INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Parallele Anfrageverarbeitung in multidimensionalen Array Datenbanksystemen

Karl Hahn

Institut für Informatik
der Technischen Universität München

Parallele Anfrageverarbeitung in multidimensionalen Array Datenbanksystemen

Karl Hahn

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Christoph Zenger
Prüfer der Dissertation:	1.	Univ.-Prof. Rudolf Bayer, Ph.D., em.
	2.	Univ.-Prof. Dr. Klaus R. Dittrich, Universität Zürich / Schweiz

Die Dissertation wurde am 27.01.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.05.2005 angenommen.

Für meine Frau Tanja,
Alissa (meine Schnecke)
und Jannik (meinen Wusler)

Inhalt

Kapitel 1 Einführung 1

1.1 Motivation 1
1.2 Multidimensionale Array Daten: ein erster Eindruck 3
1.3 Datenbanksysteme und Parallelität 7
1.4 ESTEDI: Fusion von Datenbanktechnologie und High Performance Computing 8
1.5 Ziele und Anforderungen 13
1.6 Überblick 14

Kapitel 2 Datenmodell und Terminologie 17

2.1 Multidimensionale Array Daten: ein einführendes Beispiel 18
2.2 Logisches Datenmodell 21
2.2.1 Trägermenge: Multidimensionale Objekte 21
2.2.2 Operationen auf multidimensionalen Daten 23
2.3 Einbettung in das relationale Modell 29
2.3.1 Der Datentyp MDD für multidimensionale Arrays 29
2.3.2 Einschränkungen und Probleme 31
2.4 Physikalisches Datenmodell 32
2.5 Das Array DBMS rasdaman 35
2.5.1 Architektur 35
2.5.2 Anfragesprache, Anfragebaum, Anfrageverarbeitung 36
2.6 Zusammenfassung 39

Kapitel 3 Parallelität in relationalen Datenbanksystemen 41

3.1 Skalierbarkeit und deren Grenzen 42
3.2 Parallele Architekturen 45
3.3 Arten der Parallelität 48
3.3.1 E/A Parallelität und Verarbeitungsparallelität 48
3.3.2 Datenparallelität und Pipeline-Parallelität 49
3.3.3 Granularität der parallelen Verarbeitung 50
3.4 Parallele Ausführung einer Anfrage 52
3.5 Zusammenfassung 55

Kapitel 4 Parallele Verarbeitung von Array Daten in einem DBMS 57

4.1 Besonderheiten gegenüber paralleler Verarbeitung von relationalen Daten 59
4.1.1 Komplexität der Daten und der Datenübertragung 60
4.1.2 Heterogenität der Operatorbäume 62
4.1.3 Granularität der Datenverarbeitung und des Datenflusses 64
4.2 Inter-Operator Parallelisierung 67
4.2.1 Pipeline Parallelität (Vertikale Inter-Operator Parallelisierung) 67
4.2.2 Bushy-Tree Parallelisierung (Horizontale Inter-Operator Parallelisierung) 71
4.3 Intra-Operator Parallelisierung (Datenparallelität) 73
4.3.1 Analyse der Iteratoren 75
4.3.2 Fusion von Iteratoren zu parallelen Blöcken 79
4.3.3 Adaption des Operatorbaumes 97
4.3.4 Integration in die Anfrageausführung 101
4.3.5 Analyse bezüglich Anfälligkeit für Skew 104
4.4 Intra-Objekt Parallelisierung 107
4.4.1 Analyse der Operatoren 109
4.4.2 Parallele Ausführung von Verknüpfungen von Operatoren 115
4.4.3 Adaption des Operatorbaumes 117
4.4.4 Partitionierungsstrategien 122
4.4.5 Fusion von Zwischenergebnissen 138
4.4.6 Analyse bezüglich Anfälligkeit für Skew 144
4.5 Wahl der Parallelisierungsstrategie 146
4.5.1 Analyse der parallelen Effizienz 147
4.5.2 Algorithmus zur Wahl der Parallelisierungsmethode 148
4.5.3 Kombination der Parallelisierungsstrategien 149
4.6 Zusammenfassung 151

Kapitel 5 Implementierung: rasdaman 5.0 PE 155

5.1 Prämissen der Implementierung 155
5.2 Designentscheidungen 157
5.2.1 Adaptionsstrategie 157
5.2.2 Parallele Zielplattformen und Kommunikation 158
5.3 Architektur des parallelen Array DBMS rasdaman 5.0 PE 159
5.3.1 Anpassung der Module für rasdaman Server Prozesse 160
5.3.2 Kommunikation zwischen parallelen Prozessen 162
5.4 Zusammenfassung 163

Kapitel 6 Performanz der Implementierung 165

6.1 Beschreibung der Testdaten 165
6.1.1 Satellitenbilder (2D) 166
6.1.2 Simulation der globalen Erderwärmung (3D) 167
6.1.3 Simulation von Windverhältnissen nach einem Klimamodell (4D) 168
6.2 Theoretische Analyse der Kosten einer Parallelverarbeitung 170
6.3 Multiprozessor-Architekturen 172
6.3.1 Messumgebung 172
6.3.2 Inter-Objekt Parallelisierung 173
6.3.3 Intra-Objekt Parallelisierung 179
6.3.4 Zusammenfassung der Ergebnisse für Shared-Everything Architekturen 181
6.4 Rechnercluster 181
6.4.1 Messumgebung 182
6.4.2 Inter-Objekt Parallelisierung 183

6.4.3 Intra-Objekt Parallelisierung 186
6.4.4 Zusammenfassung der Ergebnisse für Shared-Disk Architekturen 188
6.5 Zusammenfassung und Bewertung der parallelen Architekturen 188
Kapitel 7 Zusammenfassung und Ausblick 191

Literaturverzeichnis 195
Appendix A Notation 201
Appendix B Abbildungsverzeichnis 203
Appendix C Verzeichnis der Definitionen 205
Appendix D Verzeichnis der Beispiele 206

Kapitel 1 Einführung

Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?
(T.S. Eliot)

1.1 Motivation

Speicherung und Analyse von wissenschaftlichen Daten hat die letzten Jahrzehnte ungemein an Bedeutung gewonnen. Diese aus Experimenten, Simulationen oder Beobachtungen gewonnenen Daten werden meist als multidimensionale Rasterdaten (auch: multidimensionale Array Daten) gespeichert. Beispiele reichen von 2-dimensionalen Satellitenbildern über 3-dimensionale, also räumliche Daten als Ergebnis eines Experimentes (etwa für Tests neuer Automobilmodelle), bis zu 4-dimensionalen Daten, die einer Klimasimulation zur Prognose von zukünftigen Wetterentwicklungen entspringen können. Tatsächlich können die Daten beliebige Dimensionalität aufzeigen, wobei (aufgrund menschlicher Präferenzen) in der Praxis selten mehr als 4 Dimensionen für Rasterdaten gewählt werden. Die Inhalte der Datensätze sind mannigfaltig: Farbwerte (etwa als RGB codiert), Temperaturwerte, Druckverteilung von Gasen und Geschwindigkeitswerte sind nur einige Beispiele.

In den letzten Jahren lässt sich der Trend erkennen, dass immense Datenvolumina dieser Rasterdaten gesammelt werden, sei es durch Experimente, Computer-gestützte Simulationen oder Observierung. Beispielsweise soll im Rahmen des EOS (engl.: Earth Observing System, Erdbeobachtungssystem) Projektes der NASA (engl.: National Aeronautics and Space Administration, Raumfahrtbehörde der USA) pro Tag ungefähr ein Terabyte, also 10^{12} Byte, an Daten gesammelt werden [EOS03]. Die Speicherung dieser gewaltigen Datenmengen liefert die erste Herausforderung, die am besten durch Datenbanksysteme und Tertiärspeicher-Archive bewältigt werden kann. Institutionen wie die NASA, das DLR (Deutsches Luft- und Raumfahrtzentrum), CERN (franz.: Conseil Europeen pour la Recherche Nucleaire, Europäisches Zentrum für Nuklearforschung), etc. betreiben einen beträchtlichen Aufwand, die gewonnenen Daten zu sichern. Das Deutsche Klimarechenzentrum des Max-Planck Instituts für Meteorologie etwa, einer der Partner im Projekt ESTEDI (siehe Kapitel 1.4), unterhält einen Daten-

server mit aktuell etwas mehr als 3400 Terabyte Kapazität, um die in Simulationen gewonnenen Daten sicher archivieren zu können.

