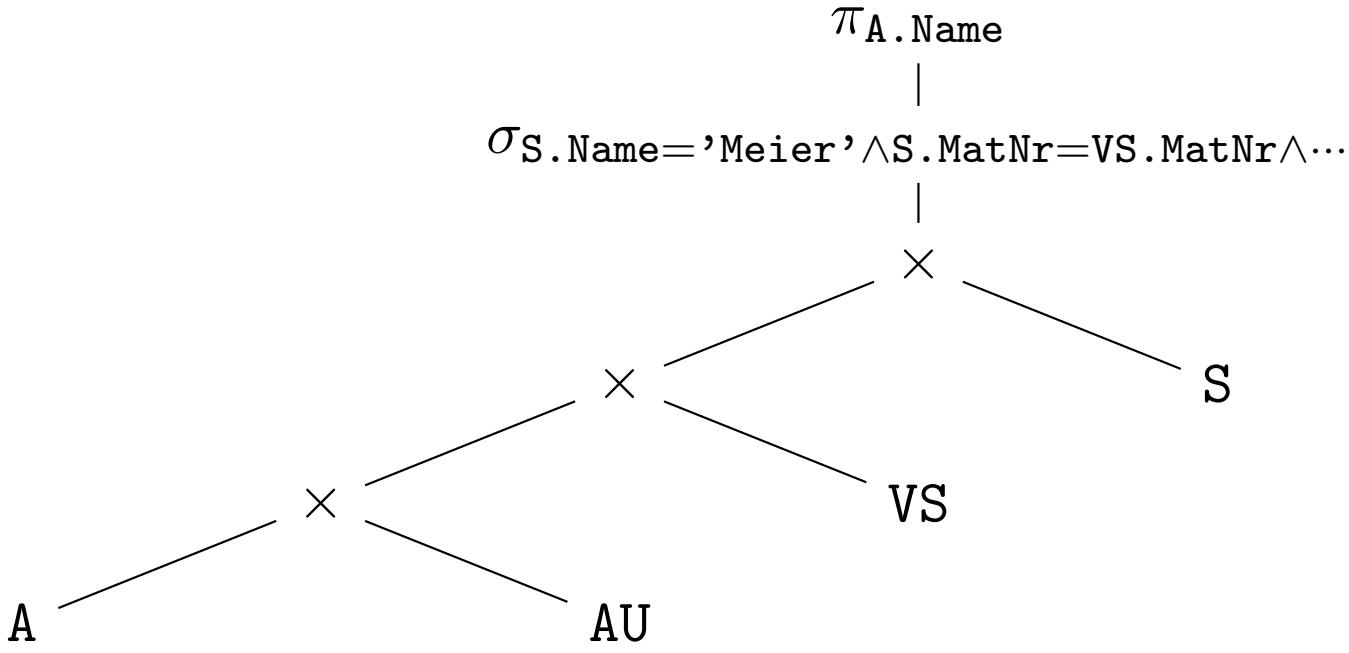
   Es ist zu beachten, dass in der Regel Projektionen *nicht* distributiv über Differenzen sind.

