Entwurf und Implementierung eines algebraischen Optimierers für das verteilte Datenbanksystem HE$_A$D

Diplomarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von

Lüneberg, Jens

geboren am 20.09.1968 in Templin

Betreuer: Prof. Dr. A. Heuer

Dr. H. Meyer

Dipl. Inf. U. Langer

Abgabedatum: 30.05.1995

Danksagung

Herrn Prof. Dr. A. Heuer, Dr. H. Meyer, Dipl. Inf. U. Langer danke ich für die aktiven Diskussionen und die sehr gute Unterstützung und Förderung bei der Durchführung dieser Diplomarbeit.

Rostock, 25.05.1995 Jens Lüneberg

Zusammenfassung

Die algebraische Optimierung basiert auf Heuristiken. Bei der Generierung äquivalenter optimierter Anfragen werden nur Informationen aus dem globalen Schema und dem Fragmentationsschema genutzt. Entscheidungen bezüglich bestimmter Transformationen beruhen damit auf Annahmen bezüglich ihres Nutzens und nicht auf konkreten Kosten. Innerhalb dieser Arbeit werden eine Vielzahl von Heuristiken für die Anfrageoptimierung in einem verteiltem Datenbanksystem vorgestellt und deren Zusammenwirken innerhalb eines Optimierers beschrieben. Neben den aus zentralisierten Datenbanksystemen bekannten Heuristiken kommen in verteilten Datenbanksystemen weitere hinzu. Diese berücksichtigen die Fragmentierung der Relationen. Bei Verwendung der Relationenalgebra ist es nicht möglich, die Fragmentierungsinformation von horizontalen Fragmenten für die Optimierung zu nutzen. Die Erweiterung der Relationenalgebra zur Algebra AQUAREL löst dieses Problem. Der in dieser Arbeit entworfene Optimierer wurde mit Hilfe des Werkzeugs Twig implementiert.

Abstract

The algebra optimization applicates heuristics. It uses only information from the global and fragmentation layer. Therefore the decision to select a concrete transformation rule depends on the expection of advantage but not on specific costs. This paper represents a multitude of heuristics for the query optimization in distributed database systems and their cooperation during the optimization. In distributed database systems there are additional heuristics in comparison to the well-known heuristics from centralized database systems. These heuristics take into account the fragmentation of relations. The relational algebra does not support the handling of fragmentation information of horizontal fragments. To solve this problem the relational algebra was extended to the algebra AQUAREL. The optimizer designed in this work was implemented on the base of the tool Twig.

CR-Klassifikation

C.2.4	Distributed systems (H.2.4)	Verteilte Systeme
	Distributed databases	Verteilte Datenbanken
D.3.4	Processors	Sprachprozessoren
	Optimization	Optimierung
H.2.3	Languages	Sprachen
	Query languages	Anfragesprachen
H.2.5	Heterogeneous databases	Heterogene Datenbanken

Key Words: Verteilte Datenbanksysteme, Verteilte Anfragebearbeitung, Algebraische Optimierung, Optimierer-Generatoren

Abkürzungen

Abb.	Abbildung
DB	Datenbank
DBI	Datenbankimplementierer
DBS	Datenbanksystem
EM	Execution Monitor
GAG	Gerichteter Azyklischer Graph
GQM	Global Query Manager
HEAD	Heterogeneous Extensible and Distributed Database Management System
LAN	Local Area Network
RA	Relationenalgebra
VDBS	Verteiltes Datenbanksystem

1. Einleitung

Ziel dieser Arbeit soll der Entwurf eines algebraischen Optimierers für die erweiterte Relationenalgebra im System HEAD sein. Ausgangspunkt der Optimierung ist hierbei ein Ausdruck der erweiterten Relationenalgebra bezüglich des globalen konzeptuellen Schemas. Im Ergebnis der algebraischen Optimierung müssen normalisierte, optimierte Algebraausdrücke vorliegen. Dabei ist zur Beschränkung der Komplexität eine Zerlegung der Optimierung in Einzelschritte anzustreben. Da die Optimierung heuristisch erfolgen soll, ist der Einsatz verschiedener Heuristiken dahingehend zu untersuchen. Weiterhin ist im Rahmen dieser Arbeit der Einsatz der Optimierer-Generatoren Volcano OptGen und Twig zu prüfen, und es ist die teilweise Implementierung des Optimierers anzustreben.

Die Optimierung von Anfragen kann nach dem Wissen, daß innerhalb der Optimierung Berücksichtigung findet, in Phasen zerlegt werden. Die 3-Ebenen-Architektur von relationalen DBS beinhaltet die Abstraktionsebenen externes, globales und internes Schema und die Unterteilung in konzeptuelle und interne Optimierung. Bei dem in HEAD zugrundeliegenden Architekturmodell wird eine Erweiterung des konzeptuellen Schemas zum globalen konzeptuellen Schema, Fragmentationsschema, Allokationsschema und lokalem konzeptuellem Schema vorgenommen. In HEAD erfolgt eine Zerlegung der Optimierung in algebraische und interne Optimierung. Die in dieser Arbeit zu behandelnde algebraische Optimierung nutzt ausschließlich Wissen des globalen konzeptuellen Schemas und des Fragmentationsschemas.

Das Ziel der algebraischen Optimierung ist es, einen die Anfrage repräsentierenden Algebraausdruck mittels algebraischer Transformationsregeln in einen äquivalenten optimierten Ausdruck zu überführen. Unter Nutzung der existierenden Äquivalenzregeln von Ausdrücken der Relationenalgebra lassen sich jedoch zu einem gegebenen Algebraausdruck eine Vielzahl äquivalenter Ausdrücke generieren und der Reihe nach auf der Basis von Kostenmodell, Zielfunktion und Datenbankzustand bewerten. Ein Beispiel hierfür ist die Vorgehensweise in R* [ML86]. Solche exakten Optimierungsverfahren durchsuchen den Lösungsraum bezüglich der algebraischen Dimension vollständig und wählen anschließend unter Nutzung einer Kostenfunktion den besten Bearbeitungsplan aus.

Sie liefern damit zwar bessere Ergebnisse als heuristische Verfahren, dennoch sind folgende Punkte zu berücksichtigen:

- Neben der Normalisierung schlecht formulierter Anfragen sollte kein vermeidbarer Mehraufwand für bereits effizient formulierte Anfragen entstehen.
- Eine exakte Optimierung wird dadurch erschwert, daß die Ermittlung der Bearbeitungskosten auf Schätzungen beruht und somit keine präzisen Informationen vorliegen.
- Für HEAD wird vorausgesetzt, daß Anfragen vorwiegend im Dialogbetrieb an das System gestellt werden, wodurch der nötige Aufwand für eine exakte Optimierung eventuell nicht gerechtfertigt ist [Leh88]. Dies gilt besonders für VDBS, da hier aufgrund weiterer Optimierungsdimensionen der zu bearbeitende Suchraum drastisch anwächst. Dieses Anwachsen basiert auf der Möglichkeit der parallelen Bearbeitung einer Anfrage aufgrund der Existenz autonomer Rechner ([Kil90]) und einer möglichen Fragmentierung und/oder Replikation von Teilen der Datenbank, die zusätzliche Alternativen zur Bearbeitung einer globalen Relation hervorbringen.

Heuristische Verfahren hingegen basieren auf Annahmen und nehmen bewußt eine Einschränkung der möglichen Alternativen und damit eventuell ungünstigere Ausführungspläne in Kauf. Sie ermöglichen jedoch die Beschränkung des Suchraumes und somit die Reduzierung des Optimierungsaufwandes.

Die algebraische Optimierung ist wegen ihrer Unabhängigkeit von Implementierungsdetails konzeptionsbedingt heuristisch. Ein relationenalgebraischer Ausdruck wird unter Verwendung allgemeingültiger Regeln in einen äquivalenten optimierten Ausdruck überführt, wobei die Existenz von Replikaten, der Speicherungsort von Fragmenten, die aktuellen Datenbankzustände und die realisierten Zugriffspfade, Dateiorganisationsformen und Auswahlstrategien (z.B. beim Join: Merge-Join, Nested-Loop, Hash-Join usw.) unberücksichtigt bleiben.

Bezüglich der vorgenommenen Transformationen soll die algebraische Optimierung unterschieden werden in:

- Minimierung
- Optimierung

Die Minimierung beschreibt den Prozeß der Eliminierung unrelevanter Operationen und Fragmente innerhalb eines algebraischen Ausdrucks. Der Ausdruck wird auf die zur Beantwortung der Anfrage notwendigen Operationen und Fragmente reduziert.

Die Optimierung beinhaltet die Bestimmung einer optimalen Reihenfolge der Operationen. Der die Anfrage repräsentierende Algebraausdruck ist dazu derart zu transformieren, daß die Reihenfolge der Operationen die bestmögliche Bearbeitungsfolge für die Anfrage darstellt.

Im weiteren soll bei der Betrachtung der algebraischen Optimierung zwischen Minimierung und Optimierung unterschieden werden. Nur sofern eine Unterscheidung nicht erforderlich oder möglich ist, soll verallgemeinert von Optimierung gesprochen werden.

Das HEAD-Projekt

Im Rahmen des HEAD-Projektes (**H**eterogeneous **E**xtensible **a**nd **D**istributed Database Management System) wird der Prototyp eines heterogenen und verteilten Datenbankverwaltungssystems entwickelt. Ziel der Untersuchungen und Analysen ist dabei die Optimierung und Bearbeitung einer Anfrage in einem VDBS.

Aufgrund der unterschiedlichen Leistungsfähigkeit der Rechnerknoten in einer heterogenen Umgebung wird die Einbeziehung der verschiedenen Rechnerknoten entsprechend ihrem Leistungsvermögen in die Anfrageverarbeitung angestrebt.

Dies wird in HEAD durch die Zerlegung des Systems in die relativ eigenständigen Funktionseinheiten:

- Anfrageübersetzung und -optimierung,
- Abarbeitung der Ausführungspläne

und deren leistungsabhängiger Zuordnung zu den Rechnerknoten erreicht. Eine weitere Verfeinerung ergibt sich durch die Aufteilung der Funktionen zur Abarbeitung der Ausführungspläne in:

- Funktionen zur Abbildung physischer Datenstrukturen auf Tupelmengen
- und Funktionen zur mengenorientierten, datenflußgesteuerten Weiterverarbeitung der Tupel.

Die beschriebene Zerlegung des Systems in Funktionseinheiten sowie die Anwendung von Pipelining und Datenflußsteuerung bedingt die in Abb. 1 dargestellte funktionelle Architektur von HEAD.

Der SQL-Compiler und der Global Query Manager (GQM) beinhalten die Übersetzung, Optimierung und Zerlegung einer Anfrage. Der SQL-Compiler liefert als Ergebnis einen zur SQL-Anfrage äquivalenten Ausdruck der erweiterten Relationenalgebra (primäres Datenflußprogramm), welcher auf dem globalen konzeptuellen Schema basiert.

Der GQM nimmt eine Optimierung des primären Datenflußprogramms bezüglich der Antwortzeit vor. Dazu wird dieser zunächst algebraisch optimiert, wobei gleichzeitig eine Abbildung des Datenflußprogramms vom konzeptuellen Schema auf das Fragmentationsschema erfolgt. Im Anschluß an die algebraische Optimierung erfolgt unter Einbeziehung von Wissen aus dem Allokationsschema und dem internen Schema sowie von Informationen über die aktuelle Lastsituation eine kostenbasierte Optimierung mittels eines auf Simulated Annealing beruhenden approximativen Optimierungsverfahrens. Die hierbei durch das Kostenmodell zu berücksichtigenden Kosten lassen sich wie folgt einteilen: Übertragungskosten, I/O-Kosten, CPU-Kosten und aus Konkurrenzsituationen resultierende Kosten (Warten auf Ressourcen). Die Kosten werden durch eine Kostenfunktion bestimmt, in die z.B. Informationen über

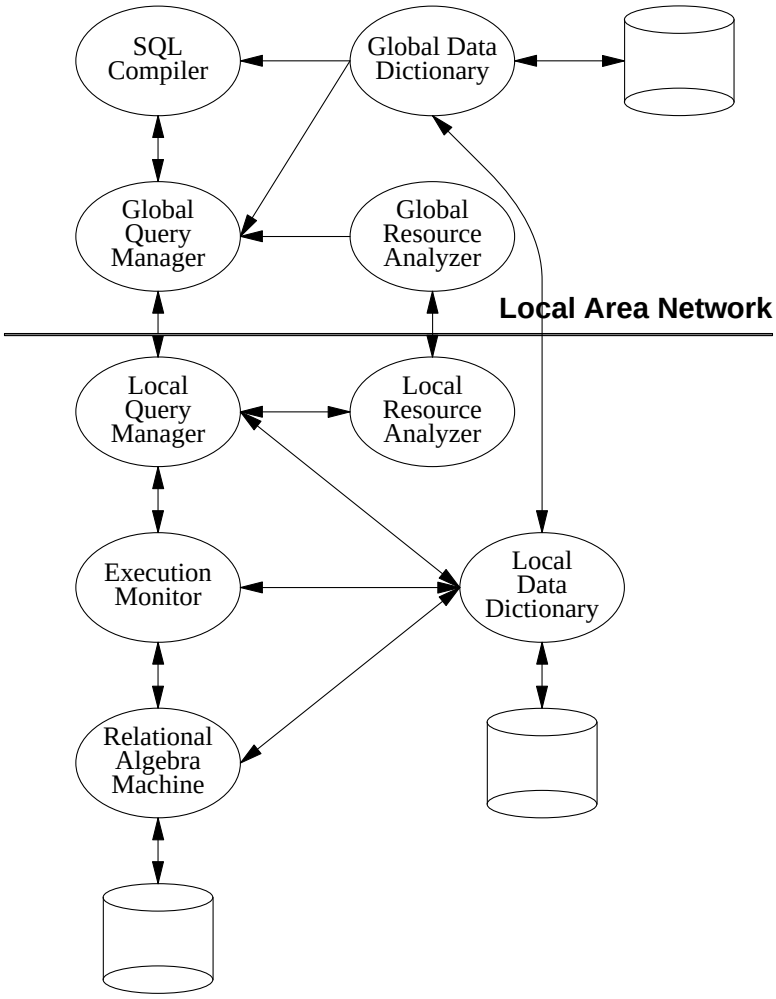


Abb. 1: Funktionelle Architektur von HE$_A$D

Größe (belegte Seiten) und Tupelanzahl von Relationen, über die Selektivität von Attributwerten und Informationen über die Existenz und die Belastung von Rechnerknoten eingehen. Ausführungen zum Kostenmodell und zur Lastbalancierung von HE$_A$D finden sich in [Fla93, Lin94]. Der Aufbau des HE$_A$D-Optimierers ist in Abbildung 2 dargestellt.

Im Anschluß an die Optimierung wird durch den GQM eine Zerlegung des Anfragebaums in parallel ausführbare Teilbäume und deren Zuordnung zu Rechnerknoten vorgenommen. Hierzu startet der GQM auf jedem betroffenen Rechnerknoten einen Local

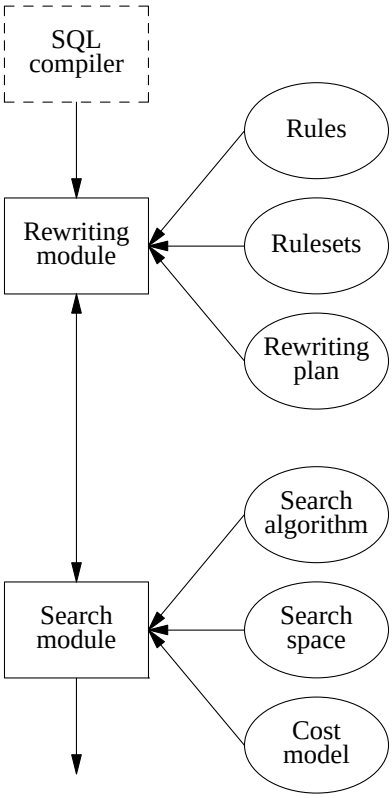


Abb. 2: Aufbau des HEAD-Optimierers

Query Manager, der die auf dem Rechnerknoten zu bearbeitende Teilanfrage entgegennimmt und einen Execution Monitor (EM) aktiviert. Der EM analysiert die Teilanfrage und organisiert die Abarbeitung der einzelnen Algebraoperationen.

Die Relationale Algebra Maschine stellt die Implementierungen für die Operationen der Relationenalgebra bereit und organisiert Pipelining und Interprozeßkommunikation.

Die für die Optimierung und Bearbeitung der Anfrage notwendigen Lastinformationen werden durch den Global und Local Resource Analyser gewonnen und verwaltet, während die Metadaten vom Global und Local Data Dictionary bereitgestellt werden.

Gliederung der Arbeit

Im Anschluß an diese Einleitung werden im Kapitel 2 die Algebren vorgestellt, auf denen die algebraische Optimierung in HEAD beruht. Hierbei handelt es sich um die Relationenalgebra, die einerseits um Operationen erweitert wird und andererseits eine Erweiterung bezüglich der Operanden erfährt, die es ermöglicht, die Fragmentierungsbedingung horizontaler Fragmente in den Optimierungsprozeß einzubeziehen.

Im Kapitel 3 wird ein algebraischer Optimierer entworfen. Dabei werden zunächst mögliche Heuristiken unabhängig voneinander vorgestellt. Anschließend wird deren Zusammenwirken innerhalb eines Optimierers und die Zerlegung des Optimierungsprozesses in mehrere Phasen beschrieben.

Kapitel 4 beschäftigt sich mit der Betrachtung von Werkzeugen, die die Generierung eines Optimierers unterstützen. Darauf basierend wird im Kapitel 5 die Realisierung des in dieser Arbeit entworfenen Optimierers beschrieben.

Verschiedene Ansatzpunkte für weitergehende Arbeiten im Rahmen der Entwicklung eines algebraischen Optimierers für HEAD werden im Kapitel 6 aufgeführt.

Der algebraische Optimierer basiert auf die im Anhang 1 aufgeführten Transformationsregeln.

2. Algebren

2.1. Erweiterte Relationenalgebra

Die in HEAD intern verwendete relationale Anfragesprache ist die um Operationen erweiterte Relationenalgebra. Dem algebraischen Optimierer wird ein die Anfrage repräsentierender Ausdruck der erweiterten Relationenalgebra übergeben und als Ergebnis ein nach Heuristiken optimierter Ausdruck zurückgeliefert.

Die elementaren Operationen der Relationenalgebra sind einstellige und zweistellige Operationen, die DB-Relationen als Operanden haben und DB-Relationen als Ergebnis liefern. Welche Operationen genau die Relationenalgebra bilden, ist in der Literatur nicht genau festgelegt. Hierbei wird in vielen Veröffentlichungen [Sau91, CP85] die Umbenennungsoperation vergessen. Dabei ist die Relationenalgebra ohne die Umbenennung nicht vollständig. Um z.B. zwei Relationenschemata für eine Vereinigung kompatibel oder um die Joinattribute vor einem natürlichen Verbund gleichnamig zu machen, ist die Umbenennung unbedingt erforderlich. Die Einbeziehung der Umbenennungsoperation in die Relationenalgebra findet sich in [Heu92, Rec93, KK93].

Datenbankanwendungen erfordern oft Operationen, die nicht mit der Relationenalgebra ausgedrückt werden können. Der algebraische HEAD-Optimierer basiert auf einer erweiterten Relationenalgebra, die sich aus den Operationen der Relationenalgebra und den Operationen *GROUP*, *SCAN*, *DUP* und *FRAG* zusammensetzt. Hierbei verkörpert die Operation SCAN den Zugriff auf Daten einer physischen Datenbank, DUP die Realisierung eines gerichteten azyklischen Graphen und FRAG die Eigenschaft einer Zwischenrelation. Jedoch stellen diese Operationen lediglich Implementierungsalternativen dar, die keine Weiterverarbeitung von (Zwischen)-Relationen vornehmen. Nachfolgend werden die Operationen der Relationenalgebra mit der Erweiterung GROUP beschrieben.

Operationen der erweiterten Relationenalgebra

Selektion: $SL_F(R)$

Die Selektion wählt aus einer Relation mittels einer Bedingung über Attributwerte bestimmte Tupel aus. F sei eine logische Formel, die sich wie folgt zusammensetzt:

(1) Operanden sind Attributnamen oder Konstanten

(2) Operanden werden mit den arithmetischen Vergleichsoperatoren $\{<, \leq, \geq, >, =, \neq\}$ zu Termen verbunden

(3) Terme können mit den logischen Operatoren $\wedge, \vee, \neg$ verknüpft werden.

Die Relation $SL_F(R)$ enthält alle Tupel aus R, die die Bedingung F erfüllen.

Projektion: $PJ_X(R)$

Mit der Projektion werden Attribute aus einer Relation ausgeblendet. $PJ_{A1,\ldots,Ak}(R)$, mit $\{A_1, \ldots, A_k\} \subseteq Attr(R)$ liefert die Menge aller k-Tupel, für die die Relation R ein Tupel enthält, das in den Attributen $A_1, \ldots, A_k$ mit diesem übereinstimmt, wobei Duplikate unter den k-Tupeln eliminiert werden.

Kartesisches Produkt: $R_1\ CP\ R_2$

Sei $R_1(A_1, \ldots, A_n)$ vom Grad n und $R_2(B_1, \ldots, B_m)$ vom Grad m. Die Relation $R_1\ CP\ R_2$ enthält dann (n+m)-Tupel derart, daß die ersten n Attribute ein Tupel aus R_1 und die letzten m Attribute ein Tupel aus R_2 repräsentieren. Jedes Tupel aus R_1 wird mit jedem Tupel aus R_2 verbunden.

Union (Vereinigung): $R_1\ UN\ R_2$

Es gelte $Attr(R_1) \equiv Attr(R_2)$, dann enthält die Relation $R_1\ UN\ R_2$ alle Tupel, die entweder in R_1 oder in R_2 oder in R_1 und in R_2 enthalten sind.

Differenz: $R_1\ DF\ R_2$

Es gelte $Attr(R_1) \equiv Attr(R_2)$, dann enthält die Relation $R_1\ DF\ R_2$ alle Tupel aus R_1, die nicht in R_2 vorkommen.

Umbenennung: $REN_{RenL}(R)$

$REN_{RenL}(R) \Rightarrow R'$ nimmt für die Attribute von R eine Umbenennung entsprechend der Liste *RenL* vor. *RenL* bezeichnet eine Liste von einem oder mehreren Ausdrücken der Form $A_{alt} \rightarrow A_{neu}$. Diese Liste beschreibt die vorzunehmenden Umbenennungen, wobei gilt:

$$A_{alt} \in Attr(R),\ A_{neu} \notin (Attr(R) - A_{alt}),\ Attr(R') = (Attr(R) - A_{alt}) \cup A_{neu}$$

Die Umbenennung der Attribute $Attr(R)$ in $Attr(R')$ erfolgt hierbei durch eine Bijektion *ren*, die jedem in der Umbenennungsliste aufgeführten altem Attribut A ein dazu isomorphes Attribut A' zuordnet, daß sich von A nur durch seinen Namen unterscheidet. Die nicht in *RenL* aufgeführten Attribute werden von *ren* identisch abgebildet.

Join: $R_1\ JN_F\ R_2$

$R_1\ JN_F\ R_2$ enthält alle Tupel des Kartesischen Produktes R_1 und R_2, die die Bedingung F, über Attribute von R_1 und R_2, erfüllen. F sei eine logische Formel, die sich wie folgt zusammensetzt:

(1) Operanden sind Attributnamen.

(2) Operanden werden mit einem arithmetischen Vergleichsoperator $\Theta \in \{<, \leq, \geq, =, \neq\}$ zu Termen $A\ \Theta\ B$ verbunden, wobei $A \in R_1$ und $B \in R_2$.

(3) Terme können mit den logischen Operatoren $\wedge, \vee, \neg$ verknüpft werden.

Es gilt: $R_1\ JN_F\ R_2 \equiv SL_F(R_1\ CP\ R_2)$

Werden in der Joinbedingung zwei Operanden auf Gleichheit getestet, dann ist der Join ein "Equijoin". Ein Equijoin, bei dem nach dem Join alle doppelten Attribute entfernt werden, ist ein Natural-Join.

Semijoin: $R_1\ SJ_F\ R_2$

$R_1\ SJ_F\ R_2$ enthält alle Tupel des Joins bezüglich der Bedingung F über R_1 und R_2, beschränkt auf die Attribute von R_1. Es gilt:

$$R_1\ SJ_F\ R_2 \equiv PJ_A(R_1\ JN_F\ R_2) \equiv SL_F(PJ_A(R_1\ JN_F\ R_2)), \text{ mit } A = Attr(R_1)$$

Division (Quotient): $R_1\ DV\ R_2$

Sei $R_1(A_1, \ldots, A_n, B_1, \ldots, B_m)$ vom Grad (n+m) und $R_2(B_1, \ldots, B_m)$ vom Grad m, dann enthält die Relation $R_1\ DV\ R_2$ alle n-Tupel x derart, daß für alle m-Tupel y aus R_2 das zusammengesetzte Tupel xy in R_1 enthalten ist. Es gilt:

$$R_1\ DV\ R_2 \equiv PJ_{A1,\ldots,An}(R_1)\ DF\ PJ_{A1,\ldots,An}((PJ_{A1,\ldots,An}(R_1)\ CP\ R_2)\ DF\ R_1).$$

Intersection (Durchschnitt): $R_1\ IN\ R_2$

Es gelte $Attr(R_1) \equiv Attr(R_2)$, dann enthalte $R_1\ IN\ R_2$ alle Tupel von R_1, die auch in R_2 enthalten sind. Es gilt:

$$R_1\ IN\ R_2 \equiv R_1\ DF\ (R_1\ DF\ R_2)$$

GROUP: $GB_{G,AF}(R)$

Die Operation GROUP realisiert das Gruppieren von Tupel in disjunkte Teilmengen und das Anwenden einer beliebigen Funktion. G ist die Attributmenge nach der R gruppiert werden soll und AF ist die Funktion, die über jede Gruppe angewendet werden soll. $GB_{G,AF}(R)$ liefert eine Relation, dessen Relationenschema aus den Attributen von G und der Funktion AF gebildet wird. Die Anzahl der Tupel ist gleich der Anzahl der Gruppen, die sich bezüglich G in R bilden lassen. Die Attribute von G enthalten die Werte der Attribute, nach denen gruppiert wurde, und das Attribut AF enthält die Werte, die die Funktion für die Gruppen liefert. Bei der GROUP-Operation ist es möglich, nur G oder nur AF zu verwenden. Damit kann erreicht werden, daß nur gruppiert bzw. nur eine Funktion berechnet wird.

Transformationsregeln

Die algebraische Optimierung und Minimierung basiert auf Regeln der starken Äquivalenz von relationenalgebraischen Ausdrücken. Unter Verwendung der zuvor beschriebenen Operationen sind im Anhang 1 die dem Optimierer zur Verfügung stehenden Transformationsregeln aufgeführt. Diese Regeln finden sich unter anderem in [CP85,Ull80,Ull88,CH82].

Die aufgeführten Regeln sind bezüglich der durch ihre Anwendung verursachten Manipulationen geordnet. Diese Ordnung erleichtert die Zuordnung der Regeln zu den entsprechenden Heuristiken.

In der Arbeit von [Gru94] wird die Umbenennung als Operation betrachtet, die für die Optimierung unrelevant ist. Jedoch sind viele Transformationsregeln mit einer Bedingung oder/und Konstruktionsregel verknüpft, welche Mengenoperationen über Attributmengen vornehmen und die Kenntnis der aktuellen Namensausprägung eines Attributs voraussetzen. Hierbei reicht das interne Führen einer Symboltabelle bezüglich der vorgenommenen Umbenennungen nicht aus. Ein Attribut kann innerhalb des Gesamtbaumes z.B. drei verschiedene Namen haben, deren aktuelle Ausprägung etwa in der Qualifizierung einer Selektionsoperation allein von der Lage der Selektion im Baum abhängig ist. Deshalb muß die Umbenennungsoperation als eine für den Optimierungsprozeß relevante Operation angesehen werden. Aus Gründen der Vereinfachung soll in dieser Arbeit jedoch das Problem des Domain mismatch, also das Auftreten von Namens-, Datenrepräsentations- und Wertebereichskonflikten, nicht berücksichtigt werden. Es sei deshalb vorausgesetzt, daß durch den Optimierer nur solche Anfragen bearbeitet werden, die das Auftreten von Domain mismatch ausschließen und somit auch keine Umbenennung von Attributnamen erfordern. Aus diesem Grund berücksichtigen die aufgeführten Transformationsregeln die Umbenennungsoperation nicht. Transformationsregeln bezüglich des Umbenennungsoperators finden sich in [Rec93].

Implementierung der Operationen der erweiterten Relationenalgebra

In [Kru94] ist die Implementierung der Operationen der erweiterten Relationenalgebra des HEAD-Systems beschrieben.

Die Implementierung erfolgt unter Ausnutzung von vertikaler und von horizontaler Parallelität. Diese wird erreicht durch die separate Bearbeitung unabhängiger Teilpläne und durch Algorithmen zum Partitionieren und Zusammenführen des Datenstroms. Des weiteren erfolgt die Bearbeitung von Projektion und Selektion grundsätzlich mittels Pipelining. Eine allgemeine Vorstellung der Parallelisierungsarten wird im Abschnitt 3.5.4 vorgenommen.

Die Implementierung der Projektion in HEAD unterscheidet sich von der relationen-algebraischen Projektion dadurch, daß sie keine Duplikateneliminierung vornimmt. Die Entfernung von Duplikaten erfolgt, sofern erforderlich, explizit.

2.2. AQUAREL - Algebra der qualifizierten Relationen

2.2.1. Vorbetrachtungen

Die Relationenalgebra ist ohne Einschränkung für Ausdrücke verwendbar, die auf Fragmente und nicht auf komplette Relationen basieren. Der Grund ist, daß Fragmente lediglich horizontale bzw. vertikale Beschränkungen von logischen DB-Relationen darstellen und damit wie Relationen (Zwischenrelationen) behandelt werden können. Allerdings gehen bei der Verwendung der Relationenalgebra für die Optimierung hilfreiche Fragmentierungsinformationen verloren.

Unter Verwendung der Algebra AQUAREL kann dieses Problem gelöst werden. Die Algebra AQUAREL ist eine Erweiterung der Relationenalgebra, die lediglich deren Operanden, die Relationen, um eine Qualifizierung ergänzt und vorschreibt wie diese Qualifizierung zu bilden ist. Dadurch wird es möglich, Informationen aus dem Fragmentationsschema für den Optimierungsvorgang zu verwenden.

2.2.2. Fragmentierung

Das Fragmentationsschema beschreibt die Zerlegung von globalen Relationen in Fragmente. Globale Relationen können auf zwei verschiedene Arten fragmentiert sein, horizontal und/oder vertikal. Bei der horizontalen Fragmentierung ist die globale Relation bezüglich der Tupelmenge partitioniert. Dabei kann die horizontale Fragmentierung primär nach einem Attribut der zu fragmentierenden Relation oder abgeleitet nach dem Attribut einer anderen Relation erfolgen. Bei der vertikalen Fragmentierung ist die globale Relation mittels der Projektion in Gruppen von Attributen zerlegt.

Des weiteren können beide Fragmentierungsarten kombiniert auftreten. In HEAD wird jedoch davon ausgegangen, daß eine Relation entweder vertikal oder horizontal fragmentiert ist. Die Kombination von Fragmentierungen wird ausgeschlossen.

2.2.3. Einführung der Algebra AQUAREL

Die Algebra AQUAREL ermöglicht das Verwalten von Informationen aus dem Fragmentationsschema. Fragmente werden wie in der Relationenalgebra als Relationen betrachtet, jedoch erweitert um Informationen über deren Fragmentierung. Die Eigenschaft vertikaler Fragmente ist die Attributmenge eines jeden Fragments. Zur Berücksichtigung dieser Eigenschaft sind keine Zusatzinformationen notwendig, da die Attributmenge inhärente Eigenschaft einer jeden Relation ist. Jedoch ist die Fragmentierungsinformation horizontaler Fragmente mit der Relationenalgebra nicht beschreibbar bzw. sie kann lediglich durch zusätzliche Selektionsoperationen auf die Fragmente ausgedrückt werden [Ull88]. Dies stellt in den meisten Fällen zwar eine akzeptable Lösung dar, ist aber für Relationen, die nach einem abgeleiteten Attribut fragmentiert sind, nicht anwendbar. Dieses Problem wird gelöst durch die Verwendung der Algebra AQUAREL, denn sie erlaubt das uneingeschränkte Einbeziehen von Wissen über die Fragmentierung horizontaler Fragmente. Sie wird in [CP85] als "Algebra der qualifizierten Relationen" eingeführt und stellt eine Erweiterung der Relationenalgebra dar. Die Verwendung von AQUAREL ermöglicht es, Anfragen, die sich nur auf einzelne Fragmente und nicht auf komplette Relationen beziehen, durch Eliminierung nicht benötigter Fragmente zu minimieren.

AQUAREL nutzt qualifizierte Relationen als Operanden. Eine qualifizierte Relation ist eine um eine Qualifizierung erweiterte Relation. Bei der Qualifizierung handelt es sich um eine Bedingung oder Eigenschaft, die von allen Tupeln der Relation erfüllt werden muß. Eine qualifizierte Relation wird beschrieben durch das Paar $[R: q_R]$. R ist hierbei eine durch einen Ausdruck der Relationenalgebra beschriebene Relation und wird in AQUAREL als Körper der qualifizierten Relation bezeichnet; q_R ist ein Prädikat und wird als die Qualifizierung der qualifizierten Relation bezeichnet. Für die Algebra AQUAREL gilt: Zwei Ausdrücke sind äquivalent, wenn ihre Körper äquivalente Ausdrücke der Relationenalgebra sind und ihre Qualifizierungen die gleiche Aussage repräsentieren.

