Anfragebearbeitung und Optimierung

In diesem Kapitel geht es darum, wie ein DBMS eine SQL-Anfrage verarbeitet. Also:

1. Schritte der Anfragebearbeitung
2. Parsen und Validieren
3. Optimieren und Erstellen des Zugriffsplans

Schritte der Anfragebearbeitung

Die einzelnen Schritte in Abb. 1 sind:

- Parsen – die SQL-Anweisung wird entsprechend der SQL-Grammatik in Tokens zerlegt und daraus der Syntaxbaum erzeugt:
  Input: SQL-Anweisung
  Akteur: Parser
  Output: Syntaxbaum
- Validieren – die Bezeichner im Syntaxbaum werden gegen den Systemkatalog geprüft und die Benutzerrechte werden überprüft:
  Akteur: Präprozessor
  Output: Valider Syntaxbaum
- Initialen logischen Zugriffsplan erstellen:
  Akteur: Plan-Generator
  Output: Initialer logischer Zugriffsplan
- Zugriffsplan restrukturieren:
  Akteur: Plan-Restrukturierer (Query Rewriter)
  Output: Logischer Zugriffsplan
- Ausführbaren Plan erzeugen:
  Akteur: Kostenschätzer
  Output: Zugriffsplan

Die letzten drei Schritte zusammen werden auch als Optimierung bezeichnet.

Um die Schritte der Anfragebearbeitung zu verstehen, gehen wir sie im Folgenden an einem Beispiel durch. Wir betrachten das Datenbankschema in Abb. 2 sowie die SQL-Anweisung

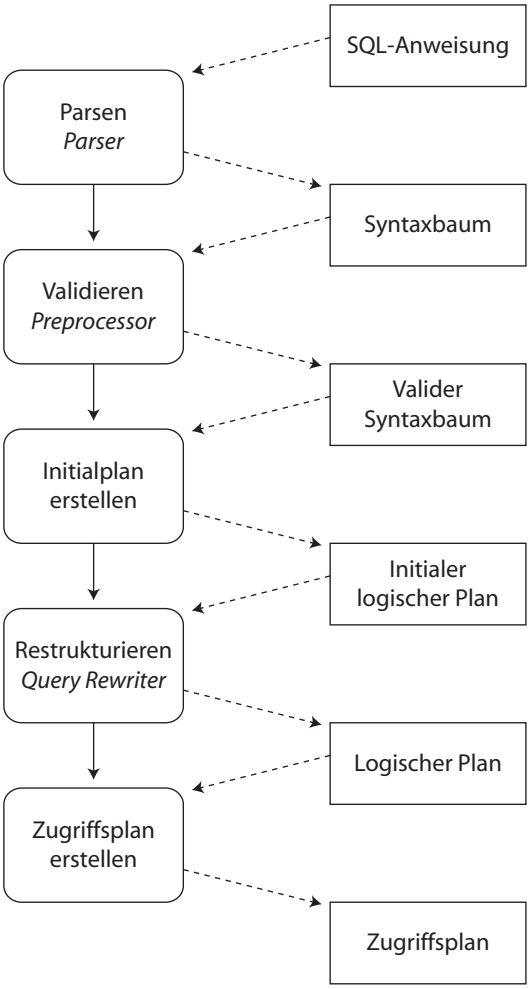

Abbildung 1: Schritte und Datenstrukturen der Anfragebearbeitung

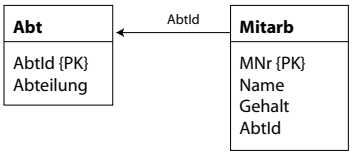

Abbildung 2: Datenbankschema für das Beispiel

```
SELECT Abteilung, Name
  FROM Abt NATURAL JOIN Mitarb
  WHERE Gehalt > 5000
```

und spielen durch, wie aus dieser SQL-Anweisung schrittweise ein ausführbarer Zugriffsplan entsteht.

Schritt 1: Parsen der SQL-Anweisung

Im ersten Schritt der Anfragebearbeitung wird die SQL-Anweisung geparst, d.h. sie wird in Tokens zerlegt und dann auf Basis der SQL-Grammatik in einen Syntaxbaum transformiert.

