

Studiengang: Informatik
Prüfer: Prof. Dr. B. Mitschang
Betreuer: Dipl. Inform. Holger Schwarz

begonnen am: 1. August 2001

beendet am: 11. April 2002

CR-Klassifikation: H.2.4, H.4.2

Diplomarbeit Nr. 1967

Rewrite-Strategien für generierte Abfragesequenzen im Online Analytical Processing

Tobias Kraft

Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Gliederung	2
2	Definitionen	4
2.1	Anfrage	4
2.2	Anfragesequenz	8
3	Single-Query-Rewrite	10
3.1	SQL (als Zeichenkette)	10
3.2	QGM	11
3.3	Relationenalgebra	14
3.4	GENTREES	15
3.5	COKO-KOLA	17
4	Multi-Query-Rewrite	20
4.1	Regeln	20
4.1.1	Regeln der Klasse 1	22
4.1.2	Regeln der Klasse 2	33
4.1.3	Regeln der Klasse 3	38
4.1.4	Erweiterung der Regelmenge	40
4.2	Interndarstellung	40
4.2.1	Bewertung der vorgestellten Interndarstellungen	41
4.2.2	Eigene XML-Interndarstellung	44
5	Der Prototyp	50
5.1	Bewertung einiger XML-Technologien	50
5.1.1	XSLT	50
5.1.2	XQuery	51
5.1.3	DOM	52
5.2	Beschreibung der Regel-Implementierungen	52
5.2.1	Merge Select	53
5.2.2	Where To Group	61
5.2.3	Predicate Pushdown	65
5.2.4	Concat Queries	68
5.2.5	Eliminate Redundant References	69
5.3	Das Regelsystem und mögliche Erweiterungen	72

5.4 Ein Anwendungsbeispiel	72
6 Zusammenfassung und Ausblick	76

Kapitel 1

Einleitung

1.1 Motivation

Im Bereich Business Intelligence werden verschiedenste Methoden und Werkzeuge eingesetzt, um unternehmensweite Datenbestände in einem Data Warehouse zu analysieren. Wichtige Vertreter sind hier unter anderem das Online Analytical Processing (OLAP) und das Data Mining. Viele Werkzeuge in diesem Bereich generieren zur Bearbeitung einer einzigen Anfrage des Benutzers eine ganze Sequenz von SQL-Anfragen. Die Ergebnisse der einzelnen SQL-Statements gehen entweder direkt in die für den Benutzer aufbereiteten Ergebnisse ein oder sie dienen lediglich als temporäre Zwischenergebnisse.

<pre>INSERT INTO A1 (orderyearkey, ordermonthkey, partkey, sumquantity) SELECT od.orderyearkey, od.ordermonthkey, lo.partkey, SUM(lo.quantity) FROM lineitem.orders lo, orderday od WHERE od.orderdate = lo.orderdate AND od.ordermonthkey IN (199401, 199402) GROUP BY od.orderyearkey, od.ordermonthkey, lo.partkey;</pre>
<pre>INSERT INTO A2 (ordermonthkey, partkey, sumquantity) SELECT od.ordermonthkey, lo.partkey, SUM(lo.quantity) FROM lineitem.orders lo, orderday od WHERE od.lastmonthdate = lo.orderdate AND od.ordermonthkey IN (199401, 199402) GROUP BY od.ordermonthkey, lo.partkey;</pre>
<pre>INSERT INTO A3 (ordermonthkey, ordermonthname, orderyearkey, orderyear, partkey, partname, sumquantity, lmsumquantity, incrquantity, incrquantity2) SELECT om.ordermonthkey, om.ordermonthname, oy.orderyearkey, oy.orderyear, pa.partkey, pa.partname, A1.sumquantity, A2.sumquantity, A1.sumquantity - A2.sumquantity, (A1.sumquantity - A2.sumquantity) / A2.sumquantity FROM A1, A2, ordermonth om, orderyear oy, part pa WHERE A1.ordermonthkey = A2.ordermonthkey AND A1.partkey = A2.partkey AND A1.ordermonthkey = om.ordermonthkey AND A1.orderyearkey = oy.orderyearkey AND A1.partkey = pa.partkey;</pre>
<pre>INSERT INTO A4 (ordermonthkey, ordermonthname, orderyearkey, orderyear, partkey, partname, sumquantity, lmsumquantity, incrquantity, incrquantity2) SELECT A3.ordermonthkey, A3.ordermonthname, A3.orderyearkey, A3.orderyear, A3.partkey, A3.partname, A3.sumquantity, A3.lmsumquantity, A3.incrquantity, A3.incrquantity2 FROM A3 WHERE A3.incrquantity2 >= 0.98;</pre>

Abbildung 1.1: Beispiel für eine Anfragesequenz [SWM01].

Abbildung 1.1 [SWM01] zeigt ein Beispiel für eine Anfragesequenz, die mit den DSS-Tools

von MicroStrategy (www.strategy.com) generiert wurde. Die Abfragesequenz beantwortet die folgende Frage: “Welches sind die Produkte, von denen im Januar und Februar 1994 98% mehr verkauft wurden als im Vormonat?” Sie arbeitet auf einer modifizierten Version des TPC-H-Schemas [Wagn00]. Es sind nur die INSERT-Anweisungen aufgeführt; die Anweisungen zur Erzeugung und zur Löschung der Tabellen sind aus Gründen der Übersichtlichkeit nicht abgedruckt. Es handelt sich bei diesem Beispiel um eine recht kurze und einfache Abfragesequenz; es gibt aber auch Fragestellungen, die zu viel komplexeren Abfragesequenzen, sowohl bezüglich der Zahl der Anfragen als auch bezüglich der Abhängigkeitsbeziehungen, führen (siehe Fragestellung C in [SWM01]).

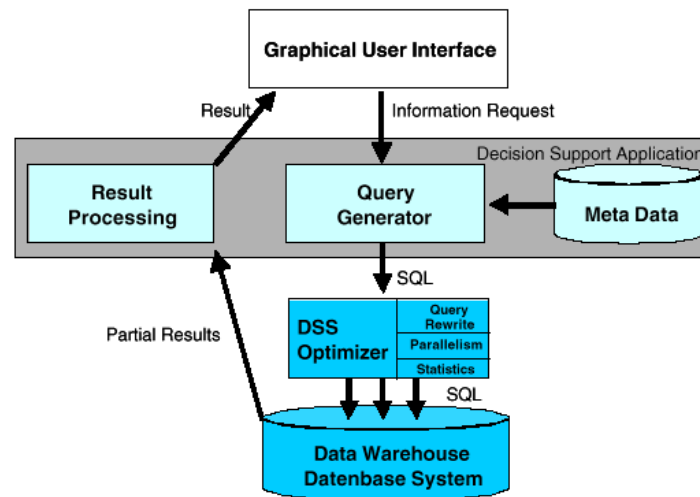


Abbildung 1.2: Architektur mit DSS-Optimierer [SWM01]

Generierte Abfragesequenzen sind meist nicht optimal, sondern spiegeln den generischen, modularen Aufbau des Generierungsprozesses wider. Dies hat hohe Laufzeiten und eventuell auch einen hohen Platzverbrauch für die Speicherung der Zwischenergebnisse zur Folge. Ähnlich wie bei der Optimierung einzelner SQL-Anfragen können hier Rewrite-Strategien eingesetzt werden, um eine gegebene Abfragesequenz in eine optimierte Version zu transformieren. Die Analyse und Formalisierung der möglichen Transformationen, sowie die Implementierung eines Prototyps sind Bestandteil dieser Diplomarbeit. Der Prototyp soll belegen, dass die aufgestellten Regeln tatsächlich implementiert werden können und diese Implementierung auch effizient ist. Der DSS-Optimierer [SWM01], in dem zukünftig die Transformationen durchgeführt werden sollen, ist zwischen der DSS-Anwendung und dem Datenbanksystem positioniert (siehe Abbildung 1.2). D.h. sowohl seine Eingabe als auch seine Ausgabe ist eine Sequenz von SQL-Anweisungen. Dies muss bei der Suche nach einer geeigneten Interndarstellung für die Abfragesequenzen und bei der Formulierung der Rewrite-Regeln berücksichtigt werden.

1.2 Gliederung

In Kapitel 2 werden die Begriffe “Anfrage” bzw. “Abfragesequenz” genau definiert. Außerdem wird eine Untermenge von SQL92 gebildet, mit der später die Interndarstellung und der Prototyp arbeiten.

In Kapitel 3 werden einige Ansätze, d.h. Interndarstellungen und dazugehörige Regelnnotationen, aus der klassischen Anfragerestrukturierung, die sich mit der Optimierung einzelner SQL-Anfragen beschäftigt, vorgestellt.

Kapitel 4 konzentriert sich auf die Optimierung von Anfragesequenzen. Zuerst werden die im Rahmen dieser Diplomarbeit erstellten Rewrite-Regeln vorgestellt. Danach wird eine dazu passende, auf XML basierende Interndarstellung für Anfragesequenzen entwickelt.

Kapitel 5 beschäftigt sich mit der Implementierung der Regeln und dem im Rahmen dieser Diplomarbeit erstellten Prototyp. Zuerst werden einige XML-Technologien auf ihre Verwendbarkeit zur Implementierung der Regeln überprüft. Danach werden für die im Prototyp implementierten Regeln die wichtigsten Algorithmen anhand von Codeausschnitten erklärt. Anschließend wird kurz die im Prototyp implementierte Kontrollstrategie für das Regelsystem, das für die Auswahl der auszuführenden Regel verantwortlich ist, beschrieben und erläutert welche weiteren Kontrollstrategien es gibt. Zuletzt folgt noch ein kleines Beispiel, das die Anwendung des Prototyps zeigt.

Abschließend werden in Kapitel 6 die wichtigsten Aspekte dieser Arbeit zusammengefasst und Anregungen für weitere Forschungsarbeiten gegeben.

Kapitel 2

Definitionen

Eine Anfragesequenz dient der Beantwortung einer meist recht komplexen Fragestellung. Die Berechnung des Ergebnisses erfolgt daher schrittweise. Eine einzelne Anfrage der Anfragesequenz spiegelt einen solchen Berechnungsschritt wider. Folglich ist das Ergebnis einer Anfrage ein Zwischenergebnis der Anfragesequenz, das als Eingabe für eine oder mehrere weitere Anfragen der Anfragesequenz dienen kann und daher temporär oder im Falle des Endergebnisses sogar permanent zwischengespeichert wird. Somit entsteht eine Abhängigkeitsbeziehung zwischen den einzelnen Anfragen der Anfragesequenz.

Dies ist eine recht grobe und intuitive Definition des Begriffs “Anfragesequenz”. Da die genaue Bedeutung der Begriffe “Anfrage” und “Anfragesequenz” jedoch für die nachfolgenden Kapitel von Relevanz ist, werden diese Begriffe im folgenden noch detaillierter definiert. Im anschließenden Abschnitt wird zunächst erläutert, was innerhalb dieser Diplomarbeit unter einer Anfrage als Komponente einer Anfragesequenz verstanden wird und aus welchen Komponenten eine solche Anfrage besteht. Anschließend wird erläutert, was innerhalb dieser Diplomarbeit unter einer Anfragesequenz verstanden wird.

2.1 Anfrage

Eine Anfrage im Kontext der Anfragesequenz ist eine Folge von SQL-Anweisungen, die eine Tabelle oder Sicht erzeugen, diese mit Tupeln füllen und, falls sie nicht mehr benötigt wird, wieder löschen.

Um die Komplexität der möglichen Anfragekonstrukte zu begrenzen und damit die Implementierung eines Prototypen zu vereinfachen, beschränke ich mich bei der Formulierung der SQL-Anweisungen zunächst auf eine Untermenge von SQL92 [DD97]. Hierbei wird auf die Möglichkeit zur Nestung von Anfragen durch Unteranfragen verzichtet, da die Auflösung von Nestungen Aufgabe des Single-Query-Rewrite ist und unabhängig vom Multi-Query-Rewrite schon im voraus durchgeführt werden kann. Auch Mengenoperationen, wie z.B. UNION, wurden außer acht gelassen. Es wird außerdem vorausgesetzt, dass WHERE- und HAVING-Klausel schon in konjunktiver Normalform (KNF) vorliegen. Dies stellt jedoch keine große Einschränkung dar, da jede WHERE- und HAVING-Klausel durch entsprechende logische Transformationen auf KNF gebracht werden kann. Diese Normalisierung kann ebenfalls vor dem Multi-Query-Rewrite durchgeführt werden. Es wird angenommen, dass alle Attribute, nach denen in der GROUP BY-Klausel gruppiert wird, auch in der SELECT-Klausel aufgeführt sind. Anfragen, die einen skalaren Aggregatwert berechnen und deren

GROUP BY-Klausel somit leer ist, sind nicht zugelassen. Die Verwendung des Sterns “*” ist nur innerhalb der Aggregatfunktion COUNT erlaubt, aber nicht zur Totalprojektion in der SELECT-Klausel. Um die Handhabung der Attributbezeichner zu vereinfachen und die Verständlichkeit der SQL-Ausdrücke zu erhöhen ist jedem Attributbezeichner ein zum SQL-Standard konformes Präfix voranzustellen, das den Bezeichner der zugehörigen Tabelle oder Sicht identifiziert. Bei den Integritätsbedingungen habe ich mich auf “PRIMARY KEY” (Primärschlüssel), “UNIQUE” (eindeutig) und “NOT NULL” (nicht null) beschränkt, wobei eine sinnvolle Vergabe dieser Attributeigenschaften vorausgesetzt wird. Ein implizites Type-Casting ist ebenfalls zu unterlassen, z.B. dürfen einem Term, der in mehreren SELECT-Klauseln enthalten ist, nicht unterschiedliche Datentypen zugewiesen werden.

Die für die Diplomarbeit relevanten SQL-Anweisungen sind in Form von Syntax-Diagrammen in Abbildung 2.1 und 2.2 dargestellt. Als Bezeichner und als Datentyp ist alles erlaubt, was dem SQL92-Standard [DD97] entspricht.

Grundsätzlich gibt es zwei verschiedene Typen von Anfragen innerhalb einer Anfragesequenz. Zum einen Anfragen vom Typ “Tabelle”, die ihre Ausgabe in einer Tabelle ablegen, zum anderen Anfragen vom Typ “Sicht”, die ihre Ausgabe als Sicht definieren. Für Anfragen vom Typ “Tabelle” gilt:

- Die Tabelle wird mit einer CREATE TABLE-Anweisung (siehe Abb. 2.2) erstellt.
- An dieser Tabellendefinition, d.h. den zur Tabelle gehörenden Metadaten, wird während der gesamten Lebenszeit der Tabelle nichts mehr verändert.
- Durch einmalige Ausführung einer INSERT INTO-Anweisung (siehe Abb. 2.2) wird die Tabelle mit den Tupeln der Anfrage gefüllt. Die eigentliche Anfrage, d.h. die SELECT-Anweisung, bildet hierbei den Körper der INSERT INTO-Anweisung.
- Sobald die Tabelle nicht mehr für die Berechnung anderer Anfragen benötigt wird bzw. spätestens nach Erstellung aller Ergebnistabellen und -sichten wird die Tabelle mit einer DROP TABLE-Anweisung (siehe Abb. 2.2) wieder entfernt.
- Alle weiteren Operationen auf dieser Tabelle sind Leseoperationen, die der Berechnung weiterer Anfragen der Anfragesequenz dienen.

Für Anfragen vom Typ “Sicht” gilt:

- Die Sicht wird mit einer CREATE VIEW-Anweisung (siehe Abb. 2.2) erstellt. Die eigentliche Anfrage, d.h. die SELECT-Anweisung, bildet hierbei den Körper der CREATE VIEW-Anweisung.
- An dieser Sichtdefinition, d.h. den zur Sicht gehörenden Metadaten, wird während der gesamten Lebenszeit der Sicht nichts mehr verändert.
- Sobald die Sicht nicht mehr für die Berechnung anderer Anfragen benötigt wird bzw. spätestens nach Erstellung aller Ergebnistabellen und -sichten wird die Sicht mit einer DROP VIEW-Anweisung (siehe Abb. 2.2) wieder entfernt.
- Alle weiteren Operationen auf dieser Tabelle sind Leseoperationen, die der Berechnung weiterer Anfragen der Anfragesequenz dienen.

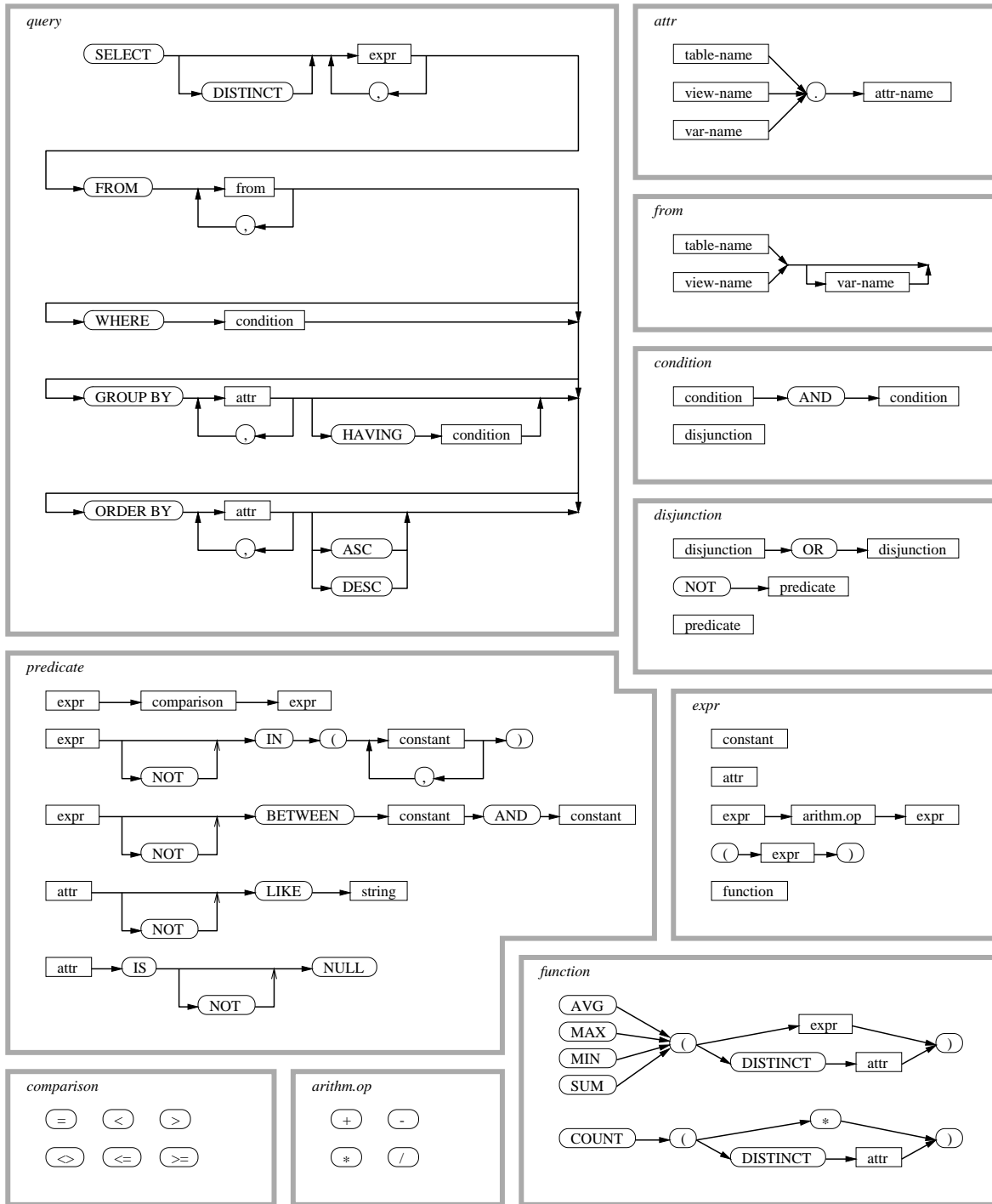


Abbildung 2.1: Syntaxdiagramme für eine Untermenge von SQL92 - Teil 1

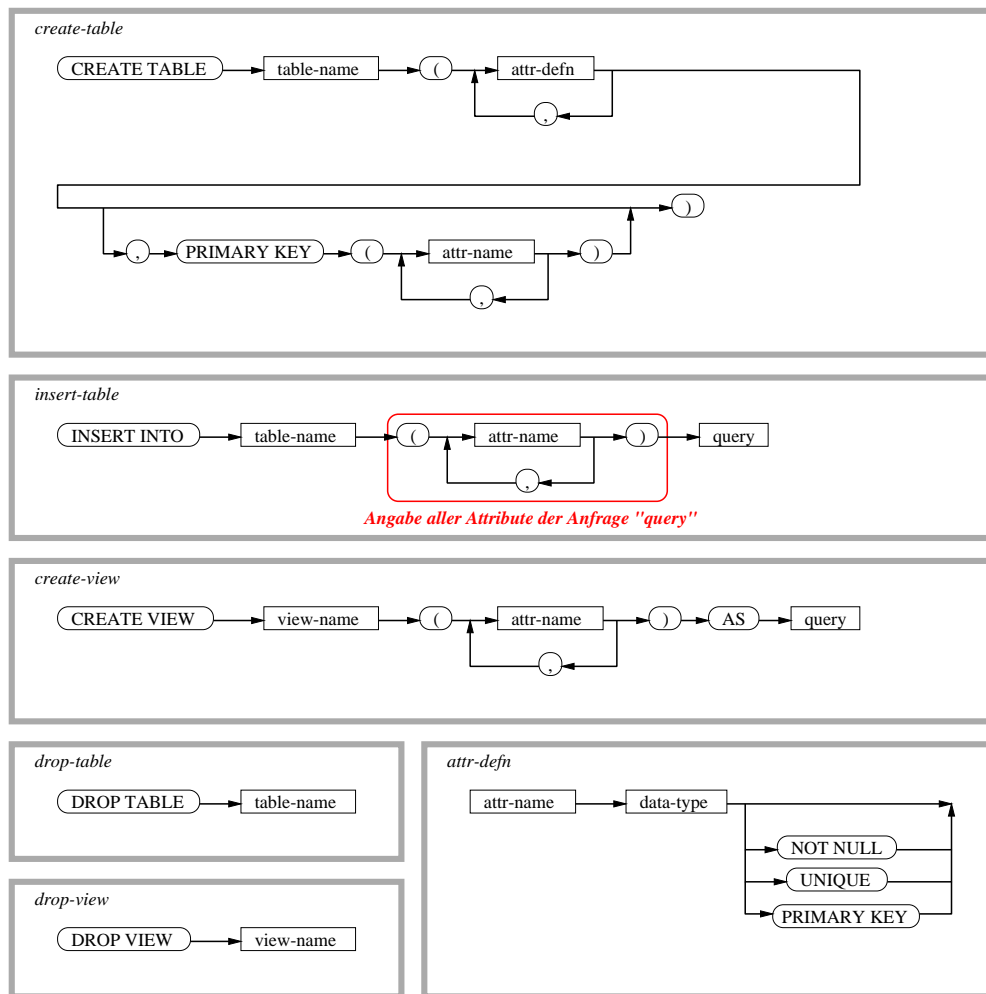


Abbildung 2.2: Syntaxdiagramme für eine Untermenge von SQL92 - Teil 2

Es ist bei Anfragen beiden Typs zu beachten, dass die SQL-Anweisungen in der Reihenfolge ausgeführt werden, in der sie aufgelistet sind.

Ein Sonderfall von Anfragen sind die Ergebnisanfragen. Sie legen ebenfalls ihre Ausgabe in einer Tabelle bzw. Sicht ab und sind Teil der Anfragesequenz. Der Unterschied besteht jedoch darin, dass ihre Tabelle bzw. Sicht das Endergebnis bzw. einen Teil des Endergebnisses der Anfragesequenz enthält und damit deren Inhalt auch nach der Ausführung der Anfragesequenz noch verfügbar sein muss. Daher gehört die Entfernung der Tabelle bzw. Sicht einer Ergebnisanfrage nicht zu den Operationen der Anfragesequenz bzw. nicht zu den SQL-Anweisungen der zugehörigen Anfrage. Solange nicht alle Operationen der Anfragesequenz ausgeführt wurden, dürfen keine anderen Operationen auf der Tabelle bzw. Sicht einer Ergebnisanfrage ausgeführt werden. D.h. erst nachdem alle Berechnungen der Anfragesequenz durchgeführt wurden, können Operationen die nicht zur Anfragesequenz gehören, z.B. das Auslesen einer Ergebnistabelle durch eine Anwendung, ausgeführt werden.

Genügt eine Anfrage nicht diesen Anforderungen, so ist sie keine Anfrage im Sinne der Definition von Anfragesequenzen und daher auch nicht Teil einer Anfragesequenz.

2.2 Anfragesequenz

Zwischen den Anfragen einer Anfragesequenz bestehen Abhängigkeiten in der Form, dass eine Anfrage auf die Ergebnisse anderer Anfragen in der Anfragesequenz zugreift. Eine Anfrage kann dabei sowohl von mehreren anderen Anfragen abhängig sein als auch mehrere Anfragen können von derselben Anfrage abhängig sein, wodurch eine gewisse Parallelität der Teilsequenzen einer Anfragesequenz ermöglicht wird. Zusätzlich kann eine Anfrage auch von Tabellen und Sichten außerhalb der Anfragesequenz, sogenannten Basistabellen und -sichten, abhängig sein. Am Ende einer Anfragesequenz stehen eine oder mehrere Tabellen bzw. Sichten, die das Endergebnis beinhalten und von der Anwendung, die die Anfragesequenz generiert hat, gelesen werden. Stellt man sich die Anfragen als Knoten vor und die Abhängigkeitsbeziehungen als gerichtete Kanten so erhält man einen gerichteten Graph.

Für einen Graph, der eine Anfragesequenz bzw. die Abhängigkeiten zwischen den Anfragen der Sequenz repräsentiert, gilt:

- Jeder Knoten ist eine Anfrage nach der Definition in Abschnitt 2.1.
- Es führt genau dann eine gerichtete Kante von Knoten Q_a nach Knoten Q_b , wenn Q_b eine Anfrage ist, deren zugehörige Sichtdefinition bzw. Tabelleninhalt von der zu Anfrage Q_a gehörenden Sichtdefinition bzw. Tabelleninhalt abhängt, d.h. wenn die zu Q_b gehörende Sichtdefinition bzw. Tabellenfülloperation in ihrer FROM-Klausel die zu Q_a gehörende Sicht bzw. Tabelle referenziert.
- Jeder Knoten kann mehrere Eingangskanten und mehrere Ausgangskanten besitzen.
- Von jedem Nicht-Ergebnisknoten führt ein gerichteter Weg zu einem Ergebnisknoten.
- Es existieren keine Schleifen in diesem Graph.
- Lässt man die Richtung der Kanten außer acht, so ist jeder Knoten mit jedem anderen Knoten der Anfragesequenz über einen (ungerichteten) Weg verbunden, d.h. der Graph ist zusammenhängend.
- Der Bezeichner einer Anfrage, genauer der Bezeichner der zugehörigen Tabelle oder Sicht, ist eindeutig innerhalb der Anfragesequenz. D.h. derselbe Bezeichner wird nicht (zeitlich versetzt) für mehrere Anfragen innerhalb einer Anfragesequenz verwendet.

Genügt eine Menge von Anfragen nicht diesen Anforderungen, so ist sie in diesem Sinne keine Anfragesequenz.

Die einzelnen SQL-Anweisungen der Anfragen einer Anfragesequenz können in beliebiger Reihenfolge ausgeführt werden, d.h. es ist auch eine verzahnte Ausführung von SQL-Anweisungen verschiedener Anfragen möglich, solange die folgenden Bedingungen erfüllt sind:

- Innerhalb einer Anfrage wird die in Abschnitt 2.1 beschriebene Ausführungsreihenfolge der SQL-Anweisungen beibehalten.
- Eine Anfrage muss ihre Berechnungen abgeschlossen, d.h. ihre Tabelle bzw. Sicht gefüllt haben, bevor eine andere Anfrage auf deren Ergebnisse zugreift.
- Die Tabelle bzw. Sicht einer Anfrage darf erst gelöscht werden, wenn nachfolgend keine andere Anfrage der Anfragesequenz mehr auf sie zugreift.

Werden diese Bedingungen nicht eingehalten, kann es zu Zugriffen auf noch nicht gefüllte Tabellen oder auf noch nicht oder nicht mehr existierende Sichten und Tabellen kommen.

Kapitel 3

Single-Query-Rewrite

Das Query-Rewrite, zu Deutsch “Anfragerestrukturierung”, ist Teil der Optimierung innerhalb eines DBMS und der Anfragetransformation vorgelagert. Vor der Restrukturierung muss die gegebene Anfrage in eine logische Interndarstellung überführt werden, auf die dann die Restrukturierungsregeln angewandt werden können. Da die Restrukturierung ausschließlich auf logischer Ebene arbeitet und daher auch meist keine Kostenabschätzung stattfindet, handelt es sich bei den Restrukturierungsregeln um Heuristiken, d.h. Daumenregeln, die im allgemeinen zu einer Verbesserung der Ausführungszeiten führen.

In den heute verfügbaren DBMS werden Anfragen einzeln und unabhängig voneinander optimiert, ohne Ähnlichkeiten und Zusammenhänge zwischen mehreren aufeinanderfolgenden Anfragen zu berücksichtigen. Typische Anwendungsbeispiele sind: das Auflösen einer Nestung (flattening), das Auflösen von Sichten, das Umwandeln von Quantoren, Mengenoperationen und Pfadausdrücken in Join-Operationen, die Duplikatbehandlung und die Verschiebung von Prädikaten und Projektionen. Diese Form der Restrukturierung, wird im folgenden als “Single-Query-Rewrite” bezeichnet. Es werden nun einige Ansätze, d.h. Interndarstellungen und die zugehörigen Regelnotationen, hierzu betrachtet.

3.1 SQL (als Zeichenkette)

Eine relativ simple Form der Interndarstellung ist die Repräsentation einer Anfrage als nicht geparste Zeichenkette. D.h. der SQL-Ausdruck wird als solcher belassen und mit Hilfe von Pattern Matching und Zeichenkettenfunktionen analysiert und manipuliert. Für die Definition von Pattern können reguläre Ausdrücke verwendet werden.

Beim Single-Query-Rewrite haben die zugehörigen Regeln die Form $L \Rightarrow R$. L steht für ein Pattern, das bei der Anwendung der Regel durch das Pattern R ersetzt wird. Zuerst wird geprüft, ob L auf die gegebene Anfrage, eine Unteranfrage oder einen Ausdruck darin passt. Falls ja, dann wird die Substitution θ ermittelt, die festlegt, wie die Variablen in L durch die entsprechenden Ausprägungen in der Anfrage zu ersetzen sind. Nun wird in der Anfrage $L\theta$ durch $R\theta$ ersetzt. Handelt es sich um eine korrekte Regel, so wird damit eine Anfrage in eine zu ihr semantisch äquivalente Anfrage überführt. In [FRV95] sind einige Regeln und der zugehörige Matching-Algorithmus für ein Single-Query-Rewrite von OQL-Anfragen beschrieben. Abbildung 3.1 zeigt ein Regelbeispiel hieraus, in dem eine Anfrage mit einer Unteranfrage in der FROM-Klausel verschmolzen wird.

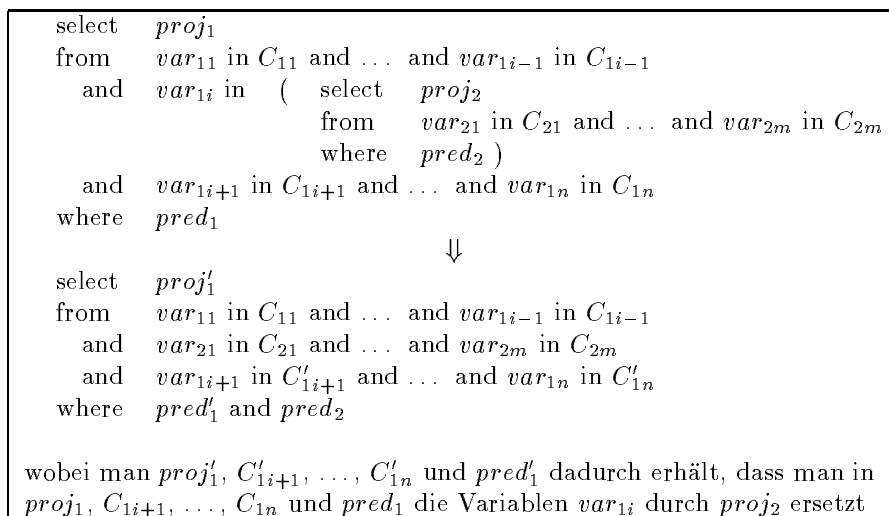


Abbildung 3.1: Beispiel einer Single-Query-Rewrite-Regel für OQL [FRV95].

