Abfrageoptimierung

7

In diesem Kapitel geht es darum, wie ein Datenbanksystem Abfragen effizient, d. h. schnell, beantworten kann. Dazu betrachten wir im ersten Teil sogenannte Indizes. Das sind Hilfsobjekte, welche die Suche nach bestimmten Daten unterstützen. Insbesondere führen wir Indizes ein, welche auf B^+-Bäumen oder Hash-Funktionen basieren.

Anschliessend studieren wir Methoden zur logischen und physischen Query-Optimierung. Bei der logischen Optimierung geht es darum, eine gegebene Abfrage so umzuformulieren, dass sie dasselbe Resultat liefert aber effizienter berechnet werden kann, beispielsweise weil kleinere Zwischenresultate erzeugt werden. Damit wird weniger Speicherplatz benötigt und die Berechnung kann schneller ausgeführt werden.

Die physische Optimierung behandelt die Auswahl der Algorithmen, welche die Operationen der relationalen Algebra implementieren. Wir führen drei Algorithmen ein, um Joins zu berechnen: Nested Loop Join, Merge Join und Hash Join. Wir zeigen auch, wie Auswertungspläne von PostgreSQL die verschiedenen Algorithmen darstellen.

7.1 Indizes

Indizes sind Hilfsobjekte in einer Datenbank, welche der Beschleunigung von Suchabfragen dienen. Es sind Hilfsobjekte in dem Sinn, dass sie für die Semantik der Datenbank irrelevant sind. Indizes enthalten somit keine zusätzlichen Daten. Sie ermöglichen nur alternative (d. h. schnellere) Zugriffswege auf bereits bestehende Daten in der Datenbank.

Um die Wirkungsweise von Indizes zu illustrieren, betrachten wir die Tabelle `Filme`, gegeben durch:

T. Studer, *Relationale Datenbanken*,
https://doi.org/10.1007/978-3-662-58976-2_7

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

Wir möchten nun alle Filme des Jahres 2010 suchen. Dazu verwenden wir die SQL Abfrage:

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

Um diese Abfrage zu beantworten, muss das Datenbanksystem die Tabelle `Filme` sequentiell, d. h. Tupel für Tupel, abarbeiten und bei jedem Tupel testen, ob

$$\texttt{Jahr} = 2010$$

erfüllt ist.

Dieses Vorgehen ist natürlich nicht besonders effizient und führt bei grossen Tabellen zu einem bedeutenden Zeitaufwand. Wir können die Suche nach den Filmen des Jahres 2010 wesentlich beschleunigen, wenn wir die Tabelle nach dem Wert des Attributs `Jahr` sortieren. Dies ist genau die Idee der Indizes. Ein Index auf dem Attribut `Jahr` für die Tabelle `Filme` ist eine Hilfstabelle, welche nach dem Jahr sortiert ist und für jedes Jahr Zeiger auf die Filme dieses Jahres enthält. Ein solcher Index hat also die Form:

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

Diese Sortierung nach dem Jahr können wir nun ausnutzen, um eine effiziente Suche zu implementieren. Im Kontext von Datenbanksystemen wird diese häufig mit sogenannten B^+-Bäume realisiert. In solchen Bäumen sind die Daten in den Blattknoten gespeichert. Die Knoten, welche keine Blätter sind, enthalten sogenannte Suchschlüssel. Blätter mit Werten kleiner als der Suchschlüssel befinden sich dann im linken Teilbaum, Blätter mit Werten grösser als oder gleich dem Suchschlüssel befinden sich im rechten Teilbaum.

Abb. 7.1 zeigt den Baum für einen Index auf dem Attribut `Jahr` der Tabelle `Filme`. Um die Filme des Jahres 2010 zu suchen, beginnen wir beim Wurzelknoten. Dieser enthält den Suchschlüssel 2011. Also gehen wir in den linken Teilbaum und kommen zum Knoten mit dem Suchschlüssel 2009. Wir gehen in den rechten Teilbaum und kommen zum Suchschlüssel 2010. Wiederum gehen wir in den rechten Teilbaum und finden den Blattknoten für das Jahr 2010 mit den Referenzen auf die Filme mit `FId` 4 und 9. Der Zeitaufwand für diese Suche ist logarithmisch in der Anzahl der Blattknoten. Der Index ermöglicht also eine wesentlich effizientere Suche als die (lineare) Suche in der ungeordneten Originaltabelle.

In unserem B^+-Baum sind die Blattknoten zusätzlich noch als geordnete verkettete Liste miteinander verbunden. In Abb. 7.1 ist diese Verkettung durch die Pfeile der Form $\circ\!\!\longrightarrow$ angegeben. Dank dieser zusätzlichen Struktur 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. Ohne die Verkettung der Blattknoten wäre der Index für Queries dieser Art nutzlos.