Ablauf der Umformungen

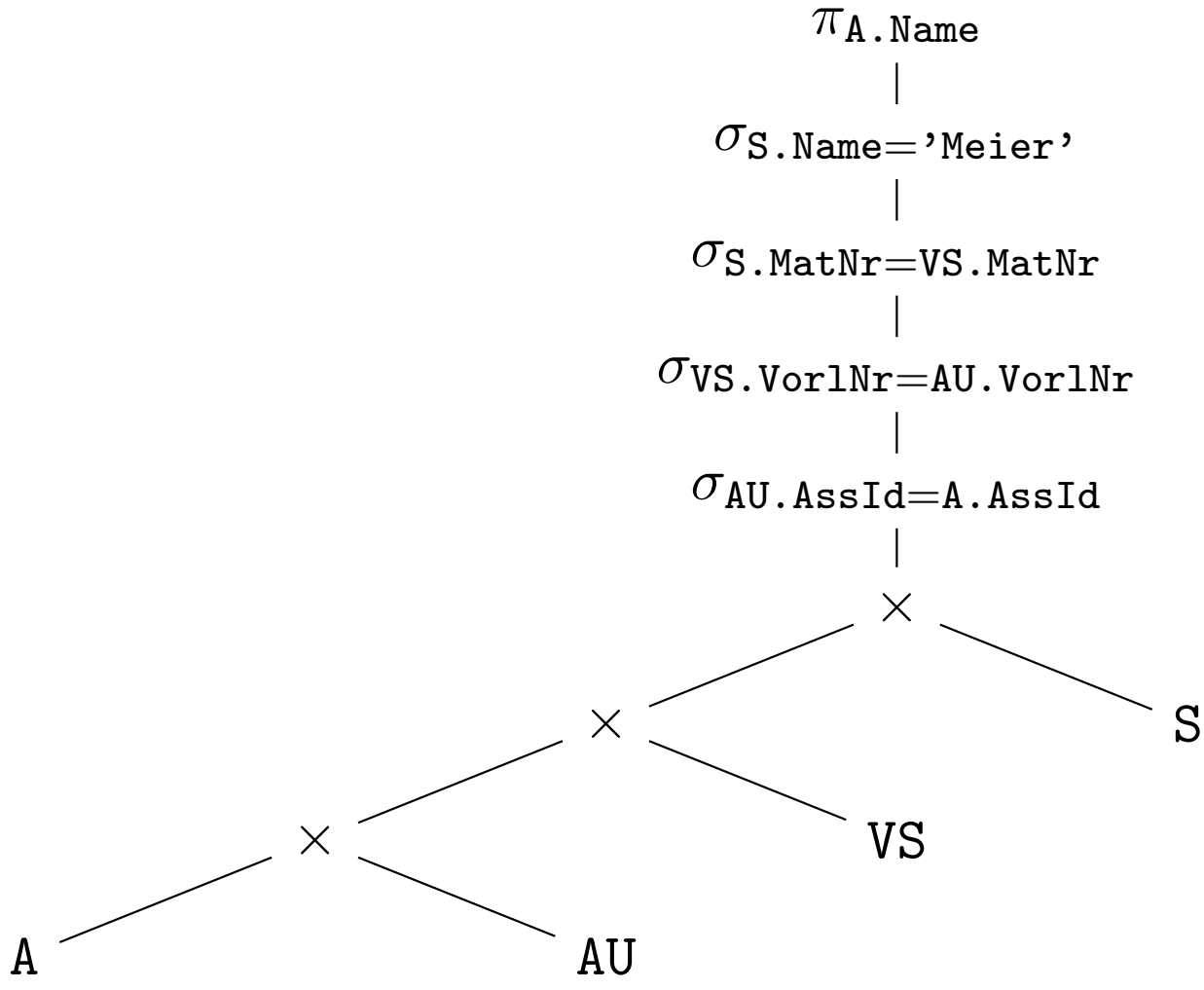
1. Mittels der ersten Regel werden konjunktive Selektionsprädikate in Kaskaden von Selektionsoperationen zerlegt.
2. Mittels der Regeln 1, 3, 6 und 8 werden Selektionsoperationen soweit wie möglich nach innen propagiert.
3. Wenn möglich, werden Selektionen und kartesische Produkte zu Θ-Joins zusammengefasst.
4. Mittels Regel 5 wird die Reihenfolge der Joins so vertauscht, dass möglichst kleine Zwischenresultate entstehen.
5. Mittels der Regeln 2, 3, 7 und 9 werden Projektionen soweit wie möglich nach innen propagiert.

Beispiel: Suche die Namen aller Assistierenden, welche die Studierende Meier betreuen