Eine weitaus größere Herausforderung neben der Speicherung riesiger Datenmengen tritt jedoch aktuell immer deutlicher zutage: die spätere Analyse der gewonnenen multidimensionalen Rasterdaten:

- Datenanalyse „ad hoc“ ist wegen der gewaltigen Größe der Quelldaten und der Komplexität der auszuführenden Operationen praktisch unmöglich. Jede Analyse bedarf somit einer langen Vorbereitungs- und Ausführungszeit. Das Sammeln von Indizien, also das schnelle „Probieren“, welches wissenschaftlich zur Vorbereitung von Thesen und Schlüssen dient, wird somit unterbunden.
- Analysen sind wertlos, da die Ergebnisse zu spät vorliegen. Auf den Punkt gebracht: was nützt eine Wettervorhersage für morgen, die übermorgen vorliegt?
- Eine umfassende Analyse durch systematisches Probieren aller Möglichkeiten, neudeutsch häufig als „brute force attack“ bezeichnet, ist durch die Anzahl der Möglichkeiten, verbunden mit den mangelnden Berechnungskapazitäten, nicht möglich. Diese Methode wird zwar wissenschaftlich häufig belächelt, ist jedoch seit Jahrzehnten erfolgreich und wird auch aktuell für komplexe Probleme eingesetzt. Beispiele sind das „Knacken“ der codierten deutschen Kommunikation im zweiten Weltkrieg, der Versuch, außerirdische Signale im kosmischen Rauschen zu identifizieren (Projekt SETI, engl.: Search for Extraterrestrial Intelligence, Suche nach außerirdischer Intelligenz [SETI]) oder die komplette Entschlüsselung der menschlichen Gensequenz.

Genau betrachtet besteht das Problem in der inadäquaten Performanz der Datenanalyse. Diese kann theoretisch mit Qualität oder Quantität angegangen werden: die Qualität der Datenanalyse wird durch Verbesserung von Datenmodell, Anfragesprache und einer Optimierung der Anfrage erreicht. Die Praxis hat jedoch gezeigt, dass diese Anstrengungen nicht ausreichen. In der Regel ist Quantität nötig, also „das Auffahren schwerer Geschütze“. Das heißt, die Analyse wird durch den Einsatz komplexer und oftmals teurer paralleler Rechnerarchitekturen beschleunigt. Parallele Verarbeitung erfordert jedoch eine Adaption der Verarbeitung und damit der Analysealgorithmen. Die zentrale Herausforderung, die durch vorliegende Arbeit angegangen und gelöst wird, ist:

> Wie kann die Analyse von multidimensionalen Rasterdaten effizient parallelisiert werden, um gängige parallele Architekturen optimal auszunutzen? Welche Konzepte und Algorithmen unterstützen typische Daten und Anfragen optimal und vermeiden pathologische Fälle für typische Szenarien?

Die Qualität der vorgestellten theoretischen Konzepte und der Implementierung wird durch Performanzanalysen bestätigt, die unter typischen Voraussetzungen eine zum Parallelitätsgrad fast lineare Verbesserung der Anfragezeiten zeigen. Die Messungen erfolgten auf Mehrprozessorrechnern sowie auf Rechnerclustern und zogen vielfache Randbedingungen in Betracht, etwa Datenkompression und den Einfluss von verbesserter Datenanalyse, die im Rahmen des Projekts ESTEDI (siehe 1.4) von den Anwendern gefordert und bei FORWISS entwickelt wurden.

1.2 Multidimensionale Array Daten: ein erster Eindruck

Anmerkungen zu wissenschaftlichen Veröffentlichungen, die im Verlauf der Bearbeitung dieses Dissertationsthemas entstanden, sowie Diskussionen mit Kollegen auf Konferenzen haben immer wieder gezeigt, dass das primäre Missverständnis bezügliche der hier vorgestellten Arbeit die genaue Art der Daten ist. Um einer falschen Einordnung dieser Arbeit entgegenzuwirken, werden die Daten sowie Analysen darauf kurz beispielhaft vorgestellt und gegen verwandte Bereiche der Speicherung und Analyse von multidimensionalen Daten abgegrenzt. Eine formale Definition folgt in Kapitel 2.

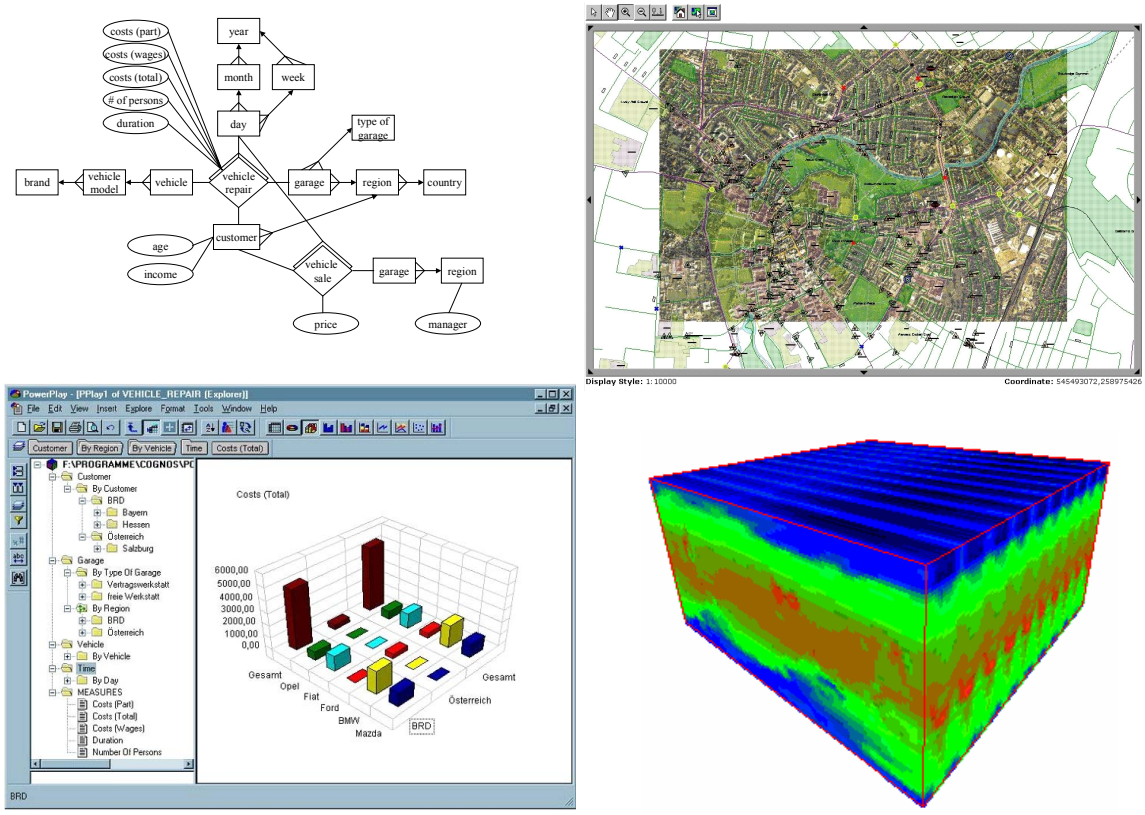


Abbildung 1.1: Beispiel für Applikationen aus den Bereichen Data Warehouse, Online Analytical Processing, Geografische Informationssysteme, Array DBMS

Auf dem Gebiet der Datenbanksysteme ist der Begriff „multidimensionale Daten“ mehrfach belegt. Im Zusammenhang mit DBMS wird der Begriff meist mit Data Warehouse Lösungen oder Geografischen Informationssystemen assoziiert. In Abbildung 1.1, links sieht man eine grafische Repräsentation eines Data Warehouse Schema. Die verwendete grafische Notation basiert auf dem Multidimensional Entity Relationship Model MER [SBHD98], einer Erweiterung des Entity Relationship Modells von Chen [Che75] und zeigt Fakten, Kennzahlen, Dimensionen und deren hierarchischen Aufbau. Darunter sieht man ein Beispiel für ein OLAP (engl.: Online Analytical Processing) Frontend, welches für die (grafische) Repräsentation der Warehouse Daten verantwortlich ist. Auf der rechten Seite oben ist ein Beispiel für eine (2-dimensionale) GIS (engl.: Geographic Information System(s), Geografische Informationssystem(e)) Anwendung dargestellt, darunter ein 3D Würfel, der eine Visualisierung von Rasterdaten (Arraydaten) zeigt.

Von den drei ersten Bereichen wollen wir uns differenzieren. Die vorliegende Arbeit geht von Array- bzw. Rasterdaten aus, die im Gegensatz zu diesen folgende Eigenheiten aufweisen:

- Array Daten werden im Gegensatz zu Data Warehouse Daten komplett in linearisierter Form und nicht in vektorisierter Form gespeichert. Mit anderen Worten sind Rasterdaten komplett besetzte und als Array gespeicherte Datenräume. Ein multidimensionaler Data Warehouse Datenwürfel ist ein extrem dünn besetzter Raum und wird meist in relationalen DBMS als eine Menge von Tupel (nur Datenpunkte ungleich null) bestehend aus Koordinatenvektor und Wert(en) gespeichert. OLAP Werkzeuge stellen ein Frontend für die visuelle Aufbereitung solcher Warehouse Daten dar. Operationen auf diesen Daten lassen sich einteilen in Einschränkungen in einer oder mehreren Dimensionen, Gruppierungen (zur Analyse von hierarchischen Dimensionen) und Aggregationen. Wie später gezeigt wird, sind diese Operationen auch für multidimensionale Rasterdaten relevant.
- GIS Datenbanken speichern 2D Daten, Array Daten sind in ihrer Dimensionalität nicht prinzipiell beschränkt. Ferner enthält die Speicherung von Arrays keine Semantik per se, während bei GIS die Semantik der geometrischen Objekte für die Speicherung genutzt wird (etwa Polygon, Linie, etc.).