3.2 QGM

Das Query Graph Model [HFLP89] [PHH92] [PLH97] [M95], kurz QGM, ist ein von IBM entwickeltes Darstellungsschema für SQL-Anfragen und wird in Starburst und DB2 im Rahmen der Anfragerestrukturierung und Anfragetransformation eingesetzt. Es handelt sich hierbei um eine Mischung aus Operatorgraph und Objektgraph. D.h. ein QGM-Graph ist ein Graph dessen Knoten Operatoren sind und dessen Kanten den Datenfluss beschreiben. Die Operatoren werden hier als Tabellenoperatoren bezeichnet, da sowohl ihre Eingabe als auch ihre Ausgabe aus Tabellen besteht. Sie beschreiben wie aus der Eingabetabelle die Ausgabetafel berechnet wird, was wiederum als Objektgraph dargestellt ist. Die Objekte innerhalb eines solchen Tabellenoperators stehen für die einzelnen Eingabequellen und werden hier als Tupelvariablen bezeichnet; die Kanten stellen die Beziehungen zwischen den Tupelvariablen dar.

Am besten lässt sich QGM an einem kleinen Beispiel erklären. Gegeben sei die folgende SQL-Anfrage [PHH92]:

```

SELECT DISTINCT q1.partno, q1.descr, q2.suppno
FROM      inventory q1, quotations q2
WHERE     q1.partno=q2.partno AND
           q1.descr='engine' AND
           q2.price ≤ ALL
           ( SELECT   q3.price
             FROM     quotations q3
             WHERE    q2.partno=q3.partno)

```

Diese Anfrage liefert für jedes Teil den/die billigsten Lieferanten. Der dazugehörige QGM-Graph ist in Abbildung 3.2 dargestellt. Die Tabellenoperatoren sind hier als Boxen abgebildet. Box (1) und (2) sind Tabellenoperatoren vom Typ ACCESS und repräsentieren jeweils den Zugriff auf eine Basistabelle. Box (3) und (4) sind Tabellenoperatoren vom Typ SELECT und umfassen die Operationen Selektion, Projektion und Join. Box (3) beschreibt hierbei den äußeren Teil der Anfrage und Box (4) den inneren Teil, also die Unteranfrage. QGM bietet

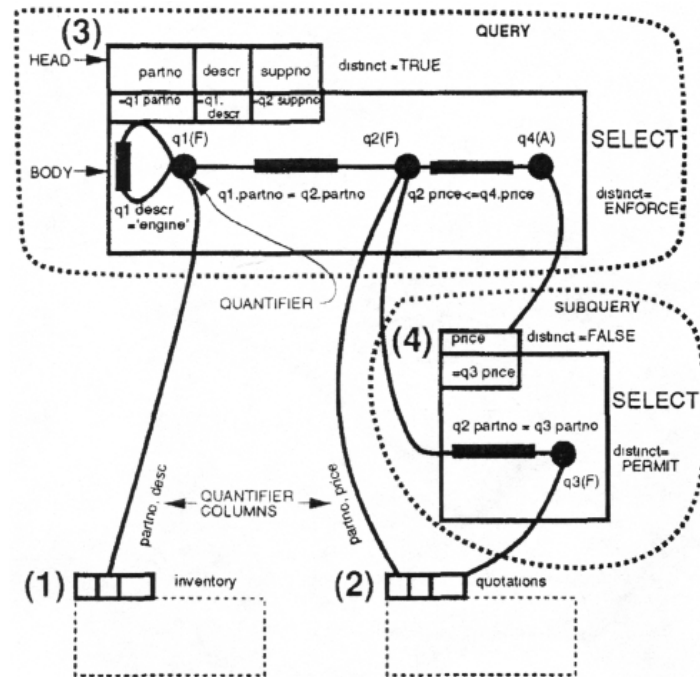


Abbildung 3.2: Beispiel für einen QGM-Graph [PHH92]

noch eine Menge weiterer Tabellenoperator-Typen an, z.B. `GROUP BY` für die Gruppierung und Aggregation oder `UNION` für die Vereinigung von Tabellen. Jede Box hat einen Kopf, der die Spalten der Ausgabetable spezifiziert und somit der `SELECT`-Klausel entspricht. Der Rumpf enthält bei allen Tabellenoperatoren einen Graphen mit Ausnahme der Tabellenoperatoren vom Typ `ACCESS`, bei denen der Rumpf leer ist. Die Knoten dieses Graphen repräsentieren quantifizierte Tupelvariablen. q_1 , q_2 und q_3 entsprechen den Tabellenreferenzen der `FROM`-Klausel und sind daher vom Typ *F* (FOR EACH). q_4 repräsentiert das universelle Mengenprädikat `ALL` aus der `WHERE`-Klausel und ist daher vom Typ *A*. Die existentiellen SQL-Mengenprädikate `EXISTS`, `IN`, `ANY` und `SOME` resultieren in einer Tupelvariable vom Typ *E*. Die Kanten zwischen den Tupelvariablen repräsentieren Prädikate, die Attribute dieser Tupelvariablen betreffen. Eine Schleife wie bei q_1 steht für ein lokales Prädikat, d.h. ein Prädikat, das nur eine Tupelvariable betrifft. Jede Komponente aus der Konjunktion der `WHERE`-Klausel wird auf eine solche Kante abgebildet. Jede Tupelvariable ist außerdem über eine Definitionskante mit der zugehörigen Eingabetabelle verbunden. Für die Duplikatbehandlung hat jeder Kopf ein boolesches Attribut *distinct*, das angibt, ob die Ausgabetable Duplikate enthält (*FALSE*) oder nicht (*TRUE*); zusätzlich hat jeder Tabellenoperator und jede Tupelvariable ein Attribut *distinct*, das die Werte *ENFORCE*, *PRESERVE* und *PERMIT* annehmen kann, wobei ich auf die Bedeutung dieser Werte hier nicht genauer eingehen möchte.

Für die Realisierung eines Regelinterpreters erschien eine fest vorgegebene Regelbeschreibungssprache als nicht ausreichend, da die Regeln trotz ihrer Lokaltätseigenschaft eine zum Teil recht hohe Komplexität aufweisen [PLH97]. Diese Feststellung stützt sich auf die Erfahrungen mit dem DBMS Exodus [CD89], das auf einer Regelsprache basiert. Daher sind in Starburst der Bedingungs- und der Aktionsteil einer Regel als Funktionen in C bzw. C++

kodiert. Diese Funktionspaare werden als vorcompilierter Code bereitgestellt und müssen nicht wie bei anderen Regelsystemen interpretiert werden. Als Beispiel sei die folgende, relativ einfache Regel *Fusion* gegeben, die das Verschmelzen zweier SELECT-Tabellenoperatoren, die über eine F-quantifizierte Tupelvariable und Definitionskante miteinander verbunden sind, beschreibt (TO = Tabellenoperator):

IF (TO-oben ist SELECT-TO mit F-quantif. Tupelvariable Tvar-F,
Tvar-F referenziert TO-unten,
TO-unten ist SELECT-TO)

THEN Rumpf-Kopieren (von: TO-unten, nach: TO-oben),
Kopf-Kopieren (von: TO-unten, nach: TO-oben),
Prädikatskanten-Kopieren (von: TO-unten, nach: TO-oben),
Prädikatskanten-Anpassen (Tvar-F),
Def-Kante-Anpassen (Tvar-F),
TO-Löschen (Tvar-F)

Ein Beispiel für die Anwendung dieser Regel ist in Abbildung 3.3 dargestellt. Eine ausführliche Erläuterung dieser und anderer Regeln ist in [M95] zu finden. Wie zu sehen ist, werden im Bedingungs- und Aktionsteil der Regel Tabellenoperator-Funktionen benutzt, die von QGM zur Verfügung gestellt werden. Das Regelsystem erlaubt eine Gruppierung der Restrukturierungsregeln in Regelklassen, wobei jeder Regelklasse eine lokale Konfliktlösungsstrategie zugewiesen werden kann. Als Problemlösungsstrategie wird die Vorwärtsverkettung angewandt.

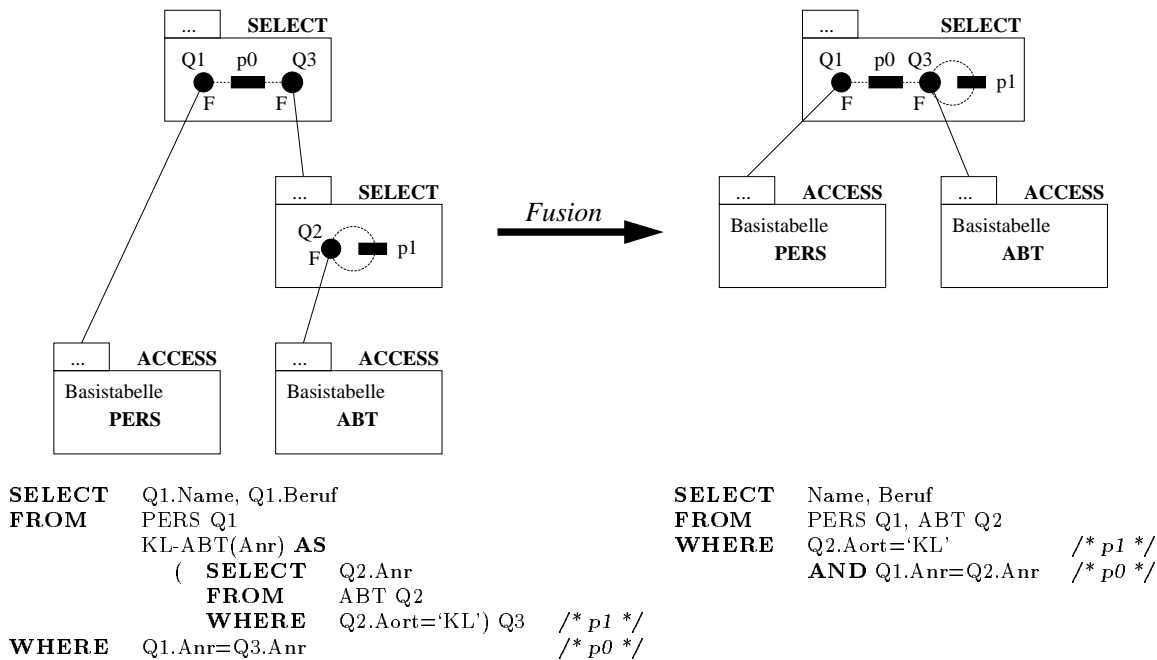


Abbildung 3.3: Beispiel für die Anwendung der Regel *Fusion* (vereinfachte Darstellung des QGM-Graphen) [M95].

Ein großer Vorteil von QGM liegt in der Erweiterbarkeit durch die Definition neuer Tabellenoperator-Typen und in der Flexibilität und Mächtigkeit, die sich durch Verwendung einer Programmiersprache für die Regelformulierung ergibt; so ist beispielsweise auch die

Überprüfung semantischer Bedingungen möglich. Die Verifizierung solcher Regeln ist jedoch schwierig [CZ98].

3.3 Relationenalgebra

Die Relationenalgebra [M95] [GUW00] ist eine in Optimierern häufig verwendete Notation zur Darstellung von Anfragen auf logischer Ebene. Sie basiert auf einer Menge von Operatoren, die Relationen bearbeiten; hierzu gehören unter anderem Selektion σ , Projektion π und Verbund/Join \bowtie . Ein Ausdruck der Relationenalgebra beschreibt eigentlich einen Algorithmus zur Konstruktion der Ergebnisrelation einer Anfrage, d.h. einen logischen Anfrageplan, und ist somit prozeduraler Natur. Daher gibt es in der Relationenalgebra auch mehrere alternative Darstellungen für eine Anfrage. In Abbildung 3.4 sind drei alternative Operatorbäume für die folgende Anfrage dargestellt:

```

SELECT *
FROM   A, B, C
WHERE  A.x=B.x AND B.x=C.x AND
        A.a=1
  
```

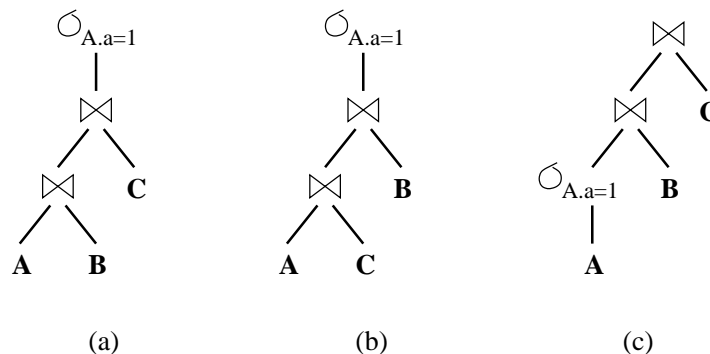


Abbildung 3.4: Drei alternative Operatorbäume zu derselben Anfrage

Ein Problem ergibt sich bei der Verwendung von Unteranfragen in der WHERE-Klausel. Der Selektions-Operator der Relationenalgebra stellt einen Test dar, der auf die einzelnen Tupel der Eingangsrelation angewandt wird. Eine Unteranfrage stellt jedoch eine weitere Relation dar. D.h. eine Bedingung, die eine Unteranfrage enthält, kann nicht mit dem Selektions-Operator dargestellt werden, sondern muss durch einen Operator ausgedrückt werden, der auf ganzen Relationen arbeitet. In vielen Fällen kann die Bedingung mit der Unteranfrage in eine Join-Operation zwischen der äußeren Anfrage und der Unteranfrage umgewandelt werden. Um auch die Fälle, in denen diese Umwandlung nicht möglich ist, abzudecken, muss das Darstellungsschema entsprechend erweitert werden. In [GUW00] wurde hierfür die Zwei-Argument-Selektion eingeführt, die eine Mischung zwischen Parse-Baum und Relationenalgebra ist. Abbildung 3.5 zeigt ein Beispiel für einen Operatorgraph mit Zwei-Argument-Selektion, die zugehörige Anfrage lautet [GUW00]:

```

SELECT title
FROM StarsIn
WHERE starName IN
    ( SELECT name
      FROM MovieStar
      WHERE birthdate LIKE '%1960')

```

Eine weitere Möglichkeit, Unteranfragen darzustellen, bietet die in [GWMS01] vorgestellte,

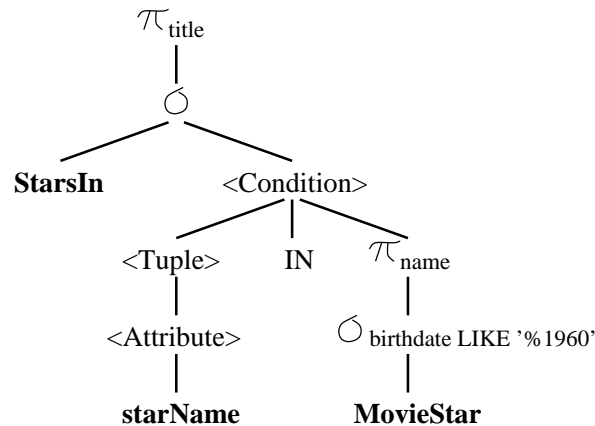


Abbildung 3.5: Operatorgraph mit Zwei-Argument Selektion σ [GUW00]

erweiterte Relationenalgebra. Dort wird als Unteranfrage-Operator der Apply-Operator eingeführt, der zwei Eingabekanten besitzt. Die linke Eingabekante repräsentiert den äußeren Block der Anfrage, die rechte Eingabekante repräsentiert die Unteranfrage, wobei die Unteranfrage freie Variablen von der linken Seite enthalten darf. Treten mehrere Unteranfragen in derselben WHERE-Klausel auf, so muss deren logische Verknüpfung auf die Verbundoperation oder auf Mengenoperationen abgebildet werden.

3.4 GENTREES

GENTREES [B97] sind Ausdrücke, die Operatorbäume beschreiben. Es handelt sich dabei um ein proprietäres Format, das in dem parallelen relationalen DBMS MIDAS (Munich parallel database system) zur Darstellung von Anfragen verwendet wird.

Die vom MIDAS-Interpreter erzeugten Operatorbäume ähneln den Operatorbäumen der Relationenalgebra. Abbildung 3.6 zeigt den Operatorbaum zu folgender SQL-Anfrage [B97]:

```

SELECT tname, owner, segno + colno * 7
FROM systable t
WHERE NOT EXISTS
    ( SELECT *
      FROM sysuser u
      WHERE u.userid=t.owner)

```

Der Wurzelknoten des Operatorbaums gibt an, um welche Art von SQL-Anweisung es sich handelt, in diesem Fall eine Anfrage (SELECT). Bei den inneren Knoten handelt es sich um

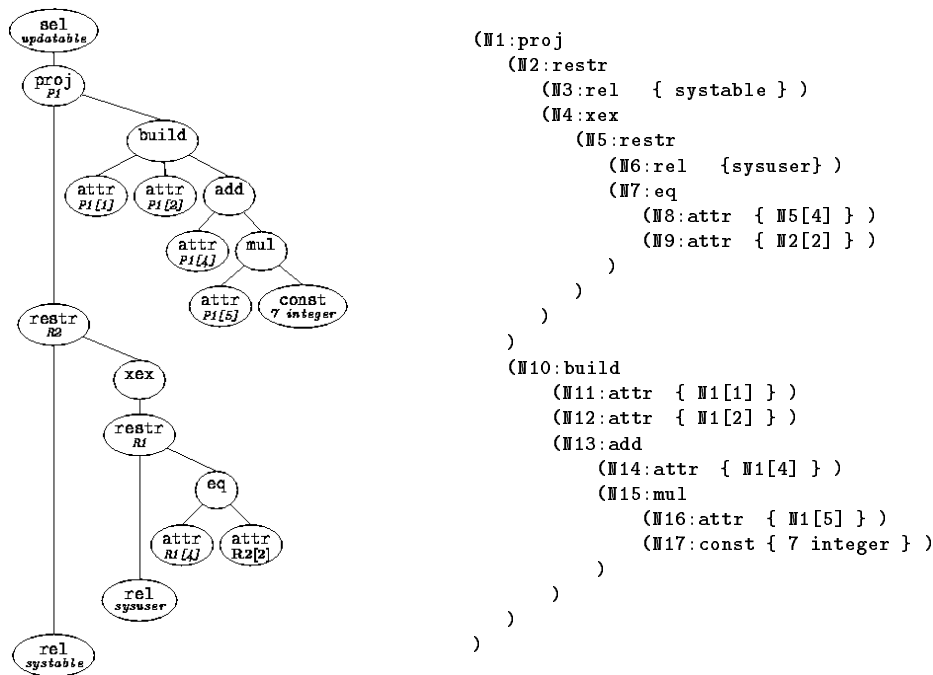


Abbildung 3.6: Beispiel für einen Operatorbaum [B97] und zugehörigen GENTREE

Operatoren verschiedenster Art (Operatoren der relationalen Algebra, Mengenoperatoren, arithmetische und boolesche Operatoren, Vergleiche von Relationen oder Werten, u.a.). Die meisten Operationen liefern Werte oder Relationen. Die erste Zeile eines Knotens enthält den Bezeichner des Operators, die zweite enthält weitere Zusatzinformationen, falls nötig. Für Knoten, auf die Verweise von ATTR-Knoten existieren, steht in der zweiten Zeile ein eindeutiger Name für den Knoten. Die Auswertung beginnt beim ersten Knoten unterhalb der Wurzel. Jeder interne Knoten ruft zuerst (von links nach rechts) seine Söhne auf, damit sie ihre Ausgabe erzeugen. Danach führt er seine Operation durch, stellt seine Ausgabe seinem Vater zur Verfügung und übergibt die Kontrolle wieder an seinen Vater.

Die Darstellung des Operatorbaums als GENTREE entspricht einem Präorder-Durchlauf des Baumes. Dabei wird jeder Knoten mit seinen Söhnen durch ein Klammerpaar eingeschlossen. Die Darstellung jedes Knotens selbst beginnt mit einem optionalen Identifikator für den Knoten, der nur bei Knoten benötigt wird, auf die in ATTR-Knoten verwiesen wird. Danach folgen der Bezeichner des Knotentyps und, falls erforderlich, in geschweiften Klammern die knotenspezifischen Komponenten.

GENTREES sind aufgrund der LISP-ähnlichen Darstellung einfach zu verarbeiten und die zugehörige Baumdarstellung ist gut lesbar. Positiv zu werten ist auch, dass Terme und Ausdrücke, wie sie in der SELECT- und WHERE-Klausel auftreten, als Operatorbäume dargestellt sind. Die daraus resultierenden alternativen Darstellungen erschweren jedoch einen Vergleich der Operatorbäume.

Als Optimierer für die Operatorbäume kommt das Cascades Optimizer Framework [S96] zum Einsatz. Die Regeln sind Objekte, die in C++ kodiert sind. Sie haben ein *before-pattern* (Ausgangsmuster oder Antezedenz) und ein *after-pattern* (Muster des transformierten Ausdrucks oder Sukzedenz), welche die Regel definieren. Zu jeder Regel gehört auch eine Bedingung, unter der die Regel angewendet werden darf. Diese Bedingung bezieht sich auf

Eigenschaften, die der Ausdruck des Ausgangsmusters haben muß, und ist in der Methode *condition* des Regelobjekts implementiert. Beispielsweise muss bei der Regel, die eine Selektion durch einen Join hindurch nach unten in den linken Eingabe-Zweig des Joins verschiebt, in der Methode *condition* überprüft werden, ob das Selektionsprädikat nur Attribute des linken Eingabe-Zweigs enthält.

3.5 COKO-KOLA

Zielsetzung des COKO-KOLA-Projekts [CZ98] ist, Restrukturierungsregeln so darzustellen, dass sie mit einem Theoremprüfer verifiziert werden können. Dabei sollen auch sehr spezifische Regeln mit semantischen Bedingungen berücksichtigt werden. Bei den bisher vorgestellten Ansätzen musste Programmiersprachen-Code eingebunden werden, um semantische Bedingungen zu realisieren. Programmiersprachen-Code hat jedoch den Nachteil, dass er sehr schwer zu verifizieren ist.

KOLA	Semantik	
id	$\mathbf{id} ! x = x$	Identität
π_1	$\pi_1 ! [x, y] = x$	Projektion
π_2	$\pi_2 ! [x, y] = y$	Projektion
$\langle \mathbf{att} \rangle$	$\langle \mathbf{att} \rangle ! x = x.\langle \mathbf{att} \rangle$	Attribut des DB-Schemas
\circ	$(f \circ g) ! x = f ! (g ! x)$	Komposition
$\langle \rangle$	$\langle f, g \rangle ! x = [f ! x, g ! x]$	Funktionspaarung
\times	$(f \times g) ! [x, y] = [f ! x, g ! y]$	paarweise Funktionsanwendung
\mathbf{K}_f	$\mathbf{K}_f(x) ! y = x$	Konstantenfunktion
\mathbf{C}_f	$\mathbf{C}_f(f, x) ! y = f ! [x, y]$	
eq	$\mathbf{eq} ? [x, y] = x == y$	Äquivalenztest
lt	$\mathbf{lt} ? [x, y] = x < y$	Ordnungsprädikat "kleiner als"
\oplus	$(p \oplus f) ? x = p ? (f ! x)$	Kombination von Prädikat und Funktion
$\&$	$(p \& q) ? x = (p ? x) \wedge (q ? x)$	Konjunktion (logisches UND)
$ $	$(p q) ? x = (p ? x) \vee (q ? x)$	Disjunktion (logisches ODER)
\sim	$\sim(p) ? x = \neg(p ? x)$	Negation
\mathbf{K}_p	$\mathbf{K}_p(b) ? x = b$	Konstante
\mathbf{C}_p	$\mathbf{C}_p(p, x) ? y = p ? [x, y]$	

Abbildung 3.7: Semantik der KOLA-Funktionen und -Prädikate [CZ98]

$\mathbf{set} ! A$	$= \{x x^i \in A\}$	Duplikateliminierung
$\mathbf{iterate} (p, f) ! A$	$= \{(f ! x)^i x^i \in A, p ? x\}$	SELECT-FROM-WHERE-Konstrukt
$\mathbf{join} (p, f) ! [A, B]$	$= \{(f ! [x, y])^i j x^i \in A, y^j \in B, p ? [x, y]\}$	Join
$\mathbf{exists} (p) ? A$	$= \exists x, j (x^j \in A \wedge p ? x)$	Existenzquantor
$\mathbf{forall} (p) ? A$	$= \forall x, j (x^j \in A \Rightarrow p ? x)$	Allquantor

Abbildung 3.8: Semantik der KOLA-Kombinatoren [CZ98]

KOLA ist eine Kombinator-basierte Algebra zur Darstellung von Anfragen. D.h. Anfragen bzw. Funktionen werden durch Kombinatoren aus anderen Funktionen zusammengesetzt und enthalten daher keine Variablen, was die Arbeit eines Theoremprüfers vereinfacht. In Abbildung 3.7 und 3.8 sind einige KOLA-Operatoren definiert. f und g stehen für beliebige

Funktionen, p und q für beliebige Prädikate, A und B für Mengen (ohne Duplikate) oder Multi-Mengen (mit Duplikaten), x und y für beliebige Werte oder Objekte. In KOLA wird die Anwendung einer Funktion durch den Infix-Operator “!” und die Anwendung eines Prädikats durch den Infix-Operator “?” dargestellt. Diese Liste an Primitiven ist jedoch nur eine Auswahl, KOLA enthält z.B. noch weitere Ordnungsprädikate. Für OQL existiert bereits ein Übersetzer.

Als Beispiel sei folgende OQL-Anfrage [CZ98] gegeben:

```
select  *
from    s1 in S, s2 in S'
where   s1.terms > 5 AND
        s1.terms == s2.terms
```

Der zugehörige KOLA-Ausdruck lautet:

$$\mathbf{join} (\rho, id) ! [S, S'] \quad \text{mit} \quad \rho = (\mathbf{eq} \oplus (\mathbf{terms} \times \mathbf{terms})) \ \& \ (\mathbf{C}_p(\mathbf{lt}, 5) \oplus \mathbf{terms} \oplus \pi_1)$$

Es können nun einfache Restrukturierungsregeln der Form $L \xrightarrow{\exists} R$ erstellt werden, wobei L und R jeweils ein KOLA-Ausdruck ist. Um komplexere Transformationen mit semantischen Bedingungen ausdrücken zu können wurde die Sprache COKO entwickelt. Eine COKO-Transformation besteht aus einer Menge von Restrukturierungsregeln und einem Algorithmus, der das Feuern dieser Regeln kontrolliert. Um nun semantische Transformation ausdrücken zu können wurden zwei neue Arten von Regeln eingeführt, Conditional Rewrite Rules und Inference Rules.

Conditional Rewrite Rules entsprechen den ‘normalen’ Restrukturierungsregeln, aber können zusätzlich noch semantische Vorbedingungen enthalten, die für bestimmte Teile der linken und rechten Regelseite erfüllt sein müssen. Sie haben die Form $C :: L \xrightarrow{\exists} R$, wobei C für die Vorbedingung steht.

Inference Rules dienen der Herleitung von Vorbedingungen. Sie werden vom COKO-Compiler in Algorithmen übersetzt, die beim Feuern der Regeln ausgeführt werden. Inference Rules haben die Form $body \implies head$ oder nur $head$, was einen Fakt darstellt. Der Regelkopf $head$ repräsentiert eine Bedingung, die durch die Regel impliziert wird. Diese Bedingung ist eine nicht-interpretierte logische Beziehung, deren Argumente entweder KOLA-Ausdrücke oder Pattern-Variablen sind. Der Regelkörper $body$ ist ein logischer Ausdruck, der sich aus Konjunktionen, Disjunktionen und Negationen von Termen zusammensetzt. Diese Terme stellen die Bedingungen dar, die erfüllt sein müssen, damit die Kopf-Bedingung impliziert werden kann.

Obige OQL-Anfrage könnte unter Ausnutzung der Transitivität von Prädikaten, die durch die Join-Bedingung “s1.terms == s2.terms” gegeben ist in folgende, semantisch äquivalente Anfrage transformiert werden:

```
select  *
from    s1 in S, s2 in S'
where   s1.terms > 5 AND
        s1.terms == s2.terms
        s2.terms > 5
```

Dies kann mit der folgenden Conditional Rewrite Rule erreicht werden:

$$\text{is_stronger}(p, q) :: p \xrightarrow{\vec{}} (p \& q)$$

Von den zugehörigen Inference Rules trifft die folgende auf das Beispiel zu:

$$\text{is_stronger}((\mathbf{eq} \oplus (f \times g)) \& (p \oplus f \oplus \pi_1), p \oplus g \oplus \pi_2)$$

In diesem Fall handelt es sich sogar um ein Fakt; die Vorbedingung einer Conditional Rule kann aber auch zur Aktivierung einer ganzen Kette von Inference Rules führen. In [CZ98] sind weitere Inference Rules und Conditional Rules aufgelistet.

Der KOLA-Ausdruck hat nach der Transformation die Form:

$$\mathbf{join} (\rho \& \tau, id) ! [S, S'] \quad \text{mit} \quad \tau = \mathbf{C}_p(\mathbf{lt}, 5) \oplus \mathbf{terms} \oplus \pi_2$$

Ein großer Vorteil des COKO-KOLA-Ansatzes ist die Möglichkeit, semantische Bedingungen formulieren zu können und diese zusammen mit den Restrukturierungsregeln durch einen Theoremprüfer verifizieren zu können. Die Variablen-freie Darstellung vereinfacht dabei die Verarbeitung dieser Ausdrücke. Allerdings leidet die Lesbarkeit unter dieser Darstellung und die Zahl der Regeln für eine gegebene Problematik ist recht groß, da wie bei anderen algebraischen Notationen sämtliche Alternativen in den Regeln zu berücksichtigen sind.

Kapitel 4

Multi-Query-Rewrite

Das Ziel dieser Diplomarbeit ist die Optimierung ganzer Anfragesequenzen, d.h. dass die Anfragen nicht wie beim Single-Query-Rewrite einzeln optimiert werden, sondern die Anfragesequenz als ganzes restrukturiert wird. Dies bedeutet, dass auch Ähnlichkeiten und Abhängigkeiten zwischen den einzelnen Anfragen der Sequenz miteinbezogen werden müssen und es zur Verschmelzung mehrerer Anfragen kommen kann, aber auch zur Erweiterung der Sequenz um zusätzliche Anfragen. Typische Anwendungsbeispiele sind: die Verschmelzung ähnlicher Anfragen, die Verschmelzung von aufeinanderfolgenden Anfragen, die Auslagerung von gemeinsamen Operationen in eine zusätzliche Anfrage und die Verschiebung von Prädikaten entlang den Abhängigkeitsbeziehungen. Diese Form der Restrukturierung, wird im folgenden als “Multi-Query-Rewrite” bezeichnet.

