Konzeption und Test eines Optimiererbaukastens — Voruntersuchungen

Studienarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von Meier, Mathias
geboren am 23.06.1974 in Rostock

Betreuer: Prof. Dr. A. Heuer
Dr.-Ing. H. Meyer
Dipl.-Inf. J. Kröger
Dipl.-Ing. A. Lubinski

Abgabedatum: 23. Dezember 1999

Inhaltsverzeichnis

Kapitel 1

Einführung

Eine zentrale Komponente in einem modernen DBMS ist der Anfrageoptimierer. Seine relativ hohe Komplexität und der Aufwand für seine Konstruktion umgaben ihn oft mit einem Hauch schwarzer Magie. In meiner Studienarbeit werde ich versuchen, einen Baukasten zu entwerfen, mit dem es auf einfache Art und Weise möglich ist, einen Optimierer für ein beliebiges DBMS zusammenzusetzen.

1.1 Einordnung

Die Optimierung von Anfragen ist eine Aufgabe des Datensystems. Das Datensystem (siehe Fünf-Schichten-Architektur von Härder [LS87]) besitzt auf der einen Seite eine mengenorientierte Schnittstelle, auf deren Basis deskriptive Anfragen formuliert werden können, und auf der anderen Seite eine satzorientierte Schnittstelle zum Zugriffssystem. Ein Teil der Aufgabe des Datensystems ist nun die Abbildung der deskriptiven Anfrage auf einen Plan zur Abarbeitung der Anfrage, der sich auf die operationale Schnittstelle des Zugriffssystems stützt. Diese Abbildung mit dem Ziel, einen möglichst effizienten Ausführungsplan zu finden, übernimmt der Optimierer. Die Bearbeitung der Anfrage an sich besteht im weiteren aus der Anfrageverarbeitung (Optimierung) und der Anfrageevaluierung. Die Evaluierung oder Ausführung der Anfrage soll in meiner Arbeit nicht weiter beachtet werden, die Aufmerksamkeit gilt im folgenden allein der Optimierung.

Abbildung 1.1 zeigt den Prozeß der Anfragebearbeitung. Zu Beginn wird die Anfrage einer **syntaktischen Analyse** unterzogen. Dabei wird die Korrektheit der Anfrage bezüglich der Sprachgrammatik festgestellt. Danach folgt die **semantische Analyse**, welche die semantische Korrektheit (korrekte Typen, Relationennamen usw.) mittels konzeptuellem und externem Schema überprüft. Bei der **Normalisierung** wird der Qualifikationsteil einer Anfrage in eine Normalform gebracht. Das kann die konjunktive (KNF) oder die disjunktive Normalform (DNF) sein. Desweiteren gehört zur Normalisierung auch die Überführung der Anfrage in Prenex-Form[1]. Durch die einheitlichen Darstellungsformen erhält man einen definierten Ausgangspunkt für die folgenden Schritte, sie beeinflussen aber auch bereits die Anfrageausführung[2]. Ein vierter Schritt ist die **Vereinfachung**. Dazu gehören die Eliminierung von Redundanzen (anhand der Idempotenzregeln für Boolesche Ausdrücke), Hüllenbildung der Qualifikationsprädikate und die Berücksichtigung von semantischen Integritätsbedingungen. Ergebnis dieser ersten Schritte der Anfragebearbeitung ist eine interne Darstellung der Anfrage (Anfragegraph). Bis hierher spricht man auch noch von der Übersetzungsphase. Die nun folgende Phase wird als die eigentliche Optimierung bezeichnet.

[1] Siehe [Mit95], S. 45.

[2] So tendiert beispielsweise eine Anfrage mit DNF-Prädikaten dazu, Verbunde eher auszuführen als Vereinigungsoperationen.

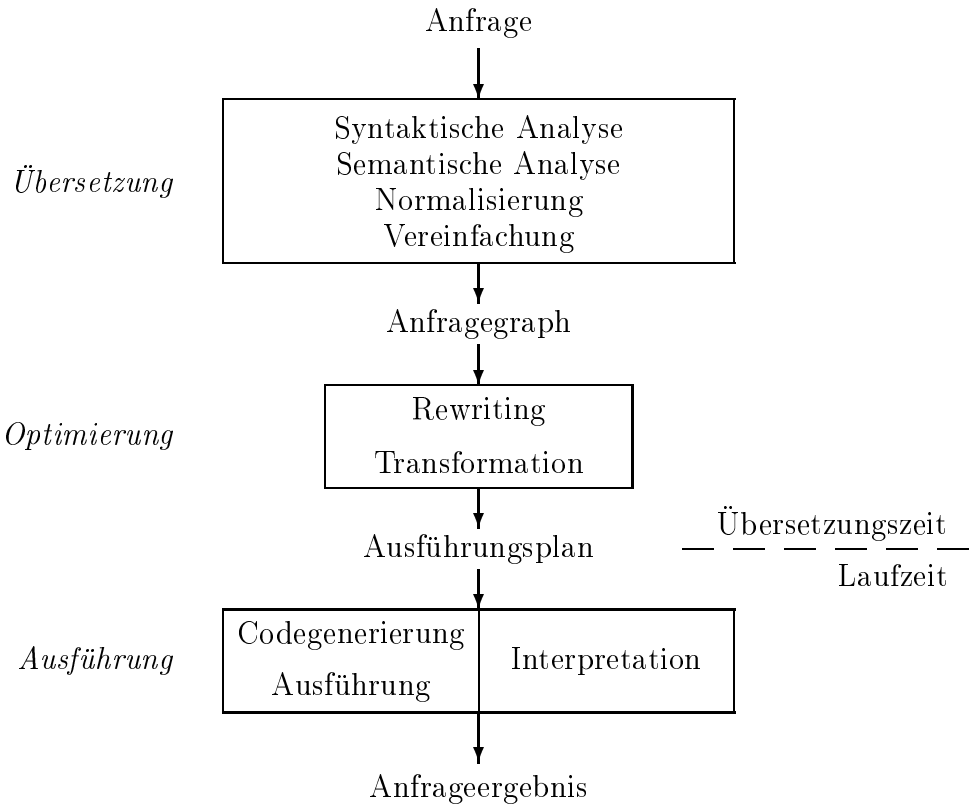

Abbildung 1.1: Der Prozeß der Anfragebearbeitung

Ziel der Optimierung ist es nun, eine Abbildung von den bisher logischen Operatoren der Interndarstellung auf physische Operatoren herzustellen und damit einen effizient ausführbaren Anfrageplan zu finden. Zunächst wird die Anfrage aber nach einer Heuristik umgeformt. Diese Umformung (**Rewriting**) bzw. algebraische Optimierung basiert nur auf den algebraischen Eigenschaften der Interndarstellung (beispielsweise der Relationenalgebra). Informationen über die internen Speicherungsformen der Relationen werden hier in der Regel nicht verwendet. Die Heuristik hat beispielsweise das Ziel, die Größe von Zwischenergebnissen zu minimieren, weil sich damit der Berechnungsaufwand verringert. Beim Rewriting kann die Anfrage auch anhand vorgegebener Regeln restrukturiert werden. Dazu gibt man direkt Regeln an, die das Separieren, Gruppieren, Kommutieren oder Ändern der Reihenfolge von Operatoren erlauben.

Bevor es zur Zuordnung von physischen Operatoren (den Planoperatoren), zu den logischen Operatoren im nun folgenden Schritt **Transformation** kommt, können noch einmal Vereinfachungen wie die Eliminierung gemeinsamer Teilausdrücke oder die Gruppierung von benachbarten Operatoren zu einem Operator vorgenommen werden. Dadurch ist dann nachfolgend eine weitgehend einfache und direkte Zuordnung von Planoperatoren zu logischen Operatoren möglich. Dabei entsteht ein Planoperatorgraph, bei dem für jeden Operator festgelegt wurde, durch welche der zur Verfügung stehenden Algorithmen er realisiert wird. Diese Methodenauswahl für die Operatoren wird auch als klassisches Optimierungsproblem bezeichnet. Hier benutzt man häufig ein anderes Verfahren als für das Rewriting. Man unterteilt diesen Schritt in die Plangenerierung, eine Kostenberechnung und eine Suchstrategie. Die Plangenerierung umfaßt das systematische Erzeugen alternativer Ausführungspläne und

liefert eine Menge von Plänen, zu denen mittels der Kostenberechnung ein Kostenwert ermittelt wird. Dieser Wert spiegelt den Aufwand für die Ausführung des Planes wieder und bezieht beispielsweise die Anzahl der Lese-/Schreiboperationen auf langsamen Massenspeicher ein. Anhand der Kosten kann daraufhin die Suchstrategie den billigsten Plan ermitteln. Dieser ist der optimale Plan, wenn er der am effizientesten ausführbare Plan ist, der das Anfrageergebnis liefern kann. Dieser Ausführungsplan ist Ergebnis der zweiten Phase, der Optimierung der Anfrage.

Bei der nun folgenden **Ausführungs**phase wird das Anfrageergebnis berechnet. Prinzipiell gibt es zwei Möglichkeiten: Der vorliegende Ausführungsplan wird interpretiert, d.h. die Planoperatoren werden nach der Optimierung nacheinander ausgeführt, oder es wird für den Plan Programmcode generiert, der gleich oder auch später ausgeführt wird.

Es ist noch zu bemerken, daß der erzeugte Ausführungsplan unterschiedliche Optimierungsziele realisieren kann. Betrachtet man einzelne Anfragen, könnte als Ziel die Bearbeitung der Anfrage bei minimalen Kosten oder bei minimaler Bearbeitungszeit vorgegeben sein. In Mehrbenutzerumgebungen können die Optimierungsziele Minimierung der Antwortzeit, Maximierung des Durchsatzes oder die Einhaltung einer Antwortzeitschranke sein.

1.2 Problemstellung

Das Ziel der Optimierung in DBS ist es, gemäß einem vorgegebenen Optimierungsziel die gestellte Anfrage umzuformulieren und einen optimalen Ausführungsplan zu finden. Um den Optimierungsprozeß zu adaptieren oder zu verteilen, ist es sinnvoll, ihn in einzelne Module aufzusplitten und diese austauschbar zu machen bzw. zu verteilen.

Ziel dieser Studienarbeit ist die Erarbeitung eines Baukastens, in dem Module für einen Optimierer, wie verschiedene Kostenmodelle oder Suchstrategien, einfach austauschbar sind.

1.3 Motivation

Die Anfrageverarbeitung ist ein komplexer Prozeß. Daher gibt es auch kaum Standardwege zur Konstruktion eines Optimierers. Das trifft besonders in den Fällen zu, bei denen es sich nicht um ein zentralisiertes DBMS mit einer SQL-artigen Anfragesprache handelt. Die Anstöße zu meiner Arbeit stammen daher auch vor allem aus den Lehrstuhl-Projekten HE_AD und MoVi.

In HE_AD [FLM96] wird am Beispiel eines verteilten Datenbanksystems gezeigt, daß die Optimierung ein vielschichtiger Prozeß ist. Das Optimierungsproblem wurde in Teilprobleme zerlegt, die einzeln bearbeitet werden. Hierbei erschien eine Beschränkung auf nur eine Technologie, wie zum Beispiel regelbasierte Techniken, nicht sinnvoll oder praktikabel. In HE_AD existiert bereits ein Werkzeugkasten von Optimiererbausteinen, der Techniken wie z.B. Transformationsregelmengen, Common Subexpression Elimination, ein Kostenmodell und Suchalgorithmen integriert.

Das Ziel des Projektes MoVi [Lub96] ist es, eine mobile Informationsverarbeitung zu ermöglichen. Mobilität ist hierbei nicht nur als Nutzung mobiler Geräte zu begreifen, sondern beinhaltet ebenfalls den Wechsel von stationären und mobilen Geräten und Netzen. Der mobile Kontext beeinflußt hier alle Softwarekomponenten. Der Einsatz eines DBMS in diesem Umfeld bringt folgende Probleme mit sich: Einerseits wird Offenheit benötigt, die auf der Transparenz bezüglich der Plattform, der Datenmodelle, Schemata, Typen usw. basiert. Andererseits wird Adaptivität gefordert, die unter anderem auch eine dynamische Anfragemodifikation und -optimierung beinhaltet. Die Optimierung umfaßt dabei etwa auch Entscheidungen, die den Ausführungsort der Optimierung selbst betreffen. Desweiteren sind Optimierungsziele erst zur Laufzeit bekannt, ins Kostenmodell fließen in starkem Maße Übertragungskosten ein, und die Ausnutzung von Replikation und der mobile Kontext an

sich erhöhen die Komplexität der Anfragebearbeitung. Der Optimierer in MoVi muß daher sehr flexibel und anpaßbar sein. Eine geeignete Aufteilung der Optimierung kann sicherlich alle diese Anforderungen erfüllen, vor allem aber auch den Entwurf eines idealerweise modularen Optimierers vereinfachen.

Im MoVi-Projekt wurde bereits ein Konzept für einen modularen DB-Kern in [Ber99] vorgestellt. Darin ist ein Ansatz zur Austauschbarkeit von Modulen enthalten, der im Zusammenhang mit meiner Arbeit zu überprüfen ist.

Ein weiterer Punkt, der diese Arbeit motiviert, ist die Leistungsbewertung von Optimierern [Jon98]. Um die Frage beantworten zu können, ob ein Anfrageoptimierer in seinem Aufgabengebiet gut oder schlecht arbeitet, muß eine Bewertung seiner einzelnen Komponenten vorgenommen werden. Man kann dann zum Beispiel Ungenauigkeiten im Kostenmodell feststellen und diese Komponente besser an die aktuelle Umgebung anpassen. Idealerweise sollte diese Anpassung ohne Einwirkung durch den Administrator oder Datenbankhersteller erfolgen. Man erweitert somit den Aufgabenbereich der Optimierung, die sich nun auch noch „selbst optimiert“ bzw. anpaßt. Dazu gehört dann allerdings eine gewisse Grundlage, die einfache Austauschbarkeit und Anpassung erlaubt. Es wäre eventuell günstig, sich auch hier auf der Ebene von Modulen zu bewegen.

Es ist ganz allgemein zu bemerken, daß es wesentlich leichter fällt, Probleme bei der Anfrageoptimierung zu lösen, wenn sich diese vom übrigen Optimierungsprozeß separieren lassen.

Letztendlich motivieren auch allgemeine Prinzipien der Softwaretechnik diese Arbeit. Durch Anwendung des Prinzip der Modularisierung läßt sich ein komplexes Softwaresystem überschaubar oder auch erst beherrschbar machen. Ein Modul als funktionale Einheit oder semantisch zusammengehörige Funktionsgruppe bietet Kontextunabhängigkeit und eine definierte Schnittstelle, ist handlich und verständlich. Die Komplexität der Optimierung und die Vielfalt verwendbarer Algorithmen und Konzepte machen die Anfrageoptimierung also durchaus zu einem Kandidat für eine Modularisierung.

Zusammenfassend bietet die Einführung eines Baukasten für Optimierer folgende Vorteile:

- Eine Modularisierung mit der Einführung eines Baukastenprinzips erleichtert den Entwurf.
- Standardkomponenten der Optimierung können zur leichteren Wiederverwendung in einer Programmbibliothek untergebracht werden.
- Durch die Wiederverwendung ergibt sich eine kürzere Entwicklungszeit für nachfolgende Optimiererentwicklungen.
- Je nach Granularität der Modularisierung können Bestandteile des Optimierers auf mehrere Rechner verteilt werden.
- Neue Möglichkeiten der (verteilten) Optimierung erhält man durch den Einsatz paralleler Algorithmen und Verfahren, zum Beispiel für die Suche.
- Durch eine klare Trennung der Komponenten können einzelne Teile gezielt an die Bedürfnisse des aktuellen DBS angepaßt werden.

Im Verlauf meiner Arbeit werden jetzt zunächst Ansätze zu ähnlich gelagerten Problemstellungen betrachtet. Die daraus gezogenen Schlußfolgerungen fließen dann in einen eigenen Vorschlag zu einem Optimiererbaukasten ein.

Kapitel 2

Existierende Ansätze

In diesem Kapitel werden Systeme vorgestellt, die eine Zielsetzung besitzen, die der dieser Studienarbeit ähnelt. Die Systeme kann man grob in erweiterbare DBMS, Spezifikationssysteme (Optimierergeneratoren) und Programmbibliotheken (Toolkits) einteilen, wobei aus jeder Klasse mindestens ein Vertreter vorgestellt wird. Den Anfang macht der Anfrageprozessor aus Starburst als Teil eines erweiterbaren DBMS. Darauf folgt der Optimierergenerator Volcano OptGen mit seinem regelbasierten Ansatz. Nach einem System, das die Arbeit mit Volcano vereinfachen soll – Prairie –, werden die beiden Klassenbibliotheken OPT++ und EROC vorgestellt, die beide auf der Programmiersprache C++ aufsetzen.

2.1 Starburst

Das erweiterbare DBMS Starburst [HFL+89] ist ein Forschungsprototyp des Forschungslabors der IBM in Almaden. In Starburst ist es möglich, Spracherweiterungen vorzunehmen und die interne Ebene um neue Datentypen, Speicherstrukturen und Zugriffsmethoden zu erweitern. Die Anfragebearbeitung wurde auf den Anfrageprozessor *Corona*, der die Anfrage in eine ausführbare Form bringt, und den Data-Manager *Core* aufgeteilt, der, verantwortlich für Datenspeicherung, Zugriffspfade usw., die Anfrageevaluierung übernimmt.

Grundlage der Anfragebearbeitung bildet das *Query Graph Model (QGM)*, in das die Anfrage nach dem Parsen überführt wird und das in allen Optimierungsphasen benutzt wird. Die SQL-artige Anfragesprache Hydrogen, die auch die Formulierung rekursiver Anfragen zuläßt, erforderte diese Form der internen Darstellung. Die Optimierung wurde in Starburst in die Phasen *query rewrite* und *plan optimization & refinement* eingeteilt.