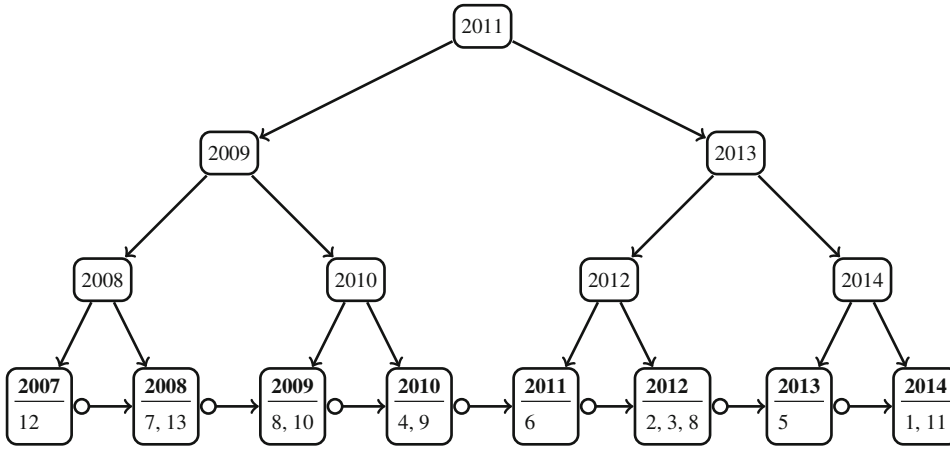

Abb. 7.1 B^+-Baum als Indexstruktur für das Attribut `Jahr`

Abb. 7.1 soll nur die Grundidee von Indexstrukturen erklären. In echten Datenbanksystemen werden nicht binäre Bäume zur Datenorganisation verwendet, da diese in der Praxis zu wenig effizient wären. Im Folgenden wollen wir einige Überlegungen zum Design der tatsächlich verwendeten Datenstrukturen angeben.

- 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.
- Die Bäume enthalten zusätzliche Links zwischen den Knoten, welche effiziente parallele Zugriffe auf den Index ermöglichen. Dieser klassische Ansatz wurde bereits in [2] eingeführt. PostgreSQL verwendet eine Variante davon, die im `README` File[1] der `btree` Implementierung ausführlich beschrieben ist.

In PostgreSQL können wir einen Baum Index für das Attribut `Jahr` der Tabelle `Filme` mit folgender Anweisung erzeugen:

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

Ein Index kann nicht nur auf einem Attribut erzeugt werden, sondern auch auf einer Attributmenge. Die allgemeine Anweisung lautet:

`CREATE INDEX ON` <*Tabelle*> (<*Attribut*$_1$>, ..., <*Attribut*$_n$>)

[1] `src/backend/access/nbtree/README`.

Anmerkung 7.1. 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.

Anmerkung 7.2. Die Verwendung von Indizes kann Abfragen beschleunigen. Die Kehrseite der Medaille ist jedoch, dass ein zusätzlicher Aufwand bei `INSERT` und `UPDATE` Operationen entsteht, da nun nicht nur die Tabelle geändert wird, sondern auch der Index angepasst werden muss.

Indizes werden angelegt, damit Daten schnell gefunden werden können. 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. Ein effizienterer Ansatz ist es, einen Index mit Hilfe einer Hashfunktion aufzubauen.

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:

$$h : S \to B,$$

wobei S die Menge der möglichen Suchschlüssel und B die Menge von (oder eine Nummerierung der) Behälter ist. Normalerweise ist die Kardinalität von S sehr viel grösser als die Anzahl der Behälter. Wichtig für das Design einer Hashfunktion ist, dass die verschiedenen Werte von S möglichst gleichmässig auf B verteilt werden.

Betrachten wir nun einen Hash Index auf dem Attribut `Jahr` der Tabelle `Filme`. Als Hashwert verwenden wir den Rest einer Ganzzahldivision durch 3:

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

Weiter nehmen wir an, dass jeder Behälter Referenzen für fünf Filme aufnehmen kann. Damit erhalten wir den Hash Index aus Abb. 7.2.

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. Die Verwendung eines Hash Indexes kann also sehr effizient sein, wenn wir auf Gleichheit testen wollen. Jedoch können wir den Hash Index nicht einsetzen um die Filme mit `Jahr` ≥ 2012 zu suchen, da wir dann wieder *alle* Behälter laden müssten und damit keinen Performancegewinn hätten.