Der Parser überprüft dabei, ob die SQL-Anweisung syntaktisch korrekt ist. Dazu benötigt er nur eine Grammatik für SQL.

In unserem Beispiel verwenden wir einen kleinen Ausschnitt aus einer (denkbaren) SQL-Grammatik, um zu veranschaulichen, wie diese Transformation erfolgt.[1]

```
/* Teil einer Grammatik für SQL */

<Query>     ::= <SFW>

<SFW>       ::= SELECT <SelList> FROM <FromList> WHERE <Condition>

<SelList>   ::= [Attribute], <SelList>
             |  [Attribute]

<FromList>  ::= <TableExpr>, <FromList>
             |  <TableExpr>

<TableExpr> ::= [Table]
             |  [Table] NATURAL JOIN [Table]

<Condition> ::= <Condition> AND <Condition>
             |  [Attribute] IN <Query>
             |  [Attribute] = [Attribute]
             |  [Attribute] = [Value]
             |  [Attribute] > [Value]
             |  [Attribute] LIKE [Pattern]
```

Die Tokens in Großbuchstaben sind Terminale, nämlich Schlüsselworte von SQL.

Eine besondere Rolle spielen die Token `[Attribute]`, `[Table]`, `[Value]` und `[Pattern]`. Sie stehen nicht für weitere Regeln der Grammatik, sondern für Bezeichner, die gewisse Eigenschaften haben müssen:

`[Table]` steht für den Namen einer Tabelle in der Datenbank, er erfüllt gewissen Namenskonventionen. Im Zuge der Validierung (im

[1] Eine (wirkliche) Grammatik für SQL-92 findet man im Internet z.B. hier: http://savage.net.au/SQL/sql-92.bnf.html

nächsten Schritt der Anfragebearbeitung) kann im Systemkatalog überprüft werden, ob der Name der Tabelle im Datenbankschema gültig ist.

`[Attribute]` steht für den Namen eines Attributs. Er kann alleine stehen oder durch den Namen einer Relation qualifiziert sein. Auch wieder im Zuge der Validierung kann überprüft werden, ob der Name gültig ist.

`[Value]` steht für einen Wert, der entsprechend der Regeln von SQL anzugeben ist. Was erlaubt ist, hängt also vom Kontext, etwa dem Attribut ab, mit dem der Wert verglichen wird. Auch das kann im nächsten Schritt überprüft werden, weil im Systemkatalog steht, welchen Typ ein Attribut hat.

`[Pattern]` steht für einen String, der auch Wild Cards, also `_` und `%` enthalten kann. Der Operator `LIKE` ist natürlich nur bei Attributen vom Typ `char` und verwandten Typen erlaubt. Noch ein Fall für Validierung.

Ergebnis des Parsens ist der Syntaxbaum für die Anweisung, in unserem Beispiel in Abb. 3.