Query Rewrite

Für die Manipulation von Anfragegraphen wurde ein eigenes regelbasiertes Rewrite-System entwickelt, in dem für QGM-Ausdrücke algebraische Regeln zur Vereinfachung des Anfrageausdrucks angegeben werden, die die *Rule-Engine* anwenden kann. Es können Regelklassen gebildet werden, die die Suche nach einer anwendbaren Regel verkürzen. Die Reihenfolge der Regelanwendungen kann ebenfalls explizit durch die Vergabe von Prioritäten angegeben werden — besonders erfolgreiche Regeln bekommen dabei eine hohe Priorität. Die Regeln werden als ein IF-THEN-Konstrukt angegeben, die Regelsprache ist somit an die Programmiersprache C angelehnt. Jede Regel besteht immer aus einer Bedingung und einer Aktion.

Bei den Ersetzungen im Graphen wird von der Rule-Engine immer eine von drei Strategien angewendet. Regeln können dabei durch eine sequentielle Strategie, prioritätsgesteuert oder zufallsbasiert (statistisch) ausgewählt werden. Zusätzlich liefert eine (Rewrite-)Suchstrategie immer die QGM-Knoten, die Kandidaten für den nächsten Rewrite-Schritt sind.

Damit kann man im Graphen eine Breiten- oder Tiefensuche durchführen. Das Rewriting terminiert, wenn ein vorgegebenes Budget (Parameter `budget`) erschöpft ist.

Plan Optimization & Refinement

Die zweite Phase der Optimierung wird in die drei Komponenten Plangenerierung, Kostenbewertung und Suchstrategie aufgeteilt. Bearbeitet wird die Anfrage nach einem festen Algorithmus, *bottom-up*, wobei jeder QGM-Operation (QGM-Knoten) einzeln mittels eines regelbasierten Plangenerators eine algorithmische Realisierung zugewiesen wird. Bei den Regeln handelt es sich um parametrisierte, grammatikähnliche Produktionsregeln, die als *strategy alternative rules (STARs)* bezeichnet werden. Die Terminale in den Grammatikregeln sind sogenannte LOLEPOPs (*low level plan operator*), also die Methoden (Filescan, Sort, usw.). Eine Besonderheit bilden die *glue STARs*, die noch Veränderungen am erzeugten Plan vornehmen können. Diese Regeln erzwingen gewisse Planeigenschaften, die dann den Einsatz von „billigeren" Algorithmen erlauben. Beispielsweise kann ein Nested-Loops-Join durch einen Merge-Join ersetzt werden, wenn vorher eine Sortierung der Eingaberelationen erzwungen wurde.

Die Suchstrategie ist in dieser Phase für die Auswahl der STARs verantwortlich. Die Regeln besitzen auch hier einen Rang beziehungsweise eine Priorität, nach der eine Anwendungsreihenfolge der Regeln bestimmt wird. Zwei weitere Parameter der Suchstrategie erlauben, gewisse Anfragepläne auszuschließen. So soll das Erzeugen von *bushy trees* oder die Bildung von kartesischen Produkten vermieden werden.

Die dritte Komponente, die Kostenbewertung, kann hier nicht genauer beschrieben werden, da sie in [HFL+89] leider vernachlässigt wurde. Zusammenfassend besteht die Planoptimierung also aus der Regelmaschine für die STARs, einer Suchstrategie für den nächsten STAR und einem Regelarray, das die STARs enthält.

2.1.1 Einschätzung

Durch das durchgängig verwendete Regelkonzept kann man in Starburst leicht Einfluß auf die Anfragebearbeitung nehmen. Das betrifft nicht nur die Definition neuer Datentypen, Speicherstrukturen oder Zugriffsmethoden, sondern vielmehr auch deren Integration in den Anfrageprozessor. Für jeden Operator, der beispielsweise eine neue Zugriffsmethode anbietet, wird einfach eine Regel angegeben. Der Nachteil, der Produktionsregelsystemen im allgemeinen anhängt — daß ein etwaiger Overhead durch die gleichzeitige Betrachtung vieler Regeln entsteht — wird durch die Regelverwaltung (Regelklassen mit priorisierten Regeln) vermieden. Desweiteren erlaubt das QGM die Verwendung mächtigerer Anfragesprachen als SQL.

Trotz aller Einflußnahme muß man sich in Starburst an den streng zweiphasigen Ablauf der Optimierung gewöhnen. Die Pläne werden immer erst generiert, wenn das Query Rewrite abgeschlossen wurde. Eine mögliche Eingrenzung während der Rewrite-Phase, um teure Planalternativen schon hier auszuschließen, fällt damit leider aus. Weiterhin legt man sich auf das QGM fest und muß jede neue Anfragesprache darauf abbilden. Außerdem erschwert das QGM das Verständnis und die Erweiterung der Regeln, die nicht so offensichtlich wie bei der Relationenalgebra sind. Auffallend ist auch, daß es zwei getrennte Regelsysteme gibt, was meiner Meinung nach etwas unhandlich ist.

Das größte Manko von Starburst ist aber, daß ein fester Algorithmus für die Plangenerierung verwendet wird. Es existiert kein „echter" Suchalgorithmus, der einen Raum von Plänen nach einer beliebigen Strategie erforscht und so den optimales Plan ermittelt.

Zusammenfassend läßt sich schlußfolgern: Ein erweiterbarer Optimierer sollte regelbasierte Techniken für Rewriting und Planerzeugung verwenden, da die Funktionalität einfach durch Hinzunahme und Löschen von Regeln verändert werden kann. Voraussetzung sollte

auch eine leistungsfähige Regelverwaltung sein. Nun kann aber auch eine einheitliche (und genügend mächtige) Anfragerepräsentation (QGM) sinnvoll sein, da vorher definiertes wiederverwendbar ist und die Regelspezifikation eine einheitliche Grundlage besitzt. Zu prüfen wäre dann noch, ob man sich dabei etwa auf das relationale Paradigma beschränkt.

2.2 Optimierer-Generator Volcano

Volcano OptGen ist ein Generator für ein erweiterbares Anfrageoptimierungssystem und war Teil eines Forschungsprojektes von Goetz Graefe und William J. McKenna [GM93] Um aus einer Spezifikation Code für einen Optimierer zu erstellen, verwendet dieser Generator einen regelbasierten Ansatz, bei dem der Anfragesuchraum in eine logische und eine physische Dimension aufgeteilt wird, die jeweils durch Anwendung von Regeln erforscht werden können.

2.2.1 Optimiererdesign mit Volcano

Ein Optimierer wird mit Volcano in zwei Etappen entwickelt. Dazu wird zuerst in einer Datei (`model.input`) das Datenmodell spezifiziert. Dazu gehören die Operatoren einer logischen Algebra, beispielsweise der Relationenalgebra, die Operatoren einer physischen Algebra (ausführbare Methoden) und die zugehörigen Umformungsregeln. Daraus wird dann C-Code generiert, der im zweiten Schritt zusammen mit der Datei `dbi.c`, die Implementationen der verwendeten Algorithmen, das Kostenmodell und eine Suchmaschine enthält, compiliert wird (Abbildung 2.1).

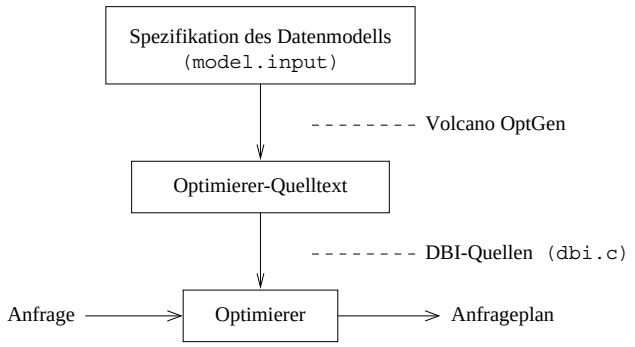

Abbildung 2.1: Herangehensweise des Volcano OptGen

Der regelbasierte Ansatz

Die Basis dieses Ansatzes bilden zwei Algebren. Für den ersten Schritt der Optimierung, das Rewriting, müssen zuerst sogenannte logische Operatoren deklariert werden. Das sind zum Beispiel einige der Operatoren der Relationenalgebra `Select`, `Proj` und `Join`. Für den zweiten Schritt, die Anfragetransformation, sind physische Operatoren anzugeben, also Algorithmen, mit denen auf den Datenbestand des DBS zugegriffen wird. Siehe dazu Abbildung 2.2. Desweiteren sind Regeln für die logische Algebra anzugeben, mit deren Hilfe sich die in Form eines Baumes dargestellte Anfrage in äquivalente Anfragen umwandeln läßt. Implementationsregeln (der physischen Algebra) sorgen für eine Abbildung eines Anfragebaumes auf einen Ausführungsplan. Gibt man für jede Regel Bedingungen an, so kann man versuchen, Regelmengen zu bilden, und somit eine gewünschte Heuristik oder Strategie bei der Umformung besser ausbauen. Eine ausführliche Darstellung der Regelspezifikation ist in der Diplomarbeit von Torsten Grust [Gru94] zu finden.

```
-- Define operators and their arities.
%operator JOIN 2

-- Define algorithms, enforcers, and their arities.
%algorithm MERGE 2
%algorithm FILE_SCAN 0
%enforcer SORT 1

-- JOIN can be implemented by MERGE.
%impl_rule (JOIN ?op_arg1 (?1 ?2)) -> (MERGE ?al_arg1 (?1 ?2))

-- This is a JOIN commutativity rule.
%trans_rule (JOIN ?op_arg1 (?1 ?2)) ->! (JOIN ?op_arg2 (?2 ?1))
-- Reverse the join argument if the rule is applied.
%appl_code
{{
reverse_JOIN_argument (?op_arg2, ?op_arg1) ;
}}
```

Abbildung 2.2: Regelspezifikation in Volcano

Search-Engine

Die Suchmaschine ermittelt zuerst die zu einem Anfragebaum gehörenden Pläne durch Anwenden der Implementationsregeln. Dann wird schrittweise in der logischen Dimension erweitert und jeweils neue Pläne werden generiert. Falls ein (Teil-)Plan die Eigenschaften eines Algorithmus (physical properties) wie z.B. die Sortierreihenfolge beim Merge-Join-Algorithmus nicht erfüllt, können sogenannte Enforcer-Algorithmen angegeben werden, die diese Eigenschaft dann herbeiführen.

Die Anfrage wird als Suchproblem betrachtet und top-down bearbeitet. Es werden nur Plan-Alternativen betrachtet, die die physical-property-Forderung des übergeordneten Teilplanes erfüllen. Desweiteren wird ein Plan nur weiter verfolgt, wenn seine Ausführungskosten ein neues Minimum darstellen. Grundlage für die Kostenbestimmung ist ein ADT Cost. Die Suchmaschine nimmt eine erschöpfende Suche in der logischen Dimension vor, *branch-and-bound-pruning* verkleinert den Raum insofern, daß Pläne bzw. Anfragen, deren Pläne bereits das Kostenlimit übersteigen, nicht weiter an der Umformung teilnehmen.

2.2.2 Einschätzung

Der Optimierergenerator Volcano OptGen trägt zur einfacheren und schnelleren Entwicklung von Optimierern bei. Durch das Regelkonzept sind Umformungsmöglichkeiten der Anfrage leicht angebbar. Vorhandene Regelmengen sind unter Umständen wiederverwendbar, wenn nur einige Regeln hinzugefügt oder entfernt werden müssen. Da die Operatoren für die Anfragealgebra direkt anzugeben sind, ist Volcano an sich unabhängig vom zugrundeliegenden Datenmodell. Die Suchmaschine ist bereits vorhanden, so daß die Entwicklungsarbeit für diese Komponente entfällt.

Mit Volcano hat man prinzipiell die Möglichkeit, Komponenten wie das Kostenmodell, implementierende Algorithmen usw. getrennt zu spezifizieren. Die Trennung der Optimiererkomponenten ist aber nicht so offensichtlich. Es stehen nur die beiden Dateien `model.input` und `dbi.c` zur Verfügung, jedoch auch der Include-Mechanismus von C. Schnittstellen existieren nur innerhalb von Volcano, so daß sich ein Import einer Komponente (eventuell die Menge der logischen Umformungsregeln) am Volcano-Stil orientieren muß. So ein Import

bzw. Austausch kann auch nicht zur Laufzeit erfolgen. Das System erlaubt nur den einen Weg der Änderung im Quelltext mit anschließender Compilierung. Die Regelmenge wird zusammen mit dem Code der Search-Engine übersetzt(Abb. 2.1). Dadurch können die Regeln dann auch nicht mehr zur Laufzeit angepaßt werden.

Desweiteren können nur baumartige Anfrage-Strukturen bearbeitet werden. Das ist in der Regel auch vollkommen ausreichend, wirft aber die Frage auf, wie gemeinsame Teilausdrücke (CSE) behandelt werden. In Volcano wurde das durch die Verwendung von Hash-Tabellen gelöst, in denen Teillösungen von schon berechneten Plänen gespeichert werden. Somit läßt sich auch die Redundanz vermeiden, wenn ein und derselbe Plan durch die algebraischen Umformungen auf mehreren Wegen erreicht wird.

Die in Volcano OptGen enthaltene Search-Engine ist wenig flexibel. Dabei wurde eine erschöpfende Suche im zweidimensionalen Lösungsraum implementiert, die nur eine Eingrenzung bei Überschreitung eines bestimmten Kostenlimits vornimmt. Dieser Algorithmus ist zwar durch die Verwendung dynamischer Programmierung effizient realisiert werden ([GM93]), alternative Suchstrategien jedoch sind nicht anwendbar.

Zusammenfassend läßt sich über Volcano folgendes sagen: Seine große Stärke ist die Regelspezifikation. Volcano ist dabei vollkommen unabhängig vom Datenmodell. Dennoch ist der Ablauf der Optimierung vorgegeben. Für einen universellen Optimiererbaukasten sollte ein unabhängiges Regelkonzept ein Muß bedeuten und die restlichen Komponenten (Suche, Kosten) austauschbar sein.

2.3 Prairie

Um bei einem regelbasierten Optimierer die Angabe von Regeln zu vereinfachen, wurde von Das und Batory [DB95] Prairie entwickelt. Es ist ein Präprozessor für Volcano, der Prairie-Regeln in Volcano-Regeln übersetzt. Die Vorteile ergeben sich durch eine einheitliche und einfachere Spezifikation von Regeln. So können Operatoren und Algorithmen in jeder Regel auftauchen, d.h. es gibt keine Extra-Klasse für Operatoren, Algorithmen und Enforcer, und es wird keine Unterscheidung zwischen logischen und physischen Eigenschaften für die Operatoren bzw. Algorithmen gemacht.

In Prairie gibt es nur zwei Arten von Regeln: T-Rules (Transformationsregeln) und I-Rules (Implementationsregeln). Letztere sehen beispielsweise so aus:

$$
\begin{array}{l}
\mathrm{SORT}(S_1) : D_2 \Longrightarrow \mathrm{Merge\_sort}(S_1) : D_3 \\
(D_2.\mathrm{tuple\_order} \mathrel{!}= \mathrm{DONT\_CARE}) \\
\{\{ \\
\qquad D_3 = D_2 \; ; \\
\}\} \\
\{\{ \\
\qquad D_3.\mathrm{cost} = D_1.\mathrm{cost} + (D_3.\mathrm{num\_records}) * log(D_3.\mathrm{num\_records}) \;\; ; \\
\}\}
\end{array}
$$

Dabei bezeichnet S_i einen Tupelstrom und D_i einen Descriptor für einen Knoten im Operatorbaum. Die erste Zeile bezeichnet die Regel an sich, und in der zweiten Zeile wird die Anwendbarkeit der Regel überprüft. Im nachfolgenden ersten Klammerpaar sind Aktionen enthalten, die vor Regelanwendung ausgeführt werden, im zweiten Klammerpaar sind die nach der Anwendung auszuführenden Aktionen enthalten.

In einer Weiterentwicklung von Prairie wurde die Verwendung von Regelmengen ermöglicht [DB96]. Aus (kleinen) primitiven Regelmengen wird eine monolithische Regelmenge zusammengesetzt, die dann an das jeweilige DBMS angepaßt ist. Für das Rewriting einer Anfrage ist eine Reihenfolge anzugeben, in der die primitiven Regelmengen zu verwenden sind. Ein Anfragebaum wird hierbei solange umgeformt, bis keine Regel einer Regelmenge mehr anwendbar ist.

Im Hinblick auf ein zukünftiges System kann man am Beispiel von Prairie erkennen, daß sich die Form der Regelspezifikation noch vereinfachen läßt. Regelmengen erlauben, den Ablauf der Umformung gezielt zu steuern, wodurch man einen beliebigen Umformungsplan erstellen kann.

2.4 OPT++

OPT++ [KD95] ist eine Klassenbibliothek für erweiterbare Anfrageoptimierer, mit der man sich auf der Grundlage der Programmiersprache C++ die Implementierung und Erweiterung eines Optimierers erleichtern kann. Unterstützung soll diese Bibliothek vor allem bei der Implementation eines Optimierers für ein neues DBMS sowie beim Experimentieren mit verschiedenen Optimierungstechniken bieten. DeWitt und Kabra entwickelten dieses System an der University of Wisconsin für ihre dortigen Forschungsprototypen.

Designablauf

In OPT++ wurden einige abstrakte Klassen definiert, die zunächst kein „Wissen“ von der konkreten Optimierung haben. Abgeleitete Klassen legen durch die Definition der virtuellen Methoden die gerade verwendete Algebra, das verwendete Kostenmodell usw. fest.

Ein mit OPT++ erstellter Optimierer besteht hauptsächlich aus den drei Komponenten **Suchstrategie**, einer Komponente für den **Suchraum** und einer **Algebra**-Komponente. Sie sollen zunächst vorgestellt werden.

2.4.1 Die Algebra-Komponente

Auch OPT++ geht, wie Volcano, von einer Anfragealgebra aus. Die Operatoren der logischen und physischen Algebra spiegeln sich hier in den Ableitungen der abstrakten Klassen `Operator` und `Algorithm` wieder.

Die Klasse `Operator` repräsentiert logische Operatoren, z.B. `Select` oder `Join`. Es ist für jeden Operator der aktuellen Anfragealgebra eine Klasse abzuleiten und zu implementieren. Um logische Vereinfachungen bzw. Umformungen anwenden zu können, ist eine Klasse `TreeDescriptor` zu definieren, die für jeden Knoten des Anfragebaumes Informationen aufnimmt, die sich während der logischen Umformung an einem Knoten im Baum ergeben.