Die Qualifizierung muß nicht bewertet werden. Werden z.B. bei einer Projektion Attribute entfernt, die in der Qualifizierung vorkommen, ist eine Bewertung nicht mehr möglich und auch nicht gefordert. Das liegt daran, daß Qualifizierungen ergänzende statt umfassende Informationen darstellen. Qualifizierungen dienen nicht der exakten Berechnung der Ergebnistupelmenge, sondern der Bewertung der in Form der Qualifizierung vorliegenden Tupeleigenschaften einer Relation. Sind diese Eigenschaften inkonsistent, liefert der durch die qualifizierte Relation repräsentierte Teilausdruck bei jeglicher Tupelbelegung die leere Menge und kann somit bereits im Vorfeld der Anfragebearbeitung eliminiert werden.

Die Qualifizierung von Fragmenten horizontal fragmentierter Relationen ist der Attributwert, nach dem partitioniert wurde. Die Qualifizierung von vertikal fragmentierten Relationen hingegen ist leer.

Die nachfolgend angegebenen Regeln definieren das Anwenden einer relationenalgebraischen Operation auf eine qualifizierte Relation. Sie beschreiben wie die Qualifizierung der resultierenden Relation zu bilden ist [CP85].

Selektion: $SL_F[R: q_R] \Rightarrow [SL_F(R): F \wedge q_R]$

Alle Tupel der durch die Selektion erzeugten Zwischenrelation müssen die Bedingung $F \wedge q_R$ erfüllen.

Projektion:

$PJ_A[R\colon q_R] \Rightarrow [PJ_A(R)\colon q_R]$

Die Qualifizierung der qualifizierten Relation bleibt nach der Anwendung einer Projektion unverändert.

Kartesisches Produkt:

$[R_1\colon q_{R1}]\ CP\ [R_2\colon q_{R2}] \Rightarrow [R_1\ CP\ R_2\colon q_{R1} \wedge q_{R2}]$

Union:

$[R_1\colon q_{R1}]\ UN\ [R_2\colon q_{R2}] \Rightarrow [R_1\ UN\ R_2\colon q_{R1} \vee q_{R2}]$

Differenz und Intersection:

$[R_1\colon q_{R1}]\ DF\ [R_2\colon q_{R2}] \Rightarrow [R_1\ DF\ R_2\colon q_{R1}]$

Wie schon bei der Projektion wird hier deutlich, daß Qualifizierungen nur ergänzende Informationen darstellen. Das Ergebnis der Differenz von R_1 und R_2 enthält als Qualifizierung q_{R1}. Diese Qualifizierung liefert keine umfassende Information, weil sie alle Tupel von R_1 und somit wahrscheinlich mehr Tupel als die Operation $R_1\ DF\ R_2$ qualifiziert. Die Qualifizierung $q_{R1} \wedge \neg q_{R2}$ wäre aber zu einschränkend. Es könnten Tupel in $[R_1\colon q_{R1}]$ existieren, die die Bedingung $q_{R1} \wedge \neg q_{R2}$ nicht erfüllen und damit auch nicht qualifiziert werden, obwohl $[R_2\colon q_{R2}]$ vielleicht überhaupt kein solches Tupel enthält. Qualifizierungen liefern niemals weniger, können aber eventuell mehr Tupel als ihr Körper liefern.

Dies wird auch bei der Betrachtung der Qualifizierung des Ergebnisses von $R_1\ IN\ R_2$ und $R_2\ IN\ R_1$ deutlich. Die Operation Intersection ist kommutativ, so daß die Qualifizierung des Ergebnisses in beiden Fällen dieselbe sein müßte. Gemäß den Transformationsregeln der Relationenalgebra gilt:

$$R_1\ IN\ R_2 \equiv R_1\ DF\ (R_1\ DF\ R_2)$$

Die Operation Intersection, angewandt auf qualifizierte Relationen, lautet dann wie folgt:

$[R_1: q_{R1}]\ IN\ [R_2: q_{R2}] \Rightarrow$
$[R_1: q_{R1}]\ DF\ ([R_1: q_{R1}]\ DF\ [R_2: q_{R2}]) \Rightarrow$
$[R_1: q_{R1}]\ DF\ [R_1\ DF\ R_2: q_{R1}] \Rightarrow$
$[R_1\ DF\ (R_1\ DF\ R_2): q_{R1}] \Rightarrow [R_1\ IN\ R_2: q_{R1}]$

$[R_2: q_{R2}]\ IN\ [R_1: q_{R1}] \Rightarrow$
$[R_2: q_{R2}]\ DF\ ([R_2: q_{R2}]\ DF\ [R_1: q_{R1}]) \Rightarrow$
$[R_2: q_{R2}]\ DF\ [R_2\ DF\ R_1: q_{R2}] \Rightarrow$
$[R_2\ DF\ (R_2\ DF\ R_1): q_{R2}] \Rightarrow [R_2\ IN\ R_1: q_{R2}]$

Als Ergebnis müßte man theoretisch $[R_1\ IN\ R_2: q_{R1} \wedge q_{R2}]$ erhalten! Die tatsächlich erhaltenen Qualifizierungen sind jedoch nicht falsch, denn die Qualifizierung $q_{R1} \wedge q_{R2}$ impliziert sowohl q_{R1} als auch q_{R2}. Es sind aber Informationen verloren gegangen, so daß die Qualifizierung q_{R1} bzw. q_{R2} mehr Tupel liefert als die Operation $R_1\ IN\ R_2$.

Join:

$[R_1: q_{R1}]\ JN_F\ [R_2: q_{R2}] \Rightarrow [R_1\ JN_F\ R_2: q_{R1} \wedge q_{R2} \wedge F]$

Die Operation Join ist abgeleitet von Selektion und Kartesischem Produkt, und es gilt:

$[R_1: q_{R1}]\ JN_F\ [R_2: q_{R2}] \Rightarrow$
$SL_F([R_1: q_{R1}]\ CP\ [R_2: q_{R2}]) \Rightarrow$
$SL_F([R_1\ CP\ R_2: q_{R1} \wedge q_{R2}]) \Rightarrow$
$[SL_F(R_1\ CP\ R_2): q_{R1} \wedge q_{R2} \wedge F] \Rightarrow$
$[R_1\ JN_F\ R_2: q_{R1} \wedge q_{R2} \wedge F]$

Semijoin

$[R_1\colon q_{R1}]\ SJ_F\ [R_2\colon q_{R2}] \Rightarrow [R_1\ SJ_F\ R_2\colon q_{R1} \wedge q_{R2} \wedge F]$

Die Operation Semijoin ist abgeleitet von Projektion und Join, und es gilt:

$[R_1\colon q_{R1}]\ SJ_F\ [R_2\colon q_{R2}] \Rightarrow$

$PJ_A([R_1\colon q_{R1}]\ JN_F\ [R_2\colon q_{R2}]) \Rightarrow$

$PJ_A([R_1\ JN_F\ R_2\colon q_{R1} \wedge q_{R2} \wedge F]) \Rightarrow$

$[PJ_A([R_1\ JN_F\ R_2)\colon q_{R1} \wedge q_{R2} \wedge F] \Rightarrow$

$[R_1\ SJ_F\ R_2\colon q_{R1} \wedge q_{R2} \wedge F]$ mit $A = Attr(R_1)$

3. Der algebraische HEAD-Optimierer

3.1. Kriterien der algebraischen Optimierung

Die innerhalb der algebraischen Optimierung verwendeten Heuristiken nutzen die Möglichkeit der Minimierung von Ausdrücken, die Mächtigkeit von Zwischenrelationen, den Parallelitätsgrad von Anfragen, den Bearbeitungsaufwand einzelner Operationen sowie die potentielle Unterstützung dieser durch eventuell vorhandene Zugriffspfade als Optimierungskriterium.

Das Problem ist, daß die meisten Transformationsregeln auf der einen Seite eine Optimierung bezüglich bestimmter Kriterien vornehmen, während sie auf der anderen Seite gleichzeitig eine Verschlechterung bezüglich anderer Kriterien bewirken können. Die algebraische Optimierung erfolgt unabhängig von konkreten Kosten, so daß eine exakte Abwägung der Vor- und Nachteile einer Transformation nicht möglich ist. Aus diesem Grunde erfolgt eine Priorisierung der Optimierungskriterien bezüglich ihres zu erwartenden Einflusses auf das Optimierungsergebnis. Vorrangig zu berücksichtigen sind die mögliche Minimierung eines Ausdrucks sowie die Reduzierung der Größe von Zwischenergebnissen. Alle anderen Optimierungskriterien haben sich diesen unterzuordnen.

Die Mächtigkeit von Zwischenrelationen beeinflußt maßgeblich das Ergebnis der Optimierung. Mit der Größe von Zwischenrelationen wächst der Bearbeitungsaufwand für nachfolgende Operationen. Des weiteren können große Zwischenrelationen die kostenaufwendige Auslagerung von Teilen der Zwischenrelation auf den Sekundärspeicher notwendig machen. Weiterhin ist in VDBS der Einfluß von Zwischenrelationen auf den Kommunikationsaufwand zu berücksichtigen, der entsteht, wenn Zwischenrelationen über das Rechnernetz zu anderen Rechnerknoten verschickt werden.

Aus diesen Gründen erfolgt die Festlegung der Operationenfolge vorrangig mit dem Ziel, die bei der Anfragebearbeitung anfallenden Zwischenrelationen so klein wie möglich zu halten.

Minimierende Transformationen ermöglichen das Eliminieren einzelner Operationen sowie ganzer Teilbäume und tragen somit erheblich zur Reduzierung des Bearbeitungs-

aufwandes einer Anfrage bei. Allerdings können solche Transformationen auch das Kriterium der Reduzierung von Zwischenrelationen negativ beeinflussen, z.B. in bestimmten Fällen der Eliminierung redundanter Selektions- und Projektionsoperationen oder der Zusammenfassung von gemeinsamen Teilausdrücken. In diesen Situationen ist von einer Minimierung abzusehen, oder es ist die Minimierung von Parametern abhängig zu machen, die den Gütegrad einer Minimierung definieren.

Im weiteren Verlauf dieses Kapitels sollen die durch den algebraischen HEAD-Optimierer verwendeten Heuristiken für Optimierung und Minimierung vorgestellt werden. Dabei wird unter anderem auf einfache Heuristiken, die sich in [Sch86, CP85,Ull80, Ull88] finden und sich auf einzelne, die Größe von Relationen reduzierenden Operationen der Relationenalgebra beziehen, eingegangen.

Aufgrund der Intensität des Auftretens von Joinoperationen und dessen großen Bearbeitungsaufwandes nimmt die Joinfolgenoptimierung innerhalb der Optimierung eine wichtige Stellung ein und soll deshalb separat behandelt werden. Abschließend wird das Zusammenwirken aller Heuristiken in Form von Optimierungsphasen in einer gemeinsamen Optimiererkomponente beschrieben.

3.2. Anfragepläne als Bäume

Die Ausdrücke der Relationenalgebra lassen sich in Form eines Operatorbaumes darstellen. Im Operatorbaum repräsentieren die Knoten die Operationen und die Kanten beschreiben den Fluß von Zwischenrelationen von einer Operation zur nächsten.

Angenommen, es werde folgende Anfrage an das DBS gestellt:

> Liefer die Namen aller Angestellten, die mehr als 3500 DM verdienen und in einer Abteilung arbeiten, dessen Abteilungsleiter die Personalnummer vier hat!
> $PJ_{Name}(SL_{Lohn>3500}(ANGEST \ \ JN_{Abt=Abt} \ SL_{PNr=4}(ABTEILUNG)))$

Der diesen Ausdruck repräsentierende Operatorbaum ist in Abb. 3 dargestellt.

Der Operatorbaum stellt für die Anfrage einen Bearbeitungsplan dar. Dieser Bearbeitungsplan verfügt aber noch über verschiedene Freiheitsgrade, deren Eliminierung einerseits für die Bearbeitung der Anfrage notwendig ist, aber andererseits

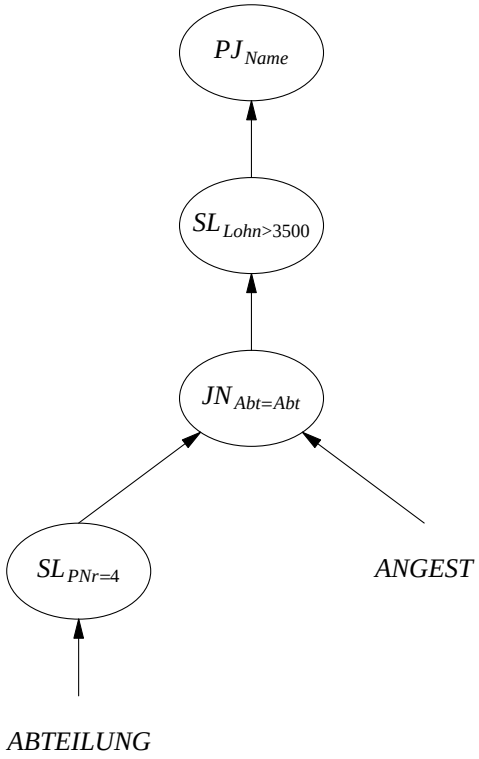


Abb. 3: Initialer Operatorbaum

auch entscheidende Ansätze für die Optimierung bietet. Das Entfernen der verschiedenen Freiheitsgrade erfolgt im Verlauf der Optimierung und beinhaltet das Ersetzen der Relationen durch ihre Fragmente, das Ersetzen der Operationen der Relationenalgebra durch adäquate Algorithmen und das Ergänzen der Baumknoten um Attribute, die unter anderem den Rechnerknoten festlegen, auf dem die Bearbeitung der Operation stattfinden soll. Dies ist besonders für freie Operatoren (Nichtblattknoten) wichtig, aber auch für gebundene Knoten (Blattknoten), im Falle der Existenz von Replikaten. Ausgangspunkt der algebraischen Optimierung in HEAD ist ein globaler Operatorbaum. Dieser Baum bezieht sich auf das globale konzeptuelle Schema der Datenbank, d.h., die Blattknoten des Baumes repräsentieren globale Datenbankrelationen. In einem VDBS können die globalen Relationen in Fragmente zerlegt sein. Die

Transformation eines globalen Operatorbaumes in einen Baum, der sich auf das Fragmentationsschema bezieht, wird als Normalisierung bezeichnet. Ein Baum mit Fragmenten als Blattknoten wird fragmentierter Baum genannt. Die Anwendung algebraischer Transformationsregeln auf Algebraausdrücke läßt sich am Operatorbaum in Form von Baumtransformationen darstellen. Dabei können diese innerhalb der algebraischen Optimierung sowohl am globalen als auch am fragmentierten Baum erfolgen.

3.3. Normalisierung des Algebraausdrucks

Die Überführung einer Anfrage basierend auf dem globalen Schema in eine Anfrage basierend auf dem Fragmentationsschema wird als Normalisierung bezeichnet. Bei horizontal fragmentierten Relationen repräsentiert ein Fragment eine Teilmenge der Tupel der globalen Relation. Die in den Fragmenten enthaltenen Tupel können mittels der Operation Union wieder zur globalen Relation zusammengefaßt werden. Bei der vertikalen Fragmentierung entstehen die Fragmente durch Zerlegung eines jeden Tupels in Teilinformationen, d.h., die Attribute eines Fragments sind eine Teilmenge der Attribute der globalen Relation. Vertikale Fragmente werden mit Hilfe der Joinoperation wieder zur globalen Relation zusammengesetzt.

Ausdrücke des globalen Schemas basieren auf globalen Relationen, Ausdrücke des Fragmentationsschemas hingegen auf Fragmente, d.h., die Blattknoten des Operatorbaumes stellen Fragmente und keine (kompletten) Relationen dar.

Ein Ausdruck wird normalisiert, indem im globalen Operatorbaum jede globale Relation durch die Namen ihrer Fragmente und einen algebraischen Ausdruck ersetzt wird. Der algebraische Ausdruck entspricht der Umkehrung des Fragmentationsschemas und gibt an, wie die globale Relation aus ihren Fragmenten wiedergewonnen wird. Unter der Voraussetzung, daß alle globalen Relationen fragmentiert sind, repräsentieren nach der Normalisierung alle Blattknoten Fragmente. Der im Ergebnis der Normalisierung vorliegende Ausdruck wird als normalisierter Ausdruck bezeichnet.

Der durch den Operatorbaum aus Abb. 3 beschriebene Ausdruck ist in Abb. 4 in seinen normalisierten Ausdruck überführt worden, wobei gilt, daß sowohl die Relation

ANGEST als auch die Relation ABTEILUNG horizontal nach dem Attribut Abt in zwei Fragmente ($Abt \leq 10$, $Abt > 10$) fragmentiert sind.

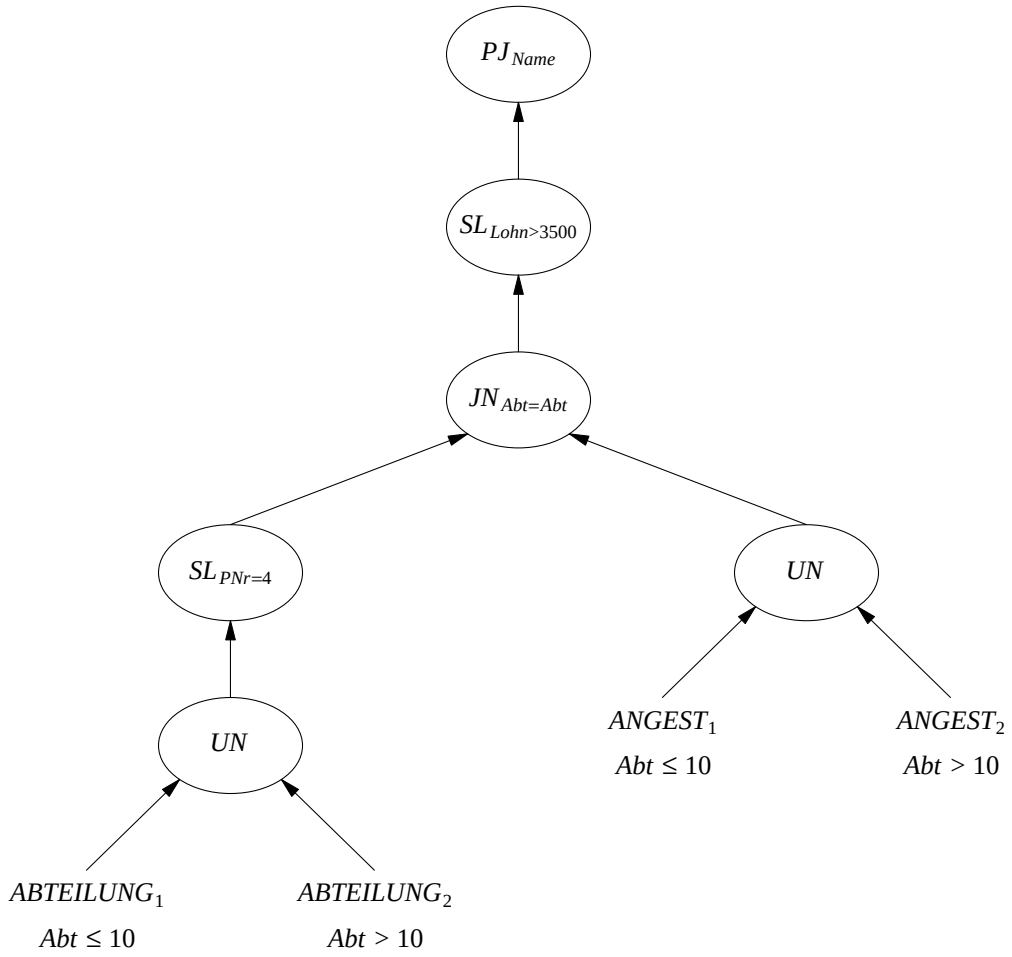


Abb. 4: Normalisierter Operatorbaum

Bei der Abarbeitung einer Anfrage entsprechend ihres normalisierten Ausdrucks werden zunächst alle Fragmente einer Relation in Form einer Zwischenrelation wieder zu ihrer globalen Relation zusammengesetzt und dann entsprechend der globalen Relation weiterverarbeitet. Dadurch werden aber potentielle Vorteile, die sich aus der Fragmentierung ergeben, ignoriert, und der normalisierte Ausdruck stellt somit in den meisten Fällen einen schlechten Plan zur Bearbeitung der Anfrage dar.

Die Normalisierung kann losgelöst vom Optimierungsprozeß betrachtet werden, wenn sie generell zu dessen Beginn vorgenommen wird. In dieser Arbeit soll die Normalisierung jedoch als Bestandteil der Optimierung betrachtet werden, da der Zeitpunkt für die Normalisierung Einfluß auf den Optimierungsprozeß hat. So könnte beispielsweise eine Teiloptimierung des globalen Ausdrucks nützlich sein. Auf den Zeitpunkt für die Normalisierung soll am Ende des Kapitels 3 eingegangen werden.

3.4. Minimierung

3.4.1. Minimierende Transformationen

Innerhalb der im Anhang 1 aufgeführten Transformationsregeln existiert eine Gruppe von Regeln, die eine Minimierung von Ausdrücken bewirkt. Im Ergebnis dieser Transformationen entstehen vereinfachte Ausdrücke, die auf einer Eliminierung von Operationen und ganzen Teilbäumen basieren. Es gelte deshalb die Heuristik:

H1 Alle im Anhang 1 als minimierende Regeln aufgeführten Transformationen sind durch den Optimierer anzuwenden.

3.4.2. Zusammenfassung gemeinsamer Teilausdrücke

Die Aufdeckung gemeinsamer Teilausdrücke innerhalb der Optimierung kann ein Beitrag zur Senkung der Bearbeitungskosten sein, denn sie vermeidet die Mehrfachberechnung von gleichen Teilausdrücken. Es ist verstärkt mit gemeinsamen Teilausdrücken zu rechnen, wenn die Anfrage über Sichten formuliert ist. Dies ist der Regelfall, denn das Sichtkonzept wird genutzt, um unterschiedliche Zugriffsrechte zu realisieren. Anfragen des externen Schemas lösen beim Überführen auf das globale konzeptuelle Schema Sichten auf, in dessen Ergebnis gleiche Teilausdrücke innerhalb des Gesamtausdrucks entstehen können. Des weiteren können gleiche Teilausdrücke im Ergebnis der Normalisierung entstehen.

Die Suche und die Zusammenfassung von gemeinsamen Teilausdrücken wird als Faktorisierung bezeichnet.

Bei der Faktorisierung sind nur diejenigen gleichen Teilausdrücke von Interesse, die identische Ergebnisse liefern. Aus diesem Grunde werden nur Teilbäume als gleich erkannt, deren Blattknoten auch Blattknoten im Gesamtausdruck sind. Sonstige lokal gleiche Teilbäume werden nicht berücksichtigt.

Idealerweise werden erkannte gemeinsame Teilausdrücke unter Anwendung von minimierenden Transformationsregeln zusammengefaßt. Ein derart minimierter Baum vermeidet die redundante Berechnung gleicher Teilausdrücke, ohne dabei den Umfang der Optimierungsmöglichkeiten zu beschränken. Unter Anwendung der Regel

$$R \; DF \; SL_F(R) \Rightarrow SL_{\neg F}(R)$$

können im Operatorbaum aus Abb. 6 zwei gleiche Teilbäume zusammengefaßt und zu dem Baum aus Abb. 5 minimiert werden.

In HEAD wird versucht, erkannte gemeinsame Teilausdrücke unter Anwendung von minimierenden Transformationen zusammenzufassen. Im Mittelpunkt stehen hierbei minimierende Transformationen für binäre Operationen, denn nur diese ermöglichen eine Zusammenfassung von Teilausdrücken. Im Ergebnis einer Zusammenfassung können weitere Minimierungsmöglichkeiten entstehen.

Im Anschluß an die Minimierung wird der Baum erneut auf gleiche Teilausdrücke untersucht, und der Vorgang wiederholt sich. Konnte aber stattdessen keine Minimierung des Baumes vorgenommen werden, erfolgt eine Markierung gleicher Teilausdrücke. Der Algorithmus aus Abb. 7 beschreibt dieses Vorgehen.

Die Markierung gleicher Teilausdrücke soll garantieren, daß die betreffenden Teilausdrücke nur einmal berechnet werden. Das resultierende Zwischenergebnis wird hierbei dupliziert und an die entsprechenden Vorgängerknoten im Baum verteilt. Um die *Zerstörung* gemeinsamer Teilausdrücke durch nachfolgende optimierende Transformationen zu vermeiden, müssen diese nicht nur markiert, sondern auch *geschützt* werden. Dieser Philosophie entspricht die Einführung des Operatorknotens *DUP*. Gleiche Teilausdrücke werden mittels des DUP-Operators zusammengefaßt, und die durch den Ausdruck gelieferte Zwischenrelation wird dann durch diesen DUP-Operator dupliziert und an die verschiedenen Vorgängerknoten weitergeleitet. Durch die Einführung des DUP-Operators wird aus dem Anfragebaum jedoch ein ge-

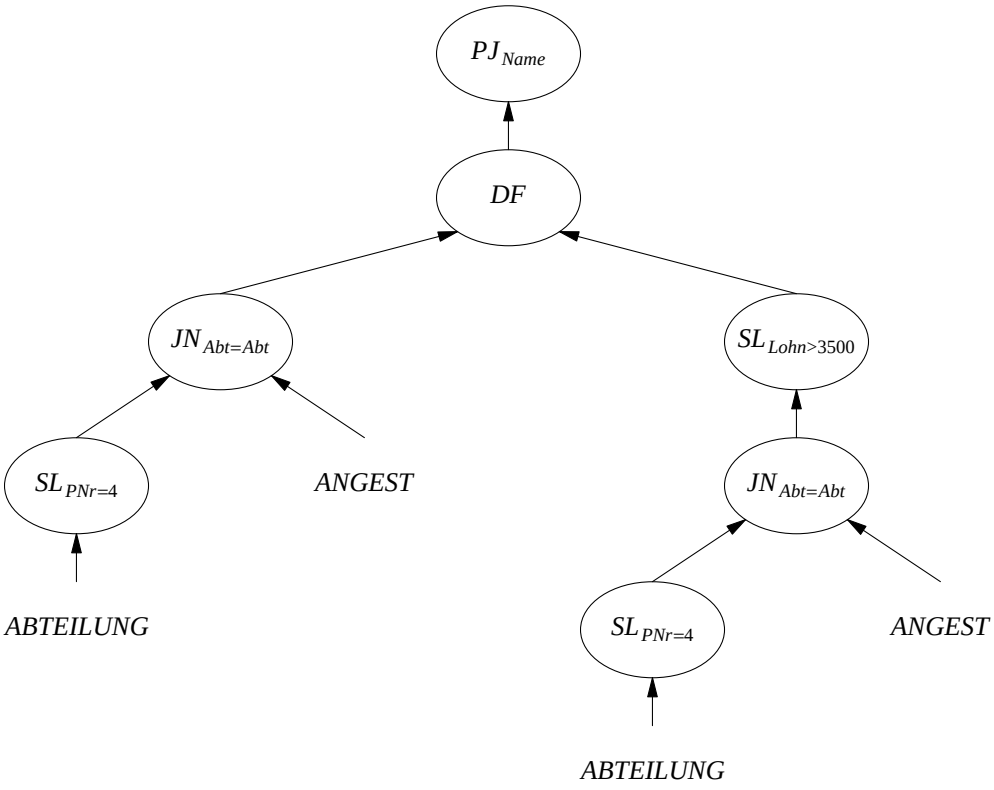


Abb. 5: Operatorbaum mit identischen Teilbäumen

richteter azyklischer Graph (GAG). Die Optimierung eines GAG ist aber wesentlich komplizierter [ASU88]. Es wird deshalb von Aho, Sethi und Ullman vorgeschlagen, den GAG wie folgt in eine Menge von Bäumen zu zerlegen:

> Es ist für die Wurzel und jeden geteilten Knoten *n* (DUP-Knoten) der maximale Teilbaum mit *n* als Wurzel und keinen sonstigen geteilten Knoten, mit Ausnahme der Blattknoten, zu ermitteln. Die auf diese Weise erhaltenen Teilbäume werden separat betrachtet und optimiert.

Unter Verzicht einer vorherigen Minimierung ist der Ausgangsbaum aus Abb. 5 in Abb. 8 durch Einführung des DUP-Operators in einen GAG überführt worden.

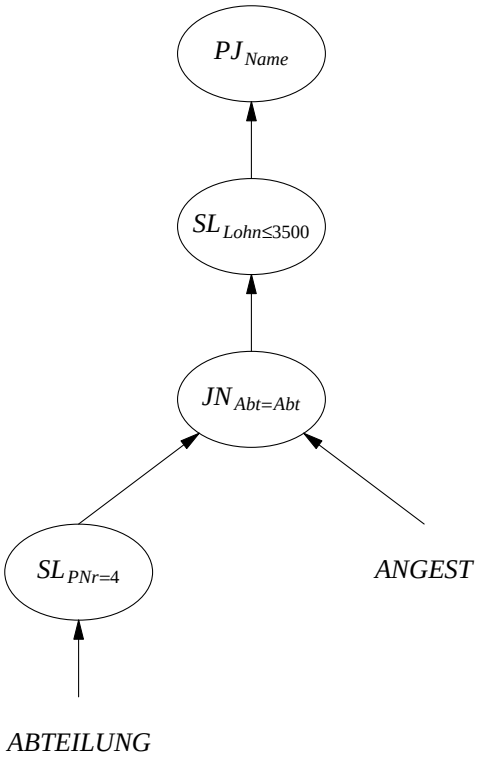


Abb. 6: Minimierter Baum

Wie schon erwähnt, treten gemeinsame Teilausdrücke verstärkt nach der Auflösung von Sichten und nach der Normalisierung der Anfrage auf. Da anschließende optimierende Baummanipulationen gleiche Teilausdrücke zerstören können, scheint eine Suche nach ihnen zu diesem Zeitpunkt am erfolgversprechendsten. Durch die Einführung des DUP-Operators wird vermieden, daß erkannte gemeinsame Teilausdrücke, die nicht durch Minimierungen zusammengefaßt werden konnten, redundant bearbeitet werden. Leider beschränkt der DUP-Operator die Mächtigkeit der Optimierung, da diese nur noch lokal in den entsprechenden Teilbäumen erfolgt. Im Gegensatz zum minimierten Baum aus Abb. 6, wo die Projektion PJ_{Name} und die Selektion $SL_{Lohn\leq 3500}$ noch vor den Join verschoben werden können, ist es nicht möglich, die einzelnen Teilbäume des GAG aus Abb. 8 weiter zu optimieren.

```
HandhabungGemeinsamerTeilausdrücke()
{
        do {
                Suche nach gleichen Teilausdrücken;
                Anwendung minimierender Transformationen;
        }
        while (Ausdruck wurde minimiert)

        Markierung gleicher Teilausdrücke
}
```

Abb. 7: Faktorisierung

Es gilt einen Kompromiß zwischen der Einschränkung der Optimierung einerseits und der redundanten Berechnung gemeinsamer Teilausdrücke andererseits zu finden. Dieser Kompromiß wird in HEAD durch die Bestimmung des optimalen Zeitpunktes für die Faktorisierung und durch nachfolgend aufgeführte Maßnahmen realisiert.

Das Zusammenfassen soll auf entsprechend große Teilausdrücke beschränkt werden. Als Kriterium kann die maximale Tiefe des gemeinsamen Teilbaumes oder die Anzahl der darin enthaltenen Knoten benutzt werden. Da ein VDBS die Ausnutzung der inhärenten Parallelität erlaubt, ist die Baumtiefe das bessere Kriterium. Mit der Größe der Teilausdrücke überwiegt der Nutzen, der sich aus der Reduzierung des Bearbeitungsaufwandes ergibt, gegenüber den durch die Einschränkung der Optimierung entstehenden Nachteilen. Bei der Ermittlung der Größe eines Teilausdrucks muß berücksichtigt werden, ob der Algorithmus auf einen globalen oder auf einen fragmentierten Ausdruck angewandt wird. Wird der Algorithmus auf einen globalen Ausdruck angewandt, muß bei der Ermittlung der Größe gemeinsamer Teilausdrücke die Größe des Fragmentierungsausdruckes einbezogen werden. Der Fragmentierungsausdruck entsteht, wenn die globale Relation durch ihre Fragmente ersetzt wird. Die Größe dieses Ausdrucks ergibt sich aus der Anzahl der Fragmente, wobei festgelegt sei, daß bei Verwendung der Baumtiefe als Größenkriterium der Fragmentierungsausdruck als links- bzw. rechtstiefer Baum realisiert wird.