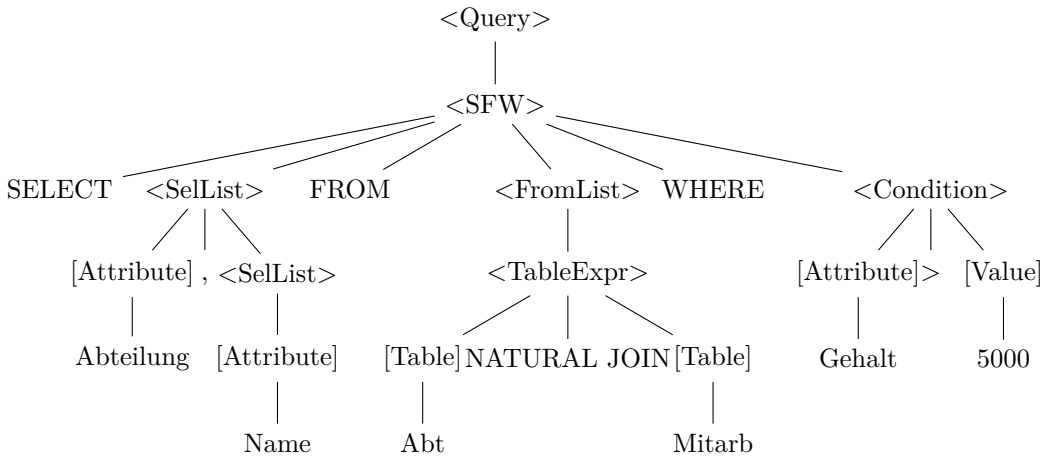

Abbildung 3: Syntaxbaum in unserem Beispiel

Schritt 2: Validierung

In der Anfragebearbeitung wird in diesem zweiten Schritt der Systemkatalog zu Hilfe genommen, um die Gültigkeit der Namen von Tabellen, Attributen, der Korrektheit von Typen usw. zu prüfen. Außerdem werden unqualifizierte Namen von Attributen „expandiert“, d.h. durch den

Namen der Tabelle ergänzt. In unserem Beispiel erhalten wir also einen Baum wie in Abb. 4.

Außerdem werden in diesem Schritt Sichten (*views*) erkannt und expandiert.

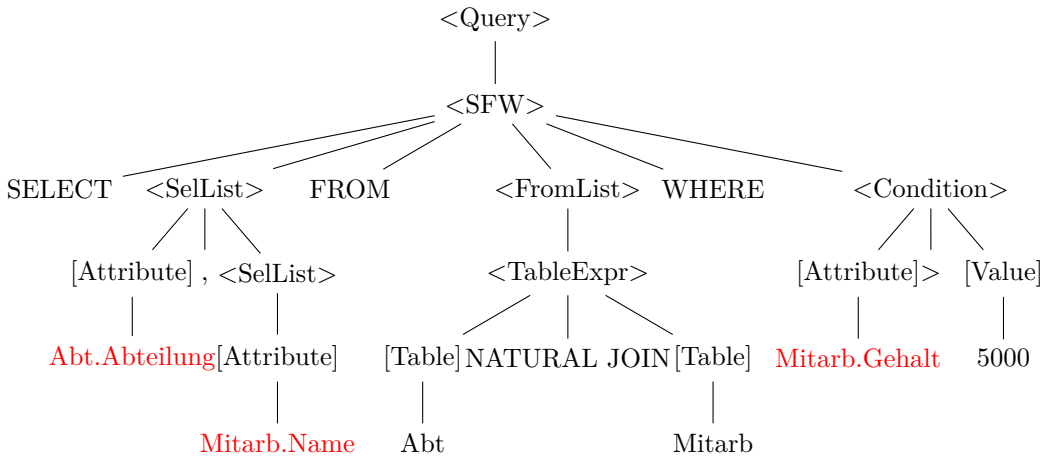

Abbildung 4: Valider Syntaxbaum in unserem Beispiel

In diesem Schritt kann auch bereits geprüft werden, ob die anfragende Transaktion überhaupt berechtigt ist, diese Anfrage zu stellen. Die Berechtigung hängt ja von den Rechten des Benutzers auf den beteiligten Tabellen ab.

Schritt 3: Initialplan erzeugen

Der Ausdruck in SQL wird in einen Syntaxbaum der relationalen Algebra transformiert. (In diesem Schritt werden insbesondere auch geschachtelte Anfragen aufgelöst.) Der initiale logische Plan ist in Abb. 5 dargestellt.

Eine wichtige Frage in diesem Schritt ist das Auflösen von geschachtelten SQL-Anweisungen. Grundsätzlich kann jede SQL-Anweisung mit geschachtelten Unteranweisungen auch als SQL-Anweisung mit Verbünden ohne Schachtelung geschrieben werden. Für das konzeptionelle Vorgehen beim „Entschachteln“ von Anfragen siehe [1, S.397ff].

Schritt 4: Anfrage restrukturieren

Der *Query Rewriter* restrukturiert die Anfrage nach gewissen heuristischen Prinzipien, indem er algebraische Umformungen vornimmt (später mehr).