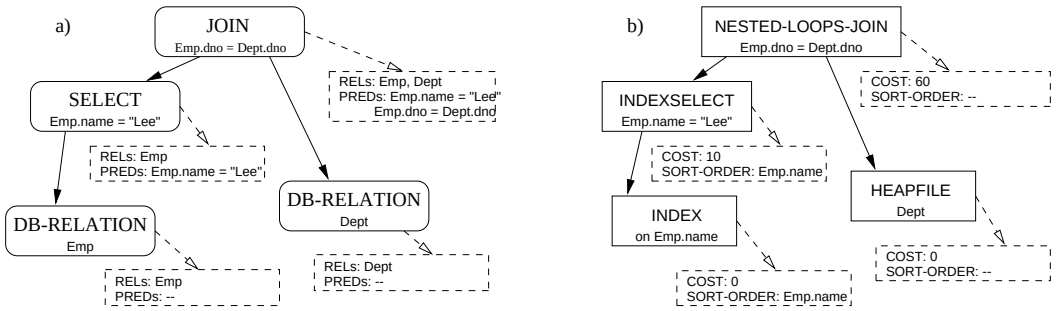

Abbildung 2.3: Beispiel eines Operatorbaumes mit Tree-Deskriptoren (a) und eines Ausführungsplans mit Plan-Deskriptoren (b)

Das sind zum Beispiel die Menge der bereits beteiligten Relationen oder die Menge der bereits erfüllten Prädikate. In der Abbildung 2.3a ist dazu ein Operatorbaum für eine Anfrage dargestellt. Jeder Knoten enthält zusätzlich einen Verweis auf seinen `TreeDescriptor`.

Die Klasse `Algorithm` repräsentiert die physischen Operatoren. Das sind z.B. die Algorithmen zur Realisierung eines Joins (`MergeJoin`) oder für einen sequentiellen Relationenscan (`HeapFile`). Ähnlich zur Klasse `Operator` existiert auch für die `Algorithm`-Klasse eine sogenannte `PlanDescriptor`-Klasse, die die physischen Eigenschaften des Planes aufnimmt — zum Beispiel die Art der Sortierung und die Kosten (siehe Abbildung 2.3b).

2.4.2 Die Suchraum-Komponente

In OPT++ existieren drei abstrakte Klassen, die für den Aufbau eines Suchraumes verantwortlich sind: `TreeToTreeGenerator`, `TreeToPlanGenerator` und `PlanToPlanGenerator`. Die Suchstrategie (siehe 2.4.3) durchsucht diesen Raum, indem sie diese Klassen verwendet, um neue Pläne zu erzeugen.

Für die logische Umformung (Rewriting) sind dabei die abgeleiteten Klassen von `TreeToTreeGenerator` verantwortlich. In einem System-R-ähnlichen Optimierer [SAC+79] wären dies z.B. `SelectExpand` oder `JoinExpand`. Der Aufruf der virtuellen Methode `Apply` nimmt dabei eine Umformung vor, so daß ein alternativer Anfragebaum erzeugt werden kann. Mit Ableitungen dieser Klasse können auch die in Volcano verwendeten (Äquivalenz-)Regeln nachgebildet werden.

Die Klasse `TreeToPlanGenerator` ist für die Zuordnung von logischen Operatoren zu physischen Operatoren (Algorithmen) verantwortlich. Auch hier implementiert jede Unterklasse die virtuelle Funktion `Apply`, die die Zuordnung ausführt. Beispielklassen wären `LoopsJoinGenerator` oder `MergeJoinGenerator`, zwei Realisierungen für einen Join.

Die `PlanToPlanGenerator`-Klasse bietet weitere Möglichkeiten, den Ausführungsplan zu modifizieren. Hier sollen Klassen wie `SortEnforcer` abgeleitet werden. Diese Klasse erinnert an den Enforcer-Mechanismus aus Volcano. Es existiert auch hier eine `Apply`-Methode.

2.4.3 Die Suchstrategie-Komponente

OPT++ definiert für die Suche eine `SearchStrategy`-Klasse, von der für einzelne Suchstrategien jeweils Klassen abgeleitet werden. `SearchStrategy` besitzt eine virtuelle Methode `Optimize`, die den Optimierungsprozeß entsprechend einer Suchstrategie initiiert. Dabei liegt es in der Hand der Suchstrategie, die Tree- und Plan-Deskriptoren (siehe 2.4.1) zu berechnen und in die Entscheidungsfindung einzubeziehen. Ob ein ein Algorithmus bzw. eine Umformung angewendet werden kann, wird anhand der `CanBeApplied`-Funktionen festgestellt, die jede Suchraum-Klasse anbieten muß.

2.4.4 Ein Beispielszenario

In Abbildung 2.4 sind die vorgestellten (Basis-)Klassen noch einmal zu sehen. Zusätzlich sind Beispielklassen für einen System-R-ähnlichen Optimierer angegeben. `Operator` und `Algorithm` bilden im linken unteren Teil das verwendete Datenmodell mit den Klassen `DBRelation`, `Select`, `HeapFile` usw. Daneben sind die Komponenten eingezeichnet, die den Optimierungsalgorithmus aus System R widerspiegeln. Die Suchstrategie-Komponente ist von diesen Definitionen relativ unabhängig. `BottomUp` bezeichnet die System-R-Strategie, weitere Suchstrategien sind Simulated Annealing(SA), Iterative Improvement(II) u.a., die bereits von OPT++ geliefert werden[1] (siehe [KD95]).

[1] Wie diese Suchstrategien im einzelnen an das System angepaßt wurden, soll an dieser Stelle offen bleiben. In jedem Fall benutzt eine Suchstrategie die drei Umformungsklassen (Abschnitt 2.4.2) über deren angebotenes Interface (die `CanBeApplied`-/`Apply`-Methoden) und wird wohl jeweils eine eigene Vorstellung von einem Suchraum haben.

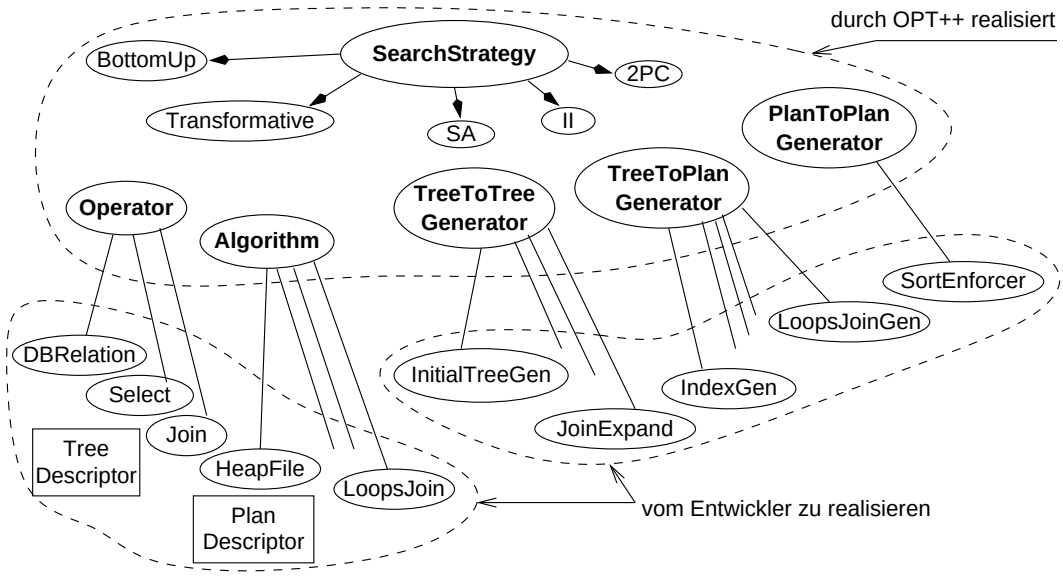

Abbildung 2.4: OPT++ mit Beispielklassen.

2.4.5 Einschätzung

Der Hauptzweck besteht auch bei OPT++ darin, den Implementierer von der großen Menge an Handarbeit bei der Erstellung eines Optimierers zu befreien. Der Optimierungsprozeß wurde dreigeteilt und die Komponenten können unabhängig voneinander spezifiziert werden. Dabei bietet OPT++ schon vorgefertigte Suchstrategien.

Aufgrund des C++-Klassenkonzeptes können Änderungen an einer Komponente ohne Beeinflussung der anderen vorgenommen werden. In einigen Fällen besteht jedoch eine Abhängigkeit untereinander. So müssen bei Einführung eines neuen logischen Operators und des dazugehörigen Algorithmus neue Klassen von der Umformungskomponente „Suchraum“ gebildet werden. Dieser Sachverhalt wird sich jedoch immer bei einer Aufteilung in „Menge der Operatoren“ und „Umformungskomponente“ ergeben. Man könnte gewisse Operatoren zwar definieren, sollte aber nicht gezwungen werden, sie zu verwenden.

Ein Mittel, mit dem die Unabhängigkeit erreichen wurde, ist das Führen von Listen über die vorhandenen Umformungsmöglichkeiten und über die Operatoren und Algorithmen. So eine Liste enthält z.B. alle logischen Operatoren, eine weitere alle Klassen zur logischen Umformung. Die Suchstrategie kann dadurch ohne Kenntnis über die Umformung einen Suchraum aufspannen, indem sie auf einen initialen Baum bzw. Plan einfach die `CanBeApplied`-Methode aller Klassen aus der entsprechenden Liste aufruft[2].

Im Bezug auf die Umformung der Anfrage blieb bei der Untersuchung dieses Systems offen, wie die Verwendung von Regelmengen aussehen soll — eine Möglichkeit ergäbe sich eventuell über die `CanBeApplied`-Methoden, die man mit dem Bedingungscode der vorangegangenen Regelsysteme vergleichen kann.

Die Kostenbestimmung erfolgt durch die Berechnung der Kosten für jeden generierten Teilplan, wobei die Werte im `PlanDescriptor` gespeichert werden. Zugrundegelegt wurde eine Klasse `Cost`, die für jeden Algorithmus eine Funktion `Compute` zur Bestimmung seiner Kosten bereitzuhalten hat. Die Definition der Kosten kann separat durchgeführt werden.

[2] Zumindest bei der mir vorliegenden Implementierung wurden Listen über die vorhandenen Algorithmen, Umformungsklassen usw. geführt und deren Elemente bei jedem Schritt nacheinander auf Anwendbarkeit geprüft. Bei langen Listen kann diese Vorgehensweise meines Erachtens nach aber auch Performance-Einbußen bewirken.

Sie hängt aber auch von der aktuellen Menge der Algorithmen ab. Eine Definition einer Kostenfunktion für einen aktuell nicht vorhandenen Algorithmus ist im C++-Design leider nicht möglich, was die Austauschbarkeit von (Algebra-)Komponenten etwas behindert.

Bei der Berechnung der Kosten wäre auch eine Möglichkeit denkbar, bei der jeder Algorithmus seine Kostenfunktion als eine „Default"-Funktion mitbringt, ein konkretes Kostenmodell diese aber redefiniert.

Ein positiver Aspekt in OPT++ ist die gute Trennung der Optimiererkomponenten. Soweit C++ eine Aufteilung von Klassendefinitionen zuläßt, wurde diese Möglichkeit genutzt. So können Operatoren und Algorithmen separat in einer Header-Datei spezifiziert werden, sowie Kostenbestimmung und Suchstrategie ebenfalls einzeln angegeben werden.

Das Vorhandensein mehrerer Suchstrategien mit einer entsprechend einfachen Austauschbarkeit verleitet hier dazu, einen Wechsel dieser Komponente zur Laufzeit auszuführen. Die dynamische Auswahl einer Suchstrategie ist in OPT++ jedoch noch selbst zu implementieren.

Fazit. OPT++ erweist sich als guter Ausgangspunkt für die Entwicklung eines Optimiererbaukastens. Es wurde eine Aufteilung der Optimierung vorgenommen, die erweiterbar, austauschbar und gut strukturiert ist. Als Komponenten wurden Klassen gewählt, die eine leichte Kombination auf der Ebene der Programmiersprache ermöglichen. Lediglich die Namen für die Klassen sollten besser formuliert werden, um sich beispielsweise nicht nur auf Anfragebäume festzulegen. Negativ zu bewerten ist die schlechte Unterstützung für die regelbasierte Umformung — diese Technik sollte meines Erachtens nach ebenfalls „mitgeliefert" werden.

2.5 EROC-Toolkit

EROC (Extensible, Reusable Optimization Components) ist ein Optimierer-Toolkit auf der Basis von C++ [MBH+96]. Im Vergleich zu OPT++ sind die Abstraktionen (Klassenkonzept) jedoch andere. EROC hat eine eigene Definition von Ausdrücken als Grundlage für die Repräsentation von Anfragen und Ausführungsplänen. Auf dieser Basis wird die Umformung (auch regelbasiert) durch einen *expression enumerator* vorgenommen.

In EROC wird als Ausdruck (`Multi_Expr`) folgendes bezeichnet: logische Algebraausdrücke (Anfragen), Ausführungspläne, Pfad-Ausdrücke (City.major.name), arithmetische Ausdrücke, Aggregat-Ausdrücke und Prädikate. Regeln werden so aufgestellt, daß sie auf eine Teilmenge der Ausdrücke anwendbar sind. Mehrere Ausdrücke werden in den Klassen `Expr_Class` bzw. `Expr_Space` zusammengefaßt, um beispielsweise algebraisch äquivalente Anfragen zusammenzufassen. Mit der Klasse `Enumerator` werden Räume aufgebaut (`Expr_Space`-Instanzen) und die Klasse `Mapper` transformiert einen Raum (logischer Ausdrücke) in einen anderen (Plan-Raum). Beispielklassen der Umformung sind `BU_Enumerator` (Aufbau des Suchraumes bottom-up im System-R-Stil) und `Trans_Enumerator` (top-down im Volcano-Stil mit zugehöriger Regel-Menge). Bemerkenswert ist, daß sich der `Trans_Enumerator` auch für Prädikat-Umformungen verwenden läßt. Ein Beispiel für einen Mapper wäre die Klasse `Volcano_Mapper`.

Die Trennung von Rewriting und Transformation führt in EROC zu getrennter Suche. Dadurch hat man die Möglichkeit, verschiedene Strategien zu verwenden, z.B. vollständiges Rewriting und nur simples Mapping mit wenigen Algorithmen. Leider war nicht ersichtlich, wie Enumerator und Mapper zusammenarbeiten. Auch war unklar, ob nur die festen Algorithmen beispielsweise aus System R übernommen wurden, oder ob sich tatsächlich beliebige Suchstrategien integrieren lassen.

Positiv an EROC ist die einheitliche interne Darstellung zu bewerten, da dadurch die Rewriting-Komponente (Enumerator) universell einsetzbar ist. Es bleibt aber noch zu überlegen, ob und wie diese Idee übernommen werden kann.

2.6 Zusammenfassung

In den frühen Datenbanksystemen bestand die Optimiererkomponente meist nur aus einem festen Algorithmus. System R von IBM machte mit seinem *Access-Path-Selection*-Algorithmus [SAC+79] den Anfang, und der Universitätsprototyp Ingres stellte eine weitere Methode, die *Query Decomposition* [SWK76], vor. Die Funktionalität war „fest verdrahtet" und der Optimierer war meistens infolge der vielen Patches nur sehr schlecht wartbar. Ein Optimierer für ein neues DBMS mußte von Grund auf entwickelt werden und der Aufwand war bei dieser Vorgehensweise für heutige Verhältnisse inakzeptabel. 1987 stellte Freytag [Fre87] als erstes den regelbasierten Ansatz für die Anfrageoptimierung vor, der in viele Optimierertools auch einfloß. Nun wurde darüber nachgedacht, wie man die Konstruktion eines Optimierers weiter vereinfachen könnte. Die hier vorgestellten Systeme präsentieren einige Vorschläge. Die meisten stützen sich auch auf das Regelkonzept und erlauben die Konstruktion mit bereitgestellten Komponenten (Standardkomponenten).

Die Wiederverwendbarkeit von selbst entwickelten Optimiererkomponenten ist allerdings nicht allzu hoch, da vieles vom zugrundeliegenden Datenmodell abhängt. Gut wiederzuverwenden sind aber in jedem Fall die Suchalgorithmen und die Regel-Maschinen.

Warum die Optimierergeneratoren und Toolkits im ganzen so wenig benutzt wurden, liegt vielleicht auch an der Mentalität der Entwickler, die lieber komplett neu entwickeln, als sich erst in andere (Hilfs-)Systeme einzuarbeiten. Desweiteren schränken die Systeme ja auch ein: Die meisten erzeugen monolithische Optimierer, die zur Laufzeit nicht mehr anpaßbar sind. Manche legen sich auf eine interne Anfragerepräsentation fest oder geben den Optimierungsablauf vor. Mit ihnen lassen sich zwar immer SQL-artige Anfrageprozessoren für ein zentralisiertes DBMS erstellen, Spielraum für andersgeartete Systeme (z.B. verteilte) ist jedoch kaum vorhanden. Auch die Optimierung selbst komplett verteilt ablaufen zu lassen, ist nach meinen Untersuchungen bisher kaum angedacht worden — lediglich der Einsatz paralleler Suchverfahren wurde z.B. in [Lan98] betrachtet.

Schlußfolgernd kann man sagen, daß in den Systemen viele Ideen und Konzepte stecken, die man vereinen sollte. Für die regelbasierte Umformung wären dies beispielsweise die priorisierten Regelmengen aus Starburst, die Regelspezifikationssprache von Prairie und ein Klassenkonzept ähnlich zu OPT++ mit vorgefertigten Suchstrategien. Weitere Forderungen an ein neues Konstruktionswerkzeug wären noch eine durchgängige Anpaßbarkeit der einzelnen Bestandteile und es dürfte nur ein Minimum an Eigenleistung für die Entwicklung erforderlich sein, was einen hohen Grad an Wiederverwendbarkeit mit sich bringt.

Die vorgestellten Systeme dieses Kapitels sind (natürlich) nicht die einzigen: Weitere sind z.B. TWIG [Tji94], ein Parser-Generator, der durch seine Fähigkeit, Bäume zu durchsuchen, auch für die Anfrageverarbeitung geeignet ist und im Lehrstuhlprojekt $\mathrm{HE_AD}$ eingesetzt wurde. Fegaras entwickelte OPTGEN [Feg99], auch ein regelbasiertes System, das sich auf eine attributierte Grammatik stützt, und Cascades [Gra95] ist eine Weiterentwicklung von Volcano.