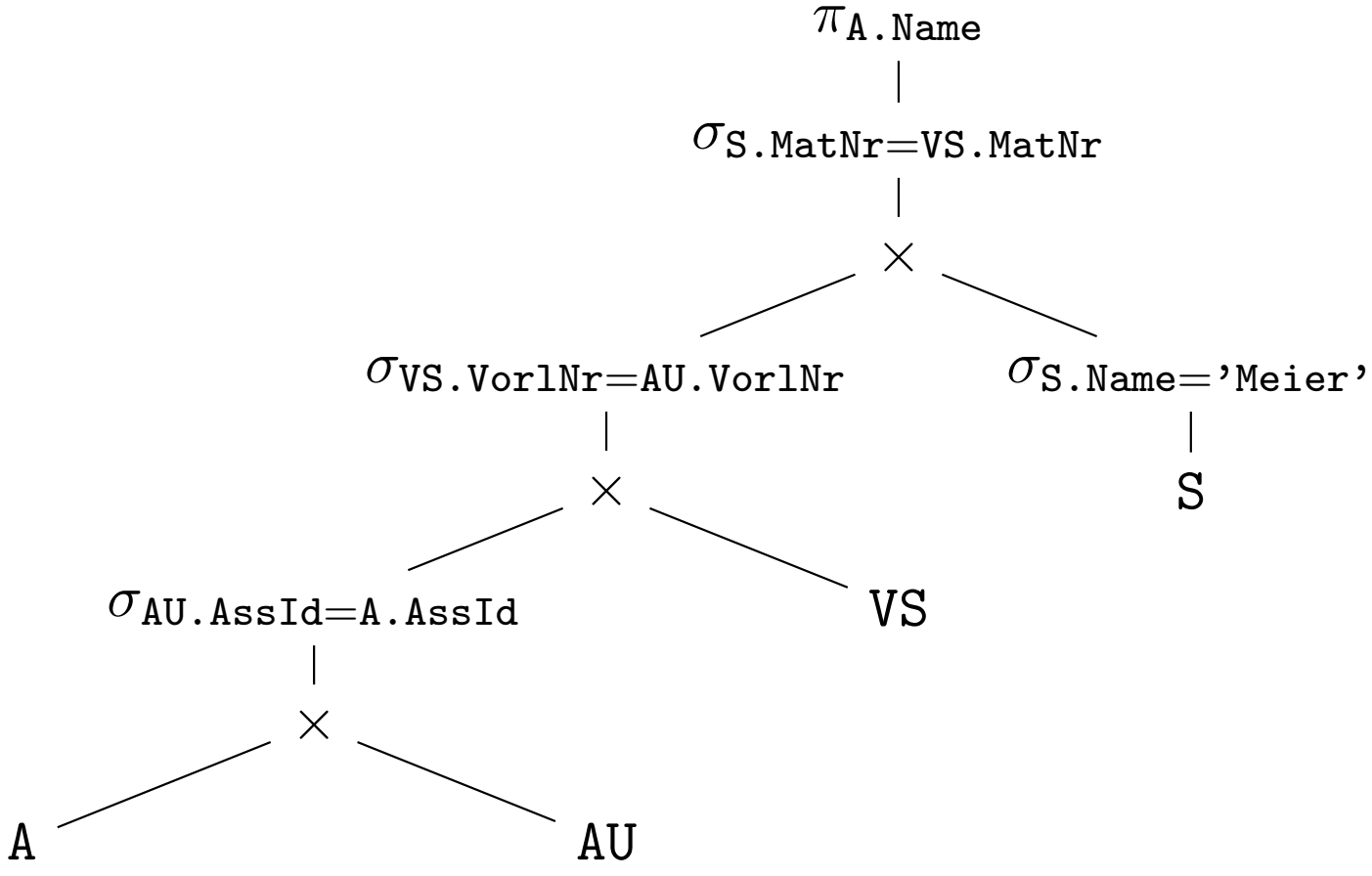
```
SELECT A.Name
FROM A, AU, VS, S
WHERE S.Name = 'Meier' AND S.MatNr = VS.MatNr AND
      VS.VorlNr = AU.VorlNr AND AU.AssId = A.AssId
```