Natürlich gibt es eine Reihe weiterer Unterschiede bezüglich des Datenmodells, der physikalischen Speicherung, typischer Indices, Anfragesprache, etc, auf die hier nicht detailliert eingegangen werden soll. Eine formale Definition von Array Daten, genauer des in dieser Arbeit verwendeten Datenmodells für Array Daten, folgt in Kapitel 2.

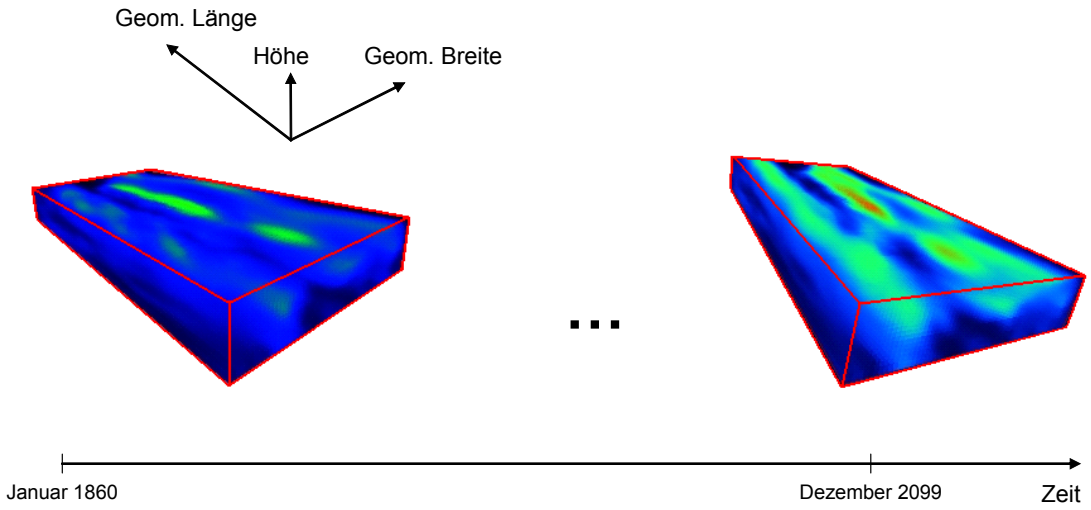


Abbildung 1.2: 4D Rasterdaten, Simulation eines Klimamodells: Windgeschwindigkeiten in äquatorialer Richtung von Januar 1860 bis Dezember 2099

Wir wollen mit Hilfe von Abbildung 1.2 einen ersten Eindruck geben, wie eine Analyse von multidimensionalen Rasterdaten aussehen kann und weshalb diese Analyse komplex ist. Die Abbildung skizziert Teile eines 4D Objekts, das das Resultat einer Klimasimulation des Max-Planck Instituts für Meteorologie beinhaltet. In dieser Simulation wurde die Entwicklung von Temperaturen und Windgeschwindigkeiten berechnet, unter der Annahme, dass die Voraussagen des (mittlerweile als zu positiv beurteilten) Kopenhagen-Protokolls von 1996 bezüglich

Emission von Treibhausgasen zutreffen[1]. Die Zellwerte der in Abbildung 1.2 visualisierten Daten entsprechen den Windgeschwindigkeiten in äquatorialer Richtung. Die ersten drei Dimensionen sind geometrische Länge, Breite und Höhenstufen (bezüglich des atmosphärischen Drucks). Die vierte Dimension beschreibt eine Zeitspanne von Januar 1860 bis Dezember 2099 in der Granularität von Monaten. Die vierte Dimension ist hier nur exemplarisch durch das erste und das letzte Element skizziert. Die Farbverteilung spiegelt die Geschwindigkeitswerte wider, von blau (bzw. dunkel) für geringe Geschwindigkeiten über grün bis hin zu rötlichen (bzw. hellen) Bereichen für hohe Geschwindigkeiten. Die Grafik zeigt hohe Windgeschwindigkeiten jeweils nördlich und südlich des Äquators in großer Höhe, diese werden in der Meteorologie „Jet Streams" genannt. Die Grafik lässt ferner eine deutliche Zunahme der Geschwindigkeiten im Jahre 2099 gegenüber 1860 erkennen. Typische Analysen auf diesen Simulationsdaten können etwa derart aussehen:

1. Zeige mir die Windgeschwindigkeiten des ersten und letzten Monats der Simulation als JPEG Bild! (Das Ergebnis entspricht Abbildung 1.2, der Datentyp JPEG erlaubt die Anzeige in einem WWW Browser.)
2. Um wie viel Prozent erhöht sich die Durchschnittsgeschwindigkeit von Januar 2000 bis Januar 2040?
3. In welchen Höhenstufen sind die Differenzen der Geschwindigkeiten bezüglich obiger Zeitpunkte am höchsten?
4. In welchen Bereichen der Erde werden 2085 die höchsten Geschwindigkeiten erwartet?
5. Berechne die Differenzen zu heute für die Bereiche aus Anfrage 4!
6. In welchen Gebieten treten insgesamt 2020 gegenüber dem Jahr 2004 die größten Differenzen auf?
7. Ab welchen Jahren übersteigt die durchschnittliche Windgeschwindigkeit in Mitteleuropa auf Meereshöhe einen kritischen Bereich von 30 km/h?
8. Gab es die letzten 50 Jahre bereits markante Veränderungen in Deutschland?

Am Beispiel dieser Daten und Anfragen lässt sich bereits eine Reihe von Herausforderungen identifizieren, die in den letzten Jahren und aktuell Gegenstand der Forschung sind:

- Eine umfassende flexible Analyse multidimensionaler Rasterdaten erfordert eine Anfragesprache, die eine möglichst große Zahl von speziell für diese Daten definierten Operationen unterstützt. Dies ermöglicht die dynamische Analyse durch Spezialisten, die über automatisierte Verfahren hinausgeht. Ferner benötigen Anfragen auf multidimensionale Rasterdaten für die effiziente Umsetzung eine speziell für diese Daten entwickelte Anfrageoptimierung. [Rit99]
- Teilbereiche der Daten sollen innerhalb dieser Anfragesprache spezifiziert werden können. Für Berechnungen werden somit nur die wirklich benötigen Daten geladen. Das effiziente Lokalisieren und der Zugriff auf die Daten muss mit speziellen multidimensionalen Indices und Zugriffsstrukturen unterstützt werden. Diese Forderung ist vor allem eine Voraussetzung für nicht-lokale Analyse der Daten. [Fur99]
- Die Größe der Daten legt eine Kompression sowohl der Datenbasis als auch der zwischen Server und Anwendung zu übertragenden Daten nahe. Berechnungen auf den Daten benötigen zuvor eine Dekompression. Diese Mechanismen sollen effizient und vom Benutzer verborgen stattfinden. [Deh02]
- Die Menge der Daten macht es nötig, dass sie, teilweise nach einer Analyse, teilweise aber auch ohne vorherige Analyse, auf Tertiärspeicher abgelegt werden. Eine nachträgli-

[1] Die Daten von 1860 bis 1990 sind real gemessene Werte, die Werte von 1990 bis 2099 sind entsprechend der Voraussagen des Kopenhagen-Protokolls auf dieser Basis simuliert.

che Analyse solcher Daten ist in den meisten Fällen nur mit beträchtlichem Aufwand möglich. Ein selektives Laden der Datensätze von Tertiärspeicher bezüglich der durch die Anfrage benötigten Teilbereiche löst dieses Problem. [Rei05]

- Trotz selektivem Laden und einer Anfragesprache für die serverseitige Ausführung komplexer Berechnungen benötigen viele Analysen enorme Ressourcen bezüglich E/A, Speicher, aber insbesondere CPU. Der bei der Datenanalyse entstehende Engpass bezüglich Rechenleistung kann mittels Verteilung der Rechenlast auf mehrere Prozessoren bzw. mehrere Rechner entschärft werden. Die Antwortzeiten für Anfragen auf Rasterdaten vermindern sich durch Realisierung von Berechnungsparallelität enorm.

