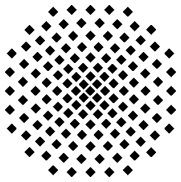


Universität Stuttgart

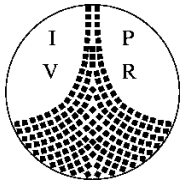
Fakultät Informatik

Prüfer:	Prof. Dr.-Ing. B. Mitschang
Betreuer:	Ralf Rantzau
Beginn am:	01.01.2000
Beendet am:	30.06.2000
CR-Nummer:	H.2.4

Studienarbeit Nr. 1772

Konzepte zur Abschätzung von Selektivität und Ausführungskosten für User-Defined Table Operators

Bernd Watzal



Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR)

Abteilung Anwendersoftware (AS)

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Anforderungen an heutige Datenbanksysteme

Mit den Möglichkeiten steigen die Wünsche und Erwartungen. Im Bereich der Datenbanksysteme (DBS) hat dies dazu geführt, dass wir nicht mehr damit zufrieden sind, Namen und Adressen effizient speichern und verwalten zu können. Im Bereich Multimedia wollen wir Bilder, Videos und Tonaufnahmen speichern und verarbeiten können. Verarbeiten bedeutet, dass wir an eine Multimedia-DB z.B. die Anfrage stellen wollen: „Suche mir aus allen Bildern diejenigen heraus, auf denen ein rotes Auto abgebildet ist.“ Von einem geografischen Informationssystem wollen wir evtl. wissen, wo wir sind, wo sich das nächstgelegenste Hotel mittlerer Preisklasse befindet und welches der kürzeste Weg dorthin ist. Damit stellen wir sehr hohe funktionale Anforderungen an die solchen Systemen zugrundeliegenden DBS. Um derartige Anfragen bearbeiten zu können, müssen die Systeme über komplexe Operatoren verfügen, die sie auf komplexe Objekte wie digitale Bilder oder Geodaten anwenden. Zu der funktionalen Anforderung kommt die zeitliche hinzu, dass wir die Antworten auf unsere Anfragen möglichst schnell haben wollen. Wir erwarten also vom System, dass es die Bearbeitung komplexer Aufgaben in sehr kurzer Zeit leistet.

Anwendungen und Erwartungen nicht nur aus den eben genannten Bereichen führten zur Entwicklung und Verwendung objektrelationaler Datenbanksysteme, kurz ORDBS. Sie ermöglichen es dem Anwender, benutzerdefinierte Objekte und Operatoren auf diesen Objekten im System zu integrieren. Der Anwender kann dadurch das System in die Lage versetzen, die von ihm gewünschten, komplexen Objekte zu speichern und so zu verarbeiten, dass komplexe, objektbezogene Anfragen an das System gestellt werden können.

Mit der Komplexität der Anfragen steigen Ausführungszeit und Ressourcenverbrauch. Da die Anfragen schnell beantwortet werden sollen, kommt der Anfrageverarbeitung in solchen Systemen eine zentrale Rolle zu. Ihre Aufgabe ist es, zu einer Anfrage einen optimierten und parallelisierten Ausführungsplan, kurz Plan, zu erzeugen. Er enthält genaue Instruktionen für das Ausführungssystem, wie die Anfrage auszuführen ist, und soll bei diesen DBS zu einer möglichst kurzen Antwortzeit führen. (Anwendungen können von der Anfrageverarbeitung stattdessen auch verlangen, einen möglichst hohen Durchsatz zu erzielen. Ein Beispiel dafür sind Transaktionssystemen von Banken, denen ein DBS zugrunde liegt.)

Das DBS belegt wie jedes andere System Ressourcen, die es zur Abarbeitung der Anfrage einsetzt. Typischerweise sind dies die CPUs, der Hauptspeicher, Externspeicher wie Festplatten und das Netz. Unter dem *Ressourcenverbrauch einer Anfrage* verstehen wir die Summe der Belegungszeiten einzelner CPUs und Festplatten, den Hauptspeicherverbrauch und die durch die Kommunikation anfallende Netzlast. Das Anfrageverarbeitungssystem hat durch die Optimierung Einfluss auf den Ressourcenverbrauch einer Anfrage und entscheidet bei der Anfrageparallelisierung darüber, wie der Ressourcenverbrauch auf die zur Verfügung stehende Hardware (vernetzte Rechner bzw. Muliprozessormaschinen) verteilt wird. Von der Optimierung wird erwartet, dass sie unter allen alternativen Plänen einer Anfrage denjenigen findet, der in der Summe am wenigsten Ressourcen beansprucht und somit die effizienteste Ausführung ermöglicht. Von der Parallelisierung wird erwartet, dass sie den optimierten Plan in Teilpläne aufspaltet, die parallel zueinander ausgeführt werden können. Einerseits steigen mit der Anzahl solcher Teilbäume die Kommunikationskosten und somit der Ressourcenverbrauch der gesamten Anfrage; andererseits nimmt die Antwortzeit ab, da mehr Ressourcen in die Ausführung der Anfrage miteingebunden werden. Der Parallelisierer muss also eine in dem Sinne optimale Parallelisierung suchen, dass dem zusätzlich verursachen Kommunikationsaufwand eine verhältnismäßig hohe Reduzierung der Antwortzeit gegenübersteht.

Ein solches System, das den beiden Ansprüchen gerecht wird, komplexe objektrelationale Anfragen durch Parallelität schnell ausführen zu können, heißt paralleles objektrelationales Datenbanksystem, kurz PORDBS.

1.2 Einsatz objektrelationaler Konzepte

Wie im vorigen Abschnitt erwähnt, wollen wir benutzerdefinierte Datenobjekte in Datenbanken speichern können. Das DBS soll uns als Anwender also die Möglichkeit bieten, den vorhandenen Datentypen selbst definierte hinzuzufügen. Bezüglich dieser Objekte wollen wir Anfragen stellen können. Dazu müssen wir dem DBS Werkzeuge zur Verfügung stellen, um unsere Objekte analysieren und manipulieren zu können. Das DBS muss uns die Möglichkeit bieten, benutzerdefinierte Funktionen und Operatoren im System registrieren zu können.

Eine benutzerdefinierte Funktion, im folgenden UDF (User-Defined Function) verarbeitet als Eingabe Attribute eines Tupels oder einer Tabelle und liefert einen Wert oder eine Tabelle als Ergebnis. Man klassifiziert sie je nach Ein- und Ausgabe in

Skalarfunktionen (UDSF), Aggregatfunktionen (UDAF) und Tabellenfunktionen (UDTF), die jeweils ein Tupel auf einen Wert, eine Tabelle auf einen Wert und ein Tupel auf eine Tabelle abbilden. UDFs werden in gängigen prozeduralen Programmiersprachen geschrieben und können daher beliebig komplexe Berechnungen durchführen. Sie stellen ein wichtiges, objektrelationales Konzept dar, ermöglichen es dem Anwender aber nicht, mehrstellige Datenbankoperatoren zu implementieren, da ihre Eingabeparameter auf eine Tabelle beschränkt sind (genaueres über UDFs siehe [5] Kapitel 2).

Ein benutzerdefinierter Tabellenoperator, kurz UDTO (User-Defined Table Operator) kann als Eingabe Attribute aus mehreren Tabellen und skalare Werte verarbeiten. Eine Tabelle kann dabei temporär sein oder eine beliebige Sicht auf die Datenbank darstellen. Als Ausgabe liefert der UDTO eine Tabelle oder einen skalaren Wert. Er wird ebenfalls in einer prozeduralen Programmiersprache geschrieben, kann aber auch deklarative Anweisungen wie eingebettetes SQL enthalten, um die Eingabetabellen zu verarbeiten. Damit kann ein UDTO einerseits voll auf die Funktionalität von SQL zurückgreifen, was sich besonders bei der Verarbeitung von ganzen Tabellen anbietet. Andererseits kann er von SQL erzeugte Zwischenergebnisse mit Mitteln der prozeduralen Programmierung weiterverarbeiten. Mit UDTOs stellt das ORDBS dem Anwender ein mächtiges und effizientes Konzept zur Verfügung, das es ihm gestattet, komplexe Operatoren für die Objekte im System zu implementieren.

1.3 Aufgabenstellung

MIDAS ist der Prototyp eines PORDBS, das zu Forschungszwecken an den Universitäten Stuttgart und München (TU) entwickelt wird. Die objektrelationalen Konzepte UDF und UDTO sind bereits teilweise implementiert worden. Ziel dieser Studienarbeit ist die weitere Integration von UDTOs in MIDAS. SQL-Anfragen, die UDTOs enthalten, können vom System geparst und ausgeführt werden. Optimierer und Parallelisierer müssen dabei allerdings deaktiviert sein, da sie UDTOs nicht verarbeiten können.

Die Aufgabenstellung ist daher, den Operator UDTO in die Anfrageverarbeitung von MIDAS zu integrieren, damit ihn das System effizient ausführen kann. Optimierer und Parallelisierer sollen also in die Lage versetzt werden, zu einer Anfrage, die UDTOs enthält, alternative Ausführungspläne erstellen und jeweils deren Gesamtkosten berechnen zu können, um den optimalen Plan auszuwählen. Diese Anforderung impliziert ein allgemeines Modell zur Berechnung der Ausführungskosten eines UDTO. Da ein UDTO sehr komplexe Funktionalitäten realisieren kann, ist zu untersuchen, welche Art von Information Optimierer und Parallelisierer über den UDTO benötigen, um dessen Ausführungskosten und Kardinalität abschätzen zu können.

Außerdem soll der Optimierer eine benutzerdefinierte Funktionen unter bestimmten Voraussetzungen alternativ durch einen passenden UDTO realisieren können. Die UDF muss als Prädikat in einem Join, einer Restriktion, Projektion oder Aggregation verwendet werden und es muss mindestens ein UDTO im System registriert sein, der die Gesamtfunktionalität des jeweiligen Operators einschließlich der als Prädikat verwendeten UDF realisiert. Unter den Alternativen (Operator mit UDF oder einer der in

Frage kommenden UDTOs) soll er durch Kostenrechnung die optimale Realisierung ermitteln.

Die erarbeiteten Konzepte sollen dokumentiert, implementiert und getestet werden.

1.4 Verlauf der Arbeit

Während der Einarbeitung in das Konzept UDTO und in Aufbau und Funktionsweise von Optimierer und Parallelisierer sowie beim Erstellen der Systemspezifikation wurden viele grundsätzliche und konzeptionelle Fragen aufgeworfen. Bedingt durch die Mächtigkeit des UDTO war konzeptionelle Arbeit in größerem Umfang als erwartet zu leisten, speziell bei der Modellierung der Kostenberechnung. Zur Klärung dieser Fragen war es letzten Endes erforderlich, vier Diplomarbeiten komplett und drei weitere Arbeiten teilweise zu lesen. Einarbeitung und Konzeption fielen damit so umfangreich aus, dass die Implementierung der erarbeiteten Konzepte nur ansatzweise für den Optimierer in Angriff genommen werden konnte.

Die konzeptionelle Arbeit wurde vollständig abgeschlossen und dokumentiert. Sie enthält Systemanalyse und Systementwurf von Optimierer und Parallelisierer, beschreibt also deren Funktionsweise und alle Implementierungsschritte, die vorgenommen werden müssen, um den UDTO in beide Komponenten zu integrieren. Einen weiteren Schwerpunkt stellt die Kostenabschätzung des UDTO in Optimierer und Parallelisierer dar. Die damit verbundene Problematik wird in einem eigenen Kapitel behandelt.

Für UDTOs, die mit der Tabellenoption „.+“ definierte Ausgabeattribute enthalten, wurde eine Erweiterung des Parsers und der MIDAS-Datenstruktur *Udto* implementiert und dokumentiert. Sie ist notwendige Voraussetzungen dafür, dass die Konvertierungsroutinen überhaupt um solche UDTOs erweitert werden können.

Im Rahmen der Systemanalyse stellte sich heraus, dass für die Erweiterung des Optimierers zur Realisierung von UDFs durch UDTOs zwei wesentliche Voraussetzungen fehlten: Erstens kennt der Optimierer keine UDFs. Der Operator UDF müsste zuerst in Optimierer (und Parallelisierer) integriert werden. Zweitens ist die Registrierung eines UDTO als funktionale Alternative zu einem Operator mit einer UDF als Prädikat in MIDAS noch nicht implementiert worden. Der Systemkatalog müsste dazu erweitert werden, und der DDL-Compiler müsste das entsprechende Statement zur Registrierung einer UDF mit alternativem UDTO parsen können.

Ein Entwurf zur Realisierung von UDFs durch UDTOs wurde ausgearbeitet, um den konzeptionellen Teil der Studienarbeit vollständig abzuschließen. Die Implementierung des Entwurfs im Rahmen dieser Studienarbeit wurde aus genannten Gründen fallengelassen.

Rückblickend möchte ich anmerken, dass ich die Thematik der Studienarbeit sehr vielschichtig, komplex und deswegen interessant fand. Sie gab Einblick in die internen Abläufe eines Datenbanksystems, in die mit Anfrageoptimierung und -parallelisierung verbundene Problematik und in objektrelationale Konzepte und die mit deren Realisierung verbundene Komplexität. Der zur Bewältigung der Aufgabenstellung erforderliche Arbeitsaufwand ging jedoch nicht zuletzt wegen dieser Komplexität weit

über den einer Studienarbeit hinaus, was zum fast kompletten Ausfall der Implementierung führte. Ein ungünstiger Umstand war, dass Michael Jaedicke im Dezember 1999 die Universität verließ und somit während der gesamten Bearbeitungszeit kein Ansprechpartner mehr da war, der über Detailwissen in MIDAS verfügt hätte. Dies brachte weiteren Zeitaufwand mit sich, der wiederum bei der Implementierung fehlte.

Die bei der Integration des UDTO in Anfrageverarbeitung und Kostenabschätzung zutage getretenen Aspekte und deren Komplexität, die in diesem Ausmaß nicht vorhersehbar waren, betrachte ich neben dem erarbeiteten Konzept als ein weiteres, interessantes Ergebnis dieser Arbeit.

1.5 Gliederung

Kapitel 2 gibt einen Überblick über das Datenbanksystem MIDAS, seine Entwicklung, Architektur und prinzipielle Funktionsweise, soweit es für das Verständnis dieser Arbeit erforderlich ist.

In Kapitel 3 werden das theoretische Konzept UDTO und typische Einsatzmöglichkeiten des Operators beschrieben. Die bisherige Implementierung des Konzepts in MIDAS wird vorgestellt.

Kapitel 4 enthält einen Entwurf, der alle notwendigen Schritte aufführt, um den UDTO in den Optimierer zu integrieren und die dazugehörigen Konvertierungsroutinen anzupassen. Kenntnisse über Aufbau und Funktionsweise des Model-M-Optimierers sowie über Cascades Optimizer Framework (COF) werden vorausgesetzt. Es beschreibt im letzten Teil eine am Parser vorgenommene Erweiterung bezüglich Attributpropagierung, die eine notwendig Vorarbeit zur Erweiterung der Konvertierungsroutinen darstellt.

Kapitel 5 führt den Entwurf zur Integration des UDTO für den Parallelisierer fort. Es setzt analog dem Vorgängerkapitel Kenntnisse über Aufbau und Funktionsweise des in MIDAS implementierten Parallelisierers sowie über COF voraus und beschreibt die notwendigen Arbeitsschritte zur Erweiterung des Parallelisierers.

Die Integration des UDTO in die Kostenmodelle von Optimierer und Parallelisierer wird im Kapitel 6 behandelt. Die zentrale Frage, welche Informationen dem System über einen UDTO zur Abschätzung seiner Kosten zur Verfügung stehen müssen, wird erörtert.

Kapitel 7 stellt als Zusammenfassung die Kernpunkte und Ergebnisse dieser Studienarbeit heraus und gibt einen Ausblick bezüglich der Kostenberechnung des UDTO.

Der Anhang listet die Arbeitsschritte auf, die noch zu erbringen sind, um das Konzept UDTO in vollem Umfang in MIDAS zu implementieren. Es handelt sich dabei um bislang nicht oder nur teilweise implementierte Arbeitsschritte. Er enthält außerdem eine Korrektur des Systemkatalogs von MIDAS.

2 Das Datenbanksystem MIDAS

2.1 Entwicklungsgeschichte

MIDAS ist ein paralleles, objektrelationales Datenbanksystem, das zu Forschungszwecken als Prototyp an den Universitäten Stuttgart und München (TU) unter Leitung der Professoren Mitschang und Bayer entwickelt wird. Aufbauend auf dem sequentiellen DBS TransBase wurde MIDAS durch Arbeiten wissenschaftlicher Hilfskräfte und verschiedene Studien-, Diplom- und Doktorarbeiten weiterentwickelt. Objektrelationale Konzepte wurden bereits teilweise integriert, um Lösungsansätze für die im vorigen Kapitel genannten Anforderungen zu suchen.

2.1.1 Bisherige Arbeiten

Dieser Abschnitt soll einen Überblick über die vorangegangenen Arbeiten geben, die mit der Thematik dieser Studienarbeit in unmittelbarem Zusammenhang stehen. Anspruch auf Vollständigkeit besteht nicht.

Franz Brandmayer erweiterte 1997 mit seiner Diplomarbeit [7] das sequentielle Anfrageausführungssystem von MIDAS zu einem parallelen. Der Operatorbaum einer Anfrage wird in Teilbäume zerlegt und von mehreren Rechnern parallel bearbeitet. Die Kommunikation und Synchronisation der Rechner wurde durch das Konzept der Kommunikationssegmente realisiert, das in der Arbeit detailliert beschrieben ist.

Michael Fleischhauer entwickelte im selben Jahr den Parallelisierer von MIDAS, der die Zerlegung von Operatorbäumen in parallel ausführbare Teilbäume übernimmt. Seine Diplomarbeit [1] enthält Aufbau und Funktionsweise des Parallelisierers, der mit dem

Cascades Optimizer Framework erstellt wurde, sowie das Modell zur Kostenberechnung.