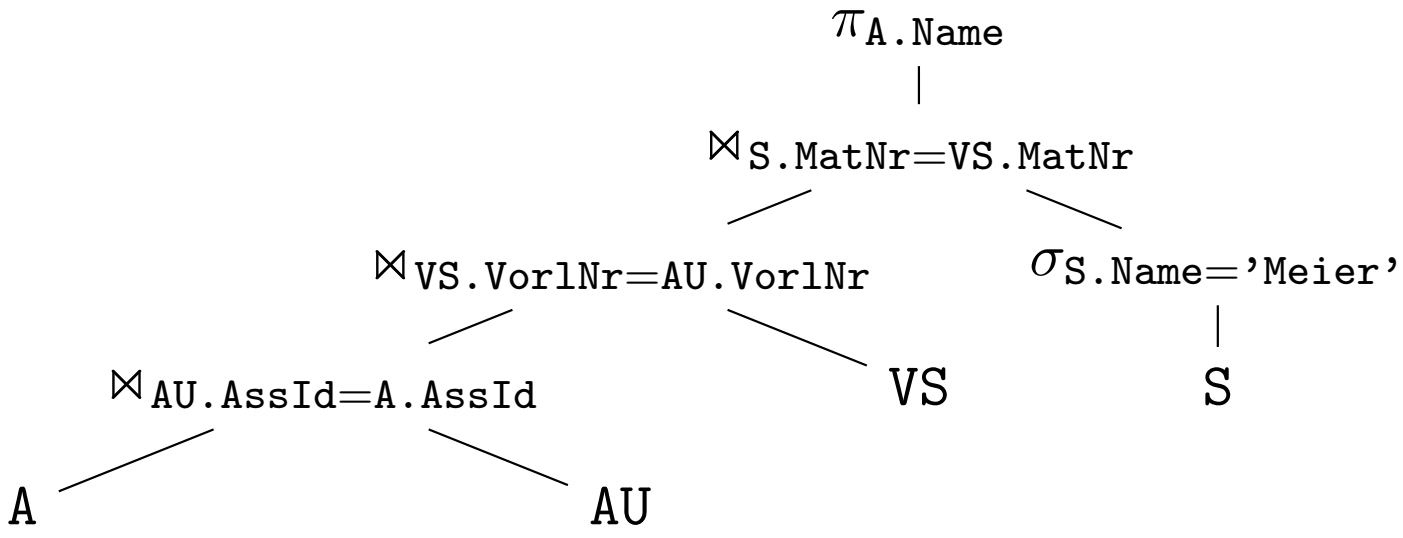
Aufspalten der Selektionsprädikate



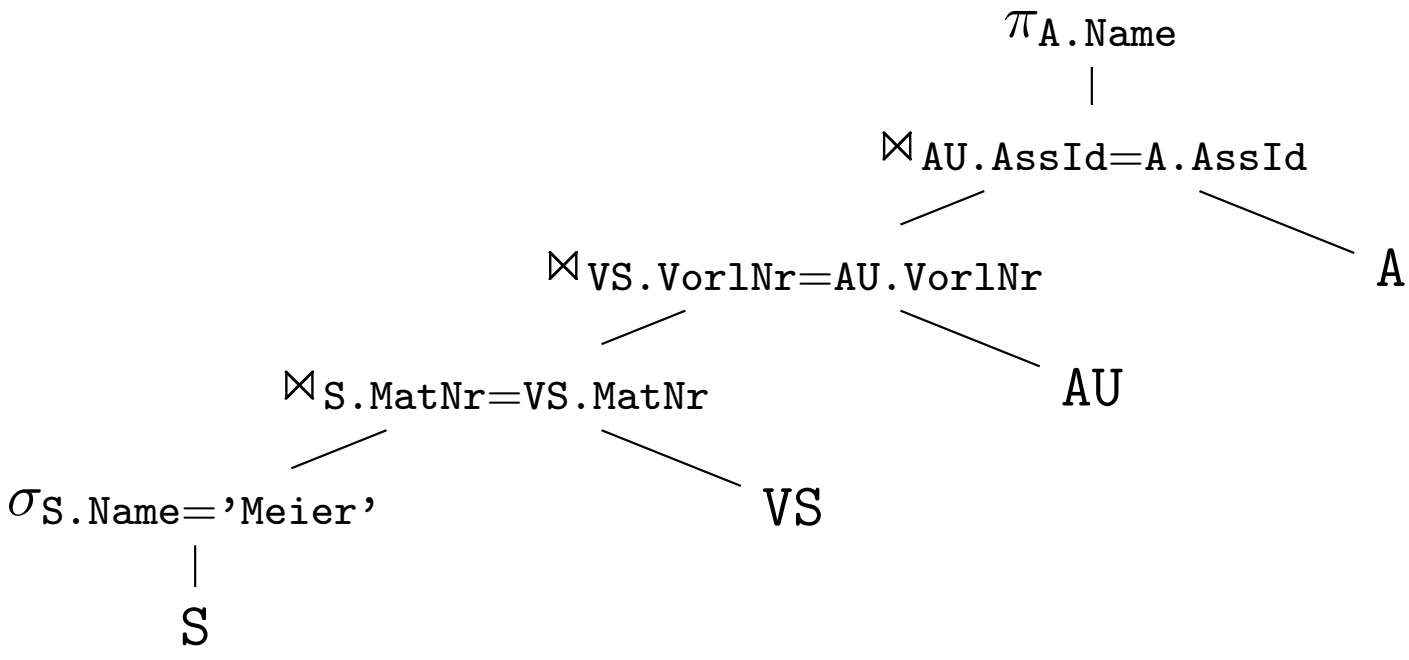
Verschieben der Selektionsoperationen