Der Inhalt dieser Arbeit ist der letzte und wichtigste Punkt der vorgestellten Liste, die parallele Verarbeitung von Anfragen an Rasterdaten. Die Motivation für diesen Forschungsschwerpunkt war die Beobachtung, dass trotz aller Anstrengungen bezüglich Anfragesprache, Anfrageoptimierung, Zugriffsoptimierung, etc., die Dauer von Anfragen zum Teil zu hoch war. Beim Einsatz des Array DBMS **rasdaman** war für typische Einsatzgebiete und Analysen eine Anfragezeit von Stunden und teilweise sogar Tagen keine Seltenheit. Neben genervten Analysten war damit auch der Wert des Systems in Frage gestellt, da ad-hoc Analysen nicht möglich waren und verspätete Ergebnisse wertlos wurden. Auf den Punkt gebracht: als Lösung für die Speicherung von und den Zugriff auf multidimensionale Rasterdaten hatte sich **rasdaman** etabliert, der Einsatz als Analysewerkzeug war zumindest für wichtige Einsatzgebiete fraglich.

Sehr früh zeichnete sich ab, dass diese langen Antwortzeiten nicht etwa durch E/A, also das Laden der Daten von Sekundärspeicher, oder durch das Übertragen über Netzwerk zustande kamen, sondern meist durch Berechnungen auf den Rasterdaten. Im Gegensatz zu einer Verbesserung der Performanz von E/A und des Netzwerks, die in der Praxis ohne Anpassung des Programms durch Einsatz neuer Hardware realisiert werden kann, zeigt die Skalierung bezüglich Berechnungsressourcen sehr viel deutlicher Schranken. Ab einem gewissen Bedarf an diesen Ressourcen muss das Programm adaptiert werden, um eine parallele Ausführung zu ermöglichen.

Verschärfend zu diesen extremen Antwortzeiten der Anfragen kam hinzu, dass auf der anderen Seite Datenlieferanten gegenüber standen, die typischerweise massiv parallele Rechnerarchitekturen nutzen. So generiert etwa das Deutsche Klimarechenzentrum des Max-Planck Institut für Meteorologie, einer der Partner im Projekt ESTEDI (siehe Kapitel 1.4), Daten für Simulationen von Klimaentwicklungen mit Hilfe ihres Höchstleistungsrechnersystems für Erdsystemforschung namens HLRE, einem Server basierend auf 24 NEC-SX-6 mit insgesamt 192 CPUs, 1,5 Terabyte Hauptspeicher und 1,5 Teraflops Rechenkapazität. Die HPC (engl.: High Performance Computing) Zentren, welche die aus Simulationen, Beobachtungen oder Versuchen gewonnenen Daten kommerziell anbieten, sind sehr daran interessiert, ihren Kunden die gewünschten Daten und Analysen mit der geforderten Performanz zu bieten und nutzen hierzu teure parallele Architekturen.

Diese erste Vorstellung von multidimensionalen Rasterdaten und die Historie des ersten kommerziellen Array DBMS **rasdaman** für die Speicherung und die Analyse von Rasterdaten beliebiger Dimensionalität und Ausdehnung verdeutlicht die Motivation für diese Dissertation: eine effiziente Analyse dieser Daten bedarf der parallelen Anfrageverarbeitung.

1.3 Datenbanksysteme und Parallelität

Parallele Anfrageverarbeitung in einem DBMS für Rasterdaten stützt sich selbstverständlich immer auf das „große Vorbild“, der parallelen Verarbeitung in relationalen DBMS. Parallele Anfrageverarbeitung in RDBMS (engl.: Relational Database Management System, Relationales Datenbanksystem) wurde in den letzten 2 Jahrzehnten eingehend wissenschaftlich durchleuchtet. Beginnend mit ersten Publikationen und wissenschaftlichen Prototypen von parallelen Datenbankmanagementsystemen in den 80ern, gefolgt von immer besseren und spezielleren parallelen Algorithmen für relationale Operatoren in den 90, werben heute alle großen Datenbankhersteller mit Parallelität als Verkaufsargument für ihr Produkt. Die drei Marktführer Oracle, IBM und Microsoft unterstützen verschiedene parallele Architekturen wie Multiprozessorrechner und Rechnercluster, evtl. mit gemeinsamem Sekundärspeicher, und nutzen intern verschiedene Algorithmen zur parallelen Ausführung, etwa Datenpartitionierung und Pipeline Parallelität. All diese Konzepte wurden jedoch bisher nicht auf ihre Eignung bezüglich extrem großer multidimensionaler Rasterdaten analysiert. Diese Lücke schließt die vorliegende Arbeit:

- Relationale Konzepte (parallele Architekturen, parallele Algorithmen) sowie bekannte Probleme der Parallelverarbeitung wurden eingehend untersucht. Geeignete Methoden wurden für die parallele Analyse von multidimensionalen Rasterdaten angewandt und entsprechend der Besonderheiten der Daten und deren Verarbeitung adaptiert.
- Erfahrungen bezüglich Grenzen und Einschränkungen bekannter Parallelisierungsstrategien, die sich im Projekt ESTEDI bei der Analyse von multidimensionalen Rasterdaten zeigten, führten zu eigens entwickelten Algorithmen, die speziell auf große Rasterdaten abgestimmt sind und bei relationalen Systemen bisher keine Rolle spielen.

Das Ziel einer Parallelausführung von Anfragen an ein DBMS ist primär die Steigerung der Performanz einer Menge von Anfragen (engl.: inter-query) oder einzelner Anfragen (engl.: intra-query). Als Kriterium für den erzielten Performanzgewinn dient meist der so genannte Speed-Up (engl.: speed up, beschleunigen), die Verringerung der Ausführungszeit einer Anfrage, also der Faktor, um den sich die Anfragezeit bei Parallelausführung gegenüber einer Ausführung auf dem entsprechenden nicht-parallelen System reduziert. Ein ähnliches Maß für eine Performanzsteigerung ist das Scale-Up (engl.: scale up, vergrößern), die Skalierbarkeit der Datenmenge bei Parallelausführung mit identischer Ausführungszeit. Als Optimum paralleler Effizienz gilt ein linearer Speed-Up, das heißt bei einem Parallelitätsgrad von p (abhängig von der verwendeten Hardware) erfolgt die Anfrage p mal schneller. Ein optimaler (linearer) Scale-Up hat einen konstanten Wert 1, das heißt, bei Parallelitätsgrad p kann eine um den Faktor p größere Datenmenge in identischer Zeit verarbeitet werden. Neben Performanzsteigerung sind wichtige Ziele der Parallelität die Verfügbarkeit (engl.: high-availability), also die Ausfallsicherheit von DBMS Diensten, und die Erweiterbarkeit (engl.: extensibility), also eine einfache Verbesserung der Performanz durch Erweiterung der Hardware (statt Ersatz der Hardware). In unserer Arbeit konzentrieren wir uns vor allem auf das Primärziel, also die Steigerung der Performanz. Verfügbarkeit wird meist durch Replikation (von Daten aber auch Serverdiensten) sichergestellt, Erweiterbarkeit meist durch den Einsatz von entsprechender Hardware (Rechnercluster, Beowulf-Architekturen, RAID, etc.; siehe Kapitel 3).

Bekannte Probleme, die einem optimalen linearen Speed-Up entgegenstehen, sind vor allem Kosten für das Initialisieren der parallelen Ausführung (engl.: startup costs), für die Kommunikation zwischen den parallelen Instanzen (engl.: interference costs) und Lastprobleme (engl.: skew), die durch eine ungleiche Verteilung der Arbeitslast auf parallele Prozesse entstehen. Letztere wurden eingehend wissenschaftlich untersucht und werden vor allem durch

dynamische Datenverteilung umgangen. Kommunikationskosten hingegen sind weiterhin das primäre Hindernis bezüglich Skalierbarkeit (etwa bei einem Parallelitätsgrad von 10 und mehr). Bei großen Rasterdaten zeigt sich, dass vor allem Kommunikationskosten ein Problem darstellen. Die Größe der Zwischenergebnisse sowie deren Komplexität führen dazu, dass ein Transfer extrem kostenintensiv ist und folglich möglichst vermieden werden muss. Probleme der Lastverteilung werden einerseits durch dynamische Strategien, andererseits durch verschiede Granularitäten der Datenpartitionierung vermieden.

Bereits 1990 prophezeiten David DeWitt und Jim Gray in einer der am meisten zitierten Publikationen über parallele DBMS [DG90] eine Ära der Shared-Nothing Systeme, also lose gekoppelter Rechnercluster (eine detaillierte Beschreibung der verschiedenen parallelen Architekturen folgt in Kapitel 3). Bessere Erweiterbarkeit und vor allem ein deutlicher Vorteil bezüglich Kosten von Hardware sollten zu einer Ablösung von Multiprozessor-Architekturen (Shared-Everything) führen. Rückblickend kann man feststellen, dass dies so nicht eingetreten ist. Nach wie vor basieren die leistungsfähigsten parallelen DBMS zumeist auf Multiprozessor-Architekturen (diese Beobachtung basiert auf Wertungen des Transaction Processing Performance Council [TPC]). Auch die Bewertung bezüglich Preis-/Leistungsverhältnis wird hier widererwartend nicht durch Rechnercluster dominiert. Allerdings beginnt sich aktuell abzuzeichnen, dass Shared-Disk Systeme, also Rechnercluster mit einer gemeinsam genutzten Datenbasis, aufgrund schneller Netzwerke wie zum Beispiel InfiniBand [Inf] an Bedeutung gewinnen [Ora03]. Bezüglich unserer Arbeit wollen wir keinerlei Restriktionen bezüglich der parallelen Architektur zulassen. Durch den Einsatz des Kommunikationsstandards Message Passing Interface MPI-2 [MPI] können alle drei angesprochenen Architekturen genutzt werden.