Baum Indizes sind also vielseitiger einsetzbar als Hash Indizes. Aus diesem Grund werden Bäume als Standardstruktur für Indizes verwendet. Wir können jedoch explizit

Abb. 7.2 Hash Index

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

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)
```

Das Verfahren zur Bildung von Hash Indizes, wie wir es oben beschrieben haben, ist natürlich zu statisch für praktische Datenbanksysteme. Im obigen Beispiel können wir beispielsweise keinen weiteren Film mit `Jahr` 2014 einfügen, da der entsprechende Behälter bereits voll ist. In konkreten Implementierungen werden deshalb Formen von *erweiterbarem* Hashing eingesetzt. Wir verzichten hier aber auf die Beschreibung dieser dynamischen Ansätze.

Anmerkung 7.3. Hash Indizes sind in PostgreSQL erst ab Version 10 transaktionssicher. In älteren Versionen musste der Index neu erzeugt werden (von Hand mit `REINDEX`), falls es ungeschriebene Änderungen gab. Ansonsten hätten Queries, die auf den Index zugreifen, falsche Resultate geliefert.

Neben B^+-Bäumen und Hash Indizes unterstützt PostgreSQL noch weitere Index Arten wie GiST, GIN und BRIN Indizes. Diese sind in der PostgreSQL Dokumentation ausführlich beschrieben [3].

Ebenfalls unterstützt werden 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. Ein partieller Index kann verwendet werden, um uninteressante Werte (die aber häufig vorkommen) aus dem Index auszuschliessen. 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.

In der PostgreSQL Dokumentation [3] werden partielle Indizes unter anderem durch folgendes Beispiel illustriert.

Beispiel 7.4. Betrachten wir eine 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. In dieser Situation kann ein partieller Index auf den unbezahlten Bestellungen die Performance verbessern.

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
```

Hier muss das Datenbanksystem den gesamten Index scannen. Falls der Anteil der unbezahlten Rechnungen relativ klein ist, kann sich dies aber lohnen (verglichen mit dem Aufwand, die ganze Tabelle zu durchsuchen).

7.2 Logische Optimierung

SQL ist eine *deklarative* Sprache. Das bedeutet, dass eine SQL Abfrage nur spezifiziert, welche Eigenschaften ein Tupel haben muss, damit es in die Resultatrelation aufgenommen wird. Die SQL Abfrage sagt aber nicht, wie diese Tupel gefunden werden. Das heisst die Abfrage deklariert, *was* gesucht werden soll, aber nicht *wie* gesucht werden soll.

Es ist die Aufgabe des Datenbanksystems eine SQL Abfrage in einen geeigneten Algorithmus zu übersetzen, um die Resultatrelation effizient zu berechnen. Diese Übersetzung geschieht in folgenden Schritten:

1. Die SQL Abfrage wird geparst und in einen entsprechenden Ausdruck der relationalen Algebra übersetzt.[2] Dies beinhaltet auch das Auflösen von Views: für jede Verwendung einer View wird die entsprechende Definition der View eingesetzt.
2. Der Abfrageoptimierer erzeugt nun aus dem relationalen Ausdruck einen sogenannten Auswertungsplan (Queryplan), 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 (dies wäre zu aufwändig), sondern nur eine gute.

Der Abfrageoptimierer arbeitet auf zwei Ebenen. Zum einen kann man zu einem gegebenen relationalen Ausdruck einen logisch äquivalenten relationalen Ausdruck suchen, der schnell und mit wenig Speicherbedarf berechnet werden kann. Zum anderen müssen möglichst effiziente Algorithmen gefunden werden, um die Operationen (Selektionen, Joins) eines gegebenen Ausdrucks zu implementieren. Den ersten Aspekt nennen wir logische Optimierung, den zweiten physische Optimierung.

Zur Illustration betrachten wir die Hochschul-Datenbank aus Beispiel 3.3. Das vollständige Schema dazu ist in Abb. 3.18 dargestellt. Wir werden im Folgenden die Tabellen nur durch den Anfangsbuchstaben ihres Namens bezeichnen. Das DB-Schema besteht also aus den Tabellen: `D`, `A`, `DV`, `AU`, `V`, `U`, `VS`, `UH`, `S` und `H`.

Wir wollen zuerst mit einem einfachen Beispiel die Grundidee der logischen und physischen Optimierung zeigen. Mit folgender SQL Abfrage finden wir den Namen derjenigen Dozierenden, der die Assistentin `Meier` zugeordnet ist.