4.1 Regeln

Die in den nachfolgenden Unterabschnitten beschriebenen Regeln basieren hauptsächlich auf den Überlegungen in [SWM01]. Einige der dort vorgestellten Regeln sind jedoch sehr spezifisch und betreffen große Teile einer Anfragesequenz. Dies hat zur Folge, dass diese Regeln nur in einigen speziellen Fällen oder sogar nur bei der gegebenen Beispielsequenz anwendbar sind. Meist wird auch die Kenntnis der Semantik der gesamten Anfragesequenz oder einzelner Teilsequenzen vorausgesetzt. Dies stellt für einen menschlichen Betrachter, der zumal die Fragestellung kennt, kein allzu großes Problem dar. Die Umsetzung solcher Regeln auf eine maschinelle Verarbeitung ist jedoch nicht trivial. Schon die exakte Formulierung einer Regelbedingung, die alle Eventualitäten und Randbedingungen miteinschließt, fällt hier schwer. Die daraus resultierende Implementierung ist recht komplex und aufwendig. Daher habe ich einige der Regeln modifiziert und in mehrere Regeln aufgesplittet, d.h. ich verfolge einen Ansatz mit kleineren Regeln, die nicht sehr spezifisch sind und meist nur eine einzelne Anfrage und die direkt davon abhängigen Anfragen betreffen. Die Formulierung von Regelbedingung und -aktion bleibt damit überschaubar. Diese Regeln sind flexibel, können meist an mehreren Stellen einer Anfragesequenz angewendet werden und schließen nicht den Kontext größerer Teilsequenzen mitein. Es besteht sogar mit gewissen Einschränkungen eine Orthogonalität zwischen den Regeln. Durch die sequenzielle Ausführung mehrerer dieser kleineren Regeln können ebenso recht komplexe Umformungen erzielt werden. Für einige der Regeln in [SWM01] wäre sogar eine Kostenabschätzung notwendig. So sehe ich beispielsweise die Verschiebung von Join-Operationen in Richtung Sequenzende nicht immer als sinnvoll an,

auch wenn dadurch eine weitere Join-Operation eingespart werden kann.

Um diese Behauptungen zu untermauern, möchte ich die Transformationen (2) und (3) in [SWM01, S. 10f] genauer betrachten. Abbildung 4.1 zeigt die vorgenommenen Umformungen und entspricht Abbildung 7 in [SWM01]. Bei der Transformation (2) *MoveJoin* wird der Join mit Anfrage *C4* aus der modifizierten Version der Anfrage *C5* in die Anfrage *C7* verschoben mit der Begründung, dass die Attribute aus *C4* in *C5* außer im Join-Prädikat keine Verwendung finden. Geht man jedoch davon aus, dass der Join mit *C4* als Filter dient, der die Tupel aus *lineitem_orders* auswählt, die im weiteren Verlauf der Sequenz benötigt werden, dann ist die Verschiebung dieses Joins nicht sinnvoll. Sie erhöht nämlich in *C5* die Zahl der Tupel, die gruppiert bzw. aggregiert werden müssen, und auch die Zahl der Tupel im Ergebnis von *C5*. Bezieht man die Transformation (3) mitein, so kann durch die Verschiebung jedoch ein Join mit *C4* eingespart werden. Hier müsste also eine Kostenabschätzung durchgeführt werden, die feststellt, ob die Einsparung des Joins den erhöhten Verarbeitungsaufwand durch die erhöhten Tupelzahlen aufwiegt. Es handelt sich hierbei folglich um keine Heuristik. Bei der Transformation (3) *MoveJoin* werden in Anfrage *C7* die drei Joins mit den Anfragen *C1*, *C2* und *C3* durch einen Join mit der Anfrage *C4* ersetzt. Dies setzt allerdings voraus, dass in *C7* nur die Daten, d.h. Tupel und Attribute, von *C1*, *C2* und *C3* benötigt werden, die auch in *C4* enthalten sind und dass nicht drei Joins mit *C4* nötig sind, sondern ein Join ausreicht. Eigentlich ist hierfür die Kenntnis der Semantik der gesamten Abfragesequenz bzw. der Bedeutung und Wirkung der einzelnen Anfragen notwendig. Hierfür eine Regelbedingung zu formulieren ist nicht einfach und die daraus resultierende Implementierung dürfte recht komplex werden.

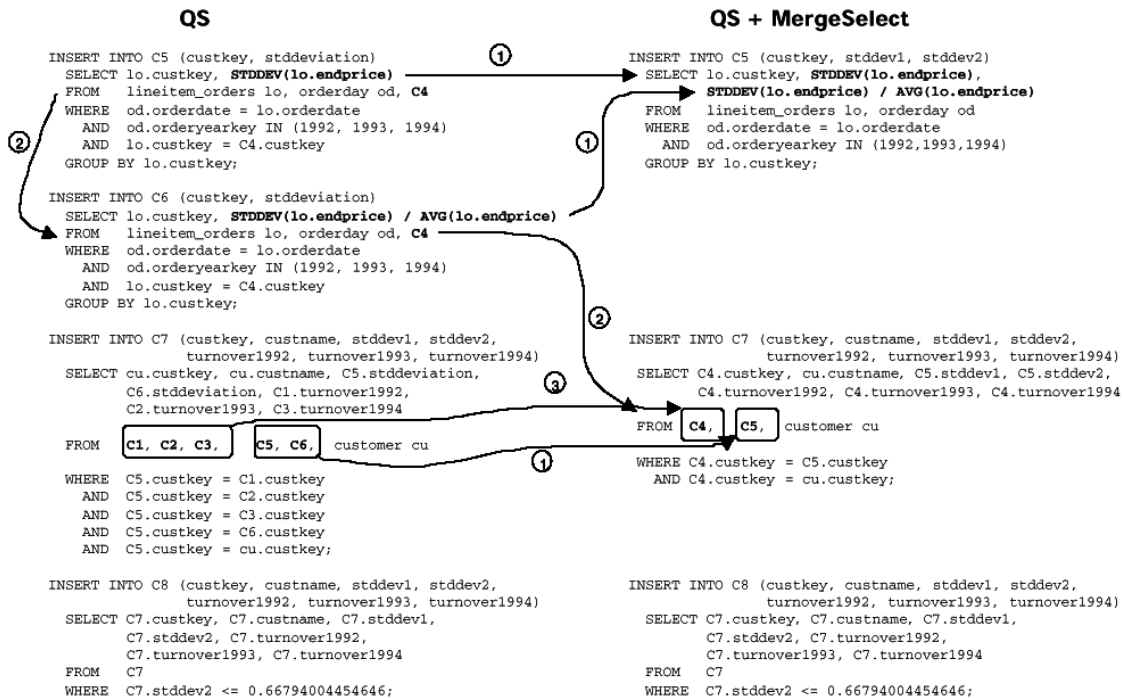


Abbildung 4.1: Beispiel für Transformationen auf einer Abfragesequenz [SWM01, Abb. 7]

Die Regeln des Multi-Query-Rewrite unterscheiden sich in der Problematik und damit auch in der Komplexität von den Regeln des Single-Query-Rewrite. Wie in Kapitel 3 zu

sehen war, wird beim Single-Query-Rewrite meist nach dem Auftreten eines bestimmten Musters innerhalb einer Anfrage gesucht, d.h. es wird der Teil der Anfrage ermittelt in dem dieses Muster auftritt. Die Restrukturierungsregeln betreffen daher meist nur eine einzelne Unteranfrage oder eine (Unter-) Anfrage und die in diese eingebettete Unteranfrage. Sie sind im allgemeinen nicht sehr komplex und haben nur einen relativ kleinen Kontext und lokale Auswirkungen. Für die Suche nach einer anwendbaren Regel reicht in den meisten Fällen ein einfaches Pattern-Matching und für die Evaluation der Regelaktion die Substitution des gefundenen Patterns aus. Die Produktionsregeln haben entsprechend die Form $L \Rightarrow R$, wobei es sich bei L und R jeweils um ein Pattern handelt, das eine (Unter-) Anfrage und eventuell deren Unteranfrage repräsentiert. Beim Multi-Query-Rewrite betrifft eine Restrukturierungsregel meist mehrere Anfragen, d.h. es handelt sich bei der Anwendung einer Restrukturierungsregel um eine Abbildung von n Anfragen auf m Anfragen. Die Schwierigkeit und Komplexität liegt daher beim Vergleich und der Auswahl der für eine Regelanwendung passenden Anfragen. Es wird also nicht wie beim Single-Query-Rewrite nach einem bestimmten Muster innerhalb einer Anfrage gesucht, sondern es geht vielmehr darum, eine Menge von Anfragen, die bestimmte Übereinstimmungen und Unterschiede aufweisen und daher miteinander verschmolzen oder gemeinsam umstrukturiert werden können, zu finden. Eine Umformung innerhalb einer oder mehrerer Anfragen kann Umformungen in den von diesen abhängigen Anfragen zur Folge haben, d.h. die Restrukturierungsregeln beschränken sich nicht auf lokale Auswirkungen und einen lokalen Kontext sondern beziehen auch die Abhängigkeiten zwischen den Anfragen mitein. Ein simples Pattern-Matching reicht hier nicht aus.

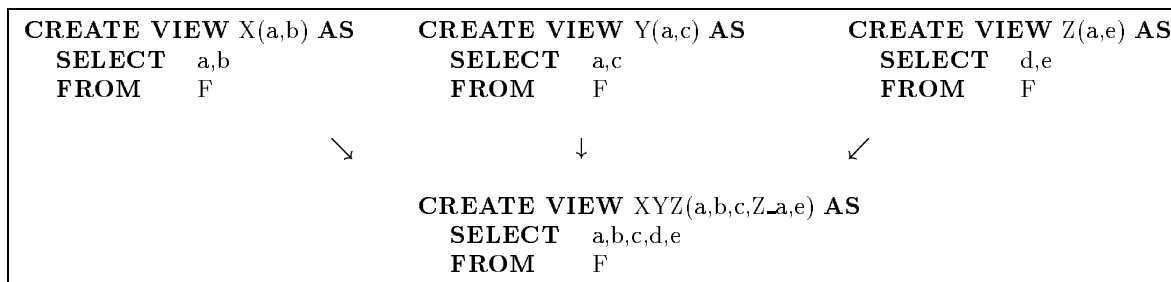
In den nachfolgenden Abschnitten sind einige Multi-Query-Rewrite-Regeln beschrieben. Es wird eine nicht-formale, natürlichsprachliche Notation verwendet, die einer intuitiven Beschreibung der Regeln entspricht. Da sowohl Eingabe als auch Ausgabe der Regeln SQL-Anweisungen sind bzw. die Regeln eigentlich nur SQL-Anweisungen restrukturieren, beziehen sich die Formulierungen auf die SQL-spezifische Klauselstruktur und deren Komponenten. Es werden daher die für SQL typischen Begriffe "Klausel", "Prädikat" und "Term" verwendet. Die Regeln habe ich aufgrund ihrer Eigenschaften grob in drei Klassen eingeteilt. Bei der Formulierung der Regelbedingungen habe ich eine eher deskriptive Form gewählt, bei den Regelaktionen eine imperative, prozedurale Form, die die zur Transformation notwendigen Schritte auflistet. Die angeführten Beispiele sollen die Anwendung der Regeln verdeutlichen; zur Vereinfachung sind nur die Sichtdefinitionen angegeben.

4.1.1 Regeln der Klasse 1

Bei den Regeln dieser Klasse handelt es sich um Regeln, die speziell auf das Multi-Query-Rewrite zugeschnitten sind und für die es keine Anwendung und auch keine Analogie im Single-Query-Rewrite gibt. Hierzu zählen insbesondere die Regeln, die Anfragemengen berechnen. D.h. es wird nach Anfragen gesucht, die in bestimmten Komponenten übereinstimmen müssen und sich in bestimmten Komponenten unterscheiden können oder müssen. Die Regelbedingung kann hierbei auch als eine Art Äquivalenzbedingung aufgefasst werden, die paarweise für alle Anfragen einer Anfragemenge gelten muss. Die Abhängigkeitsbeziehungen zwischen den Anfragen der Anfragesequenz spielen hier nur indirekt, beim Vergleich der FROM-Klauseln eine Rolle; der Kontext einer Regelbedingung beschränkt sich auf die jeweils zu vergleichenden Anfragen.

Merge Select

Diese Regel vereinigt mehrere Anfragen, die sich nur in der SELECT-Klausel unterscheiden, zu einer einzigen Anfrage. Im nachstehenden Beispiel werden die Anfragen X, Y und Z zur Anfrage XYZ vereinigt:



Die Regel *Merge Select* lautet:

- **Bedingung:** Es existiert eine Anfragemenge, für die gilt:
 - a) Jede Anfrage dieser Menge stimmt mit jeder anderen Anfrage dieser Menge in allen Klauseln außer der SELECT-Klausel überein.
 - b) Keine der Anfragen in dieser Menge ist eine Ergebnisanfrage.
 - c) Keine der Anfragen in dieser Menge enthält ein DISTINCT.
- **Aktion:**
 1. Eine neue Anfrage erzeugen.
 2. Alle übereinstimmenden Klauseln, d.h. alle Klauseln außer der SELECT-Klausel, in die neue Anfrage übernehmen.
 3. SELECT-Klauseln und die zugehörigen Attributdefinitionen aller Anfragen der Anfragemenge vereinigen und der neuen Anfrage hinzufügen.
 4. Duplikate in der vereinigten SELECT-Klausel entfernen.
 5. Bezeichnerkollisionen in den vereinigten Attributdefinitionen durch Umbenennung beheben.
 6. Typ der neuen Anfrage ('Tabelle' oder 'Sicht') bestimmen.
 7. Primärschlüssel der neuen Anfrage bestimmen.
 8. Alle Referenzen auf die Anfragen der Anfragemenge und deren Attribute an die neue Anfrage anpassen.
 9. Alle Anfragen der Anfragemenge löschen.

Einige Anmerkungen hierzu:

Bei der Überprüfung zweier Anfragen auf Übereinstimmung (a) muss berücksichtigt werden, dass die Elemente in den einzelnen Klauseln bei beiden Anfragen unterschiedlich angeordnet sein können. Nur bei der ORDER-BY-Klausel ist die Reihenfolge der Elemente relevant und muss übereinstimmen. Ebenso müssen die Alias-Bezeichner zusammengehöriger FROM-Klausel-Elemente nicht übereinstimmen und von übereinstimmenden Alias-Bezeichnern kann nicht auf zusammengehörige FROM-Klausel-Elemente geschlossen werden. Außerdem kann dieselbe Tabelle bzw. Sicht mehrfach von einer Anfrage referenziert werden. D.h. das eigentliche Ziel beim Vergleich zweier Anfragen ist es, eine Zuordnung der FROM-Klausel-Elemente

beider Anfragen zu finden, unter der auch die restlichen Klauseln mit Ausnahme der SELECT-Klausel übereinstimmen.

Die Anfragen in der Anfragemenge dürfen keine Ergebnisanfragen sein (b), da bei Ergebnisanfragen die Attributdefinitionen nicht verändert werden dürfen, was jedoch bei der Verschmelzung der SELECT-Klauseln und der zugehörigen Attributdefinitionen der Fall ist.

Um zu begründen, dass die Anfragemenge keine Anfrage mit DISTINCT enthalten darf (c), möchte ich den Fall betrachten, in dem die Anfragemenge Anfragen mit DISTINCT enthält. In diesem Fall ist es möglich dass sich eine Anfrage mit DISTINCT in der Zahl der Tupel von einer Anfrage ohne DISTINCT bzw. von einer anderen Anfrage mit DISTINCT unterscheidet. Letzteres beruht auf den unterschiedlichen Projektionen und der anschließenden Duplikateliminierung. Eine Vereinigung dieser Anfragen könnte bei den nachfolgenden Operationen, besonders bei Aggregatberechnungen, zu einem anderen als dem ursprünglichen Ergebnis führen.

Bei der Duplikateliminierung (4.) in der SELECT-Klausel müssen die unterschiedlichen Attributeigenschaften der zugehörigen Attributdefinitionen berücksichtigt werden. Da ein implizites Type-Casting bei der Definition der zugrundeliegenden Anfragesprache in Abschnitt 2.1 ausgeschlossen wurde, sind die Duplikate zwar vom gleichen Datentyp, in den Integritätsbedingungen wie “PRIMARY KEY”, “UNIQUE” und “NOT NULL” können sie sich jedoch unterscheiden. Daher muss festgelegt werden, wie in einem solchen Konfliktfall die Integritätsbedingung bestimmt wird.

Bezeichnerkollisionen (5.) können entstehen, wenn in zwei zu verschmelzenden Anfragen, dieselben Bezeichner für Attributdefinitionen verwendet werden, aber die dazugehörigen Terme in den SELECT-Klauseln nicht übereinstimmen; d.h. zwei Attributdefinitionen tragen denselben Bezeichner, sind aber keine Duplikate. Dieses Problem kann durch Umbenennung der Attributbezeichner gelöst werden. Die Bezeichneränderung muss entsprechend auch in den referenzierenden Anfragen nachgeführt werden (8.).

Bei der Ermittlung des Anfragetyps (6.) gibt es mehrere Alternativen. Man könnte beispielsweise, wenn eine der Anfragen der Anfragemenge vom Typ ‘Tabelle’ ist, auch den Typ der vereinigten Anfrage auf ‘Tabelle’ setzen. Ebenso könnte man aber auch mit dem Typ ‘Sicht’ verfahren oder man könnte ermitteln, welcher Anfragetyp häufiger in der Anfragemenge auftritt, und diesen wählen. Will man jedoch Anfragen unterschiedlichen Typs nicht vermischen, dann muss man die Anfragemenge in eine Menge mit Anfragen vom Typ ‘Sicht’ und eine Menge mit Anfragen vom Typ ‘Tabelle’ aufsplitten und die Vereinigung für diese beiden Anfragemengen getrennt durchführen.

Bei der Wahl des Primärschlüssels (7.) kommen wiederum zwei Alternativen in Frage. Man könnte zum einen den kürzesten Primärschlüssel unter den Anfragen der Anfragemenge ermitteln, zum anderen den Primärschlüssel einer beliebigen Anfrage der Anfragemenge übernehmen. Der zweite Ansatz ist nicht immer korrekt, da die Minimalität des Primärschlüssels bei diesem Ansatz nicht immer garantiert ist. Beim ersten Ansatz ist sie aber auch nur dann garantiert, wenn die Primärschlüssel der Anfragen der Anfragemenge schon minimal sind. Enthalten die Anfragen der Anfragemenge eine GROUP-BY-Klausel, dann stellt die Menge der Gruppierungsattribute eine weitere Alternative für den Primärschlüssel dar, da die Gruppierungsattribute ein Tupel eindeutig identifizieren. Dieser Ansatz ist aber auch nicht immer korrekt, da bei entsprechenden Abhängigkeiten zwischen den Gruppierungsattributen eventuell schon eine Teilmenge der Gruppierungsattribute als Identifikator ausreicht und somit der so gewählte Primärschlüssels nicht minimal ist.

Bei der Anpassung der referenzierenden Anfragen (8.) müssen in diesen zum einen alle Referenzen auf Tabellen bzw. Sichten, die zu den Anfragen der Anfragemenge gehören, gegen Referenzen auf die Tabelle bzw. Sicht der neuen Anfrage ausgetauscht werden. Zum anderen müssen alle Bezeichner von Attributen, die zu den Anfragen der Anfragemenge gehören und deren Bezeichner sich aufgrund von Duplikateliminierung (4.) oder Bezeichnerkollision (5.) geändert hat, in den referenzierenden Anfragen entsprechend umbenannt und somit an die Attributdefinitionen der neuen Anfrage angepasst werden.

Where To Group

Diese Regel vereinigt mehrere gruppierende Anfragen, die sich nur in der SELECT-Klausel unterscheiden und in einer Selektion, die in allen diesen Anfragen enthalten ist und auch in allen das gleiche Attribut betrifft, aber jeweils unterschiedliche Werte bzw. Wertebereiche auswählt. Die vereinigte Anfrage selektiert den vereinigten Wertebereich aller Anfragen, weshalb die Selektion des anfragespezifischen Ausschnitts den referenzierenden Anfragen hinzugefügt werden muss. Im nachstehenden Beispiel werden die Anfragen X und Y zur Anfrage XY vereinigt und es wird eine entsprechende Selektion der referenzierenden Anfrage Z hinzugefügt:

CREATE VIEW X(a,s) AS	CREATE VIEW Y(a,t) AS	CREATE VIEW Z(a) AS
SELECT a,SUM(c)	SELECT a,SUM(d)	SELECT a
FROM F	FROM F	FROM X
WHERE b=1	WHERE b=2	
GROUP BY a	GROUP BY a	
↓	↙	↓
CREATE VIEW XY(a,b,s,t) AS		CREATE VIEW Z(a) AS
SELECT a,b,SUM(c),SUM(d)		SELECT a
FROM F		FROM XY
WHERE b IN (1,2)		WHERE b=1
GROUP BY a,b		

Die Regel *Where To Group* lautet:

- **Bedingung:** Es existiert eine Anfragemenge, für die gilt:
 - a) Jede Anfrage dieser Menge stimmt mit jeder anderen Anfrage dieser Menge in allen Klauseln außer der SELECT-, der WHERE- und eventuell der GROUP-BY-Klausel überein.
 - b) Keine der Anfragen in dieser Menge ist eine Ergebnisanfrage.
 - c) Die WHERE-Klauseln aller Anfragen dieser Menge stimmen bis auf jeweils ein Prädikat pro Anfrage überein, d.h. jede Anfrage hat $n-1$ Prädikate, von denen $n-1$ bei allen Anfragen der Menge übereinstimmen.
 - d) Das Prädikat in dem sich die Anfragen unterscheiden hat entweder die Form $a = k$ oder $k = a$ oder $a \text{ IN } K$, wobei a ein Attribut, k eine Konstante und K eine Menge von Konstanten ist.
 - e) Das Attribut a ist bei allen Anfragen der Menge das gleiche Attribut, k und K sind von Anfrage zu Anfrage verschieden.
 - f) Die GROUP-BY-Klausel ist bei keiner Anfrage der Menge leer.
 - g) Die GROUP-BY-Klausel stimmt bei allen Anfragen der Menge, die sich in einem Prädikat der Form $a = k$ oder $k = a$ unterscheiden, überein.

- h) Die GROUP-BY-Klausel stimmt bei allen Anfragen der Menge, die sich in einem Prädikat der Form $a \text{ IN } K$ unterscheiden, überein. Zusätzlich zu den GROUP-BY-Klausel-Elementen der Anfragen, die sich in einem Prädikat der Form $a = k$ oder $k = a$ unterscheiden, enthält sie noch das Attribut a dieses Prädikats.

• **Aktion:**

(Das Prädikat, in dem sich die Anfragen der Anfragemenge unterscheiden, wird im folgenden “charakteristisches Prädikat” genannt; das Attribut in diesem Prädikat wird entsprechend als “charakteristisches Attribut” bezeichnet.)

1. Eine neue Anfrage erzeugen.
2. Alle übereinstimmenden Klauseln, d.h. alle Klauseln außer der SELECT-, der WHERE- und der GROUP-BY-Klausel in die neue Anfrage übernehmen.
3. SELECT-Klauseln und die zugehörigen Attributdefinitionen aller Anfragen der Anfragemenge vereinigen und der neuen Anfrage hinzufügen.
4. Den Datentyp des charakteristischen Attributs ermitteln und das charakteristische Attribut der vereinigten SELECT-Klausel hinzufügen.
5. Duplikate in der vereinigten SELECT-Klausel entfernen.
6. Bezeichnerkollisionen in den vereinigten Attributdefinitionen durch Umbenennung beheben.
7. WHERE-Klausel bis auf das charakteristische Prädikat in die neue Anfrage übernehmen.
8. Alle Konstanten und Mengen von Konstanten aus den charakteristischen Prädikaten der Anfragen der Anfragemenge zu einer duplikatfreien Konstantenmenge M vereinigen.
9. Das Prädikat $a \text{ IN } M$ der WHERE-Klausel der neuen Anfrage hinzufügen, wobei a das charakteristische Attribut ist und M die im vorigen Schritt berechnete Konstantenmenge.
10. Die GROUP-BY-Klausel mit allen Elementen, die bei den Anfragen der Anfragemenge übereinstimmen, in die neue Anfrage übernehmen.
11. Das charakteristische Attribut der GROUP-BY-Klausel der neuen Anfrage hinzufügen.
12. Typ der neuen Anfrage (‘Tabelle’ oder ‘Sicht’) bestimmen.
13. Primärschlüssel der neuen Anfrage bestimmen.
14. Allen Anfragen, die Anfragen der Anfragemenge referenzieren, für jede Referenz, das entsprechende charakteristische Prädikat hinzufügen, falls dieses oder ein strengeres nicht bereits in der WHERE-Klausel enthalten ist.
15. Alle Referenzen auf die Anfragen der Anfragemenge und deren Attribute an die neue Anfrage anpassen.
16. Alle Anfragen der Anfragemenge löschen.

Einige Anmerkungen hierzu:

Wie bei der Regel *Merge Select* dürfen auch hier die Anfragen der Anfragemenge keine Ergebnisanfragen sein (b). Die Anfragen dürfen jedoch ein DISTINCT enthalten, da ihre GROUP-BY-Klauseln nicht leer sind und auch die GROUP-BY-Klausel der vereinigten Anfrage nicht leer ist und somit die Tupel schon in den Gruppierungsattributen eindeutig sind. D.h. ein DISTINCT ist in diesem Fall überflüssig.

Bei der Überprüfung von Anfragen auf Übereinstimmung (a) gilt das gleiche wie bei *Merge Select*. Analog verläuft auch die Überprüfung von einzelnen Klauseln, Prädikaten oder Attributen.

Mit d, g und h werden eigentlich zwei verschiedene Typen von Anfragen abgedeckt. Zum einen Anfragen, die das Ergebnis einer früheren Anwendung von *Where To Group* sind. Bei

diesen Anfragen hat das charakteristische Prädikat die Form $a \text{ IN } K$ und die GROUP-BY-Klausel enthält bereits das charakteristische Attribut. Zum anderen Anfragen, bei denen *Where To Group* noch nicht angewandt wurde. Bei diesen Anfragen hat das charakteristische Prädikat die Form $a = k$ bzw. $k = a$ und das charakteristische Attribut ist noch nicht Element der GROUP-BY-Klausel. Im nachstehenden Beispiel werden zwei Anfragen des ersten Typs, X und Y, mit einer Anfrage Z des zweiten Typs vereinigt:

CREATE VIEW X(a,s) AS	CREATE VIEW Y(a,t) AS	CREATE VIEW Z(a,u) AS
SELECT a,SUM(c)	SELECT a,SUM(d)	SELECT a,SUM(e)
FROM F	FROM F	FROM F
WHERE b IN (1,2)	WHERE b IN (3,4)	WHERE b=5
GROUP BY a,b	GROUP BY a,b	GROUP BY a
↘	↓	↙
CREATE VIEW XYZ(a,b,s,t,u) AS		
SELECT a,b,SUM(c),SUM(d),SUM(e)		
FROM F		
WHERE b IN (1,2,3,4,5)		
GROUP BY a,b		

Die Regelaktion ist ähnlich wie bei *Merge Select*. Die großen Unterschiede liegen in der Vereinigung der WHERE- und der GROUP-BY-Klausel. In 7. wird der Teil der WHERE-Klausel, in dem alle Anfragen der Anfragemenge übereinstimmen, in die neue Anfrage übernommen. In 9. wird dann das Prädikat hinzugefügt, das die Vereinigung aller charakteristischen Prädikate repräsentiert. Ähnlich wird bei der GROUP-BY-Klausel verfahren. Hier wird ebenfalls zuerst der Teil in dem alle Anfragen der Anfragemenge übereinstimmen, in die neue Anfrage übernommen und anschließend das charakteristische Attribut hinzugefügt.

Bei der Ermittlung des Anfragetyps (12.) gibt es die schon bei *Merge Select* beschriebenen Alternativen.

Bei der Wahl des Primärschlüssels (13.) kommen ebenfalls die schon bei *Merge Select* beschriebenen Alternativen in Frage. Da das charakteristische Attribut den Gruppierungsattributen hinzugefügt wird (11.), muss allerdings beachtet werden, dass die Attributdefinition, die das charakteristische Attribut enthält, dem gewählten Primärschlüssel hinzugefügt wird, sofern dieser nicht bereits die Tupel der vereinigten Anfrage eindeutig identifiziert.

Da bei der Regel *Where To Group* die in den einzelnen Anfragen selektierten Wertebereiche des charakteristischen Attributs vereinigt werden, muss die Selektion des jeweiligen Wertebereichs in den Anfragen erfolgen, die die vereinigte Anfrage referenzieren (15.). Die Anfragen der Anfragemenge, die von der Form $a \text{ IN } K$ sind, können bereits das Ergebnis einer Anwendung von *Where To Group* sein. In diesem Fall enthalten die referenzierenden Anfragen schon eine entsprechend strengere Selektion. Es sollte daher überprüft werden, ob die WHERE-Klauseln der referenzierenden Anfragen bereits das Prädikat enthalten, das hinzugefügt werden soll, oder sogar ein Prädikat, das eine noch strengere Selektion darstellt.

Merge Where

Diese Regel ist den Regeln *Merge Select* und *Where To Group* sehr ähnlich. Sie vereinigt mehrere nicht-gruppierende Anfragen, die sich nur in der SELECT- und der WHERE-Klausel unterscheiden, indem sie deren WHERE-Klauseln über ein logisches ODER miteinander ver-

knüpft. Folglich müssen die Prädikate der ursprünglichen Anfragen den referenzierenden Anfragen hinzugefügt werden. Im nachstehenden Beispiel werden die Anfragen X und Y zur Anfrage XY vereinigt und das Prädikat aus der Anfrage X wird der referenzierenden Anfrage Z hinzugefügt:

<pre>CREATE VIEW X(a) AS SELECT a FROM F WHERE b=1</pre>	<pre>CREATE VIEW Y(a) AS SELECT a FROM F WHERE c>d+e</pre>	<pre>CREATE VIEW Z(a) AS SELECT a FROM X</pre>
↓	↙	↓
<pre>CREATE VIEW XY(a,b,c,d,e) AS SELECT a,b,c,d,e FROM F WHERE (b=1) OR (c>d+e)</pre>		<pre>CREATE VIEW Z(a) AS SELECT a FROM XY WHERE b=1</pre>

Die Regel *Merge Where* lautet:

- **Bedingung:** Es existiert eine Anfragemenge, für die gilt:
 - a) Jede Anfrage dieser Menge stimmt mit jeder anderen Anfrage dieser Menge in allen Klauseln außer der SELECT- und der WHERE-Klausel überein.
 - b) Keine der Anfragen in dieser Menge enthält eine GROUP-BY-Klausel.
 - c) Keine der Anfragen in dieser Menge ist eine Ergebnisanfrage.
 - d) Keine der Anfragen in dieser Menge enthält ein DISTINCT.
- **Aktion:**
 1. Eine neue Anfrage erzeugen.
 2. Alle übereinstimmenden Klauseln, d.h. alle Klauseln außer der SELECT- und der WHERE-Klausel, in die neue Anfrage übernehmen.
 3. SELECT-Klauseln und die zugehörigen Attributdefinitionen aller Anfragen der Anfragemenge vereinigen und der neuen Anfrage hinzufügen.
 4. Ist bei keiner Anfrage der Anfragemenge die WHERE-Klausel leer, dann die WHERE-Klauseln aller Anfragen der Anfragemenge über ein logisches ODER (OR) vereinigen und der neuen Anfrage hinzufügen, ansonsten bleibt die WHERE-Klausel der neuen Anfrage ebenfalls leer.
 5. Duplikate in der Disjunktion der vereinigten WHERE-Klausel entfernen.
 6. Alle in der vereinigten WHERE-Klausel vorkommenden Attribute der vereinigten SELECT-Klausel hinzufügen.
 7. Duplikate in der vereinigten SELECT-Klausel entfernen.
 8. Bezeichnerkollisionen in den vereinigten Attributdefinitionen durch Umbenennung beheben.
 9. Typ der neuen Anfrage ('Tabelle' oder 'Sicht') bestimmen.
 10. Primärschlüssel der neuen Anfrage bestimmen.
 11. Allen Anfragen, die Anfragen der Anfragemenge referenzieren, für jede Referenz, die Prädikate der WHERE-Klausel der entsprechenden Anfrage hinzufügen, falls diese nicht bereits in der WHERE-Klausel enthalten sind.
 12. Alle Referenzen auf die Anfragen der Anfragemenge und deren Attribute an die neue Anfrage anpassen.

13. Alle Anfragen der Anfragemenge löschen.

Einige Anmerkungen hierzu:

Bei der Regelbedingung und -aktion gelten die schon bei der Regel *Merge Select* erläuterten Besonderheiten und Alternativen.