Die in dieser Arbeit vorgestellten Konzepte und Algorithmen wurden im Rahmen des von der Europäischen Union geförderten Projektes ESTEDI (5. Rahmenprogramm ISF 1999-11009) praktisch implementiert und in ein kommerzielles Array DBMS integriert. Das von der Firma rasdaman GmbH vertriebene Array DBMS **rasdaman** wurde bezüglich paralleler Anfrageverarbeitung erweitert. Praktische Erfahrungen aus diesem Projekt, insbesondere Anregungen bzw. Anforderungen der Anwender bei der Evaluierung des parallelen Array DBMS, sind in die Weiterentwicklung der ursprünglichen Konzepte eingeflossen und haben diese Arbeit maßgeblich beeinflusst. Ferner sichert erst eine Implementierung die Machbarkeit der entwickelten theoretischen Konzepte. Überlegungen, die in Theorie keine Rolle spielen, haben sich in der Praxis als ernsthafte Herausforderung gezeigt, beispielsweise das Linearisieren von multidimensionalen Arrays vor dem Versenden an Prozesse und das Restaurieren dieser Objekte, sowie die Fehlerbehandlung, falls Übertragungsfehler auftreten.

1.4 ESTEDI: Fusion von Datenbanktechnologie und High Performance Computing

ESTEDI (engl.: European Spatio-Temporal Data Infrastructure for High-Performance Computing), eine Initiative von Softwareherstellern sowie diverser Forschungs- und Rechenzentren, wurde von 1999 bis 2003 von der Europäischen Union als Projekt gefördert (IST-1999-11009). Die Motivation für das Projekt leitete sich aus folgenden Beobachtungen ab:

- multidimensionale Rasterdaten (engl.: multidimensional data, spatio-temporal data) werden von diversen europäischen Rechenzentren (HPC) in großem Umfang erstellt bzw. gesammelt und als Dateien archiviert. Als Hindernis für eine effiziente Analyse stellte sich schon früh heraus, dass relevante Daten bzw. in den Daten inhärente Informationen nicht

effizient genug extrahiert werden konnten. So waren multidimensionale Bereichsanfragen sowie die Analyse von Daten einer entfernten Datenquelle (ohne Transfer der kompletten Daten) nicht möglich. Dieses Problem sollte durch den **Einsatz eines für multidimensionale Rasterdaten optimierten DBMS** namens rasdaman gelöst werden. Ferner sollte dieses DBMS an die Anforderungen der Rechenzentren angepasst werden, insbesondere bezüglich Nutzung paralleler Architekturen, Anbindung an Tertiärspeichersysteme, erweiterte Analysemöglichkeiten, Mehrbenutzerbetrieb und benutzerdefinierte Funktionen (UDF, engl.: User Defined Functions).

- es existiert eine Vielzahl von Formaten für multidimensionale Rasterdaten (CERA, HDF, CGNS, etc.). Diese Heterogenität der Datenformate war nicht nur zwischen verschiedenen Fachgebieten, sondern auch innerhalb dieser zu beobachten. Diese Formatvielfalt führte dazu, dass eine Zusammenarbeit von fachverwandten Institutionen oftmals kaum möglich war. Ziel von ESTEDI war es daher sekundär, einen **Standard für die Speicherung und den Austausch von Rasterdaten beliebiger Dimensionalität** zu entwickeln. Zu diesem Zweck sollten ein generelles Metadatenformat sowie auf die speziellen Applikationsszenarien abgestimmte optionale Metadaten bestimmt werden.

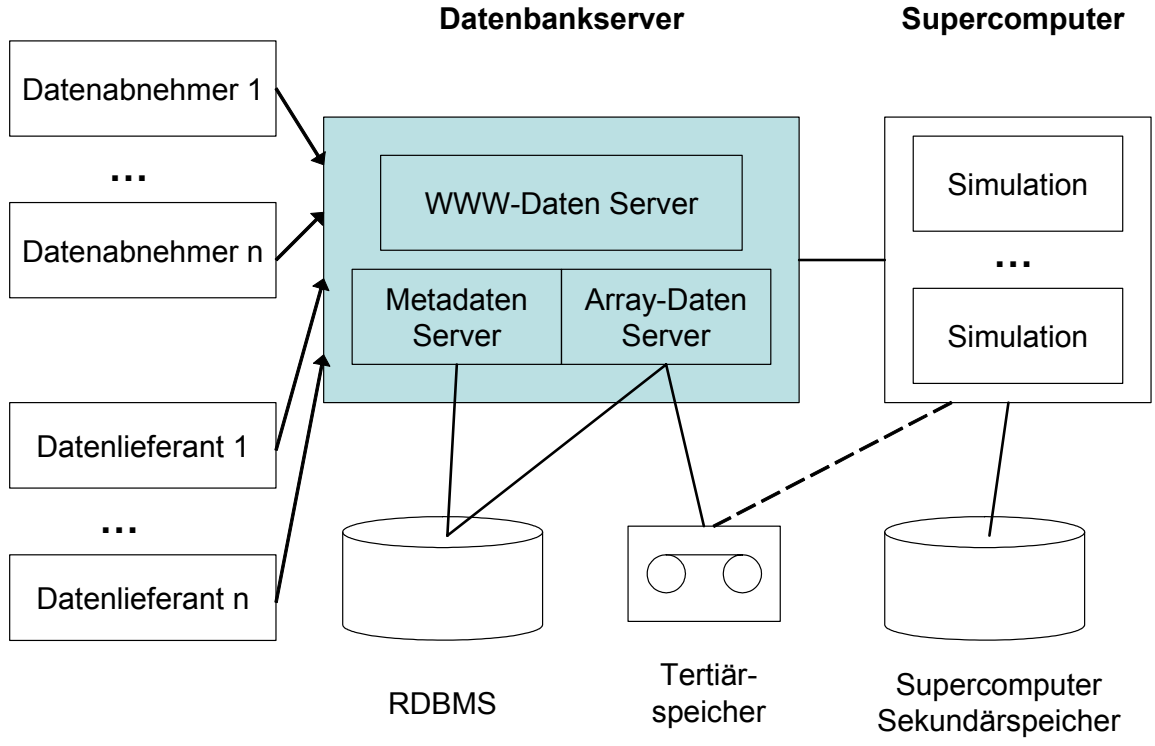


Abbildung 1.3: ESTEDI Applikationsarchitektur

Abbildung 1.3 skizziert die allgemeine Architektur, die im Projekt ESTEDI für verschiedene Anwendungsgebiete realisiert wurde. Kern der Architektur bildet ein Datenbankserver, der die multidimensionalen Array- bzw. Rasterdaten sowie obligatorische Metadaten in einem relationalen Datenbankmanagementsystem (RDBMS) speichert. Die zu verwaltende Datenmenge macht eine Speicherung komplett auf Sekundärspeicher unmöglich, weshalb Tertiärspeicher integriert wird. Ein Zugriff auf Tertiärspeicher soll in der Semantik der Rasterdaten und effizient über den Array-Daten Server erfolgen [Rei05]. Die zu verwaltenden Daten werden primär von Supercomputern der Rechenzentren geliefert. Diese Daten entstehen durch

numerische Berechnungen von Simulationen oder aufbereiteten Sensordaten (z.B. Satellitendaten). Sekundär können Daten von externen Datenlieferanten in das System integriert werden. Der Zugriff auf und die Analyse von Daten erfolgt in der Regel über einen WWW Server. Typische Datenabnehmer werden angebotene Daten genau analysieren, d.h. aggregierende Berechnungen und Ausschnitte der Daten anfordern, bevor sie bestimmte Daten bzw. Bereiche aus diesen evtl. komplett übertragen. Dieser navigierende Zugriff der Benutzer muss durch entsprechende Applikationen auf Seite des WWW Servers unterstützt werden. Die gestrichelte Linie in Abbildung 1.3 entspricht der ursprünglichen Speicherung der gewonnenen Daten auf Tertiärspeicher. Diese Speicherung, die keinerlei Semantik der Daten kennt und somit keinen selektiven Zugriff oder Analysen zulässt, sollte durch Standarisierung eines Speicherformats für multidimensionale Rasterdaten überflüssig werden.