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

Die kanonische Übersetzung (siehe Abschn. 5.2) dieser Abfrage lautet

$$\pi_{\texttt{D.Name}}(\sigma_{\texttt{A.Name='Meier'} \wedge \texttt{D.DozId=A.Assistiert}}(D \times A)). \tag{7.1}$$

Nehmen wir nun an, es gibt zehn Dozierende und fünfzig Assistierende. Mit (7.1) wird zuerst das kartesische Produkt berechnet, welches aus $10 \cdot 50 = 500$ Tupeln besteht. Aus

[2] Dies ist eine Vereinfachung, da es SQL Queries gibt, zu keinen äquivalenten Ausdruck in der relationalen Algebra besitzen (bspw. rekursive Queries).

diesen werden dann diejenigen Tupel selektiert, welche

$$\texttt{V.Name} = \texttt{'Meier'} \wedge \texttt{D.DozId} = \texttt{A.Assistiert}$$

erfüllen. In diesem Fall ist das nur ein Tupel. Am Schluss wird dann noch auf `D.Name` projiziert. Offensichtlich ist das keine effiziente Methode, um das Resultat der SQL Query zu berechnen.

Viel besser ist die Auswertung, welche durch folgenden Ausdruck beschrieben wird

$$\pi_{\texttt{D.Name}}(\sigma_{\texttt{D.DozId=A.Assistiert}}(D \times \sigma_{\texttt{A.Name='Meier'}}(A))). \tag{7.2}$$

Hier wird aus nur 50 Tupeln die gesuchte Assistentin selektiert. Anschliessend wird das Kreuzprodukt gebildet, welches nun nur *zehn* Tupel enthält. Aus diesen wird dann das Tupel selektiert, welches die Join Bedingung erfüllt. Mit diesem Ausdruck haben wir viel kleinere Zwischenresultate. Damit wird zum einen der Speicherverbrauch reduziert und zum anderen können die Selektionen schneller berechnet werden. Die Tatsache, dass jetzt zwei Selektionsoperationen ausgeführt werden müssen fällt nicht ins Gewicht, da die Selektionsbedingungen einfacher sind.

Eine weitere Verbesserung erreichen wir mit folgendem Ausdruck, welcher das kartesische Produkt mit der anschliessenden Selektion zu einem Θ-Join zusammenfasst:

$$\pi_{\texttt{D.Name}}(D \bowtie_{\texttt{D.DozId=A.Assistiert}} (\sigma_{\texttt{A.Name='Meier'}}(A))). \tag{7.3}$$

Dieser Ausdruck sagt, dass wir nicht das komplette kartesische Produkt bilden müssen, um das Resultat zu berechnen. Mit (7.3) wird nämlich durch physische Optimierung folgender Auswertungsplan erzeugt. Wie bisher wird zuerst die passende Assistentin selektiert. Damit kennen wir den Wert ihres `Assistiert` Attributs und wissen auch, welchen Wert das `DozId` Attribut der gesuchten Dozierenden haben muss. Wir können also die entsprechende Dozierende mit Hilfe des Indexes auf dem Attribut `DozId` effizient suchen. Beachte, dass dieser Index existiert, weil `DozId` der Primärschlüssel ist. Hier wird also dank einer guten Umformung in der Phase der logischen Optimierung die Verwendung eines Indexes in der Phase der physischen Optimierung ermöglicht, was schliesslich zu einem effizienten Auswertungsplan führt.

Randbemerkung aus der Praxis: Im konkreten Fall mit zehn Dozierenden ist die Relation wahrscheinlich zu klein, als dass sich die Verwendung eines Indexes lohnt. Bei grösseren Relationen wird jedoch ein Auswertungsplan erzeugt, welcher den Index wie beschrieben berücksichtigt (siehe auch Anmerkung 7.6).

Wir wollen nun die Ansätze der logischen Optimierung im Detail studieren. Die Methoden der physischen Optimierung betrachten wir dann im nächsten Abschnitt.

Wir geben eine Reihe von Paaren äquivalenter relationaler Ausdrücke an. Dabei heisst äquivalent, dass die Reihenfolge der Attribute bei den beschriebenen Relationen keine Rolle spielt. Für relationale Ausdrücke E_1 und E_2 verwenden wir die Notation $E_1 \equiv E_2$

um auszudrücken, dass die entsprechenden Relationen dieselben Attribute enthalten und bis auf die Reihenfolge der Spalten gleich sind.

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

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

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

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

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

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

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.

Die eben beschriebenen Äquivalenzen können natürlich für Transformationen in beide Richtungen verwendet werden. Die entscheidende Frage bei der Optimierung ist, welche Richtung in einem konkreten Fall günstiger ist. Die folgenden Heuristiken haben sich dabei als vorteilhaft erwiesen.

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.