Ist bei einer Anfrage der Anfragemenge die WHERE-Klausel leer (4.), dann entspricht dies dem Wahrheitswert 'wahr' und es werden alle Tupel selektiert. D.h., dass in diesem Fall auch die WHERE-Klausel der neuen Anfrage alle Tupel selektieren und somit leer bleiben muss. Es ist außerdem zu beachten, dass durch die ODER-Verknüpfung der Prädikate die Beschränkung auf KNF in der WHERE-Klausel verletzt werden kann, was dadurch zu beheben ist, dass man bei der Auswahl der Anfragen nur Anfragen mit einem einzigen Prädikat in der WHERE-Klausel berücksichtigt.

Diese Regel kann eventuell auch zu einer Verschlechterung der Anfragesequenz bezüglich der Ausführungszeit führen. Dies ist insbesondere dann der Fall, wenn die FROM-Klauseln der zu vereinigenden Anfragen nur wenige Elemente enthalten, die Tupelmengen der zu vereinigenden Anfragen nahezu disjunkt sind und die Zahl der Tupel in diesen disjunkten Tupelmengen relativ hoch ist und/oder die Zahl der Referenzen auf die zu vereinigenden Anfragen relativ hoch ist. Dies soll die folgende Beispielrechnung belegen:

Gegeben:

- n ist die Zahl der zu vereinigenden Anfragen.
- m ist die Zahl der Referenzen auf die zu vereinigenden Anfragen.
- i ist die Zahl der Tupel in der Tupelmenge, die den Berechnungen der zu vereinigenden Anfragen zugrunde liegt.

Annahme:

- Die Zahl der Elemente in der FROM-Klausel ist relativ gering, sodass der Aufwand für die Berechnung von Joins vernachlässigt werden kann.
- Die Zahl der Tupel o sei in den zu vereinigenden Anfragen in etwa gleich groß.
- Die Tupelmengen der zu vereinigenden Anfragen seien nahezu disjunkt, d.h. für die Zahl der Tupel o_v in der vereinigten Anfrage gilt $o_v \approx n * o$.

Dann gilt bezüglich der Zahl gelesener Tupel für den optimierenden Fall:

$$\begin{aligned} n * i + m * o &> i + m * o_v \\ n * i + m * o &> i + m * n * o \\ (n - 1) * i &> (n - 1) * m * o \\ i &> m * o \end{aligned}$$

D.h. nur solange $i > m * o$ gilt, tritt eine Optimierung ein.

Hinzu kommt, dass die Zahl der Attribute bzw. Spalten in der vereinigten Anfrage viel höher sein kann als in den ursprünglichen Einzelanfragen, wie schon das Beispiel zu Beginn dieser Regelbeschreibung zeigt. Dies kann ebenfalls zu einer Verschlechterung der Verarbeitung der Anfragesequenz beitragen. Es ist allerdings auch zu beachten, dass in obiger Beispielrechnung als Maß für den Aufwand nur die Zahl der gelesenen Tupel verwendet wurde! Enthalten die FROM-Klauseln der zu vereinigenden Anfragen mehrere Elemente, dann fällt auch der Berechnungsaufwand für Joins in's Gewicht und muss miteinbezogen werden. D.h. enthält die FROM-Klausel viele Elemente, dann kann sich auch unter den gegebenen Bedingungen die

Vereinigung der Anfragen lohnen, da nach der Vereinigung der Join über diese Elemente nur einmal für die gesamte Anfragenmenge berechnet werden muss und nicht mehr gesondert für jede Anfrage.

Ich habe diese Regel trotz alldem hier aufgeführt, da in der Realität nur selten der Worst-Case eintreten wird.

Merge Having

Wie bei *Merge Where* die WHERE-Klauseln, so werden hier die HAVING-Klauseln vereinigt. Im nachstehenden Beispiel werden die Anfragen X und Y zur Anfrage XY vereinigt und das Prädikat aus der HAVING-Klausel der Anfrage X wird entsprechend modifiziert der WHERE-Klausel der referenzierenden Anfrage Z hinzugefügt:

<pre>CREATE VIEW X(a,s) AS SELECT a,AVG(d) FROM F GROUP BY a HAVING SUM(b)> SUM(c+1)*2</pre>	<pre>CREATE VIEW Y(a,t) AS SELECT a,AVG(e) FROM F GROUP BY a HAVING SUM(b)>0</pre>	<pre>CREATE VIEW Z(a) AS SELECT a FROM X</pre>
↓	↙	↓
<pre>CREATE VIEW XY(a,s,t,u,v) AS SELECT a,AVG(d),AVG(e),SUM(b),SUM(c+1) FROM F GROUP BY a HAVING SUM(b)>SUM(c+1)*2 OR SUM(b)>0</pre>		<pre>CREATE VIEW Z(a) AS SELECT a FROM XY WHERE u>v*2</pre>

Die Regel *Merge Having* lautet:

- **Bedingung:** Es existiert eine Anfragenmenge, für die gilt:
 - a) Jede Anfrage dieser Menge stimmt mit jeder anderen Anfrage dieser Menge in allen Klauseln außer der SELECT- und der HAVING-Klausel überein.
 - b) Jede der Anfragen in dieser Menge enthält eine GROUP-BY-Klausel.
 - c) Keine der Anfragen in dieser Menge ist eine Ergebnisanfrage.
- **Aktion:**
 1. Eine neue Anfrage erzeugen.
 2. Alle übereinstimmenden Klauseln, d.h. alle Klauseln außer der SELECT- und der HAVING-Klausel, in die neue Anfrage übernehmen.
 3. SELECT-Klauseln und die zugehörigen Attributdefinitionen aller Anfragen der Anfragenmenge vereinigen und der neuen Anfrage hinzufügen.
 4. Ist bei keiner Anfrage der Anfragenmenge die HAVING-Klausel leer, dann die HAVING-Klauseln aller Anfragen der Anfragenmenge über ein logisches ODER (OR) vereinigen und der neuen Anfrage hinzufügen, ansonsten bleibt die HAVING-Klausel der neuen Anfrage ebenfalls leer.
 5. Duplikate in der vereinigten HAVING-Klausel entfernen.
 6. Alle in der vereinigten HAVING-Klausel vorkommenden Aggregatterme und außerhalb der Aggregatterme vorkommenden Attribute der vereinigten SELECT-Klausel hinzufügen.
 7. Duplikate in der vereinigten SELECT-Klausel entfernen.

8. Bezeichnerkollisionen in den vereinigten Attributdefinitionen durch Umbenennung beheben.
9. Typ der neuen Anfrage ('Tabelle' oder 'Sicht') bestimmen.
10. Primärschlüssel der neuen Anfrage bestimmen.
11. Allen Anfragen, die Anfragen der Anfragemenge referenzieren, für jede Referenz die Prädikate der HAVING-Klausel der entsprechenden Anfrage zur WHERE-Klausel hinzufügen, falls diese nicht bereits in der WHERE-Klausel enthalten sind. Die Aggregate in diesen Prädikaten müssen zuvor durch die entsprechenden Attribute der vereinigten Anfrage ersetzt werden.
12. Alle Referenzen auf die Anfragen der Anfragemenge und deren Attribute an die neue Anfrage anpassen.
13. Alle Anfragen der Anfragemenge löschen.

Einige Anmerkungen hierzu:

Bei der Regelbedingung und -aktion gelten die schon bei der Regel *Where To Group* erläuterten Besonderheiten und Alternativen.

Ist bei einer Anfrage der Anfragemenge die HAVING-Klausel leer (4.), dann entspricht dies dem Wahrheitswert 'wahr' und es werden alle Tupel selektiert. D.h., dass in diesem Fall auch die HAVING-Klausel der neuen Anfrage alle Tupel selektieren und somit leer bleiben muss. Es ist außerdem zu beachten, dass durch die ODER-Verknüpfung der Prädikate die Beschränkung auf KNF in der HAVING-Klausel verletzt werden kann, was dadurch zu beheben ist, dass man bei der Auswahl der Anfragen nur Anfragen mit einem einzigen Prädikat in der HAVING-Klausel berücksichtigt.

Diese Regel kann eventuell auch zu einer Verschlechterung der Anfragesequenz bezüglich der Ausführungszeit führen. Das in der vorigen Regelbeschreibung (*Merge Where*) aufgestellte Rechenbeispiel kann hier in ähnlicher Weise angewendet werden.

Ich habe diese Regel trotzdem hier aufgeführt, da in der Realität nur selten der schon bei der Regel *Merge Where* beschriebene Worst-Case eintreten wird.

Join To Table

Wird in mehreren Anfragen der gleiche Join durchgeführt, d.h. ein Join über dieselben Tabellen und Sichten, der das gleiche Ergebnis liefert, dann erzeugt diese Regel eine neue Anfrage, die diesen Join durchführt und somit die Joins in den Anfragen ersetzt. Im nachstehenden Beispiel werden in den Anfragen Y und Z die Tabellen F, G und H durch einen Join miteinander verbunden, wobei die Attribute mit (lokalem) Bezeichner a Primärschlüssel der jeweiligen Tabelle sind. Die Regel *Join To Table* lagert nun diesen Join in eine zusätzliche Anfrage X aus:

<pre> CREATE VIEW Y(a,b) AS SELECT F.a,F.b FROM F,G,H,I WHERE F.a=G.a AND F.a=H.a AND F.a=I.a AND </pre>	<pre> CREATE VIEW Z(a) AS SELECT F.a FROM F,G,H,J WHERE F.a=G.a AND G.a=H.a AND G.a=J.a AND F.c=1 </pre>	
↓		
<pre> CREATE VIEW X(a,b,c) AS SELECT F.a,F.b,F.c FROM F,G,H WHERE F.a=G.a F.a=H.a </pre>	<pre> CREATE VIEW Y(a,b) AS SELECT X.a,X.b FROM X,I WHERE X.a=I.a AND </pre>	<pre> CREATE VIEW Z(a) AS SELECT X.a FROM X,J WHERE X.a=J.a X.c=1 </pre>

Die Regel *Join To Table* lautet:

- **Bedingung:** Es existiert eine Anfragemenge, für die gilt:

Fall 1: Es existiert eine Menge von Join-Prädikaten, in denen alle Anfragen der Menge übereinstimmen.

Fall 2: In der FROM-Klausel der Anfragen dieser Menge existieren mehrere Elemente, in denen die Anfragen dieser Menge übereinstimmen und die durch Vergleich der Primärschlüssel direkt oder transitiv, d.h. über Elemente, die eine andere Tabelle bzw. Sicht referenzieren, miteinander verbunden sind.

- **Aktion:**

1. Eine neue Anfrage erzeugen.
2. Referenzen auf die am Join beteiligten Tabellen und Sichten in die FROM-Klausel der neuen Anfrage einfügen.
3. Join-Prädikate der WHERE-Klausel hinzufügen:
 - Fall 1: Die Join-Prädikate, in denen die Anfragen der Anfragemenge übereinstimmen, in die WHERE-Klausel der neuen Anfrage übernehmen.
 - Fall 2: Die Referenzen auf die am Join beteiligten Tabellen und Sichten in der WHERE-Klausel der neuen Anfrage über deren Primärschlüssel verbinden.
4. Die nicht mehr benötigten Join-Prädikate aus den WHERE-Klauseln der Anfragen der Anfragemenge entfernen.
5. Alle Attribute, die jetzt noch in den Anfragen der Anfragemenge vorkommen und die aus den am Join beteiligten Tabellen und Sichten stammen, mit den entsprechenden Attributdefinitionen in die SELECT-Klausel der neuen Anfrage einfügen.
6. Bezeichnerkollisionen in den Attributdefinitionen der neuen Anfrage durch Umbenennung beheben.
7. Duplikate in der SELECT-Klausel der neuen Anfrage entfernen.
8. Referenzen auf die am Join beteiligten Tabellen und Sichten aus den FROM-Klauseln der Anfragen der Anfragemenge entfernen.
9. Eine Referenz auf die neue Anfrage in die FROM-Klausel jeder Anfrage der Anfragemenge einfügen.
10. Alle Referenzen auf Attribute der am Join beteiligten Tabellen und Sichten in den Anfragen der Anfragemenge durch Referenzen auf die entsprechenden Attribute der neuen Anfrage ersetzen.
11. Typ der neuen Anfrage ('Tabelle' oder 'Sicht') bestimmen.

12. Primärschlüssel der neuen Anfrage bestimmen.

Einige Anmerkungen hierzu:

Fall 1 beschreibt den Fall, in dem die am Join beteiligten Tabellen und Sichten auf die gleiche Weise, d.h. über die gleichen Join-Prädikate, miteinander verknüpft sind. Fall 2 betrifft Joins, bei denen die beteiligten Tabellen und Sichten nicht direkt, sondern transitiv über eine 1:1-Beziehung miteinander verbunden sind (vgl. Regel *Eliminate Redundant References*).

Als Typ der neuen Anfrage (11.) sollte ‘Tabelle’ gewählt werden, da diese Anfrage einen Join enthält, der aus den Anfragen der Anfragemenge entfernt wurde, um nicht in jeder Anfrage erneut berechnet zu werden. D.h. es fällt nur einmalig Aufwand an, um diesen Join zu berechnen, und das Ergebnis des Joins wird allen Anfragen, die es benötigen, zur Verfügung gestellt. Wählt man hingegen den Typ ‘Sicht’, dann wird diese Sicht im nachfolgenden Optimierlauf des Datenbanksystems wieder in die Anfragen, die diese Sicht verwenden, eingesetzt mit dem Resultat, dass der Join in jeder dieser Anfragen erneut berechnet wird.

Trifft Regelbedingung Fall 2 zu, dann kann der Primärschlüssel (12.) aus einer der am Join beteiligten Tabellen und Sichten übernommen werden. Bei Regelbedingung Fall 1 erweist sich die Bestimmung des Primärschlüssels als äußerst kompliziert, da hier durch den Join eventuell Duplikate entstehen können. In diesem Fall ist die einfachste Lösung, gar keinen Primärschlüssel zu bestimmen.

Im Fall 1 der Regelbedingung kann es eventuell auch zu einer Verschlechterung der Anfragesequenz bezüglich der Ausführungszeit kommen. Dies ist dann der Fall, wenn das Ergebnis des Joins größer ist als die Summe der Tupelzahlen der Tupelmengen, die der Berechnung des Joins zugrundeliegen, und die eingesparten Join-Berechnungen den dadurch entstehenden Mehraufwand beim Lesen der Tupel nicht aufwiegen. Dies tritt beispielsweise dann auf, wenn die Join-Prädikate Attribute vergleichen, die nicht eindeutig sind, sondern Duplikatwerte enthalten.

4.1.2 Regeln der Klasse 2

Zu dieser Klasse zähle ich alle Regeln, die eine Teilsequenz betrachten und damit die Abhängigkeiten zwischen den Anfragen miteinbeziehen. D.h. der Kontext der Regelbedingung ist nicht auf eine Anfrage beschränkt, sondern bezieht auch Anfragen, die über eine Abhängigkeitsbeziehung mit dieser verbunden sind, mitein. Es handelt sich hierbei meist um Regeln, die in ähnlicher Weise auch bei genesteten Anfragen im Single-Query-Rewrite angewandt werden; die betreffenden Regeln aus dem Single-Query-Rewrite werden dazu so modifiziert, dass sie auch im Multi-Query-Rewrite verwendbar sind.

Predicate Pushdown

Diese Regel verschiebt Selektionen entlang den Abhängigkeitsbeziehungen der Anfragen in Richtung der Basistabellen bzw. -sichten. Das Verschieben von Selektionen ist eine wesentliche Optimierungsheuristik im Single-Query-Rewrite und wird dort auch als “Selection Pushdown” oder “Predicate Pushdown” bezeichnet [M95] [PLH97]. Die im folgenden beschriebene Variante ist eine Anpassung dieser Heuristik an das Multi-Query-Rewrite. Im nachstehenden Beispiel wird das Prädikat $a > 5$ aus den Anfragen Y und Z in die Anfrage X verschoben:

<pre>CREATE VIEW X(a,b) AS SELECT a,b FROM F</pre>	<pre>CREATE VIEW Y(a) AS SELECT X1.a FROM X X1, X X2 WHERE X1.a>5 AND X2.a>5</pre>	<pre>CREATE VIEW Z(a) AS SELECT a FROM X WHERE a>5 AND b=0</pre>
↓	↓	↓
<pre>CREATE VIEW X(a,b) AS SELECT a,b FROM F WHERE a>5</pre>	<pre>CREATE VIEW Y(a) AS SELECT X1.a FROM X X1, X X2</pre>	<pre>CREATE VIEW Z(a) AS SELECT a FROM X WHERE b=0</pre>

Die Regel *Predicate Pushdown* lautet:

- **Bedingung:** Es existiert eine Anfrage, für die gilt:
 - a) Diese Anfrage ist keine Ergebnisanfrage.
 - b) Alle Anfragen, die diese Anfrage referenzieren, stimmen für jede einzelne Referenz in einem Prädikat in der WHERE-Klausel überein, das nur Attribute dieser Referenz enthält.
- **Aktion:**
 1. Referenzen auf Attribute der referenzierten Anfrage im gemeinsamen Prädikat durch die entsprechenden Attributdefinitionen der referenzierten Anfrage ersetzen.
 2. Falls die GROUP-BY-Klausel der referenzierten Anfrage leer ist und das gemeinsame Prädikat noch nicht in der WHERE-Klausel der referenzierten Anfrage enthalten ist, dann das in 1. modifizierte gemeinsame Prädikat der WHERE-Klausel der referenzierten Anfrage hinzufügen.
 3. Falls die GROUP-BY-Klausel der referenzierten Anfrage nicht leer ist und das gemeinsame Prädikat noch nicht in der HAVING-Klausel der referenzierten Anfrage enthalten ist, dann das in 1. modifizierte gemeinsame Prädikat der HAVING-Klausel der referenzierten Anfrage hinzufügen.
 4. Aus allen referenzierenden Anfragen das gemeinsame Prädikat entfernen.

Einige Anmerkungen hierzu:

Im Gegensatz zum Single-Query-Rewrite bei dem genau ein Prädikat verschoben wird, muss hier darauf geachtet werden, dass für jede Referenz auf die Anfrage, in die dieses verschoben werden soll, ein solches Prädikat existiert (b). Die Übereinstimmung von Prädikaten zu ermitteln, ist hier einfacher als bei den vorgestellten Regeln der Klasse 1, da die in Frage kommenden Prädikate nur Attribute enthalten dürfen, die zu derselben Referenz in der FROM-Klausel gehören, und für jede Referenz auf die Anfrage ein solches Prädikat existieren muss. D.h. es muss keine Zuordnung für die FROM-Klausel-Elemente zweier Anfragen gefunden werden.

Die Anfrage, in die das gemeinsame Prädikat verschoben werden soll, darf keine Ergebnisanfrage sein, da das Prädikat als Selektion die Tupelmenge weiter einschränken und somit die Semantik der Ergebnistabelle/-sicht verändern kann. Bei den referenzierenden Anfragen ändert sich durch die Verschiebung nichts am Inhalt der Tabellen und Sichten, die diese produzieren.

Da keine Vereinigung von Anfragen stattfindet und Selektionen nur den Wertebereich einschränken, aber keinen Einfluss auf Duplikateliminierung und Eindeutigkeit eines Tupels

haben, dürfen die Anfrage und die referenzierenden Anfragen ein `DISTINCT` enthalten. Eine Verschiebung von Selektionen ändert nichts an diesem `DISTINCT`.

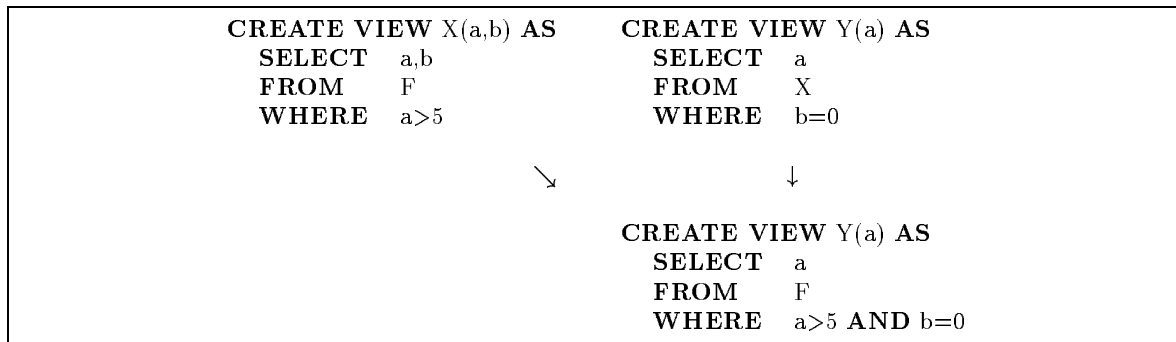
Da das gemeinsame Prädikat in eine Anfrage verschoben wird, die von den Anfragen, die dieses Prädikat enthalten referenziert wird, müssen die darin vorkommenden Attribute durch die entsprechenden Attributdefinitionen bzw. durch die zugehörigen Terme in der `SELECT`-Klausel der referenzierten Anfrage substituiert werden (1.).

Es muss darauf geachtet werden, ob die Anfrage, in die das Prädikat verschoben wird, eine `GROUP-BY`-Klausel enthält oder nicht. Enthält sie keine `GROUP-BY`-Klausel, dann wird das Prädikat in die `WHERE`-Klausel eingefügt (2.), sonst in die `HAVING`-Klausel (3.).

Bei der Konzeption des Regelsystems ist zu beachten, dass diese Regel zusammen mit der Regel *Replicate By Transitivity* schnell zu einer Endlosschleife führen kann [PLH97]. Dies muss durch entsprechende Mechanismen verhindert werden (siehe Abschnitt 5.3).

Concat Queries

Diese Regel vereinigt aufeinanderfolgende Anfragen, d.h. Anfragen die direkt über eine Abhängigkeitsbeziehung miteinander verbunden sind. Im nachstehenden Beispiel werden die Anfragen `X` und `Y` vereinigt, sodass nur noch Anfrage `Y` übrigbleibt:



Die Regel *Concat Queries* lautet:

- **Bedingung:** Es existiert eine Anfrage, für die gilt:
 - a) Diese Anfrage ist keine Ergebnisanfrage.
 - b) Die Anfrage wird nur von einer einzigen Anfrage und dort von genau einem `FROM`-Klausel-Element referenziert.
 - c) Es gilt einer der folgenden vier Fälle:
 - Fall 1: Die Anfrage enthält keine `GROUP-BY`-Klausel und kein `DISTINCT`.
 - Fall 2: Die Anfrage und die Anfrage, die diese referenziert, enthalten keine `GROUP-BY`-Klausel, aber ein `DISTINCT`.
 - Fall 3: Die Anfrage enthält eine `GROUP-BY`-Klausel, die Anfrage, die diese referenziert, enthält alle Attribute aus dieser `GROUP-BY`-Klausel in ihrer `SELECT`-Klausel, enthält keine `GROUP-BY`-Klausel und ist von keiner anderen Tabelle oder Sicht abhängig.
 - Fall 4: Die Anfrage enthält eine `GROUP-BY`-Klausel, die Anfrage, die diese referenziert, enthält nicht alle Attribute aus dieser `GROUP-BY`-Klausel in ihrer `SELECT`-Klausel, ist keine Ergebnisanfrage, enthält keine `GROUP-BY`-Klausel und ist von keiner anderen Tabelle oder Sicht abhängig.

- **Aktion:**

1. Die FROM-Klausel der Anfrage an die FROM-Klausel der referenzierenden Anfrage anhängen.
2. Bezeichnerkollisionen in der vereinigten FROM-Klausel durch Umbenennung beheben.
3. Entferne in der vereinigten FROM-Klausel das Element, das die Anfrage referenziert.
4. Die Referenzen auf Attribute der referenzierten Anfrage in der referenzierenden Anfrage durch die entsprechenden Attributdefinitionen der referenzierten Anfrage, wobei zuvor noch die Umbenennungen von 2. durchgeführt werden müssen, ersetzen.
5. *Fall 1 und 2:*
 - 5.1 In der WHERE-Klausel der referenzierten Anfrage die Umbenennungen von 2. durchführen und diese an die WHERE-Klausel der referenzierenden Anfrage anhängen.
 - 5.2 Duplikate in der vereinigten WHERE-Klausel entfernen.
6. *Fall 3 und 4:*
 - 6.1 In der GROUP-BY-Klausel der referenzierten Anfrage die Umbenennungen von 2. durchführen und diese in die referenzierende Anfrage übernehmen.
 - 6.2 In der HAVING-Klausel der referenzierten Anfrage, die Umbenennungen von 2. durchführen, und diese in die referenzierende Anfrage übernehmen.
 - 6.3 Die WHERE-Klausel aus der referenzierenden Anfrage entfernen und an die HAVING-Klausel der referenzierenden Anfrage anhängen.
 - 6.4 Duplikate in der HAVING-Klausel der referenzierenden Anfrage entfernen.
 - 6.5 *Fall 4:* Der SELECT-Klausel der referenzierenden Anfrage die Attribute aus der GROUP-BY-Klausel der referenzierten Anfrage, die noch nicht in ihr enthalten sind, hinzufügen.
 - 6.6 In der WHERE-Klausel der referenzierten Anfrage die Umbenennungen von 2. durchführen und diese in die referenzierende Anfrage übernehmen.
7. Entferne in der WHERE-Klausel Join-Prädikate, der FORM $a = a$, d.h. Äquivalenzvergleiche bei denen die linke mit der rechten Seite übereinstimmt.
8. Typ der referenzierenden Anfrage ('Tabelle' oder 'Sicht') bestimmen.
9. Primärschlüssel der referenzierenden Anfrage bestimmen.
10. Die referenzierte Anfrage löschen.

Einige Anmerkungen hierzu:

Die referenzierte Anfrage darf keine Ergebnisanfrage sein (a), da sie bei der Vereinigung entfernt wird (10.). Außerdem darf sie nur von einer einzigen Anfrage und dort von genau einem FROM-Klausel-Element referenziert werden (b), da sie sonst nach der Vereinigung mit der referenzierenden Anfrage nicht entfernt werden kann.

Die Regelbedingung deckt vier verschiedene Fälle unterschiedlicher Konstellationen von referenzierter und referenzierender Anfrage ab (c). Es existieren aber noch weitere Fälle, die hierdurch nicht abgedeckt werden. Enthält nämlich die referenzierte Anfrage ein DISTINCT und keine GROUP-BY-Klausel, so muss auch die referenzierende Anfrage ein DISTINCT enthalten, damit die Regelbedingung (Fall 2) erfüllt ist. Wäre eine Duplikateliminierung jedoch unnötig, da die Tupel schon eindeutig sind, dann wäre damit auch das DISTINCT überflüssig und Fall 1 käme zur Anwendung. D.h. es müsste überprüft werden, ob ein in einer Anfrage enthaltenes DISTINCT überhaupt seinen Zweck erfüllt oder überflüssig ist und daher entfernt werden kann bzw. nicht beachtet werden muss. Wenn die Anfrage einen Primärschlüssel hat, dann ist die Lösung trivial, da dieser die Eindeutigkeit der Tupel garantiert.

Fall 3 und Fall 4 unterscheiden sich eigentlich nur darin, dass einmal die Gruppierungsattribute der referenzierten Anfrage vollständig in der SELECT-Klausel der referenzierenden Klausel aufgeführt sind und das andere mal nicht. Sind nicht alle Gruppierungsattribute aufgeführt, so müssen die fehlenden der SELECT-Klausel der konkatenierten Anfrage hinzugefügt werden (6.5). Dies ist allerdings nur möglich, wenn die konkatenierte, d.h. die referenzierende Anfrage keine Ergebnisanfrage ist.

Bei der Konkatenierung ist wichtig, dass in der referenzierenden Anfrage die Referenzen auf Attribute der referenzierten Anfrage durch die entsprechenden Terme in der SELECT-Klausel der referenzierten Anfrage ersetzt werden (4.).

Bei Fall 1 und 2 entspricht die referenzierte Anfrage einer Art Filter und/oder Join, da sie keine GROUP-BY-Klausel besitzt. Bei Fall 3 und 4 hingegen führt die referenzierte Anfrage eine Gruppierung durch und die referenzierende Anfrage stellt nun eine Art Filter und/oder Join dar. Daher wird die WHERE-Klausel der referenzierenden Anfrage Teil der HAVING-Klausel der konkatenierten Anfrage.

Die Bestimmung des Anfragetyps und des Primärschlüssels wurde bereits bei der Beschreibung der Regel *Merge Select* erläutert.

Eliminate Unused Attributes

Diese Regel entfernt Attribute die überflüssig sind, d.h. Duplikate und Attribute, die nicht benötigt werden. Letztere können beispielsweise durch zuvor ausgeführte Optimierungsregeln entstanden sein. Im nachstehenden Beispiel handelt es sich bei Attribut *X.b* um ein Duplikat und bei Attribut *X.c* um ein Attribut, das von keiner Anfrage benutzt wird:

<pre>CREATE VIEW X(a,b,c) AS SELECT a,a,c FROM F</pre>	<pre>CREATE VIEW Y(s) AS SELECT a+b FROM X</pre>
↓	↓
<pre>CREATE VIEW X(a) AS SELECT a FROM F</pre>	<pre>CREATE VIEW Y(s) AS SELECT a+a FROM X</pre>

Die Regel *Eliminate Unused Attributes* lautet:

- **Bedingung:** Es existiert eine Anfrage, für die gilt:
 - a) Die Anfrage ist keine Ergebnisanfrage.
 - b) In der SELECT-Klausel der Anfrage existieren Duplikate und/oder
 - c) In der SELECT-Klausel der Anfrage existieren Attribute, die von keiner referenzierenden Anfrage verwendet werden, und die Anfrage enthält kein DISTINCT.
- **Aktion:**
 1. Entferne alle Duplikate aus der SELECT-Klausel.
 2. Entferne alle Attribute aus der SELECT-Klausel, die von den referenzierenden Anfragen nicht benutzt werden.
 3. Alle Referenzen auf die Attribute der Anfrage an die modifizierten Attributdefinitionen bzw. die modifizierte SELECT-Klausel anpassen.

Einige Anmerkungen hierzu:

Die Anfrage darf keine Ergebnisanfrage sein (a), da bei Ergebnisanfragen die Attributdefinitionen nicht verändert werden dürfen.

Enthält die Anfrage ein `DISTINCT`, so kann die Entfernung von in referenzierenden Anfragen nicht verwendeten Attributen die Zahl der Tupel in der Tabelle bzw. Sicht der Anfrage verändern. Daher ist ein `DISTINCT` genau dann erlaubt, wenn nur Duplikate vorkommen (b, c).

4.1.3 Regeln der Klasse 3

Hierzu zähle ich alle Regeln, deren Kontext auf eine Anfrage beschränkt ist. D.h. diese Regeln betrachten einzelne Anfragen, eventuell sogar nur bestimmte Komponenten innerhalb einer Anfrage. Es handelt sich daher nicht um typische Multi-Query-Rewrite-Regeln, sondern um Regeln aus dem Single-Query-Rewrite und Regeln, die Redundanzen und Duplikate, die zuvor ausgeführte Regeln hinterlassen haben, beseitigen.

Eliminate Redundant References

Diese Regel entfernt redundante und damit unnötige Referenzen auf dieselbe Tabelle bzw. Sicht. Redundante Referenzen können entstehen, wenn Anfragen vereinigt werden und Referenzen auf zuvor unterschiedliche Anfragen nun dieselbe Anfrage referenzieren. Im nachstehenden Beispiel werden die redundanten Referenzen *F2* und *F3*, die über den Primärschlüssel *a* mit *F1* verbunden sind, aus der Anfrage *X* entfernt:

<pre> CREATE VIEW X(c,d,e,f) AS SELECT F1.c, F2.d, F3.e, F4.f FROM F F1, F F2, F F3, F F4 WHERE F1.a=F2.a AND F1.a=F3.a AND F2.b=F4.b </pre> <p style="text-align: center;">↓</p> <pre> CREATE VIEW X(c,d,e,f) AS SELECT F1.c, F1.d, F1.e, F4.f FROM F F1, F F4 WHERE F1.b=F4.b </pre>
--

Die Regel *Eliminate Redundant References* lautet:

- **Bedingung:** Es existiert eine Anfrage, für die gilt:
 - a) In der `FROM`-Klausel dieser Anfrage existieren mehrere Elemente, die dieselbe Tabelle bzw. Sicht referenzieren und durch Vergleich der Primärschlüssel direkt oder transitiv, d.h. über Elemente, die eine andere Tabelle bzw. Sicht referenzieren, miteinander verbunden sind.
- **Aktion:**
 1. Wähle eine Referenz aus der gefundenen Menge von Referenzen, die die anderen Referenzen der Menge ersetzen soll und nicht gelöscht wird.
 2. Ändere den Bezeichner der Quelle in den Attributen, deren Quelle eine Referenz der Menge ist, in allen Klauseln, sodass sie alle die gewählte Referenz als Quelle besitzen.