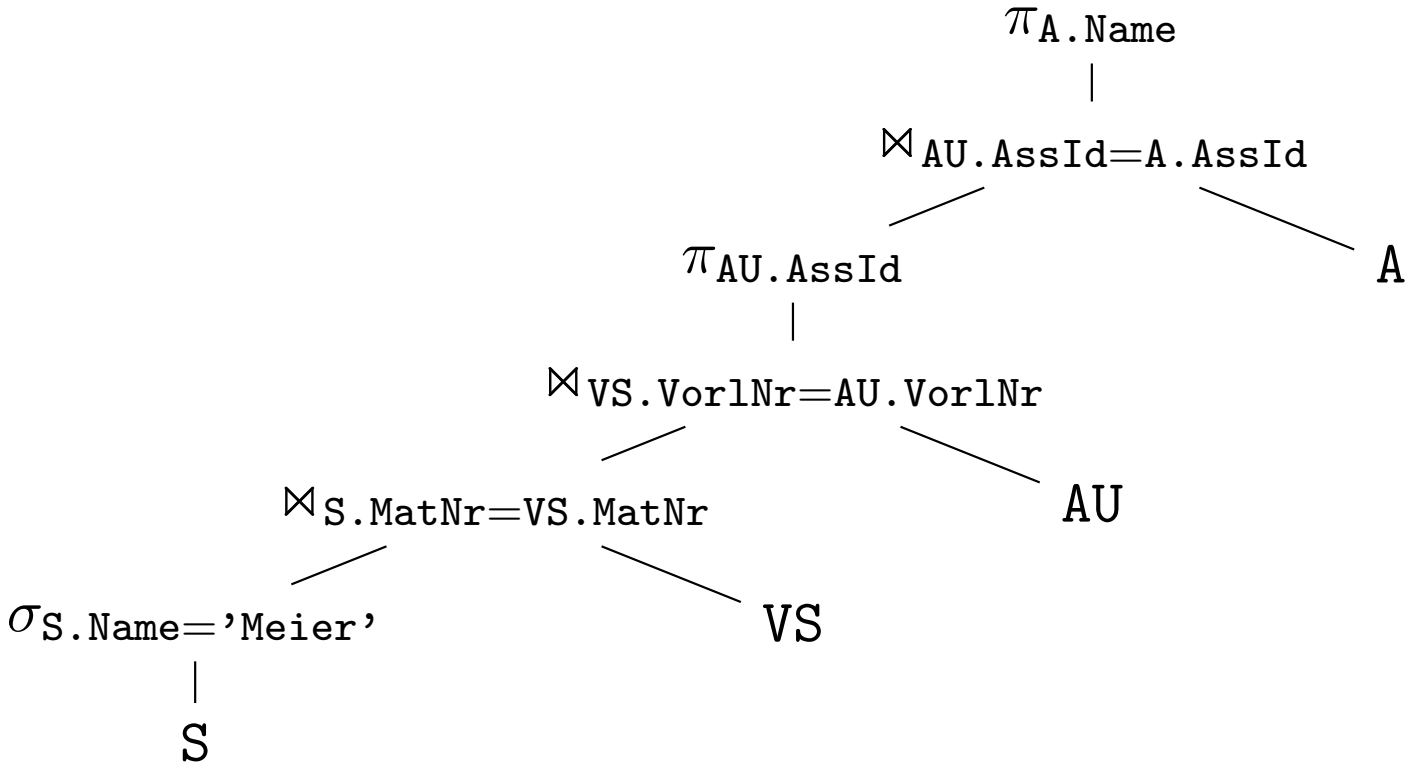
Zusammenfassen von Selektionen und kartesischen Produkten zu Join Operationen