Aus dieser Architektur für Applikationen des ESTEDI Projektes lassen sich neben den beiden oben genannten Zielen, der effizienten Verwaltung von Rasterdaten und der Generierung eines Standards für Speicherung und Austausch von Daten und Metadaten, weitere Anforderungen ableiten, wie etwa die Entwicklung geeigneter Analysewerkzeuge, die Unterstützung paralleler Hardware für effiziente Analysen, den Zugriff auf Tertiärspeicher, etc. Bei Abschluss des Projektes existierte neben einer Reihe von Applikationen, welche diverse Spezialgebiete abdeckten, auch ein um Basisfunktionalität erweitertes Array DBMS, das von diesen diversen Anwendungen in identischer Architektur aber mit sehr unterschiedlichen Anforderungen getestet und evaluiert wurde:

1. **Paralleles Array DBMS** rasdaman
   Die Basis der ESTEDI Architektur ist das rasdaman Array DBMS. Es wurde von **FORWISS** (Bayerisches Forschungszentrum für Wissensbasierte Systeme, München) bezüglich paralleler Anfrageverarbeitung (auf allen relevanten parallelen Architekturen) und direkten Tertiärspeicherzugriff erweitert. Sekundär wurden von FORWISS Methoden der erweiterten Datenanalyse, Datenkonvertierungen für die Datenintegration, sowie Verbesserungen der Anfrageoptimierung entwickelt und implementiert. **Active Knowledge GmbH**, München (seit 2003 rasdaman GmbH) integrierte Mehrbenutzerfunktionalität sowie benutzerdefinierte Funktionen in das rasdaman Array DBMS.
2. **Strömungsdynamik**
   Das Softwareunternehmen **Numeca International**, Brüssel, Belgien, ist spezialisiert auf die Visualisierung von Daten aus dem Bereich Strömungsdynamik. Im Rahmen des Projektes erfolgte eine Kopplung von rasdaman an das Visualisierungswerkzeug von Numeca. Mit Hilfe der ESTEDI Architektur konnte ein selektiver Zugriff auf die oft umfangreichen Datensätze sowie eine Verlagerung von großen Teilen der Berechnungen auf einen entfernten Server realisiert werden.
3. **Kosmologie**
   Das Rechenzentrum **Cineca**, Bologna, Italien (Inter-University Consortium for the Management of the Electronic Computing Centres of North-Eastern Italy) ist ein Spezialist für Simulationen und Visualisierung von Daten aus dem Bereich Kosmologie. rasdaman wird hier als Datenserver eingesetzt. Das von Cineca weiterentwickelte Visualisierungswerkzeug AstroDB wurde um ein Modul für Anfragen an rasdaman erweitert. Für dieses Anwendungsgebiet hatte die Auslagerung komplexer Berechnungen auf Hochleistungsserver zentrale Bedeutung.
4. **Meteorologie**
   CLRC (Central Laboratory of the Research Councils) unterhält Einrichtungen in Rutherford, Daresbury und Chilbolton, Großbritannien, und ist eine Institution für die Unterstützung von Forschern und Spezialisten vor allem aus dem Bereich Meteorologie (größten-

teils Daten aus Observation, aber auch Simulation). CLRC stellte eine GRID (engl.: grid, Gitter, Zusammenfassung der Ressourcen von diversen Computern in einem Netzwerk zur Lösung eines Problems) Umgebung zur Verfügung, die rasdaman als Datenserver nutzt aber auch den Zugriff auf weitere Datenquellen weltweit einfach realisieren konnte. Somit war das Array DBMS rasdaman eine von vielen Datenquellen, wenn auch die zentrale. Das Max-Planck-Institut für Meteorologie **MPI-Met**, Hamburg, Deutschland versteht sich als Servicezentrum für Forscher aus dem Bereich Klimaforschung in Deutschland und Europa. ESTEDI bot die Möglichkeit, diesen Nutzern einen Service für effiziente Datenextraktion und Datenanalyse anzubieten. Zentrale Rolle spielten hierbei die serverseitige Selektion von Bereichen aus extrem großen 3D und 4D Simulationsdaten sowie die Möglichkeit einer dynamischen Datenanalyse vor deren Komplettübertragung.

5. **Chemie**
   Das Swiss Center for Scientific Computing, **CSCS**, Lugano, Schweiz ist das Rechenzentrum der ETH Zürich. Die Expertise von CSCS liegt vor allem im Bereich der Simulation und Analyse von chemischen Reaktionen. Ein effizienter Zugriff auf entsprechende Daten und Analysemöglichkeiten per WWW wurden realisiert.
6. **Satellitengestützte Erdbeobachtung**
   Das Deutsche Fernerkundungsdatenzentrum (DFD) des Deutschen Zentrums für Luft- und Raumfahrt e.V., **DLR**, mit Zentrale in Oberpfaffenhofen bei München, sammelt und vertreibt Satellitendaten. Somit war das Hauptinteresse die effiziente Extraktion von Anfragebereichen aus extrem großen 2D Satellitendaten und deren Bereitstellung über das WWW. Ferner wurden durch die neue Technologie erstmals effiziente Zeitreihenanalysen auf Satellitendaten möglich.
7. **Genforschung**
   IHPC&DB, Russland (Institute for High Performance Computing and DataBases) in St. Petersburg bietet Simulation, Datenaufbereitung und Datenanalyse im Bereich Genetik. Ein Web Portal mit rasdaman als Basis wurde hierfür im Rahmen des ESTEDI Projektes entwickelt. Das Hauptinteresse von IHPC&DB an rasdaman galt nur sekundär dessen Vorteilen im Umgang mit großen Datenmengen, in der Tat waren die Datenmengen von IHPC&DB eher gering im Verhältnis zu den anderen Partnern. Das Hauptaugenmerk liegt hier auf komplexen Berechnungen zur Datenanalyse.
8. **Strömungsdynamik und Verbrennungsprozesse**
   ERCOFTAC (European Research Community on Flow, Turbulence and Combustion) ist eine Vereinigung von Forschungszentren und Universitäten. Als Ziel wird der Wissenstransfer von Forschungsergebnissen in die Industrie genannt. Speziell abgedeckte Fachrichtungen beinhalten die Analyse der Strömungsdynamik von Flüssigkeiten und Gasen sowie der Untersuchung von Verbrennungsprozessen. Die Aufgabe von ERCOFTAC im ESTEDI Projekt bestand vor allem darin, Spezialisten aus oben genannten Gebieten als externe Gutachter zu gewinnen. Auch die Bemühungen um eine Standardisierung von multidimensionalen Daten sollte unterstützt werden. ERCOFTAC implementierte keine eigene Applikation, sondern stand in beratender Position und als Leiter der Evaluierung zur Verfügung.

Abbildung 1.4 zeigt beispielhaft die Frontend Werkzeuge für die Analyse, die im Rahmen des ESTEDI Projekts entstanden. Von oben links nach recht unten sieht man die Applikationen von CSCS, Cineca, IHPC&DB, DLR, MPIM und Numeca.