Wir betrachten nochmals das Schema der Hochschul-Datenbank aus Abb. 3.18, wobei wir wieder die Tabellennamen abkürzen. Folgende SQL Abfrage liefert 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
```

Abb. 7.3 zeigt die kanonische Übersetzung dieser Abfrage, wobei der relationale Ausdruck als Baum dargestellt wird.

Aufspalten der Selektionsprädikate liefert den Ausdruck in Abb. 7.4.

Verschieben der Selektionsoperationen liefert den Ausdruck in Abb. 7.5.

Zusammenfassen von Selektionen und kartesischen Produkten zu Join Operationen liefert den Ausdruck in Abb. 7.6.

Optimierung der Join Reihenfolge liefert den Ausdruck in Abb. 7.7.

Wir können in diesem Beispiel die Projektion nicht direkt nach innen schieben. Jedoch können wir durch geschickte Anwendung der Regeln 2 und 7 zusätzliche Projektionen einfügen. Damit erreichen wir, dass gewisse Zwischenresultate weniger Spalten haben und somit kleiner sind. Mit diesem Ansatz können wir beispielsweise den Baum aus Abb. 7.8 erhalten, indem wir die Projektion $\pi_{\texttt{AU.AssId}}$ einfügen.

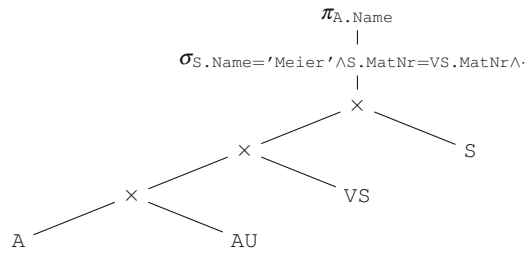

Abb. 7.3 Kanonische Übersetzung

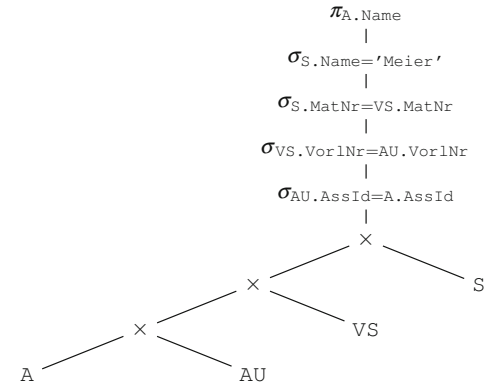

Abb. 7.4 Aufspalten der Selektionsprädikate

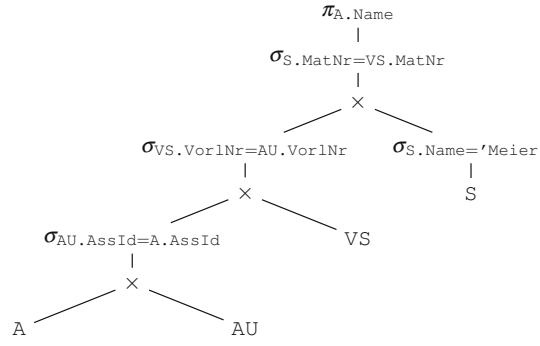

Abb. 7.5 Verschieben der Selektionsoperationen

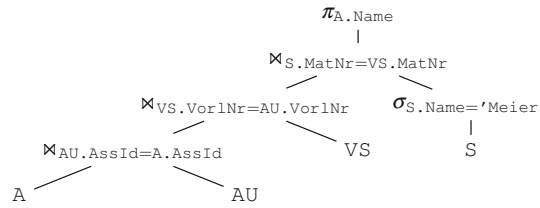

Abb. 7.6 Selektionen und kartesische Produkte zu Joins zusammenfassen

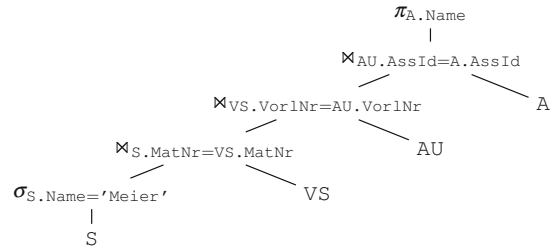

Abb. 7.7 Optimierung der Join Reihenfolge

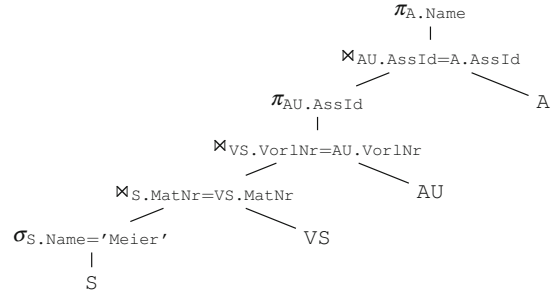

Abb. 7.8 Zusätzliche Projektionen einfügen

7.3 Physische Optimierung

Das Problem der physischen Optimierung besteht darin, die logischen Operationen der relationalen Algebra durch effiziente Algorithmen praktisch zu realisieren. Dabei kann es zu einem Operator durchaus mehrere mögliche Implementierungen geben. Es geht also beispielsweise darum, zu einer gegebenen Abfrage herauszufinden, ob und welche Indizes zur Berechnung des Resultats eingesetzt werden können und welche Algorithmen am effizientesten sind.

PostgreSQL bietet die Möglichkeit, die Auswertungsstrategie einer Abfrage anzuzeigen. Mit der Anweisung