Optimierung der Join Reihenfolge



Zusätzliche Projektionen, weniger Spalten in den Zwischenresultaten



EXPLAIN

Query:

```
EXPLAIN
    SELECT *
    FROM T
    WHERE v = 700
```

Auswertungsplan:

```
Seq Scan on t
  Filter: (v = 700)
```

Wir erstellen nun einen Index:

```
CREATE INDEX ON T (v)
```

Jetzt:

```
Index Scan using t_v_idx on t
  Index Cond: (v = 700)
```

Tests auf NULL

```
EXPLAIN
    SELECT *
    FROM T
    WHERE v IS NULL
```

liefert

```
Index Scan using t_v_idx on t
  Index Cond: (v IS NULL)
```

Aber:

```
EXPLAIN
    SELECT *
    FROM T
    WHERE v IS NOT NULL
```

liefert

```
Seq Scan on t
  Filter: (v IS NOT NULL)
```

Nested Loop Join

```
FOR EACH s IN S
  FOR EACH t IN T
    IF s[v] = t[v] THEN OUTPUT(s,t)
```

```
Nested Loop
  Join Filter: (s.v = t.v)
  ->  Seq Scan on s
  ->  Materialize
        ->  Seq Scan on t
```

Index Join

Mit Index `I` auf dem Attribut `v` in der Tabelle `T`

```
FOR EACH s IN S
  t := FIRST t IN I WITH t[v] = s[v]
  WHILE t EXISTS
        OUTPUT (s,t)
        t := NEXT t in I WITH t[v] = s[v]
```

```
Nested Loop
  ->  Seq Scan on s
  ->  Index Scan using t_v_idx on t
        Index Cond: (v = s.v)
```

Zwei Queries, ein Auswertungsplan

Annahme: `S` hat 100 Einträge und `T` hat 99 Einträge.

Beide Queries

```
SELECT * FROM S, T
```

und

```
SELECT * FROM T, S
```

liefern den Auswertungsplan

```
Nested Loop
  ->  Seq Scan on s
  ->  Materialize
        ->  Seq Scan on t
```

Die *kleinere* Tabelle materialisiert und für die äussere Schleife verwendet.

Merge Join

`S[i]` ist das `i`-te Tupel in `S`, `#S` ist die Anzahl Tupel in `S`.

```
S := SORT(S,v)
T := SORT(T,v)
i := 1
j := 1
WHILE ( i <= #S  AND  j <= #T )
   IF ( S[i][v] = T[j][v] ) THEN
      jj = j
      WHILE ( S[i][v] = T[j][v]  AND  j <= #T )
         OUTPUT (S[i],T[j])
         j++
      j =jj
      i++
   ELSE IF ( S[i][v] > T[j][v] ) THEN
      j++
   ELSE
      i++
```

Merge Join: Auswertungsplan

```
Merge Join
  Merge Cond: (t.v = s.v)
  ->  Sort
        Sort Key: t.v
        ->  Seq Scan on t
  ->  Sort
        Sort Key: s.v
        ->  Seq Scan on s
```

Hash Join

Sei `T` kleiner als `S`
Erzeuge Hashtabelle für `T`
`BT(i)` bezeichne den i-ten Behälter und `h` sei die Hashfunktion