Kapitel 3

Design

Dieses Kapitel beschreibt das Design eines neuen Optimiererbaukastens. Dazu werden zuerst einige kurze Betrachtungen zu Entscheidungen bezüglich des Designs vorgenommen, die in die Vorstellung eines Modells münden. Dieses wird im Verlauf konkretisiert und verfeinert. Die Bestandteile werden detailliert beschrieben, bevor die Einsatz- und Verwendungsmöglichkeiten des entworfenen Systems den Abschluß des Kapitels bilden.

3.1 Generelles Modell

Der Motivation zur vorliegenden Arbeit ist zu entnehmen, daß eine Bibliothek von Optimiererbausteinen es unter Umständen ermöglichen würde, den Optimierungsprozeß leichter anzupassen oder zu verteilen. Es ist daher nötig, eine gewinnbringende Aufteilung jetzt vorzunehmen, um die in der Motivation genannten Vorstellungen zu erfüllen.

Die existierenden Systeme aus Kapitel 2 liefern dazu zwei wesentliche Ansatzpunkte: Erstens besitzen alle Systeme grob betrachtet einen regelbasierten Ansatz. Und zweitens verläuft das Umschreiben einer Anfrage bzw. das Generieren eines Planes teilweise unter der Kontrolle einer Art Suchmechanismus. Man muß aber in jedem Fall darauf achten, daß die Architektur des neuen Systems nicht zu starr ist. So konnte beispielsweise beim System OPT++ der Suchalgorithmus nur mit den drei Klassen zur Umformung des Anfragebaumes zusammenarbeiten. Das schließt andere Optimierungsaspekte wie eine Rechnerknotenzuordnung zu Operatoren[1] aber aus. Volcano ist sogar als sehr starr zu bezeichnen.

Grundlage einer neuen Optimiererbibliothek sollen Module bilden. Jedes Modul realisiert einen Teil der Gesamtfunktionalität und kann aufgrund einer zu definierenden Schnittstelle leicht gegen ein verwandtes Modul ausgetauscht werden, ohne das restliche System zu beeinflussen. Der Datenbank-Implementierer (DBI) ist somit flexibel, das heißt, er kann aus einer Menge von Optimiererkomponenten schnell einige auswählen und nur die Komponente erneuern, deren Funktionalität er verändern will.

Für einen Optimiererbaukasten ergeben sich folgende Anforderungen:

- Der Optimierungsprozeß ist in funktionale Einheiten (Module) zu zerlegen.
- Diese Module müssen austauschbar sein. Das ermöglicht einen flexiblen Entwurf.
- Module müssen auch zur Laufzeit austauschbar sein. Damit wird eine dynamische Anfragebearbeitung ermöglicht. Zum Beispiel kann die Auswahl der Suchstrategie dann dynamisch erfolgen und es wäre möglich, eine Strategie einzusetzen, bei der bei weniger als sechs JOINs in der Anfrage der Suchraum erschöpfend durchsucht und andernfalls mit Simulated Annealing erforscht wird.

[1] Siehe HE$_A$D.

- Module benötigen eine definierte Schnittstelle, um eine schnelle und einheitliche Austauschbarkeit zu gewährleisten.

Um diese Anforderungen zu realisieren, wird in den nächsten Abschnitten ein Modell für einen Optimiererbaukasten entworfen, welches sich streng an einer funktionalen Aufteilung orientiert.

3.1.1 Modularisierungsvorschlag

Die nun folgende Aufteilung ist bewußt sehr allgemein gehalten. Das hat seinen Grund unter anderem darin, daß es eine Vielzahl von Optimiererlösungen gibt, die alle speziell auf ihr Datenbanksystem zugeschnitten sind. Jedes DBMS besitzt eine eigene Anfragesprache, speziell auf die Speicherungsstrukturen zugeschnittene Algorithmen und es werden Informationen bzw. Statistiken aus anderen Teilen des DBMS benutzt. Man denke hier an verschiedene Arten der Lastmessung einer Betriebssystemressource[2], und wie diese Parameter in das Kostenmodell eines Optimierers einfließen sollten.

Die Unterschiede und Einschränkungen der bisherigen „Baukasten"-Lösungen sind ebenfalls für diesen allgemeinen Modularisierungsvorschlag verantwortlich. Als Ausgangspunkt diente das System OPT++ mit seinen abstrakten Klassen.

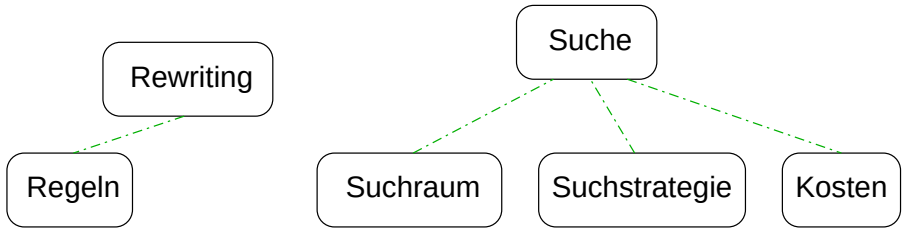

Abbildung 3.1: Komponenten bzw. Module eines Optimiererbaukastens.

Abbildung 3.1 zeigt die Aufteilung des Optimierungsprozesses. Insgesamt ließen sich sechs Komponenten identifizieren: Rewriting, Regeln, Suchraum, Suchstrategie, Kosten und Suche. Dabei übernehmen diese Komponenten, die die Basis für die Module darstellen, folgende Aufgaben:

Rewriting. Ein Modul vom Typ Rewriting ist für das Umschreiben der Anfrage verantwortlich. Dazu wird sich hinter dieser Komponente in den meisten Fällen ein Pattern-Matcher verbergen, der einen Anfragebaum mit Hilfe eines Regelmechanismus' in einen alternativen Baum transformiert. Dieses Modul beschränkt sich nicht nur auf regelbasiertes Umschreiben, sondern soll auch Umformungen erlauben, die sich nicht mit Regeln realisieren lassen.

Regeln. Ein Regeln-Modul repräsentiert Regelmengen. Für ein konkretes Teilproblem der Optimierung werden hier die entsprechenden Regeln zusammengefaßt. Diese Komponente gewährleistet, daß ein Rewriting-Modul mehrfach, jeweils mit einem anderen Regeln-Modul, verwendet werden kann.

Suchraum. Um einer Suchstrategie einen einheitlichen Zugriff auf den Raum der zu einer Anfrage gehörenden Pläne, den Suchraum, zu ermöglichen, verbergen sich hinter diesem Modul einzelne Suchraumdimensionen. Eine Suchraumdimension steht dabei für eine Komponente, die einen bestimmten Aspekt der Optimierung behandelt —

[2]Die Lastmessung soll nicht als Teil eines Optimierers angesehen werden, da ja z.B. auch die Pufferverwaltung einen günstigen Moment für das Zurückschreiben ihrer Puffer bestimmen muß.

es kann sich dabei um die Zuordnung von Algorithmen zu Operatoren im Anfragebaum oder eine Joinfolgeoptimierung handeln. Die Idee hinter dieser Definition ist die Entkoppelung der Suchstrategie vom konkreten Suchraum.

Suchstrategie. In dieser Komponente sollen die verschiedenen Suchstrategien zusammengefaßt werden. Dies wären beispielsweise Iterative Improvement (II), Simulated Annealing (SA) oder ein erschöpfendes Verfahren. Sie nutzen das durch das Suchraum-Modul angebotene Interface, um den Suchraum zu erforschen. Dabei wird auch das Kosten-Modul zur Bestimmung von Plankosten genutzt.

Kosten. Diese Komponente enthält das Kostenmodell, welches in der Regel auf ein konkretes DBMS zugeschnitten ist. Der Wiederverwendungswert eines Kosten-Moduls ist eher schwach, da viele unterschiedliche Informationen verwendet werden, z.B. DB-Statistiken, die aus den Interna der jeweiligen DBMS beschafft werden müssen und architekturabhängig sind. Hauptbestandteil dieser Komponente ist eine Funktion, die zu einem Plan dessen Kosten berechnet.

Suche. Das Modul Suche kombiniert die drei Module Suchraum, Suchstrategie und Kosten. Es erlaubt die Kontrolle und Steuerung der Suche. Dies erweist sich als nützlich, falls bestimmte Anpassungen der drei Module zur Laufzeit erfolgen müssen, diese gar ausgetauscht werden (siehe MoVi-Projekt), oder wenn eine Bewertung der Optimierung vorgenommen werden soll (siehe [Jon98]).

Die in Abbildung 3.1 eingezeichneten Linien deuten die Abhängigkeiten zwischen den Modulen an. Es ist zu sehen, daß des Rewriting-Modul mit dem Regeln-Modul zusammenarbeitet. Damit läßt sich schon eine regelbasierte Optimierung realisieren. Das Suche-Modul vereint die restlichen drei Module unter sich. Dies bedeutet, daß das Suche-Modul nicht nur die Steuerung übernimmt, sondern auch, daß die Suchstrategie auf Funktionen des Suchraum- und des Kosten-Moduls zurückgreift. Nicht eingezeichnet ist allerdings, daß auch eine Rewriting-Modul Teil eines Suchraumes sein kann (siehe Abschnitt 3.2.3).

Nachfolgend sollen die sechs Komponenten im einzelnen betrachtet werden.

3.2 Spezielle Module

In diesem Abschnitt werden nicht nur die in Abbildung 3.1 vorgeschlagenen Module erläutert, sondern auch Verfeinerungen zu diesen. Desweiteren sollen die Schnittstellen der Module festgelegt werden.

3.2.1 Module Rewriting und Regeln

Die ersten beiden Module, die jetzt genauer beschrieben werden, sind Rewriting und Regeln, denn in ihnen spiegelt sich das in den letzten Jahren sehr häufig eingesetzte Verfahren der regelbasierten Anfrageumformung wider. Bei dieser Art der Anfrageverbesserung werden — ausgehend von einem Algebraausdruck für die Anfrage — Operatoren in diesem Algebraausdruck mit Hilfe von Regeln ersetzt. Diese Regeln müssen sicherstellen, daß man eine Anfrage erhält, die dasselbe Ergebnis liefert wie diejenige vor der Umformung. Um nun eine Verbesserung der Anfrage zu erzielen, verfolgen die Regeln bestimmte Heuristiken, die die Optimierungsziele widerspiegeln.

Die Nutzung von Regeln hat den großen Vorteil, erweiterbar zu sein. Aufgrund der abstrakten Form, in der Regeln unabhängig von einer Programmiersprache angegeben werden können (vergleiche Volcano, Absatz 2.2.1, und Prairie, Abschnitt 2.3), ist ein Regelsystem leicht modifizierbar.

Mit dem auf dem Regelmechanismus basierenden Rewriting-Modul sollen sich folgende Arten von Umformungen vornehmen lassen:

1. Umformungen aufgrund algebraischer Äquivalenzen
2. Methodenzuweisungen
3. Planumformungen

Die erste Umformungsart kann auch als logische Umformung bezeichnet werden, da man sich nur auf Eigenschaften der Anfragealgebra stützt und im Allgemeinen keine Bewertung der erzeugten Anfragen vornimmt. Es werden hier nun Regeln für das Separieren, Gruppieren, Kommutieren und für die Reihenfolgeänderung logischer Operatoren aufgestellt, wobei mit Hilfe dieser Regeln eine konkrete Strategie entworfen werden kann, die immer ein Ziel verfolgt: die Verringerung von Tupelanzahlen und der zu verarbeitenden Attribute. Außerdem muß auf die Vermeidung von Redundanzen geachtet werden.

Bei der zweiten Umformungsart, der Methodenzuweisung oder logisch-physischen Transformation, werden den logischen Operatoren des anfänglichen Algebraausdrucks Algorithmen bzw. Methoden (physische Operatoren) zugeordnet, die die Funktionalität realisieren. Dabei gibt es in der Regel mehrere Alternativen für einen logischen Operator und es können mehrere logische Operatoren zusammengefaßt werden. Komplett generierte Pläne werden durch eine Kostenfunktion bewertet, die ermittelt, welchen Verarbeitungsaufwand der Plan erzeugt.

Während die beiden ersten Umformungsarten das Anfrage-Rewriting im Wesentlichen ausmachen, können die Planumformungen als dritte Variante noch Verbesserungen am bereits generierten Plan erzielen. Es können unter Umständen noch Algorithmen durch andere ersetzt werden, die sich vorher nicht direkt einsetzen ließen. Man denke hier an den Enforcer-Mechanismus von Volcano.

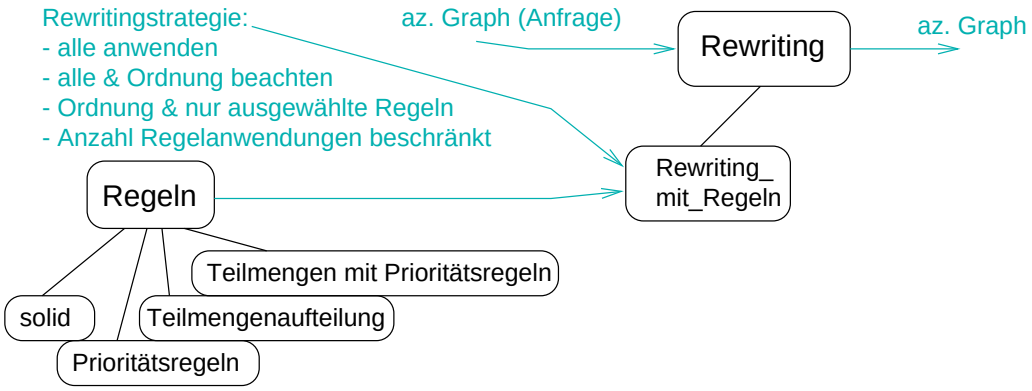

Abbildung 3.2: Module Rewriting und Regeln und Verfeinerungen.

In Abbildung 3.2 kann man noch einmal einen Blick auf die Module Rewriting und Regeln werfen. Zusätzlich sind ein Modul Rewriting_mit_Regeln und mehrere Regeln-Verfeinerungen abgeleitet worden. Dabei steht Rewriting_mit_Regeln für die regelbasierte Verarbeitung einer Anfrage. Hier kann nun beispielsweise ein Mustererkennungssystem für (Anfrage-)Bäume integriert werden, das alle drei oben genannten Umformungsarten beherrscht. Prinzipiell sieht diese Umformungskomponente wie in Abbildung 3.3 aus. Es handelt sich um eine Art Inferenzmaschine, die aus bekannten Fakten (Anfragegraph) neues Wissen (optimierter Anfragegraph) herleitet. Dazu wendet sie die Regeln nach einer Problemlösungsstrategie an. Die Strategie (Rewritingstrategie in Abbildung 3.2) richtet sich danach, ob eine Reihenfolge für die Regelanwendungen existiert bzw. wie die Regelmenge strukturiert ist. Neben der einfachen Vorgehensweise, bei der in jedem Schritt alle Regeln auf Anwendbarkeit geprüft werden, existieren weitere Auswahlstrategien wie z.B. die Agendakontrolle, bei der konkurrierende Regeln, denen eine Priorität zugeordnet wurde, einen Konflikt selbst auflösen, das

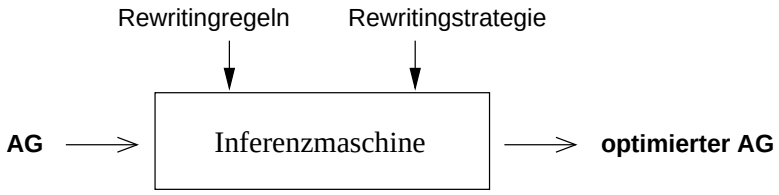

Abbildung 3.3: Architektur einer Rewriting-Komponente.

heißt, daß die erfolgversprechendere Regel ausgewählt wird und zu einer besseren Umformung führt. Um flexibel zu bleiben, ist von Zeit zu Zeit eine Neuberechnung der Prioritäten angeraten[3]. Um die Regelverarbeitung unter anderem auch effektiv zu gestalten, sind also die folgenden Aspekte relevant:

- Strukturierung der Regelmenge
- Regelnotation
- Kontrollmöglichkeiten (Rewritingstrategie)
- Terminierung

Die Arten, die Regeln zu strukturieren, betreffen natürlich das Regeln-Modul. Dieses dient als Container für Regeln und es kann durch Kombination mehrerer Regeln-Module problemspezifisch die aktuell zu verwendende Regelmenge zusammengesetzt werden. Schlußfolgernd aus den vorangegangenen Betrachtungen ist es notwendig, dieses Modul ebenfalls zu verfeinern. Ein Vorschlag ist in Abbildung 3.2 enthalten. Dort gibt es ein Modul für eine einfache Menge von Regeln (*solid*), ein Modul, in dem Regeln einen Prioritätswert tragen (*Prioritätsregeln*), ein Modul das mehrere Regeln gruppiert (*Teilmengen*) und ein kombiniertes Modul, in dem Regeln gruppiert und mit einer Priorität versehen wurden. Damit wird es möglich, in der Rewriting-Komponente eine einfache Auswahlstrategie, sowie die Agendakontrolle zu verwenden.

Das Terminierungskriterium garantiert den Abbruch des Umformungsprozesses. Bei Anwendung gerichteter Regeln ergeben sich hier keine Besonderheiten, da ja stets eine Verbesserung erzielt werden soll und Regeln sich normalerweise nicht gegenseitig aufheben. Allerdings könnte es interessant sein, den Prozeß bei Erreichen einer bestimmten Marke abzubrechen, um den Aufwand von vornherein festzulegen. Dabei bliebe zu beachten, daß eine konsistente Anfrage zurückgeliefert wird.

Bisher war mit Anfrage-Rewriting immer vom regelbasierten Ansatz die Sprache. Dieser Ansatz ist jedoch nicht zwingend und ist nur eine Spezialisierung des generellen Rewriting-Moduls. Es gibt beim Rewriting durchaus Bereiche, bei denen ein Modul direkt in einer Programmiersprache verfaßt wird und lediglich gefordert werden muß, die Schnittstelle des Rewriting-Moduls einzuhalten. Ein Beispiel hierzu ist noch im Verlauf dieses Abschnitts zu finden.