```
EXPLAIN <Abfrage>
```

wird der Queryplan für die Query *<Abfrage>* angezeigt. Dadurch kann man sehen, wie die Abfrage tatsächlich bearbeitet wird.

Beispiel 7.5. Betrachten wir eine Tabelle `T` mit den Attributen `TId` und `v`. Mit der Anweisung

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

wird der Auswertungsplan für die Query angezeigt, mit der alle Tupel aus der Tabelle `T` gefunden werden, welche $v = 700$ erfüllen. Wir erhalten den Auswertungsplan:[3]

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

Dabei bedeutet `Seq Scan on T`, dass die Tabelle `T` sequentiell, d. h. Tupel für Tupel, durchgegangen wird und für jedes Tupel $v = 700$ getestet wird.

Wir erstellen nun einen Index für das Attribut `v` mit der Anweisung:

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

Mit der obigen `EXPLAIN` Anweisung erhalten wir nun den Auswertungsplan:

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

Dabei ist `t_v_idx` der Name, welcher PostgreSQL unserem Index zugewiesen hat. Es erfolgt also jetzt nicht mehr ein sequentieller Scan der ganzen Tabelle, stattdessen werden die Tupel mit $v = 700$ im Index gesucht.

Anmerkung 7.6. Es bedeutet natürlich einen gewissen Mehraufwand, zuerst auf den Index zuzugreifen und erst dann die eigentliche Tabelle auszulesen. Dieser Mehraufwand lohnt sich nur bei grossen Tabellen. Bei kleinen Tabellen ist es günstiger die Tabelle sequentiell zu durchsuchen als zuerst auf den Index zuzugreifen. Das heisst, falls die Tabelle `T` im obigen Beispiel weniger als 500 Einträge hat, wird PostgreSQL wahrscheinlich einen sequentiellen Scan ausführen und den Index ignorieren.

Anmerkung 7.7. Der Index kann auch bei einem Test auf `Null` verwendet werden. Mit der Anweisung

[3] Wir erinnern uns, dass PostgreSQL intern alle Identifier in Kleinbuchstaben übersetzt, siehe Anmerkung 5.1. Deshalb hat die Tabelle `T` im Auswertungsplan den Namen `t`.

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

erhalten wir den Auswertungsplan:

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

PostgreSQL kann Baum Indizes auch verwenden, um Abfragen mit Tests auf `NOT NULL` zu optimieren.

Anmerkung 7.8. Bei einem `Index Scan` wird jeweils *ein* Zeiger auf ein Tupel aus dem Index geholt und dann direkt auf das entsprechende Tupel zugegriffen. Somit wird jedes Tupel, das die Selektionsbedingung erfüllt, einzeln geladen. Dieses Vorgehen ist gut, falls nur wenige Tupel die Selektionsbedingung erfüllen.

Es ist jedoch nicht effizient, falls viele Tupel die Bedingung erfüllen. In diesem Fall wird folgender Auswertungsplan erzeugt:

```
Bitmap Heap Scan on t
  Recheck Cond: (v = 700)
  ->  Bitmap Index Scan on t_v_idx
        Index Cond: (v = 700)
```

Ein `Bitmap Index Scan` holt in einem Durchgang *alle* Zeiger, welche die Selektionsbedingung erfüllen, aus dem Index und speichert diese in einer Bitmap-Struktur im Hauptspeicher. Diese hat folgenden Aufbau: Nehmen wir an, die Tabelle `T` hat n Einträge. Dann besteht die Bitmap aus einer Sequenz von n Bit, wobei das i-te Bit genau dann den Wert 1 hat, wenn ein Zeiger auf den i-ten Eintrag aus dem Index geholt wurde (d. h. der i-te Eintrag der Tabelle `T` erfüllt die Selektionsbedingung).

Der `Bitmap Heap Scan` iteriert nun durch die Bitmap (vom ersten bis zum n-ten Bit) und lädt das i-te Tupel aus der Tabelle `T`, falls das i-te Bit 1 ist. Damit werden die Tupel in der Reihenfolge ausgegeben, in der sie physikalisch abgespeichert sind. Dies ermöglicht einen effizienten Speicherzugriff, da räumliche Lokalität ausgenutzt werden kann.

Falls die Bitmap-Struktur zu gross wird, repräsentieren wir nicht jedes Tupel einzeln in der Bitmap, stattdessen repräsentiert ein Bit der Bitmap eine Seite des Speichers. Wir merken uns in der Bitmap also nicht mehr die einzelnen Tupel, welche die Selektionsbedingung erfüllen, sondern nur noch die Seiten, welche diese Tupel enthalten. Der `Bitmap Heap Scan` lädt dann die i-te Seite, falls das i-te Bit der Bitmap gesetzt ist. In dieser

Seite muss dann für jedes Tupel geprüft werden, ob es die Selektionsbedingung erfüllt. Dies ist der `Recheck` im Auswertungsplan.

Wir wollen hier das Problem der physischen Optimierung nicht im Allgemeinen untersuchen, sondern nur mögliche Implementierungen der Join Operation betrachten. PostgreSQL unterstützt drei Algorithmen zur Berechnung eines Joins:

1. Nested Loop Join
2. Merge Join
3. Hash Join

Wir werden diese nun etwas genauer studieren. Dazu nehmen wir an, dass wir zusätzlich zur obigen Tabelle `T` noch eine Tabelle `S` haben mit den Attributen `SId` und `v`. Wir betrachten nun den Join $S \bowtie T$.

Nested Loop Join

Der einfachste Algorithmus zur Berechnung eines Joins heisst Nested Loop Join. Dabei werden mit zwei verschachtelten Schleifen alle möglichen Kombinationen von Elementen aus `S` und `T` erzeugt und jeweils getestet, ob die Join Bedingung erfüllt ist.

Folgender Pseudocode Algorithmus beschreibt einen Nested Loop Join. Dabei heisst `OUTPUT(s,t)`, dass die Tupel `s` und `t` miteinander verbunden (entsprechend dem Schema der Resultat-Tabelle) und dem Resultat hinzugefügt werden sollen.

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

Mit der `EXPLAIN` Anweisung erhalten wir folgenden Queryplan für einen Nested Loop Join.

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

Dabei bedeutet `Materialize`, dass das Resultat der Operation unter dem `Materialize` Knoten im Speicher materialisiert wird, bevor der obere Knoten ausgeführt wird. Eine Materialisierung wird üblicherweise durchgeführt, wenn Daten mehrfach verwendet werden müssen (hier werden sie für jedes Tupel aus `S` wiederverwendet). Dank der Materialisierung müssen sie nicht jedes mal neu erzeugt werden.

Falls auf dem Attribut v in der Tabelle T ein Index erzeugt wurde, dann ist ein sogenannter *Index Join* möglich. Bei dieser Variante wird der Index verwendet, um die passenden Tupel der inneren Relation zu finden. Im untenstehenden Beispiel bezeichnet I den Index und wir nehmen an, dass wir durch diesen Index iterieren können (vergleiche die verkettete Liste der Blätter eines Baum Index).

```
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]
```

Wir erhalten dazu folgenden Queryplan:

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

Nested Loop Joins liefern auch ein erstes einfaches Beispiel dafür, dass verschiedene Queries denselben Auswertungsplan liefern können. Wieder gehen wir aus von den Tabellen S und T. Wir nehmen an, S habe 100 Einträge und T habe 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
```

Unabhängig von der Reihenfolge der Tabellen in der FROM Klausel wird also immer zuerst die *kleinere* Tabelle materialisiert. Die äussere Schleife iteriert dann durch die *grössere* Tabelle und greift auf die materialisierte, kleinere Tabelle zu.

Merge Join

Die Idee des Merge Joins ist einfach. Wenn die Tabellen S und T nach dem Join Attribut sortiert sind, so muss man bei der Berechnung des Joins ein Tupel aus S nicht mit allen Tupeln aus T vergleichen, sondern nur mit denjenigen, die etwa gleich gross sind.

Um einen Merge Join der Tabellen S und T durchzuführen, sortieren wir also zuerst sowohl S als auch T nach den Werten des Attributs v. Wir werden folgende Notation

verwenden, wobei i eine natürliche Zahl ist: S[i] bezeichnet das i-te Tupel der Tabelle S. Mit #S bezeichnen wir die Anzahl Tupel, welche in der Tabelle S enthalten sind. Analog verwenden wir T[i] und #T.

```
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++
```

Im entsprechenden Queryplan sieht man deutlich, dass beide Tabellen zuerst sortiert werden.

```
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

Wir beschreiben hier zuerst den klassischen Hash Join Algorithmus. Dazu nehmen wir an, dass T die kleinere Tabelle ist als S. Der Algorithmus erzeugt zuerst eine Hashtabelle für T. Mit BT(i) bezeichnen wir den i-ten Behälter der Hashtabelle und h sei die Hashfunktion. Ein klassischer Hash Join arbeitet nun wie folgt.