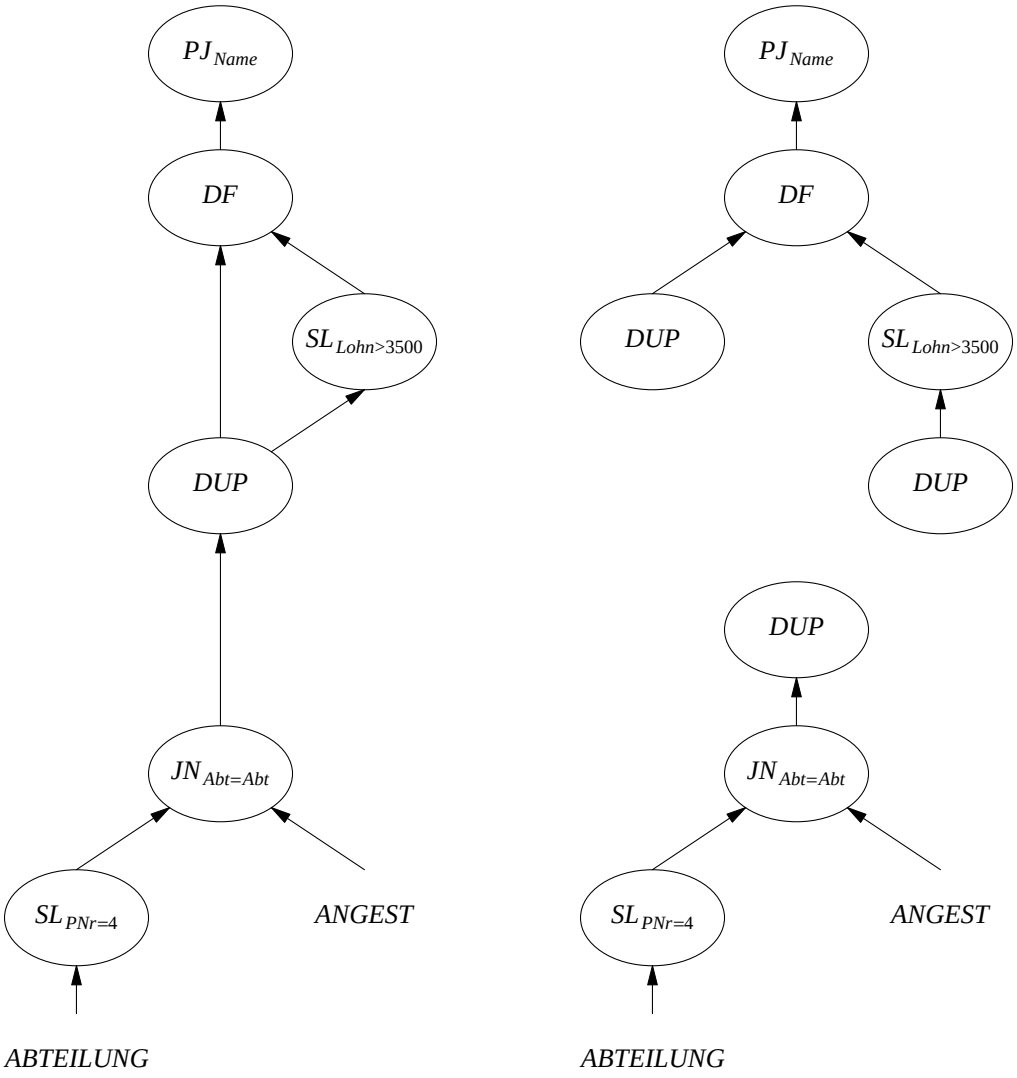


Abb. 8: Erzeugter GAG und seine Zerlegung in Teilbäume

Einen weiteren Ansatzpunkt bietet das Zulassen bzw. Nichtzulassen von DUP-Knoten innerhalb eines durch einen anderen DUP-Knoten erzeugten Teilbaumes. Ein Zulassen solcher Knoten würde den betreffenden Teilbaum zersplittern und eventuell die Optimierung derart lokalisieren, daß diese nicht mehr möglich ist. Daraus resultiert die Notwendigkeit, entweder das Zulassen von DUP-Knoten in anderen DUP-Teilbäumen gänzlich auszuschließen und damit nur die maximalen Teilbäume zu betrachten oder zu

gewährleisten, daß die durch diese Knoten erhaltenen DUP-Teilbäume befriedigend groß sind. Diese Gewährleistung erfordert sehr große Teilbäume und dürfte demzufolge nur in Ausnahmesituationen realisierbar sein. Der algebraische Optimierer von HEAD läßt keine DUP-Knoten innerhalb anderer DUP-Teilbäume zu.

Des weiteren wird im HEAD-Optimierer angestrebt, den gemeinsamen Teilausdruck im Verlaufe der Optimierung durch Aufnahme weiterer Knoten zu vergrößern. Dabei soll die Suche nach identischen Knoten nicht nur auf die direkten Vorgängerknoten des DUP-Operators beschränkt sein, sondern es werden alle unären Vorgängerknoten (Projektions- und Selektionsoperatoren) bis zum ersten binären Knoten auf Identität untersucht.

Die unären Knoten Projektion und Selektion besitzen ein großes Optimierungspotential. Sie reduzieren Zwischenrelationen und sind außerdem im Vergleich zu binären Knoten sehr variabel im Baum verschiebbar. Aus diesem Grund versucht der algebraische Optimierer erkannte identische unäre Knoten derart zu verschieben, daß sie zu direkten Vorgängerknoten des DUP-Operators werden, wo sie zusammengefaßt und in den gemeinsamen Teilausdruck mit aufgenommen werden können. Im Ergebnis dessen können sich für diesen Teilbaum neue Optimierungsmöglichkeiten ergeben, die es erlauben, den unären Knoten weiter in Richtung Blattknoten zu verschieben.

Angenommen, ein DUP-Knoten habe *n* direkte Vorgängerknoten. Beginnend beim DUP-Knoten werden für alle seine *n* Vorgängerpfade sämtliche Projektions- und Selektionsoperatoren bis zum ersten binären Knoten nach der Identität in Gruppen von gleichen Knoten geordnet. Ein Vorgängerpfad eines DUP-Knoten beinhaltet hierbei alle Knoten, die auf dem Pfad vom DUP-Knoten zum Wurzelknoten liegen. Können mehrere Gruppen identischer Knoten aufgedeckt werden, wird zuerst versucht, die Knoten der mächtigsten Gruppe zum DUP-Operator zu verschieben. Ist dies nicht möglich, wird die nächste Gruppe betrachtet usw. bis entweder die Verschiebeoperation für mindestens zwei Knoten erfolgreich war oder keine weitere Gruppe existiert und somit zum nächsten DUP-Operator übergegangen werden muß. War die Verschiebeoperation hingegen erfolgreich, und betrifft die Identität alle Pfade, können alle direkten Vorgängerknoten zusammengefaßt und in den gemeinsamen Teilausdruck aufgenommen werden. Bezieht sich die Identität nur auf einzelne Vorgängerknoten, werden diese zusammengefaßt, und für die betreffenden Pfade wird ein DUP-Knoten erzeugt. Da DUP-

Knoten innerhalb anderer Knoten nicht erlaubt sind, wird zuvor der alte DUP-Operator eliminiert und der zugehörige Teilbaum entsprechend der Anzahl der Vorgängerknoten dupliziert.

Zur Unterstützung dieser als Heuristik zu realisierenden Erweiterung der DUP-Teilbäume ist es vorteilhaft, im voraus die Heuristiken "Verschieben von Selektion und Projektion zu den Blättern" zu realisieren.

Abschließend lassen sich für diesen Abschnitt folgende Heuristiken formulieren:

H2 Erkannte gemeinsame Teilausdrücke sind unter Anwendung minimierender Transformationsregeln zusammenzufassen. Ist dies nicht möglich, ist eine Zusammenfassung mittels des DUP-Operators vorzunehmen. Die Zusammenfassung mittels des DUP-Operators ist hierbei von der Größe der Teilausdrücke abhängig zu machen.

H3 Die Aufnahme weiterer Knoten in DUP-Bäume ist anzustreben. Hierbei sind besonders unäre Knoten von Interesse. Eine Aufnahme dieser ist entsprechend dem beschriebenen Verfahren vorzunehmen.

3.4.3. Minimierung auf Basis der Algebra AQUAREL

Die Verwendung der Algebra AQUAREL erlaubt im Gegensatz zur Relationenalgebra die Berücksichtigung der Fragmentierungsbedingung horizontal fragmentierter Relationen als eine gemeinsame Eigenschaft aller Tupel eines Fragments. Die Qualifizierung horizontaler Fragmente repräsentiert diese Eigenschaft. Steht diese Eigenschaft im Widerspruch zu anderen durch Operatoren der Relationenalgebra erzwungenen Eigenschaften, kann der betreffende Teilbaum durch die leere Relation ersetzt werden, denn er würde unabhängig von der Tupelbelegung immer die leere Menge liefern. Die Verwendung der Algebra AQUAREL unterstützt die Eliminierung von Fragmenten und Algebraoperationen.

Ein Teilausdruck liefert keine Tupel, wenn dessen Qualifizierung inkonsistent ist, d.h., wenn ein Widerspruch zwischen den einzelnen durch die Qualifizierung geforderten Eigenschaften existiert. Der Teilausdruck ist dann für die Bearbeitung der Anfrage

unrelevant und kann somit mittels minimierender Transformationen bereits zur Übersetzungszeit eliminiert werden. Wie bereits bei der Behandlung des Umbenennungsoperators erwähnt wurde, wird in dieser Arbeit das Auftreten von Domain mismatches und somit auch das Auftreten von Wertebereichskonflikten in der Qualifizierung ausgeschlossen. Damit wird das Auftreten einer Qualifizierung, wie $a < 20 \wedge a' > 19$, wobei a vom Typ Integer und a' vom Typ Real und a==a' ist, und das daraus folgende Problem ihrer Bewertung vermieden.

Der HEAD-Optimierer soll nicht für jede Zwischenrelation eine Bewertung der Qualifizierung vornehmen. Von Interesse sind lediglich diejenigen Zwischenrelationen, die im Ergebnis einer Algebraoperation entstehen, die neue Tupeleigenschaften erzwingt (Join, Selektion). Durch den Optimierer sind nachfolgende Heuristiken umzusetzen:

H4 Die Qualifizierung des Ergebnisses von Selektionsoperationen ist zu bewerten und durch die leere Relation zu ersetzen, falls die Qualifizierung inkonsistent ist.

H5 Die Qualifizierung von Joinoperanden ist zu bewerten und der Unterbaum, bestehend aus dem Join und seinen Operanden, ist durch die leere Relation zu ersetzen, falls die Qualifizierung des Joinergebnisses inkonsistent ist.

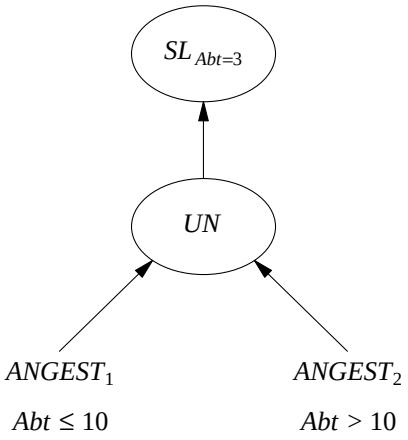
Bsp.:

Eine Anfrage soll alle Angestellten liefern, die in der Abteilung drei arbeiten. In Abb. 9(a) ist der entsprechende normalisierte Operatorbaum dargestellt. Es soll hierbei gelten, daß sich die Relation ANGEST in zwei horizontale Fragmente aufteilt, entsprechend der Fragmentierungsbedingung $Abt \leq 10$ bzw. $Abt > 10$.

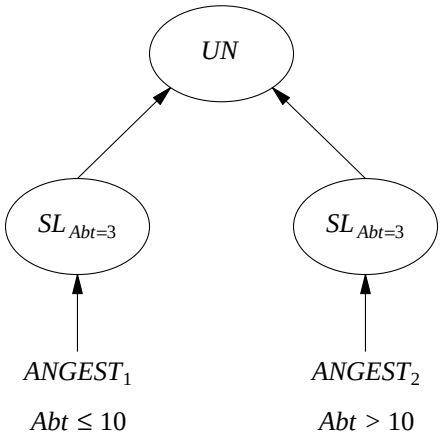
Durch Verschieben der Selektion vor die Union erhält man den Baum aus Abb. 9(b). Für beide Selektionsoperationen wird die Qualifizierung ermittelt und bewertet, wobei festgestellt werden kann, daß die Qualifizierung des rechten Teilbaumes inkonsistent ist.

$$SL_{Abt=3}[ANGEST_2: Abt > 10] \Rightarrow [SL_{Abt=3}(ANGEST_2): Abt = 3 \wedge Abt > 10]$$

Der rechte Teilausdruck wird durch die leere Menge ersetzt, und anschließend wird der Gesamtausdruck unter Verwendung der Transformationsregel $R\ UN\ \varnothing \Rightarrow R$ weiter minimiert, siehe Abb. 10.

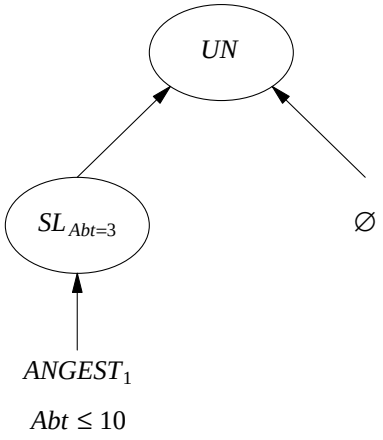


(a) Normalisierter Baum

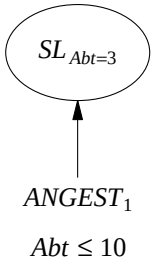


(b) Optimierter Baum

Abb. 9: Operatorbaum vor der Minimierung



(a) Eliminierung von $ANGEST_2$



(b) Minimierter Ergebnisbaum

Abb. 10: Stufen der Minimierung

Die Aufdeckung von Inkonsistenzen beruht darauf, daß Join- und Selektionsoperationen auf Fragmente und nicht auf komplette Relationen angewendet werden. Deshalb ist zur Unterstützung der Heuristik H4 und H5 die Anwendung von Join- und Selektionsoperationen auf Fragmente anzustreben.

3.5. Optimierung

3.5.1. Optimierung auf Basis der Selektion

Selektionen nehmen in Abhängigkeit von der Selektionsbedingung eine Reduzierung der eingehenden Relation vor, indem nichtqualifizierte Tupel eliminiert werden. Diese Reduzierung bedingt eine Senkung des Kommunikations- und des Bearbeitungsaufwandes der nachfolgenden Operationen. Außerdem erlauben Selektionen über physischen Datenbankrelationen die Ausnutzung von eventuell vorhandenen Zugriffspfaden. Daraus resultiert folgende Heuristik:

H6 Transformationen, die eine Verschiebung von Selektionsoperationen in Richtung Blattknoten vornehmen, sind durch den Optimierer anzuwenden.

Förderlich für die Umsetzung dieser Heuristik ist die Existenz einer Vielzahl von unterstützenden Transformationsregeln. Zu diesen gehören alle Regeln, die Selektionen in Richtung Blattknoten verschieben und im Anhang 1 im Abschnitt zur Veränderung der Reihenfolge unärer Operationen und im Abschnitt zur Verschiebung von unären Operationen vor eine binäre Operation aufgeführt sind. Bezüglich folgender Regeln:

$$SL_F(R_1\ IN\ R_2) \Rightarrow SL_F(R_1)\ IN\ SL_F(R_2)$$
$$SL_F(R_1\ IN\ R_2) \Rightarrow SL_F(R_1)\ IN\ R_2 \qquad \text{und}$$

$$SL_F(R_1\ DF\ R_2) \Rightarrow SL_F(R_1)\ DF\ SL_F(R_2)$$
$$SL_F(R_1\ DF\ R_2) \Rightarrow SL_F(R_1)\ DF\ R_2\ ,$$

ist jeweils nur die erste zur Umsetzung der Heuristik zu verwenden. In beiden Fällen stellt die erste Regel eine bessere Alternative dar, denn sie ermöglicht auch die Reduzierung der Größe des zweiten Operanden, wobei dessen Selektionsoperation

parallel zur anderen Selektion bearbeitet werden kann.

Die Zahl der auf eine Selektionsoperation anwendbaren Transformationsregeln steigt mit abnehmender Attributmenge der in der Selektionsbedingung verwendeten Attribute.

Aus diesem Grund ist vor der Umsetzung obiger Heuristik folgende Heuristik durch den Optimierer zu realisieren:

H7 Unter Anwendung der Transformationsregel $SL_{F1 \wedge F2}(R) \Rightarrow SL_{F1}(SL_{F2}(R))$ ist eine Selektionsoperation in eine Folge von Selektionsoperationen zu zerlegen.

Die Realisierung dieser Heuristik und der Umstand, daß keine der durch den Optimierer angewandten Regeln die Entstehung einer zusammengesetzten Selektion verursacht, erlauben innerhalb der ersten Heuristik H6 die Nichtberücksichtigung aller Regeln, die auf einer zusammengesetzten Selektionsoperation beruhen, ohne daß dadurch die Optimierungsmöglichkeiten eingeschränkt werden.

3.5.2. Optimierung auf Basis der Projektion

Projektionen nehmen auf der Basis ihrer Projektionsliste eine Reduzierung der Tupelgröße vor, indem einzelne Attribute mit ihren Werten ausgeblendet werden. Transformationen, die Projektionen zu den Blättern des Baumes verschieben, verringern die Größe der eingehenden Relation und demzufolge den Kommunikations- und den Bearbeitungsaufwand für die nachfolgenden Operationen, jedoch nicht in dem Maße wie die Selektion. Es gelte folgende Heuristik:

H8 Transformationen, die eine Verschiebung von Projektionsoperationen in Richtung Blattknoten vornehmen, sind durch den Optimierer anzuwenden.

Zur Realisierung dieser Heuristik stehen dem Optimierer eine Vielzahl von Transformationsregeln zur Verfügung. Zu diesen gehören alle Regeln, die Projektionen in Richtung Blattknoten verschieben und im Anhang 1 im Abschnitt zur Veränderung der Reihenfolge unärer Operationen und im Abschnitt zur Verschiebung von unären Operationen vor eine binäre Operation aufgeführt sind.

Unter diesen Regeln finden sich auch drei, die die redundante Erzeugung einer Projektion zur Folge haben. Diese betreffen die Verschiebung der Projektion vor die Selektion, den Join bzw. den Semijoin. Diese Verschiebungen ermöglichen die Reduzierung von Relationen, verursachen aber allerdings auch durch die redundante Berechnung der Projektion zusätzliche Bearbeitungskosten. Diese Kosten können jedoch vernachlässigt werden, aufgrund der effektiven Implementierung der Projektionsoperation. Die Bearbeitung der Projektion in HE$_A$D erfolgt grundsätzlich mittels Pipelining. Außerdem wird eine Auftrennung der relationenalgebraischen Projektion in Projektion und in Duplikateneliminierung vorgenommen. Eine Duplikateneliminierung erfolgt hierbei nicht implizit mit jeder Projektion, sondern nur sofern erforderlich.

Die Selektionsoperation reduziert in stärkerem Maße als die Projektion die Größe von Relationen, so daß für den im Ergebnis der Optimierung erhaltenen Baum die folgende Heuristik gelten soll:

H9 Selektionen sind vor Projektionen zu verschieben.

3.5.3. Umwandlung von Operationen

Es kann sinnvoll sein, Operationen innerhalb des Baumes zu einer neuen Operation zusammenzufassen, wenn dadurch die resultierende Zwischenrelation reduziert werden kann oder sich der Bearbeitungsaufwand durch diese Transformation für die betrachteten Operationen vermindert. Eine solche Transformation ist das Zusammenfassen von spezifischen Selektionsoperationen mit nachfolgendem Kartesischem Produkt zu einem Join. Ein Kartesisches Produkt, gefolgt von einer Selektion, kann durch einen Join substituiert werden, falls für die Operation

$$SL_F(R_1\ CP\ R_2) \Rightarrow R_1\ JN_F\ R_2$$

die Bedingung F aus einer und-Verknüpfung von Vergleichen von Attributen aus R_1 und R_2 besteht. Durch diese Transformation wird nicht nur die Zwischenrelation gegenüber dem Kartesischen Produkt reduziert, sondern die gesamte Selektionsoperation kann eingespart werden. Der Bearbeitungsaufwand sinkt. Des weiteren kann gegenüber dem Kartesischen Produkt die durch den Join zu bearbeitende Tupelanzahl reduziert

werden, sofern mindestens eine der Relationen R_1 oder R_2 nach dem Joinattribut sortiert bzw. über dem Joinattribut ein Zugriffspfad definiert ist. Es gelte deshalb die Heuristik:

H10 Kartesisches Produkt und Selektion sind zu einem Join zusammenzufassen.

Eine weitere optimierende Umwandlung von Operationen ist die Zusammenfassung von zwei aufeinanderfolgenden Differenzen zur Operation Intersection:

$$R_1 \ DF \ (R_2 \ DF \ R_3) \Rightarrow R_1 \ IN \ R_2$$

Ohne zusätzliche Bearbeitungskosten zu verursachen, werden auf diese Weise zwei binäre Operationen zu einer Operation zusammengefaßt. Es gelte die Heuristik:

H11 Zwei aufeinanderfolgende Differenzen sind zu der Operation Intersection zusammenzufassen.

3.5.4. Möglichkeiten paralleler Anfragebearbeitung

In einem VDBS bieten sich vielfältige Möglichkeiten zur Parallelisierung der Anfragebearbeitung und damit zur Verkürzung der Antwortzeit. Möglichkeiten der Parallelisierung einer Anfrage werden in HEAD innerhalb der Optimierungsphase durch den Optimierer und zum Zeitpunkt der Bearbeitung einer Anfrage durch eine dynamische Lastbalancierung [Lin94] geprüft und umgesetzt. In HEAD werden folgende vier Formen der Parallelität betrachtet ([FLM93]):

- horizontale Parallelität
- Bushy-Tree-Parallelität
- Pipeline-Parallelität
- Intra-Operator-Parallelität

Die Existenz mehrerer zur Bearbeitung einer Anfrage zur Verfügung stehender Rechnerknoten, die physische Verteilung der Datenbank über diese Rechnerknoten sowie die Existenz von Fragmenten und Replikaten erzwingt und ermöglicht die Aufteilung eines

Anfrageplanes in mehrere parallel bearbeitbare Teilanfragepläne, die den Zugriff auf lokale Datenbanken und die Abarbeitung auf verschiedenen Rechnerknoten beschreiben. Diese Form der Parallelität wird horizontale Parallelität genannt. Als ein Bestandteil der horizontalen Parallelität ist die Bushy-Tree-Parallelität zu betrachten. Sie beruht auf äquivalenten Umformungen des Operatorbaumes, welche mit dem Ziel erfolgen, die Baumhöhe zu reduzieren und damit den horizontalen Parallelitätsgrad zu erhöhen. Ein typisches Beispiel ist die Manipulation von Join-Sequenzen.

Durch die Verwendung einer Datenflußsteuerung und aufeinander abgestimmter Algorithmen wird in HEAD eine parallele Ausführung logisch aufeinanderfolgender Operationen möglich. Tupel einer Relation können somit bereits vor der kompletten Abarbeitung dieser Relation durch nachfolgende Operationen bearbeitet werden. Diese Art der Parallelität wird als vertikale Parallelität oder auch als Pipelining bezeichnet.

Einige Operationen, wie z.B. der Join und die Sortierung, stoppen den Datenfluß aus Gründen der Komplexität bzw. aufgrund der verwendeten Algorithmen. Dies führt zu einer Einschränkung der vertikalen Parallelität. Durch die Partitionierung des Datenstroms können diese Teilströme durch voneinander unabhängige Operatoren gleichzeitig bearbeitet werden. Diese Art der Parallelität wird Intra-Operator-Parallelität oder Datenparallelität genannt und bewirkt eine Erhöhung der vertikalen und der horizontalen Parallelität.

die vertikale Parallelität und die Intra-Operator-Parallelität basieren auf den gewählten Implementierungen der relationenalgebraischen Operationen. Deren Realisierung in HEAD ist die Arbeit von Kruse [Kru94] gewidmet. Innerhalb der horizontalen Parallelität beschäftigt sich die Bushy-Tree-Parallelität mit der inhärenten Parallelität eines Operatorbaumes. In Abhängigkeit von dem Datenbankschema kann sich der betrachtete Operatorbaum auf komplette logische Relationen (globales Schema), auf einzelne Fragmente (Fragmentationsschema) oder auf die physischen Relationen der Datenbank (Allokationsschema) beziehen. Die Parallelisierungsmöglichkeiten der Anfrage nehmen hierbei mit abnehmendem Abstraktionsgrad (globales Schema -> Allokationsschema) zu.

Innerhalb der algebraischen Optimierung ist nur die Betrachtung der Bushy-Tree-Parallelität möglich. Das Ziel ist hierbei die Generierung eines Baumes mit maximalem

horizontalem Parallelitätsgrad. Von allen möglichen Baumstrukturen unterstützen bushy-trees die horizontale Parallelität optimal. Die im Anhang 1 angegebenen Transformationsregeln zum Splitten von binären Operationen erlauben die Generierung eines solchen Baumes. Jedoch ist die Erhöhung der horizontalen Parallelität einer Anfrage kein ausreichendes Kriterium für die Formulierung einer Heuristik, denn die Erhöhung der horizontalen Parallelität ist verbunden mit zusätzlichen Operationen und damit mit einer Steigerung der Systemlast, die wiederum eine Erhöhung der Antwortzeit verursacht [LIN94]. Diesbezügliche Entscheidungen sind deshalb nur unter Einbezug konkreter Kosten und somit außerhalb der algebraischen Optimierung sinnvoll.

3.6. Minimierungs- und Optimierungsmöglichkeiten von Joinoperationen

3.6.1. Bearbeitungsfolge von Join und Kartesischem Produkt

Bei einem Kartesischem Produkt wird jedes Tupel der einen Relation mit jedem Tupel der anderen Relation zu einem neuen Tupel verbunden. Dies läßt die Ergebnisrelation sehr groß werden. Bei einem Join hingegen erfolgt dieser Verbund in Abhängigkeit von der Joinbedingung und nur im schlechtesten Fall ist ein Verbund aller Tupel notwendig. Außerdem kann die durch den Join zu bearbeitende Tupelmenge gegenüber dem Kartesischem Produkt reduziert werden, falls mindestens eine der beiden Eingangsrelationen nach dem Joinattribut sortiert bzw. über dem Joinattribut ein Zugriffspfad definiert ist. Aus diesem Grund gelte folgende Heuristik:

H12 Durch den Optimierer sind nachfolgend angegebene Transformationsregeln, die die Verschiebung eines Joins vor ein Kartesisches Produkt ermöglichen, anzuwenden. Die Transformationsregeln gestatten dabei nicht nur die Betrachtung der Operationenfolge Join und Kartesisches Produkt, sondern es sollen auch diejenigen Operationenfolgen betrachtet werden, die zusätzlich die unären Operationen Projektion und Selektion enthalten.

Zur Durchsetzung der Heuristik sind durch den Optimierer folgende Transformationen anzuwenden:

$$(R_1\ CP\ R_2)\ JN_F\ R_3 \Rightarrow R_2\ CP\ (R_1\ JN_F\ R_3)$$

$$R_1\ CP\ R_2 \Rightarrow R_2\ CP\ R_1$$

$$R_1\ JN_F\ R_2 \Rightarrow R_2\ JN_F\ R_1$$

$$(R_1\ CP\ R_2)\ CP\ R_3 \Rightarrow R_2\ CP\ (R_1\ CP\ R_3)$$

$$(R_1\ JN_{F1}\ R_2)\ JN_{F2}\ R_3 \Rightarrow R_2\ JN_{F1}\ (R_1\ JN_{F2}\ R_3)$$

Sollten Join und Kartesisches Produkt nicht direkt aufeinanderfolgen, sondern ist diese Folge durch Projektionen oder Selektionen unterbrochen, muß untersucht werden, ob grundsätzlich die Bedingung zum Vorbeischieben des Joins am Kartesischem Produkt erfüllt wird. Ist dies der Fall sind die hemmenden unären Operationen aus dieser Folge herauszuschieben. Dazu sind alle Regeln zur Manipulation von Teilausdrücken zu nutzen, die folgende Operationen beinhalten:

- Projektion und Join
- Projektion und Kartesisches Produkt
- Selektion und Join
- Selektion und Kartesisches Produkt.

3.6.2. Semijoin-Programm

In VDBS können sich die Relationen der Datenbank auf verschiedenen Rechnerknoten befinden. Außerdem ist durch die Existenz mehrerer Rechnerknoten eine Parallelisierung der Anfragebearbeitung möglich. Die dadurch verursachte Abarbeitung der Operationen auf verschiedenen Rechnerknoten macht die Kommunikation zwischen diesen notwendig.

Maßgeblichen Einfluß auf den Umfang des Kommunikationsaufwandes hat die Größe der Relationen, die über das Netz zu verschicken sind.

Eine Möglichkeit zur Reduzierung ihrer Mächtigkeit bietet die Transformation eines Joins in einen Semijoin:

$$R_1 \; JN_F \; R_2 \Rightarrow R_2 \; JN_F \; (R_1 \; SJ_F \; PJ_A(R_2)) \quad \text{mit A} = Attr(R_1) \cap Attr(R_2)$$

Diese Transformation kann in dem Fall hilfreich sein, daß sich R_1 und R_2 auf verschiedenen Rechnerknoten befinden. Bei einem Join müßte dann eine der beiden Relationen, z.B. R_2, komplett zum Rechnerknoten der anderen Relation geschickt werden. Bei der Entscheidung, welche Relation über das Netz zu übertragen ist, würde die kleinere bevorzugt werden. Sollte jedoch auch diese noch sehr groß sein, führt dies zu einem beträchtlichen Kommunikationsaufwand.

Bei obiger Transformation in einen Semijoin wird die Größe der Relation R_2 zunächst durch eine Projektion auf diejenigen Attribute reduziert, die auch Attribute von R_2 sind. Das Ergebnis der Projektionsoperation wird anschließend zum Rechnerknoten von R_1 geschickt, wo zusammen mit der Relation R_1 der Semijoin gebildet wird. Im Ergebnis liegt eine Relation vor, die aus den Attributen von R_1 besteht und nur die Tupel von R_1 enthält, die an einem Join mit R_2 teilnehmen. Die derart in ihrer Kardinalität reduzierte Relation R_2 wird nun zum Rechnerknoten von R_1 geschickt und dort erfolgt dann der abschließende Join zwischen beiden Relationen. Eine Transformation in einen Semijoin bietet somit den Vorteil der Reduzierung von über das Rechnernetz zu verschickenden Relationen.

Werden aufeinanderfolgende Joins in eine Sequenz von Semijoins überführt, bezeichnet man diese Sequenz als Semijoin-Programm. Ob das Semijoin-Programm den Joins vorzuziehen ist, richtet sich danach, wie drastisch durch das Semijoin-Programm die Anzahl der Tupel, die nicht am abschließenden Join teilnehmen, reduziert werden kann. Die nicht am Join teilnehmenden Tupel werden als dangling bezeichnet. Ein Semijoin-Programm unterstützt einen Join in dieser Hinsicht optimal, wenn es einen full reducer darstellt. Zur Erklärung:

> Die ideale Bedingung für einen Join $R_1 \; JN \cdots JN \; R_n$ ist, daß kein R_i dangling Tupels enthält, d.h., es gibt keine Tupel, die nicht am Join teilnehmen. R_i wird reduziert genannt, wenn es keine dangling Tupel enthält. Es gilt dann $R_i = PJ_A \; (R_1 \; JN \; ... \; JN \; R_n)$, mit $A = Attr(R_i)$. Ein Semijoin-Programm wird für die Rela-

tionen R_1, ..., R_n full reducer genannt, wenn nach der Ausführung der Semijoins jedes R_i reduziert ist und somit sämtliche dangling Tupels entfernt wurden.

Ausgangspunkt bei der Ermittlung eines full reducers für eine Anfrage ist die Erstellung eines Joingraphen. Ein Joingraph ist ein Graph, dessen Knoten Relationen und dessen Kanten Joinbedingungen darstellen und aus dem sich die möglichen Join- bzw. Semijoin-Folgen ableiten lassen.

Bsp.:

Anfrage: $R_1.A = R_2.A \wedge$
$R_2.B = R_3.B \wedge$
$R_2.C = R_4.C$

Der diese Anfrage repräsentierende Joingraph ist in Abb. 11 dargestellt.

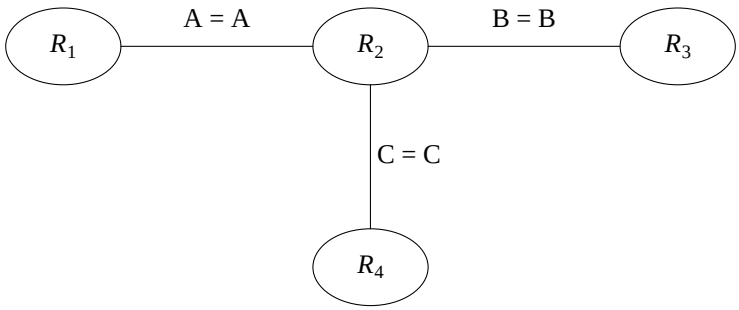


Abb. 11: Joingraph

Zu einer Anfrage existiert ein full reducer, wenn deren Joingraph azyklisch ist. In [Ull88] ist ein Algorithmus vorgestellt, der für eine Anfrage prüft, ob ein full reducer existiert, indem er den Joingraphen auf Zyklenfreiheit untersucht.

Azyklische Joingraphen sind nicht nur Garant für die Existenz eines full reducer, sondern sie liefern auch eine obere Grenze für die Länge des Semijoin-Programms

[CP85]. Diese Grenze ist $n-1$, wobei n die Anzahl der Knoten des Graphen ist. Zyklische Graphen bedingen dagegen nicht die Existenz eines full reducers. Falls aber ein solcher existiert, nimmt dessen Länge mit der Anzahl der Tupel in den Relationen zu.

Die meisten in der Literatur beschriebenen Semijoin-Algorithmen beschränken sich auf die Betrachtung von azyklischen Joingraphen und versuchen eventuell vorab die Transformation eines zyklischen in einen azyklischen Graphen. Zyklische Joingraphen können unter bestimmten Voraussetzungen in azyklische überführt werden. In [YC84] sind hierzu drei verschiedene Algorithmen aufgeführt.

Ein full reducer ist optimal bezüglich der Beschränkung des Datenvolumens. Er muß aber nicht den optimalen Bearbeitungsplan liefern [BC81], denn die Kosten (Kommunikations- und Bearbeitungskosten) des full reducers können seine Vorteile überwiegen. Das Finden eines optimalen Semijoin-Programms stellt ein NP-vollständiges Problem dar [Wan90]. Allerdings gibt es eine bedeutende Klasse von Anfragen, die verketteten Anfragen, für die ein Algorithmus mit polynomialer Zeitkomplexität existiert [ÖV91]. Eine *verkettete Anfrage* hat einen Joingraphen, deren Knoten (Relationen) sequentiell angeordnet sind. Es existiert somit zu jedem Knoten des Joingraphen genau eine Kante zu einem anderen Knoten, siehe Abb. 12.