Schnittstellen. Zur Ein- und Ausgabe gehört beim Rewriting-Modul in jedem Fall die Datenstruktur, die die Anfrage repräsentiert. Das wird meistens ein Baum mit Operatorknoten sein, da sich Anfragealgebren und -kalküle gut darauf abbilden lassen. Eine zweite Möglichkeit wäre ein azyklischer Graph, der entsteht, wenn gleiche Teilbäume zusammengefaßt werden. Folgendes Codefragment bezeichnet das Rewriting-Modul:

```
modul Rewriting {
```

[3] Die Regelstrukturierung wird in [KPH98] eingehend betrachtet und ist auch in [Mit95], S. 221 zu finden.

```
void init( anfrage, abbruchzeit ) {...}
// Initialisierung mit der umzuformenden Anfrage und einem
//    Abbruchkriterium

AnfrageTyp rewrite() {...}
// startet Rewrite und gibt eine neue Anfrage zurueck

}
```

Das Rewriting_mit_Regeln-Modul als Spezialfall des Rewriting-Moduls benötigt desweiteren Parameter zur Strategieauswahl und die Angabe eines Regeln-Moduls.

```
modul Rewriting_mit_Regeln {

void init( anfrage, regeln, strategie, maxRegeln ) {...}
// Initialisierung mit Anfrage, einem Regeln-Modul, einer
//    Auswahlstrategie fuer Regeln und einem Abbruchkriterium (die
//    maximale Anzahl auszuwertender Regeln)

AnfrageTyp rewrite() {...}
// startet Regelinterpreter oder aehnliches und
//    gibt umgeformte Anfrage zurueck

}
```

Der Strategie-Parameter gibt dabei an, wie Regeln verwendet werden sollen, sofern das konkrete Modul und die Regeln dies ermöglichen. Ein Regeln-Modul kann nur genutzt werden, falls es die gewünschte Strategie unterstützt. Die Funktion `rewrite()` nimmt je nach Strategie entweder nur eine Regelanwendung vor, oder führt alle Regeln aus — das ist vom Terminierungskriterium abhängig.

Kleines Beispiel

Das in Abbildung 3.4 angegebene Beispiel ist an den Optimierer des Lehrstuhlprojektes $\text{HE}_{\text{A}}\text{D}$ angelehnt. Dazu wurden im Bild die konkreten Module in kursiver Schrift angegeben. Die beiden durchgezogenen Pfeile sind Parameter, die gestrichelten Pfeile stehen für die

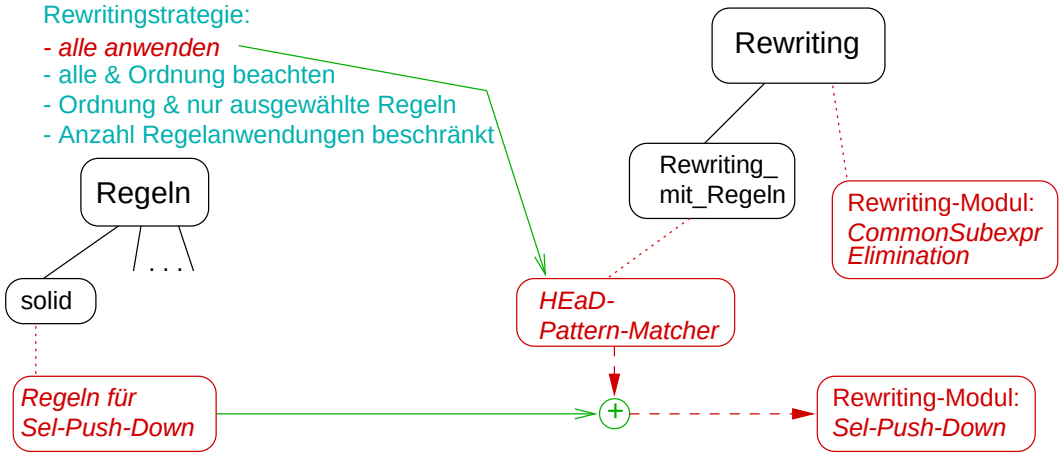

Abbildung 3.4: Beispiel für konkrete Rewriting-Komponenten.

Konstruktion eines zusammengesetzten Rewriting-Moduls. Der Rest des Bildes entstammt der Abbildung 3.2.

Viele Umformungsschritte werden in HE$_A$D mit Hilfe eines externen[4] Pattern-Matcher realisiert. Diesen könnte man in ein Rewriting-Modul integrieren, das dann zusammen mit einem Regeln-Modul eine Phase des HE$_A$D-Optimierers realisieren würde (Selection-Push-Down im Bild). Desweiteren wird in HE$_A$D die Eliminierung gemeinsamer Teilausdrücke durch C++-Code realisiert — zu sehen an der direkten Ableitung vom Rewriting-Modul.

An dieser Stelle ist noch eine Bemerkung zum Rewriting_mit_Regeln-Modul angebracht: Einen Regelinterpreter *„from scratch“* zu entwerfen, ist mit hohem Entwicklungsaufwand verbunden, den man einsparen kann, wenn man stattdessen ein bestehendes Produktionsregelsystem einsetzt. In HE$_A$D wurde dies mit TWIG ([Tji94]) gemacht, im Optimierer von Starburst ist dagegen eine Eigenentwicklung enthalten.

3.2.2 Modul Suchstrategie

Eine Suchstrategie legt im Zusammenhang mit der Anfrageoptimierung die Vorgehensweise für die Ermittlung des effizientesten Planes fest. Es kommen Suchverfahren zum Einsatz, die einen Zustandsraum voraussetzen. Dieser Raum, der alle Pläne zu einer Anfrage enthält — der Suchraum, wird nach einem Ziel, dem optimalen Plan, durchsucht. Werden erst durch das Suchverfahren neue Zustände erzeugt, entsteht ein Suchbaum von Zuständen. In diesem Fall lassen sich blinde, breitenorientierte und tiefenorientierte Verfahren unterscheiden. Ganz allgemein spricht man aber von deterministischen und nichtdeterministischen Verfahren. Eine Übersicht über in der DB-Optimierung verwendete Verfahren ist in der Dissertation von Uwe Langer [Lan98] zu finden.

Bisher hält sich der Einsatz von beliebigen (kombinatorischen) Suchverfahren in existierenden Systemen noch in Grenzen. In den meisten Fällen wird ein Algorithmus gewählt, der fest mit dem Optimierer verbunden ist (siehe Volcano und viele kommerzielle DBMS wie Oracle usw.). Nichtsdestotrotz lohnt sich der Einsatz verschiedener Suchverfahren, denn die meisten Verfahren sind immerhin noch besser als eine vollständige Suche im gesamten Suchraum[5].

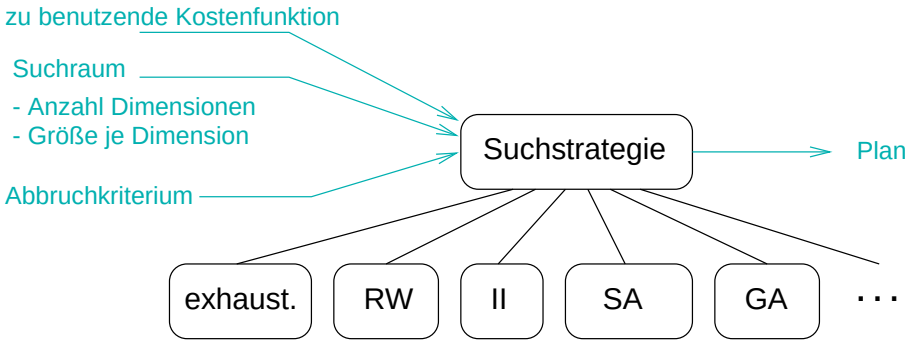

Abbildung 3.5: Das Modul Suchstrategie.

Ein Suchverfahren ist in der Regel mehrdimensional. Das ergibt sich aus den verschiedenen Ebenen der Umformung (logische Umformung und Planerzeugung) bzw. den verschiedenen Optimierungszielen, wie z.B. der Zuordnung von Ausführungsoperatoren zu Rechnerknoten[6].

[4] Benutzt wurde TWIG, ein Tool zur Manipulation von Bäumen (siehe [Lün95]).

[5] Dabei ist die Anpassung an „unser“ Optimierungsproblem allerdings nicht zu vernachlässigen (insbesondere bei Simulated Annealing und genetischen Algorithmen).

[6] In HE$_A$D existieren beispielsweise fünf Dimensionen, siehe [Lan98], Abbildung 22, S. 65.

Einen Vorschlag für die Realisierung eines Suchstrategie-Modules enthält Abbildung 3.5. Auf der linken Seite sind die Parameter für dieses Modul zu sehen. Dazu gehört der Suchraum (siehe Abschnitt 3.2.3), eine verwendbare Kostenfunktion (Abschnitt 3.2.4) und ein Abbruchkriterium. Beim Suchraum-Parameter ist darauf zu achten, daß Informationen über die Anzahl der Dimensionen und die Größe je Dimension zur Verfügung stehen, denn einige Algorithmen könnten nicht nur auf einen eindimensionalen, sondern auch auf einen zwei- oder dreidimensionalen Raum anwendbar sein. Eine Kostenfunktion muß vom Kosten-Modul bereitgestellt werden, und das Abbruchkriterium begrenzt den Suchaufwand, wobei es bei jedem Verfahren eine unterschiedliche Ausprägung besitzen wird. Im unteren Bereich der Abbildung 3.5 sind einige abgeleitete Module dargestellt. Dabei bedeuten die Bezeichnungen exhaust. erschöpfende Verfahren, RW Random Walk-Verfahren, II Iterative Improvement, SA Simulated Annealing und GA genetische Algorithmen. Zu diesen Verfahren gibt es jeweils noch Varianten, die von ihrem Basis-Modul abgeleitet werden können.

Als nächstes folgt eine Vorstellung der genannten Suchverfahren in Anlehnung an [Ros96].

Exhaustive Search

Die erschöpfende Suche (*exhaustive search*) ist ein sehr einfaches Verfahren. Es betrachtet den gesamten Suchraum, durchsucht ihn also vollständig. Dabei werden zu jedem Plan die entsprechenden Kosten ermittelt. Der Plan mit den geringsten Kosten bildet das Ergebnis. Dies ist immer das globale (Kosten-) Minimum. Im einfachsten Fall liegt bereits eine Menge von Plänen vor, die lediglich bewertet und verglichen werden müssen, die Pläne können aber auch erst durch das Suchverfahren erzeugt werden — das hängt von der Gestaltung des Suchraumes ab (siehe später). Zum Zugriff auf den Suchraum würde eine Funktion wie GET_NEXT(dim) genügen, um den nächsten Plan in der Dimension zu bestimmen. Das Abbruchkriterium bildet bei diesem Verfahren eine Zeitschranke, im Falle eines erst zu erzeugenden Suchraumes könnte man auch die maximale Anzahl Pläne festlegen, wobei der bis dahin gefundene kostenminimale Plan zurückgeliefert würde. Eine Zeitschranke sollte immer zusätzlich gewählt werden, um trivialerweise auch Endlosschleifen auszuschließen.

Dieses Verfahren läßt sich gut bei kleinen Suchräumen anwenden, jedoch erhöht sich die Laufzeit sehr stark bei komplexen Anfragen oder großen Räumen. Da alle Pläne besucht werden und immer das globale Minimum gefunden werden kann, wird dieses Verfahren oft zu Vergleichen mit anderen Verfahren herangezogen.

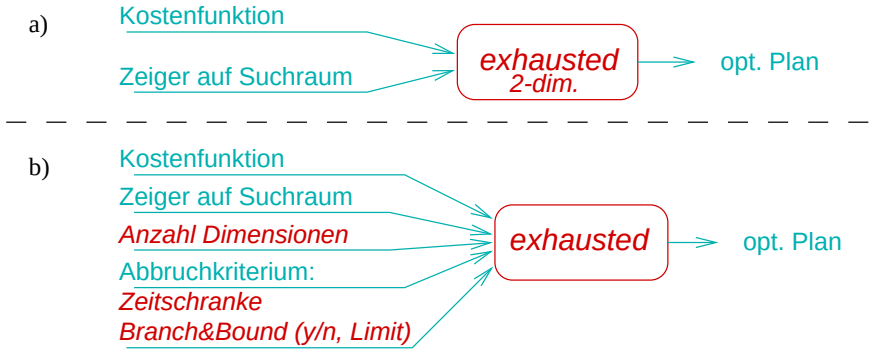

Abbildung 3.6: Module zu Exhaustive Search.

Abbildung 3.6 zeigt zwei Formen eines ExhaustiveSearch-Moduls. Die erste Variante (a) beschränkt sich auf zweidimensionale Räume, wie sie im „klassischen“ Fall mit logischem Rewriting und einer Planerzeugungs-/Planumformungskomponente auftreten. In der logischen Dimension wird durch die algebraische Umformung eine äquivalente Anfrage erzeugt,

der in der physischen Dimension mehrere Pläne zugeordnet werden. Die Suche ist beendet, wenn sich keine Pläne mehr erzeugen lassen.

Das zweite Version (b) zeigt ein universelles Modul, daß mit Räumen unterschiedlicher Dimensionalität arbeiten kann und nach einer bestimmten Zeit (Antwortzeit) den bis dahin besten betrachteten Plan zurückliefert. Neben den zusätzlichen Parametern für diese Besonderheiten, ist auch ein Schalter für *branch & bound pruning* vorhanden. Diese Technik schränkt den Suchraum insofern ein, daß das Rewriting bzw. die Planerzeugung an anderer Stelle fortgesetzt wird, wenn ein Kostenlimit überschritten wird. Dies muß in enger Zusammenarbeit mit der Suchraum-Komponente erfolgen. Branch & bound pruning beschleunigt zwar die Suche, unter Umständen wird das globale Minimum jedoch nicht mehr gefunden.

Random Walk

Das Suchverfahren Random Walk(RW) ist auch ein relativ einfaches Verfahren. Es wird eine rein zufallsbasierte Suche durchgeführt. Zu Beginn wird ein zufälliger Startpunkt im Suchraum gewählt. Dieser Plan stellt zugleich die erste Lösung dar und wird als Optimum vorgemerkt. Danach wird Schritt für Schritt zufällig ein neuer Punkt ausgewählt. Dieser wird als Lösung (optimaler Plan) gespeichert, falls er ein neues Kostenminimum darstellt. Man kann sich dieses Verfahren als ein zufälliges Hineingreifen in den Suchraum vorstellen, wobei die Schritte unabhängig voneinander sind. Abbildung 3.7 veranschaulicht dieses Suchverfahren.

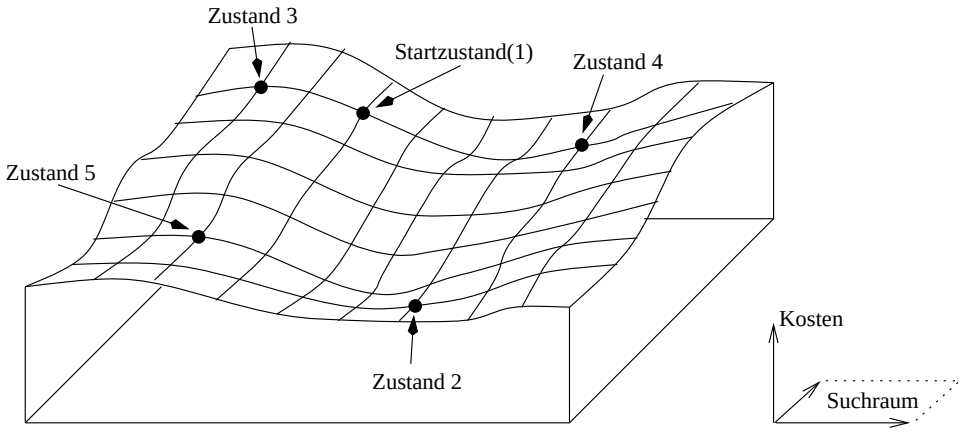

Abbildung 3.7: Random Walk.

Als Abbruchkriterium dient die maximale Anzahl von zu betrachtenden Plänen, man könnte aber auch vorzeitig abbrechen, wenn man eine Kostenschranke durch eine vorherige Abschätzung festlegen kann. Bei Random Walk-Verfahren kann nicht garantiert werden, daß überhaupt ein lokales Minimum gefunden wird.

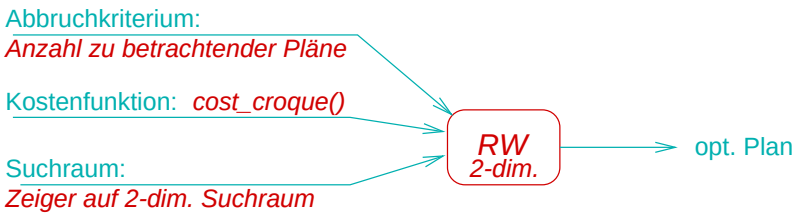

Abbildung 3.8: Ein Modul zu Random Walk.

Für die Durchführung dieses Verfahrens ist es notwendig, zufällig in den Suchraum greifen zu können, d.h., daß ein beliebiger Plan vom Suchraum geliefert werden muß, um anschließend durch die Kostenfunktion bewertet zu werden. Abbildung 3.8 zeigt ein Random-Walk-Modul.

Iterative Improvement

Das Verfahren Iterative Improvement (II) ist wie Random Walk ebenfalls ein zufallsbasiertes Suchverfahren. Es geht zwar auch von einem zufällig gewählten Startzustand aus, sucht dann aber lokal nach einem Minimum. Ein zufällig gewählter Nachbar wird immer dann zum neuen Ausgangspunkt erhoben, wenn seine Kosten geringer als die des aktuellen Zustandes sind. Vom neuen Ausgangspunkt wird dann wieder zufällig ein Nachbar ausgewählt. Kann zu keinem neuen Zustand verzweigt werden, wurde ein lokales Minimum gefunden und die Suche ist beendet.