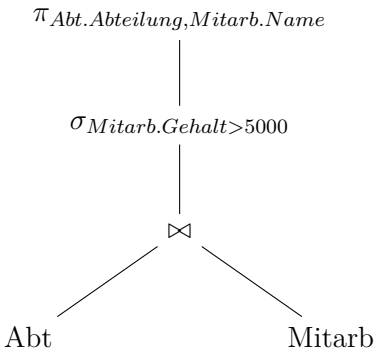

Abbildung 5: Initialer logischer Plan in unserem Beispiel

In unserem Fall wird die Restriktion $Gehalt > 5000$ möglichst früh durchgeführt (Man nennt das auch *push down* der Selektion) Das Ergebnis der Restrukturierung ist in Abb. 6 dargestellt.

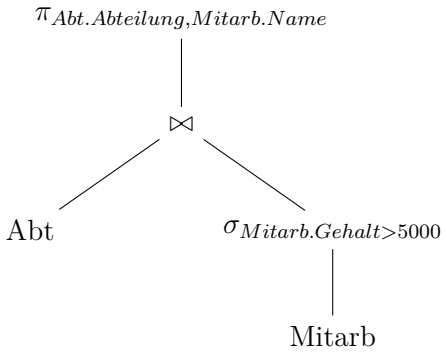

Abbildung 6: Restrukturierter logischer Plan

Schritt 5: Ausführbaren Plan erzeugen

Es wird eine Kostenschätzung durchgeführt und dann entschieden, mit welchen konkreten Mechanismen und Algorithmen auf die Tabellen tatsächlich zugegriffen wird. Das Ergebnis könnte zum Beispiel so aussehen wie in Abb. 7.

Dieser Zugriffplan wird dann in eine binäre Struktur transformiert, die den eigentlichen Zugriff schließlich steuert. Wie der eigentliche physische Zugriffsplan aussieht, hängt von der Implementierung des jeweiligen DBMS ab.

Beispiel mit PostgreSQL SQL kennt die Anweisung `explain`, die das DBMS veranlasst, den Zugriffsplan auszugeben. Wenn man im obigen

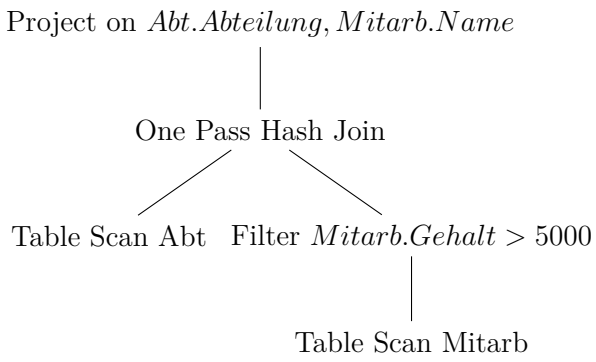

Abbildung 7: Ausführbarer Plan in unserem Beispiel

Beispiel in einer Datenbank mit wenigen Datensätzen eingibt:

```
explain select Bez, Name
  from Abt natural join Mitarb
  where Gehalt > 5000.00
```

erhält man folgendes Ergebnis von `explain`:

```
   Hash Join  (cost=18.55..34.53 rows=117 width=356)"
   Hash Cond: (mitarb.abtid = abt.abtid)"
   ->  Seq Scan on mitarb  (cost=0.00..14.38 rows=117 width=182)"
         Filter: (gehalt > 5000.00)"
   ->  Hash  (cost=13.80..13.80 rows=380 width=182)"
         ->  Seq Scan on abt  (cost=0.00..13.80 rows=380 width=182)"
```

Die Ausgabe von PostgreSQL ist so zu lesen:

Der Zugriffsplan sieht einen Hash-Verbund vor, wobei die Behälter bezüglich der `AbtId` in den beiden beteiligten Tabellen gebildet wird.

Die beiden beteiligten Tabellen sind die Restriktion von `Mitarb` mit dem Filter während des sequenziellen Scans der Tabelle sowie von `Abt`, bei der deren Einlesen eine Hash-Datenstruktur gebildet wird.