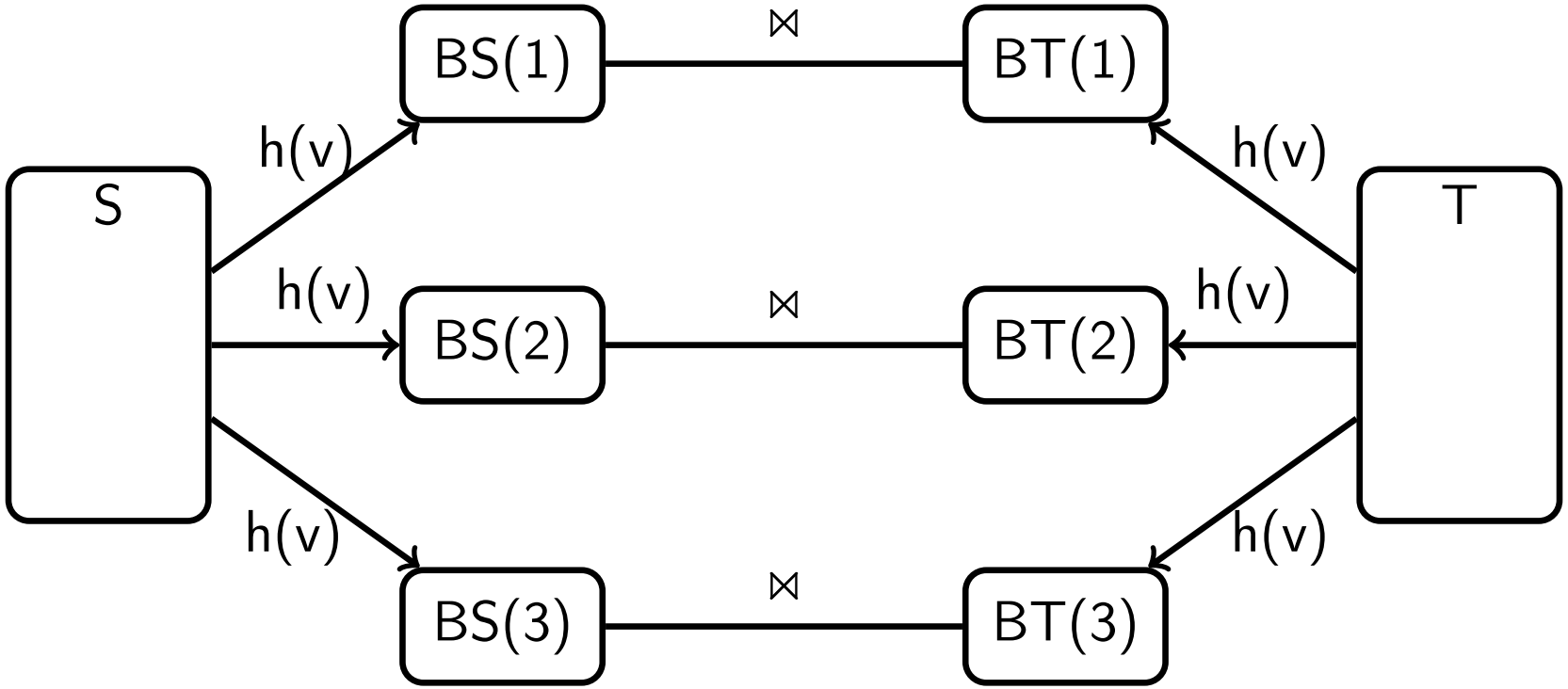
```
FOR EACH t IN T
    i := h( t[v] )
    ADD t TO BT(i)
FOR EACH s IN S
    i = h(s[v])
    FOR EACH t in BT(i)
        IF ( s[v] = t[v] ) THEN OUTPUT(s,t)
```

Hash Join Auswertungsplan

```
Hash Join
  Hash Cond: (s.v = t.v)
  ->  Seq Scan on s
  ->  Hash
        ->  Seq Scan on t
```

Hash Join Variante

Falls S und T sehr gross sind, so können beide Relationen mit Hilfe einer Hashfunktion partitioniert werden.



Hash Join Variante

```
FOR EACH s IN S
   i := h( s[v] )
   ADD s TO BS(i)
FOR EACH t IN T
   i := h( t[v] )
   ADD t TO BT(i)
FOR EACH i IN 0..n
   FOR EACH s in BS(i)
      FOR EACH t in BT(i)
         IF s[v] = t[v] THEN OUTPUT(s,t)
```