MIDAS verfügte über einen sequentiellen, heuristischen Optimierer, der nicht mehr den Anforderungen des um Parallelität erweiterten Systems entsprach. Der neue Optimierer, auch Model-M genannt, wurde 1998 in zwei Diplomarbeiten von Kay Krüger-Barvels [2] und Matthias Hilbig [3] entwickelt. Er ist ebenfalls mit Cascades implementiert, arbeitet kosten- und regelbasiert und optimiert Anfragen sowohl für sequentielle als auch für parallele Ausführung. Die Arbeiten beschreiben Cascades sowie Aufbau und Funktionsweise des Optimierers. Um die vom Optimierer benötigten Statistiken im Systemkatalog ablegen zu können, wurde er im Rahmen der Praktikumsarbeit [6] von Sabine Perathoner erweitert.

Im selben Jahr integrierte Sebastian Heupel den UDTO in das Ausführungssystem [4]. Seine Diplomarbeit erläutert das Konzept von benutzerdefinierten Tabellenoperatoren und dessen Implementierung im Ausführungssystem von MIDAS.

Michael Jaedicke, der einige der vorangegangenen Arbeiten betreut hatte, stellt in seiner 1999 erschienenen Doktorarbeit [5] die Konzepte UDF und UDTO im Zusammenhang dar und erläutert konzeptionell die grundlegenden Aspekte der Implementierung des UDTO in MIDAS.

2.1.2 Die aktuelle Version

Die Version, die dieser Arbeit zugrunde liegt, ist in der Abteilung Anwendersoftware der Uni Stuttgart unter dem Pfad ´/home/midas1´ zu finden. Sie enthält alle im vorigen Abschnitt aufgeführten Erweiterungen. Die von Klaus Julisch 1999 im Rahmen seiner Diplomarbeit an Optimierer und Cascades Optimizer Framework vorgenommenen Änderungen enthält diese Version nicht, da sie zur Folge hatten, dass der Parallelisierer nicht mehr korrekt funktionierte.

Die aktuelle Version von MIDAS kann also zu einer SQL-Anfrage einen optimalen Ausführungsplan erstellen, diesen parallelisieren und entsprechend ausführen. Unter gewissen Einschränkungen können UDTOs und UDFs ausgeführt werden. Details zur bisherigen Implementierung von UDTOs sind im Kapitel 3.2 beschrieben.

2.2 Die Systemarchitektur

MIDAS ist ein Mehrbenutzer-Datenbanksystem, das auf einer flexiblen, mehrschichtigen Client-Server-Architektur aufbaut. Das Datenbanksystem fungiert als Server, an den ein oder mehrere Clients Anfragen stellen können. Intern baut der Datenbankserver wiederum auf verschiedenen Prozessen auf, die zur Bearbeitung der Anfragen miteinander in Client-Server-Beziehungen interagieren. Die interne Client-Server-Architektur ermöglicht parallele Anfrageausführung.

Um hohe Skalierbarkeit und effiziente, parallele Anfrageverarbeitung zu gewährleisten, können beliebig viele Rechner in den Systemverbund aufgenommen werden. Mit Hilfe

der Bibliothek *Parallel Virtual Machine* (PVM) werden die Rechner zu einem virtuellen Parallelrechner mit verteiltem Speicher zusammenfasst. Die auf ihnen laufenden Prozesse kommunizieren dann über die sog. PVM-Schnittstelle miteinander.

Die Datenbank selbst liegt auf mehrere Festplatten verteilt vor. Ihre *Shared-Disk-Architektur* ermöglicht allen Rechnern über das *Network File System* Zugriff auf die komplette Datenbank. Mehrere Prozesse können parallel auf Daten zugreifen, sofern diese auf verschiedenen Platten abgelegt sind.

2.2.1 Das Zugangssystem

MIDAS besteht aus einem Zugangs- und einem Ausführungssystem. Das Zugangssystem ist intern wiederum in zwei Komponenten, den Administrations- und den Applikationsserver, gegliedert.

Der **Administrationsserver** stellt für Datenbankanwendungen den Kontakt zum Datenbanksystem her, indem er ihnen einen Applikationsserver zuteilt. Es gehört außerdem zu seinen Aufgaben, Datenbanken zu erzeugen, zu löschen, zu starten und herunterzufahren.

Der **Applikationsserver** empfängt Datenbankanfragen von der ihm zugeteilten Anwendung. Seine Hauptaufgabe ist die Anfrageverarbeitung, die Compilierung, Optimierung, Parallelisierung und Scheduling der Anfrage umfasst. Diese Teilkomponenten des Applikationsservers sind für diese Studienarbeit von zentralem Interesse. Der Vollständigkeit halber sei erwähnt, dass zu seinen weiteren Aufgaben die Transaktionsverwaltung gehört.

2.2.2 Das Ausführungssystem

Das Ausführungssystem ist in der Lage, Anfragen von mehreren Anwendungen gleichzeitig und einzelne Anfragen parallelisiert auszuführen. Es besteht aus den Komponenten Interpreter, Segmentserver und Segmentslave, deren Prozesse auf den beteiligten Rechnern aktiv sind und miteinander kommunizieren.

Da das Ausführungssystem von dieser Arbeit nicht berührt wird, verzichten wir an dieser Stelle auf eine genauere Darstellung und verweisen interessierte Leser auf die Diplomarbeiten [1] und [4].

2.3 Das Anfrageverarbeitungssystem

Eine SQL-Anfrage wird in MIDAS, wie bereits erwähnt, von mehreren Komponenten des Applikationsservers vorverarbeitet, bis sie zur Ausführung gelangt. Diese Komponenten bilden aufeinander aufbauend das sogenannte Anfrageverarbeitungssystem. Die Anfrage selbst wird dabei in verschiedene Darstellungsformen abgebildet, bevor sie schließlich dem Ausführungssystem übergeben wird. Der SQL-Compiler bildet

sie z.B. von der Darstellung als Textstring in einen strukturierten und normierten Operatorbaum ab. In Abbildung 2.1 wird das Anfrageverarbeitungssystem anhand des Weges, den eine Anfrage in MIDAS durchläuft, graphisch dargestellt. Seine Komponenten, ihre prinzipielle Aufgabe und Funktionsweise sowie die jeweilige Darstellungsform der Anfrage werden beschrieben.

Der **Compiler** überprüft syntaktische und semantische Korrektheit der textuellen SQL-Anfrage und übersetzt sie in einen strukturierten MIDAS-Operatorbaum. Bäume in MIDAS-Format sind physische Bäume, d.h. sie enthalten alle zur Ausführung notwendigen Informationen und könnten so, wie sie sind, direkt dem Ausführungssystem übergeben werden.

Der **Voroptimierer:** Von den acht Phasen des alten, sequentiellen Optimierers wurden fünf zur Voroptimierung beibehalten. Er arbeitet heuristisch und bewirkt eine Normierung des Baumes, soweit sie für die anschließende, kostenbasierte Optimierung sinnvoll ist. Es werden einfache strukturelle Umordnungen und Ersetzungen von Operatoren vorgenommen wie z.B. das Umwandeln von Unterabfragen mit *In*- und *Exists*-Knoten in *Joins*. (siehe [2] Kapitel 4.3.1).

Der eigentliche **Optimierer,** auch Model-M Optimierer genannt, arbeitet regel- und kostenbasiert, d.h. er erzeugt zu seinem Eingabebaum unter Benutzung eines Regelsatzes alternative Bäume, die er gemäß eines Kostenmodells bewertet. Ihm stehen zwei Kostenmodelle zur Verfügung, je nachdem ob die Anfrage anschließend sequentiell oder parallel ausgeführt wird. Anhand des aktiven Modells wählt er unter allen Alternativen den jeweils günstigsten Baum aus und gibt ihn weiter.

Der Optimierer wurde mit dem Werkzeug Cascades Optimizer Framework, kurz COF oder Cascades, erstellt. Er benötigt einen logischen Operatorbaum in COF-Format als Eingabe und produziert als Ausgabe einen physischen Baum wiederum in COF-Format. Aus diesem Grund sind ihm Konvertierungsroutinen vor- und nachgeschaltet.

Mit **Konvert_O in COF** benennen wir die Konvertierungsroutine des Optimierers, die aus dem physischen Eingabebaum in MIDAS-Format einen logischen Baum in COF-Format erzeugt. Ein logischer Baum enthält nur logische Knoten und ist daher nicht ausführbar. Ein logischer Knoten beschreibt die Funktionalität seines Operators, ohne sich auf dessen Realisierung festzulegen. Beispielsweise beschreibt ein logischer Join-Knoten, dass zwei Tabellen verknüpft werden. Er enthält aber nicht die Information, ob dies durch ein Merge-Join, Hash-Join, Nested-Loops-Join oder anderweitig realisiert wird.

Die Routine **Konvert_O in MIDAS** erzeugt aus dem physischen COF-Baum einen (physischen) MIDAS-Baum, der dann entweder sequentiell ausgeführt oder dem Parallelisierer übergeben wird.

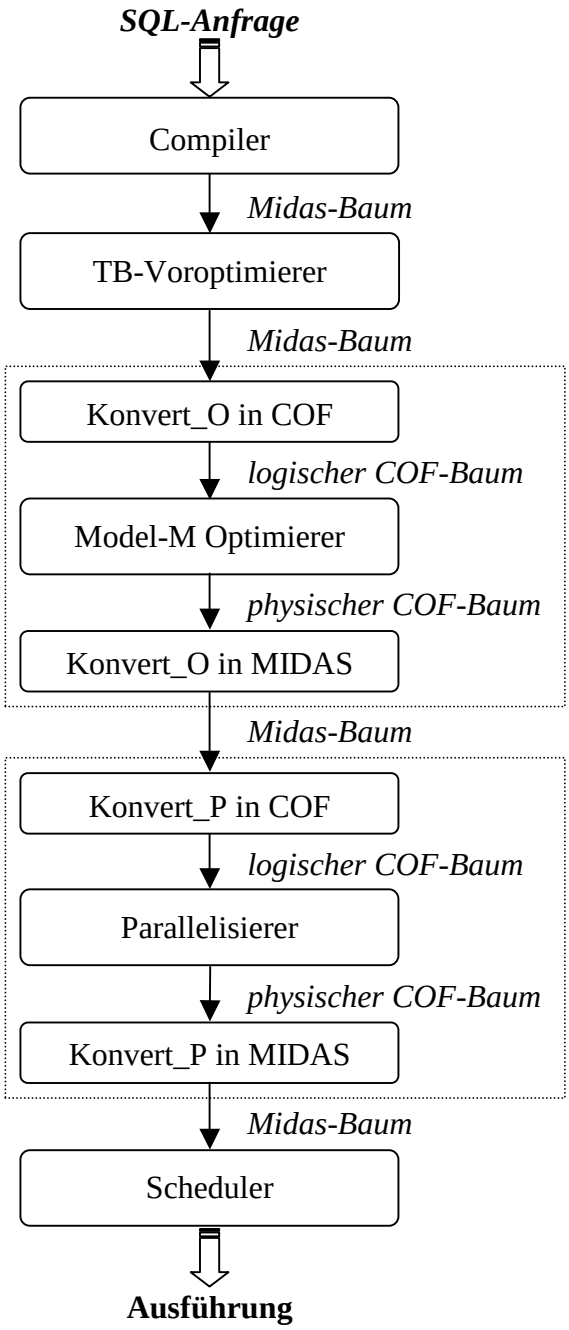


Abbildung 2.1: Komponenten der Anfrageverarbeitung von MIDAS

Der **Parallelisierer** hat die Aufgabe, in dem ihm übergebenen Operatorbaum nach parallel zueinander ausführbaren Einheiten zu suchen und unter allen möglichen Einteilungen eine optimale auszuwählen. Eine solche Einheit nennen wir Teilbaum oder Block. Der Datenaustausch zwischen den Teilbäumen erfolgt über Send- und Receive-Knoten, die der Parallelisierer an den entsprechenden Stellen im Operatorbaum eingefügt. Der Parallelisierer arbeitet regel- und kostenbasiert und ist in mehrere aufeinanderfolgende Phasen unterteilt. Zuerst wird der Baum durch Einfügen von Send-Receive-Paaren, kurz SR-Paaren, mit Inter- und Intra-Operator-Parallelität versehen. Danach bildet der Parallelisierer durch Verschieben und Zusammenfassen dieser SR-Paare möglichst große, parallel ausführbare Blöcke. In jeder Phase werden anhand ihrer Regeln alternative Bäume erzeugt und mit Hilfe des Kostenmodells bewertet. Am Ende einer Phase wird ein für diese Phase optimaler Baum ausgewählt und an die nächste

Phase weitergereicht. Als Endergebnis steht ein zusammenhängender Baum, der an denjenigen Stellen SR-Paare enthält, an denen er zur parallelen Ausführung in Teilbäume zerlegt werden soll.

Der Parallelisierer wurde vor dem Optimierer mit demselben Werkzeug Cascades erstellt. Als Schnittstelle zu MIDAS sind ihm ebenfalls eigene Konvertierungsroutinen vor- und nachgeschaltet. Die Routine **Konvert_P in COF** wandelt einen MIDAS-Baum in einen physischen COF-Baum um, der allerdings als logischer Baum deklariert wird, damit Transformationsregeln auf ihn angewandt werden können. Die Rückkonvertierung **Konvert_P in MIDAS** überträgt analog der Routine des Optimierers einen physischen COF-Baum wieder in einen MIDAS-Baum.

Der **Scheduler** ist die letzte Komponente des Anfrageverarbeitungssystems. Er bereitet die Ausführung des ihm übergebenen MIDAS-Baums vor, indem er ihn gemäß den Vorgaben des Parallelisierers in Teilbäume zerlegt und diese an die Interpreter der zur Verfügung stehenden Rechner verschickt. Diese führen die ihnen zugewiesenen Teilbäume aus, versenden ihre Zwischenergebnisse mittels Kommunikationssegmente untereinander und arbeiten auf diese Weise die gesamte Anfrage ab.

3 Das Konzept UDTO

In der Einleitung wurde das Konzept der benutzerdefinierten Tabellenoperatoren bereits kurz vorgestellt. Im ersten Teil dieses Kapitels wollen wir das Konzept theoretisch, also unabhängig von einer konkreten Implementierung vorstellen. Wir beschreiben, was ein UDTO leisten soll, welche Eigenschaften er besitzt geben und typische Beispielszenarien für seinen Einsatz an. Auf diesem Grundwissen aufbauend beschreiben wir im zweiten Teil das Konzept des UDTO, so wie es bisher in MIDAS implementiert ist, und erörtern aufgetretene Probleme, Erweiterungs- und Änderungsbedarf bezüglich der bisherigen Implementierung.

3.1 Beschreibung von UDTOs

Ein UDTO besteht aus einer Funktion, die entweder prozeduralen Code und darin eingebettetes SQL oder eine einzige SQL-Anweisung enthält. Im ersten Fall sprechen wir von prozeduralen UDTOs, im zweiten von SQL-Makros. UDTOs können als Eingabe sowohl skalare Werte als auch Attribute aus mehreren Tabellen verarbeiten, wobei diese Tabellen erstens Tabellen der Datenbank, zweitens Sichten auf die Datenbank und drittens temporäre Zwischenergebnisse derjenigen Anfrage sein können, in deren Kontext der UDTO verwendet wird. Sie sind flexible und mächtige Operatoren, da sie die Funktionalitäten von SQL und prozeduraler Programmierung vereinigen. Mittels SQL eingelesene bzw. erzeugte Werte können vom prozeduralen Code weiterverarbeitet werden. Der UDTO kann als Ergebnis entweder einen skalaren Wert oder eine komplette Tabelle produzieren. Sein Ergebnis übergibt er der Anfrage, in deren Kontext er aufgerufen wurde.

Für das DBS stellt ein prozeduraler UDTO eine „black box“ dar. Weder seine Funktionalität noch seine Eigenschaften sind ihm bekannt. Zudem können in ein und demselben System mehrere verschiedene UDTOs registriert sein. Das DBS benötigt daher Angaben über jeden registrierten UDTO, die ihn hinreichend beschreiben. Die Menge dieser Angaben nennen wir *charakterisierende Information*. Sie muss alle

Angaben beinhalten, die das System benötigt, um einen bestimmten UDTO so einsetzen zu können, wie es andere Operatoren auch einsetzt. Jede Komponente der Anfrageverarbeitung und des Ausführungssystems soll mit dem UDTO operieren können. Dabei ist es erstrebenswert, eine kleinste, für alle Systemkomponenten ausreichenden Menge charakterisierender Information zu definieren. Das DBS ist dann intelligent, wenn es vom Entwickler eines UDTO möglichst wenige Angaben über diesen benötigt. Umgekehrt verlangt dies vom System, möglichst umfangreiche allgemeingültige Informationen zu enthalten, um aus der charakterisierenden Information alle notwendigen Angaben ableiten zu können. Mit allgemeingültigen Informationen kann z.B. eine Wissensbasis gemeint sein, die es ermöglicht, zu einem Knoten unter Verwendung der ihn charakterisierenden Information die knotenspezifischen Kostenformeln zu synthetisieren. Eine solche Wissensbasis nimmt dem Benutzer dann die Aufgabe ab, die knotenspezifischen Kostenformeln eines UDTO selbst zusammenzustellen und dem System mitzuteilen. Die Frage nach der kleinsten Menge charakterisierender Information ist daher zentraler Gegenstand dieser Arbeit und wird bezüglich Optimierer, Parallelisierer und besonders bei der Kostenberechnung erörtert.

3.1.1 Verwendung von UDTOs in SQL-Anfragen

UDTOs werden benötigt, um die Funktionalität von SQL auf benutzerdefinierte Objekte zu erweitern und um Anfragen stellen zu können, die in herkömmlichem SQL alleine nicht formulierbar sind. Sie können dazu direkt in SQL-Anfragen eingebunden werden und erhöhen so die Mächtigkeit von SQL.