Die Angaben zu den Kosten bei den beteiligten Algorithmen bedeuten[2]:

- `cost` benennt die geschätzte Zeit für die Initalisierung sowie die Gesamtkosten der Zugriffs – die Einheit ist Zahl der Blocktransfers.
- `rows` ist die geschätzte Zahl der Datensätze (Tupel), die durch den Algorithmus geliefert werden
- `width` ist die geschätzte durchschnittliche Zahl der Bytes pro Datensatz der Ausgabe des Algorithmus.

[2] siehe Dokumentation von PostgreSQL 14.1. Using Explain

Optimierung der Datenzugriffe

Ziel des Optimierens ist es, einen Zugriffsplan zu finden, der bei gegebenem Hauptspeicher durch möglichst wenige I/O-Zugriffe die Ergebnismenge der SQL-Anweisung ermittelt.

Diese Aufgabe wird in zwei Schritte zerlegt:

- Heuristische Anfragerestrukturierung

  Die Leitfrage ist: welche algebraisch äquivalente Form der Anfrage wird voraussichtlich den optimalen Zugriffsplan ergeben? Man nennt diesen Schritt *heuristisch*, weil nur Gesetze der relationalen Algebra berücksichtigt werden, ohne den Systemkatalog (der z.B. Informationen über vorhandene Indexe enthält) oder statistische Informationen (etwa über die Größe von Tabellen oder Verteilung von Werten eines Attributs) zu berücksichtigen.

- Kostenbasierte Optimierung

  In diesem Schritt geht es darum, welche konkreten Algorithmen für die benötigen relationalen Operatoren eingesetzt werden, und wie diese Algorithmen kombiniert werden. Diesen Schritt bezeichnet man als *kostenbasiert*, weil der Systemkatalog und eventuell auch statistische Daten verwendet werden, um mögliche Vorgehensweisen mit Kostenschätzungen zu versehen und auf dieser Grundlage zu vergleichen.

Prinzipien der Anfragerestrukturierung

Prinzipien, nach denen eine Anfrage restrukturiert werden kann:

- Redundante Operationen können entfernt werden. Diese Situation tritt auf, wenn SQL-Anfragen „ungeschickt“ formuliert sind, insbesondere aber ergeben sie sich durch das Auflösen von Sichtdefinitionen.

  Beispiel:
  Sicht Manager definiert durch $\sigma_{Gehalt>5000}(Mitarb \bowtie (Abt))$
  Anfrage $\pi_{Budget}(Manager \bowtie Abt)$

  (In diesem Beispiel ist unterstellt, dass die Tabelle *Abt* ein Attribut *Budget* hat, das Budget der Abteilung.)

- Restriktionen sollten so früh wie möglich durchgeführt werden. Dazu ist es sinnvoll, konjunktive Bedingungen in Teilbedingungen aufzuteilen, die in kaskadierenden Schritten durchgeführt werden.

Man kann dann für jeden Teilschritt prüfen, ob man die Restriktion mit einem Verbund z.B. vertauschen kann (*Push down* der Selektion).

- Auch Projektionen kann man manchmal *nach unten schieben*. Dabei muss man beachten, welche Attribute für spätere Verbund-Operatoren oder Restriktionen benötigt werden, es genügt also nicht, nur die Attribute zu betrachten, die man für das Endergebnis braucht.
- Kartesische Produkte können oft durch den natürlichen Verbund oder durch einen Equi-Verbund ersetzt werden, für die effizientere Algorithmen zur Verfügung stehen.
- Duplikatelimination kann man manchmal ganz weglassen oder verschieben. Ist etwa ein Primärschlüssel vorhanden, dann kann man wissen, dass es keine Duplikate gibt und deshalb erkennen, dass δ überflüssig ist. Oder man kann die Duplikatelimination mit einem Verbund vertauschen (nicht jedoch mit einer Projektion!).