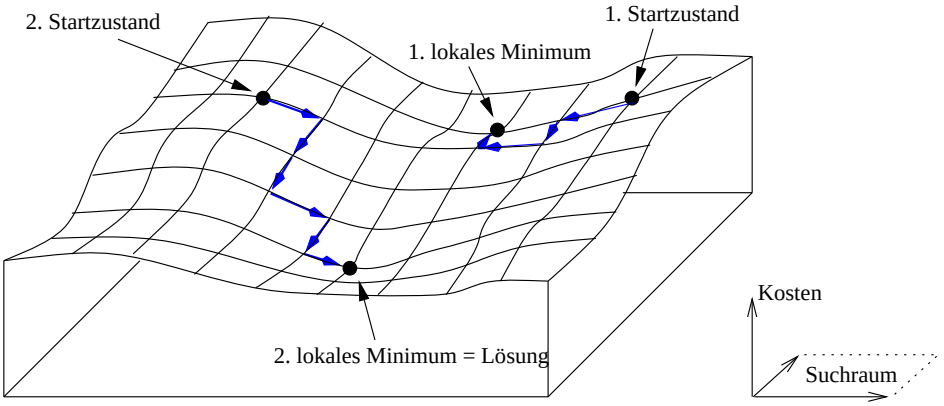

Abbildung 3.9: Iterative Improvement.

Dieses Verfahren kann für eine vorgegebene Anzahl von Startpunkten wiederholt werden, wobei das niedrigste lokale Minimum aller Versuche zurückgeliefert wird. Abbruchkriterien sind somit die Anzahl der Startpunkte und die maximale Anzahl von Versuchen, um zu einem Nachbarn fortzuschreiten. Man kann aber auch die maximale Anzahl der Schritte im Raum oder eine Zeitschranke festlegen. Abbildung 3.9 illustriert dieses Suchverfahren.

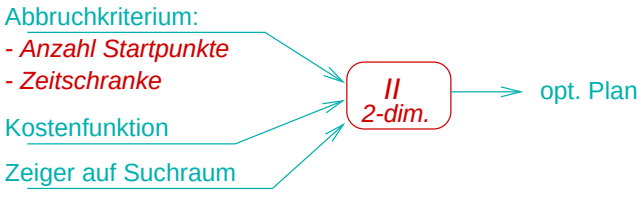

Abbildung 3.10: Ein Modul zu Iterative Improvement.

Bei Iterative Improvement ist es einerseits notwendig, im Suchraum den nächsten Nachbarn zu bestimmen, andererseits müssen zufällig Pläne (Startpunkte) ermittelt werden können. Der Suchraum muß hierfür geeignet sein.

In Abbildung 3.10 ist eine Ausprägung eines Iterative Improvement-Moduls dargestellt, das sich auf einen zweidimensionalen Raum beschränkt, für den die Anzahl von Startpunkten vorgegeben werden kann. Der Parameter *Zeitschranke* begrenzt die Gesamtlaufzeit.

Simulated Annealing

Das Verfahren Simulated Annealing (SA) ähnelt Iterative Improvement, stellt aber einen verbesserten Ansatz dar, da die Einschränkung, daß nur Zustände mit geringeren Kosten neue Ausgangspunkte bilden können, aufgehoben wird. Mit einer gewissen Wahrscheinlichkeit, der aktuellen Temperatur, wird hier auch zu Zuständen mit einem höherem Kostenniveau verzweigt. Dieses Verhalten kann helfen, ein lokales Minimum wieder zu verlassen, um in einem besseren oder sogar im globalen Minimum zu landen. Da die Temperatur im Verlauf der Suche reduziert wird, treten Verzweigungen zu Plänen mit höheren Kosten immer seltener auf. Dieses Verhalten ist in Abbildung 3.11 illustriert.

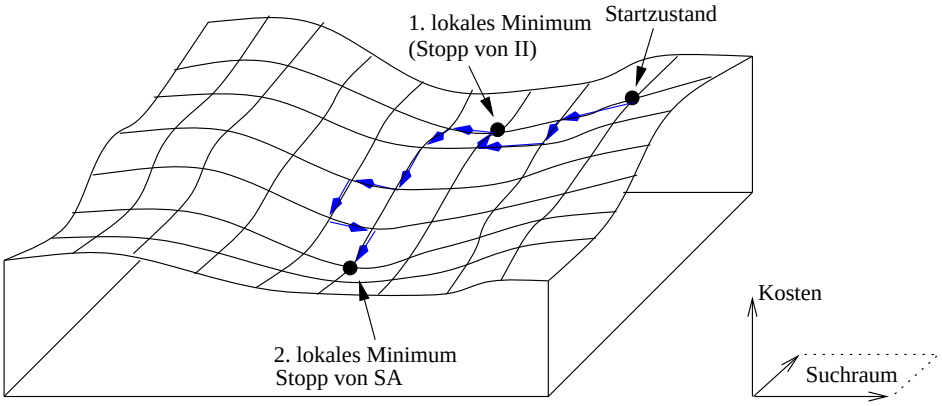

Abbildung 3.11: Simulated Annealing.

Parameter dieses Verfahrens sind die Starttemperatur und der Wert, um den die Temperatur reduziert wird. Das Verfahren bricht ab, falls eine gewisse Anzahl von Nachbarzuständen ergebnislos, d.h. ohne daß es zu einer Verzweigung kommt, besucht wurde, oder wenn eine Gefriertemperatur erreicht wurde.

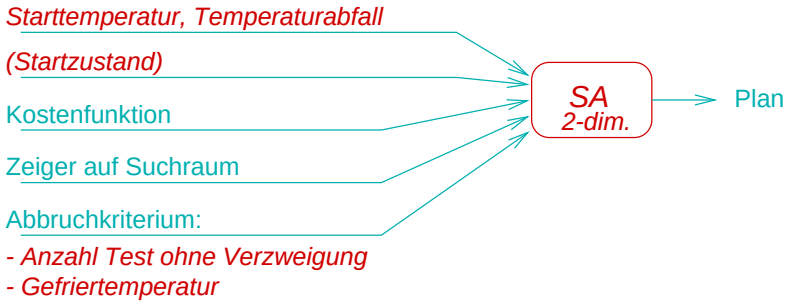

Abbildung 3.12: Ein Modul zu Simulated Annealing.

In der Abbildung 3.12 ist ein Simulated Annealing-Modul für einen 2-dimensionalen Suchraum und mit entsprechenden Parametern angegeben. Ein Parameter, der Startzustand, ist für eine besondere Anwendungsform dieses Verfahrens gedacht: Wurde bereits mit einem anderen Verfahren nach einem lokalen Minimum gesucht, so kann die weitere Umgebung (Radius > 1) mit Simulated Annealing noch nach einem besseren Minimum abgesucht werden.

Weitere Informationen zu Simulated Annealing hält unter anderem auch [Ing99] bereit.

Genetische Algorithmen

Genetishce Algorithmen basieren auf der Simulation des biologischen Evolutionsprozesses. Die Evolution stellt ein Optimierungsverfahren dar, welches durch Manipulation der Erbinformation in der Lage ist, die unterschiedlichsten Lebensformen in relativ kurzen Entwicklungszyklen an ihre Umwelt anzupassen. Sie führt also eine Suche im Raum der genetischen Informationen durch, mit dem Ziel, für die jeweiligen Individuen die optimalen Erbanlagen zu finden.

Genetische Algorithmen arbeiten stets auf einer Menge von Lösungen, einer Population. Lösungen werden als Strings (Chromosomen) dargestellt, die sich aus einzelnen Zeichen (Genen) zusammensetzen. Beim Einsatz dieser Algorithmen spielt eine geeignete Repräsentation der Mitglieder einer Population eine zentrale Rolle. An der Entwicklung von Lebewesen sind ursächlich die Evolutionsfaktoren beteiligt. Dazu gehören Mutation, Rekombination(Crossover) und Selektion (Auslese).

Die allgemeine Vorgehensweise genetischer Algorithmen kann folgendermaßen festgehalten werden:

1. Generieren einer zufälligen Population von Chromosomen (Generation 0)
2. Wiederhole 3.-6. bis
   - eine gewünschte Qualität eines Individuums erreicht wurde
   - eine maximale Anzahl Generationen erzeugt wurde
   - der Qualitätszuwachs von einer Generation zur nächsten unter eine Schranke fällt
3. Bewertung aller Mitglieder der aktuellen Generation mittels einer Bewertungs- und / oder Fitneßfunktion
4. Selektion der fittesten Mitglieder und Erzeugung von Nachkommen mittels Crossover
5. Mutation der Nachkommen
6. Ersetzen der Mitglieder der aktuellen Generation durch Nachkommen (neue Generation)

Die Selektion hat das Ziel, die Güte einer Generation durch Aussortieren „schlechter“ Individuen zu erhöhen. Die dazu notwendige Bewertung der Population setzt sich aus einer Bewertungsfunktion, die feststellt, wie nahe ein Individuum dem Optimum ist, und einer Fitneßfunktion, die die Wahrscheinlichkeit für die Teilnahme eines Individuums an der Erzeugung von Nachkommen bestimmt, zusammen.

Beim Crossover wird versucht, aus suboptimalen Individuen mittels Kombination ein besseres Ergebnis zu erhalten. Stark vereinfacht läßt sich dies wie in Abbildung 3.13 veranschaulichen. Wird ein Individuum besonders gut bewertet, weil es die Gene A und B besitzt,

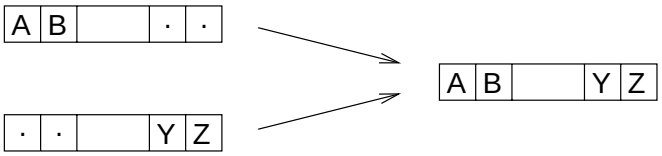

Abbildung 3.13: Crossover (stark vereinfacht).

ein anderes Individuum die guten Gene Y und Z hat, so könnte aus der Kombination von A und B mit Y und Z ein noch besseres Individuum entstehen. Wenn man die einfache

Annahme macht, daß A,B,Y und Z Teilpläne darstellen, liefert die Kombination ein noch besseres lokales Optimum im Plan-Raum.

Die Mutation wird verwendet, um neues genetisches Material in in die Population einzubringen. Gewöhnlich ist die Mutation eine Änderung einzelner Gene oder Genketten. Die Mutation ist mit einem zufälligen Greifen in die Umgebung eines Planes im Suchraum zu vergleichen.

Betrachtet man die genetischen Algorithmen nur unter dem Gesichtspunkt der DB-Optimierung, so stellen sich neben der Frage nach der Repräsentation der Mitglieder einer Population auch verfahrenstechnische Fragen. Wie sollen die Bewertungs- und die Fitneßfunktion im Bezug auf die Optimierung aussehen? Wie muß der Crossover-Mechanismus realisiert werden, so daß keine ungültigen Pläne erzeugt werden? Und wie werden bei der Mutation die Pläne bestimmt — im besonderen, wenn der Suchraum nicht vorberechnet ist? Eine Umsetzung dieses Verfahrens ist somit nicht so einfach wie bei den anderen Suchverfahren, so daß auf einen Modulvorschlag verzichtet wird. Desweiteren ist beim Einsatz dieses Verfahrens zu berücksictigen, daß vor allem sehr große und mehrdimensionale[7]Räume effizient durchsucht werden können.

Schnittstellen

Mit dem Modul Suchraum erhalten die Suchverfahren die Möglichkeit, den Such- bzw. Lösungsraum einheitlich zu durchschreiten. Wurden bereits alle Pläne generiert, so ist nur ein Verweis auf den Suchraum nötig. Die meisten Verfahren benötigen jedoch zusätzlich noch die Anzahl der Dimensionen und die Größe je Dimension. Etwaige besondere Eigenschaften, die die Beschaffenheit des Raumes betreffen, sollen explizit vom Suchraum-Modul beschafft werden. Das betrifft z.B. die Information, ob jeder Punkt des Raumes eine gültige Lösung enthält oder ähnliches.

Weitere Parameter bilden immer ein algorithmenspezifisches Abbruchkriterium und eine zu verwendende Kostenfunktion, die vom Kosten-Modul bereitgestellt werden muß. Ergebnis der Suchstrategie ist die ausgewählte Lösung, und damit der beste gefundene Ausführungsplan.

Ein Codefragment zur Verdeutlichung der Schnittstellen ist wegen der unterschiedlichen Verfahren nur schlecht zu spezifizieren. Stellvertretend dafür stehen aber die Abbildungen der Modulvorschläge zu den einzelnen Verfahren.

3.2.3 Modul Suchraum

Die Suchverfahren wurden bisher relativ losgelöst betrachtet. Die wesentliche Schnittstelle, die aber alle Verfahren benutzen, ist die zum Suchraum. Er enthält die Menge aller zu einer Anfrage erzeugbaren Pläne. Als Suchraumdimensionen können die verschiedenen Arten der Anfrageumformung betrachtet werden. So ergibt sich ein zweidimensionaler Suchraum, wenn regelbasiert äquivalente (logische) Anfragen erzeugt werden (1. Dimension) und ebenfalls regelbasiert zu einer (logischen) Anfrage mehrere zugehörige Pläne generiert werden (2. Dimension). In vielen DBMS werden auch noch weitere Dimensionen hinzugenommen — so hat man in einem verteilen DBMS beispielsweise zu entscheiden, welche Algorithmen für den jeweiligen Operator auf welchem Rechnerknoten auszuführen sind (eine 3. Dimension).

Die Aufgabe des Suchraum-Moduls besteht nun in der Verwaltung dieses Raumes. Dazu sind die Suchraumdimensionen so zu integrieren, daß ein Rewriting-Modul oder ein spezifischer Programmcode, z.B. für die Rechnerknotenzuordnung, unter einer einheitlichen Oberfläche erscheinen. Die Suchverfahren sollen alle möglichst in der gleichen Art und Weise auf

[7]Im Beispiel aus Abbildung 3.13 wurde angenommen, A,B usw. seien Teilpläne. Es scheint aber ebenfalls sinnvoll, von Dimensionen zu sprechen, die den Dimensionen des Suchraumes entsprechen. Daraus folgt dann, daß das Crossover in Räumen mit einer hohen Anzahl von Dimensionen nützlicher ist.

den Raum zugreifen können, d.h. daß Funktionen zum Zugriff auf die Pläne vom Suchraum-Modul bereitzustellen sind. Mögliche Funktionen wären für einen Zugriff auf einen beliebigen Plan im Raum, GET(vector), eine Funktion, um nur entlang einer Dimension zu schreiten, GET_NEXT(dim), oder eine Funktion, die alle Pläne aus einer ε-Umgebung zurückliefert, GET(vector,ε). Welche Zugriffsarten realisiert werden können, hängt aber immer von den zugrundeliegenden Komponenten ab. Mit einem regelbasierten Rewriting-Modul wird man z.B. nicht auf einen beliebigen Plan zugreifen können, wenn nicht alle Umformungen vorher vorgenommen wurden.

Unterscheidet man bei der Suche zwischen einem Lösungs- und einem Suchraum, so ist eine Suchraumabbildungsfunktion notwendig. Man vereinfacht z.B die Suchverfahren, indem man einfache diskrete Verfahren einsetzt, die in einem integerbasierten Raum suchen. Die Umsetzung zwischen diesem Suchraum und dem Plan- bzw. Lösungsraum ist dann ebenfalls im Suchraum-Modul vorzunehmen. Eine Abbildungsfunktion ist im HEAD-Projekt [Lan98] zu finden.

Zu den weiteren Aufgaben des Suchraum-Moduls zählt u.a. auch die Ersetzung von Zugriffsfunktionen für eine spezielle Suchstrategie durch primitive vom aktuellen Suchraum unterstützte Funktionen. Benutzt der Simulated Annealing-Algorithmus beispielsweise nur GO_LEFT(), GO_RIGHT(), GO_UP() und GO_DOWN(), so sind diese Funktionen durch GET_NEXT(d-1) und ähnliche Aufrufe zu ersetzen. Das Suchstrategie-Modul sollte dazu nicht modifiziert werden.

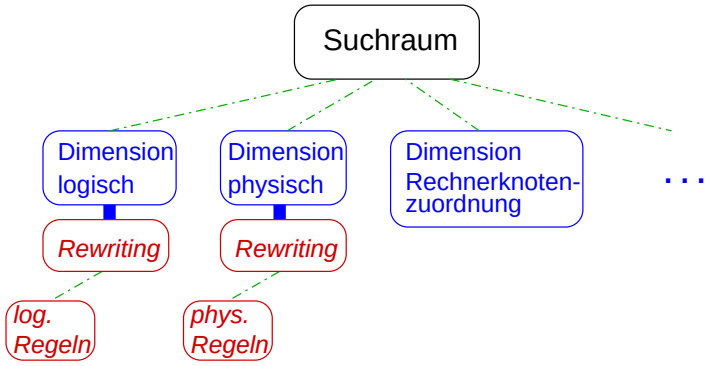

Abbildung 3.14: Das Modul Suchraum.

Abbildung 3.14 zeigt ein Suchraum-Modul. Es ist zu sehen, daß das Suchraum-Modul auch andere Module benutzt, um daraus seinen Raum aufzubauen. Auf eine formale Schnittstellendefinition möchte ich verzichten, es würde sich dabei lediglich um die Zugriffsfunktionen handeln, die oben angerissen wurden.

3.2.4 Modul Kosten

Für die Abschätzung der Abarbeitungskosten werden in einem Kostenmodell in der Regel die Ressourcen berücksichtigt, die bei der Anfrageauswertung in Anspruch genommen werden, so z.B. benötigter Externspeicherplatz, die Anzahl der Ein-/Ausgabeoperationen, benötigter Systempuffer und CPU-Zeit. Die verschiedenen DBMS behandeln diese Parameter jedoch auf unterschiedliche Art und Weise. Die Faktoren sind zum einen nicht nur unterschiedlich gewichtet, zum anderen legt jedes DBMS ganz allgemein andere Schwerpunkte. Handelt es sich z.B. um ein verteiltes DBMS, müssen Kosten für die Kommunikation einfließen, und diese in einem mobilen VDBMS[8] auch stärker gewichtet werden. Um die verschiedenen Einflußgrößen miteinander kombinieren bzw. vergleichen zu können, müssen diese normiert und

[8] Siehe Lehrstuhl-Projekt MoVi [Lub96].

gewichtet werden. Dazu gehört die Festlegung von Einheiten für die Faktoren. Beispielsweise wird festgelegt, daß die Messung der CPU-Zeit nur in Mikrosekunden(μs) erfolgt.