Nehmen wir an, wir hätten eine Datenbank mit mehrdimensionalen geometrischen Objekten angelegt und einen UDTO *nachbar* entworfen, der uns eine Liste aller Objektpaare liefert, die in einer bestimmten Nachbarschaftsbeziehung zueinander stehen. Als Eingabe benötigt der UDTO zwei Definitionstabellen der Objekte. Der Tabelle *anker* entnimmt er die Koordinaten des Ankerpunktes. Von ihm ausgehend bestimmt er die Ausrichtung des Objekts, wofür er der zweiten Tabelle *lage* die fiktiven Werte r (Richtungsvektor) und g (Größe) entnimmt. Das folgende SQL-Statement zeigt, wie der Operator innerhalb einer Anfrage auf die Tabellen der Objekte angewandt werden kann:

```
SELECT *
FROM nachbar(anker (x1, x2, x3), lage (r, g))
```

Jeder Eingabetabelle folgt eine Liste, die dem UDTO diejenigen Attribute der Tabelle angibt, die er als Eingabe verwenden soll. Die Attribute müssen dabei der formalen Spezifikation seiner Eingabeparameter entsprechen. Der UDTO wurde in der FROM-Klausel verwendet. Im Allgemeinen kann ein UDTO, der eine Tabelle zurückliefert, in einem SQL-Statement überall dort eingesetzt werden, wo Tabellenausdrücke erlaubt sind.

Der UDTO verbirgt, von den Eingabeparametern abgesehen, die gesamte Komplexität, die mit der Untersuchung der Objekte auf Nachbarschaft verbunden ist. Aufgrund dieser Eigenschaft ist es auch sinnvoll, sogenannte SQL-Makros zu erstellen. Ein SQL-Makro ist ein UDTO, der nur aus einer einzigen SQL-Anweisung besteht. Solche UDTOs erweitern die Funktionalität von SQL nicht, doch können mit Hilfe dieser Technik komplexe SQL-Anfragen vereinfacht und lesbarer gemacht werden, indem wiederholt benötigte Anweisungsteile oder Unteranfragen in einem UDTO gekapselt werden.

3.1.2 Realisierung von UDFs durch UDTOs

Abgesehen von der direkten Verwendung in SQL-Statements können UDTOs vom System intern dazu eingesetzt werden, alternative Ausführungspläne zu erzeugen. Dies ist dann interessant, wenn der alternative Plan eine effizientere Ausführung der Anfrage verspricht. Potential zur Erzeugung effizienterer Pläne ist vorhanden, wenn eine UDF als Prädikat in einem der Operatoren Join, Restriktion, Projektion oder Aggregat verwendet wird. In einer solchen Konstellation liefert die UDF ihrem übergeordneten Operator für jedes Eingabetupel einen Wert, der von diesem weiterverarbeitet wird. Das DBS muss also für jedes Eingabetupel einen Funktionsaufruf der UDF durchführen. Steht dem System ein UDTO mit entsprechender Funktionalität zur Verfügung, so kann es alternativ die UDF samt übergeordnetem Operator durch den UDTO realisieren. Im Fall der folgenden Beispielanfrage sehen wir, dass so vom System ein effizienterer Ausführungsplan erzeugt werden kann.

```
SELECT kunden.name
FROM   anrufe, kunden
WHERE  equal(anrufe.tel_nr, kunden.tel_nr)
```

Mit diese Anfrage möchte eine Firma über die Telefonnummern herausfinden, welche Kunden an einem bestimmten Tag angerufen haben. Da die Telefonnummern in den Tabellen *anrufe* und *kunden* in verschiedenen Datenformaten vorliegen, vergleicht die eigens dafür entworfene UDSF *equal* die Nummern auf Übereinstimmung. Die Anfrage wird gemäß dem linken Operatorbaum aus Abbildung 3.1 ausgeführt: Da eine UDF ihre Eingabeattribute nur aus einer Tabelle entnehmen kann, muss das DBS zuerst das kartesische Produkt aus den Tabellen *anrufe* und *kunden* erzeugen. Die nachfolgende Restriktion ruft dann für jedes Tupel dieser Tabelle die UDF *equal* auf und filtert so die gesuchten Übereinstimmungen heraus. Der in der Anfrage enthaltene Equi-Join der Tabellen *anrufe* und *kunden* wird also durch ein kartesisches Produkt (Times) mit nachfolgender Restriktion realisiert.

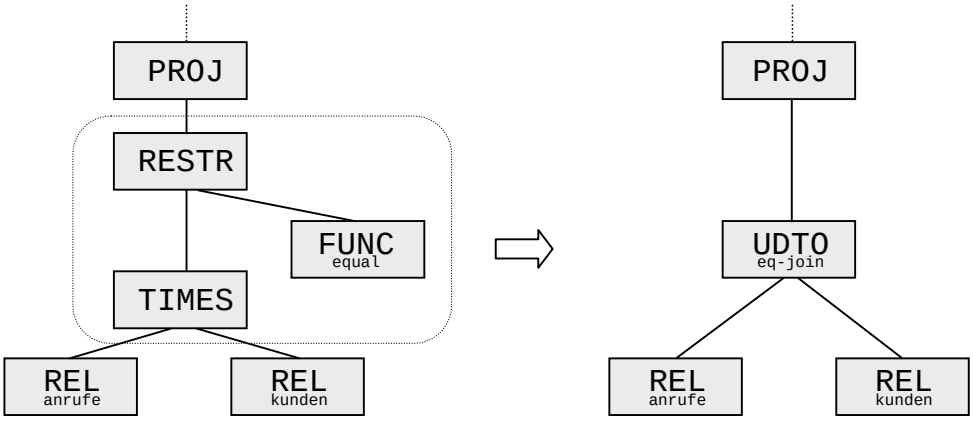


Abbildung 3.1: Operatorbaum mit UDF Realisierung durch einen UDTO

Steht dem System nun ein entsprechender UDTO zur Verfügung, der die Verknüpfung der Tabellen und den Wertevergleich vornimmt, kann die Abfrage eventuell wesentlich effizienter ausgeführt werden. Der Equi-Join aus der SQL-Anfrage wird dann alleine durch den UDTO *eq-join* realisiert (rechter Operatorbaum). Er ersetzt also im Vergleich die Funktionalität der drei Knoten `TIMES`, `RESTR` und `FUNC` des linken Baumes.

Welchen Aufwand bringen diese Realisierungen für *n* Kunden und *m* Anrufe mit sich? Im linken Operatorbaum wird eine Zwischenrelation der Größe $n \times m$ erzeugt; die Restriktion führt also $(n \times m)$-Funktionsaufrufe der UDF durch. Steigt mit der Anzahl der Kunden die Anzahl der Anrufe proportional, was der Normalfall sein dürfte, liegt der Aufwand dieser Realisierung in $O(n^2)$.

Im Fall des rechten Operatorbaumes hängt der Aufwand von der internen Implementierung des UDTO *eq-join* ab. Er könnte z.B. die Tabellen zuerst nach den Telefonnummern sortieren und dann den Vergleich durchführen. Von konstanten Abweichungen abgesehen beträgt der Aufwand für Sortieren auf Basis von Vergleichen $n \cdot log(n) + m \cdot log(m)$; für Sortieren mit einer Hash-Funktion ist der Aufwand lediglich $(n+m)$ und somit linear. Der Wertevergleich muss bei sortierten Tabellen nur noch $(n+m)$-mal durchgeführt werden, womit der Gesamtaufwand in $O(n \cdot log(n))$ bzw. in $O(n)$ liegt. Das System muss nur einen Funktionsaufruf für den UDTO durchführen.

Ersetzungen wie die eben beschriebene können auch bei Verwendung einer UDF innerhalb einer „reinen“ Restriktion, d.h. ohne vorgeschaltetes kartesisches Produkt, eines Aggregats oder einer Projektion zu einem effizienteren Ausführungsplan führen. Der UDTO besitzt grundsätzlich den Vorteil, dass ihm die gesamte Eingabetabelle zur Verfügung steht und er daher nur einmal aufgerufen wird, wohingegen eine UDF vom entsprechenden systemeigenen Operator (Join, Restriktion, Aggregation, Projektion) für jedes Tupel der Eingabetabelle aufgerufen werden muss.

Damit das System solche Ersetzungen vornehmen kann, muss für die Verwendung der UDF innerhalb des entsprechenden Operators mindestens ein UDTO als dazu alternative Realisierung eingetragen sein. Der Benutzer könnte dies dem System z.B. bei der Registrierung der UDF mitteilen. Ein entsprechendes DDL-Statement für unser Beispiel könnte so aussehen:

```
CREATE FUNCTION equal (INTEGER, STRING)
RETURNS BOOLEAN
ALLOW eq-join AS JOIN OPERATOR
EXTERNAL NAME...
```

Der UDTO *eq-join* muss im System bereits registriert sein. Nun weiß das System, dass es einen Join mit der UDF *equal* als Prädikat alternativ durch den UDTO *eq-join* ersetzen kann. Welche Realisierung letztlich die bessere ist, entscheidet der Anfrageoptimierer des DBS durch Kostenabschätzung. Im Kapitel 6.2 beschäftigen wir uns intensiv mit der Frage, welche charakterisierende Informationen der Optimierer benötigt, um die Kosten eines beliebigen UDTO abschätzen zu können.

3.2 Das bisher implementierte Konzept

Teile der Konzepte von UDTOs und UDFs sind in MIDAS bereits implementiert worden. Mit jeder Implementierung gehen Festlegungen von Details einher, die ein theoretisches Konzept in eine konkrete, implementierbare Ausprägung abbilden. Was an einer Stelle opportun und sinnvoll erscheint, kann an anderer Stelle unerwünschte Probleme bereiten. Wir stellen im zweiten Teil dieses Kapitels die bisherige

Implementierung von UDTOs in MIDAS und die damit verbundene Ausprägung des Konzepts UDTO vor und erörtern Probleme und Fragen, die für eine vollständige Implementierung des Konzepts zu lösen sind.

3.2.1 Registrierung von UDTOs

MIDAS erstellt zu jeder SQL-Anfrage einen Operatorbaum, den es bottom-up ausführt. Jeder Operator im Baum empfängt Daten von seinem Sohn bzw. seinen Söhnen, verarbeitet sie und reicht die Ergebnisdaten an seinen Vaterknoten weiter. Die Blätter sind im Allgemeinen Scans, die Tupel aus den in der FROM-Klausel aufgeführten Tabellen auslesen. Der Wurzelknoten liefert schließlich das Ergebnis der Anfrage. Um diesen Datenfluss innerhalb des Baumes realisieren zu können, benötigt das System Informationen über die Ein- und Ausgangsdatenströme eines jeden Knotens. Es muss die Identität der Daten (welche Attribute aus welchen Tabellen) und deren Format (Datentyp) kennen. Bei systemeigenen Operatoren steht ihm diese Information zur Verfügung; bei einem UDTO muss sie dem System mitgeteilt werden. Der Datenfluss eines UDTO wird in MIDAS bisher spezifiziert durch:

- Anzahl seiner Eingabetabellen
- für jede Eingabetabelle Anzahl und Datentypen derjenigen Attribute, die vom UDTO verarbeitet werden
- Anzahl und Datentypen der Ausgabeattribute

Mit diesen Angaben ist der Datenfluss für das Ausführungssystem hinreichend beschrieben, nicht jedoch für Optimierer und Parallelisierer, wie wir später sehen werden.

Die folgende Notation gibt die Syntax des zugehörigen DDL-Statements zur Registrierung eines UDTO in MIDAS an. Die Notation lehnt sich an die BNF an, wurde hier aber zur besseren Lesbarkeit abgewandelt. Eine Klammerung gefolgt von einem Stern beschreibt eine Liste. Der Inhalt der Klammerung kann beliebig oft durch Komma getrennt eingesetzt werden und die gesamte Liste ist eingeklammert. Zum Beispiel kann (<Eingabe>)* abgebildet werden auf „(<Eingabe>, <Eingabe>, <Eingabe>).“

```
Statement = CREATE TABLE_OPERATOR <Operatorname> (<Eingabe>)*
            RETURNS <Ausgabe> <Parallelisierungsparameter>
            AS <Prozedur>
Eingabe   = [<Variablenname>] <Datentyp> | TABLE <Eingabetabellenname> (<Attribut>)*
Ausgabe   = [<Variablenname>] <Datentyp> |
            TABLE <Ausgabetabellenname> (<Attribut> | <Eingabetabellenname>.+ )*
Attribut  = <Attributname> <Datentyp>
```

Der **Operatorname** gibt dem UDTO einen eindeutigen Namen innerhalb der Datenbank in der er registriert wird. Unter diesem Namen kann er in SQL-Statements verwendet werden.

Eingabe beschreibt eine Liste, die sich aus Platzhaltern für die Eingabetabellen und Eingabevariablen zusammensetzt. Jeder Eingabetabelle folgt eine Attributliste, die Platzhalter für ihre Attribute und deren Datentypen enthält. Die Angabe von Variablennamen ist optional, lediglich der Datentyp muss angegeben werden.

Die **Ausgabe** enthält einen Platzhalter entweder für eine Variable oder eine Tabelle. Wie bei der Eingabe sind die Datentypen anzugeben. Die Angabe einer Eingabetabelle gefolgt von der „.+“-Option als Ausgabeattribut bewirkt, dass ihre übrigen, in der

Eingabe nicht enthaltenen Attribute in die Ausgabetabelle integriert werden. Genaueres dazu wird im Abschnitt Attributpropagierung erläutert.

Die **Parallelisierungsparameter** sind optional und dienen zur Beschreibung, ob und wie Intra-Operator-Parallelität auf den UDTO angewandt werden darf. Sie werden im Kontext des Parallelisierers in Kapitel 1 angegeben.

Die **Prozedur** an sich wird entweder durch ein SQL-Statement oder durch das Schlüsselwort `EXTERNAL` gefolgt von einen Verweis auf die Bibliothek, die den prozeduralen Code des UDTO enthält, angegeben. Im ersten Fall sprechen wir von UDTOs als SQL-Makros, im zweiten von prozeduralen UDTOs.

Beispiele zur Registrierung:

Das folgende DDL-Statement definiert den prozeduralen UDTO *overlaps*, der zwei räumliche Objekte daraufhin untersucht, ob sie einander überlappen. Jedes Objekt ist als Punktemenge in einer eigenen Tabelle abgelegt. Als Ergebnis liefert der Operator einen Boolean-Wert. Der ausführbare Programmcode des Operators liegt im angegebenen Verzeichnis in der DLL *libudto.so* unter der Referenz *overlaps_3D* vor.

```
CREATE TABLE_OPERATOR overlaps
  (TABLE object_A (x FLOAT, y FLOAT, z FLOAT),
   TABLE object_B (x FLOAT, y FLOAT, z FLOAT) )
RETURNS BOOLEAN
AS EXTERN '/path/.../libudto.so'#'overlaps_3D'
```

Um die Definition eines SQL-Makros am Beispiel zu zeigen, verwenden wir den UDTO *eq-join* aus Abschnitt 3.1.2. Um die SQL-Anweisung des Makros kurz zu halten, soll der UDTO in diesem Beispiel allerdings nicht zwei Attribute verschiedenen Datentyps, sondern einfach zwei Attribute gleichen Datentyps (Integer) miteinander vergleichen. Die AS-Klausel enthält das eigentliche SQL-Makro, abgeschlossen mit einem Semikolon. Die Ausgabe ist eine Tabelle, die die Attribute der Eingabetabellen in folgender Reihenfolge enthält: alle Attribute von *input1* außer dem Join-Attribut *num1*, das gemeinsame Join-Attribut (jetzt *num* genannt), alle Attribute von *input2* außer dem Join-Attribut *num2*.

```
CREATE TABLE_OPERATOR eq-join
  (TABLE input1 (num1 INTEGER),
   TABLE input2 (num2 INTEGER) )
RETURNS TABLE output (input1.+, num INTEGER, input2.+)
AS (INSERT INTO output
    SELECT input1.+, num1, input2.+
    FROM input1, input2
    WHERE input1.num1 = input2.num2 );
```

Registrierte UDTOs können mit dem DROP-Statement wieder aus der Datenbank entfernt werden.

```
DROP UDTO eq-join
```

3.2.2 Attributpropagierung

Ein UDTO verfügt zur Erzeugung seiner Ausgabe über die Eingabewerte bzw. –attribute und evtl. daraus errechnete Werte. Um allerdings bestimmte Operationen mit UDTOs realisieren zu können, ist es darüber hinaus erforderlich, diejenigen Attribute der

Eingabetabellen, die vom UDTO nicht verwendet werden, in die Ausgabetabelle integrieren zu können. Nehmen wir an, wir wollten eine benutzerdefinierte Restriktion mit Hilfe eines UDTO erstellen. Sie filtert anhand eines Attributs *x* die gewünschten Tupel der Eingabetabelle heraus. Dieses Attribut steht uns zur Verfügung und wir können es in die Ausgabetabelle integrieren. Wie aber teilen wir dem System mit, dass es alle übrigen Attribute der Eingabetabelle propagieren, d.h. in die Ausgabetabelle übertragen soll?