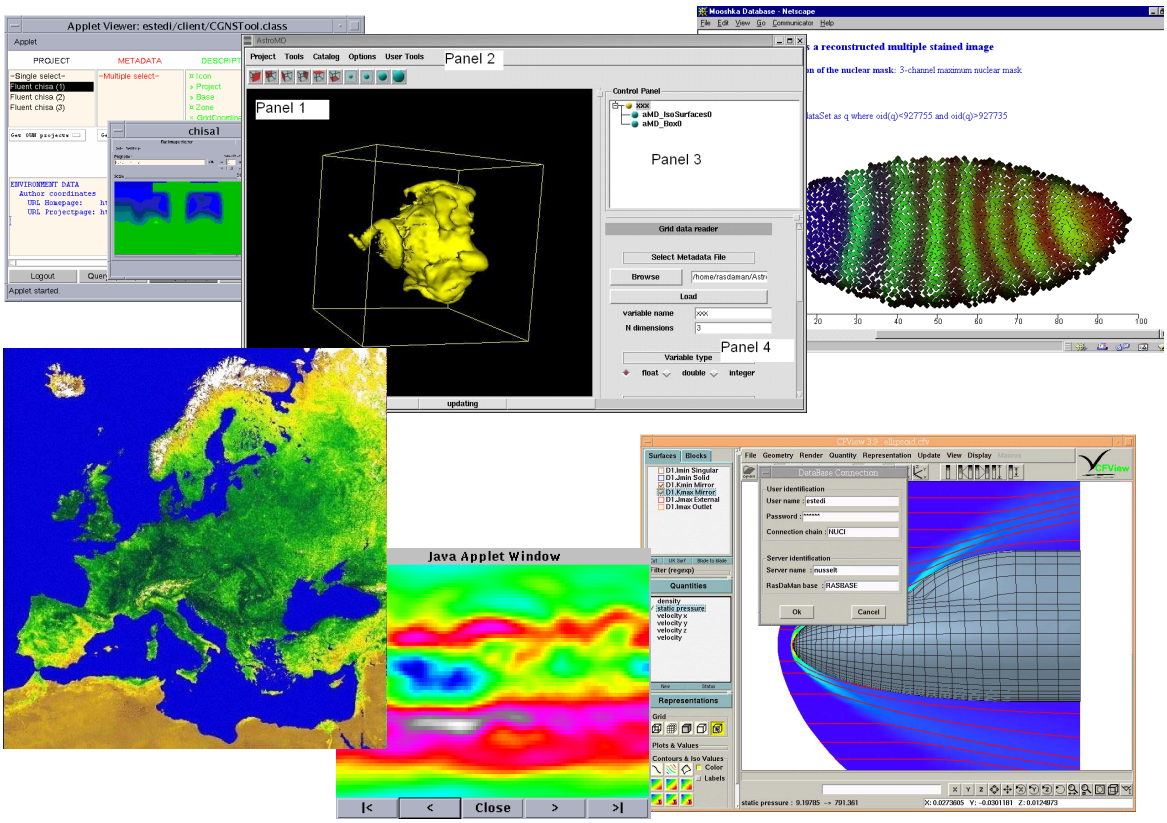


Abbildung 1.4: ESTEDI Applikationen

Wie bei diesem Umfang des Konsortiums zu vermuten, wurde das Sekundärziel von ESTEDI, die Schaffung eines einheitlichen Standards für multidimensionale Daten, nicht verwirklicht. Bereits zur Laufzeit des Projektes zeichnete sich ab, dass in den einzelnen Fachgebieten neue proprietäre Formate (CGNS[2], CERA[3], HDF[4], etc.) eingesetzt werden und die Beteiligten eine Neuentwicklung zu vermeiden suchten. Das Primärziel, die effiziente Datenextraktion und Datenanalyse, wurde hingegen realisiert. Die von ERCOFTAC zur Evaluation gewonnenen Experten bescheinigten den Systemen eine herausragende Performanz.

Auch die Forschungsschwerpunkte von FORWISS, die zu zahlreichen Publikationen und erweiterten Versionen von rasdaman führten, wurden von den Gutachtern der Europäischen Union und den Anwendern hervorragend beurteilt. So wurde die Entwicklung von Algorithmen zur Parallelisierung und somit die Nutzung paralleler Architekturen für die Analyse von Rasterdaten, die das Thema dieser Arbeit ist, vielfach theoretisch und praktisch evaluiert und ausgezeichnet bewertet. Ferner wurde eine direkte Anbindung von Tertiärspeichersystemen entwickelt, die die Unterstützung von effizienten Bereichsanfragen beibehält. Außerdem wurden (teilweise auf Anregung der Experten) Erweiterungen der Analysemöglichkeiten (und damit verbunden der Anfragesprache) und verbesserte Optimierungsalgorithmen entwickelt und implementiert.

[2] CGNS, engl.: CFD General Notation System (CFD, engl.: Computational Fluid Dynamics)
[3] CERA, engl.: Climate and Environmental data Retrieval and Archive system
[4] HDF, engl.: Hierarchical Data Format

1.5 Ziele und Anforderungen

Diese Arbeit entstand zum Großteil während des Projektes ESTEDI. Aus diesem Grund standen bei Entwicklung und Implementierung der Konzepte praktische Überlegungen im Vordergrund. Projektziel war diesbezüglich, möglichst schnell einen voll funktionsbereiten parallelen rasdaman Server zu entwickeln, der dann von Experten im laufenden Betrieb getestet werden konnte und weiterentwickelt wurde. Die Implementierung der in dieser Arbeit beschriebenen Konzepte ist folglich nicht nur ein „proof of concept", sondern ein erprobtes paralleles Array DBMS. Das Primärziel dieser Arbeit wurde bereits zu Beginn des ESTEDI Projektes festgesetzt und bis zum Abschluss des Projektes voll erreicht:

> Die Analyse von großen multidimensionalen Rasterdaten muss effizient möglich sein. Diese effiziente Datenanalyse ist nur durch den Einsatz paralleler Architekturen und einer damit einhergehenden Performanzsteigerung möglich

Aus dieser Zielvorgabe lassen sich diverse Anforderungen ableiten. Diese vereinfachen den Einsatz des parallelen Array DBMS und sichern die Akzeptanz durch die Anbieter von Analysediensten und deren Benutzern:

- Verbesserung der **Performanz** für Einzelanfragen
  Der primäre Grund für die Integration paralleler Anfrageverarbeitung in rasdaman im Projekt ESTEDI war die Beobachtung, dass bereits einzelne Operationen auf Array Daten die CPU zum Flaschenhals einer Anfrage werden lassen können [Rit99]. Eine Beschleunigung von Anfragen, die oftmals mehrere Minuten oder Stunden in Anspruch nehmen können, durch Nutzung paralleler Architekturen ist also häufig möglich. Da Array DBMS keine große Anzahl von Anfragen gleichzeitig bearbeiten, sondern eher wenige, dafür komplexe Anfragen, genügt auch eine Parallelverarbeitung von verschiedenen Transaktionen nicht für eine Verbesserung der Antwortzeiten. Die parallele Ausführung innerhalb von Anfragen (engl.: intra-query parallelism) ist notwendig.
- Unterstützung aller relevanter **paralleler Architekturen**
  Die Entscheidung der zu unterstützenden Architektur(en) stellte bereits zu Beginn des Projektes die Weichen für die weitere Entwicklung. Die beteiligten Rechenzentren setzten zu dieser Zeit Multiprozessorrechner mit bis zu vier CPUs als DBMS Server ein. Diese Shared-Everything Architektur kann am effektivsten durch den Einsatz von Threads oder darauf basierenden (Quasi-) Standards wie etwa OpenMP [OMP] genutzt werden. Inter-Prozess Kommunikation erfolgt in diesem Fall über gemeinsam genutzten Speicher und Sperrmechanismen. Die damit konkurrierende parallele Architektur bezeichnet man als Shared-Nothing, Rechnercluster oder auch lose gekoppelte Systeme, der große Vorteil gegenüber Multiprozessorrechnern ist vor allem die einfache Erweiterbarkeit. Die Kommunikation zwischen parallelen Instanzen auf verschiedenen Rechnern erfolgt mit Hilfe von Nachrichten oder darauf basierenden Standards wie Parallel Virtual Machine [PVM] oder Message Passing Interface [MPI]. Auch wenn diese Architektur in ESTEDI anfangs nicht genutzt wurde, sollte die Option gewahrt werden. Aus diesen Überlegungen fiel die Wahl beim parallelen Programmiermodell auf das Message Passing Interface MPI, welches 1997 in Version 2 standardisiert wurde. MPI wurde ursprünglich für Shared-Nothing entwickelt, kann jedoch ebenso in Shared-Everything Systemen (mit sehr geringen Performanzeinbussen gegenüber Threads) eingesetzt werden[5]. Das parallele Array DBMS

[5] Konkret wird LAM [LAM] eingesetzt, eine frei verfügbare Implementierung des MPI-2 Standards.

rasdaman ist folglich nicht auf eine parallele Architektur beschränkt, sondern wird sowohl in Multiprozessorrechnern wie auch Rechnerclustern eingesetzt.

- **Vermeiden von Performanzeinbussen** für Anfragen
  Parallelität erfordert die Umstrukturierung des internen Anfragebaumes, der dann von mehreren Prozessen verarbeitet werden kann. Diese Analyse und Restrukturierung des Anfragebaumes erfolgt zumeist nach einer Optimierungsphase durch ein Parallelisierungsmodul und kann unter Umständen extremen Ressourcen- und Zeitbedarf bedeuten. Dieser Aufwand soll jedoch nicht zu einer merkbaren Verschlechterung von bestimmten Anfragen führen. Mit anderen Worten darf die Analyse nicht deutlich länger dauern als die durch die parallele Verarbeitung gewonnene Verkürzung der Anfragezeit. Dies wurde durch ein Parallelisierungsmodul realisiert, welches nicht oder schlecht parallelisierbare Anfragen extrem schnell erkennt und sofort die Parallelisierung stoppt.
- Volle **Transparenz** für Benutzer
  Die einzige Auswirkung von Parallelität für den Endbenutzer soll die bessere Performanz für „gut parallelisierbare“ Anfragen sein bei keiner merkbaren Verschlechterung von „schlecht oder nicht parallelisierbaren“ Anfragen.
- Einfache **Administration** und Tuning
  Diese Forderung beinhaltet die Integration paralleler Funktionalität in die verschiedenen Administrations- und Tuning-Werkzeuge von rasdaman sowie die Anpassung von Hilfetexten, Fehlermeldungen, Dokumentation, etc.
- **Skalierbarkeit** bei Erweiterung der Hardware
  Hauptproblem der Skalierbarkeit sind Grenzen der Inter-Prozess Kommunikation (engl.: interference problem). Bei Multiprozessorsystemen wird häufig bereits bei 8 Prozessoren erwartet, dass die einzelnen Prozesse durch den gemeinsamen Zugriff auf Hauptspeicher und Bussystem deutlich gebremst werden. Für Rechnercluster stellt das Netzwerk das limitierende Element für die Skalierbarkeit dar. Aus Sicht des Administrators kann diese Grenze durch Einsatz neuer Technologien (Myrinet [Myr], InfiniBand [Inf], PCI-X [PCIX], etc.), aus Sicht des Programmierers durch die Minimierung von Kommunikation zurückgedrängt werden. Für den parallelen rasdaman Server wurde daher versucht, Kommunikation möglichst zu vermeiden. Insbesondere der Transfer von Zwischenergebnissen kann in der Praxis das Versenden von mehreren GB an Daten bedeuten. Die Algorithmen wurden folglich so ausgewählt und optimiert, dass eine minimale Kommunikation gewährleistet ist.