```
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)
```

Ein Tupel `s` aus `S` muss somit nicht mit allen Tupeln aus `T` verglichen werden, sondern nur mit denjenigen aus dem entsprechenden Behälter der Hashtabelle. Dies funktioniert, weil

$$\texttt{s[v] = t[v]} \quad \text{impliziert} \quad \texttt{h(s[v]) = h(t[v])}.$$

Falls die Joinbedingung nicht auf Gleichheit basiert, so kann kein Hash Join durchgeführt werden.

Folgender Auswertungsplan beschreibt einen Hash Join.

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

Falls bei einem Hash Join die beteiligten Relationen `S` und `T` sehr gross sind, so können beide Relationen mit Hilfe einer Hashfunktion partitioniert werden. Für die Berechnung des Joins müssen dann nur jeweils die beiden sich entsprechenden Behälter im Hauptspeicher sein. Die restlichen Behälter können in den Sekundärspeicher ausgelagert werden. Es werden also der Reihe nach alle sich entsprechenden Behälter geladen und auf ihren Daten ein Join berechnet, welcher zum Endresultat hinzugefügt wird. Abb. 7.9 stellt diese Idee graphisch dar.

Wir können diesen allgemeinen Hash Join wie folgt beschreiben, wobei `n` die Anzahl Behälter angibt. Wir geben hier jedoch nicht an, welche Behälter im Hauptspeicher sind

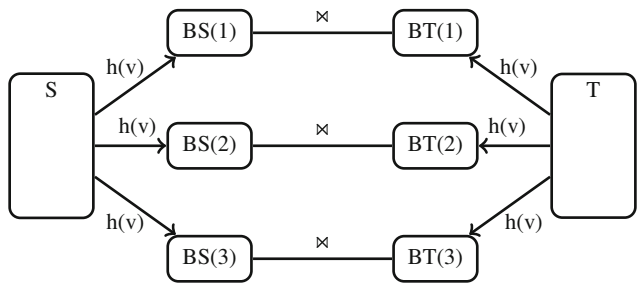

Abb. 7.9 Hash Join

und welche ausgelagert werden. Um den Verbund zweier sich entsprechender Behälter zu berechnen, verwenden wir hier einen Nested Loop Join. In der Praxis wählt man dazu einen klassischen Hash Join, der jedoch eine von `h` verschiedene Hashfunktion verwendet.

```
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)
```

Wir gehen hier nicht im Detail darauf ein, wie ein Datenbanksystem die physische Optimierung genau vornimmt und die anzuwendenden Algorithmen bestimmt. Nur soviel: die Grundidee ist es, die Kosten für die verschiedenen möglichen Implementierungen zu schätzen (bezüglich Zeitaufwand, das heisst bezüglich der Anzahl der benötigten Page Loads). Für diese Schätzungen führt das Datenbanksystem umfangreiche Statistiken über die Grösse der verschiedenen Tabellen und über die Verteilung der Daten in diesen Tabellen (Histogramme).

Weiterführende Literatur[4]

1. Kemper, A., Eickler, A.: Datenbanksysteme. Oldenbourg (2013)
2. Lehman, P.L., Yao, S.B.: Efficient locking for concurrent operations on b-trees. ACM Trans. Database Syst. **6**(4), 650–670 (1981). https://doi.org/10.1145/319628.319663
3. The PostgreSQL Global Development Group: PostgreSQL Documentation, Indexes (2018). https://www.postgresql.org/docs/current/static/indexes.html. Zugegriffen am 11.06.2019
4. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, 6. Aufl. McGraw-Hill (2010)
5. Veldhuizen, T.L.: Triejoin: a simple, worst-case optimal join algorithm. In: Schweikardt, N., Christophides, V., Leroy, V. (Hrsg.) Proceedings of 17th international conference on database theory (ICDT), S. 96–106. OpenProceedings.org (2014). https://doi.org/10.5441/002/icdt.2014.13

[4]Wir haben in diesem Kapitel nur eine Einführung in Indexstrukturen und Query-Optimierung geben können. Eine vertiefte Darstellung dieser Themen findet sich beispielsweise in [1, 4]. Dort werden unter anderem weitere Indexstrukturen besprochen. Auch werden weitere Algorithmen zur Implementierung der Query-Operatoren gezeigt. Es wird ausserdem detailliert auf die Aufwandschätzung zur Query-Optimierung eingegangen. Zum Schluss möchten wir noch festhalten, dass Indexstrukturen, Optimierung und Join Algorithmen hochaktuelle Forschungsgebiete sind, in denen laufend neue Ideen vorgeschlagen werden. Beispielsweise wird in [5] der *leapfrog triejoin* vorgestellt und es wird gezeigt, dass dieser Join Algorithmus worst-case optimal ist.