In MIDAS wurde dazu die „.+“-Option implementiert. Um alle übrigen Attribute einer Eingabetabelle *input1* zu propagieren, muss lediglich an der gewünschten Stelle innerhalb der Liste der Ausgabeattribute „input1.+“ eingetragen werden. MIDAS fügt dann dort alle übrigen Attribute von *input1* in die Ausgabetabelle ein, wobei es ihre Reihenfolge beibehält. Jetzt sind wir in der Lage, die Ausgabetabelle für unsere Restriktion spezifizieren zu können:

```
RETURNS TABLE output (input1.+, x <Typ von x>)
```

3.2.3 Prozedurale UDTOs

Prozedurale UDTOs sind das Herzstück des UDTO-Konzepts, da durch sie die Menge der Datenbankoperatoren und somit die Funktionalität von SQL erweiterbar wird. Sie ermöglichen eine in diesem Sinne universelle objektrelationale Anfrageverarbeitung. Die Grundlagen, um prozedurale UDTOs in MIDAS ausführen zu können, sind durch Erweiterungen an Compiler und Ausführungssystem (siehe [4] und [5]) größtenteils geschaffen worden. Die noch bestehenden Einschränkungen sind allerdings zu erheblich, als dass von Ausführung prozeduraler UDTOs gesprochen werden könnte.

Der **Compiler** erkennt und verarbeitet UDTOs, wenn sie in der FROM-Klausel einer Anfrage stehen ([4] Kapitel 5.1). Er erzeugt den Knoten Udto und versieht ihn und den Operatorbaum der Anfrage, in die der UDTO eingebettet ist, mit den notwendigen Send- und Receive-Knoten. Jedes Send-Receive-Paar stellt eine Kommunikationsschnittstelle zwischen dem Udto und dem Operatorbaum der übergeordneten Anfrage dar. In Abbildung 3.2 ist der Ausschnitt eines Operatorbaumes dargestellt, den der Compiler im Falle eines UDTO mit zwei Eingabetabellen produziert. Er fügt drei Send-Receive-Paare ein: Die Teilbäume, die aus den Eingabetabellen lesen, werden mit einem Send-Knoten abgeschlossen. Als Blätter des Udto-Teilbaumes werden Receive-Knoten eingefügt, die von diesen Send-Knoten Daten empfangen. Ebenso wird der Teilbaum des Udto mit einem Send-Knoten abgeschlossen, aus dem wiederum der Receive-Knoten des übergeordneten Teilbaumes der Anfrage liest ([4] Kapitel 5.1.2).

Der **Scheduler** spaltet den Operatorbaum zwischen jedem SR-Paar (Pfeile) in einzelne Teilbäume auf, die er zur Ausführung an verschiedene Interpreter verschickt. Zusätzlich startet oder reaktiviert der Scheduler für jeden dieser Interpreter einen Funcstart-Prozess. Damit werden die Interpreter in die Lage versetzt, mit Hilfe des ihnen zugeordneten Funcstarts den UDTO innerhalb derselben Transaktion auszuführen. Jeder Funcstart ruft dazu einen eigenen Applikationsserver auf, der die eingebetteten SQL-Anfragen (bzw. die entsprechenden Gentrees) ausführt.

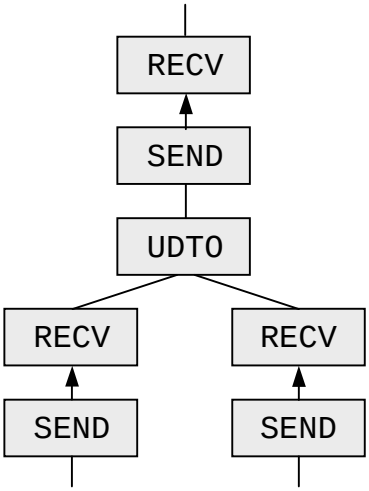


Abbildung 3.2: UDTO mit umgebenden Send- und Receive-Knoten

Da die einzelnen Teilbäume voneinander getrennt ausgeführt werden, müssen sie miteinander kommunizieren können. Hierfür wird auf das Konzept der Kommunikationssegmente zurückgegriffen, das zur parallelen Anfrageausführung implementiert wurde (siehe [7] und [4]). Die Datenströme werden also mit Hilfe von Kommunikationssegmenten, vom Send-Knoten zum entsprechenden Receive-Knoten übertragen, was in der Abbildung durch die Pfeile dargestellt ist. Auch parallele Ausführung von UDTOs im Sinne der Intra-Operator-Parallelität ist daher möglich: Ein parallelisierter Teilbaum, der einen UDTO enthält, wird dann vom Scheduler wie andere Teilbäume auch gemäß seines Parallelitätsgrades an mehrere Interpreter zur Ausführung verschickt, für die entsprechend viele Funcstart-Prozesse aktiviert werden. (Um echte Parallelität bei der Ausführung zu gewährleisten, läuft in MIDAS pro Prozessor höchstens ein Interpreterprozess. Genaueres zur Parallelisierung wird im Kapitel 1 erläutert.)

MIDAS kann lediglich UDTOs ausführen, die als Gentree vorliegen. Formal gesehen ist solch ein UDTO prozedural, da er einen Prozedurrumpf besitzt, auf den der „AS EXTERN“-Teil seines DDL-Statements zur Registrierung verweist. Dieser Prozedurrumpf macht allerdings nichts anderes, als den entsprechenden Gentree einzulesen und mittels der TBX-Schnittstelle (TransBase eXecutable) auszuführen. Ein Gentree (generated tree) ist eine Darstellungsform für MIDAS-Operatorbäume, d.h. er entspricht einer compilierten SQL-Anfrage. Letztlich werden also SQL-Makros ausgeführt, die formal als prozedurale UDTOs deklariert sind. Diese pseudo-prozeduralen UDTOs besitzen daher nicht die Mächtigkeit, die Funktionalität von SQL zu erweitern.

Prozeduraler Code mit eingebettetem SQL kann anstelle des Gentree nicht ausgeführt werden, da bislang die Schnittstelle vom Code zu den Send- und Receive-Knoten des UDTO fehlt. Die Prozedur ist also nicht in der Lage, mit der übergeordneten SQL-Anfrage zu kommunizieren. Sobald diese Schnittstelle implementiert ist, wird es möglich sein, prozedurale UDTOs auszuführen. An ihrer Erstellung wird zur Zeit im Rahmen einer hilfswissenschaftlichen Tätigkeit am Lehrstuhl von Prof. Mitschang gearbeitet.

3.2.4 SQL-Makros

SQL-Makros können in MIDAS ebenfalls nur in der FROM-Klausel der übergeordneten Anfrage verwendet werden. Ihre Ausführung unterscheidet sich grundsätzlich von der prozeduraler UDTOs, da dem DBS durch das SQL-Makro die Definition des UDTO unmittelbar zur Verfügung steht. MIDAS verarbeitet solche UDTOs, indem es das zugehörige Makro in einen Operatorbaum umwandelt und ihn in den Operatorbaum der übergeordneten Anfrage integriert. Dieser Vorgang wird Makroexpansion genannt und ist in [5] Kapitel 7.3 genau beschrieben. Der so entstandene Operatorbaum enthält keinen Knoten UDTO und kann daher wie gewohnt optimiert und parallelisiert werden. Das System benötigt keine charakterisierende Information zu SQL-Makros.

4 Erweiterung des Optimierers

Die Lektüre dieses Kapitels setzt voraus, dass der Leser Aufbau und Funktionsweise des Model-M-Optimierers kennt und weiß, was kosten- und regelbasierte Anfrageoptimierung ist; Strukturen und Arbeitsweise von Cascades müssen geläufig sein. Zur Einarbeitung in die Thematik empfehlen sich die Diplomarbeiten [2] und [3].

Das Kapitel stellt einen Entwurf vor, der alle Schritte enthält, die notwendig sind, um den UDTO in den Optimierer zu integrieren. In diesem Sinne dient der Entwurf auch als Anleitung für die noch ausstehende Implementierung. Die Integration des UDTO in die Kostenmodelle des Optimierers wird zusammen mit der Integration in das Kostenmodell des Parallelisierers in Kapitel 6 behandelt.

4.1 Grundsätzliche Überlegungen

Der Optimierer wurde mit COF erstellt und besteht aus fünf Komponenten, die alle erweitert werden müssen. Diese sind die beiden Konvertierungsroutinen, die die Schnittstelle zu MIDAS darstellen, der Regelsatz, der dem Optimiervorgang zugrunde liegt und die beiden Kostenmodelle für sequentielle und quasi-parallele Anfrageoptimierung.

Dabei sind folgende, grundsätzliche Fragen zu klären: Wie soll der UDTO einschließlich seiner SR-Knoten im Optimierer (im Cascades-Format) dargestellt werden? Sind Optimierungsregeln auf ihn anzuwenden und wenn ja welche? Benötigt der Optimierer, von der Kostenberechnung einmal abgesehen, zusätzliche charakterisierende Informationen?

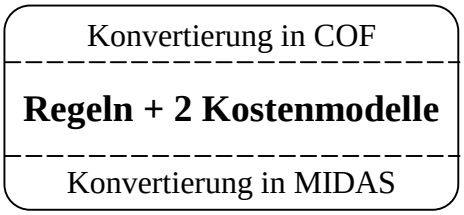


Abbildung 4.1: Komponenten des Optimierers

4.1.1 Darstellung des UDTO

Die Konvertierungsroutinen überführen einen Operatorbaum von MIDAS- in Cascades-Format und umgekehrt. Im MIDAS-Format ist ein UDTO stets von Send- und Receive-Knoten umgeben. Da MIDAS-Bäume ausführbare Bäume sind, enthalten sie diese SR-Knoten, die dem Ausführungssystem mitteilen, wie es die Kommunikation zwischen UDTO und umgebenden Operatorbaum zu bewerkstelligen hat. Für die Anfrageoptimierung ist diese Information allerdings nicht von Interesse. Sie untersucht den Baum auf strukturelle Verbesserungen (Transformationen) und bestmöglichste Implementierungen von logischen Knoten (z.B. Join als Hash-Join oder Merge-Join implementieren). An den UDTO umgebenden SR-Knoten würde und dürfte der Optimierer keine Änderungen vornehmen, weshalb es sinnvoll ist, einen UDTO samt den ihn umgebenden SR-Paaren in Cascades als einen Knoten darzustellen. Alle sieben in Abbildung 3.2 dargestellten Knoten können also auf einen logischen Cascades-Knoten Udto abgebildet werden, ohne dass für die Optimierung relevante Information verloren ginge. Im Vergleich zur Darstellung der Send- und Receive-Knoten als eigene Knoten in Cascades spricht für unser Vorgehen, dass die Cascades-Bäume möglichst klein gehalten werden und damit der Aufwand für Konvertierung und Optimierung geringer ausfällt. Bei der Kostenberechnung des UDTO darf allerdings nicht vergessen werden stets, die durch die SR-Knoten verursachten Kosten miteinzubeziehen. Diese Kosten sind besonders dann relevant, wenn der Optimierer einen korrekten Vergleich zwischen der unmittelbaren Realisierung einer UDF und einer alternativen Realisierung durch einen UDTO durchführen möchte, da eine UDF im Gegensatz zu einem UDTO keine zusätzlichen Kommunikationskosten durch SR-Knoten verursacht.

4.1.2 Optimierungsregeln

Die Optimierungsregeln lassen sich allgemein in Transformationsregeln und Implementierungsregeln einteilen. Erstere verändern die Struktur eines Baumes und erzeugen so einen zu ihm logisch äquivalenten, alternativen Baum. Sie werden in Cascades nur auf logische Operatoren angewandt. Zweitere legen die Implementierung eines logischen Operators fest, indem sie ihn durch einen physischen Operator ersetzen (z.B. einen Scan durch einen Relationenscan), und überführen so einen logischen in einen physischen Baum (Ausführungsplan). Der Model-M-Optimierer wendet fünf Transformationsregeln an, die zum Ziel haben, Häufungen von Joins zu erzeugen und für diese die optimale Anordnung zu finden. (Mit einer Häufung von Joins ist das unmittelbare Aufeinanderfolgen von mindestens zwei Join-Knoten im Baum gemeint; siehe dazu [2] Kapitel 4.2.4). Dies ist der Kernvorgang der Optimierung. Eine sechste Regel fasst aufeinanderfolgende Projektionen zusammen.

Ein UDTO wird im Allgemeinen dazu verwendet, eine vorher im DBS nicht vorhandene, oft komplexe Operation auf benutzerdefinierten Objekten durchzuführen. Der Knoten, auf den er im Operatorbaum abgebildet wird, ist daher immer eine spezielle Ausprägung des allgemeinen Knotens UDTO. Man muss sich an dieser Stelle klar machen, dass ein UDTO-Knoten im Gegensatz zu allen anderen Knoten des DBS nicht einen Operator mit konstanten logischen und physischen Eigenschaften repräsentiert, sondern eine unendlich große Menge an Operatoren. Da Transformationsregeln darauf basieren, logische Äquivalenz zwischen dem Baum vor und dem Baum nach ihrer Anwendung zu garantieren, ist es nicht sinnvoll und auch nicht möglich, allgemeingültige Transformationsregeln für den logischen Knoten UDTO formulieren zu wollen.

Damit der Optimierer die in Kapitel 3.1.2 geforderte Realisierung von UDF samt übergeordnetem Knoten durch einen oder mehrere dafür in Frage kommende UDTOs durchführen kann, muss er mit den entsprechenden Transformationsregeln versehen werden. Für jeden der Operatoren Join, Restriktion, Projektion und Aggregation ist eine Regel zu formulieren, die dann ausgeführt wird, wenn sie auf die Struktur *Operator+UDF* trifft und im Systemkatalog mindestens einen zu dieser Struktur alternativen UDTO findet (siehe Abbildung 3.1). Da Cascades Regeln unterstützt, die zu einem Muster verschiedene Substitute erzeugen können (siehe [2] Kapitel 6.5, Methode `next_substitute`), ist es möglich, zu einer UDF bei mehreren zur Verfügung stehenden UDTOs mit ein und derselben Regel alle alternativen Operatorbäume zu erzeugen. Die Regel muss im Falle eines prozeduralen UDTO lediglich seinen Namen und seine logischen Eigenschaften dem Systemkatalog entnehmen und in den UDTO-Knoten des Substituts eintragen. Zu den logischen Eigenschaften gehören in erster Linie das Schema (von Ein- und Ausgaberelation) sowie die Kardinalität der Ausgaberelation. Im Falle eines SQL-Makros muss die Regel eine Makroexpansion in der Gruppenstruktur von Cascades vornehmen. Dieser Vorgang wird schwierig zu implementieren sein und eventuell Änderungen an der Suchmaschine von Cascades erforderlich machen ([5] Kapitel 7.6), weshalb er in einer eigenständigen Arbeit zu behandeln wäre.

Den Regeln ist eine Implementierungsregel hinzuzufügen, die den logischen Knoten UDTO auf den physischen Knoten UDTO abbildet. Sie benötigt ebenfalls Informationen aus den Systemtabellen, um den Knoten mit den jeweiligen physischen Eigenschaften des UDTO zu versehen. Zu diesen Eigenschaften gehören die Sortierordnung seiner Ausgaberelation und ob bzw. wie der Operator zur parallelen Ausführung repliziert bzw. partitioniert werden kann.

4.1.3 Charakterisierende Informationen

Die zuletzt genannten physischen Eigenschaften des UDTO können von Operator zu Operator variieren und sind daher in die Menge der charakterisierenden Information aufzunehmen. Der Benutzer soll dem DBS auf irgendeine Weise mitteilen können, welche der Ausgabeattribute sortiert sind und wie sie sortiert sind (auf- oder absteigend). Liefert der UDTO keine sortierte Ausgabe, ist es wichtig zu wissen, ob er bestehende Sortierordnungen seiner Eingaberelationen beibehält, umkehrt oder zerstört. Diese Informationen werden gebraucht, um die benötigten physischen Eigenschaften und die erzeugten physischen Eigenschaften in Cascades berechnen zu können. Beispielsweise prüft Cascades bei einem Merge-Join, ob seine beiden Unterbäume die vom Merge-Join

benötigte Sortierung der Eingaberelationen liefern. Ist dies nicht der Fall, wird der gesamte Ausführungsplan verworfen.

Bei der Optimierung mit dem quasi-parallelen Kostenmodell werden Pläne bevorzugt, die gut parallelisiert werden können im Sinne der Intra-Operator-Parallelität. Der Optimierer muss daher für die Kostenberechnung wissen, ob die Eingaberelationen eines UDTO partitioniert bzw. repliziert werden können, um den Operator zu parallelisieren. Zur Angabe dieser charakterisierenden Information ist in MIDAS bereits ein DDL-Statement implementiert, das im nächsten Kapitel erläutert wird. An dieser Stelle wollen wir lediglich der Vollständigkeit halber festhalten, dass bereits der Optimierer charakterisierende Informationen bezüglich der Parallelisierbarkeit eines UDTO benötigt.

4.2 Konkrete Implementierungsschritte

Zunächst ist anzumerken, dass der Optimierer bisher noch keine User-Defined Functions verarbeiten kann. Bevor also an die Realisierung von UDFs durch UDTOs und deren Implementierung im Optimierer zu denken ist, sind zuerst einmal die Konvertierungsroutinen und der Optimierer um den Knoten UDF zu erweitern.

Zur Integration des UDTO in den Optimierer sind nacheinander folgende Schritte durchzuführen:

1. Erstellen des **logischen Knotens** UDTO in Cascades.
   Logische Knoten sind aus der Klasse `LOG_OP_ARG` abzuleiten. Konstruktoren und Destruktoren müssen entworfen und mehrerer Methoden redefiniert werden. Die genauen Schritte sind in [2] Kapitel 6.3 aufgeführt.
2. Erstellen des **physischen Knotens** UDTO in Cascades
   Physische Knoten sind aus der Klasse `PHYS_OP_ARG` abzuleiten. Konstruktoren und Destruktoren müssen entworfen und mehrerer Methoden redefiniert werden. Die genauen Schritte sind in [3] Kapitel 7.1 aufgeführt.
3. Erstellen der **Implementierungsregel**, die den logischen auf den physischen Operator abbildet. Alle Regeln leiten sich in Cascades aus der Klasse `FUNCTION_RULE` ab. Regelmuster, Regelsubstitute und Bedingungen zur Regelanwendung sind wie in [3] Kapitel 7.3 erläutert anzugeben.
4. Erweiterung der **Konvertierungsroutinen**
   Die Konvertierungsroutine `Midas_to_Model_M` durchläuft den MIDAS-Baum einmal rekursiv in Nachordnung (d.h. die Söhne werden vor dem Vaterknoten konvertiert), um den entsprechenden Cascades-Baum aufzubauen ([2] Kapitel 4.3.2). Sie erkennt weder UDTOs noch Send- und Receive-Knoten und ist daher so zu erweitern, dass sie alle SR-Knoten überliest und nur den UDTO-Knoten nach Cascades konvertiert. (Vor der Optimierung befinden sich lediglich zu UDTOs gehörige SR-Knoten im Baum.) Sie überträgt dabei seine logischen Eigenschaften in die Cascades-Struktur. Eine für ihre Erweiterung notwendige Vorarbeit wurde bereits vorgenommen und ist im folgenden Abschnitt beschrieben.