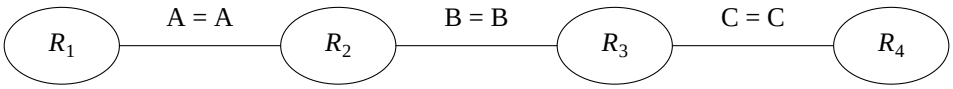


Abb. 12: Joingraph einer verketteten Anfrage

In der Literatur lassen sich eine Reihe von Algorithmen zur Bestimmung eines optimalen Semijoin-Programms finden. Der durch Bernstein [BGW+81] entwickelte SDD-1-Algorithmus gehört zur Klasse der heuristischen Verfahren. Er bewertet für alle potentiellen Semijoins deren Kosten und den Umfang, in welchem der Semijoin die Relation reduziert und wählt auf dieser Grundlage den profitabelsten aus. Der Algorithmus bricht ab, wenn sich keine weiteren profitablen Semijoins finden lassen. Dieser Algorithmus ist in der Literatur für verschiedene Klassen von Anfragen wei-

terentwickelt worden.

Apers, Hevner und Yao [AHY79] entwickelten einen optimalen Algorithmus für einfache Anfragen. Einfache Anfragen sind solche, bei denen die mittels Join zu verbindenden Relationen jeweils nur ein Joinattribut besitzen, das gleichzeitig das einzige gemeinsame Attribut aller mittels Join zu verbindenden Relationen ist. Dieser Algorithmus, der ein optimales Semijoin-Programm liefert, wurde zu einem heuristischen Verfahren weiterentwickelt.

Chiu, Bernstein und Ho [CBH84] entwickelten einen Algorithmus für die Klasse der verketteten Anfragen. Chiu und Ho [CH80] verallgemeinerten diesen Algorithmus für die Behandlung von Anfragen, dessen Joingraph einen Baum darstellt.

In [PV88] wird ein Algorithmus für eine allgemeinere Klasse von Baumanfragen vorgestellt, bei denen die Relationen mehr als ein gemeinsames Attribut haben können. Außerdem berücksichtigt dieser Algorithmus im Gegensatz zum SDD-1-Algorithmus auch Semijoins, die lokal unprofitabel sind. Solche Semijoins können global betrachtet aber profitabel sein, aufgrund der Auswirkung des Reduktionseffekts auf nachfolgende Semijoins,

Die meisten Semijoin-Algorithmen erzeugen nur einen sequentiellen Ausführungsplan für die Semijoins. Die sequentielle Bearbeitung von Semijoins hat den Vorteil, daß durch den Reduktionseffekt eines Semijoins die Kosten für nachfolgende Semijoins reduziert werden können. Auf der anderen Seite bedingt es aber einen Verlust der horizontalen Parallelität. Der optimale Algorithmus von Wang [WCS92] und der heuristische von Pramanik und Vineyard [PV88] berücksichtigen die parallele Ausführung von Semijoins.

Der Nutzen von Semijoin-Programmen kann aber nach [ÖV91] und nach [YC84] sehr zweifelhaft sein, etwa in dem Fall, daß die Kommunikationskosten aufgrund eines sehr leistungsfähigen LAN nicht dominierend sind, und dadurch die Nachteile eines Semijoins seine Vorteile überwiegen. Die Nachteile von Semijoin-Programmen sind die zusätzlich anfallenden Bearbeitungskosten für die Ausführung der Projektion und des Semijoins, der zweimalige Zugriff auf die Relation R_2 und der Umstand, daß die nach dem Semijoin entstehende Zwischenrelation R_1' eventuell existierende Indizes auf der Basisrelation R_1 nicht mehr für den abschließenden Join nutzen kann. Außerdem ist der

Nutzen eines Semijoin-Programms von der Güte der Semijoin-Bedingungen abhängig. Eine geringe Selektivität der Semijoin-Bedingungen verursacht eine Verminderung des Reduktionseffektes bezüglich des Datenvolumens. Bei geringer Selektivität tritt somit die erwartete Reduzierung der Kommunikationskosten nicht ein bzw. selbige erhöhen sich sogar noch.

Im HEAD-Projekt basiert das VDBS auf einem leistungsfähigen LAN. Dies macht es erforderlich, bei der Bewertung der Operationen auch die lokalen Bearbeitungskosten zu berücksichtigen, was eine kritischere Betrachtung von Semijoin-Programmen bedingt. Es läßt sich somit keine allgemeine Heuristik betreffs der Nutzung von Semijoin-Programmen innerhalb der algebraischen Optimierung formulieren. Entscheidungen bezüglich ihres Einsatzes sind in HEAD nur auf der Grundlage von Metadaten und einem Kostenmodell, daß neben den Kommunikationskosten auch die lokalen Bearbeitungskosten berücksichtigt, möglich.

3.6.3. Join-Reducer

Ein Join-Reducer ist eine Sequenz von aufeinanderfolgenden Joins, der für diese eine optimale Reihenfolge darstellt. Bei der Ermittlung der Reihenfolge wird sich die Tatsache zu nutze gemacht, daß Joinoperationen mitunter auch selektiv sein können, d.h., daß die Größe der Ergebnisrelation gegenüber den Eingangsrelationen reduziert wird. Da der Bearbeitungsaufwand einer Joinoperation maßgeblich durch die Größe der Eingangsrelationen bedingt wird, werden außerdem Joinoperationen mit kleinen Eingangsrelationen favorisiert. In einem Join-Reducer werden reduzierende Joinoperationen mit kleinen Eingangsrelationen zuerst ausgeführt, denn sie senken nicht nur die Kommunikationskosten, sondern auch die Bearbeitungskosten für die nachfolgenden Joins. Im Gegensatz zu Semijoin-Programmen entfällt damit die meist einseitige Betrachtung der Kommunikationskosten. Außerdem fallen keine zusätzlichen Bearbeitungs- und Kommunikationskosten an. In [CY90] und [CY94] wird ein Verfahren vorgestellt, welches unter Nutzung von Heuristiken für eine Folge von Joinoperationen einen Join-Reducer ermittelt. In [CY92] wird gezeigt wie obiger Join-Reducer durch das Einfügen von Semijoin-Operationen weiter optimiert werden kann.

Die Ermittlung einer optimalen Joinfolge nach oben genannten Kriterien macht jedoch die Nutzung von Metadaten und die Verwendung einer Kostenfunktion notwendig. Auf deren Basis erfolgen Abschätzungen über die Größe von Zwischenrelationen, die Ausgangspunkt bei der Ermittlung der optimalen Joinfolge sind.

In [Ull80] wird unter Verwendung der Heuristik, daß Relationen, über die eine Selektion ausgeführt wurde, als klein zu betrachten sind, bereits innerhalb der algebraischen Optimierung eine Entscheidungsgrundlage für die Joinfolgen-Optimierung geliefert. Allerdings stellt diese Heuristik nur in dem Fall eine akzeptable Basis für den Größenvergleich zweier Relationen dar, wenn vorausgesetzt werden kann, daß alle Basisrelationen etwa gleich mächtig sind. Eine solche Voraussetzung ist jedoch unrealistisch. Die Festlegung der optimalen Joinfolge entsprechend den oben aufgeführten Kriterien muß somit Bestandteil der internen Optimierung sein und soll in dieser Arbeit nicht weiter betrachtet werden.

3.6.4. Join über horizontal fragmentierte Relationen

Im Gegensatz zu früheren Arbeiten im Rahmen des HEAD-Projekts [Pap91] soll die Fragmentierung von Relationen bei der Optimierung berücksichtigt werden. Nach der Normalisierung einer Anfrage werden neben Joinoperationen verstärkt auch Unionoperationen erzeugt. Diese entstehen durch den Verbund bzw. durch die Vereinigung von Fragmenten. Basierend auf der Transformationsregel zur Änderung der Reihenfolge von Join und Union und unter Ausnutzung von zusätzlichen Informationen, die durch die Verwendung der Algebra AQUAREL zur Verfügung stehen, ergeben sich Möglichkeiten zur Minimierung eines Ausdrucks. Dies soll im weiteren ausführlicher betrachtet werden.

Ein Join zwischen horizontal fragmentierten Relationen kann auf zwei verschiedene Arten vollzogen werden. Bei der klassischen Variante erzeugt man aus den einzelnen Fragmenten in einem ersten Schritt mittels der Operation Union die vollständigen Relationen und führt über diese dann den Join aus. Bei der zweiten Variante, bezeichnet als verteilter Join, wird zuerst jedes Fragment der einen Relation mit jedem Fragment der anderen Relation durch eine Joinoperation verbunden, und anschließend werden die Ergebnisse der partiellen Joins durch die Unionoperation zusammengefaßt. Der ver-

teilte Join ermöglicht damit die parallele Bearbeitung der bearbeitungsaufwendigen Joinoperation.

In Abhängigkeit von der Fragmentierung und den Joinattributen und unter Verwendung der Algebra AQUAREL muß aber nicht jeder mögliche Join zwischen den Fragmenten beider Relationen ausgeführt werden. Nicht betrachtet werden die Joins, deren Qualifizierung inkonsistent ist und die damit die leere Menge liefern. Damit wird es möglich, den Joingraph in seiner Komplexität zu reduzieren. Die Entscheidung, welche der beiden Varianten der anderen vorzuziehen ist, richtet sich vor allem nach der Komplexität des Joingraphen [CP85].

Der Joingraph ist von maximaler Komplexität, wenn für mindestens eine der beiden betrachteten Relationen gilt, daß deren Fragmentierungsattribut kein Joinattribut dieses Joins ist. Sind die Fragmentierungsattribute beider Relationen Joinattribute, kann der Joingraph dennoch komplex sein. Das ist der Fall, wenn beide Fragmentierungsbedingungen der Relationen in hohem Maße unkompatibel sind.

Der Joingraph ist in seiner Komplexität minimal, wenn beide Relationen nach den gleichen Attributwerten partitioniert sind. Die Fragmentierungsbedingungen sind in diesem Fall voll kompatibel. Jedes Fragment wird hierbei mit maximal einem Fragment der anderen Relation verbunden. Der Joingraph besteht hierbei aus mehreren einfachen Teilgraphen, siehe Abb. 13.

Die für die Optimierung interessanten Fälle werden nachfolgend kurz skizziert. Es ist der verteilte Join $JN_{R.A=S.B}$ zwischen den Relationen R und S auszuführen. Dieser Join soll für verschiedene horizontale Fragmentierungen der beiden Relationen betrachtet werden.

1. Fall: Joingraph bei kompatibler Fragmentierung

Fragmentierung von R:	R1: $A \leq 4$
	R2: $4 < A \leq 10$
	R3: $10 < A \leq 20$
	R4: $20 < A$
Fragmentierung von S:	S1: $B \leq 4$
	S2: $4 < B \leq 10$
	S3: $10 < B \leq 20$
	S4: $20 < B$

Diese Fragmentierung unterstützt den verteilten Join optimal. Der Joingraph besteht aus vier einfachen Joingraphen, so daß lediglich vier Joinoperationen notwendig werden, siehe Abb. 13.

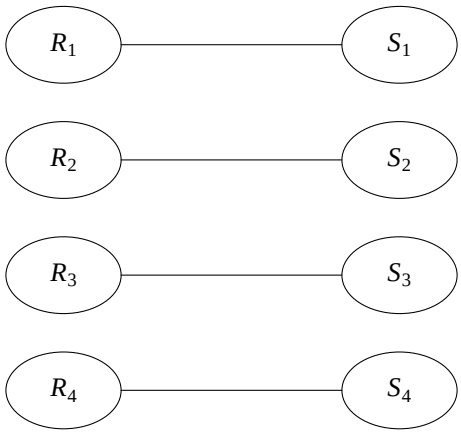


Abb. 13: Joingraph bei kompatibler Fragmentierung

2. Fall: Joingraph bei in hohem Maße unkompatibler Fragmentierung

Fragmentierung von R:

R1: $A \leq 4$

R2: $4 < A \leq 10$

R3: $10 < A \leq 20$

R4: $20 < A$

Fragmentierung von S:

S1: $B \leq 6$

S2: $6 < B \leq 8$

S3: $8 < B \leq 15$

S4: $15 < B$

Diese Fragmentierung bildet eine Unterstützung für den verteilten Join, denn es muß nicht jedes Fragment der einen mit jedem Fragment der anderen Relation verbunden werden. Jedoch ist die gegebene Fragmentierung von R und S in hohem Maße unkompatibel, so daß sieben Joinoperationen notwendig werden, siehe Abb. 14.

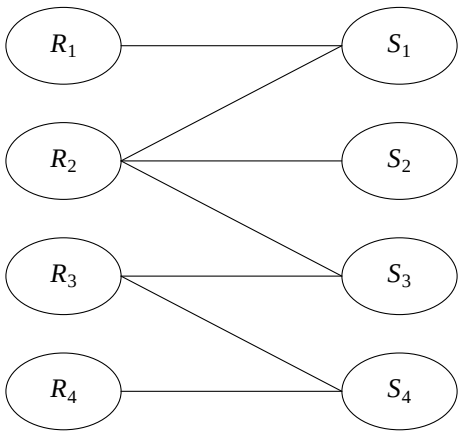


Abb. 14: Joingraph bei in hohem Maße unkompatibler Fragmentierung

3. Fall: Joingraph mit maximaler Komplexität

R sei nach Attribut A und S sei nach Attribut C fragmentiert, wobei beide Relationen aus vier Fragmenten bestehen. Diese Fragmentierung stellt bei gegebener Joinbedingung keine Unterstützung für den verteilten Join dar. Es entsteht ein Joingraph mit maximaler Komplexität, der 16 Joins notwendig macht, siehe Abb. 15.

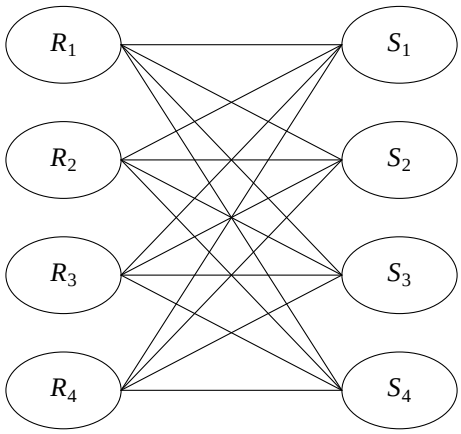


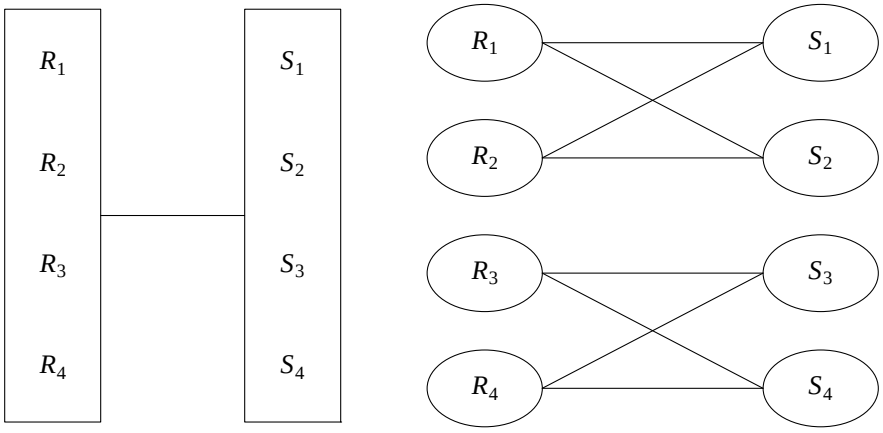
Abb. 15: Joingraph mit maximaler Komplexität

Mit der Komplexität des Joingraphen reduziert sich der Nutzen, der sich aus der parallelen Verarbeitung der Joinoperationen entsprechend dem verteilten Join ergibt. Die Zahl der zu betrachtenden Tupelverbunde steigt. Außerdem erhöht sich die Zahl der Joinoperationen, was gleichzeitig eine Steigerung des durch die Joinoperationen verursachten Kommunikationsaufwandes und der Zahl der Unionoperationen, die die Ergebnisse der Joins zusammenführen, bewirkt. Von einem Einsatz des verteilten Joins bei hoher Komplexität ist somit abzuraten.

Bei den Joingraphen kommt im Rahmen der Betrachtung von klassischem und verteiltem Join dem partitionierten Joingraphen eine besondere Bedeutung zu. Ein partitionierter Joingraph besteht aus mehreren, nicht zusammenhängenden Graphen. Zur Klassifikation von Joingraphen siehe [CGP86]. Der partitionierte Joingraph ermöglicht das Zusammenwirken von klassischem und verteiltem Join im Falle einer hohen Komplexität der Teilgraphen. Bisher wurde davon ausgegangen, daß bei hoher

Komplexität ausschließlich der klassische Join zum Einsatz kommt.

Das Wesen und die Bedeutung des partitionierten Joingraphen sollen an nachfolgendem Beispiel erklärt werden. Es ist eine Joinoperation über die Relationen R und S auszuführen. R und S sind jeweils wieder in vier Fragmente zerlegt. Die Abb. 16(a) zeigt den Joingraphen für den klassischen Join. Hierbei werden die Fragmente durch die Operation Union zuerst wieder zu ihren globalen Relationen R und S zusammengefaßt, und anschließend wird der Join zwischen R und S ausgeführt. In Abb. 16(b) ist der verteilte Join für R und S dargestellt, den man aufgrund einer gedachten Fragmentierung erhält.



(a) Klassischer Join (b) Verteilter Join

Abb. 16: Joingraph für den klassischen und den verteilten Join

Der verteilte Join erfordert die Ausführung von acht Joinoperationen, deren Ergebnisse letztendlich mit Hilfe der Unionoperation zusammengeführt werden.

Der in Abb. 16(b) dargestellte Graph, bestehend aus zwei unabhängigen Teilgraphen, die separat bearbeitet werden können, ist ein partitionierter Joingraph. Die Joins zwischen $\{R_1, R_2\}$ und $\{S_1, S_2\}$ und zwischen $\{R_3, R_4\}$ und $\{S_3, S_4\}$ können somit separat ausgeführt werden. Eine optimierte Bearbeitung des Joins zwischen R und S zeigt der Joingraph aus Abb. 17. Entsprechend diesem Graphen wird als erstes die Union zwischen R_1 und R_2, R_3 und R_4, S_1 und S_2 sowie S_3 und S_4 gebildet. Über das

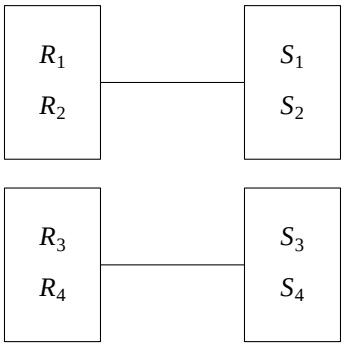


Abb. 17: Partitionierter Joingraph

Ergebnis der Unions werden dann zwei separate Joins ausgeführt. Abschließend werden die Ergebnisse der Joinoperationen mit einer Union zum Endergebnis zusammengeführt. Im Gegensatz zum klassischen Join wird dadurch der Join auf relevante Tupel begrenzt, und außerdem wird im Vergleich zum Joingraphen aus Abb. 16(b) die Anzahl der Joinoperationen verringert. Ein derartiges Vorgehen ermöglicht somit auch bei bestimmten komplexen Joingraphen eine Reduzierung der zu betrachtenden Tupelverbunde bei gleichzeitig begrenzter Anzahl von Joinoperationen. Damit läßt sich folgende Heuristik aufstellen, die durch den Optimierer anzuwenden ist:

H13 Existiert ein Join über zwei fragmentierten Relationen, deren Fragmentierungsattribute gleichzeitig auch Joinattribute sind, ist der Joingraph für die Fragmente der beiden Relationen aufzustellen. Anschließend ist der betreffende Teilbaum derart zu manipulieren, daß eine separate Bearbeitung der einzelnen partitionierten Joingraphen und die Zusammenfassung ihrer Ergebnisse mittels der Operation Union erfolgt.

Für die einzelnen unabhängigen Teilgraphen ist die Zahl der notwendigen Joinoperationen zu ermitteln. Ist die Anzahl der notwendigen Joinoperationen zu groß (>MaxJoinAnzahl), wird für diesen Teilgraphen zuerst die Union der einzelnen Fragmente und dann der Join darüber ausgeführt. Ansonsten sind, entsprechend des partitionierten Joingraphen, zuerst die Joins zwischen den Fragmenten zu bilden und anschließend die Ergebnisse mittels der Operation Union zusammenzufassen.

Angenommen, der Joingraph für den Join über zwei horizontal fragmentierten Relationen bestehe aus drei unabhängigen Teilgraphen, dann ist im Gesamtbaum der Teilbaum bestehend aus Join und den beiden Relationen durch den in Abb. 18 dargestellten Baum zu ersetzen, wobei die Blattknoten dieses Baumes entsprechend der Heuristik des verteilten Joins zu bilden sind.

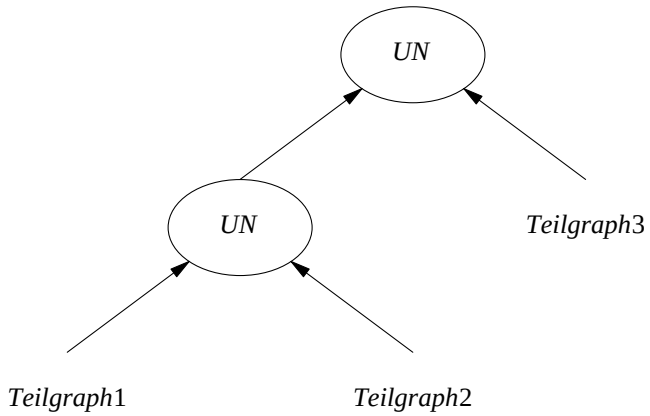


Abb. 18: Realisierung des verteilten Joins

Die Anwendung der Heuristik des verteilten Joins auf einen fragmentierten Operatorbaum erfordert, daß innerhalb des Teilbaumes, der die Sammlung von Fragmenten repräsentiert, außer dem Unionoperator und unären Operatoren auf den Blättern des Baumes keine weiteren Operatoren vorkommen.

Sollte der Baum eine links- bzw. rechtstiefe Struktur aufweisen, ist, trotz der Existenz einer Vielzahl von Joinoperationen und horizontal fragmentierten Relationen, maximal für ein Join die Ausführung als verteilter Join möglich. Eine optimale Unterstützung für den verteilten Join stellt dagegen ein Baum dar, in dem sämtliche Joinoperationen direkt auf die Relationen angewendet werden.

Zur Unterstützung der Heuristik des verteilten Joins ist durch den Optimierer die nachfolgende Heuristik zu realisieren:

H14 Vor der Umsetzung der Heuristik des verteilten Joins ist jeder Join dahingehend zu untersuchen, ob zwei Relationen existieren, die die Abarbeitung dieses Joins als verteilten Join unterstützen. Existieren zu einem Join mehrere Relationenpaare, die diese Bedingung erfüllen, ist das Relationenpaar mit der größten Zahl unabhängiger Teilgraphen auszuwählen. Anschließend sind durch den Optimierer alle diejenigen Transformationen anzuwenden, die die Anwendung des Joins auf dieses Relationenpaar ermöglichen.

Zur Begrenzung der Komplexität dieser Heuristik sollen mögliche Transformationen auf Operationenfolgen beschränkt sein, die nur die Operationen Join, Kartesisches Produkt sowie Selektion und Projektion enthalten. Die durch den Optimierer zur Umsetzung dieser Heuristik zu verwendenden Regeln entsprechen denen der Heuristik zum Verschieben von Joins vor Kartesische Produkte.

3.6.5. Eliminierung von vertikalen Fragmenten

Im Prozeß der Normalisierung wird eine vertikal fragmentierte Relation auf den Join ihrer Fragmente abgebildet. Aufgrund von Projektionen innerhalb der Anfrage ist es möglich, daß einzelne Fragmente einer Relation für die Beantwortung der Anfrage unrelevant sind. Das Ziel ist, diese Fragmente nicht erst im Verlauf der Abarbeitung durch Projektionsoperationen auszublenden, sondern bereits innerhalb der Optimierung zu eliminieren. Das wird erreicht, indem im Prozeß der Normalisierung eine vertikal fragmentierte Relation nur durch den Join derjenigen Fragmente ersetzt wird, die zur Beantwortung der Anfrage erforderlich sind. Folgende Heuristik ist aus diesem Grund durch den Optimierer anzuwenden:

H15 Zu jeder fragmentierten Relation soll der maximale Teilbaum betrachtet werden, für den gilt, daß diese Relation dessen einziger Blattknoten ist. Existiert innerhalb dieses Teilbaumes eine Projektion, ist die Attributmenge zu bilden, die sich aus der Projektionsliste und allen weiteren Attributen ergibt, die als Operator-

argumente in Nachfolgerknoten der Projektion Verwendung finden. Die globale Relation ist dann durch die minimale Anzahl von Fragmenten zu ersetzen, die diese Attributmenge realisieren.

Diese Heuristik soll als Heuristik des vertikalen Joins bezeichnet werden.

Bsp.:

Eine Anfrage soll den Namen und die Personalnummer aller Angestellten liefern, die in der Abteilung drei arbeiten, siehe Abb. 19.

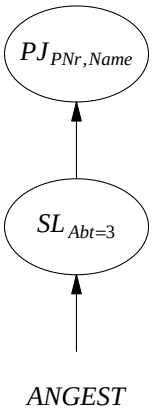


Abb. 19: Ausgangsbaum

Für die Relation ANGEST soll gelten, daß sie vertikal in drei Fragmente fragmentiert ist:

$ANGEST_1$ = {PNr, Name, Adresse},

$ANGEST_2$ = {PNr, Tätigkeit, Gehalt} und

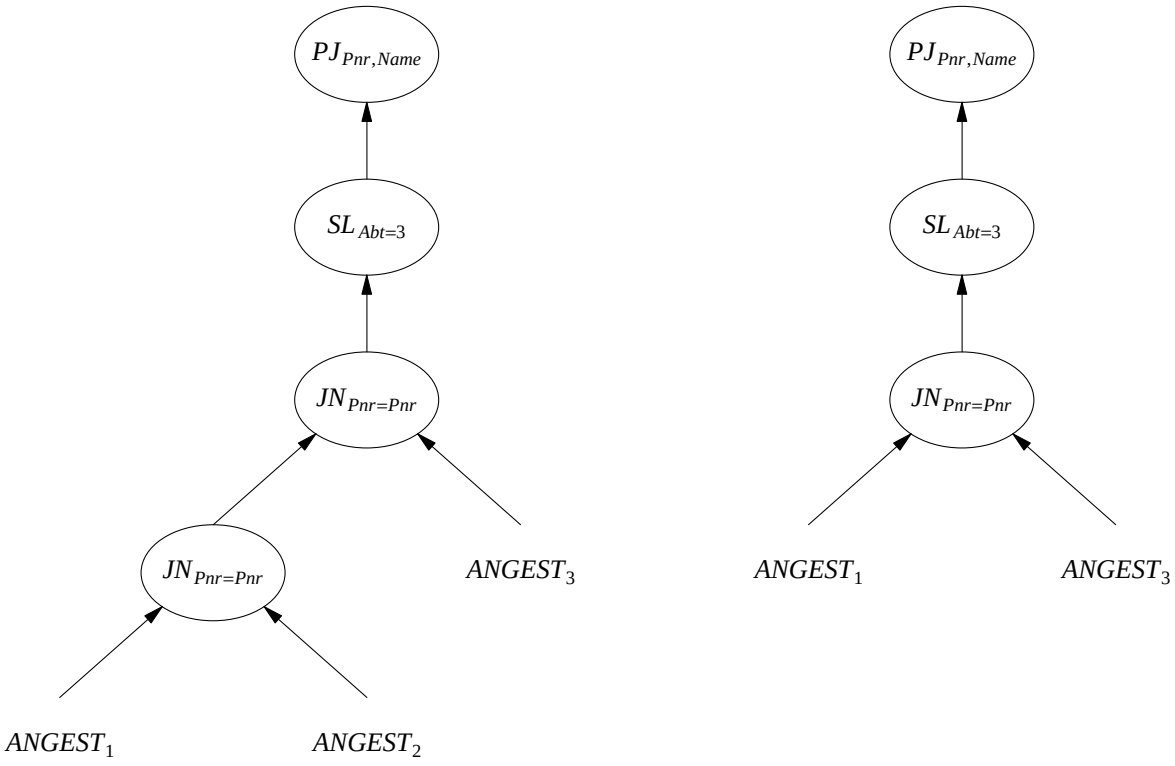
$ANGEST_3$ = {PNr, Abt, Chef).

Es soll geprüft werden, ob innerhalb des Normalisierungsprozesses eine Minimierung des Baumes durch Eliminierung unrelevanter Fragmente möglich ist.

Unter Anwendung der Heuristik des vertikalen Joins wird die minimale Attributmenge, die zur Beantwortung der Anfrage benötigt wird, ermittelt. Diese ergibt sich aus der Projektionsliste und der Selektionsbedingung.

$Attr_{\min}$ = {PNr, Name, Abt}

Diese Attributmenge wird abgedeckt durch die Fragmente $ANGEST_1$ und $ANGEST_3$. Die Relation braucht somit innerhalb der Normalisierung nicht durch alle ihre Fragmente ersetzt werden, sondern es genügt der Join zwischen $ANGEST_1$ und $ANGEST_2$. Abb. 20 zeigt den normalisierten Baum ohne und mit Minimierung.



(a) Normalisierung ohne Minimierung (b) Normalisierung mit Minimierung

Abb. 20: Normalisierter Baum

3.7. Zusammenspiel der Heuristiken in einer Optimiererkomponente

Bisher erfolgte eine separate Betrachtung der durch den algebraischen Optimierer in HEAD umzusetzenden Heuristiken. Deren Zusammenwirken innerhalb einer Komponente soll nun beschrieben werden.

Der algebraische Optimierer in HEAD soll aus mehreren Phasen bestehen, in denen die Umsetzung der verschiedenen Heuristiken erfolgt. Bei der Realisierung des Optimierers ohne eine Zerlegung des Optimierungsprozesses, wären komplizierte Prioritätenrangfolgen zur Durchsetzung der Heuristiken notwendig. Außerdem ist hierbei den Abhängigkeiten zwischen den Regelmengen der verschiedenen Heuristiken besondere Aufmerksamkeit zu schenken. Beispielsweise müß verstärkt auf die Zyklenfreiheit des Optimierungsprozesses geachtet werden. Eine Zerlegung des algebraischen Optimierers hilft die Komplexität des Optimierungsprozesses zu beschränken und erleichtert die Einführung neuer Heuristiken. Des weiteren erlaubt es die beliebige Kombination und mehrfache Anwendung von Heuristiken und bietet somit gute Voraussetzungen für die Sammlung von Erfahrungswerten bezüglich der Anwendungsreihenfolge der Heuristiken. Außerdem ist zu beachten, daß aufgrund von Erfordernissen der internen Optimierung eine Beschränkung auf bestimmte Heuristiken notwendig werden kann. Eine derartige Einschränkung der Menge der anzuwendenden Heuristiken ist bei einer phasenhaften Zerlegung des Optimierers einfach zu realisieren.

Die Heuristiken des verteilten und des vertikalen Joins sind eng mit dem Normalisierungsprozeß verbunden, denn diese Heuristiken werden auf globale Relationen angewendet und nehmen deren Ersetzung durch Fragmente vor. Deshalb soll die Normalisierung zusammen mit der Realisierung der Heuristiken des verteilten und des vertikalen Joins in einer Optimierungsphase erfolgen.

Die Anwendung minimierender Regeln ist in hohem Maße von dem jeweils aktuellen Ausdruck abhängig. Im Ergebnis einer jeden Baummanipulation kann eine Minimierung möglich, aber auch revidiert werden. Deshalb sollen die minimierenden Regeln nicht als eine separate Optimierungsphase realisiert werden, sondern in jede Phase integriert werden. Minimierende Transformationen sind hierbei in jeder Phase gegenüber allen anderen Regeln zu favorisieren, da sie durch die Eliminierung von Operationen und ganzen Teilbäumen in jedem Fall zu einer Vereinfachung der Anfrage beitragen. Als

minimierende Regel soll auch die Zusammenfassung zweier Differenzen zu einer Intersectionoperation zählen.

Der Optimierer soll aus den nachfolgend aufgeführten Phasen bestehen:

(1) Zerlegen der Selektionsoperation in Selektionsfolgen.

(2) Verschieben der Selektionsoperation zu den Blattknoten.

(3) Verschieben der Projektionsoperation zu den Blattknoten.

(4) Zusammenfassen von Selektion und Kartesischem Produkt zu einem Join.

(5) Verschieben der Joinoperation vor das Kartesische Produkt.

(6) Unter Nutzung der Algebra AQUAREL, sind die Qualifizierungen von Zwischenrelationen, die Ergebnis einer Join- oder einer Selektionsoperation sind, zu ermitteln und zu bewerten.

(7) Das Modul sieben beinhaltet die Normalisierung des Ausdrucks unter gleichzeitiger Umsetzung der Heuristiken des verteilten und des vertikalen Joins.

(8) Existiert zu einem Join ein Relationenpaar, daß die Behandlung dieses Joins als verteilten Join unterstützt, ist die Joinoperation zu diesen Relationen zu verschieben.

(9) Faktorisierung

(10) Mit dem DUP-Operator zusammengefaßte gemeinsame Teilausdrücke sollen durch Aufnahme weiterer Knoten vergrößert werden. Dabei wird innerhalb des Baumes nach gemeinsamen unären Knoten gesucht, und es wird versucht, diese durch Verschieben vor den gemeinsamen Teilausdruck in selbigen aufzunehmen.

(11) Erbringung von Dienstleistungen für die interne Optimierung. Denkbar wäre hier etwa die Zusammenfassung von Selektionsfolgen zu einer Operation oder die Erzeugung von reinen Joinfolgen, d.h., innerhalb einer solchen Joinfolge treten keine anderen Operationen auf.