Bearbeitungskosten für einen vollständigen Plan werden rekursiv aus den einzelnen Kostenabschätzungen der Planoperatoren berechnet. Prinzipiell ergibt sich hierdurch die Möglichkeit, jedem Operator bereits eine Kostenfunktion mitzugeben, die standardmäßig einen Kostenwert aus CPU-Zeit und Speicherplatzverbrauch berechnet und die nur um eine Berechnung für die Kommunikationskosten zu erweitern wäre. Man hätte somit Grundbausteine für ein Kostenmodell und müßte nur Formeln für spezifische Kosten hinzufügen. Dieses Verfahren scheint aber angesichts der Vielzahl der Operatoren nicht praktikabel. Auch wird es kaum ein kleinstes gemeinsames Minimum an Standardfaktoren geben, das in allen DBMS sinnvoll ist oder überhaupt berechnet werden kann.

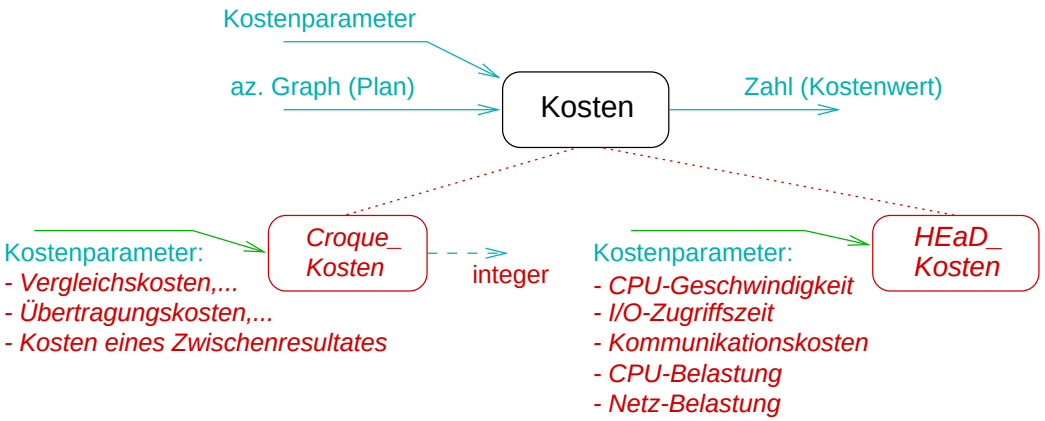

Abbildung 3.15: Das Modul Kosten.

Das Kosten-Modul (siehe Abbildung 3.15) beschränkt sich auf die Ableitung konkreter Kosten-Module, die je ein maßgeschneidertes Kostenmodell für ein DBMS realisieren. Allgemein muß davon ausgegangen werden, daß die gesamte Kostenberechnung abhängig von der Architektur des DBMS ist. Genauer gesagt stellt sich die Frage, wie Informationen zur Kostenberechnung (die Kostenparameter) bereitgestellt werden. Außerdem besteht immer auch eine Abhängigkeit von den konkreten physischen Operatoren.

Schnittstellen. Das Kosten-Modul soll in jedem Fall eine Funktion anbieten, die als einzigen Parameter einen Plan entgegen nimmt und dafür einen Kostenwert zurückliefert. Eine Vorgabe für dessen Format soll nicht gemacht werden. Ob dieser Wert zum jeweiligen Plan in einer Liste vermerkt oder aber als Attribut eines Knotens im Anfragebaum gespeichert wird, ist nicht Sache des Kosten-Moduls und soll an anderer Stelle erfolgen.

```
modul Kosten {

void init( fixe Kosten ) {...}
// Initialisierung mit Kostenparametern

int cost( plan ) {...}
// startet Kostenberechnung und gibt Kostenwert als Integer zurueck
//   benutzt dynamische Kostenparameter

}
```

Die in Abbildung 3.15 angegebenen Kostenparameter treten hier nicht wieder auf. Man kann sie aber in fixe Kostenparameter (CPU-Geschwindigkeit, I/O-Zugriffszeit, ...) und

dynamische Kostenparameter (z.B. die Kommunikationskosten aus der aktuellen Netzlast, ...) unterteilen, die nur bei den speziellen Modulen angegeben werden können.

3.2.5 Modul Suche

Die bisher vorgestellten Module besitzen einen klar abgegrenzten Aufgabenbereich. Das Suche-Modul seinerseits versucht nun, eine optimiererspezifische Steuerfunktion für den Suchvorgang zu übernehmen. Es vereint die drei Module Suchraum, Suchstrategie und Kosten unter sich und kann folgende Aufgaben übernehmen:

Initialisierung der benutzten Module. So kann z.B. die Vorberechnung des Suchraumes angestoßen werden, falls dies notwendig ist, oder die Kostenkomponente mit der aktuellen Lastmessungskomponente verbunden werden.

Kombination zweier Suchstrategien. Wie bereits erwähnt, läßt sich das Verfahren Simulated Annealing gut mit Iterative Improvement kombinieren. Man kann dies zwar auch durch ein neues Suchstrategie-Modul erreichen, sie im Suche-Modul zu vereinen, läßt aber noch die Möglichkeit der Anpassung dieser Kombination offen. Im besten Fall könnte das zu einer dynamischen Auswahl einer Suchstrategie führen.

Anpassungen an spezielle DBMS-Umgebungen. In einem verteilten DBMS will man sicherlich auch die Verteilung der Optimiererkomponenten selbst anstreben. Ein derartiger Algorithmus ließe sich hier integrieren. Ein spezieller Anwendungsfall ergibt sich im mobilen Szenario. Man könnte einen Algorithmus entwickeln, der zur Laufzeit auf einem PDA[9] entscheidet, welche Optimierer-Komponenten (Module) noch auf dem PDA ausführbar sind und welche von einem stationären Gerät übernommen werden müssen. Auch die Konzeption eines Optimierers, der verschiedene Modi für interaktive und vorzuübersetzende Anfragen anbietet, fällt in dieses Aufgabenfeld.

Die Daseinsberechtigung für diese Komponente ergibt sich also aus den doch eher optimiererspezifischen Details, die der Einsatz der anderen Module noch mit sich bringen kann. Nicht zuletzt sind die hier enthaltenen Funktionseinheiten zu einem gewissen Grade auch wiederverwendbar. Die Wiederverwendbarkeit hängt natürlich davon ab, wie speziell dieses Modul gestaltet wird — das heißt, daß lediglich eine Kombination zweier Suchstrategien diesen Aspekt begünstigen wird, eine komplexe Strategie zur dynamischen Auswahl von Modulen eher weniger wiederverwendet werden wird.

3.3 Verwendung des Modulsystems

Mit dem in den letzten Abschnitten erarbeiteten Modulen sollte es nun möglich sein, ein beliebiges Optimierersystem zusammensetzen. Diese Behauptung möchte ich im folgenden Abschnitt anhand eines Beispiels belegen, bevor ich dann zu einer Einschätzung dieses Modulsystems komme.

3.3.1 Einsatzbeispiel

Beim Lehrstuhl-Projekt HE_AD [FLM96] handelt es sich um ein verteiltes DBMS, das eine umfangreiche, aber gut strukturierte Optimiererkomponente besitzt. Diese besteht aus einem Rewriting-System und einer sich anschließenden Suche. Abbildung 3.16 zeigt dieses Optimierersystem.

Zu Beginn der Optimierung wirken diverse Rewriting-Schritte auf die Anfrage. Es werden eine Normalisierung, *Selection-Push-Down*, *Projection-Push-Down* und einige weitere

[9] PDA = Personal Digital Assistant.

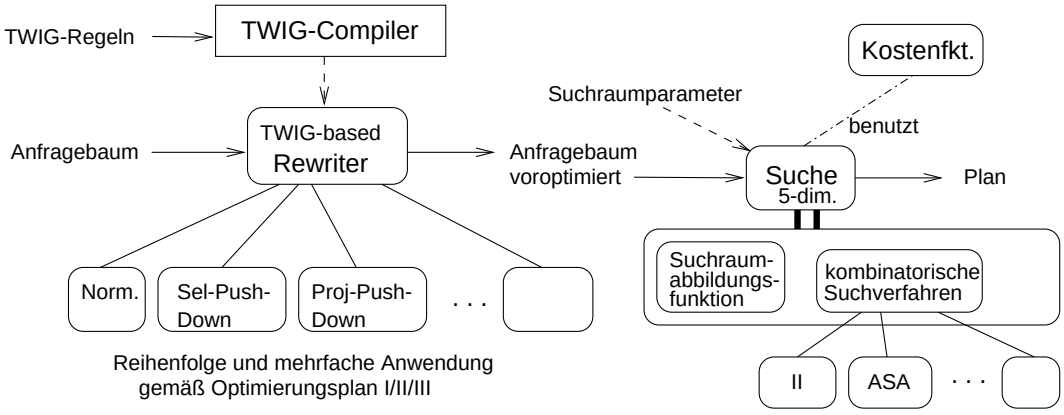

Abbildung 3.16: Komponenten des HE$_A$D-Projektes.

Umformungen durchgeführt, um die Anfrage zu verbessern. Die Anwendung dieser Umformungsschritte erfolgt nach einem vorher festgelegten Umformungsplan. Die Rewriting-Komponenten gründen sich hier jeweils auf eine Regelspezifikation für das allgemeine Baum-Rewriting-System TWIG. Mittels des TWIG-Compilers werden die Regelspezifikationen in Rewriting-Module übersetzt und ergeben entsprechend Umformungsplan I, II, oder III das HE$_A$D-Rewriting-System. Es liefert als Ergebnis eine voroptimierte Anfrage.

Im zweiten Teil der Optimierung wird eine Suche in einem 5-dimensionalen Raum durchgeführt. Dabei entsprechen die Dimensionen folgenden Aufgaben: Zuweisung von Methoden an die Operatoren, Anordnung der Eingangsdatenströme, Bestimmung von Join-Folgen und die Zuordnung von Rechnerknoten an gebundene bzw. ungebundene Operatoren. Es kommen Suchverfahren zum Einsatz, die in einem integer-basierten Raum voraussetzen und der mittels einer Abbildungsfunktion auf den Plan-Raum umgesetzt wird. Es existiert auch ein eigenes Kostenmodell mit entsprechender Kostenfunktion.

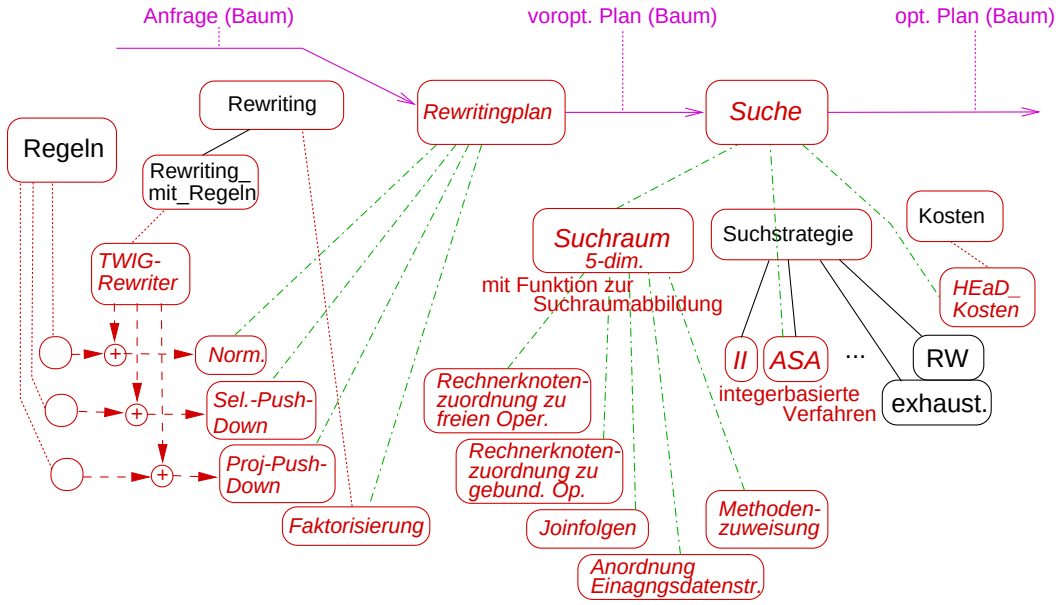

Abbildung 3.17: Module für das HE$_A$D-Projekt.

In Abbildung 3.17 habe ich versucht, das HE$_A$D-System auf das neue Modulsystem umzusetzen. Die zweistufige Optimierung läßt sich auch hier von links nach rechts verfolgen. Im ersten Teil habe ich vom Rewriting_mit_Regeln-Modul den TWIG-Rewriter abgeleitet, der, real zwar nicht sichtbar, aber jeweils zusammen mit einem Regeln-Modul (Regeln im TWIG-Stil) eine Rewriting-Komponente bildet[10]. Mehrere Rewriting-Komponenten werden in der Komponente Rewritingplan in eine feste Abarbeitungsreihenfolge gebracht[11]. Nachdem die Anfrage voroptimiert wurde, wird die Kontrolle an ein Suche-Modul weitergegeben, das in diesem Fall nur für Initialisierungzwecke benutzt wird, denn eine Suchstrategie wird bereits vorher ausgewählt. Diese benutzt dann einen Integer-Suchraum, der im Suchraum-Modul die Abbildungsfunktion voraussetzt. Die Suchraumdimensionen werden durch eigenständige Softwarekomponenten gebildet und die Kostenfunktion wird von einem HE$_A$D-spezifischen Kosten-Modul bereitgestellt.

3.4 Vergleich und Bewertung

Zusammenfassend präsentiert sich der Modularisierungsvorschlag folgendermaßen: Es existiert eine Aufteilung in die sechs Haupt-Module Rewriting, Regeln, Suchstrategie, Suchraum, Kosten und Suche, wobei von einigen spezielle Formen abgeleitet wurden. Diese Modulhierarchie ist noch einmal in Abbildung 3.18 dargestellt.

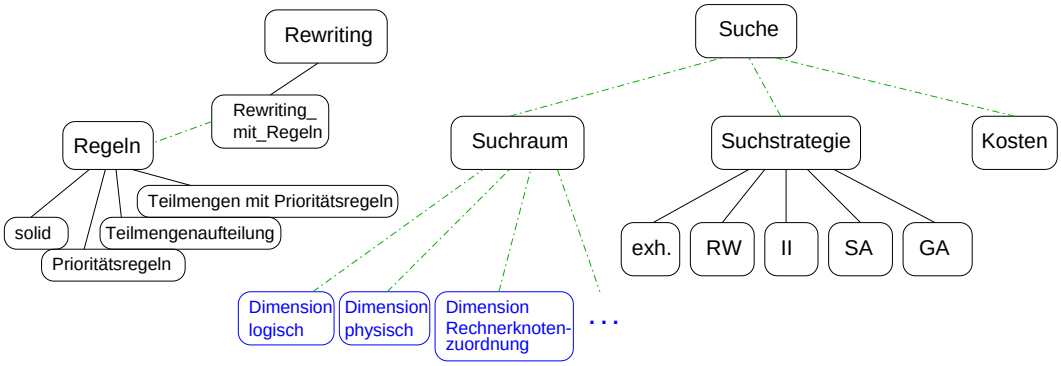

Abbildung 3.18: Module - Gesamtsicht.

Durch einem Vergleich mit den in Kapitel 2 vorgestellten Systemen und einer anschließenden Einschätzung wird sich nun zeigen, inwieweit sich der Entwurf eines Optimierers tatsächlich verbessern bzw. noch weiter vereinfachen läßt, als mit den bisherigen Systemen.

Dazu sind in Tabelle 3.1 erst einmal die sechs Module den Systemen Volcano, Volcano mit Prairie, OPT++ und Starburst gegenübergestellt. Es soll zunächst die Frage geklärt werden, ob sich die existierenden Systeme auf das neue Modulsystem abbilden lassen.

In **Volcano** gibt es eine regelbasierte Anfrageumformung, die das Rewriting der Anfrage, die Planerzeugung und die Planumformung umfaßt. Dafür existieren jeweils eigene Regelformen. Dem gegenüber steht im neuen System ein Rewriting-Modul mit drei Regeln-Modulen, da es nur drei Regelmengen zur gleichen Zeit gibt. Desweiteren existiert eine Suchstrategie (erschöpfende Suche mit *branch&bound-pruning*), die in dem zwei-dimensionalen (Plan-) Raum, nach dem Optimum sucht. Entsprechend gibt es nur ein festes Suchstrategie-Modul mit einem festgelegten Suchraum-Modul. Die selbst zu definierende Kostenfunktion kann im

[10] Zu Abbildung 3.17: Konkrete Ausprägungen eines Moduls sind kursiv dargestellt.

[11] Hier werden lediglich Module aufgerufen, so daß es meines Erachtens nach nicht notwendig ist, eine solche Komponente in das Modulsystem mit aufzunehmen.

Module	Rewr.	Regeln	S.-Raum	S.-Strat.	Kosten	Suche
Volcano	1	3	1	1	$\surd$	
V.&Prairie	1	$\surd$	1	1	$\surd$	
OPT++	($\surd$)	($\surd$)	$\surd$	$\surd$	$\surd$	mgl.
Starburst	2	$\surd$	–	–	?	

Tabelle 3.1: Vergleich des Modulsystems mit existierenden Ansätzen.

Kosten-Modul verankert werden und eine separate Steuerung oder Beeinflussung der Suche gibt es nicht.

Wird Volcano im Zusammenhang mit **Prairie** verwendet, so bietet sich auch die Möglichkeit, auf einfache Art und Weise mehrere Regelmengen zu benutzen, wobei Prairie nur die Spezifikation vereinfacht. Das neue System entspricht auch dieser Konstellation, obwohl die Regelspezifikation zwar von der Regelmaschine, also dem Rewriting-Modul abhängig ist, andererseits aber die Aneinanderreihung mehrerer Rewriting-Module mit jeweils anderer Regelmenge erlaubt ist.

Das System **OPT++** besitzt schon mehrere Gemeinsamkeiten mit dem Modulsystem. Auch hier gibt es einen Rewriting-Mechanismus (`TreeToTreeGenerator`, usw.). Leider wurde das Regelkonzept vernachlässigt, wobei es sich problemlos integrieren ließe. Die entsprechenden Module sind wieder Rewriting und Regeln. Die Rewriting-Komponenten bilden in OPT++ auch den Suchraum. Prinzipiell läßt sich der Suchraum erweitern, die Suchstrategien, sind aber immer von diesem abhängig. Da der Suchraum über eine einheitliche Schnittstelle angesprochen wird, kann man ihn dem Suchraum-Modul zuordnen, auch die Suchstrategien kann man dem Suchstrategie-Modul bzw. dessen Ableitungen direkt zuordnen. Schließlich kann das Kosten-Modul auch die dortige Kostenberechnung übernehmen. In OPT++ wurde die Möglichkeit offengelassen, den Suchprozeß gezielt zu steuern. Das ist mit dem Suche-Modul hoffentlich konkreter formuliert worden.