   Umgekehrt muss die Konvertierungsroutine `Model_M_to_Midas` so erweitert werden, dass sie einen physischen UDTO in Cascades samt seinen Eigenschaften

nach MIDAS überträgt und die ihn umgebenden SR-Paare wieder im MIDAS-Baum einfügt. Ihre Arbeitsweise ist [3] Kapitel 6.2 genau beschrieben. Beide Routinen dürfen keine unreferenzierten Strukturen im Speicher zurücklassen.

5. Erweiterung des **DDL-Statements** `CREATE TABLE_OPERATOR` zur Spezifizierung der Sortierordnung der Ausgaberelation eines UDTO. (Die Parallelisierbarkeit eines UDTO kann ja bereits in diesem DDL-Statement angegeben werden.) Die Erweiterung könnte nach der Spezifikation der Ausgaberelation eingefügt werden und durch eine `ORDER_BY`-Klausel in der gewohnten Weise die Sortierordnung der Attribute spezifizieren. Neben `asc` und `desc` kann `conserve` bzw. `invert` für Attribute angegeben werden, die nicht sortiert werden, deren Sortierordnung aber vom UDTO beibehalten bzw. umgekehrt wird. Nicht erwähnte Ausgabeattribute besitzen keinerlei Sortierung.

4.3 Am Parser vorgenommene Erweiterung

Problembeschreibung:

Zu den logischen Eigenschaften eines Knotens gehört das Schema seiner Ausgaberelation, die er an seinen Vaterknoten weiterreicht. Das Schema beinhaltet Anzahl, Namen und Datentypen der Attribute einer Relation. Die Konvertierungsroutine `Midas_to_Model_M` versieht jedes im MIDAS-Baum vorkommende Attribut mit einem global eindeutigen Namen, durch den es in Cascades identifiziert wird. Bei der Umwandlung eines Knotens überträgt sie aus den Schemas der Söhne (der Eingaberelationen) die global eindeutigen Namen derjenigen Attribute, die in der Ausgaberelation des Knotens enthalten sind. Sie muss also die Herkunft jedes Ausgabeattributs kennen und wissen, von welchem Sohn und von welcher Position innerhalb des Sohnes es stammt. Genau diese Information stand im Falle eines UDTO mit Attributpropagierung nur für die formalen, nicht aber für die propagierten Attribute zur Verfügung.

Lösung:

Es wäre möglich gewesen, die Konvertierungsroutine dahingehend zu erweitern, dass sie aus dem Systemkatalog die formale Definition des UDTO abfragt und die Herkunft der propagierten Attribute selbst rekonstruiert. Aus zwei Gründen war es jedoch angebrachter und auch im Sinne einer effizienten Implementierung, stattdessen den Parser zu erweitern. Erstens ist es von der Programmlogik her betrachtet typischer Aufgabenbereich des Parsers, Knotenstrukturen aufzubauen, die alle relevanten Informationen aus der verarbeiteten Anfrage enthalten. Dazu gehören auch die propagierten Attribute, die jeweils von den in der Anfrage enthaltenen Eingabetabellen des UDTO stammen. Zweitens waren dem Parser bereits alle notwendigen Informationen zugänglich, weil er sowieso auf die entsprechenden Tabellen des Systemkatalogs zugreifen musste. Die Erweiterung betrifft die Prozedur `udto_secp`

(aus Datei *mqlpa2.c*) zum Parsen des UDTO und die in *optree.h* definierte MIDAS-Struktur `Udto`.

Die Struktur `Udto` wurde um die Substruktur `AttributeInfo` erweitert. In ihr werden beim Parsen des UDTO zu jedem propagierten Ausgabeattribut die Nummer des Sohnes, von dem es stammt, und die Position innerhalb des Sohnes, von der es stammt, abgelegt. Die Anzahl und Datentypen aller Ausgabeattribute sind wie bei jedem MIDAS-Knoten in `outobj` abgelegt. Die Prozedur `Convert_Udto`, die den Knoten UDTO von MIDAS nach Cascades konvertiert, kann nun aus der Struktur `AttributeInfo` die benötigte Herkunftsinformationen propagierter Attribute entnehmen.

5 Erweiterung des Parallelisierers

Kenntnisse von Aufbau und Funktionsweise des Parallelisierers sowie Grundwissen über regel- und kostenbasierte Anfrageparallelisierung und die damit verbundene Problematik werden in diesem Kapitel vorausgesetzt. Das Phasenkonzept sowie Strukturen und Arbeitsweise von COF müssen geläufig sein. Zur Einarbeitung in die Thematik empfiehlt sich die Diplomarbeit [1] über den Entwurf des Parallelisierers.

Das Kapitel stellt einen Entwurf vor, der alle Schritte enthält, die notwendig sind, um den UDTO in den Parallelisierer zu integrieren. In diesem Sinne dient der Entwurf auch als Anleitung für die noch ausstehende Implementierung. Die Integration des UDTO in die Kostenmodelle des Optimierers wird im nächsten Kapitel behandelt.

5.1 Grundsätzliche Überlegungen

Der Parallelisierer wurde mit COF erstellt und besteht aus vier Komponenten. Im Unterschied zum Optimierer arbeitet er in einzelnen Phasen und besitzt ein wesentlich komplexeres Kostenmodell. Seine Konvertierungsroutinen, die auch hier die Schnittstelle zu MIDAS darstellen, sind andere als die des Optimierers. Die Erweiterung betrifft das Kostenmodell und die Regeln der einzelnen Phasen, die der Parallelisierung zugrunde liegen, sowie die Konvertierungsroutinen.

Folgende grundsätzlichen Fragen sind zu klären: Wie sollen UDTO und die ihn umgebenden SR-Knoten im Parallelisierer (Cascades) dargestellt werden? Welche Parallelisierungsregeln sind auf den UDTO anzuwenden? Welche charakterisierende Informationen benötigt der Parallelisierer (von der Kostenberechnung abgesehen)?

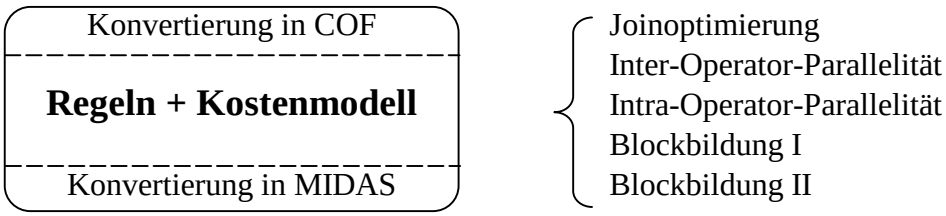


Abbildung 5.1: Komponenten und Phasen des Parallelisierers

5.1.1 Darstellung des UDTO

Im Parallelisierer wird ein Send-Receive-Paar als ein Cascades-Knoten dargestellt, der die Parameter beider Knotentypen beinhaltet ([1] Kapitel 5.1.2). Die Konvertierungsroutine nach Cascades bildet daher ein SR-Paar im MIDAS-Baum auf einen gemeinsamen SR-Knoten ab, der im Parallelisierer nur noch Send-Knoten genannt wird. Für dieses Kapitel übernehmen wir die in [1] gewählte Bezeichnung des Doppelknotens. Der Parallelisierer benötigt im Gegensatz zum Optimierer die Send-Knoten, da er mit ihnen arbeitet und dabei ihre Parameter verändert. Er fügt sie z.B. in den Baum ein, um Operatoren zu parallelisieren, verschiebt sie und fasst sie bei Bedarf zusammen, um größere Blöcke zu bilden. Wir können daher die einfache Vorgehensweise vom Optimierer nicht anwenden, sondern müssen uns dem Parallelisierer anpassen und die den UDTO umgebenden SR-Paare auf Send-Knoten abbilden. Der UDTO wird im Parallelisierer als eigener Cascades-Knoten dargestellt, der von Send-Knoten umgeben sein muss. Die sieben Knoten des Operatorbaumausschnitts aus Abbildung 3.2 werden im Parallelisierer also folgendermaßen dargestellt:

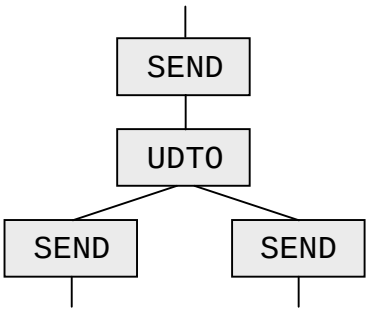


Abbildung 5.2: UDTO mit umgebenden Send-Knoten

Die Einheit des UDTO und seiner Send-Knoten darf vom Parallelisierer nicht aufgebrochen werden. Er darf Parameter der Send-Knoten ändern, was im Sinne einer guten Parallelisierung auch wünschenswert ist. Er darf aber keinesfalls Send-Knoten des UDTO verschieben oder entfernen, da der UDTO dann zum unmittelbaren Sohn oder Vater des dem Send folgenden Knotens würde, mit dem er ohne zwischengeschalteten Send-Knoten keine Daten austauschen könnte. Dasselbe gilt auch für die Send-Knoten zwischen zwei unmittelbar aufeinanderfolgenden UDTOs, da die UDTOs mit Hilfe von Funcstart-Prozessen ausgeführt werden und nicht von ein und demselben Interpreter direkt ausgeführt werden können.

5.1.2 Parallelisierungsphasen und -regeln

Intern arbeitet der Parallelisierer in fünf Phasen, die nacheinander ablaufen und verschiedene Ziele verfolgen. Innerhalb einer Phase kommt nur der zu ihr gehörige Satz von Transformationsregeln zur Anwendung. Eine kleine Gruppe von Implementierungsregeln (`LogOp_to_PhysOp`) ist in allen Phasen aktiv, um einen Knoten, auf den keine Transformationsregeln mehr angewendet werden sollen, in einen rein physischen Knoten zurückzuverwandeln. Die Phasen sind im einzelnen in [1] Kapitel 5.3 erläutert. Wir werden hier zu jeder Phase untersuchen, ob und wie sie sich auf den UDTO auswirken soll, um ihn in die Parallelisierung miteinzubeziehen.

Joinoptimierungsphase

Die Phase Joinoptimierung gehört, wie der Name schon sagt, zum Optimiervorgang und wird lediglich aus programmhistorischen Gründen als erste Phase des Parallelisierers ausgeführt. Sie ersetzt einen Merge-Join durch Hash-Join, wenn dies effizienter ist. Ihre Tätigkeit berührt den UDTO nicht, weshalb diese Phase unverändert bleibt.

Pipeliningphase

Diese Phase erzeugt Inter-Operator-Parallelität, indem sie versucht, teure Teilbäume durch eine Pipe, d.h. durch einen nicht-partitionierenden Send-Knoten, vom Rest des Baumes abzuspalten. Ein UDTO ist per se durch Send-Knoten vom umgebenden Operatorbaum abgespalten, womit die Arbeit dieser Phase bezüglich ihm bereits getan ist. Ihre Regeln bedürfen daher keiner Anpassung oder Ergänzung.

Da die Phase anhand von Kostenvorentscheidungen im Baum nach geeigneten Trennstellen für Pipes sucht, muss bei der Kostenberechnung berücksichtigt werden, dass ein UDTO mit n Söhnen den Operatorbaum bereits in n+2 Teilbäume aufspaltet und entsprechend viele Interpreter benötigt. Der in Abbildung 5.2 dargestellte UDTO spaltet den gesamten Operatorbaum in die vier Teile linker Unterbaum, rechter Unterbaum, UDTO und Vaterbaum auf.

Partitionierungsphase

Diese Phase parallelisiert teure Knoten durch Intra-Operator-Parallelität, indem sie über und unter ihnen partitionierende bzw. replizierende Send-Knoten einfügt. Da ein UDTO in aller Regel ein teurer Knoten ist, soll er von dieser Phase bei Bedarf parallelisiert werden. Dazu müssen keine neuen Send-Knoten einfügt, sondern lediglich die Send-Knoten des UDTO in partitionierende bzw. replizierende Sends umwandelt werden.

Woher aber weiß der Parallelisierer, ob und wie er die Eingaberelationen eines UDTO partitionieren darf? Bei einem Merge-Join zum Beispiel wissen wir, dass wir entweder beide Eingaben nach den Joinattributen partitionieren müssen oder eine Eingabe beliebig partitionieren können und die andere dann replizieren müssen, weil wir den Operator und seine Funktionsweise kennen. Bei einem UDTO besitzt dieses Wissen nur derjenige, der ihn entworfen hat. Damit der Benutzer dem System dieses Wissen mitteilen kann, wurde in MIDAS eine DDL-Anweisung teilweise implementiert ([5] Kapitel 3.3.2 und 6.2.4). Sie ermöglicht es, zu den Eingabetabellen eines UDTO verschiedene gültige Partitionierungskonfigurationen anzugeben. Pro Tabelle kann zwischen den Partitionierungsarten `ANY`, `EQUAL` und `RANGE` gewählt werden; ein Bindestrich zeigt an, dass die Tabelle nicht partitioniert werden darf und somit repliziert werden muss.

`ANY:` Die Tabelle kann beliebig partitioniert werden.

`EQUAL <PF>(<Attributliste>):` Auf die angegebenen Attribute der Tabelle wird die (benutzerdefinierte) Partitionierungsfunktion <PF> angewandt; alle Tupel deren Attribute den gleichen Funktionswert liefern, kommen in dieselbe Partition (Hash-Partitionierung).

`RANGE(k) <PF>(<Attributliste>):` Auf die angegebenen Attribute der Tabelle wird die (benutzerdefinierte) Partitionierungsfunktion <PF> mit Parameter k angewandt; alle Tupel, deren Attribute von der Funktion in den durch k angegebenen Bereich abgebildet werden, kommen in eine Partition (Bereichspartitionierung).

Das Beispielstatement spezifiziert für den in Kapitel 3.1.2 beschriebenen UDTO *eq-join* drei mögliche Konfigurationen zur Partitionierung. Entweder wird eine Eingabe beliebig partitioniert und die jeweils andere repliziert, oder die Tupel beider Eingaben werden durch die Funktion *pf_int* und *pf_str* auf die jeweiligen Partitionen verteilt. Die beiden Partitionierungsfunktionen verarbeiten unterschiedliche Datentypen, müssen aber für Attribute, die der Operator *eq-join* als äquivalent betrachtet, jeweils denselben Funktionswert liefern.

```
CREATE TABLE_OPERATOR eq-join
  (TABLE input1 (num1 INTEGER),
   TABLE input2 (num2 STRING) )
RETURNS TABLE output (input1.+, num INTEGER, input2.+)
ALLOW PARALLEL ((ANY, -), (-, ANY),
  (EQUAL pf_int(num1), EQUAL pf_str(num2)))
AS EXTERN '/my_path/libudto.so'#'equi_join_int2str'
```

Der Regelsatz dieser Phase ist also um eine Regel zu erweitern, die einen UDTO parallelisiert, sofern es sich um einen teuren Operator handelt und seine Parallelisierung erlaubt ist. Um dies zu prüfen, muss sie auf den Systemkatalog zugreifen und die Kosten des Operators abschätzen. Kommt die Regel zur Anwendung, wandelt sie die Send-Knoten des UDTO gemäß einer der zulässigen Konfigurationen in partitionierende bzw. replizierende Send-Knoten um.

In dieser Phase wird auch über die Realisierung teurer Relationenscans durch parallele Scans (Rrscan, Pscan) entschieden. Die parallelen Scans liefern die Tupel in ungeordneter Reihenfolge und können folglich nur dann zum Einsatz kommen, wenn die benötigten physischen Eigenschaften der im Baum befindlichen Operatoren dies zulassen. Physische Eigenschaften werden in Cascades bottom-up berechnet. Daher muss sichergestellt sein, dass erzeugte Sortierordnungen von Operatoren, die sich unterhalb eines UDTO im Baum befinden, korrekt über den UDTO hinweg nach oben weitergegeben werden, wo sich eventuell ein Operator befindet, der eine bestimmte Sortierordnung benötigt. Die bereits beim Optimierer in Kapitel 4.1.3 erwähnte charakterisierende Information über Sortierordnungen wird also auch vom Parallelisierer benötigt.

Blockbildungsphase I

In der Blockbildungsphase I werden die in der letzten Phase eingefügten, partitionierenden Send-Knoten von den teuren Operatoren weg nach oben bzw. unten verschoben. Das Ziel ist, größere parallelisierte Teilbäume (Blöcke) zu bilden, die dann auch die weniger teuren, noch nicht parallelisierten Knoten enthalten. Treffen durch diese Verschiebungen zwei partitionierende Send-Knoten aufeinander, werden sie zu einem repartitionierenden Send-Knoten zusammenfasst, was einen Interpreter bei der Ausführung einspart. Trifft ein partitionierender Send-Knoten auf einen Relationenscan, werden beide durch einen parallelen Scan ersetzt, sofern die Partitionierung beliebig ist.