3. Entferne in der WHERE-Klausel Join-Prädikate, der FORM $a = a$, d.h. Äquivalenzvergleiche bei denen die linke mit der rechten Seite übereinstimmt.
4. Entferne alle Referenzen der Menge bis auf die gewählte Referenz aus der FROM-Klausel.

Einige Anmerkungen hierzu:

Wichtig ist, dass die Referenzen der Menge über den Primärschlüssel miteinander verbunden sind (a), weil dadurch sichergestellt wird, dass es sich dabei um eine 1:1-Beziehung handelt. In diesem Fall reicht eine Referenz aus, da durch den 1:1-Join keine neue sondern nur redundante Information hinzukommt. Die Suche nach solchen Referenzen entspricht einer Berechnung der transitiven Hülle. D.h. zuerst muss eine Menge von Referenzen ermittelt werden, deren Elemente direkt oder transitiv durch Join über Primärschlüssel miteinander verbunden sind. Aus dieser Menge von Referenzen muss eine Untermenge ausgewählt werden, die nur noch Referenzen auf dieselbe Tabelle bzw. Anfrage enthält.

In 3. werden die Join-Prädikate, die die Redundanten Referenzen miteinander verbunden haben, die aber nun nicht mehr benötigt werden, aus der WHERE-Klausel der Anfrage entfernt.

Replicate By Transitivity

Diese Regel nutzt Transitivitätsbeziehungen zwischen Attributen, um Prädikate entsprechend modifiziert zu replizieren. Diese Optimierungsheuristik wird auch im Single-Query-Rewrite angewandt, meist in Verbindung mit einer anschließenden Verschiebung des originalen und des replizierten Prädikats (*Predicate Pushdown*) [PLH97]. Im nachstehenden Beispiel wird von dem Prädikat $F.a=G.b$ eine Transitivitätsbeziehung zwischen den Attributen $F.a$ und $G.b$ aufgebaut, wodurch das Prädikat $F.a>5$ als $G.b>5$ repliziert werden kann:

<pre> CREATE VIEW X(a,b) AS SELECT F.a,G.b FROM F, G WHERE F.a=G.b AND F.a>5 </pre> <p style="text-align: center;">↓</p> <pre> CREATE VIEW X(a,b) AS SELECT F.a,G.b FROM F, G WHERE F.a=G.b AND F.a>5 AND G.b>5 </pre>
--

Die Regel *Replicate By Transitivity* lautet:

- **Bedingung:** Es existiert eine Anfrage, für die gilt:
 - a) In der WHERE-Klausel der Anfrage existiert ein Prädikat der Form $a \theta t$ oder $t \theta a$, wobei a ein Attribut, t ein Term und θ eine beliebige Vergleichsoperation ist.
 - b) In der WHERE-Klausel der Anfrage existiert ein Prädikat der Form $a = b$ oder $b = a$, wobei a das obige Attribut und b ein anderes Attribut ist.
- **Aktion:**
 1. Füge der WHERE-Klausel der Anfrage das Prädikat $b \theta t$ bzw. $t \theta b$ hinzu.

Einige Anmerkungen hierzu:

Bei der Konzeption des Regelsystems ist zu beachten, dass diese Regel zusammen mit der Regel *Predicate Pushdown* schnell zu einer Endlosschleife führen kann [PLH97]. Dies muss durch entsprechende Mechanismen verhindert werden (siehe Abschnitt 5.3).

4.1.4 Erweiterung der Regelmenge

Die in den vorigen Unterabschnitten vorgestellten Regeln bilden eine Regelmenge, die speziell auf die Problematik des Multi-Query-Rewrite und die beschriebene Untermenge von SQL92 zugeschnitten ist.

Dieser Regelmenge können natürlich sämtliche Regeln aus dem Bereich Single-Query-Rewrite hinzugefügt werden. Als besonders sinnvoll erweist sich dies, wenn man die zugrundegelegte SQL-Untermenge um Unteranfragen und Mengenoperationen erweitert. Entsprechende Regeln sind in [M95] zu finden.

Die Regelmenge kann ebenfalls um Regeln, die ein überflüssiges DISTINCT entfernen oder Informationen bezüglich Duplikateliminierung und Eindeutigkeit von Attributen und Tupeln über die Abhängigkeitsbeziehungen propagieren, erweitert werden (vgl. [PHH92]).

Hebt man die Beschränkung auf, dass WHERE- und HAVING-Klausel in KNF vorliegen müssen, und läßt damit auch komplexere Prädikate zu, dann kommen Regeln hinzu, die diese Klauseln wieder auf KNF bzw. die gesamte Anfrage auf eine bestimmte Normalform bringen.

Zu den Regeln der Klasse 1 können auch Regeln hinzugezählt werden, die auf Erkenntnissen aus dem Bereich Automated Summary Tables basieren. D.h. bei Anfragen, die in einer Teilmenge der GROUP-BY-Klausel übereinstimmen, könnte ähnlich zu *Join To Table* eine gemeinsame Gruppierung vorgelagert werden. Z.B. könnte eine Anfrage hinzukommen, die eine GROUP-BY-Klausel enthält, die durch Vereinigung der GROUP-BY-Klauseln der Anfragen der Anfragemenge entstanden ist, und als gemeinsame Basis dient. Dadurch muss nicht in jeder Anfrage erneut über die ursprüngliche Datenbasis gruppiert und aggregiert werden, sondern es kann auf ein gemeinsames Zwischenergebnis zugegriffen werden, aus dem dann die jeweils benötigten Aggregate berechnet werden können. Die Anfragen der Anfragemenge stellen somit Cubes bzw. Summary Tables grober Granularität dar und die zusätzliche Anfrage einen Cube bzw. eine Summary Table feinerer Granularität. Dies entspricht also der Problematik der Cube-Berechnung. Es kann sogar eine ganze Berechnungshierarchie mit mehreren zusätzlichen Anfragen bzw. Summary Tables für eine gegebene Anfragemenge entwickelt werden. Da solche Vorausberechnungen nicht immer machbar sind und die Auswahl der zu materialisierenden Summary Tables nicht trivial ist und auf Kostenabschätzungen basiert, habe ich auf die Beschreibung solcher Regeln verzichtet. Verfahren und Bewertungsschemata für die Auswahl zu materialisierender Sichten eines Data Cube sind in [HRU96] zu finden.

4.2 Interndarstellung

In den folgenden Unterabschnitten werden die im Single-Query-Rewrite verwendeten Interndarstellungen auf ihre Verwendbarkeit im Multi-Query-Rewrite überprüft und es wird eine passende Interndarstellung entwickelt.

4.2.1 Bewertung der vorgestellten Interndarstellungen

Die in Kapitel 3 betrachteten Interndarstellungen und Regelnotationen sind nicht ohne weiteres auf die Problematik des Multi-Query-Rewrite übertragbar und teilweise hierfür auch nicht geeignet. Die nachstehende Liste enthält eine Menge von Anforderungen, die an eine an das Multi-Query-Rewrite angepasste Interndarstellung gestellt werden:

- Anforderungen, die sich aus der in der Aufgabenstellung dieser Diplomarbeit verankerten Forderung, dass die optimierte Abfragesequenz wieder in SQL verfügbar sein muss, ergeben:
 - Eine optimierte Abfragesequenz muss aus der Interndarstellung wieder in die SQL-Darstellung umgewandelt werden können.
 - Die Rückübersetzung sollte bei einer in der Rewrite-Phase nicht veränderten Abfragesequenz möglichst wieder zur ursprünglichen SQL-Abfragesequenz führen.
 - Bei der Rückübersetzung dürfen die durchgeführten Optimierungen nicht “verlorengehen”, d.h. beispielsweise, dass die Anfragen auch in der Interndarstellung voneinander abgegrenzt sein müssen.
 - Die Interndarstellung muss auch Metadaten wie z.B. Datentyp oder Primärschlüsseleigenschaft, erfassen.
- Anforderungen, die sich aus der Betrachtung der Rewrite-Regeln ergeben:
 - Die Interndarstellung sollte die SQL-typische Klauselstruktur nachbilden, da in den Formulierungen der Regelbedingungen und -aktionen die verschiedenen Klauseltypen unterschieden werden und einzelne Klauseln ausgewählt, miteinander verglichen und manipuliert werden.
 - Die Interndarstellung sollte eine effiziente Navigation entlang den Abhängigkeitsbeziehungen möglich machen, da die Regelbedingungen von Regeln der Klasse 2 auch die Abhängigkeiten zwischen den Anfragen miteinbeziehen und bei fast allen Regeln Umbenennungen und Änderungen entlang den Abhängigkeitsbeziehungen propagiert werden.
- Allgemeine Anforderungen:
 - Die Interndarstellung sollte flexibel und erweiterbar sein.
 - Die Interndarstellung sollte für den menschlichen Betrachter lesbar und verständlich sein.

Unter diesen Gesichtspunkten werden nun die vorgestellten Interndarstellungen des Single-Query-Rewrite auf ihre Verwendbarkeit für das Multi-Query-Rewrite überprüft.

SQL (als Zeichenkette)

Die Repräsentation von SQL-Anweisungen als Zeichenkette mag für die Implementierung von Single-Query-Rewrite-Strategien gut geeignet sein. Beim Multi-Query-Rewrite sind jedoch mehrere Anfragen miteinander zu vergleichen, um vollständige und partielle Übereinstimmungen festzustellen; hierbei können einige Probleme auftreten, die beim Single-Query-Rewrite gar nicht existieren. Die folgenden Anfragen zeigen ein kleines Beispiel, bei dem ein

simpler Vergleich der Zeichenketten versagt:

<pre>CREATE VIEW X(a) AS SELECT a FROM W WHERE c = d AND e = 1</pre>	\Rightarrow	<pre>CREATE VIEW Y(b) AS SELECT b FROM W WHERE c = d AND e = 1</pre>	\Rightarrow	<pre>CREATE VIEW XY(a,b) AS SELECT a, b FROM W WHERE c = d AND e = 1</pre>
<pre>CREATE VIEW X(a) AS SELECT a FROM W WHERE c = d AND e = 1</pre>	\Rightarrow	<pre>CREATE VIEW Z(b) AS SELECT b FROM W WHERE e = 1 AND d = c</pre>	\Rightarrow	<pre>CREATE VIEW XZ(a,b) AS SELECT a, b FROM W WHERE c = d AND e = 1</pre>

Vergleicht man in der oberen Zeile den Rumpf der linken Sichtdefinition mit dem Rumpf der mittleren Sichtdefinition, so fällt auf, dass bei beiden sowohl die FROM-Klauseln als auch die WHERE-Klauseln identisch sind und damit die Regel *Merge Select* angewandt werden kann, um die beiden Sichten zu einer einzigen Sicht XY, die die Attribute beider SELECT-Klauseln enthält, zu vereinigen. Sind also die Zeichenketten, die die beiden Sichtdefinitionen repräsentieren, bis auf die SELECT-Klausel Zeichen-für-Zeichen identisch, so kann dies mit Hilfe einfacher Zeichenkettenvergleiche und Pattern Matching erkannt werden. Anders ist dies bei den Sichtdefinitionen in der unteren Zeile. Sicht Z ist semantisch identisch mit Sicht Y; daher könnte hier ebenfalls eine Verschmelzung wie bei den Sichten X und Y durchgeführt werden. Die beiden Zeichenketten unterscheiden sich jedoch syntaktisch nicht nur in der SELECT-Klausel, sondern auch in der WHERE-Klausel; bei Z ist die Reihenfolge der beiden über die Konjunktion verbundenen Ausdrücke vertauscht und die Seiten der Vergleichsoperation sind vertauscht. Diese Permutationen beruhen auf der Kommutativität der AND-Operation und der Symmetrie des Äquivalenzvergleichs. In diesem Fall kann die Übereinstimmung der beiden WHERE-Klauseln nicht durch einen zeichenweisen Vergleich der Zeichenketten sichergestellt werden, sondern es ist eine geparste Darstellung erforderlich. Noch deutlicher wird dies bei den semantisch äquivalenten Prädikaten $a < b$ und $b > a$ oder bei unterschiedlichen lokalen Bezeichnern zueinandergehörender FROM-Klausel-Elemente. Ebenso gibt es Regeln, bei denen die Anfragen in der WHERE-Klausel nur partiell übereinstimmen müssen oder dürfen; hier tritt das gleiche Problem auf. Diese Anfragerrepräsentation ist daher für das Multi-Query-Rewrite nicht geeignet. Trotz dieser Einschränkungen findet bei Oracle 8i ein Rewrite zur Realisierung einer transparenten Verwendung von Automated Summary Tables auf der Basis eines Zeichenkettenvergleichs statt [B99].

Relationenalgebra, GENTREES und COKO-KOLA

Strukturen mit einem prozeduralen oder funktionalen Ansatz wie die Relationenalgebra, GENTREES oder COKO-KOLA sind für das Multi-Query-Rewrite nicht geeignet. Dies hat verschiedene Gründe. Die Abbildung von der weitestgehend deskriptiven SQL-Darstellung in eine prozedurale oder funktionale Darstellung, d.h. einen Operatorbaum oder einen algebraischen Term, ist meist nicht eindeutig, ebenso die Umkehrabbildung. Die Umkehrabbildung ist zudem nicht trivial und kann bei einer in der Rewrite-Phase nicht veränderten Anfrage zu einer anderen als der ursprünglichen Anfrage führen. Dies liegt an der unterschiedlichen Betrachtungsweise; während bei einer deskriptiven Darstellung die Beziehungen und Restriktionen zwischen den Datenquellen beschrieben werden, die zu dem gewünschten Ergebnis führen, steht bei den prozeduralen bzw. funktionalen Darstellungen im Vordergrund, welche Operationen zur Beantwortung einer Anfrage benötigt werden und in welcher Reihenfolge die-

se Operationen angewandt werden. Für eine Darstellung in deskriptiver Form gibt es daher eine Menge von Alternativen in einer prozeduralen bzw. funktionalen Notation, die sich nur in der Reihenfolge der Operationen unterscheiden. Bei der Rückübersetzung gehen natürlich Informationen über diese Reihenfolge und damit zusammenhängende Optimierungen verloren. Betrachtet man den Abstraktionsgrad, dann befinden sich deskriptive Darstellungen auf einem höheren Niveau wie prozedurale bzw. funktionale. D.h. die Optimierung würde auf einer Darstellung mit einem anderen Abstraktionsgrad arbeiten wie die Eingabe- und Ausgabedaten. Berücksichtigt man außerdem, dass sich die Regelbeschreibungen im vorigen Abschnitt an der deskriptiven Klauselstruktur von SQL orientieren, dann erweist sich die Wahl einer prozedralen oder funktionalen Darstellungsform nicht als sinnvoll.

QGM

QGM ist aufgrund seiner an der SQL-Struktur orientierten Tabellenoperatoren und seiner Mischung aus Operator- und Operandengraph auch für das Multi-Query-Rewrite unter den hier gegebenen Anforderungen geeignet. Es muss allerdings um entsprechende Kantentypen und/oder Operatoren erweitert werden, um eine Sequenz von Anfragen mit ihren Abhängigkeiten darstellen und daraus wieder die einzelnen Anfragen identifizieren zu können. Die eher abstrakten Datenstrukturen eines QGM-Graphen müssen jedoch bei einer Implementierung dieses Modells auf geeignete Speicherungsstrukturen abgebildet werden. Eine Möglichkeit ist die Abbildung auf Datenstrukturen der zur Implementierung verwendeten Programmiersprache. Die meisten Programmiersprachen stellen hierfür eine große Palette von Datentypen und Mechanismen zur Verfügung, auf Basis derer der Programmierer wiederum komplexere Datentypen konstruieren kann. Dies hat jedoch den Nachteil, dass die verwendeten Datenstrukturen programmiersprachenspezifisch oder sogar bibliothekspezifisch sind und zum Datenaustausch zwischen Programmen ein Austauschformat spezifiziert werden muss. Desweiteren müssen sämtliche Schnittstellen für den Zugriff auf und die Modifikation dieser Datenstrukturen definiert und die entsprechenden Operationen implementiert werden. Eine weitere Möglichkeit ist die Speicherung der internen Anfragedarstellung in einem standardisierten Datenformat wie XML [W3C00a]. XML hat den Vorteil, dass bereits standardisierte Schnittstellen zur Verarbeitung von XML-Dokumenten wie SAX (Simple API for XML) und DOM (Document Object Model) existieren und Implementierungen für diverse Programmiersprachen verfügbar sind. Da XML-Dokumente nicht codiert gespeichert sind, können sie in jedem beliebigen Texteditor betrachtet und geändert werden. Dies alles vereinfacht die Implementierung eines Prototypen und ermöglicht die Konzentration auf das wesentliche, die Rewrite-Regeln. Bei der Abbildung von QGM auf XML ist zu beachten, dass XML-Dokumente Bäume darstellen, wobei die Elemente und die Zeichendaten zwischen einem Start- und End-Tag die Knoten bilden, während es sich bei QGM zumindest innerhalb eines Transformationsoperators nicht um einen Baum sondern um einen Graphen handelt. D.h. die Graphstruktur muss auf eine Baumstruktur abgebildet werden. Hierfür gibt es verschiedene Möglichkeiten, die ich hier nur kurz ansprechen möchte. In einem QGM-Graphen lassen sich die folgenden drei Kantenarten erkennen:

- Kanten zwischen Tupelvariablen.
- Kanten zwischen einer Tupelvariable und einem Tabellenoperator.
- Kanten zwischen Tupelvariablen und dem Kopf eines Tabellenoperators.

Kanten zwischen Tupelvariablen und Kanten zwischen Tupelvariablen und dem Kopf eines Tabellenoperators können beliebig viele Knoten (Tupelvariablen) miteinander verbinden. Es gibt nun die Alternativen, die verschiedenen Kantenarten auf verschiedene Elementtypen abzubilden oder aber auf einen einzigen Elementtyp, der damit eine Art generischen Kantentyp darstellt. Die beiden letzteren Kantentypen kann man eventuell auch als “Zeiger” in Form von Elementattributen realisieren. Behält man die Struktur mit den Tabellenoperatoren und den darin enthaltenen Knoten bei, dann stellt sich die Frage, ob man die Kanten auch innerhalb des Tabellenoperatorelements speichert oder die Kanten aller Tabellenoperatoren einer Anfrage zu einer gemeinsamen Kantenmenge zusammenfasst. Durch die Einschränkung der Anfragesprache sind beide Varianten denkbar; lässt man jedoch die Nestung von Anfragen zu, dann können auch Tabellenoperator-übergreifende Prädikatskanten auftreten, was nur durch letztere Variante abgedeckt wird. Desweiteren wäre zu überlegen, ob man einige Tabellenoperatoren zusammenfassen kann (z.B. GROUP BY und HAVING). Bisher wurden eigentlich immer zwei Abstraktionsebenen getrennt betrachtet. Zum einen die Ebene, auf der die Anfrage als eine Menge miteinander verbundener Tabellenoperatoren dargestellt wird, zum anderen die Ebene, die den Inhalt dieser Tabellenoperatoren genauer beschreibt. Es wäre aber auch möglich diese beiden Ebenen bzw. die zugehörigen Graphen nicht getrennt zu betrachten und zu speichern, sondern sie zu einem einzigen Graphen aus Knoten und Kanten zu integrieren, indem man die Graphen der Tabellenoperatoren in den Operatorbaum einsetzt. Dazu muss man die einzelnen Komponenten eines Tabellenoperators wie beispielsweise den Kopf eines Tabellenoperators ebenfalls als Knoten darstellen.

Gegen eine XML-Darstellung von QGM als Interndarstellung spricht, dass die Repräsentation einer Anfrage bzw. Anfragesequenz als eine Menge von Knoten und Kanten für den menschlichen Betrachter schlecht lesbar und unübersichtlich ist.

4.2.2 Eigene XML-Interndarstellung

Aufgrund dessen, dass die Repräsentation einer Anfragesequenz als Graph, d.h. in Form von Knoten und Kanten, auf XML [W3C00a] abgebildet schlecht lesbar ist, habe ich mich für eine recht simple XML-Darstellung entschieden. Es handelt sich hierbei um eine Syntaxbaum-ähnliche Darstellung, die sich an der Klauselstruktur von SQL orientiert und den Anforderungen, die sich aus den Regelbeschreibungen ergeben, am nächsten kommt. Auch die Terme sind vollständig syntaktisch zerlegt. Im Gegensatz zu einem Syntaxbaum, wie ihn ein Parser erzeugen würde, sind hier die Elemente, die die Anfragen, die Klauseln und die Klausel-elemente repräsentieren jeweils auf der gleichen Ebene innerhalb der Baumstruktur angeordnet; es bilden sich somit Mengen gleichartiger Elemente. Das Beispiel in Abbildung 4.2 soll diesen Unterschied verdeutlichen; es zeigt wie Klauseln und Prädikate in einem Syntaxbaum und in der hier beschriebenen Interndarstellung angeordnet sind. Der Vorteil dieser Darstellung ist, dass sie gut lesbar ist und dass eine Rückübersetzung trivial und immer möglich ist. Da die Reihenfolge der Prädikate, Terme, usw. aus den SQL-Anweisungen in die Interndarstellung übernommen wird, liefert die Rückübersetzung - bis auf einige wenige Ausnahmen - bei einer Anfrage, die während der Optimierung nicht verändert wurde, wieder die ursprünglichen SQL-Anweisungen. Es folgt nun eine Beschreibung der einzelnen Komponenten der DTD dieser XML-Darstellung:

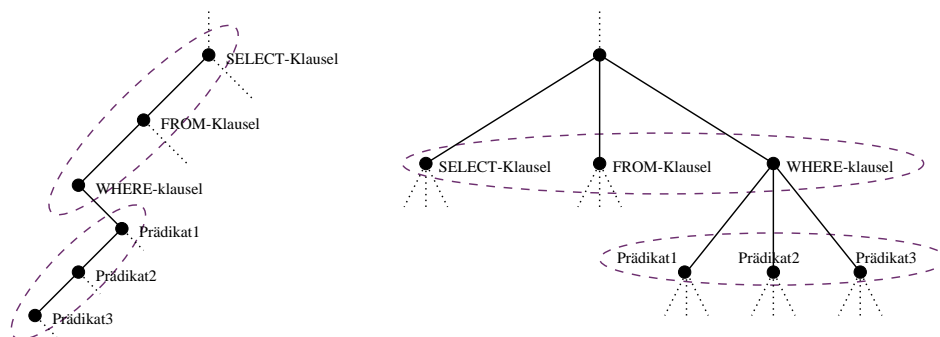


Abbildung 4.2: Beispiel für eine Anfrage als Syntaxbaum (links) und in Interndarstellung (rechts).

```

1 <?xml version="1.0" encoding="iso-8859-1"?>
2 <!ELEMENT sequence (query)*>
3 <!ELEMENT query (referenced-by, select-clause, from-clause, where-clause?,
4 (group-by-clause, having-clause?)?, order-by-clause?)>
5 <!ATTLIST query name ID #REQUIRED
6 distinct (no | yes) #REQUIRED
7 type (table | view) #REQUIRED
8 result (no | yes) #REQUIRED>

```

Das Element *sequence* ist das Wurzelement, das die gesamte Sequenz repräsentiert. Dessen Kinder sind die einzelnen Anfragen der Sequenz. Sie werden durch Elemente vom Typ *query* repräsentiert. Das Attribut *name* enthält den Bezeichner der Anfrage, der innerhalb der Sequenz auch als Identifikator dient. Das Attribut *distinct* betrifft die Duplikateliminierung und ist auf ‘yes’ gesetzt, wenn die SELECT-Klausel der Anfrage ein ‘DISTINCT’ enthält, ansonsten auf ‘no’. Das Attribut *type* ist auf ‘table’ gesetzt, falls die Ergebnisse der Anfrage in einer Tabelle gespeichert werden, und auf ‘view’, falls sie in einer Sicht gespeichert werden. Das Attribut *result* ist auf ‘yes’ gesetzt, wenn es sich um eine Ergebnisanfrage handelt, sonst auf ‘no’.

```

9 <!ENTITY % aggr "(avg | max | min | sum | count | stddev)">
10 <!ENTITY % arithm "(plus | minus | mul | div)">
11 <!ENTITY % comp "(eq | neq | leq | geq | le | gr | in | nin | betw | nbetw |
12 like | nlike | null | nnull)">
13 <!ENTITY % expr "(%aggr; | %arithm; | constant | attribute)">

```

In Zeile 9 bis 13 werden Parameter-Entities definiert, die jeweils alle Aggregatfunktionen, alle arithmetischen Operatoren und alle Vergleichsfunktionen zusammenfassen und eine Parameter-Entity, die für arithmetische Ausdrücke steht.

```

14 <!ELEMENT   referenced-by   (referencing-query)*>
15 <!ELEMENT   select-clause   (attribute-definition)+>
16 <!ELEMENT   from-clause     (source)+>
17 <!ELEMENT   where-clause    (%comp; | disjunction)+>
18 <!ELEMENT   group-by-clause (attribute)+>
19 <!ELEMENT   having-clause   (%comp; | disjunction)+>
20 <!ELEMENT   order-by-clause (attribute)+>
21 <!ATTLIST   order-by-clause order   (asc | desc) #REQUIRED>

```

Die Zeilen 15 bis 21 repräsentieren die verschiedenen Klauseln einer Anfrage. Das Attribut *order* des Elements *order-by-clause* legt fest, ob aufsteigend oder absteigend sortiert werden soll. Das Element *referenced-by* in Zeile 14 enthält als Elemente Verweise auf alle Anfragen, die diese Anfrage referenzieren, und realisiert damit eine beidseitige Verknüpfung von referenzierender und referenzierter Anfrage. Es handelt sich hierbei um redundante Information, die jedoch der Optimierung der auf dieser Darstellung arbeitenden Algorithmen dienen kann.

```

22 <!ELEMENT   referencing-query   EMPTY>
23 <!ATTLIST   referencing-query   name       IDREF #REQUIRED>
24 <!ELEMENT   attribute-definition (%expr;)>
25 <!ATTLIST   attribute-definition name        CDATA #REQUIRED
26                                     type        CDATA #REQUIRED
27                                     constraint (notnull | unique | primarykey) #IMPLIED>
28 <!ELEMENT   disjunction        (%comp;, (%comp;)+)>
29 <!ELEMENT   source              EMPTY>
30 <!ATTLIST   source              name        CDATA #REQUIRED
31                                     alias       CDATA #REQUIRED>

```

Die Zeilen 22 bis 31 beschreiben die Elemente der verschiedenen Klauseltypen. Bei dem Element *referencing-query* handelt es sich um den Verweis auf eine Anfrage, die diese Anfrage referenziert. Das Attribut *name* enthält den Bezeichner der referenzierenden Anfrage. Das Element *attribute-definition* realisiert die Definition eines Attributs der Anfrage. Das Attribut *name* enthält den Bezeichner des Attributs, das Attribut *type* den Datentyp und das Attribut *constraint* eventuell bestehende Einschränkungen. Das Element *disjunction* stellt eine Disjunktion innerhalb der WHERE- oder HAVING-Klausel dar. Bei dem Element *source* handelt es sich um ein FROM-Klausel-Element. Das Attribut *name* enthält den tatsächlichen Bezeichner der referenzierten Tabelle/Sicht, das Attribut *alias* enthält den lokalen Bezeichner, der innerhalb der Anfrage für diese Referenz verwendet wird. Wird in der FROM-Klausel einer referenzierten Tabelle/Sicht kein neuer Bezeichner zugewiesen, dann ist der Bezeichner von *name* in *alias* zu übernehmen.

```

32 <!ELEMENT   constant          EMPTY>
33 <!ATTLIST   constant          value      CDATA #REQUIRED>
34 <!ELEMENT   string            EMPTY>
35 <!ATTLIST   string            value      CDATA #REQUIRED>
36 <!ELEMENT   attribute         EMPTY>
37 <!ATTLIST   attribute         name       CDATA #REQUIRED
38                                     source     CDATA #REQUIRED>

```

In Zeile 32 bis 38 werden die atomaren Elemente eines Terms wie Konstanten, Strings und Attribute spezifiziert. Der Wert der Konstante bzw. des Strings ist in dem Attribut *value*

abgelegt. Das Attribut *name* des Elements *attribute* enthält den Bezeichner des Attributs, jedoch ohne das Tabellen/Sicht-Präfix. Dieses Präfix wird in dem Attribut *source* abgelegt. Da hiermit ein Element der FROM-Klausel referenziert wird, muss der Inhalt von *source* gleich dem Inhalt des *alias*-Attributs eines *source*-Elements dieser Anfrage sein. Die Attribute *name* und *source* ergeben zusammen eine eindeutige Referenz auf ein Attribut einer anderen Tabelle oder Sicht.

```

39 <!ELEMENT eq      (%expr;, %expr;)>
40 <!ELEMENT neq     (%expr;, %expr;)>
41 <!ELEMENT leq     (%expr;, %expr;)>
42 <!ELEMENT geq     (%expr;, %expr;)>
43 <!ELEMENT le      (%expr;, %expr;)>
44 <!ELEMENT gr      (%expr;, %expr;)>
45 <!ELEMENT in      (%expr;, constant+)>
46 <!ELEMENT nin     (%expr;, constant+)>
47 <!ELEMENT betw    (%expr;, constant, constant)>
48 <!ELEMENT nbetw   (%expr;, constant, constant)>
49 <!ELEMENT like     (attribute, string)>
50 <!ELEMENT nlike    (attribute, string)>
51 <!ELEMENT null     EMPTY>
52 <!ELEMENT nnull    EMPTY>

```

In Zeile 39 bis 52 wird jeder der unterstützten Vergleichsfunktionen ein Element zugewiesen. Der Reihe nach sind dies die Vergleichsfunktionen: “=”, “≠”, “≤”, “≥”, “<”, “>”, “IN”, “NOT IN”, “BETWEEN”, “NOT BETWEEN”, “LIKE”, “NOT LIKE”, “IS NULL” und “IS NOT NULL”. Ein Element für die logische Operation NOT ist nicht nötig, da für jede Vergleichsoperation auch die entsprechende inverse Vergleichsoperation existiert; d.h. $NOT(a < b)$ wird als $(a \geq b)$ dargestellt.

```

53 <!ELEMENT plus     (%expr;, %expr;)>
54 <!ELEMENT minus    (%expr;, (%expr;)?)>
55 <!ELEMENT mul      (%expr;, %expr;)>
56 <!ELEMENT div      (%expr;, %expr;)>
57 <!ELEMENT avg      (%expr;)>
58 <!ATTLIST avg      distinct (no | yes) #REQUIRED>
59 <!ELEMENT max      (%expr;)>
60 <!ATTLIST max      distinct (no | yes) #REQUIRED>
61 <!ELEMENT min      (%expr;)>
62 <!ATTLIST min      distinct (no | yes) #REQUIRED>
63 <!ELEMENT sum      (%expr;)>
64 <!ATTLIST sum      distinct (no | yes) #REQUIRED>
65 <!ELEMENT count    (attribute? | EMPTY)>
66 <!ATTLIST count    distinct (no | yes) #REQUIRED>
67 <!ELEMENT stddev   (%expr;)>
68 <!ATTLIST stddev   distinct (no | yes) #REQUIRED>

```

In Zeile 53 bis 68 wird jedem arithmetischen Operator und jeder Aggregatfunktion ein Element zugewiesen.

Alternativ könnte man jeweils alle Aggregatfunktionen, alle arithmetischen Operatoren und alle Vergleichsfunktionen zu einem Element zusammenfassen, das dann als generische Funktion/Operator dient, und den Bezeichner der Funktion bzw. des Operators in einem

Attribut festhalten. Dies hätte den Vorteil, dass die zugrundegelegte Anfragesprache um neue Funktionen und Operatoren erweitert werden könnte, ohne dass neue Element eingeführt werden müssen. Da sich jedoch einige Funktionen/Operatoren in der Zahl und Art ihrer Parameter unterscheiden und sie daher eine unterschiedliche Syntax haben, habe ich jeder Funktion und jedem Operator ein eigenes Element zugewiesen.