Ebenso wie die mit den Heuristiken verbundenen Baumtransformationen, stellt auch die zeitliche Reihenfolge ihrer Anwendung innerhalb des Optimierungsprozesses eine Heuristik dar. In der Literatur finden sich diesbezüglich nur begrenzt verwertbare Ansätze. Die Ursache liegt in der Beschränkung des Umfangs der verwendeten Heuristiken. In [Ull89] findet sich etwa ein Algorithmus betreffs der Anwendungsreihenfolge von Heuristiken, der lediglich die obigen Module eins, zwei, drei und vier realisiert und Folgen von Projektionen und Selektionen jeweils zu einer Operation zusammenfaßt sowie eine Gruppierung von Operationen vornimmt. Durch die inhärente Parallelität eines Anfragebaumes ergeben sich für diese Gruppen verschiedene Bearbeitungsreihenfolgen, deren optimale Reihenfolge unter Nutzung einer Kostenfunktion ermittelt wird.

Die Zusammenfassung von Projektionen und Selektionen und die Gruppierung von Operationen wird hierbei im HEAD-Optimierer als eine Dienstleistung des algebraischen Optimierers für die interne Optimierung angesehen und soll nicht als eine Heuristik realisiert werden. In der Tabelle 1 ist der Versuch unternommen worden, die Abhängigkeiten der einzelnen Module und die für die Bestimmung der Optimierungsreihenfolge wichtigen Eigenschaften der Heuristiken herauszustellen. Anhand dieser Tabelle lassen sich für die Optimierung folgende günstige Teilreihenfolgen feststellen:

(a) Phase 7 $\Rightarrow$ Phase 1

(b) Phase 1 $\Rightarrow$ Phase 2

(c) Phase 2 $\Rightarrow$ Phase 3, aber mit abschließender Reihenfolge: Phase 3 $\Rightarrow$ Phase 2

(d) Phase 7 $\Rightarrow$ Phase 2

(e) Phase 2 $\Rightarrow$ Phase 4

(f) Phase 4 $\Rightarrow$ Phase 5

(g) Phase 2 $\Rightarrow$ Phase 6

(h) Phase 9 $\Rightarrow$ Phase 6

(i) Phase 4 $\Rightarrow$ Phase 8 $\Rightarrow$ Phase 3 $\Rightarrow$ Phase 7

(j) Phase 1 $\Rightarrow$ Phase 2 $\Rightarrow$ Phase 10

Phase	Unterstützende Maßnahmen	Sonstiges
1		
2	Phase 1	Heuristik: Selektion vor Projektion. Besitzt gute algebraische Eigenschaften, unter anderem bedingungsloses Vorbeischieben an Projektion möglich.
3	Phase 2	Heuristik: Selektion vor Projektion. Besitzt gute algebraische Eigenschaften, aber Vorbeischieben an Selektion nur bedingt möglich und evtl. dabei Erzeugung einer redundanten Projektion.
4	Phase 2	
5	Phase 4	Verändert die Operationenfolge im Baum maßgeblich.
6	Phase 2 und 10	Kann die Baumstruktur erheblich verändern.
7	verteilter Join: 8 vertikaler Join: 3	Durch die Normalisierung können sich neue Möglichkeiten zum Verschieben von Projektions- und Selektionsoperationen zu den Blattknoten ergeben.
8	Phase 4	Verändert die Operationenfolge im Baum maßgeblich.
9	globaler Ausgangsbaum	Bewirkt die maßgebliche Einschränkung aller nachfolgenden heuristischen Transformationen.
10	Phase 1, 2 und 3	Voraussetzung: Phase 9. Bewirkt eine Reduzierung der durch Phase 9 erfolgten Optimierungsbeschränkungen.
11		Erbringt zusätzlich geforderte Leistungen für die interne Optimierung.

Tab. 1: Optimierungsphasen

(k) Phase 3 $\Rightarrow$ Phase 10

(l) Phase 9 $\Rightarrow$ Phase 10

Die Einordnung von Phase 9 in eine Optimierungsreihenfolge ist problematisch. Diese Phase fordert auf der einen Seite als unterstützende Maßnahme die Anwendung auf den initialen Baum, auf der anderen Seite schränkt sie aber alle nachfolgenden Heuristiken in ihrer Wirksamkeit beträchtlich ein.

Ausgehend von den günstigen Teilreihenfolgen kann man zu den drei in Abb. 21 und 22 dargestellten Optimierungsplänen kommen, die sich jeweils in der unterschiedlichen Berücksichtigung der Phase 9 unterscheiden. Der Optimierungsplan I nimmt die Suche und Kennzeichnung von gemeinsamen Teilausdrücken auf den initialen Baum vor. Der erste Teil dieses Optimierungsplanes ist in Abb. 21 und der zweite Teil in Abb. 22 dargestellt.

Der Optimierungsplan II untersucht den Baum auf gemeinsame Teilausdrücke erst im Anschluß an die Verschiebung von Projektions- und Selektionsoperationen zu den Blattknoten und der Minimierung mittels des verteilten und des vertikalen Joins.

Der Optimierungsplan III stellt einen Kompromiß zwischen Plan I und II bei der Berücksichtigung der Phase 9 dar. Im Anschluß an die Zerlegung von Selektionsoperationen und der Verschiebung von Selektionen und Projektionen zu den Blattknoten erfolgt die Suche und Kennzeichnung von gemeinsamen Teilausdrücken.

Eine heuristische Einordnung der Phase 9 und damit eine Festlegung auf einen Optimierungsplan ist nur nach Sammlung von Erfahrungswerten in Bezug auf die durchschnittliche Bearbeitungszeit einer Anfrage unter Verwendung verschiedener Optimierungspläne möglich.

Es ergeben sich zu den aufgeführten Optimierungsplänen weitere interessante Alternativen, wenn man den Prozeß des Suchens und des Zusammenfassens trennt. Das heißt eine Phase untersucht ausschließlich den Baum auf gemeinsame Teilausdrücke, während eine andere Phase die Zusammenfassung gleicher Teilausdrücke mittels des DUP-Operators vornimmt. Die Zusammenfassung mittels Minimierung wird dagegen nicht als eine separate Phase realisiert, da diese Regeln zu den minimierenden Regeln gehören und in jeder Phase angewandt werden.

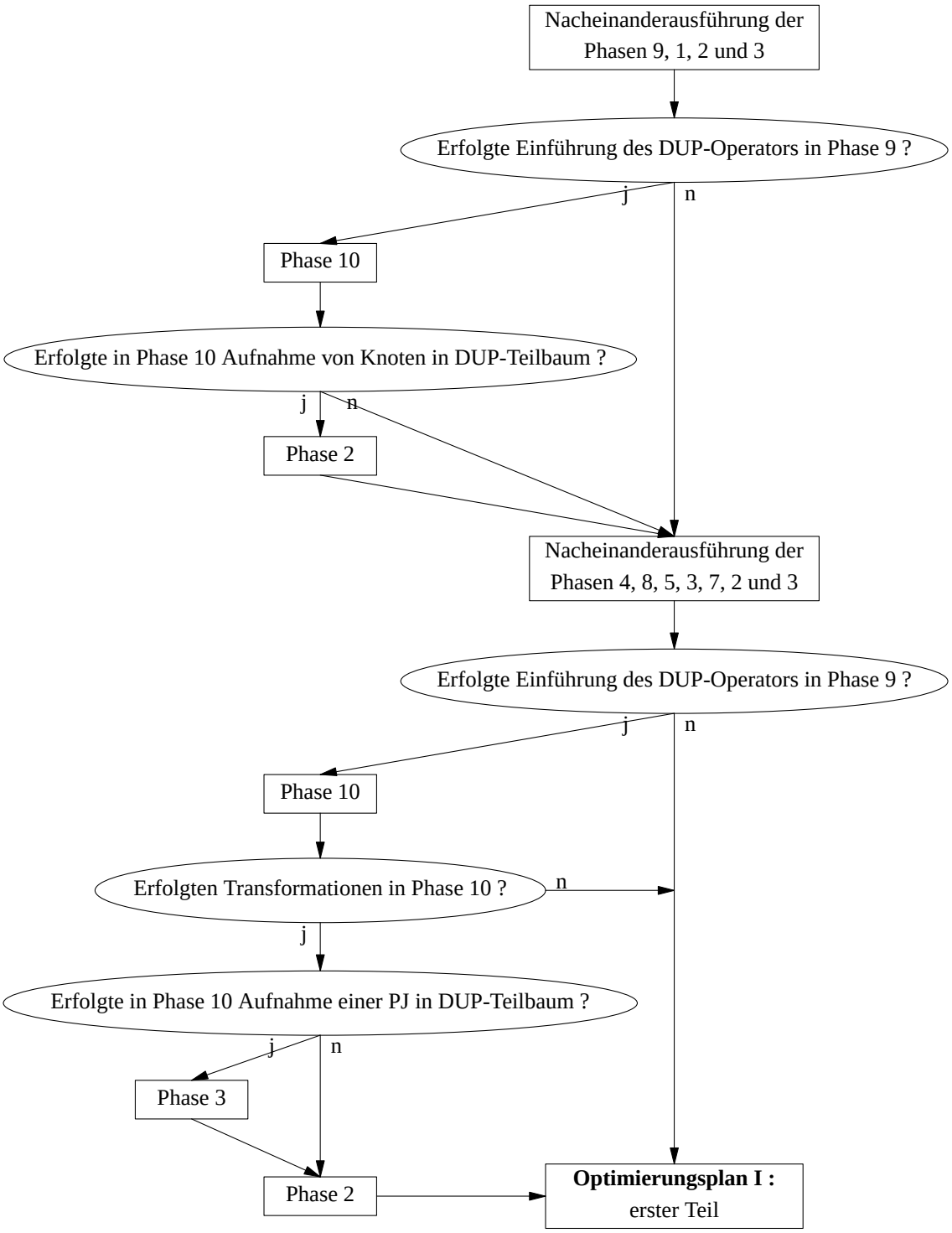


Abb. 21: Erster Teil des Optimierungsplanes I

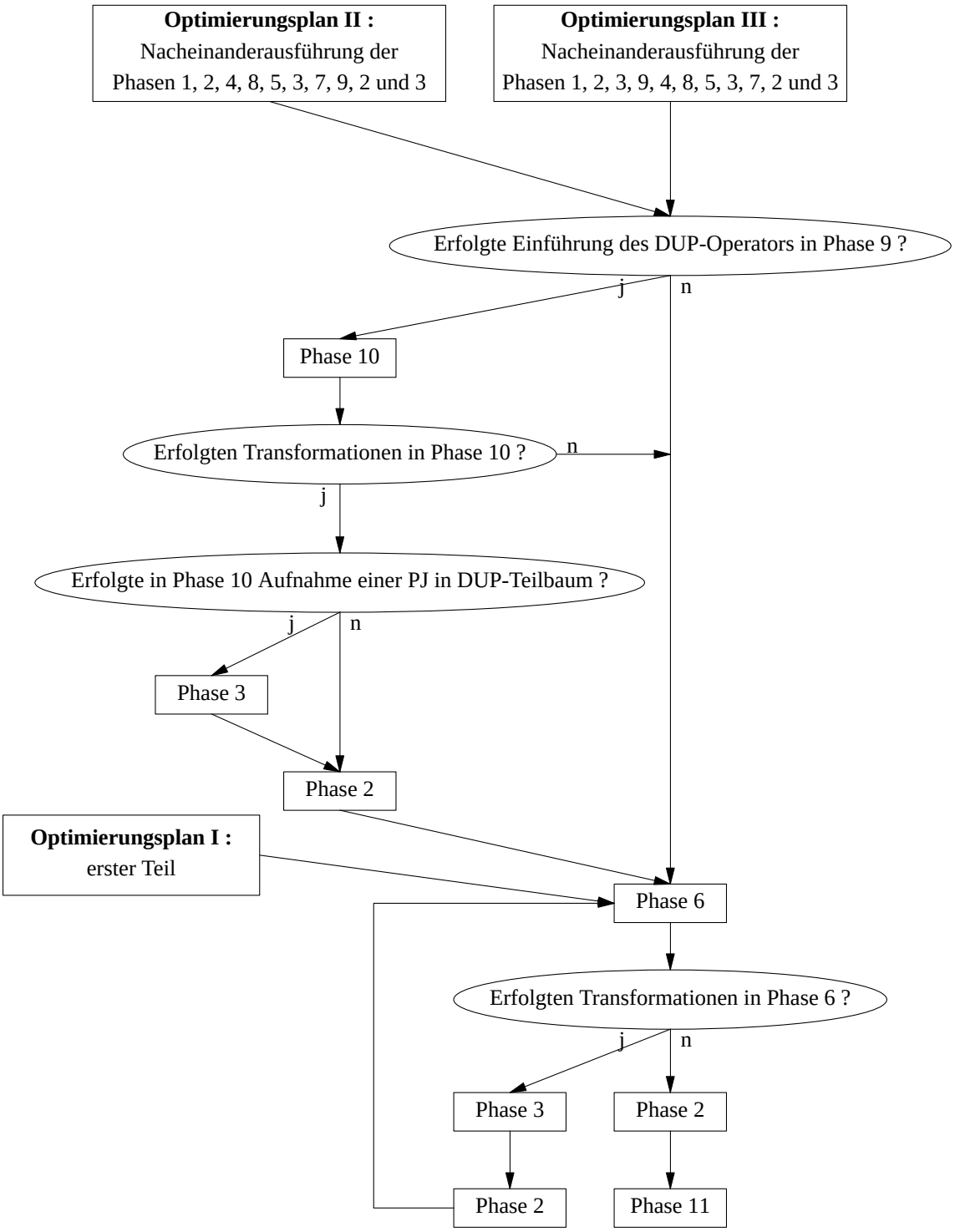


Abb. 22: Optimierungspläne I, II und III

Ein Optimierer könnte dann die Phase des Suchens zu Beginn der Optimierung und eventuell mehrmals in dessen Verlauf abarbeiten und im Anschluß daran jeweils Zusammenfassungen auf Basis minimierender Transformationsregeln vornehmen, ohne daß damit die Möglichkeiten der Optimierung eingeschränkt werden. Die Phase zur Einführung des DUP-Operators wird dagegen erst nach der Realisierung der Heuristiken des verteilten und des vertikalen Joins ausgeführt.

Für den Fall, daß die interne Optimierung die Nichtanwendung bestimmter Heuristiken innerhalb der algebraischen Optimierung vorschreibt, ergeben sich weitere Optimierungspläne.

4. Einsatz von Twig als Optimierer-Generator

4.1. Vorbetrachtungen

4.1.1. Problemstellung

Es ist zu prüfen, inwiefern ein Werkzeug existiert, das die Realisierung des zuvor entworfenen algebraischen Optimierers unterstützt. Dieses Werkzeug soll in der Lage sein, basierend auf einer Spezifikation, automatisch einen entsprechenden Optimierer zu generieren. Es muß die Formulierung und Anwendung aller im Anhang 1 aufgeführten Transformationsregeln garantieren, die heuristische Optimierung unterstützen sowie die Aufgliederung der Optimierung in verschiedene Phasen erlauben. Des weiteren sollte es die problemlose Modifizierung und Erweiterung der Regelmenge ermöglichen.

Das vorliegende Kapitel führt hierzu die Werkzeuge Twig und Volcano OptGen ein. Während Volcano direkt für die Generierung von Anfragoptimierern in DBS entwickelt und bereits in mehreren DBS-Prototypen erfolgreich eingesetzt wurde, ist Twig ein Werkzeug aus dem Bereich des Compilerbaus, das bisher nur zur Codegenerierung eingesetzt wurde, siehe dazu [Tji86]. Da Volcano in der Literatur ([GM93], [Mck93], [Gru94]) bereits ausführlich beschrieben ist und aus nachfolgend aufgeführten Gründen von einem Einsatz von Volcano im Rahmen des HEAD-Projekts abgesehen wird, soll für Volcano im Gegensatz zu Twig nur eine kurze Beschreibung erfolgen.

Zuvor sollen jedoch einige englische Fachbegriffe geklärt werden, für die keine deutsche Entsprechung existiert oder deren Verwendung unüblich ist und die im nachfolgenden benutzt werden. Als Rewriting wird die Manipulation eines Ausdrucks bzw. des ihn repräsentierenden Operatorbaumes bezeichnet. Pattern-Matching bezeichnet den Vorgang des Vergleichens von Mustern. Weitere in diesem Zusammenhang verwendete Begriffe sind Tree-Matcher, als Bezeichnung für den Prozeß, der den Mustervergleich in Bäumen steuert, und Match, als Bezeichnung für den Zustand, daß ein Muster verglichen und erkannt wurde.

4.1.2. Der Optimierer-Generator Volcano OptGen

Vorstellung von Volcano OptGen

Volcano OptGen ist von Bill McKenna mit dem Ziel entwickelt worden, dem Implementierer der Datenbank (DBI) ein Werkzeug zur Verfügung zu stellen, welches ihm ermöglicht, für ein konkretes DBS einen Anfrageoptimierer zu konstruieren. Die Optimierung erfolgt dabei kostenbasiert.

Da Volcano unabhängig von der konkreten Anfragesprache konzipiert wurde, muß diese zuvor spezifiziert werden. Diese Spezifikation wird durch Volcano in den Quelltext des Optimierers übersetzt und zusammen mit anderem Code des DBI, z.B. Kostenfunktionen, übersetzt und gebunden.

Die Spezifikation erfolgt in Form von Regeln, welche neben dem zu matchenden Muster einen Bedingungsteil und einen Aktionsteil enthalten. Mittels der Regeln werden die Äquivalenzregeln der Algebra und die Implementationsregeln der Algebraoperationen beschrieben. Diese zwei Dimensionen spannen den Suchraum auf, in dem Volcano den kostengünstigsten Bearbeitungsplan sucht. Entlang der algebraischen Dimension erfolgt eine erschöpfende Suche. Da mit den Algorithmen Kosten assoziiert sind, führt Volcano OptGen in der zweiten Dimension eine kostenbasierte Suche durch. Die zwei Dimensionen entsprechen gleichzeitig den Phasen des Suchprozesses. In der ersten Phase erzeugt Volcano OptGen alle Algebraausdrücke die äquivalent zum initialen Anfrageausdruck sind. In der zweiten Phase werden die generierten Algebraausdrücke traversiert, um auf der Basis von Implementationsregeln adäquate Algorithmen zu finden. Sowohl in der ersten als auch in der zweiten Phase ist das Pattern-Matching abhängig von den im Bedingungsteil abgefragten Eigenschaften. Volcano unterscheidet dabei logische von physischen (interne) Eigenschaften. Erstere können aus dem Operatorbaum der Anfrage abgeleitet werden, z.B. die Attributmenge von Zwischenrelationen. Die physischen Eigenschaften dagegen beziehen sich auf die konkrete Datenbank. Solche Eigenschaften sind etwa, daß die physische Darstellung der Relation sortiert nach dem Attribut A vorliegt, oder daß die Basisrelation R sich auf dem Netzwerkknoten N befindet. Das Wissen über die physischen Eigenschaften beeinflußt die Auswahl der Algorithmen. Setzt ein Algorithmus physische Eigenschaften der eingehenden Relationen voraus, z.B. die Sortierung über ein bestimmtes Attribut,

die eine Relation nicht erfüllen kann, wird dieser Algorithmus durch den Optimierer verworfen, oder der Optimierer erzwingt die entsprechenden physischen Voraussetzungen durch die Einführung eines sogenannten *enforcer*-Operators in den Baum.

Des weiteren gehen physische Eigenschaften auch bei der Kostenabschätzung von Algorithmen mit ein, denn die Größe von Zwischenrelationen ist z.B. als eine physische Eigenschaft entscheidender Bestandteil einer jeden Kostenfunktion.

Bewertung von Volcano OptGen

Wie bereits erwähnt, ist Volcano speziell für die Generierung von Optimierern in DBS entwickelt worden und wurde außerdem bereits in mehreren Datenbank-Prototypen erfolgreich integriert, u.a. in TIGUKAT [Mun94]) und im Open OODB-Projekt [BMG93].

Nachfolgend aufgeführte Punkte garantieren in Volcano eine einfache Formulierung und eine gute Lesbarkeit der Spezifikation sowie eine unkomplizierte Modifizierung und Erweiterung der Regelmenge:

- Die Spezifikation der Sprache erfolgt über die Angabe der algebraischen Äquivalenzen anstatt über eine kontextfreie Grammatik zur Beschreibung verschiedener Zwischenschritte bei der Transformation, z.B. ein Nichtterminalzeichen für den kommutativen Join.
- In Volcano werden die Regeln in Transformationsregeln für die Algebra und in Implementationsregeln für die Algebraoperationen unterteilt, womit eine klare Trennung zwischen algebraischer und interner Dimension vorgenommen wird.
- Die mit den Regeln verbundenen Baumtransformationen werden von Volcano OptGen selbständig ausgeführt. Im Aktionsteil der Regeln muß lediglich definiert werden, wie sich die Argumente der Operationen (Projektionsliste, Qualifizierung bzw. Joinbedingung) zusammensetzen.

Des weiteren ist der verwendete Suchalgorithmus sehr effizient realisiert. Um zu vermeiden, daß gleiche Ausdrücke, die aufgrund algebraischer Transformationen auf

verschiedenen Wegen erzeugt wurden, redundant optimiert werden, deckt der Suchalgorithmus redundante Ableitungen desselben Algebraoperators bzw. Algorithmus auf und speichert diese in einer Hash-Tabelle. Außerdem erfolgt die Suche zielgerichtet, da nur solche Algorithmen in Betracht gezogen werden, die die in der Wurzel des Operators geforderten physischen Eigenschaften erfüllen. Solche physischen Eigenschaften sind etwa, daß das Ergebnis aufgrund einer *ORDER-BY*-Klausel sortiert vorliegen muß, oder daß die Ergebnisrelation in einer Netzwerkumgebung auf einem bestimmten Rechnerknoten zur Verfügung stehen soll. Der Suchalgorithmus führt weiterhin ein branch-and-bound-prunning des Problemraums auf der Grundlage der Kosten durch. Dabei wird der Suchraum, unter Nutzung der Kosten für den aktuell favorisierten Bearbeitungsplan, begrenzt. Dessen Kosten bilden die obere Grenze für alle weiteren Pläne. Ein Plan wird bei fortschreitender Suche nur dann in Betracht gezogen, wenn seine Ausführungskosten ein neues Minimum bei der Lösung des Problems darstellen.

Trotz dieser Vorzüge scheidet Volcano als Werkzeug-Unterstützung aus, da es mit Volcano nicht möglich ist den in Kapitel 3 spezifizierten Optimierer zu generieren. Nachfolgend angeführte Punkte belegen dies:

- Innerhalb der ersten Dimension wird in Volcano der Suchraum stets erschöpfend durchsucht. Eine Begrenzung des Suchraumes ist lediglich durch die kritische Wahl der erlaubten Transformationen möglich. Die Anzahl der Transformationen kann begrenzt werden, indem für bestimmte äquivalente Ausdrücke die Ableitung nur in eine Richtung erlaubt wird, und indem Transformationsregeln, die unter dem Gesichtspunkt der Optimierung keinen Erfolg versprechen, nicht formuliert werden. Diese Form der heuristischen Beschränkung des Suchraumes ist aber für die Generierung, des in dieser Arbeit entworfenen, heuristischen Optimierers, unzureichend.
- Die erschöpfende Suche entlang der ersten Dimension erfordert immer die kostenbasierte Suche entlang der zweiten Dimension. Die Optimierung läßt sich damit nicht auf die ausschließliche Betrachtung der algebraischen Dimension beschränken und nicht in mehrere getrennt voneinander durchführbare Optimierungsphasen zerlegen.

- Nicht alle der für den algebraischen Optimierer in HEAD erforderlichen Transformationsregeln lassen sich mit Volcano spezifizieren. In verteilten Datenbanksystemen kann eine Anfrage zerlegt in Teilanfragen durch mehrere Rechnerknoten bearbeitet werden. Aus diesem Grund sind Transformationen, die binäre Operationen splitten und damit zu einer Erhöhung der horizontalen Parallelität führen, durch den Optimierer zu berücksichtigen. Nachfolgend angegebene Transformationsregel gehört hierzu:

$$(R_1\ UN\ R_2)\ JN_F\ R_3 \Rightarrow (R_1\ JN_F\ R_3)\ UN\ (R_2\ JN_F\ R_3)$$

  Diese Transformation bewirkt nicht nur eine Erhöhung des horizontalen Parallelitätsgrades, sondern ist auch Voraussetzung für die Realisierung der Heuristik des verteilten Joins. Transformationen dieser Art können jedoch durch Volcano nicht realisiert werden. Anstatt eines Knotens muß hierbei ein ganzer Teilbaum kopiert werden. Volcano unterstützt aber nur flaches Kopieren.

Volcano kann damit die durch den algebraischen Optimierer aus Kapitel 3 erhobenen Forderungen nicht erfüllen und scheidet als Werkzeug-Unterstützung aus.

4.2. Twigs Paradigma

Twig ermöglicht die Generierung von Programmen, die Pattern-Matching und Rewriting in Bäumen realisieren. Es wurde von Steven W. K. Tjiang entwickelt und basiert auf einem in [ASU88] beschriebenen Baum-Übersetzungsschema. Twig wurde ursprünglich für die Code-Generierung entwickelt. Auf diesem Gebiet liegen auch erste Erfahrungen vor. Mit Twig wurde bisher ein VAX-Code-Generator für den pcc2-Compiler geschrieben, der sich um 25 % schneller als der originale Code-Generator erwies.

Bei der weiteren Betrachtung von Twig soll dem Aspekt, daß für eine spezifizierte Quellsprache ein Code-Generator zu erzeugen ist, keine Beachtung geschenkt werden. Vielmehr soll Twig unter dem Gesichtspunkt des Einsatzes als Optimierer-Generator für ein DBS vorgestellt werden. Ohne Einschränkung der Allgemeingültigkeit sei fest-

gelegt, daß die zu optimierende Anfrage ein Ausdruck der Relationenalgebra ist.

Die Abb. 23 demonstriert den Einsatz von Twig als Optimierer-Generator.

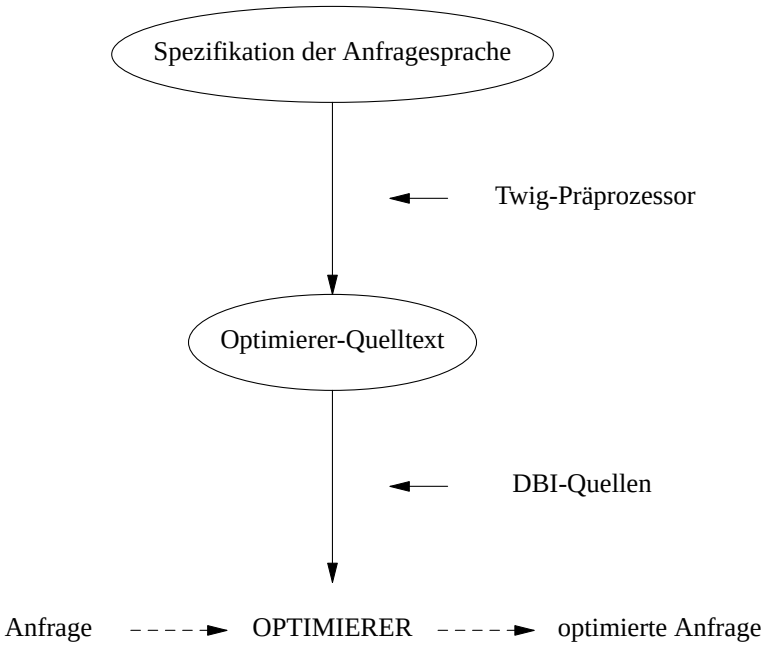


Abb. 23: Twig als Optimierer-Generator

Da Twig vollkommen unabhängig von der Anfragesprache entwickelt wurde, ist es zunächst notwendig, Konstrukte und Eigenschaften der Sprache zu spezifizieren. Der DBI stellt diese Spezifikation in Form einer Datei mit dem Suffix *.mt* dem Werkzeug Twig zur Verfügung. Die Spezifikation dient als Eingabe für den Twig-Präprozessor, der unter Verwendung des Parser-Generators YACC den C-Quelltext des eigentlichen Optimierers generiert. Das mittels YACC generierte Quelltextgerüst wird durch C-Codefragmente des DBI aus *. *mt* erweitert. Diese C-Codefragmente beschreiben z.B. Baumtransformationen und Kostenberechnungen und repräsentieren somit notwendige Optimierungskomponenten. Als Ergebnis liefert der Twig-Präprozessor die Dateien *walker . c* und *symbols.h*. Nach erfolgreichem Übersetzen und Binden der Dateien *walker.c, walker.h* und *symbols.h* zusammen mit weiterem Code des DBI liegt im Ergeb-

nis der lauffähige Optimierer vor.

Ein- und Ausgabe eines mit Twig erzeugten Optimierers sind jeweils baumartige Datenstrukturen. Als Eingabe wird dem durch Twig generierten Optimierer eine Anfrage in Form einer baumartigen Datenstruktur übergeben. Als Ergebnis liefert der Optimierer einen optimierten Anfragebaum. Die Datenstrukturen sind in Twig so allgemein deklariert, daß sich Operatorbäume der Relationenalgebra darauf abbilden lassen.

4.3. Spezifikation der Anfragesprache und der Optimierung

Es soll nun beschrieben werden, welche Angaben über die Sprache und über die Optimierung spezifiziert werden müssen.

Die Spezifikation erfolgt in der Datei *. *mt*. Basiskomponente hierbei ist die Regel:

label_id: tree_pattern [cost][action]

Der Ausdruck *tree_pattern* ist das Muster, auf deren Grundlage sich im Baum das Pattern-Matching vollzieht. Erfolgt ein Matchen des Musters im Baum, wird die dazugehörige Kostenkomponente bewertet. Die jeweiligen Endkosten bilden die Bewertungsgrundlage für die dynamische Programmierung. Mittels der dynamischen Programmierung wird der Baum mit den geringsten Kosten, der optimale Baum, ermittelt. Die Aktionskomponente aller derjenigen Regeln, deren Muster zur Erzeugung dieses Baumes gematched wurden, wird nun abgearbeitet. Auf das Wirken von Pattern-Matching und dynamischer Programmierung soll im Abschnitt 3.4 näher eingegangen werden.

Der Ausdruck *label_id: tree_pattern* einer Regel beschreibt die Syntax der Anfragesprache. Die syntaktische Struktur wird durch eine kontextfreie Grammatik beschrieben. Die Terminalzeichen (*node_id*) der kontextfreien Grammatik bezeichnen die internen Knoten und die Blätter des Baumes und die Nichtterminalzeichen (*label_id*) die Baummuster.

Bsp.:

```
label       rel;
node        SCAN;
node        PROJ;
node        SEL;
node        UNION;

rel:        SCAN;
rel:        PROJ(rel);
rel:        SEL(rel);
rel:        UNION(rel, rel);
```

Ein Ausdruck, der durch diese Grammatik als syntaktisch korrekt erkannt werden würde, ist als Operatorbaum in Abb. 24 dargestellt.

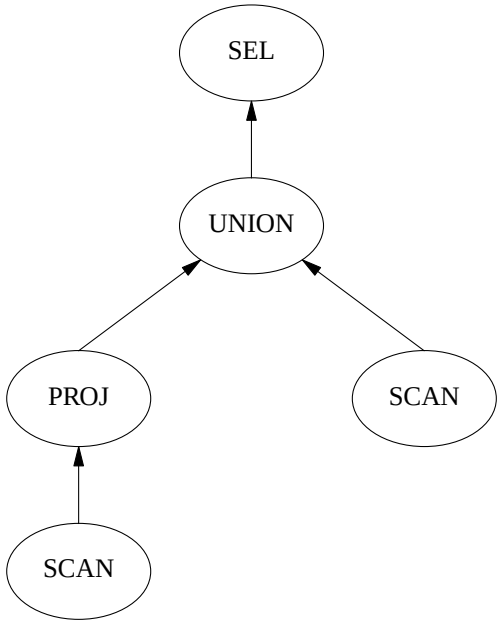


Abb. 24: Operatorbaum

Der Ausdruck *tree_pattern* und der Aktionsteil *action* realisieren in ihrer Einheit die für die Optimierung zugelassenen Transformationsregeln. Der Kostenteil *cost* diktiert die

Bedingungen, unter denen die Transformation erfolgt. Innerhalb einer Regel sind die Kosten- und die Aktionskomponente jeweils optional.

Die Kostenkomponente

Für den Tree-Matcher stellt *cost* einen abstrakten Datentyp dar, so daß Twig die Verwendung und Berechnung beliebiger Kostenarten erlaubt. Es werden lediglich folgende Definitionen erwartet:

- *DEFAULT – COST* und *INFINITY* (maximale Kosten)
- eine Funktion *COSTLESS(cost1, cost2),* die true liefert, wenn *cost1 < cost2*
- *cost* muß ein C-Typ sein

Jeder Regel können Kosten zugewiesen werden. Fehlt die Kostenkomponente, werden die Kosten der Regel auf *default* gesetzt. Das nachfolgende Beispiel zeigt die Formulierung der Kostenkomponente innerhalb einer Regel.

Bsp.:

```
rel:    SCAN {
            cost = costSCAN
        };

rel:    PROJ(rel) {
            cost = costSCAN + $%1$->cost
        };

rel:    SEL(rel);

rel:    UNION(rel, rel) {
            cost = costUNION + $%1$->cost + $%2$->cost
        };
```

Die Kosten für die Regel *rel: SEL(rel)* sind nicht definiert und werden deshalb auf default gesetzt. Der Zugriff auf die Kosten der Nachfolgerknoten wird unterstützt durch

interne Datenstrukturen. Mittels *$%n$->cost* wird auf die Kosten des n.ten Nachfolgerknotens zugegriffen. Sowohl die Kosten- als auch die Aktionskomponente einer Regel sind in C-Code geschrieben.