Als letztes steht Starburst zum Vergleich. Mit einer Einschätzung muß man jedoch vorsichtig sein, da es sich ja „nur“ um ein erweiterbares DBMS handelt. Meines Erachtens nach zeichnet Starburst besonders die Regelverwaltung aus, wenn es auch Beschränkungen bei der Form und der Verarbeitung der Anfragen gibt[12]. Die zwei verschiedenen Regelsysteme entsprechen hier nun auch zwei konkreten Rewriting-Modulen, die Möglichkeiten der Regelspezifikation, wenn auch eingeschränkt (QGM), mehreren Regeln-Modulen. Ansonsten ergeben sich keine Gemeinsamkeiten.

Vorzüge und Nachteile des Modularisierungsvorschlages

Im Abschnitt 3.1 wurde gefordert, daß für einen Optimiererbaukasten der Optimierungsprozeß in Module zu zerlegen ist, damit Teile eines Optimierers leichter ausgetauscht werden können. Mit dem vorgestellten Entwurf wurde eine solche Aufteilung erreicht, die es gestattet, prinzipiell jedes Modul auszutauschen. Ebenfalls existiert genügend funktionaler Freiraum, der zumindest die Beschränkungen der bisherigen Ansätze überwindet und die Konstruktion von Optimierern nicht nur für zentralisierte relationale DBMS, sondern auch für verteilte, mobile und objektorientierte Systeme gestattet (siehe Beispiel in Abschnitt 3.3.1). Die sechs Haupt-Module bilden die Spitze einer Hierarchie (Abbildung 3.18), die mit fortschreitender Anwendung noch um weitere Spezialisierungen erweitert werden kann. Im Grundsystem müssen dazu aber erst einmal einige konkrete Module realisiert werden. Von diesen ausgehend kann man sich dann für weitere Spezialisierungen, also den Aufbau einer

[12] Regeln mußten sich an die interne Anfrageform QGM halten und konnten deshalb nur vom Starburst-Regel-Interpreter ausgewertet werden.

tieferen Hierarchie entscheiden, wenn das Anwendungsfeld umfangreicher wird. Die Schnittstellen sind teilweise zwar nur vage und allgemein angegeben, für den Anfang schien es aber sinnvoll zu sein. In ihrer jetzigen Form sollten sie bei der Implementierung als Orientierung dienen und bei Bedarf erweitert werden. Die Erweiterbarkeit und Anpaßbarkeit dieses Systems an eine Optimiererumgebung sollte, insgesamt gesehen, gewährleistet sein, wobei eine verbindliche Schnittstellendefinition für die einzelnen Module natürlich Voraussetzung ist. Nur mit einer einheitlichen Schnittstelle ist ein Modul auch wirklich beliebig austauschbar.

Der Hauptvorteil des Modulsystems, nämlich die Möglichkeit, schnell und einfach einen Optimierer zusammenzusetzen, wird durch folgende Nachteile abgeschwächt: Am Anfang ist der Aufwand für die Entwicklung eines Grundstockes an konkreten Modulen nicht zu vernachlässigen. Dazu sind wahrscheinlich weitere Designentscheidungen nötig, die zum Beispiel die Trennung zwischen Regeln- und Rewriting-Modulen betreffen. Unterschiedliche Regelinterpreter verwenden verschiedene Regelformate — überhaupt gestaltet sich die Integration existierender Komponenten in einen neuen Rahmen eher schwierig.

Nachteilig an diesem Ansatz ist auch, daß die Module nicht beliebig miteinander kombinierbar sein werden. Am deutlichsten zeigt sich das beim Suchraum und den darauf anwendbaren Suchstrategien. Wird ein Suchraum vorberechnet, lassen sich fast alle Strategien anwenden, da ja nun beliebig auf die Pläne zugegriffen werden kann. Ist es jedoch nicht möglich, bei einem nicht vorberechneten Suchraum, alle Pläne aus einer ε-Umgebung zu bestimmen, weil mit der darunterliegenden Rewriting-Komponente der Umformungspfad hin zu diesen Plänen nicht realisieren werden kann, dann scheiden beispielsweise die genetischen Algorithmen als Suchverfahren aus. Bei einem Aufbau des Suchraumes *„on demand“* haben nur die Algorithmen Chancen, die immer nur den nächsten Plan benötigen bzw. immer einen Schritt fortschreiten (Iterative Improvement zum Beispiel). Es handelt sich hierbei also um Modulabhängigkeiten. Eine breite Modulhierarchie wird dieses Problem noch vergrößern.

Daß Module gar nicht eingesetzt werden könnten, hängt von einer wesentlichen Grundlage, der internen Repräsentationsform für die Anfrage, ab. Ein Kandidat ist das Kosten-Modul. Ausführungskosten für einen Operator sind von seiner Definition abhängig. Sind die Kosten für den Operator nicht definiert, kann das Kosten-Modul nicht verwendet werden. Will man aber nicht bei jeder neuen Operatordefinition auch das Kosten-Modul ändern, müßte man Vorgaben für die interne Repräsentation schaffen. In diesem Bereich sind Auflagen jedoch wieder hinderlich, da sie zu sehr einschränken. Ein Beispiel hierzu ist das *Query Graph Model* in Starburst. Der Entwickler muß sich zusätzlich in das QGM einarbeiten, obwohl er eine Operatormenge in seiner bisherigen Art und Weise bereits entwickelt hat und nun eine Abbildung nach QGM vornehmen muß. Schwerwiegender ist noch, daß das QGM sehr abstrakt ist und die Formulierung von Regeln nicht so einfach wie bei der Relationenalgebra ist (siehe Starburst-Kapitel 2.1).

Abhängig von der internen Repräsentation ist auch die Umformung. Regeln werden mit konkreten Operatoren angegeben und eine Rewritingkomponente kann nur eine bestimmte Klasse von Regeln bearbeiten. In diesem Bereich hatte Volcano bereits Abhilfe geschaffen, indem zur Umformung nur die Stelligkeit der Operatoren und deren Namen nötig waren.

Die Abhängigkeit schleicht sich beim Suchraum-Modul wieder ein, wenn Anfragen bzw. Anfragepläne um Informationen ergänzt werden müssen. Die Rechnerknotenzuordnung von Operatoren soll auch hier als Beispiel dienen. Kommt man sogar auf die Idee, den Kostenwert für einen Teilplan am jeweiligen Knoten im Plan zu speichern, begibt man sich noch tiefer in diese Abhängigkeit.

Ganz unabhängig von der Anfragerepräsentation sind nur die Suchstrategien, die auf einen Raum zugreifen können, ohne auf dessen Inhalt oder Erzeugung Rücksicht nehmen zu müssen[13], da sie nur an einer (Kosten-)Zahl interessiert sind.

[13] Prinzipiell zumindest, da ja bereits herausgestellt wurde, daß bestimmte Zugriffsarten bei einigen Konstellationen von Suchraum und Suchstrategie nicht möglich sind und dies auf die Abhängigkeit des Suchraumes zurückgeführt werden kann.

Nicht zuletzt rentiert sich eine Bibliothek von Optimiererbausteinen auch erst bei einer hohen Wiederverwendungsrate. Wieder sind die Suchstrategien die besten Kandidaten, da sie mit ihrem Aufgabenfeld sehr gut nach außen isoliert werden können und die Suchverfahren meist auch unabhängig von der DB-Optimierung entstanden sind. Besonders schlecht wiederverwendbar sind die Kosten-Module, die nicht nur abhängig von den Operatoren, sondern auch abhängig vom DBMS mit dessen zur Verfügung gestellten Kostenparametern sind. Desweiteren lassen sich Regeln nur bei einer ähnlichen Anfragesprache und dem gleichen Regelinterpreter wiederverwerten. Der Regelinterpreter an sich bzw. das Rewriting_mit_Regeln-Modul ist zusammen mit vielen konkreten Regeln-Modulen oft einsetzbar. Aber es ist insofern sicherlich keine Universallösung, als daß bei einem ganz anderen Optimierersystem ein neuer Regelinterpreter benutzt werden muß und bereits entwickelte Regeln konvertiert werden müssen, obwohl sie eventuell das gleiche ausdrücken. Diese Tatsache läßt sich nur unterdrücken, wenn man sich wieder einschränkt und sich auf einen Regelinterpreter festlegt.

3.5 Testimplementierung

Bei der Implementierung eines Modulsystems kann man sich auf die Sprachkonstrukte einer Programmiersprache stützen, die den Modulcharakter und die Modulhierarchie des neuen Systems fördern. Mit einer objektorientierten Sprache wie Java lassen sich beispielsweise für jedes Modul Klassen bilden, die in einer Vererbungshierarchie stehen. Spezialisierungen von Modulen sind dann einfach abgeleitete Klassen. Eine Ausnahme müßte man für die Regeln-Module machen, da Regeln ja vom Regelinterpreter abhängig sind, also eine eigene Spezifikationssprache besitzen, und lediglich in einem Container (beispielsweise einer Datei) abgelegt werden.

Ein Problem stellt in der Zukunft auch die Integration existierender Systeme, zum Beispiel eines Kostenmodells, dar. Dabei kann man von der Sprache auch unterstützt werden — Java erlaubt beispielsweise die Integration von C-Code.

Eine Testimplementierung für dieses System wurde vorerst nicht vorgenommen. Die Gründe, die gegen eine Implementierung sprechen, lassen sich von den Nachteilen dieses Modul-Systems ableiten und sollen im nun folgenden Fazit noch einmal betont werden.

3.6 Fazit

Am Anfang dieses Kapitels wurde ein System präsentiert, das die Optimierung von Anfragen in gut überschaubare Aufgabenbereiche aufteilte, die gleichzeitig auch die Grundlage für die geforderte Modularisierung bildeten. Damit wurde ein theoretischer Grundstock gelegt, der praktisch auch realisierbar wäre, wenn es nicht folgende Bedenken bezüglich seiner Realisierbarkeit gäbe.

Das entworfene System ist aus zwei Gründen praktisch sehr aufwendig. Einerseits wegen der doch schwachen Wiederverwendbarkeit konkreter Module, deren Anzahl infolge der Abhängigkeiten von der internen Repräsentationsform einer Anfrage schnell sehr groß werden wird. Für jede Klasse von Anfragen müßten eigene Module gebildet werden, wobei nur die Suchstrategien ausgeschlossen wären. Andererseits sind die Kombinationsmöglichkeiten der Module beschränkt, und das nicht nur wegen der Spezialisierungen.

Um eine Entscheidung für den Einsatz eines solchen Modul-Systems zu treffen, müßte man den Aufwand für eine Testimplementation abschätzen, indem man das Anfrage-System aus zwei vorhandenen DBMS, zum Beispiel aus $\mathrm{HE_AD}$ und CROQUE, untersucht. Anschließend könnte man dann die Anzahl der tatsächlich wiederverwendeten Module feststellen.

Kapitel 4

Zusammenfassung und Ausblick

In dieser Studienarbeit wurde versucht, einen geeigneten Rahmen für die modulbasierte Zusammenstellung von Anfrageoptimierern zu finden. Die bereits existierenden Ansätze wiesen stellenweise Hindernisse auf, die den Datenbank-Implementierer einschränkten. Dieses zu beheben und, bei einem hohen Grad an Wiederverwendbarkeit, die Konstruktion nach einem Baukastenprinzip zu vereinfachen, war das Ziel des neuen Systems.

Der geschaffene Rahmen erfordert noch viel Eigenarbeit. So sind erst einmal praktische Erfahrungen zu sammeln. Danach könnte man sich überlegen, noch weitere Konzepte einzufügen. Zum Beispiel könnte man nach einem universellen Produktionsregelsystem suchen, das als alleinige Rewriting-Komponente die Umformung der Anfrage übernimmt. Verschiedenste Formen der Regelspezifikation könnten vielleicht durch eine Art Mapping-Mechanismus diesem einen Regelsystem zugänglich gemacht werden. Die große Abhängigkeit des Kosten-Moduls könnte durch die Schaffung eines universellen Kosten-Moduls mit integrierter Lastmessung oder verallgemeinerten Operatoren verringert werden. Auch die Einführung einer einheitlichen Interndarstellung hätte den Vorteil, eine Basis für die von der Anfragerepräsentation abhängigen Module zu bilden, deren Anzahl dadurch drastisch vermindert würde.

Bei der Anfrageverarbeitung in verteilten Systeme wurde bisher hauptsächlich über die verteilte Bearbeitung der Anfrage nachgedacht. Die verteilte bzw. parallele Optimierung stand bisher kaum zur Diskussion. So ließen sich für bestimmte Suchalgorithmen[1], die Pläne aus der Nachbarschaft eines Planes benötigen, Algorithmen finden, die diese Menge parallel bestimmen könnten. Auch die etwaige Bestimmung des ganzen Suchraumes könnte unter Einbeziehung eines Rechnernetzes erfolgen. Im weiteren könnte man den Einsatz paralleler Graph-Rewriting-Systeme untersuchen (siehe [PE93]). Schließlich ließe sich auch die Kostenberechnung parallelisieren, indem beispielsweise der Anfragebaum so aufgesplittet würde, daß er parallel besucht werden könnte.

[1] insbesondere genetische Algorithmen

Literaturverzeichnis

[Ber99] Benedikt Berger. *Entwicklung einer DBS-Architektur für mobile Verarbeitungsumgebungen.* Diplomarbeit, Universität Rostock, Fachbereich Informatik, Juni 1999

[DB95] Dinesh Das, Don Batory. *Prairie: A Rule Specification Framework for Query Optimizers.* Proceedings of the 11th International Conference on Data Engineering, March 1995, Taipei, Taiwan

[DB96] Dinesh Das, Don Batory. *Synthesizing Rule Sets for Query Optimizers from Components.* Technical Report TR-96-05, Department of Computer Sciences, The University of Texas at Austin, April 1996

[Feg99] Leonidas Fegaras. *OptGen* (floß in das neue Projekt Lambda-DB ein). Nur über WWW: http://lambda.uta.edu/lambda-DB/manual/optgen.html (27.11.1999)

[FLM96] Guntram Flach, Uwe Langer, Holger Meyer. *Arbeitsbericht zum Projekt HE_AD.* Universität Rostock, Fachbereich Informatik, 1996, unveröffentlicht

[Fre87] Johann Christoph Freytag. *A Rule-Based View of Query Optimization.* In: Proceedings of the ACM SIGMOD International Conference on Managment on Data, 1987, S. 173-180

[GM93] Goetz Graefe, William J. McKenna. *The Volcano Optimizer Generator: Extensibility and Efficient Search.* Proceedings of the Ninth International Conference on Data Engineering, 1993

[Gra95] Goetz Graefe. *The Cascades Framework for Query Optimization.* In: Bulletin of the Technical Committee on Data Engineering, Vol. 18 No. 3, September 1995, S. 19-28

[Gru94] Torsten Grust. *Entwurf und Implementierung der internen Ebene von OSCAR: Optimierung und Auswertung der Objektalgebra ABRAXAS.* Diplomarbeit, Technische Universität Clausthal, Institut für Informatik, September 1994

[HFL+89] Laura M. Haas, J.C. Freytag, G.M. Lohman, H. Pirahesh. *Extensible Query Processing in Starburst.* In: Proceedings of the ACM SIGMOD International Conference on Management of Data, S. 377-388, 1989.

[Ing99] Lester Ingber's Code and Reprint Archive, WWW: http://www.ingber.com/ (26.10.1999).

[Jon98] Michael Jonas. *Entwicklung und Test von Verfahren zur Bewertung von Optimierungen.* Studienarbeit, Universität Rostock, Fachbereich Informatik, 1998

[KD95] N. Kabra, D. J. DeWitt. *OPT++ : An Object-Oriented Implementation for Extensible Database Query Optimization.* Computer Science Department, University of Wisconsin, Madison, 1995

[KPH98] Joachim Kröger, Stefan Paul, Andreas Heuer. *Über die Verwendung von Heuristiken zur regelbasierten Optimierung von Datenbank-Anfragen.* In: Rostocker Informatik-Berichte 21, Universität Rostock, Fachbereich Informatik, 1998, S. 43-55

[Lan98] Uwe Langer. *Parallelisierung und Optimierung von Anfrageplänen im heterogenen verteilten, relationalen Datenbanksystem HE_AD.* Dissertation, Universität Rostock, Fakultät für Ingenieurwissenschaften, Fachbereich Informatik, 1998

[LS87] P.C. Lockemann, J.W. Schmidt. *Datenbank-Handbuch.* Springer, Berlin, 1987, S. 135 ff.

[Lub96] Astrid Lubinski. *Mobilität und mobiler Datenbankzugriff.* In: Rostocker Informatik-Berichte 19, Universität Rostock, Fachbereich Informatik, 1996, S. 45-56

[Lün95] Jens Lüneberg. *Entwurf und Implementierung eines algebraischen Optimierers für das verteilte Datenbanksystem HE_AD.* Diplomarbeit, Universität Rostock, Fachbereich Informatik, 1995

[MBH+96] William J. McKenna, Louis Burger, Chi Hoang, Melissa Truong. *EROC: A Toolkit for Building NEATO Query Optimizers.* Proceedings of the 22nd VLDB Conference Mumbai(Bombay), India, 1996

[Mit95] Bernhard Mitschang. *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte.* Vieweg, Braunschweig, 1995

[PE93] Rinus Plasmeijer, Marko van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Wokingham (England), 1993

[Ros96] Steffen Rost. *Analyse alternativer Suchstrategien zur kostenbasierten Optimierung objektorientierter Anfragen.* Diplomarbeit, Universität Rostock, Fachbereich Informatik, Juni 1996

[SAC+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, T.G. Price. *Access Path Selection in a Relational Database Managment System.* Proceedings of the ACM SIGMOD International Conference on Management of Data, 1979

[SWK76] M. Stonebraker, E. Wong, P. Kreps. *The Design and Implementation of INGRES.* In: Proceedings of the ACM Transactions on Database Systems, Vol. 1, 1976, S. 189-222

[Tji94] Stephen W.K. Tjiang. *The TWIG Reference Manual.* 1994