Es müssen auch einige semantische Bedingungen eingehalten werden, die sich nicht in Form einer DTD ausdrücken lassen, die aber für die korrekte Darstellung einer Anfragesequenz entscheidend sind. Es gilt daher folgendes:

- Das Attribut *name* eines Elements vom Typ *referencing-query* muss den Bezeichner einer Anfrage der Anfragesequenz enthalten; d.h. es muss ein Element vom Typ *query* existieren, dessen Attribut *name* ebenfalls diesen Bezeichner enthält. Außerdem muss die Anfrage mit diesem Bezeichner die Anfrage, zu der das Element vom Typ *referencing-query* gehört, referenzieren.
- Das Attribut *name* eines Elements vom Typ *attribut-definition* muss innerhalb der SELECT-Klausel, zu der das Element gehört, eindeutig sein.
- Das Attribut *alias* eines Elements vom Typ *source* muss innerhalb der FROM-Klausel, zu der das Element gehört, eindeutig sein.
- Das Attribut *source* eines Elements vom Typ *attribute* muss den Alias-Bezeichner einer Quelle aus der FROM-Klausel der Anfrage, die dieses Element enthält, tragen; d.h. innerhalb dieser Anfrage muss ein Element vom Typ *source* existieren, dessen Attribut *alias* ebenfalls diesen Bezeichner enthält.
- Das Attribut *name* eines Elements vom Typ *attribute*, muss den Bezeichner einer Attributdefinition der durch das Attribut *source* gegebenen Quelle, tragen; d.h. das durch das Element vom Typ *attribute* referenzierte Attribut muss auch tatsächlich existieren.
- Die Aggregatfunktionen sollen in korrekter Weise verwendet werden, d.h. Aggregatfunktionen dürfen nicht geschachtelt werden und nicht innerhalb der WHERE-Klausel verwendet werden.
- Das Attribut *value* eines Elements vom Typ *constant* muss einen numerischen Wert enthalten.

Für einige Regeln sind auch Metadaten der Basistabellen und -sichten relevant, z.B. zur Ermittlung des Datentyps oder der Primärschlüsseleigenschaft eines Attributs einer Basistabelle oder -sicht. Diese sollten getrennt von der Anfragesequenz gespeichert werden, da Basistabellen und -sichten nicht das Ergebnis einer Anfrage der Anfragesequenz und somit eigentlich auch nicht Teil der Anfragesequenz sind. Außerdem werden diese Daten von sämtlichen Sequenzen in gleicher Weise benötigt. Die folgenden Zeilen zeigen die DTD für eine XML-Repräsentation der für die Regeln relevanten Metadaten:

```

1  <?xml version="1.0" encoding="iso-8859-1"?>
2  <!ELEMENT  meta-data      (relation)*>
3  <!ELEMENT  relation      (attribute-definition)+>
4  <!ATTLIST  relation      name          ID #REQUIRED
5                                     distinct    (no | yes) #IMPLIED
6                                     type        (table | view) #REQUIRED>
7  <!ELEMENT  attribute-definition EMPTY>
8  <!ATTLIST  attribute-definition name        CDATA #REQUIRED
9                                     type        CDATA #REQUIRED
10                                    constraint  (notnull | unique | primarykey) #IMPLIED>

```

Das Element *meta-data* ist das Wurzelement. Dessen Kinder sind Elemente vom Typ *relation*. Sie repräsentieren die Relationen, d.h. Tabellen und Sichten, von denen Metadaten bekannt sind. Das Attribut *name* enthält den Bezeichner der Relation. Das Attribut *distinct* dient der Duplikateliminierung und ist auf ‘yes’ gesetzt, wenn die SELECT-Klausel der Relation ein ‘DISTINCT’ enthält, ansonsten auf ‘no’; ist allerdings nicht bekannt, ob die SELECT-Klausel der Relation ein ‘DISTINCT’ enthält oder nicht, kann das Attribut *distinct* auch weggelassen werden. Das Attribut *type* ist auf ‘table’ gesetzt, falls es sich bei der Relation um eine Tabelle handelt, und auf ‘view’, falls es sich dabei um eine Sicht handelt. Elemente vom Typ *attribute-definition* stellen Attributdefinitionen dar und bilden die Kinder von Elementen des Typs *relation*. Das Attribut *name* enthält den Bezeichner des Attributs, das Attribut *type* den Datentyp und das Attribut *constraint* eventuell bestehende Einschränkungen.

In Abschnitt 5.4 findet man ein Beispiel für eine Sequenz und die zugehörigen Metadaten in Interndarstellung.

Kapitel 5

Der Prototyp

Im Rahmen dieser Diplomarbeit habe ich einen Prototyp erstellt, in dem eine Auswahl der in Kapitel 4 aufgeführten Regeln implementiert ist. In diesem Kapitel werden nun alle Themen behandelt, die diesen Prototypen betreffen.

5.1 Bewertung einiger XML-Technologien

In diesem Abschnitt werden kurz einige XML-Technologien, die ursprünglich als Möglichkeit zur Implementierung der Multi-Query-Rewrite-Regeln in Betracht gezogen wurden, vorgestellt und auf ihre tatsächliche Verwendbarkeit hin überprüft.

5.1.1 XSLT

XSLT [W3C99] [K01] ist eine Sprache, die dafür entwickelt wurde, ein XML-Dokument in ein anderes XML-Dokument zu transformieren. Es handelt sich hierbei um eine deklarative, regelbasierte Sprache, die so entworfen wurde, dass die Regelausführung frei von Seiteneffekten ist. XML-Dokumente werden hier als Bäume interpretiert. Ein XSLT-Prozessor wandelt den Eingabebaum mit Hilfe der Regelmenge, dem sogenannten Stylesheet, in einen Ausgabebaum um. Die Regeln sind ebenfalls in XML formuliert. Sie bestehen aus einem Pattern und einem Regelkörper. Das Pattern ist ein XPath-Ausdruck und spezifiziert die Art von Knoten im Eingabebaum, die das Feuern der Regel bewirkt. Im Regelkörper wird der entsprechende Teil des Ausgabebaums produziert; d.h. es werden Elemente erzeugt, Elementinhalte aus dem Eingabebaum gelesen und in den Ausgabebaum geschrieben und es wird meist selektiv die Ausführung weiterer Regeln induziert. Es können auch Variablen definiert, Bedingungen formuliert und Berechnungen durchgeführt werden. XSLT eignet sich somit besonders gut für die Umformung eines XML-Dokuments in ein anderes XML-Dokument ähnlicher Struktur, zum Ausfiltern von Daten und zur Anreicherung von XML-Dokumenten um präsentations-spezifische Elemente. Es existieren schon mehrere Implementierungen für XSLT, die teilweise auch Schnittstellen zur Einbindung externer Funktionen anbieten.

Es folgt ein Beispiel für ein Stylesheet, das auf ein XML-Dokument in der vorgestellten Interndarstellung angewandt werden kann und als Ausgabe ein HTML-Dokument erzeugt, das die Bezeichner aller Anfragen der Sequenz getrennt nach Ergebnisanfragen und sonstigen Anfragen jeweils durch Kommas separiert auflistet:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <body>
        <h1> Ergebnisanfragen: </h1>
        <xsl:apply-templates select="sequence/query[@result='yes']"/>
        <h1> Sonstige Anfragen: </h1>
        <xsl:apply-templates select="sequence/query[@result='no']"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="sequence/query">
    <xsl:value-of select="@name"/>
    <xsl:if test="not(position()=last())">, </xsl:if>
  </xsl:template>

</xsl:stylesheet>

```

XSLT eignet sich jedoch nicht dazu, mehrere Knoten zu vergleichen und nach einer Menge von Knoten, zwischen denen eine bestimmte Beziehung besteht, zu suchen, da sich dies nicht durch ein Pattern ausdrücken lässt. Dies wäre aber beispielsweise für die Regel *Merge Select* notwendig. D.h. XSLT ist nicht mächtig genug, um darin Multi-Query-Rewrite-Regeln zu formulieren.

5.1.2 XQuery

XQuery [W3C01] ist eine Anfragesprache für XML-Dokumente, in die Ideen und Konzepte aus unterschiedlichen XML-Anfragesprachen, aber auch aus SQL und OQL eingeflossen sind; aus SQL wurde beispielsweise die Idee der Klauselstruktur übernommen. XML bietet auch Kontrollstrukturen und erlaubt Joins über mehrere Dokumente, die Nestung von Ausdrücken und die Dereferenzierung von Attributen des Typs IDREF(s); auf Basis von Ausdrücken können sogar eigene Funktionen definiert werden. Der Standardisierungsprozess von XQuery ist jedoch noch nicht abgeschlossen und die Zahl der Implementierungen ist daher auch nicht groß. XQuery kann eventuell bei einigen Multi-Query-Rewrite-Regeln zur Formulierung der Regelbedingung oder von Teilen der Regelbedingung verwendet werden. Dies kann jedoch zu recht umfangreichen, komplexen und unübersichtlichen Ausdrücken führen. Beispielsweise gibt es keine Funktion, die den Vergleich zweier Knoten-Sequenzen ermöglicht; der Vergleichsoperator “=” auf zwei Knoten-Sequenzen angewandt, überprüft nur, ob in der ersten Sequenz ein Element existiert, das mit einem Element in der zweiten Sequenz übereinstimmt; d.h. ein Vergleich zweier Knoten-Sequenzen muss recht umständlich über Joins oder Mengenoperationen realisiert werden. Die Effizienz solcher XQuery-Anfragen hängt sehr stark von der Qualität des verwendeten Anfrageoptimierers ab. Aufgrund dieser Punkte und da der Aktionsteil der Regel sowieso separat in einer Programmiersprache abgefasst werden muss, sehe ich von einer Formulierung der Regelbedingungen in XQuery ab. Es ist auch fraglich, bei welchen Regeln der Bedingungsteil vollständig in Form einer XQuery-Anfrage formuliert werden kann.

5.1.3 DOM

DOM (Document Object Model) [W3C00b] [B00] ist eine programmiersprachen-unabhängige Schnittstelle, die die logische Struktur von Dokumenten und Funktionen für den Zugriff auf und die Modifikation von XML-Dokumenten definiert. Die XML-Dokumente werden hier ebenfalls als Bäume gesehen. Die Knoten repräsentieren Elemente, Text, Kommentare, Verarbeitungsanweisungen, CDATA-Sektionen, Entity-Referenzen und -Deklarationen, Notationen und ganze Dokumente. Knoten werden auch zur Repräsentation von Attributen verwendet, obwohl diese Knoten eigentlich nicht Teil eines DOM-Baums sind. Es gibt noch ein paar zusätzliche Knotentypen, die es ermöglichen Gruppen von Knoten zu verwalten und die die Verschiebung von Knoten vereinfachen. Die Knotenschnittstelle ist das Herz des DOM-Schemas. Sie bietet Methoden um die Charakteristiken eines Knotens zu erfragen, um im Baum zu navigieren und um einen Knoten bzw. Baum zu manipulieren. Damit ist DOM sehr gut geeignet, um die Interndarstellung aus dem Regelcode heraus zu durchwandern und zu manipulieren. Ein großer Vorteil ist auch, dass die Java Language Bindings standardisiert sind und man somit trotzdem unabhängig von der Bibliothek ist, die man verwendet. Für die Implementierung der Regeln habe ich hauptsächlich die Klassen *Node*, *Element* und *NodeList* verwendet. Die Klasse *Node* ist die Oberklasse aller Baumknoten-Klassen. Sie stellt sämtliche Methoden zur Navigation und zur Manipulation der Kinderknoten zur Verfügung. *Element* ist eine Unterklasse von *Node*. Aus dieser Klasse habe ich die Methoden, die den direkten Zugriff auf einzelne Elementattribute über deren Bezeichner ermöglichen, verwendet. Die Klasse *NodeList* bietet eine Datenstruktur, die einem Feld von Knoten entspricht und auf deren Elemente über Positionsindex zugegriffen werden kann. Die Kinder eines Knotens kann man sich als *NodeList* zurückgeben lassen, was besonders vorteilhaft in den Algorithmen der Regeln der Klasse 1 bei der Suche nach Anfragemengen eingesetzt werden kann.

5.2 Beschreibung der Regel-Implementierungen

Die in Kapitel 3 vorgestellten Regelsprachen sind nicht mächtig genug, um darin Multi-Query-Rewrite-Regeln zu formulieren. Eine geeignete formale Regelnotation müsste All- und Existenzquantoren und auf die Klauselstruktur abgestimmte Operatoren zur Verfügung stellen. D.h. es käme nur eine an der Prädikatenlogik orientierte Notation in Frage, was allerdings zu komplexen und unübersichtlichen Regelformulierungen führen würde, die eventuell eine schlechte Performance des Regelinterpreters zur Folge hätten. Aus diesen Gründen und um die größt mögliche Flexibilität zu haben, habe ich mich für die Kodierung der Regeln in einer Programmiersprache entschieden. Als Programmiersprache habe ich die Programmiersprache Java gewählt, weil Java objektorientiert ist, auf sämtlichen Plattformen verfügbar ist, die Java Language Bindings von DOM standardisiert sind und SUN sogar eine eigene Implementierung von DOM samt XML-Parser anbietet.

Für jede Rewrite-Regel wird eine eigene Klasse angelegt, wobei diese Klassen alle von der gemeinsamen Oberklasse *Rule* abgeleitet sind. Jede Regelklasse enthält die Methoden *condition* und *action*, die vom Regelsystem aufgerufen werden. Die Methode *condition* repräsentiert die Regelbedingung und hat als Parameter einen DOM-Baum mit den notwendigen Metadaten und einen DOM-Baum mit der Anfragesequenz. Sie gibt einen Wahrheitswert zurück, der angibt, ob die Anfragesequenz die Regelbedingung erfüllt. Die Methode *condition* darf jedoch die Anfragesequenz nicht modifizieren; hierfür ist die Methode *action* vorgesehen, die die Regelaktion repräsentiert. Sie hat keine Parameter und auch keinen Rückgabewert.

Der Ablauf einer Regelausführung sieht wie folgt aus: Zuerst wird vom Regelsystem die Methode *condition* aufgerufen. Werden hierbei Dinge (z.B. Anfragemengen) berechnet, die wiederum auch für die Regelaktion relevant sind, dann können diese in Objektvariablen zwischengespeichert werden, um sie nicht während der Regelaktion erneut berechnen zu müssen. Ist die Regelbedingung erfüllt, dann kann vom Regelsystem die Methode *action* aufgerufen werden; allerdings nur, wenn die Anfragesequenz seit Aufruf der Methode *condition* nicht verändert wurde. Danach kann wieder die Methode *condition* aufgerufen werden, um die Anwendbarkeit der Regel erneut zu überprüfen. Um Endlosschleifen im Regelsystem zu vermeiden, muss sichergestellt sein, dass die Methode *condition* nur dann 'true' zurückgibt, wenn bei einem anschließenden Aufruf der Methode *action* auch wirklich die Anfragesequenz entsprechend modifiziert wird, d.h. es muss in der Regelbedingung auch überprüft werden, ob alle für die Regelaktion relevanten Metadaten vorhanden sind. Sowohl nach Ausführung der Methode *condition* als auch nach Ausführung der Methode *action* muss sich die Anfragesequenz in einem konsistenten Zustand befinden. Die Aufspaltung der Regel in Bedingungs- und Aktionsteil hat den Vorteil, dass man bei der Entwicklung der Kontrollstrategien des Regelsystems mehr Möglichkeiten hat. So kann beispielsweise zuerst überprüft werden, welche Regeln überhaupt feuern können, und basierend auf dem Ergebnis dieser Überprüfung eine Regel ausgewählt werden.

In dem im Rahmen dieser Diplomarbeit erstellten Prototyp wurden fünf der in Kapitel 4 vorgestellten Regeln implementiert. In den folgenden Unterabschnitten werden anhand von Codeausschnitten für jede dieser fünf Regeln die wichtigsten und interessantesten Algorithmen kurz beschrieben. Da die einzelnen Schritte, die bei der Ausführung der Regelaktionen notwendig sind, schon in Kapitel 4 aufgelistet und beschrieben worden sind, werden sich die folgenden Unterabschnitte auf die Regelbedingungen konzentrieren, die bisher nur in deskriptiver Form dargestellt worden sind. Die Codeausschnitte sind Auszüge aus dem Quellcode des Prototypen. Aus Platzgründen wurden jedoch die Kommentare entfernt. Die Zeilen in den Codeausschnitten sind zwar nummeriert, die Zeilennummern stimmen aber nicht mit den Zeilennummern des Quellcodes überein, sondern sind dazu gedacht, um aus dem Text heraus auf bestimmte Stellen im Code verweisen zu können.

5.2.1 Merge Select

Bei der Regel *Merge Select* wird nach einer Menge von Anfragen gesucht, wobei die Anfragen in allen Klauseln außer der SELECT-Klausel übereinstimmen müssen, keine Ergebnisanfragen sein dürfen und kein DISTINCT enthalten dürfen (siehe Abschnitt 4.1.1). Genauer betrachtet wird hier eigentlich nach einer Menge von Anfragemengen gesucht aus der eine oder mehrere Anfragemengen ausgewählt werden können. Der zugehörige Algorithmus ist relativ komplex, lässt sich aber wie in Abbildung 5.1 dargestellt in mehrere Schichten zerlegen. In der obersten Schicht werden die Anfragemengen berechnet. Hierzu ist ein Vergleich von Anfragen notwendig. Die benötigten Vergleichsoperationen betrachten Anfragen und deren Inhalt und sind daher Teil der Anfrageschicht. Die Operationen zum Vergleich der Klauseln werden von der Klauselschicht bereitgestellt. Diese greift wiederum auf Operationen der Klausелеlementschicht zu. Aufgrund des unterschiedlichen Aufbaus und der unterschiedlichen Semantik der einzelnen Klauseltypen, kann diese Schicht bei den Operationen der WHERE- und HAVING-Klausel weiter unterteilt werden in Prädikatschicht und Termschicht.

Dieses Schichtenmodell lässt sich bei allen Regeln der Klasse 1 anwenden, da in allen Regeln dieser Klasse Anfragemengen berechnet werden, die auf bestimmten Gemeinsamkeiten und

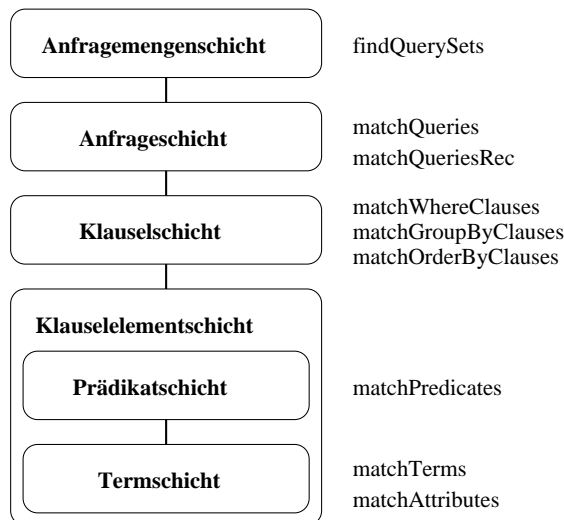


Abbildung 5.1: Schichtenmodell für die Regel *Merge Select* mit den zugehörigen Methoden.

Unterschieden in den Klauseln der Anfragen beruhen. Die Zahl und Semantik der Funktionen auf den einzelnen Schichten kann jedoch von Regel zu Regel unterschiedlich sein.

Anfragemengenschicht

Das Problem eine Menge von Anfragen zu finden, die in bestimmten Klauseln übereinstimmen, kann auf einen paarweisen Vergleich sämtlicher Anfragen reduziert werden. Der Algorithmus arbeitet wie folgt: Man nimmt die erste Anfrage der Sequenz und vergleicht sie nacheinander mit den restlichen Anfragen. Alle Anfragen, die mit der ersten Anfrage die Regelbedingung erfüllen bilden mit dieser zusammen eine Anfragemenge. Nun nimmt man die zweite Anfrage und vergleicht sie mit der dritten Anfrage und allen nachfolgenden. Alle Anfragen, die mit der zweiten Anfrage die Regelbedingung erfüllen bilden wiederum mit dieser zusammen eine Anfragemenge. Nun nimmt man die dritte Anfrage und vergleicht sie mit der vierten Anfrage und allen nachfolgenden, usw. Ist eine Anfrage schon Element einer zuvor gefundenen Anfragemenge, dann wird sie nicht mehr zu Vergleichen herangezogen, da es sich bei der Bedingung der Regel *Merge Select* um eine Äquivalenzrelation handelt. D.h. erfüllt Anfrage Q1 mit Anfrage Q2 und mit Anfrage Q3 die Regelbedingung, dann erfüllt auch Q2 mit Q3 die Regelbedingung (Transitivität); erfüllt nun Q1 mit einer Anfrage Qx die Regelbedingung nicht, dann erfüllt entsprechend auch Q2 mit Qx und Q3 mit Qx die Regelbedingung nicht; daher würde ein Vergleich der Anfragen Q2 und Q3 mit anderen Anfragen keine neue Information liefern als der Vergleich von Q1 mit allen anderen Anfragen und ist daher unnötig. Ebenso liefert der Vergleich von Q1 mit Qx das gleiche Ergebnis wie der Vergleich von Qx mit Q1 (Symmetrie).

Codeausschnitt 1

```

1  private boolean findQuerySets()
2  {
3      NodeList sequence_ML = sequence.getChildNodes();
4      int l = sequence_ML.getLength();
5      boolean[] matched = new boolean[l];
6  }

```

```

7     for (int i = 0; i < l-1; i++)
8         if (matched[i] == false) {
9             boolean setExists = false;
10            for (int j = i+1; j < l; j++) {
11                Hashtable matchSet = new Hashtable();
12                if (matched[j] == false &&
13                    matchQueries(sequence_ML.item(i), sequence_ML.item(j), matchSet)) {
14                    if (setExists == false) {
15                        setExists = true;
16                        querySets.addElement(new Vector());
17                        ((Vector) querySets.lastElement()).addElement(sequence_ML.item(i));
18                        matchSets.addElement(new Vector());
19                        matched[i] = true;
20                    }
21                    ((Vector) querySets.lastElement()).addElement(sequence_ML.item(j));
22                    ((Vector) matchSets.lastElement()).addElement(matchSet);
23                    matched[j] = true;
24                }
25            }
26        }
27    return (!querySets.isEmpty());
28 }

```

Die Methode *findQuerySets* in Codeausschnitt 1 zeigt wie dieser Algorithmus in Java implementiert werden kann. *querySets* und *matchSets* sind Variablen des Regelobjekts. *querySets* ist eine Liste, deren Elemente wiederum Listen mit Anfragen sind, d.h. *querySets* repräsentiert eine Menge von Anfragemengen. *matchSets* ist eine Liste, deren Elemente Listen mit Hashtabellen sind. Diese Hashtabellen enthalten die Zuordnungen der FROM-Klausel-Elemente zweier Anfragen. Zu Beginn sind *querySets* und *matchSets* leer. Das Feld *matched* enthält ein Flag für jede Anfrage der Anfragemenge, das anzeigt, ob die zugehörige Anfrage Teil einer Anfragemenge ist oder nicht.

Die beiden verschachtelten Schleifen (Zeile 7 bis 26 und 10 bis 24) realisieren den im vorigen Abschnitt beschriebenen Algorithmus. Die Bedingungen in Zeile 8 und 12 stellen sicher, dass eine Anfrage, die bereits Element einer zuvor berechneten Anfragemenge ist, nicht mit weiteren Anfragen verglichen wird. Mit dem Flag *setexists* wird sichergestellt, dass nur dann in *querysets* Platz für eine neue Anfragemenge geschaffen wird, wenn auch tatsächlich eine neue Anfragemenge gefunden wurde. In Zeile 13 wird die Funktion *matchQueries* aufgerufen, die die beiden als Parameter übergebenen Anfragen vergleicht und zurückgibt, ob eine Übereinstimmung im Sinne der Regelbedingung vorliegt. Konnte eine Übereinstimmung festgestellt werden, dann enthält der Parameter *matchSet* eine passende Zuordnung der FROM-Klausel-Elemente der beiden Anfragen. Ist dies die erste Übereinstimmung der *i*-ten Anfrage der Anfragesequenz mit einer anderen Anfrage, dann wird eine neue Anfragemenge gebildet (Zeile 16), die *i*-te Anfrage dieser Anfragemenge hinzugefügt (Zeile 17), eine neue Menge für Zuordnungen von FROM-Klausel-Elementen gebildet (Zeile 18) und das Flag der *i*-ten Anfrage gesetzt (Zeile 19), da sie nun Teil einer Anfragemenge ist. Unabhängig davon wird bei jeder Übereinstimmung die *j*-te Anfrage der Anfragesequenz an die aktuelle Anfragemenge angefügt (Zeile 21), ebenso wird die in *matchSet* enthaltene Zuordnung der FROM-Klausel-Elemente an die aktuelle Menge von Zuordnungen angefügt (Zeile 22). Außerdem wird noch das Flag der *j*-ten Anfrage gesetzt (Zeile 23), da sie nun ebenfalls Teil einer Anfragemenge ist. D.h. die *x*-te Zuordnung der Menge von Zuordnungen enthält die Zuordnung zwischen den FROM-Klausel-Elementen der ersten Anfrage der Anfragemenge und der *x+1*-ten Anfrage der Anfragemenge. Es ist noch zu erwähnen, dass *i* und *j* mit dem Wert '0' beginnen, d.h. die '0' entspricht der ersten Anfrage der Anfragesequenz, die '1' der zweiten Anfrage

der Anfragesequenz, usw. Am Ende der Methode *findQuerySets* wird zurückgegeben, ob eine Anfragemenge gefunden wurde oder nicht (Zeile 27). Abbildung 5.2 zeigt ein Beispiel für die Anwendung von *findQuerySets*.

Gegeben ist eine Sequenz aus 8 Anfragen, gesucht sind die Anfragemengen.
 Es seien die Anfragen 1, 3 und 5 im Sinne der Regelbedingung äquivalent, ebenso die Anfragen 2 und 4.

Der Ablauf des Algorithmus (vereinfacht dargestellt):

matchQueries(Anfrage 1, Anfrage 2) = FALSE, matchQueries(Anfrage 1, Anfrage 3) = TRUE,
 matchQueries(Anfrage 1, Anfrage 4) = FALSE, matchQueries(Anfrage 1, Anfrage 5) = TRUE,
 matchQueries(Anfrage 1, Anfrage 6) = FALSE, matchQueries(Anfrage 1, Anfrage 7) = FALSE,
 matchQueries(Anfrage 1, Anfrage 8) = FALSE
 ⇒ Die Anfragen 1, 3 und 5 bilden eine Anfragegruppe und müssen nicht mehr getestet werden.

matchQueries(Anfrage 2, Anfrage 4) = TRUE, matchQueries(Anfrage 2, Anfrage 6) = FALSE,
 matchQueries(Anfrage 2, Anfrage 7) = FALSE, matchQueries(Anfrage 2, Anfrage 8) = FALSE
 ⇒ Die Anfragen 2 und 4 bilden eine Anfragegruppe und müssen nicht mehr getestet werden.

matchQueries(Anfrage 6, Anfrage 7) = FALSE, matchQueries(Anfrage 6, Anfrage 8) = FALSE
 matchQueries(Anfrage 7, Anfrage 8) = FALSE

Abbildung 5.2: Beispiel zur Methode *findQuerySets*.

Bei einer Sequenz aus n Anfragen sind maximal $\lfloor n/2 \rfloor$ Anfragemengen möglich. Im Worst-Case sind alle Anfragen paarweise verschieden, d.h. es gibt keine Anfragemenge, die die Regelbedingung erfüllt. In diesem Fall wird $n * (n - 1)/2$ mal die Funktion *matchQueries* aufgerufen. Im Best Case sind alle Anfragen im Sinne der Regelbedingung äquivalent. In diesem Fall werden nur $n - 1$ Vergleiche mit der ersten Anfrage durchgeführt.

Sind nicht sämtliche Anfragemengen von Interesse, sondern wird nur nach einer beliebigen Anfragemenge gesucht, dann kann, sobald eine Anfragemenge gefunden wurde, die äußere Schleife abgebrochen werden. Dazu müsste die for-Schleife in Zeile 7 in eine while-Schleife umgewandelt werden.

Anfrageschicht

Codeausschnitt 2 zeigt die rekursive Implementierung von *matchQueries*. Es handelt sich hierbei um einen Backtracking-Algorithmus. Er arbeitet wie folgt: Für die erste Referenz in der FROM-Klausel der ersten der beiden zu vergleichenden Anfragen wird in der FROM-Klausel der zweiten Anfrage nach einer Referenz gesucht, die dieselbe Tabelle/Sicht referenziert. Existiert eine solche Referenz, so hat man eine mögliche Zuordnung gefunden und kann für die zweite Referenz in der FROM-Klausel der ersten Anfrage eine passende Referenz in der FROM-Klausel der zweiten Anfrage suchen, usw. Konnte jeder Referenz in der einen Anfrage eine Referenz in der anderen Anfrage zugeordnet werden, so werden die restlichen Klauseln außer der SELECT-Klausel unter dieser Zuordnung auf Übereinstimmung geprüft. Liefert diese Überprüfung keine Übereinstimmung, so wird ein Schritt zurückgegangen und versucht die vorletzte Referenz der ersten Anfrage einer anderen Referenz der zweiten Anfrage zuzordnen. Gelingt dies nicht wird wiederum ein Schritt zurückgegangen zur vorvorletz-

ten Referenz der ersten Anfrage, usw. Auf diese Weise werden alle möglichen Zuordnungen überprüft bis eine Übereinstimmung gefunden wird oder sämtliche Zuordnungen betrachtet wurden. Konnte keine Zuordnung gefunden werden bzw. keine Zuordnung, in der auch die restlichen Klauseln übereinstimmen, dann sind die beiden Anfragen nicht äquivalent im Sinne der Regelbedingung. Abbildung 5.3 zeigt ein Beispiel, in dem die Vergleiche zwischen den FROM-Klausel-Elementen der ersten und zweiten Anfrage und die rekursiven Funktionsaufrufe dargestellt sind. Die Einrückung entspricht der Rekursionstiefe. Die Suche nach einer passenden Zuordnung der FROM-Klausel-Elemente ist deshalb so wichtig, weil die Referenz auf eine mehrfach in dieser FROM-Klausel referenzierte Tabelle/Sicht nicht eindeutig einer Referenz in der FROM-Klausel der anderen Anfrage zugeordnet werden kann, wenn man sich nur auf die Betrachtung der FROM-Klauseln beschränkt. Es müssen daher alle Permutationen berücksichtigt werden und für jede muss überprüft werden, ob sie die Regelbedingung erfüllt. Erfüllt eine Zuordnung diese nicht, so sind entweder die beiden Anfragen im Sinne der Regelbedingung nicht äquivalent oder die Zuordnung ist nicht die richtige.

Codeausschnitt 2

```

29 private boolean matchQueries(Node query1, Node query2, Hashtable matchSet)
30 {
31     boolean match;
32
33     if (((Element) query1).getAttribute("distinct").equals("no") &&
34         ((Element) query2).getAttribute("distinct").equals("no") &&
35         ((Element) query1).getAttribute("result").equals("no") &&
36         ((Element) query2).getAttribute("result").equals("no") &&
37         includeSameClauses(query1, query2)) {
38         NodeList query1_NL = query1.getChildNodes();
39         NodeList query2_NL = query2.getChildNodes();
40         NodeList from1_NL = query1_NL.item(2).getChildNodes();
41         NodeList from2_NL = query2_NL.item(2).getChildNodes();
42         int l1 = from1_NL.getLength();
43         int l2 = from2_NL.getLength();
44
45         if (l1 == l2) {
46             boolean[] matched = new boolean[l1];
47             match = matchQueriesRec(query1_NL, query2_NL, from1_NL,
48                                   from2_NL, 0, matched, matchSet);
49         } else match = false;
50     }
51     else match = false;
52     return match;
53 }
54
55 private boolean matchQueriesRec(NodeList query1_NL, NodeList query2_NL,
56                                 NodeList from1_NL, NodeList from2_NL,
57                                 int i, boolean[] matched, Hashtable matchSet)
58 {
59     boolean match = false;
60     int l = from1_NL.getLength();
61
62     if (i < l) {
63         int j = 0;
64         while (match == false && j < l) {
65             String source1 = ((Element) from1_NL.item(i)).getAttribute("name");
66             String source2 = ((Element) from2_NL.item(j)).getAttribute("name");
67             String alias1 = ((Element) from1_NL.item(i)).getAttribute("alias");
68             String alias2 = ((Element) from2_NL.item(j)).getAttribute("alias");
69             if (matched[j] == false && source1.equals(source2)) {
70                 matched[j] = true;
71                 matchSet.put(alias1, alias2);
72                 match = matchQueriesRec(query1_NL, query2_NL, from1_NL,

```