Das *Ergebnis* der heuristischen Restrukturierung ist ein optimierter logischer Plan; genauer gesagt: eine *Familie* von Plänen. Denn wir haben nicht betrachtet, in welcher Reihenfolge Verbund-Operatoren oder Restriktionen durchgeführt werden sollen. Und auch die Art, wie die Algorithmen für die relationalen Operatoren miteinander verbunden werden, ist nicht in die Optimierung eingegangen. Diese Fragestellungen führen uns zur *kostenbasierten* Optimierung.

Kostenbasierte Optimierung

Bei der kostenbasierten Optimierung ziehen wir Informationen aus dem Systemkatalog und auch Statistikdaten zu Rate.

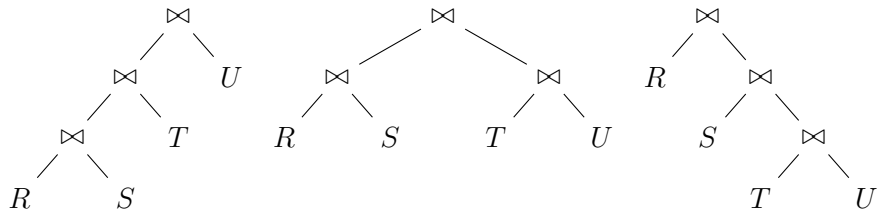

Abbildung 8: Varianten bei der Kombination von Joins

Was muss entschieden werden?

- Die *Reihenfolge*, in der Verbundoperationen, Vereinigungen, Durchschnitte und kaskadierende Restriktionen (bei allen gilt das Assoziativgesetz) durchgeführt werden, muss entschieden werden.

  Wir haben gesehen, dass bei einem Verbund z.B. es eine Rolle spielt, welche der beiden Relationen kleiner ist, bzw. ob eine der beiden Relationen komplett in den Hauptspeicher passt. Hat man nun mehrere Relationen, dann hat man mehr Möglichkeiten.

  Im Beispiel von $R \bowtie S \bowtie T \bowtie U$ kann man z.B. nicht nur die Reihenfolge der Operatoren verändern, sondern auch bei gegebener Reihenfolge hat man die Varianten in Abb. 8. Häufig verwenden DBMS nur die Variante des *left-deep tree*s.

- Für jeden Operator muss ein geeigneter Algorithmus gewählt werden.

  Wie wir im vorherigen Kapitel gesehen haben, bestehen große Unterschiede im I/O-Verhalten von verschiedenen Algorithmen je nach der konkreten Situation wie der Existenz von Indexen, der Eindeutigkeit von Attributen, der Größe von Tabellen, eventuell aber auch der Verteilung der Werte eines Attributs. All solche Faktoren kann ein Optimierer für die Wahl des Algorithmus in Betracht ziehen.

- Man benötigt zusätzliche Algorithmen, die für den physischen Zugriffsplan benötigt werden, im logischen Plan aber nicht vorkommen.

  Insbesondere muss entschieden werden, wie auf die eigentlichen Tabellen zugegriffen wird: Tabellen-Scan, Index-Scan oder Sortier-Scan. Auch hierfür spielt es eine wichtige Rolle, ob ein Index vorhanden ist und inwiefern er für den Gesamtalgorithmus nutzbringend eingesetzt werden kann.

- Schließlich muss entschieden werden, wie die Algorithmen ineinandergreifen, also wann Pipelining verwendet wird und wann Zwischenergebnisse tatsächlich als Ganzes „materialisiert“ werden.

  Wir haben bereits das Volcano-Iterator-Konzept für das Pipelining diskutiert und so scheint es offensichtlich, dass man stets Pipelining verwendet. Man muss jedoch berücksichtigen, dass sich bei Pipelining die verschiedenen Algorithmen den Hauptspeicher teilen und deshalb manchmal ein optimaleres Ergebnis erzielt wird, wenn Zwischenergebnisse materialisiert werden.

Literaturverzeichnis

[1] Gunter Saake, Andreas Heuer, Kai-Uwe Sattler. *Datenbanken: Implementierungstechniken.* Heidelberg, 2011.

Burkhardt Renz
TH Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.2 – 5. Mai 2014