Die Intra-Operator-Parallelisierung eines UDTO kann nicht auf umliegende Knoten ausgedehnt werden, da seine Send-Knoten nicht von ihm weg verschoben werden dürfen. Ebenso wenig dürfen sie durch einen parallelen Scan ersetzt werden. Die Regelmuster der Verschiebungsregeln und der Ersetzungsregeln sind daher so anzupassen, dass sie auf Send-Knoten, die zu einem UDTO gehören, nicht angewandt werden. Wird ein Send-Knoten an den Send-Knoten eines UDTO von oben oder unten herangeschoben, spricht nichts dagegen, beide zu einem repartitionierenden Send-Knoten zu verschmelzen. Die dafür zuständige Regel *SendSend_to_Send* bleibt also unverändert.

Blockbildungsphase II

Die Blockbildungsphase II entfernt repartitionierende Send-Knoten, sofern die Partitionierungsarten der angrenzenden Blöcke kompatibel sind. So entstehen noch größere Blöcke und unnötige Repartitionierungen werden vermieden. Zu einem UDTO gehörige Send-Knoten dürfen nicht entfernt werden. Daher ist die einzige Regel dieser Phase, *Remove_SendNode*, so anzupassen, dass sie nicht angewandt wird, wenn sich der repartitionierende Send-Knoten unmittelbar ober- oder unterhalb eines UDTO befindet.

Nachbehandlung

In der sich anschließenden Nachbehandlung (6. Phase) werden für die gebildeten Blöcke die optimalen Parallelitätsgrade und Partitionierungsarten ermittelt. Diese Parameter werden dann in die zugehörigen Send-Knoten eingetragen. Für den UDTO und seine Send-Knoten soll diese Nachbehandlung auch durchgeführt werden. Dazu müssen seine geschätzten Kosten und seine Parallelisierungsparameter zur Verfügung stehen.

Als weiterer Teil der Nachbehandlung wird die Anzahl der Rahmen festgelegt, die jedem Knoten im Cache exklusiv für Zwischenergebnisse zur Verfügung stehen. Besonders bei Multiprozessor-Rechnern ist es wichtig, diese Anzahl zu beschränken, da mehrere Interpreter gleichzeitig aktiv sein können und dann die Knoten eines Teilbaumes sowohl untereinander als auch mit den Knoten anderer Teilbäume aus eventuell anderen Anfragen um den vorhandenen DB-Cache konkurrieren. Die Beschränkung hat starke Auswirkungen auf den Mehrbenutzerbetrieb und die Performance (siehe [4] Anhang B.2)!

Für einen UDTO muss die Anzahl der exklusiv verfügbaren Cache-Rahmen beschränkt werden, da er mit seinen Send-Knoten und eventuell auch mit den Knoten anderer Teilbäume um den vorhandenen DB-Cache konkurriert. Der Parallelisierer muss dazu den maximalen Speicherbedarf des UDTO ermitteln (Algorithmus siehe [1] Kapitel 5.3.6), wofür er wiederum die Kardinalität seiner Ausgaberelation benötigt. (Diese wird bereits vom Optimierer berechnet und in den Baum eingetragen.) Das derzeit bei Hash-Joins und Sort-Knoten praktizierte Verfahren, einen konstanten Wert als maximalen Speicherbedarf anzugeben, kann natürlich genauso für den UDTO angewandt werden; es stellt jedoch keinen anzustrebenden Endzustand sondern lediglich eine aus der Not heraus geborene Übergangslösung dar.

Inhaltlich im Zusammenhang mit der Beschränkung der Cache-Rahmen ist ein weiterer Schritt wünschenswert, um für eine möglichst effiziente Ausführung zu sorgen: Liest ein UDTO eine Eingaberelation nur einmal, was der Normalfall sein dürfte, sollte der entsprechende Receive-Knoten mit der Option K_READONCE versehen werden. Sie erlaubt ihm, Seiten seines Kommunikationssegments nicht erst am Ende der Transaktion, sondern gleich nachdem er sie gelesen hat, aus dem DB-Cache zu entfernen ([7] Kapitel 5.1.6). Dadurch kann es vermieden werden, dass Seiten ausgelagert und

später mittels teurer Plattenzugriffe wieder eingelagert werden. Diesen Schritt sollte allerdings nicht der Parallelisierer sondern bereits der Compiler beim Erstellen der Receive-Knoten eines UDTO durchführen, um die dadurch erreichbare Effizienzsteigerung stets zu garantieren und nicht von einer eventuellen Parallelisierung abhängig zu machen. Zur Implementierung dieses Schritts muss dem DBS von jeder Eingaberelation des UDTO bekannt sein, ob er sie einmal oder mehrmals liest.

5.1.3 Charakterisierende Informationen

In diesem Abschnitt wollen wir die charakterisierenden Informationen, die sich aus den grundsätzlichen Überlegungen des Entwurfs zur Integration des UDTO in den Parallelisierer ergeben, noch einmal im Überblick zusammenstellen:

Um die Parallelisierung eines UDTO im Sinne der Intra-Operator-Parallelität überhaupt durchführen zu können und dazu den optimalen Parallelitätsgrad zu ermitteln, werden seine **Parallelisierungsparameter** (die zulässigen Partitionierungskonfigurationen) und seine **Ausführungskosten** benötigt.

Die bereits für den Optimierer geforderte Kenntnis der **Sortierordnung** eines UDTO bzw. darüber, ob er bestehende Sortierordnungen seiner Eingaberelationen beibehält, wird auch vom Parallelisierer verwendet, um in Cascades die benötigten physischen Eigenschaften und erzeugten physischen Eigenschaften von Knoten berechnen zu können.

Die **Kardinalität der Ausgaberelation** wird gebraucht, um in der Nachbehandlung den maximalen Speicherbedarf eines UDTO ermitteln zu können. Die Kardinalität zählt zu den logischen Eigenschaften einer Relation und wird daher bereits vom Optimierer berechnet und in den Operatorbaum eingetragen; sie muss also nicht neu berechnet werden.

Informationen über mehrmaliges oder **einmaliges Lesen der Eingaben** wird benötigt, um Seiten im DB-Cache möglichst früh wieder freigeben zu können. Diese charakterisierende Information wird allerdings nicht für den Parallelisierer sondern für den Compiler gefordert. Die Erweiterung des **DDL-Statements** `CREATE TABLE_OPERATOR` zur Spezifizierung, wie oft eine Eingaberelation von einem UDTO gelesen wird, ist in Kapitel 6.2.2 angegeben.

5.2 Konkrete Implementierungsschritte

Was die Integration neuer Operatoren betrifft, ist die Dokumentation des Parallelisierers leider nicht so auskunftsfreudig wie ihre Pendants vom Optimierer. Nichtsdestoweniger sollte zur Implementierung das Kapitel 6 aus [1] herangezogen werden. Zur Integration des UDTO in den Parallelisierer sind folgende Schritte durchzuführen: (Die ersten drei Schritte entsprechen den zur Erweiterung des Optimierers nötigen Schritte und werden deswegen hier nicht näher erläutert.)

1. Erstellen des **logischen Knotens** UDTO in Cascades.

2. Erstellen des **physischen Knotens** UDTO in Cascades
3. Erstellen der **Implementierungsregel**, die den logischen auf den physischen Operator abbildet.
4. Erweiterung der Cascades-Struktur der **Send-Knoten**
   Um zu vermeiden, dass unzulässige Regeln auf die Send-Knoten eines UDTO angewandt werden, sollten sie zur Kennzeichnung um ein Flag *belongs_to_UDTO* erweitert werden, das von den entsprechenden Regeln vor ihrer Anwendung überprüft wird.
5. Erweiterung der **Konvertierungsroutinen**
   Die Konvertierungsroutine nach Cascades wandelt Send-Receive-Paare bereits in einen gemeinsamen Send-Knoten um. Sie muss also lediglich dahingehend erweitert werden, dass sie zu einem UDTO gehörige Send-Knoten durch das Flag kennzeichnet und den Knoten UDTO einschließlich seiner logischen und physischen Eigenschaften vom MIDAS- ins Cascades-Format überträgt.

   Umgekehrt muss die Konvertierungsroutine nach MIDAS so erweitert werden, dass sie einen physischen UDTO in Cascades samt seinen Eigenschaften nach MIDAS zurückkonvertiert. Das Flag *belongs_to_UDTO* hat für sie keine Bedeutung. Beide Routinen dürfen keine unreferenzierten Strukturen im Speicher zurücklassen.
6. Modifizierung und Erweiterung der **Regeln** und der **Nachbehandlung**, um die Parallelisierung eines UDTO korrekt und vollständig durchzuführen.

6 Erweiterung der Kostenmodelle

Die Lektüre dieses Kapitels setzt profunde Kenntnisse in der Kostenberechnung einschließlich der Kostenformeln voraus, die in den beiden Modellen des Optimierers (sequentielles und quasi-paralleles) und im wesentlich komplexeren Modell des Parallelisierers verwendet werden. Als Literatur zur gezielten Einarbeitung in die Kostenmodelle sind [1] Kapitel 7, [2] Kapitel 5 und [3] Kapitel 5 heranzuziehen; der Ansatz des in dieser Studienarbeit konkretisierten und entwickelten Entwurfs zur Kostenberechnung des UDTO ist in [5] Kapitel 7.5.2 beschrieben.

Das Kapitel beschäftigt sich mit der Frage, wie die Kardinalität der Ausgaberelation und die Ausführungskosten eines UDTO abgeschätzt werden können und was darüber hinaus erforderlich ist, um den Operator in die Kostenmodelle von Optimierer und Parallelisierer zu integrieren. Unser Blick richtet sich dabei in besonderem Maße auf die zentrale Frage unseres Entwurfs, welche Informationen zur Kostenberechnung in allgemeingültiger Form im System verankert werden können und welche Informationen operatorspezifisch sind und dem System vom Anwender mitgeteilt werden müssen. Was zur Implementierung der Erweiterung konkret zu tun und zu beachten ist, wird wiederum im letzten Abschnitt zusammengefasst.

6.1 Einführung in die Kostenmodelle

Jedes der drei Kostenmodelle hat zum Ziel, für den Ausführungsplan, der gerade untersucht wird, eine möglichst genaue Abschätzung der Kosten zu liefern, die er bei der Ausführung verursachen wird. Sie ermitteln einen Kostenwert, der in vorgegebener Gewichtung einerseits die Antwortzeit, andererseits den Ressourcenverbrauch des gesamten Plans widerspiegelt. Als Ressourcen werden die zeitliche Inanspruchnahme von CPU, Festplatten (I/O) und Netz (Kommunikationskosten) berücksichtigt. Der

Parallelisierer berücksichtigt zudem noch das Verhältnis zwischen der Anzahl der benötigten Interpreter und der Anzahl der verfügbaren Prozessoren. Um diese Zeiten berechnen zu können, werden einerseits Eigenschaften der Systemarchitektur wie die Dauer einer arithmetischen Operation und das Einlesen einer Speicherseite, andererseits Operatoreigenschaften wie Selektivität und Größe seiner Eingaberelation benötigt. Diese system- und operatorspezifischen Angaben werden in den Systemtabellen abgelegt ([3] Kapitel 5.1).

Optimierer und Parallelisierer spannen durch Anwendung ihrer Transformations- und Implementierungsregeln den sog. Suchraum auf, der die zu einer Anfrage möglichen Ausführungspläne enthält. Das Kostenmodell ermittelt zu jedem dieser Pläne den eben beschriebenen Kostenwert, woraufhin der Plan mit dem niedrigsten Wert als Endergebnis aus der Optimierung bzw. Parallelisierung hervorgeht.

Zur Berechnung der Kosten werden bottom-up die sequentiellen und parallelen Datenflüsse im Baum nachgebildet und die Kardinalitäten der Zwischenrelationen bestimmt, die ein Operator verarbeitet und erzeugt. Die Kardinalität ist daher die wichtigste Größe bei der Kostenberechnung; die Modelle können die Kosten bestenfalls mit derselben Exaktheit abschätzen, mit der die Kardinalitäten berechnet werden!

6.1.1 Die Kostenmodelle des Optimierers

Der Optimierer besitzt zwei Kostenmodelle, das sequentielle und das quasi-parallele, von denen jeweils nur eines aktiv ist. Das **sequentielle Modell** ist das einfachste und mittlerweile auch das bedeutungsloseste, da Anfragen im Normalfall parallelisiert werden sollen. Es berechnet die Gesamtkosten eines Plans, indem es die lokalen Kosten aller im Baum befindlichen Operatoren addiert.

Das **quasi-parallele Modell** wurde für Optimierung mit anschließender Parallelisierung entworfen; es modelliert daher Inter- und Intra-Operator-Parallelität, um gut parallelisierbaren Plänen bei der Optimierung den Vorzug zu geben. Da es die konkrete Einteilung in Blöcke, die der Parallelisierer am Operatorbaum letztlich vornehmen wird, nicht kennt, geht es für die Modellierung der Inter-Operator-Parallelität von der vereinfachten Annahme aus, dass ein Operator zu seiner Umgebung in Pipelining-Parallelität ausgeführt wird, sofern er kein blockierender Operator ist. Ein blockierender Operator, z.B. ein Sort-Operator stellt das Ende einer Pipeline dar, da er seinem Vaterknoten erst dann Tupel liefert, wenn der Teilbaum mit ihm als Wurzel (also die Pipeline bis zu ihm hin) vollständig abgearbeitet ist. Ein binärer Operator wie der Hash-Join, der eine seiner Eingaben erst komplett lesen muss, bevor er beginnt, die andere Eingabe zu verarbeiten, blockiert teilweise; die Pipeline des Teilbaumes, der komplett abgearbeitet sein muss, endet dann unterhalb des Operators.

Wichtig ist, an dieser Stelle festzuhalten, dass zur korrekten Modellierung des Datenflusses bezüglich Inter-Operator-Parallelität von jedem Operator die Information zur Verfügung stehen muss, ob er ein blockierender Operator ist und wie er seine Eingaben verarbeitet (sequentiell oder parallel). Entsprechend dem Datenfluss durch einen Operator (physische Eigenschaft) müssen die jeweils passenden Kostenformeln zur Berechnung der einzelnen Kostenkomponenten zum Einsatz kommen.

Zur Modellierung der Intra-Operator-Parallelität nimmt das Modell an, dass jeder Operator, der parallelisiert werden kann auch parallelisiert wird. Unter Berücksichtigung der physischen Eigenschaften *partitioniert* und *repliziert* bezüglich seiner Eingaberelationen werden seine lokalen Kosten für die Parallelitätsgrade *1...n* ermittelt, wobei *n*

die Anzahl der zur Ausführung der Anfrage verfügbaren Prozessoren darstellt. Der niedrigste dieser Kostenwerte repräsentiert die lokalen Kosten des Operators im quasi-parallelen Modell.

Wie bereits erwähnt, ist es Aufgabe des Optimierers, die Kardinalitäten der Zwischenrelationen zu berechnen und in den Operatorbaum einzutragen. Kardinalitäten zählen zu den logischen Eigenschaften von Relationen und ändern sich durch eine nachfolgende Parallelisierung nicht.

6.1.2 Das Kostenmodell des Parallelisierers

Die Datenabhängigkeit und der Datenfluss zwischen zwei Knoten bestimmen die Qualität der Parallelität, die sie während ihrer Ausführung tatsächlich erreichen können. Jeder Knoten ist dabei irgendwo zwischen den Polen SE (sequential execution, keine Parallelität) und IPE (independent parallel execution, maximale Parallelität) einzuordnen. Aus diesem Grund besitzt der Parallelisierer das aufwendigste und detaillierteste von den drei Kostenmodellen. Da er selbst festlegt, wie der Operatorbaum in parallel ausführbare Blöcke unterteilt wird, kann letztlich nur er die durch Inter- und Intra-Operator-Parallelität entstehenden Datenflüsse genau modellieren und berechnen.

Die Auflistung der folgenden Szenarien soll einen Einblick in den Zusammenhang zwischen dem Datenfluss durch einen Operator und die verschiedenen, daraus resultierenden Arten von Parallelität geben; sie ist aus diesem Grunde nicht vollständig.

- Zwischen zwei Relationenscans, die unabhängig voneinander ablaufen, besteht keine Datenabhängigkeit. Sie können in IPE ausgeführt werden.
- Eine Projektion wird tupelweise ausgeführt und kann zu jedem Eingabetupel sofort ein Ergebnistupel zu liefern. Sie kann in Pipelining-Parallelität (PP) zu ihrem Sohn- und Vaterknoten ausgeführt werden.
- Ein gewöhnlicher Send-Knoten besitzt die besondere Eigenschaft, dass er seine Ergebnisse nicht tupel- sondern seitenweise[1] weitergibt. Das bedeutet für die PP, dass der zugehörige Receive-Knoten am Anfang warten muss, bis der Send-Knoten die erste Seite mit Tupeln gefüllt hat. (Diese Verzögerung im Datenfluss wird Pipedelay genannt.)
- Ein materialisierender Send-Knoten schreibt seine Daten auf Platte, wenn es in der Pipeline über ihm zu Verzögerungen kommt. Damit kann der Teilbaum des Send-Knotens unabhängig von seinem Vaterknoten arbeiten.
- Ein Sort-Knoten blockiert. Er muss seine gesamte Eingabe gelesen haben, bevor er das erste Tupel an den Vater ausgibt. Zwischen ihm und seinem Vater ist also nur SE möglich, zu seinem Sohn kann er allerdings in PP arbeiten.

Um den Datenfluss dieser und anderer Szenarien bei Inter- und Intra-Operator-Parallelität hinreichend genau darstellen zu können, ermittelt das Modell des Parallelisierers die Kosten der Materialisierten Front (T_{begin}, $T_{parallel}$), den Flaschenhals einer Pipeline (T_{max}), die Kosten bei wiederholter Ausführung eines Blockes (T_{again}) und die Kosten materialisierender Send-Knoten (T_{end}).