```

72         from2_NL, i+1, matched, matchSet);
73     if (match == false) {
74         matched[j] = false;
75         matchSet.remove(alias1);
76     }
77 }
78 j++;
79 }
80 } else {
81     match = true;
82     for (int k = 0; k < query1_NL.getLength() && match; k++) {
83         if (query1_NL.item(k).getNodeName().equals("where-clause"))
84             match = matchWhereClauses(query1_NL.item(k), query2_NL.item(k),
85                                     from1_NL, from2_NL, matchSet);
86         if (query1_NL.item(k).getNodeName().equals("group-by-clause"))
87             match = matchGroupByClauses(query1_NL.item(k), query2_NL.item(k),
88                                       from1_NL, from2_NL, matchSet);
89         if (query1_NL.item(k).getNodeName().equals("having-clause"))
90             match = matchWhereClauses(query1_NL.item(k), query2_NL.item(k),
91                                     from1_NL, from2_NL, matchSet);
92         if (query1_NL.item(k).getNodeName().equals("order-by-clause"))
93             match = matchOrderByClauses(query1_NL.item(k), query2_NL.item(k),
94                                       from1_NL, from2_NL, matchSet);
95     }
96 }
97 return match;
98 }

```

Gegeben:

Anfrage1:

```

SELECT *
FROM   A A1, A A2, B
WHERE  A1.x = A2.x AND
        A2.x = B.x

```

Anfrage2:

```

SELECT *
FROM   B, A A2, A A1
WHERE  A1.x = A2.x AND
        A2.x = B.x

```

Ablauf des Algorithmus (Beschränkung auf Parameter i bei $matchQueriesRec$):

$matchQueriesRec(0)$:

↪ A1 - B *referenzieren untersch. Tabelle/Sicht*

↪ A1 - A2 → $matchQueriesRec(1)$:

 ↪ A2 - B *referenzieren untersch. Tabelle/Sicht*

 ↪ A2 - A2 $matched[j] = TRUE$

 ↪ A2 - A1 → $matchQueriesRec(2)$:

 ↪ B - B → $matchQueriesRec(3)$:

 ↪ Teste restliche Klauseln unter
 der Belegung (A1-A2, A2-A1, B-B)

 ↪ B - A2 *referenzieren untersch. Tabelle/Sicht*

 ↪ B - A1 *referenzieren untersch. Tabelle/Sicht*

↪ A1 - A1 → $matchQueriesRec(1)$:

 ↪ A2 - B *referenzieren untersch. Tabelle/Sicht*

 ↪ A2 - A2 → $matchQueriesRec(2)$:

 ↪ B - B → $matchQueriesRec(3)$:

 ↪ Teste restliche Klauseln unter
 der Belegung (A1-A1, A2-A2, B-B) ✓

Abbildung 5.3: Beispiel zur Methode $matchQueriesRec$.

In den Zeilen 33 bis 37 wird sichergestellt, dass die beiden zu vergleichenden Anfragen kein `DISTINCT` enthalten, dass sie keine Ergebnisanfragen sind und dass sie die gleiche Art von Klauseln enthalten. Nach Initialisierung der Variablen wird in Zeile 45 noch überprüft, ob die `FROM`-Klauseln der beiden Anfragen auch gleichviele Elemente enthalten, bevor dann in Zeile 47 die Rekursion gestartet wird. Im Feld *matched*, das während der Rekursion immer mitgeführt wird, wird für jedes Element der zweiten `FROM`-Klausel festgehalten, ob es schon zugeordnet wurde oder noch zur Verfügung steht. Im rekursiven Teil *matchQueriesRec* wird zuerst überprüft, ob bereits jedem Element der `FROM`-Klausel der ersten Anfrage ein entsprechendes Element in der `FROM`-Klausel der zweiten Anfrage zugeordnet werden konnte (Zeile 61). Falls nicht, dann wird für das *i*-te Element in der `FROM`-Klausel ein Element in der `FROM`-Klausel der zweiten Anfrage gesucht, das noch nicht zugeordnet wurde, aber dieselbe Tabelle/Sicht referenziert (Zeile 62 bis 79). Konnte ein solches Element gefunden werden, dann wird zur nächsten Referenz in der `FROM`-Klausel der ersten Anfrage übergegangen und damit die Rekursion fortgesetzt. Ist diese Zuordnung nicht erfolgreich, dann wird nach Rückkehr aus dem rekursiven Aufruf weitergesucht, ansonsten wird die Suche erfolgreich abgebrochen. Es ist noch zu erwähnen, dass *i* und *j* mit dem Wert '0' beginnen, d.h. die '0' entspricht dem ersten Element der `FROM`-Klausel, die '1' dem zweiten Element der `FROM`-Klausel, usw. Konnte jedem Element der `FROM`-Klausel der ersten Anfrage ein entsprechendes Element in der `FROM`-Klausel der zweiten Anfrage zugeordnet werden, dann hat man eine potentiell erfolgreiche Zuordnung gefunden. Es muss allerdings noch überprüft werden, ob die anderen Klauseln außer der `SELECT`-Klausel unter dieser Zuordnung übereinstimmen (Zeile 81 bis 97). Ist dies der Fall, dann hat man eine erfolgreiche Zuordnung gefunden.

Klauselschicht

Der Vergleich zweier Klauseln gestaltet sich für die verschiedenen Klauseltypen unterschiedlich. Am einfachsten ist der Vergleich zweier `ORDER-BY`-Klauseln, da alle Elemente Attributreferenzen sind und die Reihenfolge der Klausel-Elemente berücksichtigt werden muss. Es wird schrittweise jedes Element der ersten Klausel mit dem an der gleichen Position stehenden in der zweiten Klausel verglichen. Bei allen anderen Klauseltypen müssen sämtliche Permutationen betrachtet werden, da auf den Klausel-Elementen keine Ordnung definiert ist. D.h. für jedes Klausel-Element der ersten Klausel muss ein passendes Klausel-Element in der zweiten Klausel gesucht werden. Enthalten die beiden Klauseln gleichviele Elemente und konnte jedem Klausel-Element der ersten Klausel ein Klausel-Element der zweiten Klausel zugeordnet werden, dann sind die beiden Klauseln äquivalent, ansonsten sind die beiden Klauseln nicht äquivalent und der Algorithmus bricht vorzeitig ab. Bei der `GROUP-BY`-Klausel beschränken sich die zu vergleichenden Elemente auf Attributreferenzen. Bei der `WHERE`- und der `HAVING`-Klausel handelt es sich bei den Elementen um Prädikate, die zwei Terme über eine Vergleichsoperation verbinden; eventuell handelt es sich sogar um mehrere Vergleichsoperationen die über logische Operatoren verknüpft sind. In der in Abschnitt 2.1 definierten Untermenge von SQL92 kommt als logischer Operator innerhalb eines Prädikats nur ein logisches ODER (`OR`) in Frage. Bei logischen Operatoren muss wiederum berücksichtigt werden, dass auf der Operandenmenge keine Ordnung definiert ist und daher alle Permutationen betrachtet werden müssen.

Codeausschnitt 3

```

99 private boolean matchWhereClauses(Node where1, Node where2, NodeList from1_NL, NodeList from2_NL,
100                                 Hashtable matchSet)
101 {
102     boolean match;
103     NodeList where1_NL = where1.getChildNodes();
104     NodeList where2_NL = where2.getChildNodes();
105     int l1 = where1_NL.getLength();
106     int l2 = where2_NL.getLength();
107
108     if (l1 == l2) {
109         boolean[] matched = new boolean[l2];
110         match = true;
111         int i = 0;
112         while (i < l1 && match == true) {
113             match = false;
114             int j = 0;
115             while (j < l2 && match == false) {
116                 if (matched[j] == false &&
117                     matchPredicates(where1_NL.item(i), where2_NL.item(j),
118                                     from1_NL, from2_NL, matchSet)) {
119                     matched[j] = true;
120                     match = true;
121                 }
122                 j++;
123             }
124             i++;
125         }
126     } else match = false;
127     return match;
128 }

```

Codeausschnitt 3 zeigt als Beispiel für eine Methode der Klauselschicht die Methode *matchWhereClauses*, die die beiden übergebenen WHERE-Klauseln unter der Zuordnung *matchSet* vergleicht. Zur Vereinfachung des Algorithmus wird angenommen, dass keine Prädikatduplikate auftreten. D.h. wenn Duplikate auftreten, dann werden sie nicht als solche erkannt, sondern als eigenständige Prädikate behandelt. Deshalb brauchen Klauseln die sich schon in der Zahl der Elemente unterscheiden gar nicht näher betrachtet werden (siehe Zeile 108). In der äußeren Schleife (Zeile 112 bis 125) werden die einzelnen Elemente der ersten Klausel ausgewählt, in der inneren Schleife (Zeile 115 bis 123) wird nach einem passenden Element in der zweiten Klausel gesucht. Im Feld *matched* wird für jedes Element der zweiten Klausel festgehalten, ob es schon als Partner ausgewählt wurde oder noch zur Verfügung steht. Der Algorithmus kann entsprechend erweitert werden, sodass auch Duplikate berücksichtigt werden, was allerdings die zeitliche Komplexität erheblich erhöht.

Klauselelementschicht

Beim Vergleich von Vergleichsoperationen in Prädikaten muss die Eigenschaft der Symmetrie berücksichtigt werden, d.h. $a = b$ ist äquivalent zu $b = a$ bzw. $a \neq b$ ist äquivalent zu $b \neq a$. Ebenso sollte berücksichtigt werden, dass beispielsweise $a < b$ äquivalent ist zu $b > a$ und umgekehrt.

Der Vergleich von Termen wird im Prototyp relativ simpel gehandhabt; die beiden Bäume, die die Terme repräsentieren, werden parallel traversiert und dabei unter der gegebenen Zuordnung der FROM-Klausel-Elemente verglichen; d.h. aber auch, dass die Abfolge der Operationen in beiden Termen gleich sein muss.

Aktionsteil der Regel

Da alle Anfragemengen paarweise disjunkt sind, werden im Prototyp sämtliche Anfragemengen berechnet und für jede Anfragemenge wird im Aktionsteil der Regel die Vereinigung der Anfragen durchgeführt. Die einzelnen Anfragen einer Anfragemenge werden dabei schrittweise mit der ersten Anfrage der Anfragemenge vereinigt.

Im Prototyp wurde für die Ermittlung des Primärschlüssels und des Anfragetyps die jeweils einfachste Lösung gewählt. Es wird der Primärschlüssel aus der ersten Anfrage einer Anfragemenge, d.h. aus der Anfrage, mit der die anderen Anfragen der Anfragemenge schrittweise verschmolzen werden, in die vereinigte Anfrage übernommen. Werden bei der Vereinigung der Attributdefinitionen bzw. SELECT-Klauseln Duplikate gefunden, die nicht Teil des Primärschlüssels der ersten Anfrage sind, dann gilt: *UNIQUE* > *NOT NULL* > *kein Constraint* > *PRIMARY KEY*. Ist eine der Anfragen einer Anfragemenge vom Typ ‘Tabelle’, dann wird auch der Typ der vereinigten Anfrage zu dieser Anfragemenge auf ‘Tabelle’ gesetzt, ansonsten auf ‘Sicht’.

5.2.2 Where To Group

Die Algorithmen der Regel *Where To Group* sind denen der Regel *Merge Select* sehr ähnlich. Es wird ebenso nach einer Anfragemengen gesucht. Der Unterschied ist, dass hier nicht nach Anfragen gesucht wird, die sich in einer bestimmten Klausel unterscheiden und sonst übereinstimmen, sondern nach Anfragen, die sich in einem beliebigen Prädikat auf gewisse Art und Weise unterscheiden und sonst übereinstimmen. Dies bringt aber auch neue Probleme mit sich, wie sich anhand der folgenden drei Anfragen zeigen lässt:

<pre>SELECT x, SUM(y) FROM F WHERE a=1 AND b=1 GROUP BY x</pre>	<pre>SELECT x, SUM(y) FROM F WHERE a=1 AND b=2 GROUP BY x</pre>	<pre>SELECT x, SUM(y) FROM F WHERE a=2 AND b=1 GROUP BY x</pre>
--	--	--

Die linke Anfrage könnte zusammen mit der mittleren eine Anfragemenge bilden, wobei sie sich im zweiten Prädikat unterscheiden. Die linke Anfrage könnte aber ebenso mit der rechten eine Anfragemenge bilden, wobei sie sich im ersten Prädikat unterscheiden. D.h. die linke Anfrage ist Teil zweier unterschiedlicher Anfragemengen. Der Grund hierfür ist, dass in der Regelbedingung nicht vorhergesagt werden kann, in welchem Prädikat sich die Anfragen unterscheiden, sondern, dass jedes Prädikat in Frage kommt. Die Anfragegruppen werden daher nicht nur durch die Übereinstimmungen charakterisiert, sondern auch durch das Prädikat in dem sich die Anfragen unterscheiden, wobei dieses Prädikat durch die Attributreferenz, die mit der Konstante verglichen wird, charakterisiert ist. Es muss also in der Klauselschicht festgestellt werden, ob es in der ersten Klausel ein Prädikat gibt, für das in der zweiten Klausel kein passendes Prädikat gefunden werden kann und umgekehrt. Entsprechen diese beiden Prädikate den Vorgaben der Regelbedingung, dann müssen diese charakteristischen Prädikate nach oben propagiert werden. In der obersten Schicht werden die charakteristischen Prädikate dann zur Einordnung in die richtige Anfragemenge verwendet. Die Vereinfachung, dass eine Anfrage, die schon Teil einer Anfragemenge ist, nicht mehr zu Vergleichen herangezogen werden muss, kann hier nicht angewandt werden. Nimmt man beispielsweise an, dass Anfrage Q1 mit Anfrage Q2 eine Anfragemenge bildet, die durch das Attribut ‘a’ charakterisiert ist, und Q3 nicht Teil einer Anfragemenge mit Q1 ist, dann kann Anfrage Q2 trotzdem mit Q3

eine Anfragemenge bilden, die durch ein anderes Attribut als “a” charakterisiert ist. Es kann aber untersucht werden, ob die beiden zu vergleichenden Anfragen schon Teil der gleichen Anfragegruppe sind. Ist dies der Fall, müssen sie nicht erneut überprüft werden.

Die Regel *Where To Group* lässt sich wie die Regel *Merge Select* in mehrere Schichten zerlegen. Das Schichtenmodell mit den zugehörigen Methoden ist in Abbildung 5.4 dargestellt.

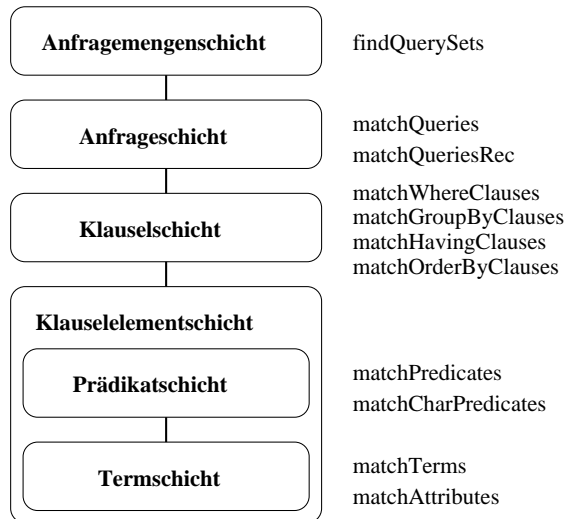


Abbildung 5.4: Schichtenmodell für die Regel *Where To Group* mit den zugehörigen Methoden.

Anfragemengenschicht

Codeausschnitt 4 zeigt die Methode *findQuerySets*. Zusätzlich zu den Objektvariablen *querySets* und *matchSets* enthält das Regelobjekt noch die Variable *charPredSets*. *charPredSets* ist eine Liste, deren Elemente Listen mit charakteristischen Prädikaten, die zu den jeweiligen Anfragen der Anfragemenge gehören, sind.

Codeausschnitt 4

```

1  private boolean findQuerySets()
2  {
3      NodeList sequence_NL = sequence.getChildNodes();
4      int l = sequence_NL.getLength();
5
6      for (int i = 0; i < l-1; i++)
7          for (int j = i+1; j < l; j++) {
8              Hashtable matchSet = new Hashtable();
9              NodePointer charPred1 = new NodePointer();
10             NodePointer charPred2 = new NodePointer();
11             int k = 0;
12             while (k < querySets.size() &&
13                 (!((Vector) querySets.elementAt(k)).
14                 contains(sequence_NL.item(i)) &&
15                 ((Vector) querySets.elementAt(k)).
16                 contains(sequence_NL.item(j))))
17                 k++;
18             if (k == querySets.size() &&
19                 matchQueries(sequence_NL.item(i), sequence_NL.item(j),

```

```

20         matchSet, charPred1, charPred2)) {
21         int m = 0;
22         while (m < charPredSets.size() &&
23             ((Node) ((Vector) charPredSets.elementAt(m)).
24             firstElement()) != charPred1.get())
25             m++;
26         if (m == charPredSets.size()) {
27             querySets.addElement(new Vector());
28             ((Vector) querySets.lastElement()).addElement(sequence_ML.item(i));
29             matchSets.addElement(new Vector());
30             charPredSets.addElement(new Vector());
31             ((Vector) charPredSets.lastElement()).addElement(charPred1.get());
32         }
33         ((Vector) querySets.elementAt(m)).addElement(sequence_ML.item(j));
34         ((Vector) matchSets.elementAt(m)).addElement(matchSet);
35         ((Vector) charPredSets.elementAt(m)).addElement(charPred2.get());
36     }
37 }
38 return (!querySets.isEmpty());
39 }

```

Zu Beginn ist *charPredSets* ebenfalls leer. In den Zeilen 11 bis 18 wird ermittelt, ob beide Anfragen des Anfragepaares bereits Teil derselben Anfragemenge sind. Ist dies der Fall, dann muss dieses Anfragepaar nicht mehr weiter untersucht werden. Die Methode *matchQueries* liefert bei dieser Regel im Vergleich zur Regel *Merge Select* zusätzlich noch die beiden charakteristischen Prädikate *charPred1* und *charPred2*, falls die beiden Anfragen die Regelbedingung erfüllen (Zeile 19 und 20). In den Zeilen 21 bis 26 wird ermittelt, ob die *i*-te Anfrage bereits Element einer Anfragemenge ist, die durch das Attribut des Prädikats *charPred1* charakterisiert wird. Dies lässt sich sehr einfach dadurch realisieren, dass man überprüft, ob das charakteristische Prädikat *charPred1* schon Teil einer Anfragemenge ist. Es wird dabei nicht überprüft, ob es sich um ein vergleichbares Prädikat handelt, sondern, ob es sich um dieselbe Instanz, d.h. denselben Prädikatsknoten, handelt. Ist die *i*-te Anfrage noch nicht Element einer Anfragemenge, dann werden die entsprechenden Listen angelegt, mit den entsprechenden Elementen initialisiert und *querySets*, *matchSets* und *charPredSets* hinzugefügt (Zeile 27 bis 31). Unabhängig davon werden auch die entsprechenden Elemente für die *j*-te Anfrage den entsprechenden Listen in *querySets*, *matchSets* und *charPredSets* hinzugefügt (Zeile 33 bis 35).

Anfrageschicht

Die Methoden *matchQueries* und *matchQueriesRec* unterscheiden sich von den gleichnamigen Methoden der Regel *Merge Select* nur darin, dass sie die charakteristischen Prädikate in den Parametern zurückliefern und dass in *matchQueriesRec* die charakteristischen Prädikate und der Typ der Anfragekombination von der Methode *matchWhereClauses* an die Methode *matchGroupByClauses* weitergereicht werden. Daher sind hier keine Codeausschnitte mit diesen Methode abgedruckt.

Klauselschicht

Codeausschnitt 5 zeigt die Methode *matchWhereClauses*. Die verschachtelten Schleifen in Zeile 55 bis 73 stimmen in etwa mit den verschachtelten Schleifen der gleichnamigen Methode in *Merge Select* überein. Einziger Unterschied ist, dass in der Regel *Where To Group* für maximal ein Prädikat in der WHERE-Klausel *where1* erlaubt ist, dass kein passendes Prädikat

in der WHERE-Klausel *where2* existiert (Zeile 68 bis 71). Dieses Prädikat *pred1* ist ein Kandidat für das charakteristische Prädikat der WHERE-Klausel *where1*. Wurde ein solches Prädikat gefunden (Zeile 74), dann wird auch in der WHERE-Klausel *where2* nach dem Prädikat gesucht, das keinem Prädikat in *where1* zugewiesen werden konnte (Zeile 75 und 76). Dieses Prädikat *pred2* ist dann ein Kandidat für das charakteristische Prädikat der WHERE-Klausel *where2*. Anschließend wird die Methode *matchCharPredicates* aufgerufen, um zu überprüfen, ob es sich bei den beiden Kandidaten tatsächlich um charakteristische Prädikate handelt und, falls dies der Fall ist, welcher Typ von Anfragekombination vorliegt (Zeile 77 und 78).

Codeausschnitt 5

```

40 private boolean matchWhereClauses(Node where1, Node where2, NodeList from1_NL, NodeList from2_NL,
41                                 Hashtable matchSet, IntPointer type,
42                                 NodePointer charPred1, NodePointer charPred2)
43 {
44     boolean match;
45     NodeList where1_NL = where1.getChildNodes();
46     NodeList where2_NL = where2.getChildNodes();
47     int l1 = where1_NL.getLength();
48     int l2 = where2_NL.getLength();
49
50     if (l1 == l2) {
51         boolean[] matched = new boolean[l2];
52         match = true;
53         Node pred1 = null;
54         Node pred2 = null;
55         int i = 0;
56         while (i < l1 && match == true) {
57             match = false;
58             int j = 0;
59             while (j < l2 && match == false) {
60                 if (matched[j] == false &&
61                     matchPredicates(where1_NL.item(i), where2_NL.item(j),
62                                     from1_NL, from2_NL, matchSet)) {
63                     matched[j] = true;
64                     match = true;
65                 }
66                 j++;
67             }
68             if (match == false && pred1 == null) {
69                 pred1 = where1_NL.item(i);
70                 match = true;
71             }
72             i++;
73         }
74         if (match == true && pred1 != null) {
75             for (int j = 0; pred2 == null; j++)
76                 if (matched[j] == false) pred2 = where2_NL.item(j);
77             match = matchCharPredicates(pred1, pred2, from1_NL, from2_NL,
78                                         matchSet, type, charPred1, charPred2);
79         }
80     } else match = false;
81     return match;
82 }

```

Klauselelementschicht

Die Vorgehensweise beim Vergleich von Prädikaten und Termen ist äquivalent zur Vorgehensweise bei der Regel *Merge Select*, d.h. die entsprechenden Methoden sind aus der Regel *Merge*

Select übernommen.

Zusätzlich ist noch die Methode *matchCharPredicates* hinzugekommen, die überprüft, ob es sich bei den beiden Kandidaten tatsächlich um charakteristische Prädikate handelt und, falls dies der Fall ist, welcher Typ von Anfragekombination vorliegt. Bei Typ 1 sind beide Prädikate Äquivalenzvergleiche eines Attributs mit einer Konstanten, wobei das Attribut bei beiden Prädikaten das gleiche ist. Bei Typ 2 sind beide Prädikate Enthaltenseinstests, wobei das Attribut bei beiden Prädikaten das gleiche ist. Bei Typ 3 ist ein Prädikat ein Äquivalenzvergleich eines Attributs mit einer Konstanten und das andere ein Enthaltenseinstest, wobei das Attribut bei beiden Prädikaten das gleiche ist. Der Typ von Anfragekombination muss auch beim Vergleich der GROUP-BY-Klauseln in der Methode *matchGroupByClauses* berücksichtigt werden.

Aktionsteil der Regel

Im Prototyp werden sämtliche Anfragemengen berechnet. Im Aktionsteil wird jedoch nur für die erste Anfragemenge die Vereinigung der Anfragen durchgeführt, da die gefundenen Anfragemengen nicht disjunkt sein müssen. Da aber alle Anfragepaare ermittelt werden, kann der Prototyp dahingehend erweitert werden, dass er für die Vereinigung eine Menge disjunkter Anfragemengen auswählt. Es wäre auch möglich nach der Vereinigung der Anfragen einer Anfragemenge, diese Anfragen aus allen anderen Anfragemengen zu entfernen und so mit allen Anfragemengen zu verfahren. Bei der Vereinigung werden die einzelnen Anfragen einer Anfragemenge schrittweise mit der ersten Anfrage der Anfragemenge vereinigt.

Im Prototyp wurde für die Ermittlung des Primärschlüssels und des Anfragetyps dieselbe Lösung gewählt, wie bei *Merge Select*. Zusätzlich wird auch für das charakteristische Attribut die Primärschlüsseleigenschaft gesetzt.

5.2.3 Predicate Pushdown

Codeausschnitt 6 zeigt die Methoden *findPushdownPredicatesForQuery* und *findPushdownPredicatesForQueryRec*, die für eine gegebene Anfrage rekursiv die Pushdown-Prädikate, die in diese Anfrage verschoben werden können, ermitteln. Der Algorithmus arbeitet wie folgt: Zuerst wird ermittelt, welche Anfragen die gegebene Anfrage referenzieren. Anschließend wird festgestellt, welche Prädikate aus der WHERE-Klausel der ersten referenzierenden Anfrage überhaupt als Pushdown-Prädikate in Frage kommen, d.h. es werden die Prädikate ausgewählt, die jeweils nur Attribute einer einzigen Referenz enthalten, wobei diese die gegebene Anfrage referenziert. Nun wird nacheinander für jede der referenzierenden Anfragen überprüft, ob für jede der dort enthaltenen Referenzen auf die gegebene Anfrage ein solches Prädikat existiert. Ist dies bei einer der referenzierenden Anfragen nicht der Fall, dann kann dieses Prädikat nicht verschoben werden und die Überprüfung der restlichen referenzierenden Anfragen bezüglich dieses Prädikats ist nicht mehr nötig. Konnte jedoch für jede Referenz auf die gegebene Anfrage ein entsprechendes Prädikat gefunden werden, dann können diese Prädikate und eventuell vorhandene Duplikate aus der Anfrage entfernt und zu der gegebenen Anfrage hinzugefügt werden.

Codeausschnitt 6

```

1 private void findPushdownPredicatesForQuery(Node query, Vector delList, Vector addList)
2 {
3     Vector queries = new Vector();

```

```

4     Vector wheres = new Vector();
5     Vector refs   = new Vector();
6
7     String queryName = ((Element) query).getAttribute("name");
8     for (Node refQuery = sequence.getFirstChild();
9         refQuery != null; refQuery = refQuery.getNextSibling())
10    for (Node ref = query.getFirstChild().getFirstChild();
11        ref != null; ref = ref.getNextSibling())
12        if (((Element) refQuery).getAttribute("name").
13            equals(((Element) ref).getAttribute("name"))) {
14            queries.addElement(refQuery);
15            Node where = refQuery.getFirstChild().getNextSibling().
16                getNextSibling().getNextSibling();
17            if (where == null || !where.getNodeName().equals("where-clause"))
18                return;
19            wheres.addElement(where);
20            refs.addElement(new Vector());
21            Node from = refQuery.getFirstChild().getNextSibling().getNextSibling();
22            for (Node source = from.getFirstChild();
23                source != null; source = source.getNextSibling())
24                if (((Element) source).getAttribute("name").equals(queryName))
25                    ((Vector) refs.lastElement()).
26                        addElement(((Element) source).
27                            getAttribute("alias"));
28        }
29
30    if (!queries.isEmpty()) {
31        Node where = (Node) wheres.firstElement();
32        for (Node pred = where.getFirstChild();
33            pred != null; pred = pred.getNextSibling()) {
34            Vector ref = (Vector) refs.firstElement();
35            StringPointer aliasName = new StringPointer();
36            if (refsOneQuery(pred, aliasName) && !aliasName.isEmpty()) {
37                int i;
38                for (i = 0; i < ref.size() && !((String) ref.elementAt(i)).
39                    equals(aliasName.get()); i++);
40                if (i < ref.size())
41                    if (findPushdownPredicatesForQueryRec(queries, wheres, refs,
42                                                            0, pred, delList)) {
43                        Node newElement[] = new Node[2];
44                        newElement[0] = pred;
45                        newElement[1] = query;
46                        addList.addElement(newElement);
47                    }
48            }
49        }
50    }
51 }

52 private boolean findPushdownPredicatesForQueryRec(Vector queries,
53 Vector wheres, Vector refs, int i, Node pred, Vector delList)
54 {
55     if (i == queries.size())
56         return true;
57     else {
58         Node query = (Node) queries.elementAt(i);
59         Vector ref = (Vector) refs.elementAt(i);
60         Node where = (Node) wheres.elementAt(i);
61         Vector delListTmp = new Vector();
62         boolean found[] = new boolean[ref.size()];
63         int notFound = ref.size();
64
65         for (Node predX = where.getFirstChild(); predX != null;
66             predX = predX.getNextSibling()) {
67             StringPointer aliasName = new StringPointer();
68             if (refsOneQuery(predX, aliasName) && !aliasName.isEmpty()) {

```