Die Aktionskomponente

Im Aktionsteil können verschiedene Aktionen realisiert werden. Im allgemeinen erfolgt hier ein Rewriting auf dem betrachteten Teilbaum. Der Zugriff auf die Nachfolgerknoten wird durch interne Datenstrukturen unterstützt. Mittels $\$n1.n2.\cdots nk-1.nk\$$ wird auf den (nk).ten Nachfolger des $(nk-1)$.ten Nachfolger des ... des $(n2)$.ten des $(n1)$.ten Nachfolger der Wurzel des betrachteten Teilbaumes zugegriffen. $$ bezeichnet die Wurzel des erkannten Teilbaumes. Mittels *return(newtree)* wird der alte Teilbaum durch den neuen Teilbaum ersetzt.

Bsp.:

Die Regel für die *UNION* soll um die Aktionskomponente ergänzt werden. Der Aktionsteil bewirkt, daß die Operanden der Unionoperation vertauscht werden.

```
rel:    UNION(rel, rel) {
                    cost = costUNION + $%1$->cost + $%2$->cost
        } = {
                    tmp = $1$;
                    setleft($$, $2$);
                    setright($$, tmp);
                    return($$);
                };
```

Ebenso wie die Kosten sind auch die Bäume als abstrakter Datentyp implementiert. Der DBI muß lediglich einige Funktionen, die das Traversieren und Manipulieren von Bäumen realisieren, definieren und die Knoten vom Typ *NODEPTR* vereinbaren.

4.4. Pattern-Matching und Rewriting in Bäumen

4.4.1. Vorgehensweise

Der Prozeß des Pattern-Matching und der Prozeß des Rewriting von Bäumen entsprechen in Twig in ihrer Einheit dem aus dem Compilerbau bekannten Prozeß der Codeerzeugung durch Umwandeln von Bäumen. Hierbei werden in einem gegebenen Baum die Regelmuster auf die Teilbäume angewendet. Falls ein Muster erkannt wurde, wird der entsprechende Teilbaum durch den Ersetzungsknoten, dem *label*, ersetzt und der an die Regel geknüpfte Aktionsteil ausgeführt. Werden mehrere Muster erkannt, erfolgt die Auswahl über die den Regeln zugeordneten Kosten. Die kostengünstigste Ersetzung, bezogen auf den kompletten Baum, wird durch dynamische Programmierung erreicht. Das Problem der kostengünstigsten Ersetzung für den kompletten Baum wird damit zerlegt in Teilprobleme bezüglich der kostengünstigsten Ersetzung von Teilbäumen.

Pattern-Matching

Der Anfragebaum wird in Preorder-Reihenfolge nach Mustern durchsucht. Mehrere Muster mit verschiedenen Ersetzungen (*labels*) können erkannt werden, aber nur das Muster mit den geringsten Kosten wird durch seinen *label* ersetzt, und der Pattern-Matching-Prozeß wird mit diesem *label* als Blattknoten fortgesetzt.

Einige Optimierer, u.a. der RELAX-Optimierer des DBMS IRIS ([LR91]), untersuchen den eingegebenen Baum nur lokal. Der Pattern-Matcher ist hierbei nur in der Lage, kleine Muster im Baum zu erkennen. Dadurch werden unter Umständen vielversprechende Optimierungsmöglichkeiten nicht erkannt, bzw. viele der im Anhang 1 aufgeführten Transformationsregeln lassen sich erst gar nicht formulieren. Twig umgeht dieses Problem, indem es die Formulierung beliebig großer Muster in den Regeln erlaubt.

Rewriting

Nachdem der Pattern-Matching-Prozeß vollständig abgelaufen ist, beginnt die Rewriting-Phase. In dieser Phase erfolgt die Umsetzung der Transformationsregeln. Auf den gematchten Teilbaum werden dazu Baumtransformationen angewendet. Die Rewriting-Phase kann auch vor der kompletten Abarbeitung der ersten Phase einsetzen. Dieser Fakt soll nachfolgend ausführlicher betrachtet werden.

4.4.2. Möglichkeiten der Steuerung

Die Prozesse Pattern-Matching und Rewriting können gesteuert werden. Dies ist zum einen über die Formulierung von Kosten und zum anderen über das Setzen eines Status für die Regel möglich. Es stehen hierbei der Status *ABORT, REWRITE* und *TOPDOWN* zur Verfügung. In Abhängigkeit von dem gesetzten Status ist es möglich, das Matchen eines Musters von einer Bedingung abhängig zu machen oder die Optimierung auf Teilbäume zu beschränken.

4.4.3. Kostenbasiertes Pattern-Matching

Über die Kosten ist es in Twig möglich, Prioritäten auf den Regeln zu definieren. Auf dieser Grundlage kann man Heuristiken definieren, nach denen die Optimierung erfolgen soll. Der im Abschnitt zur Guckloch-Optimierung spezifizierte Optimierer priorisiert die Transformation "Verschieben der Selektion vor die Projektion" vor allen anderen Regeln und bewirkt damit, wie im dortigen Beispielanfragebaum dargestellt, daß durch den Optimierer ein Baum generiert wird, dessen Selektionsoperationen in Richtung Blattknoten verschoben sind.

4.4.4. Bedingtes Pattern-Matching

Innerhalb der Kostenkomponente kann man mittels *ABORT* die Rücksetzung der betreffenden Regel erzwingen. Das geschieht meistens im Zusammenhang mit einem Bedingungstest.

Bsp.:

```
rel:    PROJ(SEL(rel) {
                    cost = costPROJ + $%1$->child;
                    if ( SelectionAttributs.subsetof(ProjectionAttributs))
                        ABORT;
                } = {
                    return(change_nodes($$));
                };
```

Nach dem Matchen des Musters *PROJ(SEL(rel))* im Baum wird der Code der Kostenkomponente ausgeführt. Bilden die Attribute aus der Selektionsbedingung eine Teilmenge der Attribute aus der Projektionsliste, wird die Regel in Abhängigkeit von ihren Kosten angewendet und die Aktionen der Aktionskomponente ausgeführt, welche die Projektion vor die Selektion verschieben. Liefert der Bedingungstest im Kostenteil jedoch den Wert *false*, bekommt die Regel den Status *ABORT*. Damit wird die Regel zurückgesetzt und bei der Auswahl der gematchten Regeln ignoriert.

4.4.5. Guckloch-Optimierung

Die Guckloch-Optimierung ([ASU88]) versucht durch Transformationen auf kleinen Teilbäumen den Gesamtbaum zu optimieren. Dabei ist charakteristisch, daß die einzelnen Optimierungsschritte neue Optimierungsmöglichkeiten bedingen. Bei der Optimierung von Ausdrücken der Relationenalgebra beruhen die meisten Optimierungsschritte nur auf relativ lokalen algebraischen Transformationen und erfordern demzufolge nur die Betrachtung von Mustern geringer Größe, außerdem ergeben sich im Ergebnis einer Baummanipulation in den meisten Fällen neue Optimierungsmöglichkeiten. Die Anwendung der Guckloch-Optimierung kommt diesem Umstand entgegen. Ein Verzicht darauf würde die Betrachtung von sehr

komplexen Mustern notwendig machen, was wiederum die Formulierung von sehr vielen Regeln mit sehr großen Regelköpfen erfordert und den Umfang und den Aufwand für die Spezifikation drastisch erhöht sowie deren Lesbarkeit erschwert.

Angenommen, in dem in Abb. 25 dargestellten Baum sollen die beiden Selektionsoperationen vor die Projektionsoperationen verschoben werden.

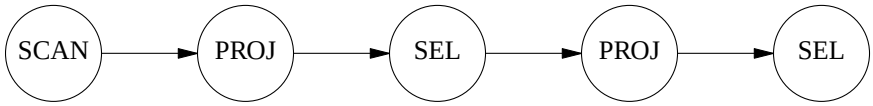


Abb. 25: Zu optimierender Operatorbaum

Bei dem Verzicht auf die Guckloch-Optimierung wird unter Nutzung der dynamischen Programmierung, bei der Wurzel beginnend, für jede Ebene der kostengünstigste Match gefunden. Basierend auf diesem Match erfolgt die Optimierung auf den darüber liegenden Baumebenen. Die Formulierung einer einfachen Regel, die die Selektion vor die Projektion verschiebt, genügt in diesem Fall nicht. Da bereits gefundene Matches für darunterliegende Teilbäume nicht mehr umstoßbar sind, werden hierbei zwar lokal Selektionen vor Projektionen verschoben, aber eventuell dadurch entstandene neue Optimierungsmöglichkeiten für den darunter liegenden Teilbaum werden nicht berücksichtigt. Man würde den in Abb. 26 dargestellten Baum erhalten.

Um in dem gegebenen Beispielbaum beide Selektionsoperationen direkt vor den Blattknoten zu verschieben, wäre die Formulierung folgender Regel notwendig:

rel: SEL(PROJ(SEL(PROJ(rel)))) {cost} = {action};

Der Regelkopf ist hierbei recht komplex, und es ist für jeden möglichen Fall die Formulierung einer weiteren Regel notwendig, beispielsweise bei Existenz einer weiteren Projektions- oder Selektionsoperation. Es soll nun im weiteren auf das Wirken der

Abb. 26: Mittels dynamischer Programmierung erzeugter Baum

Guckloch-Optimierung eingegangen werden, wobei auf den in Abb. 25 dargestellten Baum Bezug genommen wird.

Wenn im Kostenteil der Regeln nicht anders definiert, erfolgt die Optimierung bezüglich des gesamten Baumes. Dabei vollzieht sich das Rewriting der einzelnen Regeln im Baum in der Reihenfolge, in welcher sie gematcht wurden. Wird jedoch für eine Regel der Status *TOPDOWN* oder *REWRITE* gesetzt, läuft die Optimierung lokal ab, d.h. nur begrenzt auf einen Teilbaum (Guckloch-Optimierung).

Angenommen, das Muster einer Regel wurde erkannt und bei der Abarbeitung der Kostenkomponente wird die Regel auf *REWRITE* gesetzt. Sind nun die Kosten dieser Regel niedriger als die aller anderen Regeln, deren Muster ebenfalls erkannt wurden, wird der Aktionsteil dieser Regel und die damit verbundene Baumtransformation sofort ausgeführt. Anschließend wird für diesen transformierten Teilbaum der Pattern-Matching-Prozeß, beginnend bei den Blättern, neu gestartet.

Wurde die Regel dagegen auf *TOPDOWN* gesetzt, wird ebenfalls in Abhängigkeit von der Kostenbewertung sofort der dazugehörige Aktionsteil ausgeführt. Jedoch ist es darüber hinaus möglich, innerhalb der Aktionskomponente mittels *tDO($%$)* explizit die Abarbeitung der Aktionsteile aller gematchten *labels*, die in dem Muster vorkommen, zu erzwingen. Damit kann der DBI willkürlich die Ausführungsreihenfolge der Aktionskomponenten erzwingen. Abschließend wird für den transformierten Teilbaum der Pattern-Matching-Prozeß neu gestartet.

Bsp.:

```
rel:    SCAN;

rel:    PROJ(rel);

rel:    SEL(rel);

rel:    SEL(PROJ(rel) {
                cost = 0
                REWRITE;
        } = {
                return(change_nodes($$));
        };
```

Für die Standardkosten gelte: *DEFAULT_COST = 10*. Damit werden alle Regeln, deren Kosten kleiner als zehn sind, favorisiert.

Der in Abb. 25 dargestellte Algebrabaum soll auf der Basis der obigen Spezifikation optimiert werden.

Beim Traversieren des Baumes wird zuerst das Muster *SCAN* gematched. Der Pattern-Matching-Prozeß ist in diesem Fall auch ohne die Betrachtung der jeweiligen Kosten eindeutig, da *SCAN* $\rightarrow$ *rel* die einzige Regel ist, die angewendet werden kann. Der entsprechende Baum ist in Abb. 27 dargestellt.

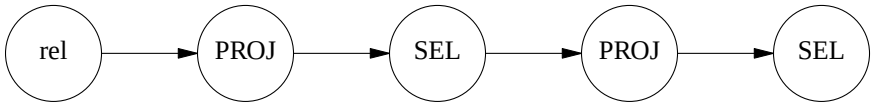


Abb. 27: Baum nach Ersetzen des Musters SCAN

Ausgehend von diesem Baum, erkennt der Pattern-Matcher zwei Muster:

PROJ(rel) -> rel und
SEL(PROJ(rel)) -> *rel*

Da die Kosten, die dem zweiten Muster zugeordnet sind, mit *cost = 0* geringer als die des ersten Musters (*cost = DEFAULT_COST = 10*) sind, wird das zweite Muster ersetzt. Aufgrund der *REWRITE*-Anweisung in der Kostenkomponente der betreffenden Regel wird der Aktionsteil sofort ausgeführt. Der Pattern-Matching-Prozeß für diesen Teilbaum wird abgebrochen und für den nun transformierten Baum (siehe Abb. 28) neu gestartet.

Abb. 28: Resultierender Baum nach dem ersten Optimierungsschritt

Nach dem Ersetzen der Muster *SCAN*, *SEL(rel)* und *PROJ(rel)* durch ihre *label* werden zwei Muster durch den Pattern-Matching-Prozeß erkannt: *PROJ(rel)* und *SEL(PROJ(rel))*. Über die Kosten wird wieder das zweite Muster zum Ersetzen ausgewählt und aufgrund des *REWRITE*-Status der Teilbaum sofort transformiert. Nach dem gleichen Schema wird die Selektionsoperation dann in einem dritten Schritt noch vor die zweite Projektionsoperation verschoben, und es entsteht der in Abb. 29 dargestellte Baum.

Der Pattern-Matching-Prozeß wird für diesen Baum erneut gestartet. Da er diesmal weder auf einen *REWRITE*-Status noch auf einen *TOPDOWN*-Status trifft, wird das Pattern-Matching vollständig abgearbeitet und im Anschluß daran das Rewriting für die ausgewählten Muster gestartet. Da keine angewandte Regel über eine

Aktionskomponente verfügt, erfolgen keine weiteren Baumtransformationen. Das Ergebnis der Optimierung ist der in Abb. 29 dargestellte Baum.

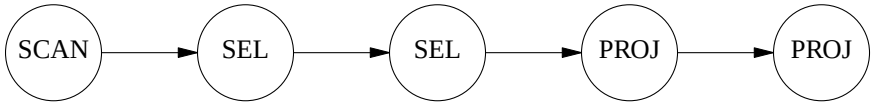


Abb. 29: Resultierender Baum nach dem dritten Optimierungsschritt

4.5. Bewertung bezüglich des Einsatzes im HEAD-Projekt

Obwohl Twig eigentlich für die Codegenerierung konzipiert wurde, ist entsprechend den Darstellungen in diesem Kapitel ein Einsatz als Optimierer-Generator möglich. Negativ ist zu bemerken, daß die vorzunehmende Baummanipulation im Gegensatz zu Volcano nicht in Form einer Regel spezifiziert wird. Die Transformationsregel wird somit nicht vordergründig beschrieben, sondern muß aus dem Aktionsteil der Regel erschlossen werden. Dies erschwert die Lesbarkeit der Spezifikation. Aufgrund dessen, daß die Transformation nicht als Regel spezifiziert ist, wird diese auch nicht durch Twig realisiert, sondern muß durch den Nutzer selbständig implementiert werden. Damit unterliegen aber die Rewriting-Möglichkeiten keiner Beschränkung seitens des Optimierer-Generators. Dies und der Umstand, daß Twig beliebig große Regelköpfe erlaubt, garantiert die Anwendung aller im Anhang 1 aufgeführten Transformationsregeln.

Des weiteren erlaubt Twig eine heuristische Optimierung. Über die Definition von Kosten ist es möglich, Prioritäten für die einzelnen Transformationsregeln zu vergeben und damit den Suchprozeß heuristisch zu steuern. Außerdem kann der Optimierungsvorgang in Phasen zerlegt werden. Hierbei kann in Abhängigkeit von der jeweiligen Optimierungsphase die Zahl der Transformationsregeln eingeschränkt und somit die Realisierung der verschiedenen Heuristiken in einzelnen Phasen erreicht werden. Die Regelmenge der einzelnen Phasen kann sich hierbei auch überschneiden. Damit kann die Anwendung von minimierenden Transformationen in jeder Phase erzwungen

werden. Twig erlaubt die beliebige Kombination und mehrfache Anwendung der Optimierungsphasen.

Durch die Verwendung der Guckloch-Optimierung wird das Pattern-Matching auf die Betrachtung kleiner Muster beschränkt. Außerdem wird damit die Umsetzung einer Transformationsregel in nur Twig-Regel möglich. Trotz expliziter Realisierung der Baummanipulationen und der Bedingungstests durch den Nutzer ist somit der Aufwand für die Modifikation und die Erweiterung der Regelmenge akzeptabel.

Es läßt sich abschließend feststellen, daß mit Twig ein Werkzeug existiert, mit dem es möglich ist, den in dieser Arbeit entworfenen Optimierer zu generieren. Im nächsten Kapitel wird die konkrete Realisierung des algebraischen Optimierers mittels Twig behandelt.

5.1. Realisierung - allgemein

In diesem Kapitel wird die Realisierung des HEAD-Optimierers unter Verwendung von Twig aufgezeigt. Hierbei soll als erstes darauf eingegangen werden, wie der phasenhaft aufgebaute Optimierer, dessen verschiedene Phasen beliebig kombinierbar und mehrfach anwendbar sein sollen, zu realisieren ist.

Twig basiert auf einer kontextfreien Grammatik. Zur Identifizierung der einzelnen Terminal- und Nichtterminalzeichen wird diesen intern ein Integer-Wert zugeordnet. Die in Abb. 30 dargestellte Funktion *mtValue* realisiert diese Abbildung und ist durch den Nutzer dem Werkzeug Twig zur Verfügung zu stellen.

```
int mtValue(node* root)
{
        switch (root->gettype()) {
        case qnode::CPROD:    switch(ostate)      {
                                    case 1:       return(oCPROD2);
                                    case 2:       return(oCPROD3);
                                    }
        case qnode::SEL:  switch(ostate)      {
                                    case 1:       return(oSEL2);
                                    case 2:       return(oSEL3);
                                    }
        case qnode::PROJ:     switch(ostate)      {
                                    case 1:       return(oPROJ2);
                                    case 2:       return(oPROJ3);
                                    }
        case qnode::SCAN:     switch(ostate)      {
                                    case 1:       return(oSCAN2);
                                    case 2:       return(oSCAN3);
                                    }
        }
}
```

Abb. 30: Abbildung der Operationen auf Terminalzeichen

Innerhalb dieser Funktion ist es möglich, die relationenalgebraischen Operationen in Abhängigkeit von einer Zustandsvariablen auf verschiedene Terminalzeichen abzubilden. Auf der Grundlage dieser Zuordnung zu den Terminalzeichen ist in Abb. 31 beispielhaft und verkürzt die Realisierung der Phasen zum Verschieben von Selektionen und Projektionen zu den Blattknoten dargestellt.

```
label     rel2;
node      oSCAN2;
node      oCPROD2;
node      oSEL2;
node      oPROJ2;

label     rel3;
node      oSCAN3;
node      oCPROD3;
node      oSEL3;
node      oPROJ3;

/* Nachfolgende Regeln bewirken die Verschiebung der Selektion zu den Blattknoten. */

rel2:     oSCAN2;
rel2:     oSEL2(rel2);
rel2:     oPROJ2(rel2);
rel2:     oCPROD2(rel2, rel2);

rel2:     oSEL2(oPROJ2(rel2));
rel2:     oSEL2(oCPROD2(rel2, rel2));

/* Nachfolgende Regeln bewirken die Verschiebung der Projektion zu den Blattknoten. */

rel3:     oSCAN3;
rel3:     oSEL3(rel3);
rel3:     oPROJ3(rel3);
rel3:     oCPROD3(rel3, rel3);

rel3:     oPROJ3(oSEL3(rel3));
rel3:     oPROJ3(oCPROD3(rel3, rel3));
```

Abb. 31: Regelmenge zur Realisierung der Phasen zum Verschieben von PJ und SL

Die Separierung der Regelmengen bezüglich der Verschiebung von Projektionen und der Verschiebung von Selektionen erfolgt über die mittels der Variablen *ostate* gesteuerten Zuordnung der Operationen zu Terminalzeichen und über die Formulierung der Regeln. Die Regeln müssen gewährleisten, daß dem Pattern-Matcher nur die Anwendung der für diese Phase erlaubten Transformationen möglich ist.

Die Steuerung der Optimierung erfolgt durch das Hauptprogramm. Dort geschieht das Setzen von *ostate*, die Angabe des zu optimierenden Baumes (*mtMyRoot*) und die Auswahl und der Aufruf der einzelnen Optimierungsphasen. Das in Abb. 32 dargestellte Hauptprogramm führt nacheinander die zwei Optimierungsphasen zum Verschieben von Selektionen und zum Verschieben von Projektionen aus. Die Funktion *_matchinit*() nimmt hierbei bestimmte interne Initialisierungen vor und muß einmal zu Beginn der Optimierung abgearbeitet werden.

```
main()
{
        _matchinit();

        mtMyRoot = operator_tree;
        ostate = 2;
        _match();

        mtMyRoot = operator_tree;
        ostate = 3;
        _match();
}
```

Abb. 32: Steuerung der algebraischen Optimierung

Entsprechend dem beschriebenen Prinzip können beliebig viele Phasen realisiert und deren Anwendung durch das Hauptprogramm gesteuert werden. Problematisch ist hierbei, daß Regeln, die in verschiedenen Phasen genutzt werden, wie z.B. minimierende Regeln, in jeder dieser Phasen explizit formuliert werden müssen, was den Umfang der Spezifikation erhöht.

Die Durchführung von Baumtransformationen und Bedingungstests sowie die Bildung der Operatorargumente ist in Twig Aufgabe des Nutzers. Im Rahmen dieser Arbeit

wurden die möglichen Baumtransformationen als Funktionen realisiert. Die Neubildung von Operatorargumenten und notwendige Bedingungstests erfolgen jedoch spezifischer und sind abhängig von der aktuellen Phase sowie von der zu realisierenden Transformationsregel. Beispielsweise ist für die Klasse aller Regeln, die ein Vertauschen der Reihenfolge von unären Operationen vornehmen, die auszuführende Baumtransformation einheitlich. Die vorzunehmenden Bedingungstests aber unterscheiden sich bezüglich der Transformationsregel und bezüglich weiteren durch die aktuelle Phase bedingten Tests. Jedoch bietet sich auch hier für typische Fälle, wie das Bilden der Differenz, des Durchschnitts und der Vereinigung von Attributmengen von Operanden und Operatorargumenten, die Bildung von Funktionen an. Ein derartiges Vorgehen erleichtert die Erweiterbarkeit der Regelmenge und ermöglicht das Verbergen von Detailfragen vor dem Nutzer.

Die Optimierung kann komplett in einem Durchlauf für den gesamten Baum oder bei der Realisierung der Guckloch-Optimierung in mehreren Durchläufen erfolgen, wobei jeder Durchlauf die lokale Optimierung eines Teilbaumes bewirkt. Welche Variante vorzuziehen ist, richtet sich danach, inwiefern durch eine Transformation neue Optimierungsmöglichkeiten entstehen können. So erfolgt beispielsweise die Optimierungsphase zur Zerlegung von Selektionsoperationen unter Verzicht auf die Guckloch-Optimierung, denn die Zerlegung einer Selektion in eine Folge von Selektionen bewirkt keine neuen Möglichkeiten der Zerlegung. Die Optimierung innerhalb dieser Phase kann somit in einem Durchlauf erfolgen. Die Anwendung der Guckloch-Optimierung würde zwar das gleiche Ergebnis liefern, erfordert jedoch mehrere Pattern-Matching-Prozesse. Allerdings wird in jeder Phase auch die Anwendung minimierender Regeln geprüft und diese erfordern die Guckloch-Optimierung, da sich im Ergebnis einer Minimierung neue Minimierungsmöglichkeiten ergeben können.

Alle Regeln, die die Guckloch-Optimierung erzwingen, realisieren dies im algebraischen Optimierer von HEAD mittels Setzen des REWRITE-Status.

Jede unter Verwendung von Twig zu bearbeitende Phase besteht aus drei verschiedenen Arten von Regeln, und zwar aus Regeln, die die in dieser Phase zu realisierende Heuristik unterstützen, aus Regeln zur Minimierung und aus Regeln, die lediglich dem Erkennen der einzelnen relationenalgebraischen Operationen dienen. Hierbei gilt, daß minimierende vor unterstützende Regeln anzuwenden sind.

Die Anwendung der Regeln ist in jeder Phase in folgender Reihenfolge zu vollziehen:

(1) Anwendung der minimierenden Regeln

(2) Anwendung der Transformationsregeln, die die Realisierung der aktuellen Heuristik unterstützen

(3) Anwendung der Regeln, die lediglich dem Erkennen von Operationen dienen

Zur Realisierung dieser Reihenfolge sind die Kosten der Regeln vom Typ (1) auf null, die vom Typ (2) auf fünf und die vom Typ (3) auf zehn gesetzt. Dies bewirkt, falls durch den Pattern-Matcher mehrere Muster erkannt wurden, eine Favorisierung der Regel mit den geringsten Kosten.

Die Transformationsregeln aus Kapitel 2 finden ihre Realisierung in Regeln des Typs (1) und (2). Dabei bildet die linke Seite einer Transformationsregel das Muster, nachdem im Baum das Pattern-Matching erfolgt. Der Bedingungsteil einer Regel ist in der Kostenkomponente realisiert. Die Aktionskomponente ist verantwortlich für das Bilden der rechten Seite der Transformationsregel, wobei Einschränkungen aus einer eventuell existierenden Konstruktionsregel berücksichtigt werden.

Die Realisierung der einzelnen Phasen wird nachfolgend erläutert. Dabei soll nur auf diejenigen Phasen ausführlicher eingegangen werden, die nicht entsprechend der Spezifikation aus Abb. 31 realisiert sind bzw. Besonderheiten aufweisen.

Die Phasen zum Verschieben von Joins vor Kartesische Produkte und zum Verschieben von Joins vor horizontal fragmentierte Relationen erfordern eine links- bzw. rechtstiefe Ausrichtung der binären Operationen. Da die Überführung eines Baumes in eine derartige Struktur innerhalb der Optimierung vorhergehende Transformationen rückgängig machen könnte, sei das Vorliegen eines linkstiefen Baumes zu Beginn der Optimierung vorausgesetzt. Alle im Verlauf der Optimierung verwendeten Transformationsregeln erhalten, sofern nicht anders aufgeführt, die linkstiefe Ausrichtung der binären Operationen.

5.2. Realisierung der einzelnen Optimierungsphasen

Phase 1: Zerlegen von Selektionen

Sollte die Anwendung minimierender Transformationen nicht möglich sein, erfolgt die Zerlegung von Selektionen unter Verzicht auf die Guckloch-Optimierung in einem Durchlauf.

Phase 2: Verschieben der Selektionsoperation zu den Blattknoten

Die Umsetzung dieser Heuristik erfolgt unter Nutzung der Guckloch-Optimierung.

Phase 3: Verschieben von Projektionsoperationen zu den Blattknoten

Die Umsetzung dieser Heuristik erfolgt unter Nutzung der Guckloch-Optimierung.

Zu beachten sind in dieser Phase diejenigen Regeln, die ein Splitten der Projektion in zwei Projektionsoperationen bewirken; nachfolgende Regel gehört hierzu:

$$PJ_A(SL_F(R)) \Rightarrow PJ_A(SL_F(PJ_B(R)))$$

Ihre Implementierung ausschließlich nach den Erfordernissen der Transformationsregel bewirkt beim Matchen einer solchen Regel während der Optimierung einen Zyklus, der die fortwährende Anwendung dieser Regel erzwingt. Das wird im algebraischen Optimierer von HEAD durch einen zusätzlichen Bedingungstest im Kostenteil der entsprechenden Regel verhindert. Dieser bezieht sich auf die Attributmenge von R. Sollte etwa bei der angeführten Regel diese Attributmenge identisch mit der Attributmenge sein, die sich aus der Vereinigung der Attributmenge der Projektionsliste und der Selektionsbedingung ergibt, ist die Anwendung dieser Regel zu verwerfen.

Phase 4: Zusammenfassung von Selektion und Kartesischem Produkt zum Join

In Abb. 33 ist die zur Realisierung dieser Heuristik notwendige Regelmenge verkürzt dargestellt. Es fehlen in dieser Darstellung Regeln zum Erkennen weiterer relationenalgebraischer Operationen sowie die Aktions- und Kostenkomponente der einzelnen Regeln.

Wird im Baum ein Muster als Kartesisches Produkt erkannt, erfolgt dessen Ersetzung durch das Nichtterminalzeichen *rel*2*CP*. Alle Projektionen und Selektionen über *rel*2*CP* erzwingen die Anwendung von Regeln, die eine Ersetzung durch das Nichtterminalzeichen *rel*2*CP* vornehmen. Hierbei gilt für die Selektionen, daß deren Zusammenfassung mit dem Kartesischen Produkt zu einem Join zuvor verworfen wurde. Damit kennzeichnet *rel*2*CP* eine Operationenfolge bestehend aus Kartesischem Produkt und nachfolgenden unären Operationen. Wird eine Selektion über *rel*2*CP* erkannt, wird zuerst die Anwendung der Regel geprüft, die die Zusammenfassung dieser Selektion mit dem zugrundeliegenden Kartesischen Produkt zu einem Join vornimmt. Sind die Bedingungen zum Zusammenfassen erfüllt, wird diese Regel angewandt bzw. im negativen Fall verworfen.

Das Erkennen einer binären Operation bewirkt die Durchbrechung der Operationenfolge *rel*2*CP* und erzwingt die Abbildung von *rel*2*CP* auf *rel*.

Die beschriebene Anwendung der Regeln wird über deren Kosten gesteuert:

- Kosten für Regeln vom Typ (a): 10
- Kosten für Regeln vom Typ (b): 5
- Kosten für Regeln vom Typ (c): 10
- Kosten für Regeln vom Typ (d): 3

```
label       rel2;
label       rel2CP;
...

/* (a) Nachfolgende Regeln dienen nur dem Erkennen der einzelnen Operationen der RA. */

rel2:       oSCAN2;
rel2:       oSEL2(rel2);
rel2:       oPROJ2(rel2);
rel2CP:     oCPROD2(rel2, rel2);
...

/* (b) Nachfolgende Regeln realisieren die Betrachtung einer Operationenfolge (rel2CP),
 *      nur bestehend aus SL, PJ und CP.
 */

rel2CP:     oCPROD2(rel2, rel2);
rel2CP:     oCPROD2(rel2CP, rel2);
rel2CP:     oCPROD2(rel2, rel2CP);
rel2CP:     oCPROD2(rel2CP, rel2CP);
rel2CP:     oSEL2(rel2CP);
rel2CP:     oPROJ2(rel2CP);


/* (c) Nachfolgende Regel realisiert die Durchbrechung der Operationenfolge rel2CP. */

rel:        rel2CP;


/* (d) Nachfolgende Regel fäßt eine Selektion
 *      mit einem Kartesischen Produkt zu einem Join zusammen.
 */

rel2:       oSEL2(rel2CP);
```

Abb. 33: Zusammenfassung von Selektion und Kartesischem Produkt zu einem Join

Entsprechend dieser Kosten werden die Regeln in folgender Reihenfolge angewendet:

(1) minimierende Regeln

(2) Regel, zur Zusammenfassung von Selektion und Kartesischem Produkt zu einem Join.

(3) Regeln, die die Betrachtung der Operationenfolge, Kartesisches Produkt gefolgt von Selektionen und Projektionen, erlauben.

(4) Regeln, die diese Operationenfolge durchbrechen und Regeln, die lediglich dem Erkennen der relationenalgebraischen Operationen dienen.

Die Zusammenfassung von Kartesischem Produkt mit einer bezüglich des Datenflusses direkt nachfolgenden Selektion wird nachfolgend beschrieben.

Wird ein Kartesisches Produkt im Baum erkannt, wird dessen Adresse mittels einer globalen Variablen gespeichert. Die Zusammenfassung mit einer Selektion geschieht durch das Ersetzen des Kartesischen Produktes im Baum durch einen neuen Knoten, der den Join repräsentiert. Die Zusammenfassung der Selektion mit dem Join erfordert zwar die Verschiebung der Selektion vor alle Knoten, die sich zwischen dieser und dem Kartesischen Produkt befinden, da aber die Selektion bedingungslos vor jegliche Selektion und Projektion verschoben werden kann, und außerdem kein Operatorargument neu definiert werden muß oder neue Operationen erzeugt werden, ist diese explizite Verschiebung nicht erforderlich, sondern es genügt die Entfernung der Selektionsoperation aus dem Baum.

Die Zusammenfassung der Selektionsoperation und des Kartesischen Produktes zu einem Join erzeugt keine neuen Möglichkeiten der Zusammenfassung und kann somit unter Verzicht der Guckloch-Optimierung erfolgen.

Phase 5: Verschieben der Joinoperation vor das Kartesische Produkt

Ausgangspunkt dieser Heuristik ist der in Phase 8 generierte Baum. Er ist dadurch gekennzeichnet, daß alle binären Operatoren linkstief angeordnet sind, ausgenommen derjenigen Joinoperationen, die als verteilter Join zu bearbeiten sind. Damit ist die Komplexität des Pattern-Matching-Prozesses auf die Betrachtung linearer Operationenfolgen von binären Operationen beschränkt.

Diese Phase betrachtet alle Operationenfolgen bestehend aus einem Kartesischen Produkt und den bezüglich des Datenflusses nachfolgenden Operationen Kartesisches Produkt, Join, Selektion und Projektion. Diese Operationenfolgen werden als Teilbäume separat durch Twig bearbeitet. Unter Anwendung von Transformationsregeln zum Verschieben der Projektions- und Selektionsoperation wird die Operationenfolge in eine Folge überführt, die nur noch aus Joins und Kartesischen Produkten besteht. Unter Verwendung der Transformationsregeln zum Vertauschen der Reihenfolge von Joins und Kartesischen Produkten wird nun die Realisierung der Heuristik vollzogen. Konnte im aktuellen Teilbaum die Verschiebung einer Joinoperation vor ein Kartesisches Produkt vorgenommen werden, wird im Gesamtbaum der alte durch den neuen Teilbaum ersetzt, ansonsten wird dieser verworfen.