Auch für dieses Modell gilt, dass seine Kostenberechnung auf den Kardinalitäten der einzelnen Zwischenrelationen aufbaut und diese ihm deshalb zur Verfügung stehen

[1] Gemeint ist eine Hauptspeicherseite im DB-Cache. Abgesehen von der letzten werden nur volle Seiten versandt. Eine MIDAS-Seite beträgt derzeit 64 KB.

müssen. Den Datenfluss durch einen Operator muss es noch detaillierter kennen als die Modelle des Optimierers, um die jeweils passenden Kostenformeln zum Einsatz zu bringen.

6.1.3 Die Kostenkomponenten

Alle Modelle bauen auf Kostenkomponenten auf, die sie zur Modellierung des Datenflusses im Baum verwenden. Je mehr Details sie dabei berücksichtigen, je exakter sie also modellieren, umso größer ist die Anzahl unterschiedlicher Kostenkomponenten, die sie in ihren Berechnungen einsetzen. In Tabelle 6.1 sind diese Kostenkomponenten aufgelistet und kurz beschrieben; die letzte Spalte gibt an, in welchen Modellen sie eingesetzt werden.

Kosten	Beschreibung der Kostenkomponente	seq	qpar	par
card(N)	Anzahl der Ergebnistupel von N	+	+	+
$T_{local}(N)$	gesamte lokale Ausführungskosten von N	+	+	+
$T_{begin}(N)$	Zeit nach der N das erste Ergebnistupel ausliefert		+	+
$T_{process}(N)$	Summe der lokalen Kosten des Teilbaumes mit N als Wurzel		+	+
$T_{max}(N)$	Arbeitskosten des teuersten Teilbaumes innerhalb einer Pipeline			+
$T_{parallel}(N)$	Kosten der materialisierten Front (T_{begin}) des letzten, zu N parallel laufenden Teilbaumes			+
$T_{again}(N)$	gesamte Kosten einer erneuten Auswertung von N			+
$T_{end}(N)$	gesamte Ausführungskosten (T_{total}) des letzten materialisierenden Send-Knotens in einer Pipeline			+
$Tl_{process}(N)$	lokale, reine Arbeitskosten von N		+	+
$Tl_{begin}(N)$	lokale Startup-Kosten, bevor N mit der Verarbeitung seiner Eingabe beginnen kann		+	+
$T^{P}_{local}(N)$	lokale Ausführungskosten eines partitionierten Knotens N		+	+
$T^{R}_{local}(N)$	lokale Ausführungskosten eines replizierten Knotens N		+	+
$T_{total}(N)$	Gesamtkosten des Ausführungsplans bis zum Knoten N	+	+	+

Tabelle 6.1: Kostenkomponenten für den Knoten N in den verschiedenen Modellen
seq: sequentielles Modell – qpar: quasi-paralleles Modell des Optimierers
par: Kostenmodell des Parallelisierers

6.2 Kostenberechnung für den UDTO

Um den UDTO in eines der Kostenmodelle zu integrieren, müssen wir zwei Anforderungen erfüllen: Wir müssen die Kardinalität seiner Ausgaberelation berechnen und wir müssen den Datenfluss durch den Operator modellieren. Letzteres bedeutet, die jeweiligen Kostenkomponenten des Modells (siehe Tabelle 6.1) anhand knotenspezifischer Kostenformeln zu berechnen. Die folgenden Abschnitte beschreiben die mit beiden Anforderungen verbundenen Probleme und wie sie gelöst werden können.

6.2.1 Berechnung der Kardinalität

Berechnung der Kardinalität im Optimierer

Die Kardinalität zählt zu den logischen Eigenschaften einer Relation. Für Basisrelationen ist sie in der Systemtabelle SYSTABSTAT eingetragen; für Zwischenrelationen muss sie berechnet werden. (Zwischenrelationen sind temporäre Relationen, die bei der Abarbeitung eines Operatorbaums von einem Knoten als Ausgaberelation erzeugt und als Eingaberelation an dessen Vaterknoten weitergereicht werden.) Unäre Operatoren verändern die Kardinalität ihrer Eingaberelation entweder gar nicht, wenn sie alle Tupel in die Ausgabe übernehmen (Projektion), oder sie selektieren anhand von Prädikaten bestimmte Tupel (Restriktion, Duplikatelimination). Binäre Operatoren erzeugen durch die Art und Weise, wie sie ihre Eingaben miteinander verknüpfen (Times im Vergleich mit Append) und durch Selektionen (Joinprädikate) stets eine neue Ausgabekardinalität *a*, die mit den Eingabekardinalitäten e_1 und e_2 in einem bestimmten mathematischen Zusammenhang steht. Im Falle eines Times ist $a = e_1 \cdot e_2$; im Falle eines Append, das die Tupel zweier Tabellen aneinander hängt ist $a = e_1 + e_2$.

Wie der Optimierer Kardinalitäten berechnet, ist in [2] Kapitel 5 ausführlich beschrieben. Als Eingabeparameter benötigt er stets die Eingabekardinalität(en) des jeweiligen Operators und die Column Unique Cardinalities (Cucards) aller Attribute der Eingaberelation(en). Die Cucard eines Attributes ist die Anzahl seiner unterschiedlichen Werte und wird zu Abschätzung der Selektivität eines Operators, der Prädikate besitzt, benötigt. Für jeden Operator wird anhand einer zu ihm gehörigen Formel die Kardinalität seiner Ausgabe bestimmt. Beispielsweise werden für einen Equi-Join mit Eingabekardinalitäten $card(S_1)$ und $card(S_2)$, der die Attribute A_1 und A_2 miteinander vergleicht, Selektivität $sel(J) = sel(A_1=A_2)$ und Kardinalität $card(J)$ berechnet durch:

$$sel(J) = 1 / \max[Cucard(A_1), Cucard(A_2)]$$

$$card(J) = sel(J) \cdot card(S_1) \cdot card(S_2)$$

Die Cucards von Attributen werden folgendermaßen bestimmt:
A sei das Attribute einer Relation *R*, die durch Selektion der Form „$A = c$“ in die neue Relation *R‘* übergeht; d_A bezeichne die *Cucard(A)* vor und d'_A nach der Selektion.
Ist *c* eine Konstante, ist $d'_A = 1$; ist *c* ein anderes Prädikat *p*, gilt $d'_A = d_A \cdot sel(p)$. Cucards von Attributen *B* der Relation *R*, die nicht im Prädikat enthalten sind, werden im Optimierer mit Hilfe eines Urnenmodells aus $card(R')$ und d_B berechnet durch:

$$d'_B = \lceil d_B \cdot (1 - (1 - 1/d_B)^{card(R')}) \rceil$$

Berechnung der Kardinalität des UDTO

Wie können wir für einen prozeduralen UDTO die Kardinalität seiner Ausgaberelation berechnen? Da wir die jeweilige Funktionalität des Operators nicht kennen, können wir keine starre Formel festlegen, wie dies für den Join-Operator und die anderen Operatoren des DBS möglich ist. Von einem binären UDTO mit Eingabekardinalitäten e_1 und e_2 wissen wir z.B. nicht, ob er seine Eingaberelationen zu einem kartesischen Produkt mit Kardinalität $e_1 \cdot e_2$ verknüpft oder ob er sie auf irgendeine Art konkateniert und die Kardinalität e_1+e_2 beträgt. Wir wissen auch nicht, ob eines oder mehrere der Eingabeattribute des UDTO von diesem als Selektionsprädikate verwendet werden und ob die Selektion auf Gleichheit, Ungleichheit, einem Ist-Größer-Als oder einem anderen Vergleich zwischen den Prädikaten beruht. Es lassen sich noch weitaus kompliziertere Beispiele konstruieren. Man denke nur an einen UDTO mit drei Eingaberelationen, aus denen er jeweils ein Attribut (Seitenlänge) entnimmt und anhand der Dreiecksbedingung $(a+b > c) \wedge (b+c > a) \wedge (c+a > b)$ selektiert.

Das Wissen, wie die Kardinalität der Ausgaberelation berechnet werden muss, besitzt nur derjenige, der den UDTO entwickelt hat. Die Idee ist daher, dem Benutzer die Möglichkeit zu geben, zu einem UDTO eine Schätzfunktion im Datenbanksystem zu registrieren, die die Kardinalität berechnet. Eine solche Schätzfunktion ist eine UDSF, die der Benutzer durch ein erweitertes DDL-Statement im System registriert, wobei er den Namen des UDTO angibt, für den sie die Kardinalität berechnet. Trifft der Optimierer im Baum auf einen UDTO, so ruft er die zugehörige Schätzfunktion, die sich in einer DLL befindet, auf und übergibt ihr die erforderlichen Eingabeparameter ([5] Kapitel 7.5.2). Diese Parameter sind die Kardinalität zu jeder Eingaberelation des UDTO und die Cucards zu jedem Eingabeattribut. Die Parameterübergabe kann entweder über den Systemkatalog erfolgen, indem der Optimierer die Meta-Daten zu den Eingaberelationen dort ablegt und die Schätzfunktion darauf zugreift, oder die Schätzfunktion wird mit einem einzigen Parameterstring aufgerufen, in den alle Meta-Daten hineincodiert sind.

Die letztere der beiden Möglichkeiten ist einfacher zu realisieren, da der Systemkatalog nicht erweitert werden muss. Sie ist zudem die effizientere Methode zur Übergabe der Parameter, da im anderen Fall für jede Eingabetabelle des UDTO ein Eintrag in den Systemkatalog geschrieben und anschließend von der Schätzfunktion gelesen werden muss. Aus diesen Gründen entscheiden wir uns für die Übergabe eines Strings, der für jede Eingaberelation durch Kommas getrennt deren Kardinalität und die Cucards ihrer Eingabeattribute in der Reihenfolge enthält, wie sie beim Registrieren des UDTO angegeben wurden. Die Schätzfunktion decodiert den String, berechnet die Kardinalität und übergibt ihren Ergebniswert dem Optimierer.

Diese Art der Kostenberechnung bringt für den Programmierer von UDTOs erhöhten Aufwand und die Auseinandersetzung mit der ihm bisher eventuell fremden Materie der Kardinalitäts- und Selektivitätsabschätzung mit sich. Es ist im Moment jedoch der einzig gangbare Weg, um die Kardinalität für die Ausgaberelation des UDTO überhaupt zu erhalten, ohne die eine Integration des Operators in die Kostenmodelle von Optimierer und Parallelisierer nicht möglich ist. Fehlt zu einem UDTO die Schätzfunktion, kann die Anfrage folglich nicht optimiert und parallelisiert werden.

Um die Cucards der Ausgabeattribute zu berechnen, kann für propagierte Attribute die oben angegebene Formel des Urnenmodells verwendet werden. Da durch die Schätzfunktion nun die Kardinalität von R' zur Verfügung steht, können aus den Cucards der Eingabeattribute die Cucards der Ausgabeattribute berechnet werden. Ungelöst bleibt allerdings die korrekte Berechnung Cucards der unmittelbaren

Ausgabeattribute des UDTO. Das Verfahren der Schätzfunktionen könnte auch auf sie ausgedehnt werden, indem für jedes Attribut eine UDSF im DBS registriert wird. Es ist einerseits die einzige Möglichkeit, um die Cucards dieser Attribute korrekt zu berechnen, macht aber andererseits das Konzept UDTO für den einzelnen Benutzer so aufwendig und kompliziert, dass es entweder für ihn unbenutzbar wird, oder er sich durch falsche, schnell erstellte Schätzfunktionen behilft. Um das zu vermeiden, sollte das Eintragen von Schätzfunktionen für Cucards optional sein. Ist zu einem Attribut keine Funktion registriert, wird die Formel des Urnenmodells zur näherungsweisen Abschätzung der Cucard verwendet.

6.2.2 Modellieren des Datenflusses

Der Datenfluss wird in den einzelnen Modellen durch die jeweiligen Kostenkomponenten modelliert. Im sequentiellen Kostenmodell sind dies T_{local} und T_{total}, im quasi-parallelen Modells kommen die Komponenten T_{begin} und $T_{process}$ hinzu und im Modell des Parallelisierers werden schließlich alle Kostenkomponenten eingesetzt. Die Berechnung der Komponenten erfolgt im Baum bottom-up, wobei für jede im Modell verwendete Komponente zu jedem Operator eine knotenspezifische Formel existiert, die den Datenfluss durch den Operator zum Ausdruck bringt.

Für einen UDTO kann das DBS wie im Falle der Kardinalitätsberechnung keine fixen Formeln einsetzen, da der Datenfluss durch einen UDTO jedes Mal ein anderer sein kann. Wiederum kennt nur der Entwickler eines UDTO dessen Datenfluss. Es wäre allerdings keine gute Idee, dem Benutzer deshalb auch noch die Angabe der Formeln der einzelnen Kostenkomponenten abzuverlangen, da er sich dazu komplett in alle Kostenmodelle einarbeiten müsste, womit das Konzept UDTO kein praktikables mehr wäre.

Das System kann zur Kostenberechnung wesentlich mehr beitragen, als dies bei der Berechnung der Kardinalität der Fall ist. Die Idee ist daher, das System in die Lage zu versetzen, die benötigten knotenspezifischen Kostenformeln selbst zu synthetisiert. Es soll dabei mit möglichst wenigen Angaben über den jeweiligen UDTO auskommen. Wir müssen also zuerst untersuchen, von welchen Eigenschaften eines Knotens der Datenfluss abhängt. Darauf aufbauend können wir uns überlegen, wie der Benutzer diese Eigenschaften dem System mitteilen kann und wie das System daraus die knotenspezifischen Formeln eines UDTO zur Berechnung der einzelnen Kosten-komponenten synthetisieren kann.

Datenflussrelevante Operatoreigenschaften

Der Datenfluss hängt von folgenden Eigenschaften eines Knotens ab:

- Handelt es sich um einen blockierenden Knoten wie den Sort-Knoten, der erst dann Tupel ausliefert, wenn er seine Eingaben vollständig verarbeitet hat? Handelt es sich um einen teil-blockierenden Knoten wie den Hash-Join, der eine seiner Eingaben komplett gelesen haben muss, bevor er Tupel ausliefert? Oder liefert der Knoten kontinuierlich während seiner gesamten Arbeitsphase Tupel aus?
- In welcher Reihenfolge werden die Eingaben gelesen? Werden sie nacheinander, teilweise parallel zueinander oder alle gleichzeitig gelesen? Man denke z.B. an einen UDTO mit drei Eingaben R_1, R_2 und R_3, der zuerst R_2 komplett und danach aus R_1 und R_3 parallel liest.

- Wie oft wird eine Eingabe gelesen? Wird sie einmal gelesen oder wird sie mehrmals in Abhängigkeit von der Kardinalität einer anderen Eingabe gelesen (wie bei einem Nested-Loops-Join)?

Diese Eigenschaften wirken sich unmittelbar auf die Berechnung *aller* Kostenkomponenten außer T_{local} aus. Sie gehören daher zu der Menge der charakterisierenden Informationen und müssen dem System zu jedem UDTO vom Benutzer mitgeteilt werden. Den Datenfluss betreffend besitzt das System ohne diese Angaben lediglich Kenntnis darüber, wie viele Söhne der UDTO besitzt und wie er im Sinne der Intra-Operator-Parallelität parallelisiert werden darf.

Wie soll nun der Benutzer dem System diese Eigenschaften zu einem UDTO mitteilen? Wir schlagen dazu vor, das DDL-Statement `CREATE TABLE_OPERATOR` wiederum geeignet zu erweitern. Zur Spezifizierung der angegebenen drei Operatoreigenschaften wird es um folgende Klauseln erweitert:

- `BLOCKS_ON:` Der Klausel folgt eine Liste derjenigen Eingabetabellen, die komplett gelesen werden müssen, bevor der UDTO das erste Ergebnistupel liefern kann. So kann bei einem teilweise blockierenden Operator genau angegeben werden, auf welchen Eingaben er blockiert. Bei einem blockierenden Operator kann anstelle aller Eingabetabellen der Einfachkeit halber auch `BLOCKS_ON ALL` angegeben werden.
- `INPUT_ORDER:` Die Klausel spezifiziert die Reihenfolge, in der die Eingaben vom UDTO gelesen werden. Ihr folgt ebenfalls eine Liste der Eingabetabellen, wobei die Tabellen durch Komma und Plus-Zeichen miteinander verknüpft und zu Ausdrücken geklammert werden können. Seien T_1 und T_2 Eingabetabellen. „T_1,T_2“ bedeutet, T_1 wird vor T_2 gelesen; „T_1+T_2“ bedeutet, T_1 und T_2 werden parallel gelesen.
  Für den oben angegebenen UDTO mit den Eingaben R_1, R_2 und R_3 sieht die Klausel folgendermaßen aus: `INPUT_ORDER` R_2`,(`R_1`+`R_3`)`
  Die Nicht-Angabe von Eingabetabellen bedeutet, dass diese nach den angegebenen Tabellen parallel verarbeitet werden. In unserem Beispiel genügt also auch die abgekürzte Angabe `INPUT_ORDER` R_2. Fehlt die Klausel ganz, so bedeutet dies, dass aus allen Eingaben parallel gelesen wird.
- `READS_N_TIMES:` Die Klausel beschreibt, wie oft eine Eingabe gelesen wird. Die Angabe von Konstanten reicht hier leider nicht aus, da es (wie bei dem Times-Knoten) von der Kardinalität der einen Eingabe abhängen kann, wie oft eine andere Eingabe gelesen wird. Der Klausel enthält je einen Eintrag zu allen Eingabetabellen, der entweder eine Konstante oder ein mathematischer Ausdruck sein kann. Eine „1“ bedeutet, dass die Eingabe nur einmal gelesen wird; ein Ausdruck der Form $0{,}5 \cdot card(T_2)$ bedeutet, dass die Eingabe im Durchschnitt für jedes zweite Tupel aus Tabelle T_2 neu gelesen werden muss. Wie solche Ausdrücke vom System verarbeitet werden können, ist in einer anderen Arbeit zu klären; wir wollen hier lediglich festhalten, dass sie zur Kostenberechnung gebraucht werden.