```

69         int j;
70         for (j = 0; j < ref.size() && !((String) ref.elementAt(j)).
71             equals(aliasName.get()); j++);
72         if (j < ref.size() &&
73             matchPredicates(pred, predX, false)) {
74             if (!found[j]) { found[j] = true; notFound--; }
75             Node newElement[] = new Node[3];
76             newElement[0] = predX;
77             newElement[1] = where;
78             newElement[2] = query;
79             delListTmp.addElement(newElement);
80         }
81     }
82 }
83
84     if (notFound == 0)
85         if (findPushdownPredicatesForQueryRec(queries, wheres, refs,
86             i+1, pred, delList)) {
87             for (int j = 0; j < delListTmp.size(); j++)
88                 delList.addElement(delListTmp.elementAt(j));
89             return true;
90         } else return false;
91     else return false;
92 }
93 }

```

Die zu löschenden Prädikate der referenzierenden Anfragen werden einzeln in die Liste *delList* aufgenommen. Entsprechend existiert auch eine Liste *addList*, die die Prädikate enthält, die der referenzierten Anfrage hinzugefügt werden müssen. In den Zeilen 8 bis 28 werden die Anfragen *queries* ermittelt, die die Anfrage *query* referenzieren, deren WHERE-Klauseln *wheres* und deren Referenzen *refs* auf die gegebene Anfrage. Die genesteten Schleifen mit der anschließenden if-Anweisung dienen dazu, die zu den Bezeichnern in der REFERENCED-BY-Klausel gehörenden Anfragen zu ermitteln. Enthält mindestens eine dieser Anfragen keine WHERE-Klausel, dann existieren auch keine Pushdown-Prädikate und es kann erfolglos abgebrochen werden (Zeile 17 und 18). In der Schleife von Zeile 22 bis 27 werden die Alias-Bezeichner der FROM-Klausel-Elemente, die die gegebene Anfrage referenzieren, ermittelt und gemeinsam als Liste an die Liste *refs* angefügt. In den Zeilen 30 bis 50 werden dann die einzelnen Prädikate der ersten referenzierenden Anfrage betrachtet und die Rekursion angestoßen, falls diese als Pushdown-Prädikate in Frage kommen. Zuerst wird jedoch geprüft, ob die gegebene Anfrage überhaupt von einer anderen Anfrage referenziert wird (Zeile 30). In Zeile 36 wird nun für jedes Prädikat überprüft, ob jeweils alle Attribute in diesem Prädikat zu derselben Referenz gehören; die Methode *refsOneQuery* liefert in *aliasName* zusätzlich den Alias-Bezeichner dieser Referenz. Anschließend wird ermittelt, ob dies eine Referenz auf die gegebene Anfrage ist (Zeile 37 bis 40). Ist dies der Fall, dann wird die Rekursion gestartet. Verläuft diese erfolgreich, d.h. es handelt sich bei dem betrachteten Prädikat um ein Pushdown-Prädikat, dann wird es zusammen mit der gegebenen Anfrage in die Liste *addList* eingetragen. Im rekursiven Teil *findPushdownPredicatesForQueryRec* wird zuerst überprüft, ob bereits alle referenzierenden Anfragen betrachtet wurden. Ist dies der Fall, wird die Rekursion erfolgreich abgebrochen werden, ansonsten wird die *i*-te Anfrage genauer betrachtet. Für jedes Prädikat dieser Anfrage wird geprüft, ob es nur Attribute derselben Referenz enthält, ob diese Referenz in der zur Anfrage gehörenden Liste in *refs* enthalten ist und ob dieses Prädikat mit dem gesuchten Prädikat *pred* übereinstimmt (Zeile 68 bis 73). Ist dies der Fall, dann wird für diese Referenz vermerkt, dass ein entsprechendes Prädikat gefunden wurde (Zeile 74) und das Prädikat zusammen mit der zugehörigen

WHERE-Klausel und Anfrage in die temporäre Löschliste *delListTmp* eingetragen. Hierbei werden auch Duplikate erfasst. Konnte für jede Referenz in der zugehörigen Liste in *refs* ein solches Prädikat gefunden werden (Zeile 84), dann wird die rekursive Methode für die nächste referenzierende Anfrage aufgerufen (Zeile 85 und 86). Liefert dieser rekursive Aufruf ein positives Ergebnis, dann handelt es sich tatsächlich um ein Pushdown-Prädikat und die Elemente aus der temporären Löschliste *delListTmp* werden in die Löschliste *delList* übernommen (Zeile 87 und 88).

5.2.4 Concat Queries

Im Bedingungsteil der Regel *Concat Queries* wird eigentlich nur für jede Anfrage der Sequenz überprüft, ob einer der vier möglichen Fälle vorliegt. Interessant in diesem Zusammenhang ist die Methode *selectClauseContainsGroupByAttributes* in Codeausschnitt 7, die prüft, ob die SELECT-Klausel der referenzierenden Anfrage alle Attribute aus der GROUP-BY-Klausel der referenzierten Anfrage enthält. In Zeile 12 bis 25 wird ermittelt, welche Attributdefinitionen der referenzierten Anfrage die Gruppierungsattribute der referenzierten Anfrage enthalten und deren Bezeichner in einer Hashtabelle gespeichert. In der äußeren Schleife (Zeile 13 bis 25) wird die GROUP-BY-Klausel der referenzierten Anfrage durchlaufen. In der inneren Schleife (Zeile 17 bis 24) wird zu jedem Gruppierungsattribut die entsprechende Attributdefinition in der referenzierten Anfrage gesucht und deren Bezeichner in die Hashtabelle *groupByAttrs* eingetragen (Zeile 23). In Zeile 27 bis 38 werden die Attributdefinitionen der referenzierenden Anfrage durchlaufen und für jede Attributdefinition, die nur aus einem Attribut dessen Bezeichner in *groupByAttrs* enthalten ist, besteht, wird dieser Bezeichner aus *groupByAttrs* entfernt. Ist am Schluß *groupByAttrs* leer, dann enthält die SELECT-Klausel der referenzierenden Anfrage alle Attribute aus der GROUP BY-Klausel der referenzierten Anfrage, ansonsten nicht.

Codeausschnitt 7

```

1  private boolean selectClauseContainsGroupByAttributes(Node refQuery, Node query)
2  {
3      String sourceName = ((Element) query).getAttribute("name");
4      Node select      = query.getFirstChild().getNextSibling();
5      Node from        = select.getNextSibling();
6      Node groupBy;
7      if (from.getNextSibling().getNodeName().equals("group-by-clause"))
8          groupBy = from.getNextSibling();
9      else
10         groupBy = from.getNextSibling().getNextSibling();
11
12     Hashtable groupByAttrs = new Hashtable();
13     for (Node groupAttr = groupBy.getFirstChild(); groupAttr != null;
14         groupAttr = groupAttr.getNextSibling()) {
15         String groupAttrName = ((Element) groupAttr).getAttribute("name");
16         String groupAttrSource = ((Element) groupAttr).getAttribute("source");
17         for (Node attrDef = select.getFirstChild(); attrDef != null;
18             attrDef = attrDef.getNextSibling()) {
19             Element attr = (Element) attrDef.getFirstChild();
20             if (attr.getNodeName().equals("attribute") &&
21                 attr.getAttribute("name").equals(groupAttrName) &&
22                 attr.getAttribute("source").equals(groupAttrSource))
23                 groupByAttrs.put(((Element) attrDef).getAttribute("name"), "");
24         }
25     }
26
27     Node source = getSourceInFromClause(sourceName, refQuery.getFirstChild());

```

```

28         getNextSibling().getNextSibling());
29     String aliasName = ((Element) source).getAttribute("alias");
30     for (Node attrDef = refQuery.getFirstChild().getNextSibling();
31         attrDef != null && !groupByAttrs.isEmpty();
32         attrDef = attrDef.getNextSibling()) {
33         Element attr = (Element) attrDef.getFirstChild();
34         if (attr.getNodeName().equals("attribute") &&
35             attr.getAttribute("source").equals(aliasName) &&
36             groupByAttrs.contains(attr.getAttribute("name")))
37             groupByAttrs.remove(attr.getAttribute("name"));
38     }
39     if (groupByAttrs.isEmpty()) return true; else return false;
40 }

```

Im Prototyp werden im Bedingunsteil alle Anfragepaare ermittelt, die konkateniert werden können. Im Aktionsteil wird aber nur das erste Anfragepaar aus dieser Liste konkateniert, da diese Konkatenierung die anderen Anfragepaare beeinflussen kann. D.h. dass eventuell die referenzierende Anfrage eines Anfragepaares gar nicht mehr existiert, da sie zuvor schon mit einer anderen Anfrage vereinigt wurde. Da aber alle Anfragepaare ermittelt werden, kann der Prototyp dahingehend erweitert werden, dass er für die Konkatenierung eine Menge disjunkter Anfragepaare auswählt. Bei der Konkatenierung wird der Primärschlüssel der referenzierenden Anfrage beibehalten. Ist eine der beiden Anfragen vom Typ ‘Tabelle’, dann wird auch der Typ der referenzierenden Anfrage auf ‘Tabelle’ gesetzt.

5.2.5 Eliminate Redundant References

Die Problematik der Regel *Eliminate Redundant References* ähnelt der Berechnung der zusammenhängenden Komponenten eines Graphen bzw. der Berechnung der transitiven Hülle. Die Kanten sind in diesem Fall Joins über Primärschlüssel; die Komponenten bzw. Knoten sind die Elemente der FROM-Klausel, d.h. Referenzen auf Tabellen/Sichten. Sind mehrere Referenzen auf dieselbe Tabelle/Sicht auf diese Weise miteinander verbunden, so können alle diese Referenzen bis auf eine entfernt werden. Codeausschnitt 8 zeigt die Methode *findRedundantReferencesInQuery*, die in der Regelbedingung für jede Anfrage der Abfragesequenz aufgerufen wird. Der Algorithmus arbeitet wie folgt: Für jede Referenz in der FROM-Klausel einer Anfrage wird eine einelementige Menge erzeugt, die genau diese Referenz enthält. Nun wird jedes Prädikat der Anfrage darauf überprüft, ob es ein Joinprädikat ist, das zwei Referenzen über den Primärschlüssel oder einen Teil des Primärschlüssels miteinander verbindet. Bei der Implementierung des Prototyps habe ich nur Joins über einfache Primärschlüssel berücksichtigt. Will man auch Joins über zusammengesetzte Primärschlüssel berücksichtigen, dann muss überprüft werden, ob auch die restlichen Komponenten des Primärschlüssels über entsprechende Prädikate miteinander verbunden sind. Ist dies der Fall, dann kann die Menge, die die eine Referenz enthält, mit der Menge, die die andere Referenz enthält, vereinigt werden. Als Ergebnis erhält man eine Menge von Mengen, wobei die Referenzen in diesen Mengen direkt oder transitiv miteinander verbunden sind. Diese Mengen müssen nun so aufgesplittet werden, dass sie nur noch Referenzen auf dieselbe Tabelle/Sicht enthalten. Alle einelementigen Mengen können entfernt werden. Übrig bleibt eine Menge von Mengen, die jeweils eine Referenz und alle dazu redundanten Referenzen enthalten. Im Aktionsteil der Regel kann dann aus jeder Menge eine Referenz ausgewählt werden, die die restlichen Referenzen der Menge ersetzt. Abbildung 5.5 zeigt ein Beispiel für den Algorithmus des Bedingunsteils.

Codeausschnitt 8

```

1  private void findRedundantReferencesInQuery(Node query)
2  {
3      Node from = query.getFirstChild().getNextSibling().getNextSibling();
4      Node where = from.getNextSibling();
5
6      if (where != null && where.getNodeName().equals("where-clause")) {
7          Hashtable aliasToName = new Hashtable();
8          for (Node source = from.getFirstChild(); source != null;
9              source = source.getNextSibling())
10             aliasToName.put(((Element) source).getAttribute("alias"),
11                             ((Element) source).getAttribute("name"));
12
13         Vector refSets = new Vector();
14         for (Node source = from.getFirstChild(); source != null;
15             source = source.getNextSibling()) {
16             Vector refSet = new Vector();
17             refSet.addElement(source);
18             refSets.addElement(refSet);
19         }
20
21         for (Node pred = where.getFirstChild(); pred != null;
22             pred = pred.getNextSibling())
23             if (pred.getNodeName().equals("eq")) {
24                 Node left = pred.getFirstChild();
25                 Node right = left.getNextSibling();
26                 if (left.getNodeName().equals("attribute") &&
27                     right.getNodeName().equals("attribute") &&
28                     !((Element) left).getAttribute("source").
29                     equals(((Element) right).getAttribute("source"))) &&
30                     primaryKeys(left, right, aliasToName))
31                     union(left, right, refSets);
32             }
33
34         for (int i = 0; i < refSets.size(); i++) {
35             Vector refSet = (Vector) refSets.elementAt(i);
36             while (refSet.size() > 1) {
37                 Vector newRefSet = new Vector();
38                 String sourceName = ((Element) refSet.firstElement()).
39                     getAttribute("name");
40                 for (int j = 0; j < refSet.size(); j++) {
41                     Node source = (Node) refSet.elementAt(j);
42                     if (((Element) source).getAttribute("name").
43                         equals(sourceName)) {
44                         newRefSet.addElement(source);
45                         refSet.removeElementAt(j);
46                         j--;
47                     }
48                 }
49                 if (newRefSet.size() > 1) {
50                     Object newElement[] = new Object[2];
51                     newElement[0] = newRefSet;
52                     newElement[1] = query;
53                     redRefList.addElement(newElement);
54                 }
55             }
56         }
57     }
58 }

```

Im Prototyp werden die Mengen, die eine Referenz und alle dazu redundanten Referenzen enthalten, zusammen mit der Anfrage in der diese Referenzen enthalten sind in der Liste *redRefSets* gespeichert. Zu Beginn der Methode *findRedundantReferencesInQuery* wird über-

<u>Gegeben:</u>	
SELECT	*
FROM	A A1, A A2, B, C
WHERE	A1.x = B.x AND B.x = A2.x AND C.y = 5
A.x und B.x sind einfache Primärschlüssel	
<u>Ablauf des Algorithmus:</u>	
Startmenge: {A1}, {A2}, {B}, {C}	
A1.x = B.x	⇒ Vereinigung von {A1} und {B}
⇒ {A1, B}, {A2}, {C}	
B.x = A2.x	⇒ Vereinigung von {A1, B} und {A2}
⇒ {A1, B, A2}, {C}	
C.y = 5	⇒ Keine Vereinigung
⇒ {A1, B, A2}, {C}	
{A1, B, A2} aufsplitten in {A1, A2} und {B}	
⇒ {A1, A2}, {B}, {C}	
Einelementige Mengen entfernen	
⇒ {A1, A2}	

Abbildung 5.5: Beispiel zur Methode *matchQueriesRec*.

prüft, ob die betrachtete Anfrage eine WHERE-Klausel enthält. Ist dies nicht der Fall, dann existieren auch keine redundanten Referenzen, ansonsten kann nach den redundanten Referenzen gesucht werden. Zuerst wird eine Hashtabelle erzeugt, die dem Alias-Bezeichner jedes FROM-Klausel-Elements den zugehörigen Bezeichner dieses Elements zuordnet. Diese Hash-tabelle wird von der Methode *primaryKeys* benötigt (Zeile 30). Der Vektor *refSets*, der die Menge von Mengen repräsentiert wird in Zeile 13 bis 19 mit den einelementigen Mengen initialisiert. In der Schleife von Zeile 21 bis 32 werden die Prädikate der WHERE-Klausel einzeln betrachtet. Für jedes Prädikat wird überprüft, ob es ein Äquivalenzvergleich ist (Zeile 23), ob linke und rechte Seite des Vergleichs Attribute sind (Zeile 26 und 27), ob diese Attribute zu verschiedenen Referenzen gehören (Zeile 28 und 29) und ob es sich bei beiden Attributen um einfache Primärschlüssel handelt (Zeile 30). Trifft all' dies zu, dann können die zugehörigen Mengen vereinigt werden (Zeile 31). In den Zeilen 34 bis 56 werden die gefundenen Mengen aufgesplittet. Dazu wird ermittelt, welche Tabelle/Sicht das erste Element einer solchen Menge referenziert (Zeile 38 und 39). Anschließend werden alle Elemente, die diese Tabelle/Sicht referenzieren, aus der Menge entfernt und zu einer neuen Menge *newRefSet* zusammengefügt (Zeile 40 bis 48). Dies wird durch zwei genestete Schleifen realisiert. Enthält *newRefSet* mehr als nur ein Element, dann wird es zusammen mit der Anfrage an die Liste *redRefList* angefügt. Nun wird ermittelt welche Tabelle/Sicht das nächste Element der Menge referenziert, usw. Da beim ersten Durchlaufen der inneren for-Schleife das erste Element der Menge entfernt wurde, ist das nächste Element wiederum das erste Element der verbleibenden Menge (Zeile 38). Dieser Vorgang wird solange fortgesetzt bis die Menge keine Elemente mehr oder nur noch ein Element enthält. Nach diesem Verfahren werden alle Mengen aufgesplittet.

5.3 Das Regelsystem und mögliche Erweiterungen

Im Regelsystem des Prototyps wird jede Regelklasse einmalig instanziiert. D.h. ein Objekt, das eine Regel repräsentiert, kann beliebig oft auf eine Anfragesequenz angewendet werden. Als Kontrollstrategie wurde ein simples Prioritätensystem implementiert, da in der Aufgabenstellung diesbezüglich keine weiteren Anforderungen spezifiziert wurden. Die Regelobjekte sind in einem Feld abgelegt, wobei die Position innerhalb des Feldes die Priorität der Regel bestimmt; je niedriger der Positionsindex, desto höher die Priorität. D.h. das erste Element des Feldes hat die höchste Priorität, das letzte die niedrigste. Diese Strategie reicht aus, um für die implementierte Regelmenge, die Terminierung des Optimierungsvorgangs zu garantieren. Nimmt man allerdings die Regel *Replace By Transitivity* zur Regelmenge hinzu, dann kann es zusammen mit der Regel *Predicate Pushdown* zu einer Endlosschleife kommen. Dies ist beispielsweise dann der Fall, wenn in einer Anfrage ein Prädikat durch die Regel *Replace By Transitivity* repliziert wird, aber anschließend durch die Regel *Predicate Pushdown* nicht alle Replikate aus der Anfrage entfernt werden und somit dasgleiche Prädikat erneut repliziert wird. Um dies zu verhindern sind spezielle Mechanismen und Konfliktlösungsstrategien notwendig. Das in IBM's Starburst implementierte Regelsystem [PLH97], das die Rewrite-Regeln auf QGM anwendet, bietet einen interessanten Ansatz mit Regelklassen. Eine Regelklasse besteht aus einer Sammlung von Regeln und einer Kontrollstrategie, die spezifiziert, wie die auszuführende Regel der Regelklasse bestimmt wird. Es folgen nun drei Beispiele für Kontrollstrategien:

- *Prioritäten-basiert*: Jede Regel in der Regelklasse ist mit einer Priorität versehen. Es wird unter den Regeln, die die Regelbedingung erfüllen, immer die Regel ausgewählt und ausgeführt, die die höchste Priorität besitzt.
- *Sequentiell*: Jede Regel in der Regelklasse ist mit einer Sequenznummer versehen. Die Regeln, die die Regelbedingung erfüllen, werden nun sequentiell durchlaufen und ausgeführt. Es kann auch eine Obergrenze für die Zahl der Iterationen festgelegt werden.
- *Zufällig*: Unter den Regeln, die die Regelbedingung erfüllen, wird immer nach Zufall eine Regel ausgewählt, die zur Ausführung gebracht wird.

Somit kann jede Regelklasse mit einer passenden Kontrollstrategie versehen werden. Ein weiterer Ansatz, der auch in Starburst integriert ist, ist die Vergabe eines Budgets. Wird eine Regelklasse aufgerufen, dann wird gleichzeitig auch ein Budget mitübergeben. Bei jeder Regelausführung wird ein Teil dieses Budgets verbraucht. Es können nun solange Regeln der Regelklasse ausgeführt werden bis das Budget aufgebraucht ist bzw. bis das restliche Budget für keine weitere Regelausführung mehr ausreicht. Damit kann sichergestellt werden, dass der Optimierer immer terminiert und keine Endlosschleifen entstehen.

5.4 Ein Anwendungsbeispiel

Nachfolgend ist die Anfragesequenz aus dem Beispiel zur Regel *Concat Queries*, die aus den beiden Anfragen *X* und *Y* besteht, in der in Unterabschnitt 4.2.2 beschriebenen Interndarstellung abgebildet:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE sequence SYSTEM "sequence.dtd">

<sequence>

  <query name="X" distinct="no" type="view" result="no">
    <referenced-by>
      <referencing-query name="Y"/>
    </referenced-by>
    <select-clause>
      <attribute-definition name="a" type="INTEGER" constraint="primarykey">
        <attribute name="a" source="F"/>
      </attribute-definition>
      <attribute-definition name="b" type="INTEGER">
        <attribute name="b" source="F"/>
      </attribute-definition>
    </select-clause>
    <from-clause>
      <source name="F" alias="F"/>
    </from-clause>
    <where-clause>
      <gr>
        <attribute name="a" source="F"/>
        <constant value="5"/>
      </gr>
    </where-clause>
  </query>

  <query name="Y" distinct="no" type="view" result="yes">
    <referenced-by>
      </referenced-by>
    <select-clause>
      <attribute-definition name="a" type="INTEGER" constraint="primarykey">
        <attribute name="a" source="X"/>
      </attribute-definition>
    </select-clause>
    <from-clause>
      <source name="X" alias="X"/>
    </from-clause>
    <where-clause>
      <eq>
        <attribute name="b" source="X"/>
        <constant value="0"/>
      </eq>
    </where-clause>
  </query>

</sequence>

```

Es wurde davon ausgegangen, dass die Anfrage Y eine Ergebnisanfrage ist; als Datentyp für die Attribute wurde der Ganzzahldatentyp *INTEGER* gewählt, als Primärschlüssel jeweils das Attribut a.

Der Optimierer wird nun von der Kommandozeile aus wie folgt gestartet:

```
java RuleTest -v q_metadata.xml q.xml q_optimized.xml
```

Die Datei *q_metadata.xml* enthält die Metadaten zur Basistabelle *F*. *F* enthält drei Attribute *a*, *b* und *c*, wobei das Attribut *a* ebenfalls Primärschlüssel ist und das in der Abfragesequenz nicht verwendete Attribut *c* eine Gleitkommazahl vom Typ *FLOAT*. Die Datei *q.xml* enthält die obige Interndarstellung der Abfragesequenz. *q_optimized.xml* ist der Name der Ausgabedatei, in der die optimierte Abfragesequenz abgelegt werden soll. Der Inhalt der Datei *q_metadata.xml* sieht wie folgt aus:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE meta-data SYSTEM "metadata.dtd">

<meta-data>
  <relation name="F" type="table">
    <attribute-definition name="a" type="INTEGER" constraint="primarykey"/>
    <attribute-definition name="b" type="INTEGER"/>
    <attribute-definition name="c" type="FLOAT"/>
  </relation>
</meta-data>
```

Der Prototyp erzeugt die folgende Bildschirmausgabe:

```
-----] Optimizer for Query-Sequences [-----
-----] by Tobias Kraft [-----
Parsing meta-data-file q_metadata.xml
Parsing input-sequence-file q.xml
Starting Rule-Engine
-----

Rule "ConcatQueries" can fire
Rule "PredicatePushdown" can fire
=> Rule "ConcatQueries" fires
    X, Y -> Y (Type 1)
-----

Runtime of Rule-Engine:  50 ms
-----

Writing output-sequence-file q_optimized.xml
```

Zuerst werden die beiden Dateien *q_metadata.xml* und *q.xml* eingelesen, geparkt und je ein DOM-Baum erstellt. Danach wird das Regelsystem initialisiert. Nun wird überprüft, bei welchen Regeln die Regelbedingung erfüllt ist. In diesem Fall ist die Regelbedingung der Regeln *ConcatQueries* und *PredicatePushdown* erfüllt. Da die Regel *ConcatQueries* eine höhere Priorität hat, wird diese gefeuert. Zum besseren Verständnis gibt der Optimierer zusätzlich noch aus, welche Anfragen nun verschmolzen werden und um welchen Fall (Type) es sich handelt. Da für die transformierte Abfragesequenz bei keiner Regel die Regelbedingung erfüllt ist, bricht der Optimierer ab und gibt aus, wieviel Zeit der Optimiervorgang benötigt hat. Die Zeit, die hierbei angegeben wird, bezieht sich rein auf den Optimiervorgang, das Einlesen der Dateien und das Erstellen der zugehörigen DOM-Bäume gehört nicht hierzu. Zuletzt wird noch die optimierte Abfragesequenz in die Datei *q_optimized.xml* geschrieben. Der Inhalt der Datei *q_optimized.xml* sieht wie folgt aus:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE sequence SYSTEM "sequence.dtd">

<sequence>
  <query name="Y" distinct="no" type="view" result="yes">
    <referenced-by/>
    <select-clause>
      <attribute-definition name="a" type="INTEGER" constraint="primarykey">
        <attribute name="a" source="F"/>
      </attribute-definition>
    </select-clause>
    <from-clause>
      <source name="F" alias="F"/>
    </from-clause>
    <where-clause>
      <eq>
        <attribute name="b" source="F"/>
        <constant value="0"/>
      </eq>
      <gr>
        <attribute name="a" source="F"/>
        <constant value="5"/>
      </gr>
    </where-clause>
  </query>
</sequence>
```

Diese optimierte Anfragesequenz entspricht der optimierten SQL-Anfragesequenz aus dem Beispiel zur Regel *Concat Queries*.

Kapitel 6

Zusammenfassung und Ausblick

Das Multi-Query-Rewrite ist ein interessanter neuer Optimieransatz. Einige Beispiele für Anfragesequenzen und Optimierungsmöglichkeiten waren durch das Paper [SWM01] bereits vorgegeben. Die Hauptaufgabe bestand nun darin, die Regeln systematisch zu erfassen und zu formalisieren. Wie sich bei der Suche nach einer geeigneten Interndarstellung und einer geeigneten Notation für die Rewrite-Regeln herausgestellt hat, unterscheidet sich das Multi-Query-Rewrite in einigen Punkten deutlich vom klassischen Single-Query-Rewrite. Die Regeln sind viel komplexer, da sie meist einen recht großen Kontext haben und ganze Anfragemengen ermittelt werden müssen. Deshalb konnte unter den betrachteten Regelnotation keine Notation gefunden werden, die für die Formulierung von Multi-Query-Rewrite-Regeln geeignet ist. Daher wurde für die Formulierung der gefundenen Regeln eine nicht-formale, natürlichsprachliche Notation verwendet. Um die Problematik etwas einzugrenzen, wurde der Sprachumfang von SQL92 zu Beginn auf ein Minimum reduziert. Es wurde allerdings darauf geachtet, dass die gegebenen Beispielsequenzen noch in dieser Untermenge von SQL92 enthalten sind. Die erstellte Regelsammlung ist speziell auf die Problematik des Multi-Query-Rewrite und die zugrundegelegte Anfragesprache zugeschnitten. Sie enthält alle Regeln, die ich im Rahmen dieser Diplomarbeit gefunden habe.

Bei der Suche nach einer Interndarstellung hat sich gezeigt, dass die vorhandenen Interndarstellungen des Single-Query-Rewrite für die Darstellung von Anfragesequenzen und die Anwendung der Multi-Query-Rewrite-Regeln nicht geeignet sind. Ein wichtiger Aspekt war hierbei die Anforderung, dass die optimierten Anfragesequenzen wieder in SQL verfügbar sein müssen. Die Repräsentation in Form von Zeichenketten, auf die dann Zeichenkettenfunktionen und Pattern Matching angewandt werden, ist für einen Vergleich von Anfragen ungeeignet. Die prozeduralen, funktionalen Ansätze hatten den Nachteil, dass sie auf einer anderen Abstraktionsebene angesiedelt sind und eine Rückübersetzung in das deskriptive SQL meist schwierig ist. QGM erfüllt zwar die gestellten Anforderungen, müsste jedoch, da es sich um eine Graph-Struktur handelt, erst noch auf eine geeignete physische Struktur abgebildet werden. Daher hab ich eine Interndarstellung entworfen, die auf XML basiert. Sie ist optimal an die Anforderungen, die sich aus den Regelformulierungen ergeben, angepasst und erweiterbar. Da für XML mehrere Schnittstellen und Anfragesprachen standardisiert sind und hierfür auch Implementierungen existieren, auf die zurückgegriffen werden kann, sinkt auch der Aufwand für die Implementierung des Prototyps.

Bei der Implementierung des Prototyps wurde DOM als Schnittstelle zur Interndarstellung verwendet, sowohl zum Navigieren und Lesen als auch zur Manipulation der Anfragesequenz.

Um den Programmieraufwand in Grenzen zu halten, wurde nur eine Auswahl der Regeln implementiert und zur Ermittlung des Primärschlüssels und des Anfragetyps jeweils die einfachste Lösung gewählt. Die Komplexität der verwendeten Algorithmen erscheint auf den ersten Blick recht hoch; betrachtet man jedoch reale Anfragen und Anfragesequenzen, dann wird deutlich, dass einige Algorithmen schon früh abbrechen und das Auftreten des Worst-Case bei einigen Algorithmen nahezu unwahrscheinlich ist bzw. nur durch bewusste Konstruktion möglich ist. Die praktische Anwendung des Prototyps auf die vorliegenden Beispielsequenzen hat dies bestätigt.

Weiterführende Forschungsarbeiten sollten sich zunächst mit der schrittweisen Erweiterung der zugrundegelegten Anfragesprache befassen und untersuchen, welche Folgen dies für die Interndarstellung und die Rewrite-Regeln hat. Beispielsweise wäre es wichtig zu untersuchen, wie Unteranfragen und Mengenoperationen in die Interndarstellung integriert und in die Rewrite-Regeln miteinbezogen werden können. Es ist auch wichtig zu untersuchen, ob die entwickelte Interndarstellung ebenso für Single-Query-Rewrite-Regeln geeignet ist und, falls nicht, woran dies liegt bzw. wie sie modifiziert werden muss, damit auch Single-Query-Rewrite-Regeln auf dieser Interndarstellung arbeiten können. Die Implementierung der restlichen Regeln und die anschließende Modifikation und Erweiterung des Regelsystems ist ein weiterer interessanter Punkt. Um die Qualität des Optimierers festzustellen, bieten sich Leistungsmessungen auf den vorliegenden Beispielsequenzen und den zugehörigen optimierten Versionen an.

Literaturverzeichnis

- [B00] Neil Bradley: *The XML Companion 2nd Edition*. Addison Wesley Publishing Company, 15. Januar 2000
- [B97] Franz Brandmayer: *Parallele Anfrageausführung in MIDAS*. Diplomarbeit, Technische Universität München, Institut für Informatik, 15. Februar 1997
- [B99] M. Bauer et al.: *Oracle8i Tuning, Part No. A67775-01, Release 8.1.5*. Oracle Corporation, Februar 1999, Kapitel 31
- [CD89] M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuh, E.J. Shekita, S.L. Vandenberg: *The EXODUS Extensible DBMS Project: An Overview*. in Readings in Object-Orient Database Systems, S. Zdonik and D. Maier, eds., Morgan-Kaufman Publ. Co., 1989
- [CZ98] M. Cherniack, S. Zdonik: *Inferring Function Semantics to Optimize Queries*. In Proc. of the 24th VLDB Conference, New York, 1998
- [DD97] C.J. Date, Hugh Darwen: *A Guide to the SQL Standard 4th Edition*. Addison Wesley Publishing Company, 6. März 1997
- [FRV95] D. Florescu, L. Raschid, P. Valduriez: *Query Reformulation in Multidatabase Systems using Semantic Knowledge*. 1995
- [GUW00] H. Garcia-Molina, J.D. Ullman, J. Widom: *Database System Implementation*. Prentice Hall, Upper Saddle River, New Jersey 07458, 2000
- [GWMS01] C. Galindo-Legaria, Q. Wang, D. Maier, L. Shapiro: *Query Catalysis: A New Approach to The Theory and Practice of Query Flattening*. Vortrag am IPVR, Uni Stuttgart, 06.08.2001
- [HFLP89] L.M. Haas, J.C. Freytag, G.M. Lohman, H. Pirahesh: *Extensible Query Processing in Starburst*. In Proc. ACM-SIGMOD International Conference on Management of Data, S. 377-388, Portland, May-June 1989
- [HRU96] V. Harinarayan, A. Rajaraman and J. D. Ullman. *Implementing data cubes efficiently*. ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 1996
- [K01] Michael Kay: *XSLT Programmer's Reference 2nd Edition*. Wrox Press Ltd, April 2001

- [M95] B. Mitschang: *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte*. Vieweg-Verlag, Braunschweig/Wiesbaden, 1995
- [PHH92] H. Pirahesh, J.M. Hellerstein, W. Hasan: *Extensible/Rule Based Query Rewrite Optimization in Starburst*. In Proc. ACM-SIGMOD International Conference on Management of Data, S. 39-48, San Diego, June 1992
- [PLH97] H. Pirahesh, T.Y.C. Leung, W. Hasan: *A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS*. In Proc. of the 13th International Conference on Data Engineering, IEEE, 1997
- [S96] Susanne Stamp: *Anfrageoptimierung von OQL mit dem Cascades Optimizer Framework*. Diplomarbeit, Technische Universität München, Institut für Informatik, 15. November 1996
- [SWM01] H. Schwarz, R. Wagner, B. Mitschang: *Improving the Processing of Decision Support Queries: Strategies for a DSS Optimizer*. Technical Report TR-2001-02, Universität Stuttgart, IPVR, 2001
- [W3C00a] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, 6. Oktober 2000, siehe <http://www.w3.org/TR/2000/WD-xml-2e-20000814>
- [W3C00b] World Wide Web Consortium: *Document Object Model (DOM) Level 2 Core Specification Version 1.0*. W3C Recommendation, 13. November 2000, siehe <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113>
- [W3C01] World Wide Web Consortium: *XQuery 1.0: An XML Query Language*. W3C Working Draft, 7. Juni 2001, siehe <http://www.w3.org/TR/2001/WD-xquery-20010607>
- [W3C99] World Wide Web Consortium: *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, 16. November 1999, siehe <http://www.w3.org/TR/1999/REC-xslt-19991116>
- [Wagn00] Ralf Wagner: *Realisierung und Optimierung einer OLAP-Anwendung im Handelsbereich*. Studienarbeit Nr. 1770, Universität Stuttgart, Fakultät Informatik, IPVR, Abteilung Anwendersoftware, 2000

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

(Tobias Kraft)