Die Umsetzung dieser Heuristik erfolgt unter Nutzung der Guckloch-Optimierung.

Phase 6: Minimierung auf Basis der Algebra AQUAREL

Die Realisierung dieser Phase erfolgt unter Nutzung von Twig und der Verwendung der Guckloch-Optimierung.

Die Qualifizierung ist eine Eigenschaft einer Zwischenrelation und nicht die eines Operatorknotens. Deshalb soll die Qualifizierung in HEAD nicht als eine zusätzliche Eigenschaft von Operatorknoten, sondern mittels eines zusätzlichen Knotens (*FRAG*) realisiert werden. Hierbei wird während der Normalisierung für jedes Fragment zusätzlich ein Fragmentierungsknoten eingefügt. Dieser Knoten ist direkter Vorgängerknoten eines Blattknotens und beschreibt die Fragmentierungsbedingung des zugrundeliegenden Fragments. Die Bildung von Qualifizierungen erfolgt durch Verschieben des Fragmentierungsknotens zur Baumwurzel. Hierbei werden bei binären Operationen die

beiden Fragmentierungsknoten der Operanden zuvor zu einem Knoten zusammengefaßt.

Jede Verschiebung des FRAG-Knotens beinhaltet die Bildung der Qualifizierung der aktuellen Zwischenrelation. Eine Bewertung dieser Qualifizierung erfolgt aber nur, wenn die Zwischenrelation das Ergebnis einer Selektions- oder einer Joinoperation ist. Durch eine Überführung der Qualifizierung vor ihrer Bewertung in eine konjunktive Normalform verbessert sich die Chance bezüglich des Aufdeckens von Inkonsistenzen.

Wird eine Inkonsistenz festgestellt, wird der Teilbaum mit dem Fragmentierungsknoten als Wurzelknoten eliminiert und durch einen Knoten ersetzt, der die leere Menge repräsentiert. Dieser Knoten erhält als Vorgängerknoten einen Fragmentierungsknoten, dessen Qualifizierung leer ist.

Die entsprechende Regelmenge zur Realisierung der Heuristik ist in Abb. 34 dargestellt.

```
label       rel;
. . .

/* (a) Nachfolgende Regeln realisieren die Verschiebung des FRAG-Knotens an
 *     sämtliche Operationen vorbei zum Wurzelknoten. Dabei werden FRAG-Knoten
 *     über Blattknoten nicht verschoben, sondern es wird ein neuer Knoten
 *     erzeugt. Kosten der einzelnen Regeln: 5
 */

rel:        oSEL(oFRAG(oSCAN));
rel:        oJOIN(oFRAG(SCAN), oFRAG(oSCAN));
. . .
rel:        oSEL(oFRAG(rel));
rel:        oJOIN(oFRAG(rel), oFRAG(rel));
. . .
```

Abb. 34: Bildung und Bewertung der Qualifizierung von Relationen

Die Regelmenge besteht außerdem noch aus minimierenden, jedoch nicht aus erkennenden Regeln.

Die aufgeführten Regeln realisieren die Verschiebung des Knotens *FRAG* zur Baumwurzel, wobei für *FRAG*-Knoten, die direkte Vorgänger von Blattknoten sind,

keine Verschiebung des *FRAG*-Knotens vorgenommen, sondern ein neuer Knoten erzeugt wird. Damit werden die *FRAG*-Knoten über den Blattknoten erhalten. Das ist notwendig, weil ansonsten die Fragmentierungsinformation der Fragmente bei nachfolgenden Baumtraversierungen nicht mehr verfügbar wäre. Wird neben einer Regel zur Verschiebung eines Fragmentierungsknotens außerdem die Möglichkeit einer Minimierung erkannt, erfolgt die Anwendung der minimierenden Regel. Diese Regel realisiert zusätzlich zur Minimierung des Baumes eine Eliminierung der im Baum existierenden Fragmentierungsknoten mit Ausnahme derjenigen, die direkte Vorgängerknoten von Blattknoten darstellen.

Phase 7: Normalisierung

Im Ergebnis der Normalisierung werden die globalen Relationen durch ihre Fragmente ersetzt. Dabei soll in HEAD die Umsetzung der Heuristiken des verteilten und des vertikalen Joins erfolgen.

Aufgrund der möglichen Existenz von unären Knoten innerhalb des betrachteten Teilbaumes, bestehend aus zwei horizontal fragmentierten Relationen als Blattknoten und dem Join darüber als Wurzelknoten, verlangt die Heuristik des verteilten Joins, daß in einem ersten Schritt diese Joinoperationen vor die unären Operationen verschoben werden. Das geschieht, indem der betreffende Teilbaum unter Verwendung von Twig und auf der Basis von Regeln zur Verschiebung des Joins vor Selektions- und Projektionsoperationen entsprechend manipuliert und im Gesamtbaum anstelle des alten Teilbaums eingefügt wird. Die verkürzte Regelmenge hierfür ist in Abb. 35 dargestellt. Die Manipulierung des Baumes erfolgt hierbei unter Verwendung der Gucklоch-Optimierung.

Im Anschluß an diese Baummanipulation erfolgt die Normalisierung des Baumes bei gleichzeitiger Realisierung des verteilten und des vertikalen Joins. Dabei wird der Baum unter Nutzung von Twig auf der Basis der in Abb. 36 dargestellten Regelmenge traversiert und entsprechend manipuliert. Die Algorithmen zur Realisierung von verteiltem und vertikalem Join werden im Aktionsteil der entsprechenden Regeln abgearbeitet. Die Manipulation des Baumes erfolgt unter Verzicht auf die Gucklоch-Optimierung.

```
label       rel;
label       Hrel;
...
/* (a) Nachfolgende Regeln dienen nur dem Erkennen der einzelnen Operationen der RA.
 *     Kosten der einzelnen Regeln: 10
 */
rel:        oSCAN;
rel:        oSEL(rel);
...

/* (b) Nachfolgende Regel realisiert die Durchbrechung der Operationenfolge Hrel
 *     bei Auftreten einer binären Operation. Kosten der Regel: 10
 */
rel:        Hrel;

/* (c) Nachfolgende Regeln realisieren die Betrachtung einer Operationenfolge, die
 *     nur aus unären Operationen, angewandt auf eine horizontal fragmentierte
 *     Relation, besteht. Kosten der einzelnen Regeln: 8
 */
Hrel:       oSCAN;
Hrel:       oSEL(Hrel);
...

/* (d) Nachfolgende Regel erkennt einen Join, welcher direkt auf horizontale Relationen
 *     zugreift. Kosten der Regel: 5
 */
rel:        oJOIN(oSCAN, oSCAN);

/* (e) Nachfolgende Regeln verschieben einen Join vor unäre Operationen
 *     des linken Join-Unterbaumes, falls der Join über zwei horizontal
 *     fragmentierte Relationen formuliert ist. Kosten der einzelnen Regeln: 5
 */
rel:        oJOIN(oSEL(Hrel), Hrel);
...

/* (f) Die Anwendung dieser Regel ermöglicht Transformationen, die aus dem rechten
 *     Unterbaum eines horizontalen Joins die unären Operationen herausschieben.
 *     Die Regel vertauscht die Joinoperanden. Kosten der Regel: 5
 */
rel:        oJOIN(oSCAN, Hrel);
```

Abb. 35: Verschieben des Joins vor Projektion und Selektion

```
label       rel;
label       Vrel;
...
/* (a) Nachfolgende Regeln dienen nur dem Erkennen der einzelnen Operationen der RA.
 *     Kosten der einzelnen Regeln: 10
 */
rel:        oSCAN;
rel:        oSEL(rel);
...

/* (b) Nachfolgende Regel dient dem Erkennen von vertikal
 *     fragmentierten Relationen. Kosten der Regel: 5
 */
Vrel:       oSCAN;

/* (c) Nachfolgende Regeln realisieren die Betrachtung einer Operationenfolge, die nur
 *     aus unären Operationen, angewandt auf eine vertikal fragmentierte Relation,
 *     besteht. Kosten der einzelnen Regeln: 5
 */
Hrel:       oSEL(Hrel);
...
Vrel:       oSEL(Vrel);
...

/* (d) Nachfolgende Regel erkennt einen Join, welcher direkt auf horizontal fragmentierte
 *     Relationen zugreift. Im Aktionsteil der Regel erfolgt die Normalisierung der
 *     Relationen bei gleichzeitiger Realisierung des verteilten Joins. Kosten der Regel: 3
 */
rel:        oJOIN(oSCAN, oSCAN);


/* (e) Nachfolgende Regel wird angewandt beim Auftreten einer binären Operation über
 *     der Operationenfolge Vrel. Im Aktionsteil der Regel erfolgt die Normalisierung der
 *     zugrundeliegenden Relation, bei gleichzeitiger Realisierung des vertikalen Joins.
 *     Kosten der Regel: 8
 */
rel:        Vrel;


/* (f) Nachfolgende Regel nimmt die Ersetzung einer horizontal fragmentierten Relation
 *     durch ihre Frgamente vor. Kosten der Regel: 5
 */
rel:        oSCAN;
```

Abb. 36: Normalisierung inklusive Realisierung von vertikalem und verteiltem Join

Beim Erkennen eines Joins über horizontal fragmentierten Relationen wird dessen Joingraph gebildet. Ausgehend vom Joingraph wird der Fragmentierungsausdruck erzeugt, und es wird im Gesamtausdruck der Teilausdruck, bestehend aus dem Join als Wurzel, durch diesen Fragmentierungsausdruck ersetzt. Kann dagegen kein Join über einer horizontal fragmentierten Relation erkannt werden, wird diese Relation normalisiert und durch den Join aller ihrer Fragmente ersetzt.

Bei dem vertikalen Join wird für den betreffenden Teilbaum die erforderliche minimale Attributmenge ($Attr_{min}$) bezüglich der zugrundeliegenden globalen Relation ermittelt. Wenn $Attr_{min}$ identisch mit der Attributmenge der globalen Relation ist, ist die globale Relation durch den Join aller ihrer Fragmente zu ersetzen. Handelt es sich dagegen um eine echte Teilmenge, ist die Heuristik des vertikalen Joins anzuwenden. Dazu werden zunächst alle Fragmente ausgewählt, die Attribute aus $Attr_{min}$ enthalten. Für den Fall, daß $Attr_{min}$ auch Schlüsselattribute enthält, würden sich alle Fragmente qualifizieren, was den Umfang der bezüglich des vertikalen Joins zu betrachtenden Fragmentkombinationen maximal werden läßt.

Um den verlustfreien Join der Fragmente zur globalen Relation zu garantieren, sind die Schlüsselattribute der globalen Relation in allen ihren Fragmenten vertreten. Daraus folgt, daß die Attributmenge eines beliebigen Joins von Fragmenten immer das Schlüsselattribut beinhaltet. Deshalb können in $Attr_{min}$ eventuell enthaltene Schlüsselattribute entfernt werden.

Innerhalb der ausgewählten Fragmentmenge ist die minimale Teilmenge zu ermitteln, für die gilt, daß der Join darüber die erforderliche minimale Attributmenge garantiert. Der entsprechende Algorithmus, der dies realisiert, ist in Abb. 37 zu sehen.

Der Algorithmus generiert für die zu betrachtende Fragmentmenge nacheinander alle möglichen Teilmengen. Dabei beginnt er mit der Betrachtung aller Teilmengen, die aus einem Fragment bestehen, und fährt jeweils mit der Betrachtung der nächstgrößeren Teilmenge fort, falls keine Teilmenge die Bedingung bezüglich $Attr_{min}$ erfüllt.

Der Algorithmus wird beendet, wenn mindestens eine Teilmenge gefunden wurde, die die Bedingung bezüglich $Attr_{min}$ erfüllt, oder wenn die Größe der im nächsten Schritt zu generierenden Teilmenge gleich der Anzahl der Fragmente der globalen Relation ist, was in diesem Fall bedeutet, daß der Join über alle Fragmente ausgeführt werden muß.

```
minFrag()
{
    set<attribute>          Attr_min = erforderliche minimale Attributmenge;
    liste<fragment>  FRAG =  {∀ fragment : Attr( fragment) ∩ Attr_min == ∅};

    set<fragment>           actFRAG;
    set<fragment>           actFRAG2;
    set<fragment>           resultFRAG;
    set<set<fragment>       MinSets = ∅;
    int                     FragNumberOfRel, whichFrag, pass, a, b, c;

    for any F ∈ FRAG {
        if (Attr_min ⊆ Attr(F))
            MinSets = MinSets ∪ F;
    }
    pass = 0;
    FragNumberOfRel = FRAG.number;
    while ((MinSets == ∅) or (pass < FragNumberOfRel -1))
    {
        whichFrag = 1;
        while (whichFrag +  pass == FragNumberOfRel) {
            actFRAG = ∅;
            for (a = whichFrag; a < whichFrag + pass; a++)
                actFRAG = actFRAG ∪ FRAG[a];
            for (b = whichFrag + pass; b < FragNumberOfRel; b++) {
                actFRAG2 = actFRAG ∪ FRAG[b];
                for (c = b+1; c ≤ FragNumberOfRel; c++) {
                    resultFRAG = actFRAG2 ∪ FRAG[b];
                    if (Attr_min ⊆ Attr(resultFRAG))
                        MinSets = MinSets ∪ resultFRAG;
                }
            }
            if (pass == 0)
                whichFrag = FragNumberOfRel;
            else
                whichFrag++;
        }
        pass++;
    }
}
```

Abb. 37: Ermittlung der minimalen Fragmentmenge für den vertikalen Join

Dieses Vorgehen garantiert das Finden der die Bedingung erfüllenden, minimalen Fragmentmenge, wobei gleichzeitig die Anzahl der generierten Teilmengen minimal gehalten wird. Außerdem wird die Generierung identischer Teilmengen vermieden.

Sollten mehrere minimale Teilmengen durch den Algorithmus erkannt worden sein, wird letztendlich diejenige Teilmenge unter ihnen für den vertikalen Join ausgewählt, die die kleinste Attributmenge repräsentiert.

Abschließend wird der die globale Relation repräsentierende Knoten durch den Join der ermittelten Fragmente ersetzt.

Phase 8: Verschieben von Joins zu horizontal fragmentierten Relationen

Innerhalb des Baumes sind alle Operationenfolgen, die aus den Operationen Join, Kartesisches Produkt, Selektion und Projektion bestehen, darauf zu untersuchen, ob für einen Join innerhalb dieser Folge zwei Relationen existieren, die die Behandlung dieses Joins als verteilten Join unterstützen. Im positiven Fall wird der kleinste Teilbaum ermittelt für den gilt, daß er diese Relationen und den betreffenden Join enthält. Dieser Teilbaum ist durch Twig zu bearbeiten. Unter Verwendung der entsprechenden Regeln verschiebt Twig alle unären Operationen derart, daß reine Folgen von Joins und Kartesischen Produkten entstehen. Der auf diese Weise modifizierte Teilbaum wird wiederum durch Twig bearbeitet. Unter Anwendung von Transformationsregeln zum Vertauschen der Reihenfolge von Join und Kartesischem Produkt und zum Vertauschen von Operanden wird versucht, diesen Teilbaum derart zu manipulieren, daß man im Ergebnis den Join mit den horizontalen Relationen als Operanden erhält. Dabei ist die Existenz von Projektions- und Selektionsoperationen über den globalen Relationen erlaubt.

Durch die linkstiefe Ausrichtung der binären Operationen im Operatorbaum ist die Komplexität des Pattern-Matching-Prozesses auf die Betrachtung linearer Folgen von Joins und Kartesischen Produkten beschränkt. Mit Ausnahme der abschließenden Transformation, die als Ergebnis den Join mit den horizontalen Relationen als Operanden erzeugt, erhalten alle angewendeten Transformationen diese Struktur.

War die abschließende Transformation erfolgreich, d.h., sind die beiden Relationen jetzt Operanden des Joins, wird im Gesamtbaum der alte durch den neuen Teilbaum ersetzt.

Im negativen Fall wird der generierte Teilbaum verworfen, und es erfolgt keine Modifikation des Gesamtbaumes.

Das Ergebnis ist ein Baum, bei dem die rechten Operanden sämtlicher binärer Operationen ausschließlich eine globale Relation oder einen Join über zwei globale Relationen repräsentieren. Die globalen Relationen dürfen hierbei durch Selektions- oder Projektionsoperationen reduziert sein.

Die Umsetzung dieser Heuristik erfolgt unter Verwendung der Guckloch-Optimierung.

Phase 9: Faktorisierung

Diese Phase besteht aus folgenden drei Teilen:

- Suche nach gemeinsamen Teilausdrücken
- Zusammenfassung gleicher Teilausdrücke mittels Minimierung
- Zusammenfassung gleicher Teilausdrücke mittels des DUP-Operators.

Die Suche nach gleichen Teilausdrücken wird in HEAD durch separaten Code, ohne die Verwendung des Werkzeugs Twig, realisiert. Der entsprechende Algorithmus ist in Abb. 38 beschrieben. Der Algorithmus traversiert den Algebrabaum bottom up und bildet dabei Gruppen von *potentiell gleichen* Teilausdrücken. Es gilt: Zwei Teilbäume sind *potentiell gleich*, wenn alle direkten Nachfolgerknoten der Wurzel eines jeden Teilbaumes identische Knoten besitzen, wobei in einem Teilbaum mindestens ein Nachfolgerknoten existieren muß, der identisch zu einem Nachfolgerknoten des anderen Teilbaumes ist. Als Ausgangssituation sind alle Blattknoten in einer solchen Gruppe zusammengefaßt.

Jede *potentiell gleiche* Gruppe wird separat auf Gleichheit untersucht und es werden Gruppen von identischen Ausdrücken gebildet. Anschließend wird für die Knoten einer jeden Gruppe geprüft, ob deren direkte Vorgängerknoten *potentiell gleiche* Teilausdrücke darstellen. Der Algorithmus wird beendet, wenn keine neuen *potentiell gleichen* Teilausdrücke erkannt werden können bzw. der Wurzelknoten erreicht wurde.

```
set<set<nodes>> Nodes = collect(alle Blattknoten)

SearchEqualSubtrees(set<set<nodes>> Nodes)
{
    set<set<nodes>> EqualNodes = ∅;
    for any collect ∈ Nodes {
        set<set<nodes>> GroupNodes = ∅;
        for any node ∈ collect {
            GroupNodes.insert(node);
            /* ordnet node in GroupNodes ein entsprechend ≡EQ */
        }
        for any group ∈ GroupNodes {
            if (group.NodeAnzahl ≥ 2) {
                for any node ∈ collect {
                    node.ExistIdentNode = TRUE;
                }
                EqualSet = group;
                /* nodes aus group bilden eine Menge ident. Knoten (EqualSet) */
                EqualNodes = EqualNodes ∪ group;
            }
        }
    }
    if (EqualNodes ≠ ∅) {
        set<set<nodes>> AllFathers = ∅;
        for any collect ∈ CollectEqualNodes {
            set<nodes> FathersIdentChildren = ∅;
            for any node ∈ collect {
                if (    (father.state ≠ ready) ∧
                        (∀ children: children.ExistIdentNode == TRUE))
                {
                    father.state = ready;
                    FathersIdentChildren = FathersIdentChildren ∪ father;
                }
            }
            if (FathersIdentChildren.NodeAnzahl ≥ 2)
                AllFathers = AllFathers ∪ FathersIdentChildren;
        }
        if (AllFathers ≠ ∅) {
            SearchEqualSubtrees(AllFathers);
        }
    }
}
```

Abb. 38: Suchalgorithmus für gemeinsame Teilausdrücke

Im Ergebnis liegen Mengen identischer Knoten (*EqualSet*) vor. Diese repräsentieren Wurzelknoten identischer Teilbäume.

Abb. 39 beschreibt, wann für zwei Knoten *nodeA* und *nodeB* gilt, daß diese gleich sind ($nodeA \equiv_{EQ} nodeB$). Da der Baum bottom up traversiert wird, gelten zwei Knoten als gleich, wenn deren direkte Nachfolgerknoten bereits als identisch erkannt wurden. Hierbei kann für diejenigen binären Operatoren, für deren Operanden das Kommutativgesetz gilt, beim Test auf Gleichheit ein Tausch der Operanden vorgenommen werden.

Im Anschluß an die Ermittlung gleicher Teilausdrücke wird versucht, diese unter Anwendung von Minimierungsregeln zusammenzufassen. Dies geschieht unter Verwendung von Twig und basierend auf der Menge der minimierenden Regeln. Bei dem Versuch der Zusammenfassung von Teilausdrücken, gehen die zuvor gewonnenen Informationen über gleiche Teilausdrücke mit ein. Im Kostenteil aller Regeln, die eine Zusammenfassung von Teilausdrücken erlauben, wird vor deren Anwendung geprüft, ob es sich bei den Operanden um identische Teilausdrücke handelt. Die Minimierung des Baumes erfolgt durch Guckloch-Optimierung.

Die Suche nach gleichen Teilausdrücken und die anschließende Minimierung werden wiederholt angewendet bis keine Minimierungen mehr möglich sind. Ist dies der Fall, wird zum Abschluß dieser Phase versucht, noch existierende gleiche Teilausdrücke mittels des DUP-Operators zusammenzufassen.

Durch die Einführung des DUP-Operators wird aus dem Operatorbaum ein GAG. Es wurde festgelegt, daß die Optimierung des GAG in die separate Betrachtung der durch den DUP-Knoten entstandenen Teilbäume zu zerlegen ist. Aus diesem Grund wird der GAG im algebraischen HEAD-Optimierer derart auf eine Menge von Bäumen abgebildet, daß zusammengefaßte gemeinsame Teilausdrücke einen Baum mit dem Knoten DUP_{root} als Wurzel und der restliche Ausdruck einen Baum, in dem der DUP_{root}-Baum durch den Knoten DUP_{leaf} repräsentiert wird, bilden. Der DUP_{root}-Knoten enthält hierbei Verweise auf seine DUP_{leaf}-Knoten und diese wiederum enthalten einen Eintrag bezüglich ihres DUP_{root}-Knotens.

Die Einführung des DUP-Operators erfolgt gemäß dem Algorithmus *TreeToGraph* aus Abb. 40. Die durch den Algorithmus *SearchEquSubtree* gebildeten Mengen identischer

Knoten (*EqualSet*) werden entsprechend der Größe der durch sie repräsentierten Teilausdrücke nacheinander abgearbeitet. Alle der durch die Knoten des jeweils aktuellen *EqualSet* repräsentierten Teilbäume werden eliminiert und durch einen DUP_{leaf}-Knoten ersetzt. Anschließend wird ein neuer Baum erzeugt, der den gemeinsamen Teilausdruck darstellt und als Wurzelknoten DUP_{root} hat.

$nodeA \equiv_{EQ} nodeB \Leftrightarrow Equal == TRUE$

```
TestAufIdentität≡EQ(node nodeA, node nodeB)
{
    Bool Equal = FALSE;
    if (nodeA.Op == nodeB.Op) {
        switch (Operator von nodeA und nodeB) {
            case SCAN: {
                if (nodeA.rel == nodeB.rel)
                Equal = TRUE;
            }
            case ∈ {PROJ,SEL,GB}: {
                if (nodeA.ChildOp == nodeB.ChildOp)
                    Equal = TRUE;
            }
            case ∈ {JN,SJ,CP,UN,IN,DF,DV}: {
                if (    nodeA.LChildOp == nodeB.LChildOp and
                    nodeA.RChildOp == nodeB.RChildOp ) {

                    Equal = TRUE;
                }
                else {
                    if (Operator ∈ {JN,CP,UN,IN}) {
                        if (    nodeA.LChildOp == nodeB.RChildOp and
                            nodeA.RChildOp == nodeB.LChildOp ) {

                            Equal = TRUE;
                        }
                    }
                }
            }
        }
    }
}
```

Abb. 39: Test auf Identität zweier Knoten

```
set<set<nodes>> SetEquS = all EqualSet;
node treeRoot; /* enthält die Wurzel des Baumes */

TreeToGraph(set<set<nodes>> SetEquS, node treeRoot)
{
    set<nodes> maxEqualSet;
    while (∃ EqualSet: EqualSet ∈ SetEquS mit size ≥ SetEquS.MinSizeDUPExpr) {
        maxEqualSet = SetEquS.maxEquSet;
        for someone node ∈ maxEqualSet {
            new dup_root.tree = node.Subtree;
            /* Der erzeugte Baum hat dup_root als Wurzelknoten. */
        }
        for any node ∈ maxEqualSet {
            treeRoot.replaceSubtree(node.Subtree, dup_leaf);
            /*  Der Teilbaum node.subtree wird im Baum treeRoot durch
             *  dup_leaf ersetzt.
             */

            for any treenode of node.Subtree {
                if (treenode ∈ someone EqualSet in SetEquS) {
                    EqualSet = EqualSet - treenode;
            }
            delete node.Subtree;
        }
    }
}
```

Abb. 40: Zusammenfassung von Teilausdrücken mittels des DUP-Operators

Da die Zusammenfassung von gleichen Teilausdrücken mittels des DUP-Operators die Möglichkeiten der Optimierung beschränkt, sind nur ausreichend große Teilausdrücke (> *MinSizeDUPExpr*) zusammenzufassen. Da ein VDBS die Ausnutzung der inhärenten Parallelität erlaubt, wird nicht die Anzahl der Operationen eines Teilbaumes, sondern dessen maximale Baumtiefe als Größenkriterium verwendet. Die Festlegung von *MinSizeDUPExpr* muß hierbei auf noch zu sammelnde Erfahrungswerte beruhen.

Aufgrund des höheren Bearbeitungsaufwandes von binären Operationen ist eine unterschiedliche Wichtung von unären und binären Operationen empfehlenswert. Diese Wichtung kann dadurch realisiert werden, daß bei Ermittlung der Baumtiefe binäre Operationen nicht mit einer, sondern mit zwei Baumebenen ins Ergebnis eingehen.

Phase 10: Aufnahme weiterer Knoten in DUP-Bäume

Die Realisierung dieser Phase erfolgt teils durch separaten Code, teils unter Verwendung von Twig.

Zunächst werden aus der Menge der unären Nachfolgerknoten des DUP-Operators Mengen identischer unärer Knoten gebildet. Beginnend mit der mächtigsten Menge wird versucht, die identischen Knoten derart zu verschieben, daß sie zu direkten Vorgängerknoten des DUP_{leaf}-Knotens werden. Dazu ist für jeden dieser identischen Knoten der Teilbaum, der aus dem DUP_{leaf}-Operator als Blattknoten und dem identischen Knoten als Wurzel besteht, zu manipulieren.

Mittels von Twig generiertem Code, der auf Regeln zur Änderung der Reihenfolge von unären Operationen beruht, wird versucht, die betreffenden identischen Knoten innerhalb ihrer Teilbäume zum jeweiligen DUP_{leaf}-Knoten zu verschieben. Das erfolgt unter Nutzung der Guckloch-Optimierung. Ist diese Transformation für mindestens zwei identische Knoten erfolgreich, ohne daß es zu einer Änderung ihrer Operatorargumente kam, werden deren alte Teilbäume durch die manipulierten Teilbäume ersetzt. Im Anschluß daran werden die identischen Knoten eliminiert und als ein Knoten in den entsprechenden DUP_{root}-Baum aufgenommen. Die Realisierung der Heuristik zur Aufnahme weiterer Knoten in DUP-Bäume ist in Abb. 41 dargestellt. Der Algorithmus wird beendet, wenn es für keinen DUP-Operator mehr möglich ist, identische unäre Knoten zu selbigem zu verschieben und dort zusammenzufassen.

Zum Abschluß dieser Phase erfolgt die Berücksichtigung direkter binärer Vorgängerknoten von DUP_{leaf}-Knoten. Kann eine Identität zwischen diesen festgestellt werden, erfolgt die Aufnahme in den entsprechenden DUP_{root}-Baum. leaf

```
node treeRoot; /* enthält die Wurzel des Operatorbaumes */

addNodeToDUPTree(node treeRoot)
{
    Bool extended;
    set<nodes> maxEquSet;
    set<set<nodes>> GroupNodes;
    for any Knoten dup_root {
        do {
            GroupNodes = ∅;
            for any dup_leaf von dup_root {
                node = dup_leaf.Vorgänger;
                while (node.Typ == unär ∧ node <> treeRoot) {
                    GroupNodes.insert(node),
                    /* ordnet node in GroupNodes entsprechend ≡_EQ ein */

                    node = node.Vorgänger;
                }
            }
            greaterDUPTree = FALSE;
            while ((∃ maxEquSet: maxEquSet = GroupNodes.collect mit
                                    collect.NodeNumber ≥ 2 ∧
                                    collect.NodeNumber == GroupNodes.max) ∧
                    ¬(extended = ExpandedDUPTree(maxEquSet, dup_root, treeRoot)))
            {
                GroupNodes = GroupNodes - maxEquSet;
            }
        } while (greaterDUPTree);
    }
}


Bool ExpandedDUPTree(set<nodes> EquSet, nodeDUP dup_root, node treeRoot)
{
    set<nodes> maxEquSet = ∅;
    set<set<nodes>> GroupNodes = ∅;
    for any node ∈ EquSet {
        node.tmpSubtree = matchTwig(node);
        /*  Verschieben des Knoten node direkt vor dessen dup_leaf unter
         *  Nutzung von Twig. Bei Erfolg setzen von succes auf TRUE.
         *  tmpSubtree enthält den derartig manipulierten Teilbaum
         */
        if (¬ succes) {
            EquSet = EquSet - node;
            if (EquSet.NodeNumber < 2)
                return FALSE;
        }
    }
```

```
        for any node ∈ EquSet {
                GroupNodes.insert(node);
                /* ordnet node in GroupNodes entsprechend ≡_EQ ein */
        }
        if (∃ maxEquSet: maxEquSet = GroupNodes.collect mit
                        collect.NodeNumber ≥ 2   ∧
                        collect.NodeNumber == GroupNodes.max) {
                for any node ∈ maxEquSet {
                        treeRoot.replaceSubtree(node.Subtree, node.tmpSubtree);
                        /*      Im Baum treeRoot ersetzen des Teilbaums mit der Wurzel node
                         *      durch die mittels Twig generierten Teilbäume node.tmpSubtree.
                         */
                }
                if (dup_root. Number_dup_leaf == maxEquSet.NodeNumber) {
                        for someone node ∈ maxEquSet {
                                dup_root.insert(node);
                                /*      Fügt den Knoten node in den Baum mit der Wurzel
                                 *      dup_root als direkten Nachfolger der Wurzel ein.
                                 */
                        }
                        for any node ∈ maxEquSet {
                                treeRoot.deleteNode(node);
                        }
                }
                else {
                        for any dup_leaf von dup_root {
                                treeRoot.replaceSubtree(dup_leaf, dup_root. tree);
                        }
                        delete dup_root.tree;
                        for someone node ∈ maxEqualSet {
                                new dup_root.tree = node.Subtree;
                                /* Erzeugen eines Baumes mit der Wurzel dup_root */
                        }
                        for any node ∈ maxEqualSet {
                                new dup_root.dup_leaf;
                                treeRoot.replaceSubtree(node.Subtree, dup_root. dup_leaf);
                        }
                }
                return TRUE;
        }
        else    { return FALSE; }
}
```

Abb. 41: Aufnahme weiterer unärer Knoten in einen DUP-Baum

6. Ausblick

Im Rahmen dieser Arbeit erfolgte der Entwurf, die Beschreibung der Realisierung sowie bereits die teilweise Implementierung des algebraischen HEAD-Optimierers. Aufgrund der Komplexität des Themas konnten jedoch nicht alle Probleme behandelt werden, die mit der Realisierung des algebraischen HEAD-Optimierers in Zusammenhang stehen. Diese werden nachfolgend beschrieben.

Die Aufstellung der einzelnen Heuristiken des algebraischen Optimierers sowie deren Anwendungsreihenfolge innerhalb einer Optimiererkomponente basieren auf Erfahrungswerten. Die in der Literatur aufgeführten und für den HEAD-Optimierer nutzbaren Erfahrungswerte sind jedoch beschränkt. Ziel weiterführender Arbeiten muß es deshalb sein, unter Nutzung der existierenden HEAD-Kostenfunktion eigene Erfahrungswerte zu sammeln, die es ermöglichen, aus den sich bietenden Alternativen den erfolgversprechendsten Optimierer auszuwählen. Bezüglich nachfolgend aufgeführter Punkte sind Entscheidungen zu treffen:

- Die Anwendungsreihenfolge der Heuristiken läßt mehrere erfolgversprechende Alternativen zu. Diese basieren vor allem auf dem günstigsten Zeitpunkt für die Suche und Zusammenfassung gemeinsamer Teilausdrücke. Hierbei ist nur die Zusammenfassung mittels des DUP-Operators problematisch. Verschiedene Alternativen betreffs der Einführung des DUP-Operators wurden aufgezeigt. Ziel zukünftiger Arbeiten muß es sein aus der Menge der möglichen Alternativen den besten Optimierer auszuwählen. Des weiteren können Vorgaben der internen Optimierung die Nichtanwendung von Heuristiken notwendig machen. Für diese Fälle ist ebenfalls die optimale Reihenfolge der zu realisierenden Heuristiken zu ermitteln.
- Da die Einführung des DUP-Operators die Möglichkeiten der Minimierung und Optimierung reduziert, ist die Zusammenfassung auf entsprechend große Teilausdrücke zu beschränken. Dies soll gewährleisten, daß der Nutzen des DUP-Operators seine Nachteile überwiegt. Aus diesem Grund ist die Ermittlung einer unteren Grenze für eine Zusammenfassung von gleichen Teilausdrücken erforderlich.

Das Größenkriterium basiert auf den Erwartungen bezüglich der Bearbeitungskosten des Teilausdrucks und somit deren Reduzierung, wenn dieser nur einmal berechnet werden muß. Die Festlegung dieses Grenzwertes muß deshalb in Zusammenhang mit der Anzahl der Teilausdrücke, die zusammengefaßt werden können, untersucht werden. Außerdem ist bei der Ermittlung der Größe die unterschiedliche Wichtung von in diesem Teilausdruck vorkommenden Operationen zu untersuchen. Selektionen verursachen beispielsweise geringere Bearbeitungskosten als ein Join.