Die Klauseln sind in dieser Reihenfolge nach der Spezifikation der Eingabetabellen einzufügen.

Synthese der Kostenformeln

Im Abschnitt 6.1.2 wurden einige Szenarien zu bekannten Operatoren aufgeführt, denen bedingt durch die datenflussrelevanten Eigenschaften jeweils andere Datenflüsse zugrunde liegen. Der UDTO besitzt die Mächtigkeit, jeden dieser Operatoren (außer den Send- und Receive-Knoten) zu simulieren. Da er in der Anzahl seiner Eingaberelationen nicht beschränkt ist, ist die Menge aller möglichen Datenfluss-Szenarien durch einen

UDTO enorm. Die Vielfalt aller tatsächlich möglichen Datenflüsse, die sich durch Variationen der Operatoreigenschaften ergeben, ist so groß, dass sie in einer eigenständigen Arbeit klassifiziert werden müssen, um darauf aufbauend eine vollständige, abgeschlossene Wissensbasis aus Teilformeln zu erstellen, aus der zu jedem Szenario die passenden Kostenformeln mit Hilfe der datenflussrelevanten Operatoreigenschaften abgeleitet werden können. Wir wollen hier lediglich exemplarisch zeigen, dass dies machbar ist.

Sei *U* ein binärer UDTO mit den Eingaberelationen R_1 und R_2, die ihm von seinen Receive-Knoten übergeben werden. Er liest erst R_1, dann R_2; er blockiert allerdings nicht, sondern führt eine Art Append-Operation durch. Wir gehen davon aus, dass die Kardinalität seiner Ausgaberelation und seine lokalen Kosten $T_{local}(U)$ bereits berechnet sind. Des weiteren stehen zu den Eingaberelationen alle Kostenkomponenten zur Verfügung. Wir werden nun nacheinander alle Kostenkomponenten des UDTO berechnen:

Da U nicht blockiert und zuerst R_1 liest, gilt: $T_{begin}(U) = T_{begin}(R_1)$.

Da ein UDTO mit seinen Send- und Receive-Knoten immer einen eigenen Block bildet, gilt: $T_{process}(U) = T_{process}(R_1) + T_{process}(R_2) + T_{local}(U)$

Anmerkung: Die Komponenten T_{begin} und $T_{process}$ müssen für das quasi-parallele Modell des Optimierers und für das Modell des Parallelisierers gleichsam berechnet werden. Die nun folgenden Komponenten T_{max}, $T_{parallel}$, T_{again} und T_{end} müssen dagegen nur noch für das Modell des Parallelisierers berechnet werden.

Da *U* zuerst mit R_1 und danach mit R_2 in Pipelining-Parallelität arbeitet, blockiert entweder die Pipe von R_2, bis der Unterbaum von R_1 abgearbeitet ist, oder der Unterbaum von R_2 ist mit einem materialisierenden Send abgeschlossen und kann unabhängig abgearbeitet werden. Die Arbeitskosten des teuersten Unterblocks der gesamten Pipeline unter *U* ergeben sich daher zu $T_{max}(U) = T_{max}(R_1) + T_{max}(R_2)$.
Im Falle des materialisierenden Sends ist $T_{max}(R_2) = 0$ und die Rechnung ebenfalls korrekt.

Die Komponente $T_{parallel}(U)$ wird benötigt, um die Kosten der Materialisierten Front eines binären Operators korrekt berechnen zu können, wenn sich beide Söhne und der Operator im selben Block befinden. Sie wird hier nicht benötigt, da die Söhne des UDTO immer Receive-Knoten sind und sich somit keine anderen Operatoren im Block des UDTO befinden können. Für die Folgerechnung wird diese Komponente vom Send-Knoten *S* des UDTO auf $T_{parallel}(S) = T_{begin}(S)$ gesetzt.

Für wiederholte Ausführung gilt stets $T_{again}(U) = T_{again}(R_1) + T_{again}(R_2) + T_{local}(U)$.

Die Komponente T_{end} beschreibt, wann ein Teilbaum mit materialisierendem Send als Wurzel seine Ausgabe vollständig geschrieben hat. Da solch ein Teilbaum unabhängig von nachfolgenden Teilbäumen arbeitet, kann $T_{end}(U)$ wie sonst auch als das Maximum von $T_{end}(R_1)$ und $T_{end}(R_2)$ berechnet werden.

Die Gesamtkosten im Baum einschließlich der Abarbeitung von *U* ergeben sich für den Parallelisierer zu

$$T_{total}(U) = T_{begin}(U) + (T_{process}(U) \,||\, T_{max}(U) \,||\, (T_{end}(U) - T_{begin}(U)),$$

wobei der Ausdruck $(T_{end}(U) - T_{begin}(U))$ für negative Werte auf Null gesetzt wird.

Der Optimierer berechnet die Gesamtkosten aus:

$$T_{total}(U) = T_{begin}(U) + T_{process}(U)$$

Nun müssen wir noch erläutern, wie die lokalen Kosten von *U* berechnet werden können. Die lokalen Startup-Kosten $T_{l_{begin}}(U)$ müssen dem DBS vom Benutzer mitgeteilt

werden, da nur er diesen Wert kennt. Die lokalen Arbeitskosten $Tl_{process}(U)$ während der Abarbeitung der Eingaberelation(en) können mit Hilfe der Kardinalitäten $card(U)$, $card(R_1)$ und $card(R_2)$ berechnet werden, wenn der Benutzer dem System die geschätzten Kosten c_i pro gelesenem Tupel der jeweiligen Eingabe und pro erzeugtem Tupel der Ausgabe angibt. Für beide Werte (Startup und Arbeitskosten) soll der Benutzer die Möglichkeit haben, die jeweils geschätzte Anzahl an CPU-Instruktionen und an I/O-Operationen anzugeben. Nun können wir die letzten in unserer Kostenrechnung noch fehlenden Werte berechnen:

$$Tl_{process}(U) = c_1 \cdot card(U) + c_2 \cdot card(R_1) + c_3 \cdot card(R_2)$$

$$T_{local}(U) = Tl_{begin}(U) + Tl_{process}(U)$$

Aus $T_{local}(U)$ können anhand der Parallelisierungsparameter des UDTO die Komponenten für Partitionierung $T^P_{local}(U)$ und Replizierung $T^R_{local}(U)$ des Operators bei Intra-Operator-Parallelität bestimmt werden. Diese Komponenten werden zur Berechnung der Kosten der verschiedenen Parallelitätsgrade anstatt $T_{local}(U)$ in den Formeln verwendet. Die Kosten der den einzelnen Parallelitätsgrade werden in einem Kostenvektor abgelegt.

6.2.3 Charakterisierende Informationen

Wir wollen hier nochmals die charakterisierenden Informationen zusammenstellen, die sich im Rahmen der Integration des UDTO in die Kostenmodelle von Optimierer und Parallelisierer ergeben haben:

Die Schätzfunktion eines UDTO enthält die Information zur Berechnung seiner Ausgabekardinalität. Dazu gehören auch die optional registrierbaren Schätzfunktionen zur Berechnung der Cucards der Ausgabeattribute.

Die datenflussrelevanten Operatoreigenschaften werden durch die DDL-Klauseln `BLOCKS_ON`, `INPUT_ORDER` und `READS_N_TIMES` angegeben. Sie ermöglichen es unter Hinzunahme der Parallelisierungsparameter, den Datenfluss durch einen UDTO hinreichend genau modellieren zu können.

Die geschätzte Anzahl an CPU-Instruktionen und an I/O-Operationen zum Startup und für die lokalen Arbeitskosten sind unverzichtbare Basisdaten, um die lokalen Ausführungskosten des UDTO berechnen zu können.

6.3 Konkrete Implementierungsschritte

Die Erweiterungen sind bis auf eine Ausnahme bereits weitgehend beschrieben worden. Wir wollen aber der Übersicht halber alle notwendigen Implementierungsschritte an dieser Stelle nochmals auflisten:

1. Im Optimierer ist eine **Schnittstelle** zu schaffen, die den Aufruf der Schätzfunktion durch einen dynamischen Link vornimmt. Sie codiert die von der Funktion benötigten Meta-Daten in einen String, den sie der Funktion als Eingabeparameter übergibt. Zur Berechnung der Cucards ist eine analoge Schnittstelle zu implementieren, die bei

Vorhandensein einer Schätzfunktion diese aufruft und mit den notwendigen Parametern versorgt.

2. Das Urnenmodell des Optimierers zur Berechnung der **Cucards** ist bei propagierten und als Näherung auch bei den anderen Ausgabeattributen eines UDTO einzusetzen.
3. Das **DDL-Statement** `CREATE TABLE_OPERATOR` ist wie beschrieben um die Klauseln für die datenflussrelevanten Operatoreigenschaften zu erweitern. Es sind weiterhin die Klauseln `STARTUP` und `COST_PER_TUPLE` zu implementieren. Auf `STARTUP` folgt ein durch Komma getrennter Integer-Wertepaar, wovon der erste Wert die CPU-Instruktionen und der die zweite die I/O-Zugriffe angibt. Auf `COST_PER_TUPLE` folgen geklammert und durch Kommas getrennt dieselben Wertepaare, wobei die ersten *n* Klammern die Werte für die n Eingaberelationen des UDTO angeben und die *(n+1)*-te Klammer die Werte für die Ausgabetupel enthält.
4. Der **Systemkatalog** ist um mindestens zwei Tabellen zu erweitern, in denen alle einen UDTO betreffenden charakterisierenden Informationen abgelegt werden können. Die erste Tabelle soll Informationen zum Operator selbst, die zweite Informationen zu seinen Ein- und Ausgaberelationen aufnehmen.
5. Im **quasi-parallelen Kostenmodell** sind die durch die Send- und Receive-Knoten verursachten Kommunikationskosten bei der Berechnung zu berücksichtigen. Die Formeln können dem Modell des Parallelisierers entnommen werden. Des weiteren muss das Modell berücksichtigen, dass ein Operatorbaum, der einen UDTO mit *n* Sohn-Knoten enthält, durch diesen bereits in *n+2* Blöcke unterteilt ist.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurden Arbeitsweise und Implementierung von Optimierer und Parallelisierer einschließlich ihrer zugehörigen Kostenmodelle mit dem Ziel untersucht, eine Entwurfsstrategie zur Integration des UDTO in diese beiden Komponenten des Anfrageverarbeitungssystems zu entwickeln. Im Laufe der Studienarbeit wurde deutlich, dass die Integration des UDTO eine sehr aufwendige und vielschichtige Aufgabe darstellt, bei deren Implementierung viele Details zu berücksichtigen sind.

Ergebnis dieser Arbeit ist die in den Kapiteln 4 bis 6 dargestellte Integrationsstrategie, die die Festlegung der charakterisierenden Information für den UDTO enthält. Implementiert wurde die Erweiterung des Parsers für propagierte Attribute des UDTO, und es wurde bezüglich des Optimierers mit der Erweiterung der Konvertierungsroutine `Midas_to_Model_M` und der Implementierung des logischen Knotens UDTO begonnen. Nebenergebnis der Arbeit sind die im Anhang aufgelisteten Informationen zur vollständigen Integration des UDTO in MIDAS sowie die Korrektur der Dokumentation des Systemkatalogs. Zum Schluss soll noch als Vision ein anderer Ansatz zur Kostenabschätzung des UDTO vorgestellt werden:

Eine Alternative zur Kostenabschätzung durch UDSF besteht eventuell darin, ein Analysewerkzeug zu entwickeln, das den Datenfluss eines unbekannten UDTO untersucht und eine Schätzfunktion für ihn berechnet. Es müsste in der Lage sein, funktionale Zusammenhänge zwischen Metadaten der Eingaberelationen und der Ergebnisrelation zu erkennen oder Näherungen dafür zu ermitteln. Der Programmierer wäre dann von der Pflicht entbunden, die Schätzfunktion zu seinem UDTO selbst erstellen zu müssen. Fehler beim Erstellen von Schätzfunktionen, die einem damit nicht vertrauten Programmierer leicht unterlaufen können, wären von vorne herein ausgeschlossen.

Entwurf und Implementierung eines solchen Analysewerkzeugs wird allerdings wegen der Komplexität der Thematik sehr viel Aufwand erfordern, und die Frage ist offen, ob die von ihm erstellten Schätzfunktionen ein ausreichend hohes Maß an Genauigkeit erreichen.

Anhang

Unvollständige und noch fehlende Implementierungen in MIDAS

Die folgende Zusammenstellung an unvollständig vollzogenen und noch nicht durchgeführten Implementierungsschritten erhebt keinen Anspruch auf Vollständigkeit. Sie soll es dem Leser vielmehr ersparen, bestimmte „Entdeckungen“ durch zeitraubendes Code-Lesen selbst machen zu müssen.

- Die Implementierung des in dieser Arbeit entwickelten Entwurfs zur Integration des UDTO ist wie beschrieben für Optimierer, Parallelisierer und deren Kostenmodelle durchzuführen bzw. fertigzustellen.
- Eine SQL-Schnittstelle, die die an den Receive-Knoten ankommenden Daten dem UDTO-Prozess zur Verfügung stellt und analog seine Ergebnistupel an den Send-Knoten weiterleitet, gibt es noch nicht ([5] Kapitel 7.4). Mit dem Aufruf `tbx(GENTREE, dbid, taid, gentree, &qd)` werden zwar die Eingabetabellen durch eine als Gentree vorliegende SQL-Anfrage geöffnet und mit `tbx(EVAL, &qd, Null)` die Ergebnisse der Anfrage tupelweise abgefragt ([4] Kapitel 5.3.3.1); Manipulationen der Tupel und des Ausgabestroms durch den Code sind allerdings nicht möglich. Die TBX-Schnittstelle liefert dem UDTO-Prozess lediglich eine Kopie des betreffenden Tupels, welches sie sofort an den Send-Knoten weiterreicht. (`&qd` ist von der Struktur `Query_descr` und enthält eine Kopie des aktuellen Tupels.
  Das Fehlen dieser Schnittstelle hat zur Folge, dass nur formal prozedurale UDTOs ausgeführt werden können.
- UDFs sind bisher nicht in Optimierer und Parallelisierer integriert worden. Da sie zusammen mit benutzerdefinierten Datentypen und UDTOs ein objektrelationales Gesamtkonzept bilden, sollten sie in Optimierer und Parallelisierer integriert werden.

Ebenso fehlt bisher ein DDL-Statement einschließlich der „ALLOW AS“-Klausel, um UDFs in MIDAS zu registrieren. Zur Registrierung einer UDF muss der Systemkatalog manipuliert werden. UDFs werden allerdings von Compiler und Ausführungssystem verarbeitet.

- Die „ALLOW PARALLEL“-Klausel des DDL-Statements zur Registrierung eines UDTO kann nur die Partitionierungsarten „ANY“ (beliebig) und „–“ (Replikation) verarbeiten; die Arten „EQUAL“ und „RANGE“ sind nicht implementiert.
- Die Joinoptimierungsphase gehört in den Optimierer; sie ist aus dem Parallelisierer zu entfernen und in den Optimierer zu integrieren.

Korrektur bezüglich des Systemkatalogs

Die Tabelle SYSFUNCIMP stimmt nicht mehr mit der in [6] angegebenen Dokumentation überein. Sie ist folgendermaßen aufgebaut:

fname	tname	tsegno	typ	partdesc
STRING	STRING	INTEGER	INTEGER	STRING

Tabelle SYSFUNCIMP

In SYSFUNCIMP werden Informationen zu den Ein- und Ausgabetabellen und die Parallelisierungsparameter eines UDTO abgelegt. Die Attribute haben folgenden Inhalt:

fname → Namen des UDTO
tname → formaler Name der Ein- oder Ausgabetabelle des UDTO
tsegno → negative Segmentnummer der formalen Tabelle aus tname
typ → gibt an, ob die Tabelle zur Parallelisierung des UDTO beliebig partitioniert (Eintrag = 0) oder repliziert (Eintrag = 1) wird.
partdesc → gibt an, ob das Tupel eine Eingabetabelle (INPUT TABLE), eine Ausgabetabelle (OUTPUT TABLE) oder eine Parallelisierungsinformation (leerer String) enthält.

Literaturverzeichnis

[1] Fleischhauer, Michael: Parallelisierung relationaler Datenbankanfragen in MIDAS. Diplomarbeit, Technische Universität München, Institut für Informatik, 1997.

[2] Krüger-Barvels, Kay: Entwicklung eines regelbasierten Anfrageoptimierers für das parallele, objektrelationale Datenbanksystem MIDAS. Diplomarbeit, Technische Universität München, Institut für Informatik, 1998.

[3] Hilbig, Matthias: Entwicklung eines kostenbasierten Anfrageoptimierers für das parallele, relationale Datenbanksystem MIDAS. Diplomarbeit, Technische Universität München, Institut für Informatik, 1998.

[4] Heupel, Sebastian: Implementierung erweiterbarer Datenbankoperatoren in einem parallelen, objektrelationalen Datenbanksystem. Diplomarbeit, Technische Universität München, Institut für Informatik, 1998.

[5] Jaedicke, Michael: New Concepts for Parallel Object-Relational Query Processing. Doktorarbeit, Universität Stuttgart, IPVR, 1999.

[6] Perathoner, Sabine: Erweiterung des System-Kataloges von MIDAS um Statistiken und objekt-relationale Funktionalität. Fortgeschrittenenpraktikum, Technische Universität München, Institut für Informatik, 1998.

[7] Brandmayer, Franz: Parallele Anfrageausführung in MIDAS. Diplomarbeit, Technische Universität München, Institut für Informatik, 1997.

Erklärung zur Studienarbeit

Hiermit erkläre ich, dass ich diese Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Stuttgart, den 30. Juni 2000

(Bernd Watzal)