Nicht berücksichtigt wurde in dieser Arbeit das innerhalb der algebraischen Optimierung auftretende Problem des Domain mismatch. Weiterführende Arbeiten müssen sich deshalb mit der Lösung nachfolgend aufgeführter Probleme beschäftigen:

- Bei der Bewertung der Qualifizierung einer Relation wird bisher angenommen, daß, falls Attribute miteinander verglichen werden, diese den gleichen Wertebereich haben. Dies kann aber nicht vorausgesetzt werden. Es ist mit diesem Problem in DBS sogar verstärkt zu rechnen, da die Beschränkung von Wertebereichen der Durchsetzung von Integritätsbedingungen dient.
- Bei der Manipulation des Operatorbaumes können Namenskonflikte auftreten. Namenskonflikte entstehen unter anderem aufgrund unterschiedlicher Attributnamen in den Relationen und angewandten binären Operationen darüber. Viele Transformationsregeln sind an Bedingungen bezüglich der Attributmenge von Relationen und Operatorargumenten gebunden und diese erfordern die aktuelle Namensausprägung der Attribute. Die Umbenennungsoperation muß deshalb berücksichtigt und durch Bereitstellung von entsprechenden Transformationsregeln in die algebraische Optimierung integriert werden.

Eine Implementierung der verschiedenen Optimierungsphasen wurde bereits teilweise im Rahmen dieser Arbeit vorgenommen. So erfolgte die Implementierung der Phase des Suchens und des Zusammenfassens gleicher Teilausdrücke. Weiterhin erfolgte die Implementierung der Phase zum Zerlegen von Selektionsoperationen und für ausgewählte Operationen die Implementierung der Phasen zum Verschieben von Selektion und Pro-

jektion zu den Blattknoten sowie die Implementierung der in allen Regeln adäquat zu realisierenden minimierenden Regeln. Für die möglichen Baumtransformationen wurden Funktionen bereitgestellt. Weiterführende Implementierungsarbeiten müssen die Realisierung der verbleibenden Optimierungsphasen sowie die Betrachtung aller Operationen der erweiterten Relationenalgebra zum Inhalt haben. Außerdem sollte mit einer Funktionsbibliothek zur Bildung von Attributmengen aus Relationen und Operatorargumenten die Erweiterung der Regelmenge erleichtert werden.

Die Phase zum Suchen und Zusammenfassen gemeinsamer Ausdrücke wurde nicht mit Twig implementiert. Sie wurde mittels separatem Code realisiert und basiert auf dem spezifischen HEAD-Operatorbaum. Um Unabhänigkeit von den Spezifika des HEAD-Operatorbaumes zu erhalten, ist es notwendig, Funktionen wie sie in Twig verwendet werden und die die Behandlung der Knoten des Operatorbaumes als abstrakten Datentyp ermöglichen, zu nutzen.

Twig erlaubt gegenwärtig die Generierung ausschließlich einer Optimiererkomponente. Bei Realisierung mehrerer Phasen erfolgt hierbei innerhalb einer Phase durch den Pattern-Matcher die Betrachtung der gesamten Regelmenge, obwohl nur die Anwendung der Regeln der aktuellen Phase möglich ist. Da der Quellcode von Twig öffentlich ist, ist zu prüfen, inwiefern durch eine Änderung des Quellcodes die Generierung unabhängiger Optimiererkomponenten möglich ist. Jede Optimiererkomponente würde in diesem Fall eine Phase realisieren. Die damit bewirkte Separierung der Regelmenge erleichtert außerdem die Lesbarkeit und Wartbarkeit der Twig-Spezifikation.

ASU88.

Aho, A.V., Sethi, R., and Ullman, J.D., *Compilerbau*, Addison-Wesley Publ. Comp., Bonn (1988).

AHY79.

Apers, P. M. G., Hevner, A. R., and Yao, S. B., "Optimization Algorithm for Distributed Queries," *IEEE Trans. on Software Eng.*, 5, 3, pp. 57-68 (May 1979).

BC81.

Bernstein, P.A. and Chiu, D.W., "Using Semi-Joins to Solve Relational Queries," *Journal of the Association for Computing Machinery*, 28, 1, pp. 25-40 (January 1981).

BGW+81.

Bernstein, P.A., Goodman, N., Wong, E., Reeve, C.L., and Rothnie, J.B., "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems*, 6, 4, pp. 602-625 (December 1981).

BMG93.

Blakeley, J.A., McKenna, W.J., and Graefe, G., "Experiences Building the Open OODB Query Optimizer," *Proc. ACM SIGMOLD Conf.*, Washington, DC (Mai 1993).

CP85.

Ceri, S. and Pelagatti, G., "Distributed Databases: Principles and Systems," *McGraw-Hill*, New York (1985).

CGP86.

Ceri, S., Gottlob, G., and Pelagatti, G., "Taxonomy and formal properties of distributed joins," *Information Systems*, 1, 1 (1986).

CY90.

Chen, M. and Yu, P.S., *Using combination of join and semijoin operations for distributed query processing*, IBM Thomas J. Watson Research Center (1990).

CY92.

Chen, M.-S., Yu, P. S., and Wu, K.-L., “Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries,” *IEEE*, 7 (July 1992).

CY94.

Chen, M.S. and Yu, S., “A Graph Theoretical Approach to Determine a Join Reducer Sequence in Distributed Query Processing,” *IEEE Transactions on Knowledge and Data Engineering*, 6, 1 (February 1994).

CH80.

Chiu, D.M. and Ho, Y.C., “A method for interpreting tree queries into optimal semi-join expressions” in *ACM SIGMOD Int. Conf. on Management of Data* (1980).

CBH84.

Chiu, D.M., Bernstein, P.A., and Ho, Y.C., “Optimizing chain queries in distributed database system,” *SIAM J. Sci. Stat. Comput.*, 13, 1 (1984).

CH82.

Chu, W.W. and Hurley, P., “Optimal Query Processing for Distributed Database Systems,” *IEEE Transaction on Computers*, C-31, 9 (September 1982).

Fla93.

Flach, G., “Konzeption eines Kostenmodells für ein verteiltes Datenbanksystem,” *Diplomarbeit*, Universität Rostock, Fachbereich Informatik, Rostock (Mai 1993).

FLM93.

Flach, G., Langer, U., and Meyer, H., “Kostenbasierte Anfrageoptimierung in HEAD,” *Rostocker Informatik-Berichte*, 15, Universität Rostock, Fachbereich Informatik (1994).

GM93.

Graefe, G. and McKenna, B., “The Volcano Optimizer Generator: Extensibility and Efficient Search,” *IEEE Conf. on Data Eng.*, p. pp.209, Vienna (Apr. 1993).

Gru94.

Grust, T., “Entwurf und Implementierung der internen Ebene von OSCAR: Optimierung und Auswertung der Objektalgebra ABRAXAS,” *Diplomarbeit*,

Technische Universität Clausthal, Institut für Informatik, Clausthal (September 1994).

Heu92.

Heuer, A., *Objektorientierte Datenbanken, Konzepte, Modelle, Systeme,* Addison-Wesley, Bonn (1992).

KK93.

Kandzia, P. and Klein, H.J., *Theoretische Grundlagen relationaler Datenbanksysteme,* Wissenschaftsverlag, Mannheim (1993).

Kil90.

Kilger, C., "Regelbasierte Strategien zur optimalen Parallelisierung relationaler Anfragen," *Diplomarbeit,* Universität Karlsruhe, Karlsruhe (1990).

Kru94.

Kruse, H.-H., "Entwurf und Implementierung einer relationalen Algebramaschine," *Diplomarbeit,* Universitaẗ Rostock, Fachbereich Informatik, Rostock (1994).

Leh88.

Lehnert, K., "Regelbasierte Beschreibung von Optimierungsverfahren für relationale Datenbankabfragesprachen," *Dissertation,* Technische Universität München, München (1988).

LR91.

Lemmen, T. and Rudolph, C., "Abschlußbericht: RELAX-Optimierung," *Softwarepraktikumsbericht,* Technische Universität Clausthal, Clausthal (Dezember 1991).

Lin94.

Linke, A., "Lastmessung und -verteilung in heterogenen, verteilten Datenbanksystemen," *Diplomarbeit,* Universität Rostock, Fachbereich Informatik (Mai 1994).

ML86.

Mackert, L. F. and Lohman, G. M., "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Research Journal,* 5050, IBM Almaden

Research Center, San Jose, California (Apr. 1986).

Mck93.

McKenna, B., *Volcano Query Optimizer Generator Manual*, University of Colorado, Boulder (1993).

Mun94.

Munoz, A., "An Extensible Query Optimizer Architecture for the TIGUKAT Objectbase Management System," *Diplomarbeit*, Universität Alberta, Alberta (Frühjahr 1994).

ÖV91.

Ozsu, M.T. and Valduriez, P., "Principles of distributed database systems" in *Prentice-Hall, Inc., Englewood Cliffs* (1991).

Pap91.

Pape, K., "Ein Beitrag zur Anfrageoptimierung und Anfragezerlegung in einem heterogenen verteilten Datenbanksystem," *Diplomarbeit*, Universitaẗ Rostock, FB Informatik (1991).

PV88.

Pramanik, S. and Vineyard, D., "Optimizing Join Queries in Distributed Databases," *IEEE Transaction on Software Engineering*, 14, 9 (September 1988).

Rec93.

Rechenburg, A., "Algebraische Eigenschaften der EXTREM-Objektalgebra," *Diplomarbeit*, Technische Universität Clausthal, Institut für Informatik, Clausthal (Januar 1993).

Sau91.

Sauer, H., *Relationale Datenbanken, Theorie und Praxis*, Addison Wesley (Deutschland) GmbH, Muñchen (1991).

Sch86.

Scholl, M., "Theoretical foundation of algebraic optimization utilzing unnormalized relations," *1. International Conference on Database Theory* (1986).

Tji86.

Tjiang, S. W. K., "Twig Reference Manual," Computing Science Technical Report

#120, AT&T Bell Laboratories (1986).

Ull80.

Ullman, J. D., *Principles of Database Systems*, Computer Science Press, Potomac, Maryland (198O).

Ull88.

Ullman, J. D., *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, Maryland (1988).

Wan90.

Wang, C., "The Complexity of Processing Tree Queries in Distributed Databases," *IEEE* (1990).

WCS92.

Wang, C., Chen, L. P., and Shyu, S. C., "A Parallel Execution Method for Minimizing Distributed Query Response Time," *IEEE Transaction on Parallel and Distributed Systems*, 3, 3 (May 1992).

YC84.

Yu, C.T. and Chang, C.C., "Distributed query processing.," *ACM Computing Surveys*, 16, 4, pp. 399-433 (1984).

Abbildungsverzeichnis

Tabellenverzeichnis

Transformationsregeln

Nachfolgend wird eine Auflistung von Transformationsregeln der erweiterten Relationenalgebra des HEAD-Systems basierend auf Regeln der starken Äquivalenz vorgenommen. Dabei soll gelten: Zwei Relationen werden als äquivalent betrachtet, wenn ihre Tupel dieselbe Abbildung von Attributnamen auf deren Werte darstellen, wobei sich die Reihenfolge der Attribute in den beiden Relationen unterscheiden darf. Zwei Ausdrücke sind äquivalent, wenn man in beiden Ausdrücken identische Relationenvariablen durch dieselben Relationen ersetzt und das gleiche Ergebnis erhält.

Voraussetzung für die Anwendung einer Regel ist, daß die mit dieser Regel verknüpfte Bedingung erfüllt wird. Sollte eine Transformationsregel unabhängig von irgendwelchen Bedingungen anwendbar sein, ist dies im Bedingungsteil mit einem '-' gekennzeichnet. Des weiteren existiert für Regeln, bei denen die Transformation nicht eindeutig ist, eine Konstruktionsregel, welche die Bildung des Ergebnisses vorschreibt.

Nachfolgend werden mit R bzw. R_i Relationen, mit F bzw. F_j Join- und Selektionsbedingungen und mit sonstigen großen Buchstaben (A,B,C...) Attributmengen bezeichnet; Θ ist $\in \{<, \leq, \geq, =, \neq\}$.

Zusammenfassen und Zerlegen der Selektionsoperation

$SL_{F1}(SL_{F2}(R)) \Rightarrow SL_{F1 \wedge F2}(R)$
Bedingung: -

$SL_{F1 \wedge F2}(R) \Rightarrow SL_{F1}(SL_{F2}(R))$
Bedingung: -

Ändern der Reihenfolge unärer und der Operandenfolge binärer Operationen

Selektion:

$SL_{F1}(SL_{F2}(R)) \Rightarrow SL_{F2}(SL_{F1}(R))$

Bedingung: -

Selektion und Projektion:

$SL_F(PJ_A(R)) \Rightarrow PJ_A(SL_F(R))$

Bedingung: -

$PJ_A(SL_F(R)) \Rightarrow SL_F(PJ_A(R))$

Bedingung: $Attr(F) \subseteq A$

$PJ_A(SL_F(R)) \Rightarrow PJ_A(SL_F(PJ_B(R)))$

Bedingung: $\exists\, C\colon C = Attr(F) - A \wedge C \neq \varnothing$

Konstruktionsregel: $B = A \cup C$

Kartesisches Produkt:

$R_1\ CP\ R_2 \Rightarrow R_2\ CP\ R_1$

Bedingung: -

Join:

$R_1\ JN_F\ R_2 \Rightarrow R_2\ JN_F\ R_1$

Bedingung: -

Union:

$R_1\ UN\ R_2 \Rightarrow R_2\ UN\ R_1$

Bedingung: -

Intersection:

$R_1\ IN\ R_2 \Rightarrow R_2\ IN\ R_1$

Bedingung: -

Ändern der Reihenfolge binärer Operationen

Kartesisches Produkt:

$(R_1\ CP\ R_2)\ CP\ R_3 \Rightarrow R_2\ CP\ (R_1\ CP\ R_3)$

Bedingung: -

Join:

$(R_1\ JN_{F1}\ R_2)\ JN_{F2}\ R_3 \Rightarrow R_2\ JN_{F1}\ (R_1\ JN_{F2}\ R_3)$

Bedingung: $Attr(F2) \subseteq (Attr(R_1) \cup Attr(R_3))$

Join und Kartesisches Produkt:

$(R_1\ JN_F\ R_2)\ CP\ R_3 \Rightarrow R_2\ JN_F\ (R_1\ CP\ R_3)$

Bedingung: -

$(R_1\ CP\ R_2)\ JN_F\ R_3 \Rightarrow R_2\ CP\ (R_1\ JN_F\ R_3)$

Bedingung: $Attr(F) \subseteq (Attr(R_1) \cup Attr(R_3))$

Union:

$(R_1\ UN\ R_2)\ UN\ R_3 \Rightarrow R_2\ UN\ (R_1\ UN\ R_3)$

Bedingung: -

Intersection:

$(R_1\ IN\ R_2)\ IN\ R_3 \Rightarrow R_1\ IN\ (R_2\ IN\ R_3)$

Bedingung: -

Differenz:

$(R_1\ DF\ R_2)\ DF\ R_3 \Rightarrow (R_1\ DF\ R_3)\ DF\ R_2$

Bedingung: -

Division:

$(R_1\ DV\ R_2)\ DV\ R_3 \Rightarrow (R_1\ DV\ R_3)\ DV\ R_2$

Bedingung: -

Ändern der Reihenfolge binärer Operationen mittels Splitten von Operationen

Union und Kartesisches Produkt:

$(R_1\ UN\ R_2)\ CP\ R_3 \Rightarrow (R_1\ CP\ R_3)\ UN\ (R_2\ CP\ R_3)$

Bedingung: -

Union und Join:

$(R_1\ UN\ R_2)\ JN_F\ R_3 \Rightarrow (R_1\ JN_F\ R_3)\ UN\ (R_2\ JN_F\ R_3)$

Bedingung: -

Union und Intersection:

$(R_1\ UN\ R_2)\ IN\ R_3 \Rightarrow (R_1\ IN\ R_3)\ UN\ (R_2\ IN\ R_3)$

Bedingung: -

$(R_1\ IN\ R_2)\ UN\ R_3 \Rightarrow (R_1\ UN\ R_3)\ IN\ (R_2\ UN\ R_3)$

Bedingung: -

Union und Differenz:

$(R_1\ UN\ R_2)\ DF\ R_3 \Rightarrow (R_1\ DF\ R_3)\ UN\ (R_2\ DF\ R_3)$

Bedingung: -

Intersection und Kartesisches Produkt:

$(R_1\ IN\ R_2)\ CP\ R_3 \Rightarrow (R_1\ CP\ R_3)\ IN\ (R_2\ CP\ R_3)$

Bedingung: -

Intersection und Join:

$(R_1\ IN\ R_2)\ JN_F\ R_3 \Rightarrow (R_1\ JN_F\ R_3)\ IN\ (R_2\ JN_F\ R_3)$

Bedingung: -

Intersection und Differenz:

$(R_1\ IN\ R_2)\ DF\ R_3 \Rightarrow (R_1\ DF\ R_3)\ IN\ (R_2\ DF\ R_3)$

Bedingung: -

$(R_1\ DF\ R_2)\ IN\ R_3 \Rightarrow (R_1\ IN\ R_3)\ DF\ (R_2\ IN\ R_3)$

Bedingung: -

Intersection und Division:

$(R_1\ IN\ R_2)\ DV\ R_3 \Rightarrow (R_1\ DV\ R_3)\ IN\ (R_2\ DV\ R_3)$

Bedingung: -

Differenz und Kartesisches Produkt:

$(R_1\ DF\ R_2)\ CP\ R_3 \Rightarrow (R_1\ CP\ R_3)\ DF\ (R_2\ CP\ R_3)$

Bedingung: -

Differenz und Join:

$(R_1\ DF\ R_2)\ JN_F\ R_3 \Rightarrow (R_1\ JN_F\ R_3)\ DF\ (R_2\ JN_F\ R_3)$

Bedingung: -

Ändern der Reihenfolge binärer Operationen durch Zusammenfassen von Operationen

Union und Kartesisches Produkt:

$(R_1\ CP\ R_3)\ UN\ (R_2\ CP\ R_3) \Rightarrow (R_1\ UN\ R_2)\ CP\ R_3$

Bedingung: -

Union und Join:

$(R_1\ JN_F\ R_3)\ UN\ (R_2\ JN_F\ R_3) \Rightarrow (R_1\ UN\ R_2)\ JN_F\ R_3$

Bedingung: -

Union und Intersection:

$(R_1\ IN\ R_3)\ UN\ (R_2\ IN\ R_3) \Rightarrow (R_1\ UN\ R_2)\ IN\ R_3$

Bedingung: -

$(R_1\ UN\ R_3)\ IN\ (R_2\ UN\ R_3) \Rightarrow (R_1\ IN\ R_2)\ UN\ R_3$

Bedingung: -

Union und Differenz:

$(R_1\ DF\ R_3)\ UN\ (R_2\ DF\ R_3) \Rightarrow (R_1\ UN\ R_2)\ DF\ R_3$

Bedingung: -

Intersection und Kartesisches Produkt:

$(R_1\ CP\ R_3)\ IN\ (R_2\ CP\ R_3) \Rightarrow (R_1\ IN\ R_2)\ CP\ R_3$

Bedingung: -

Intersection und Join:

$(R_1\ JN_F\ R_3)\ IN\ (R_2\ JN_F\ R_3) \Rightarrow (R_1\ IN\ R_2)\ JN_F\ R_3$

Bedingung: -

Intersection und Differenz:

$(R_1\ DF\ R_3)\ IN\ (R_2\ DF\ R_3) \Rightarrow (R_1\ IN\ R_2)\ DF\ R_3$

Bedingung: -

$(R_1\ IN\ R_3)\ DF\ (R_2\ IN\ R_3) \Rightarrow (R_1\ DF\ R_2)\ IN\ R_3$

Bedingung: -

Intersection und Division:

$(R_1\ DV\ R_3)\ IN\ (R_2\ DV\ R_3) \Rightarrow (R_1\ IN\ R_2)\ DV\ R_3$

Bedingung: -

Differenz und Kartesisches Produkt:

$(R_1\ CP\ R_3)\ DF\ (R_2\ CP\ R_3) \Rightarrow (R_1\ DF\ R_2)\ CP\ R_3$

Bedingung: -

Differenz und Join:

$(R_1\ JN_F\ R_3)\ DF\ (R_2\ JN_F\ R_3) \Rightarrow (R_1\ DF\ R_2)\ JN_F\ R_3$

Bedingung: -

Verschieben von unären Operationen vor eine binäre Operation

Projektion und Kartesisches Produkt:

$PJ_A(R_1\ CP\ R_2) \Rightarrow PJ_A(R_1)\ CP\ R_2$

Bedingung: $A \subseteq Attr(R_1)$

$PJ_A(R_1\ CP\ R_2) \Rightarrow PJ_B(R_1)\ CP\ PJ_C(R_2)$

Bedingung: -

Konstruktionsregel: $B = A - Attr(R_2)\ \wedge\ C = A - Attr(R_1)$

Projektion und Join:

$PJ_A(R_1\ JN_F\ R_2) \Rightarrow PJ_A(R_1)\ JN_F\ R_2$

Bedingung: $(Attr(F) \cap Attr(R_1)) \subseteq A\ \wedge\ A \subseteq Attr(R_1)$

$PJ_A(R_1\ JN_F\ R_2) \Rightarrow PJ_B(R_1)\ JN_F\ PJ_C(R_2)$

Bedingung: $Attr(F) \subseteq A$

Konstruktionsregel: $B = A - Attr(R_2)\ \wedge\ C = A - Attr(R_1)$

$PJ_A(R_1\ JN_F\ R_2) \Rightarrow PJ_A(PJ_B(R_1)\ JN_F\ PJ_C(R_2))$

Bedingung: -

Konstruktionsregel: $B = (A \cup Attr(F)) - Attr(R_2)\ \wedge$

$C = (A \cup Attr(F)) - Attr(R_1)$

Projektion und Semijoin:

$PJ_A(R_1\ SJ_F\ R_2) \Rightarrow PJ_A(R_1)\ SJ_F\ R_2$

Bedingung: $(Attr(F) \cap\ Attr(R_1)) \subseteq A$

$PJ_A(R_1\ SJ_F\ R_2) \Rightarrow PJ_B(R_1)\ SJ_F\ PJ_C(R_2)$

Bedingung: $Attr(F) \subseteq A$

Konstruktionsregel: $B = A - Attr(R_2)\ \wedge\ C = Attr(R_2) \cap Attr(F)$

$PJ_A(R_1\ SJ_F\ R_2) \Rightarrow PJ_A(PJ_B(R_1)\ SJ_F\ PJ_C(R_2))$

Bedingung: -

Konstruktionsregel: $B = (A \cup Attr(F)) - Attr(R_2)\ \wedge\ C = (A \cup Attr(F)) - Attr(R_1)$

Projektion und Union:

$PJ_A(R_1\ UN\ R_2) \Rightarrow PJ_A(R_1)\ UN\ PJ_A(R_2)$

Bedingung: -

Selektion und Kartesisches Produkt:

$SL_F(R_1\ CP\ R_2) \Rightarrow SL_F(R_1)\ CP\ R_2$

Bedingung: $Attr(F) \subseteq Attr(R_1)$

$SL_F(R_1\ CP\ R_2) \Rightarrow SL_{F1}(R_1)\ CP\ SL_{F2}(R_2)$

Bedingung: $\exists F_h, F_i: ((F = F_h \wedge F_i)\ \wedge$

$Attr(F_h) \subseteq Attr(R_1)\ \wedge\ Attr(F_i) \subseteq Attr(R_2))$

Konstruktionsregel: $F_1 = F_h,\ F_2 = F_i$

Selektion und Join:

$SL_F(R_1\ JN_{F1}\ R_2) \Rightarrow SL_F(R_1)\ JN_{F1}\ R_2$

Bedingung: $Attr(F) \subseteq Attr(R_1)$

$SL_F(R_1\ JN_{F3}\ R_2) \Rightarrow SL_{F1}(R_1)\ JN_{F3}\ SL_{F2}(R_2)$

Bedingung: $\exists F_h, F_i: ((F = F_h \wedge F_i)\ \wedge$

$Attr(F_h) \subseteq Attr(R_1)\ \wedge\ Attr(F_i) \subseteq Attr(R_2))$

Konstruktionsregel: $F_1 = F_h$, $F_2 = F_i$

Selektion und Semijoin:

$SL_F(R_1\ SJ_{F1}\ R_2) \Rightarrow SL_F(R_1)\ SJ_{F1}\ R_2$

Bedingung: -

Selektion und Union:

$SL_F(R_1\ UN\ R_2) \Rightarrow SL_F(R_1)\ UN\ SL_F(R_2)$

Bedingung: -

Selektion und Differenz:

$SL_F(R_1\ DF\ R_2) \Rightarrow SL_F(R_1)\ DF\ SL_F(R_2)$

Bedingung: -

$SL_F(R_1\ DF\ R_2) \Rightarrow SL_F(R_1)\ DF\ R_2$

Bedingung: -

Selektion und Intersection:

$SL_F(R_1\ IN\ R_2) \Rightarrow SL_F(R_1)\ IN\ SL_F(R_2)$

Bedingung: -

$SL_F(R_1\ IN\ R_2) \Rightarrow SL_F(R_1)\ IN\ R_2$

Bedingung: -

Selektion und Division:

$SL_F(R_1\ DV\ R_2) \Rightarrow SL_F(R_1)\ DV\ R_2$

Bedingung: -

GROUP und Union:

$GB_{G,AF}(R_1\ UN\ R_2) \Rightarrow GB_{G,AF}(R_1)\ UN\ GB_{G,AF}(R_2)$

Bedingung: $\forall i, j: (G_i \subseteq R_j\ \vee\ (G_i \cap R_j) = \varnothing)$

Verschieben einer binären Operation vor unäre Operationen

Projektion und Kartesisches Produkt:

$PJ_A(R_1)\ CP\ R_2 \Rightarrow PJ_A(R_1\ CP\ R_2)$

Bedingung: -

$PJ_A(R_1)\ CP\ PJ_B(R_2) \Rightarrow PJ_C(R_1\ CP\ R_2)$

Bedingung: -

Konstruktionsregel: $C = A \cup B$

Projektion und Join:

$PJ_A(R_1)\ JN_F\ R_2 \Rightarrow PJ_A(R_1\ JN_F\ R_2)$

Bedingung: -

$PJ_A(R_1)\ JN_F\ PJ_B(R_2) \Rightarrow PJ_C(R_1\ JN_F\ R_2)$

Bedingung: -

Konstruktionsregel: $C = A \cup B$

Projektion und Semijoin:

$PJ_A(R_1)\ SJ_F\ R_2 \Rightarrow PJ_A(R_1\ SJ_F\ R_2)$

Bedingung: -

$PJ_A(R_1)\ SJ_F\ PJ_B(R_2) \Rightarrow PJ_A(R_1\ SJ_F\ R_2)$

Bedingung: -

Projektion und Union:

$PJ_A(R_1)\ UN\ PJ_A(R_2) \Rightarrow PJ_A(R_1\ UN\ R_2)$

Bedingung: -

Selektion und Kartesisches Produkt:

$SL_F(R_1)\ CP\ R_2 \Rightarrow SL_F(R_1\ CP\ R_2)$

Bedingung: -

$SL_{F1}(R_1)\ CP\ SL_{F2}(R_2) \Rightarrow SL_F(R_1\ CP\ R_2)$

Bedingung: -

Konstruktionsregel: $F = F_1 \wedge F_2$

Selektion und Join:

$SL_F(R_1)\ JN_{F1}\ R_2 \Rightarrow SL_F(R_1\ JN_{F1}\ R_2)$

Bedingung: -

$SL_{F1}(R_1)\ JN_{F3}\ SL_{F2}(R_2) \Rightarrow SL_F(R_1\ JN_{F3}\ R_2)$

Bedingung: -

Konstruktionsregel: $F = F_1 \wedge F_2$

Selektion und Semijoin:

$SL_F(R_1)\ SJ_{F1}\ R_2 \Rightarrow SL_F(R_1\ SJ_{F1}\ R_2)$

Bedingung: -

Selektion und Union:

$SL_F(R_1)\ UN\ SL_F(R_2) \Rightarrow SL_F(R_1\ UN\ R_2)$

Bedingung: -

Selektion und Differenz:

$SL_F(R_1)\ DF\ R_2 \Rightarrow SL_F(R_1\ DF\ R_2)$

Bedingung: -

Selektion und Intersection:

$SL_F(R_1)\ IN\ R_2 \Rightarrow SL_F(R_1\ IN\ R_2)$

Bedingung: -

Selektion und Division:

$SL_F(R_1)\ DV\ R_2 \Rightarrow SL_F(R_1\ DV\ R_2)$

Bedingung: $Attr(F) \subseteq (Attr(R_1) - Attr(R_2))$

Umwandeln von Operationen

Selektion und Kartesisches Produkt ⇒ Join:

$SL_F(R_1\ CP\ R_2) \Rightarrow R_1\ JN_F\ R_2$

Bedingung: $(\neg\exists F_h, F_i\colon F = F_h \vee F_i)\ \wedge$

$(\exists F_j\colon A \in Attr(R_1)\ \wedge\ B \in Attr(R_2)\ \wedge$

$F_j = A\ \Theta\ B\ \wedge\ ((F = F_j \wedge F_{k)} \vee F = F_j))$

Projektion und Join ⇔ Semijoin:

$PJ_A(R_1\ JN_F\ R_2) \Rightarrow R_1\ SJ_F\ R_2$

Bedingung: $A = Attr(R_1)$

$R_1\ SJ_F\ R_2 \Rightarrow PJ_A(R_1\ JN_F\ R_2)$

Bedingung: -

Konstruktionsregel: $A = Attr(R_1)$

Semijoin-Programm:

$R_1\ JN_F\ R_2 \Rightarrow R_2\ JN_F\ (R_1\ SJ_F\ R_2)$

Bedingung: -

Konstruktionsregel: $A = Attr(R_1) \cap Attr(R_2)$

$R_2\ JN_F\ (R_1\ SJ_F\ R_2) \Rightarrow R_1\ JN_F\ R_2$

Bedingung: $A = Attr(R_1) \cap Attr(R_2)$

Differenz und Differenz ⇔ Intersection:

$R_1\ DF\ (R_1\ DF\ R_2) \Leftrightarrow R_1\ IN\ R_2$

Bedingung: -

Umwandlung der Operation Division:

$R_1\ DV\ R_2 \Rightarrow PJ_A(R_1)\ DF\ PJ_A((PJ_A(R_1)\ CP\ R_2)\ DF\ R_1)$

Bedingung: -

Konstruktionsregel: $A = Attr(R_1) - Attr(R_2)$

$PJ_A(R_1)\ DF\ PJ_A((PJ_A(R_1)\ CP\ R_2)\ DF\ R_1) \Rightarrow R_1\ DV\ R_2$

Bedingung: $A = Attr(R_1) - Attr(R_2)\ \wedge\ Attr(R_2) \subseteq Attr(R_{1)}$

Differenz und Differenz ⇔ Differenz und Union:

$(R_1\ DF\ R_2)\ DF\ R_3 \Leftrightarrow R_1\ DF\ (R_2\ UN\ R_3)$

Bedingung: -

Differenz und Union ⇔ Differenz und Intersection und Differenz:

$R_1\ DF\ (R_2\ UN\ R_3) \Leftrightarrow (R_1\ DF\ R_2)\ IN\ (R_1\ DF\ R_3)$

Bedingung: -

Differenz und Intersection ⇔ Differenz und Union und Differenz:

$R_1\ DF\ (R_2\ IN\ R_3) \Leftrightarrow (R_1\ DF\ R_2)\ UN\ (R_1\ DF\ R_3)$

Bedingung: -

Minimieren von Ausdrücken

Alle nachfolgenden Transformationsregeln können bedingungslos angewendet werden. Des weiteren muß für keine dieser Regeln eine Konstruktionsvorschrift beachtet werden, die das Bilden des Ergebnisses vorschreibt.

Projektion:

$PJ_A(PJ_B(R)) \Rightarrow PJ_A(R)$

$$PJ_A(\varnothing) \Rightarrow \varnothing$$

Selektion:

$$SL_F(\varnothing) \Rightarrow \varnothing$$

Union:

$$R\ UN\ R \Rightarrow R$$

$$R\ UN\ SL_F(R) \Rightarrow R$$

$$SL_{F1}(R)\ UN\ SL_{F2}(R) \Rightarrow SL_{F1 \vee F2}(R)$$

$$R\ UN\ \varnothing \Rightarrow R$$

Intersection:

$$R\ IN\ R \Rightarrow R$$

$$R\ IN\ SL_F(R) \Rightarrow SL_F(R)$$

$$SL_{F1}(R)\ IN\ SL_{F2}(R) \Rightarrow SL_{F1}(SL_{F2}(R))$$

$$R\ IN\ \varnothing \Rightarrow \varnothing$$

Differenz:

$$R\ DF\ R \Rightarrow \varnothing$$

$$R\ DF\ SL_F(R) \Rightarrow SL_{\neg F}(R)$$

$$SL_{F1}(R)\ DF\ SL_{F2}(R) \Rightarrow SL_{F1}(SL_{\neg F2}(R))$$

$$R\ DF\ \varnothing \Rightarrow R$$

$$\varnothing\ DF\ R \Rightarrow \varnothing$$

Division:

$$R\ DV\ R \Rightarrow \varnothing$$

$$R\ DV\ \varnothing \Rightarrow R$$

$$\varnothing\ DV\ R \Rightarrow \varnothing$$

Kartesisches Produkt:

$$R\ CP\ \varnothing \Rightarrow R$$

Join:

$$R\ JN_F\ \varnothing \Rightarrow \varnothing$$

Semijoin:

$$R\ SJ_F\ \varnothing \Rightarrow \varnothing$$

$$\varnothing\ SJ_F\ R \Rightarrow \varnothing$$

Erklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 25.05.1995

Inhaltsverzeichnis