1.6 Überblick

Die Gliederung dieser Arbeit spiegelt die Methodik der wissenschaftlichen Arbeit wider, die zur Entwicklung von Strategien für die parallele Anfrageverarbeitung für multidimensionale Arraydaten und deren Implementierung geführt hat. Das als Implementierungsbasis eingesetzte DBMS rasdaman war bereits zu Beginn des Projektes ESTEDI ein voll einsatzfähiges und kommerziell vertriebenes Produkt, das das Stadium eines Forschungsprototyps bereits hinter sich gelassen hatte.

Das rasdaman zugrunde liegende Datenmodell und Terminologie sowie die Implementierung wird in **Kapitel 2** beschrieben. Als Datenträger fungieren multidimensionale Objekte sowie Mengen von diesen Objekten mit identischer Struktur. Die Operationen lassen sich folglich unterteilen in Operationen auf einzelne multidimensionale Objekte und Mengenmanipulationen. Die effiziente Umsetzung des logischen Modells auf die physikalische Ebene wird nur kurz analysiert. Abschließend wird die Umsetzung des Datenmodells durch das rasdaman Array DBMS aufgezeigt.

Basis und Vergleichsmaßstab für parallele Anfrageverarbeitung im Bereich multidimensionale Rasterdaten sind die kommerziellen relationalen DBMS wie etwa Oracle, IBM DB2, Microsoft SQL Server, etc. Somit widmen wir **Kapitel 3** der Beschreibung von Parallelität für relationale Daten. Hierbei wird nicht nur eine Zusammenfassung relevanter Forschungsergebnisse aufgezeigt, sondern auch eine Wertung bezüglich deren Integration in die großen kommerziellen Datenbanken vorgenommen. Da die parallele Verarbeitung speziell von Array Daten bisher keinerlei Beachtung in der wissenschaftlichen Welt gefunden hat, ist dieser kurze Überblick über Parallelität in relationalen Systemen als komprimierter „State of the Art" zu sehen.

In **Kapitel 4** werden Algorithmen für eine parallele Ausführung von Anfragen an Rasterdaten entwickelt und detailliert beschrieben. Grundsätzlich lassen sich die Methoden in zwei Klassen einteilen: Algorithmen, die auf relationalen Methoden der Parallelisierung basieren, jedoch speziell auf die Eigenschaften von Array Daten hin optimiert wurden. Zweitens, Algorithmen, die ausschließlich für diese Art von Daten entwickelt wurden. Darunter fallen etwa Methoden wie das (feingranulare) Partitionieren von multidimensionalen Datenobjekten für eine Parallelverarbeitung.

Die Implementierung dieser Konzepte in das Array DBMS rasdaman wird in **Kapitel 5** skizziert. Hierbei wird vor allem auf allgemeine Designentscheidungen sowie die Architektur der entstandenen Implementierung rasdaman 5.0 PE eingegangen.

Die Qualität der entwickelten Parallelisierungsstrategien sowie der Implementierung wird in **Kapitel 6** durch umfangreiche Performanztests aufgezeigt. Der Schwerpunkt liegt hierbei nicht nur auf Skalierbarkeit in Multiprozessorsystemen, sondern auch auf dem Einsatz homogener Verbünde von „normalen" Rechnern. Eine solche Strategie erlaubt die dynamische Adaption an die erforderliche Rechenleistung durch einfache Integration von Rechnern in den Verbund im laufenden Betrieb.

Die Arbeit schließt mit einer Zusammenfassung des wissenschaftlichen Beitrags zur Forschung und dem obligatorischen Ausblick in **Kapitel 7**.

Kapitel 2 Datenmodell und Terminologie

Data is not information,
Information is not knowledge,
Knowledge is not understanding,
Understanding is not wisdom.
(Cliff Stoll & Gary Schubert)

In diesem Kapitel wird nach einer kurzen Einführung, die ein typisches Beispiel für multidimensionale Daten veranschaulicht, in Kapitel 2.2 ein Datenmodell für Objekte beliebiger Dimensionalität vorgestellt. Ein Datenmodell unterteilt sich in der Regel in die Definition einer Trägermenge und der Definition von Operationen, die auf dieser Trägermenge ausführbar sind. Die hier verwendete Definition von Trägermenge und Operationen basiert auf [Bau99] und liegt auch dem Datenbanksystem **rasdaman** [BFRW97] zugrunde, das in dieser Arbeit als Basis für die Implementierung benutzt wurde. Im Rahmen des ESTEDI Projektes wurden jedoch auch neue Operationen zur erweiterten Datenanalyse entwickelt, welche bereits in das Datenmodell integriert sind.

Da die definierten multidimensionalen Objekte sowie die Operationen darauf als neuer Datentyp bzw. neue Funktionen in das relationale Modell [Cod70] einbezogen werden sollen, ergibt sich eine Zweiteilung des Datenmodells. Auf der einen Seite stehen hierbei Relationen und die bekannten Operationen darauf, auf der anderen Seite werden als neuer Attributtyp multidimensionale Objekte und spezielle Operationen darauf definiert. In 2.3 wird die Fusion von multidimensionalem und relationalem Datenmodell diskutiert, was zu Erweiterungen aber auch Einschränkungen bekannter relationaler Operationen führt.

Die physikalische Speicherung multidimensionaler Objekte in einem relationalen DBMS wird in 2.4 behandelt. Dünn besetzte multidimensionale Räume (beispielsweise Daten eines Data Warehouse [Inm96], [Inm02]) werden typischerweise als Vektordaten in RDBMS abgelegt, mittels Speicherung der Position des Datenpunktes in jeder Dimension sowie dessen Wertes.

In unserem Kontext sind multidimensionale Datenobjekte immer Arrays bzw. Rasterdaten, das heißt komplett besetzte multidimensionale Räume, wobei alle Punkte explizit gespeichert werden. Die Unterschiede zwischen stark bzw. dünn besiedelten Räumen sowie die Auswirkung auf die Speicherung in RDBMS werden in diesem Kapitel näher diskutiert.

In Kapitel 2.5 wird abschließend das Array DBMS **rasdaman** vorgestellt, das im Rahmen vorliegender Arbeit für parallele Anfrageverarbeitung adaptiert wurde. **rasdaman** ist ein kommerziell vertriebenes DBMS für Daten beliebiger Dimensionalität und Größe. Zur Datenhaltung wird ein relationales DBMS verwendet, wahlweise Oracle, IBM DB2 oder Informix. Die für multidimensionale Objekte und Operationen optimierte Anfragesprache basiert auf SQL92 [ISO92].

Die Definition einer großen Menge von spezialisierten Funktionen auf Arrays und die Einbettung in eine mengenbasierte Anfragesprache bildet die primäre Abgrenzung zu anderen Datenmodellen für Array Daten. So wird etwa von Marathe [MS97] [MS99] [MS02] ein Datenmodell und eine Anfragesprache für Array Daten vorgeschlagen, welches jedoch keine Mengen von gleichartigen Arrays definiert. Ferner werden nur grundlegende Operatoren für die Kondensierung einzelner Arrays und Zusammenfassung verschiedener Arrays fest in die Anfragesprache integriert. Komplexe Funktionen werden mit einem Operator *apply* gebunden und sind immer extern definiert. Sarawagi und Stonebraker beschränken sich in [SS94] auf den effizienten Zugriff von multidimensionalen Arrays. Dies wird mit Hilfe spezieller Speicherstrategien erreicht (engl.: chunking, reordering, redundancy, partitioning; Kachelung, Restrukturierung, Redundanz, Partitionierung). Libkin et al. entwarfen in [LMW96] eine Anfragesprache basierend auf einer Algebra für multidimensionale Arrays. Dieser sehr theoretische Ansatz wurde in einem System implementiert, welches aber nie über den Stand eines Prototyps hinauskam.

2.1 Multidimensionale Array Daten: ein einführendes Beispiel

Bevor ein multidimensionales Datenmodell formal spezifiziert wird, soll in diesem Abschnitt eine Einführung mittels eines Beispiels gegeben werden. Verwendete Definitionen und Notationen werden anhand dieses Beispiels erklärt, die Formalismen werden im folgenden Kapitel nachgeliefert.

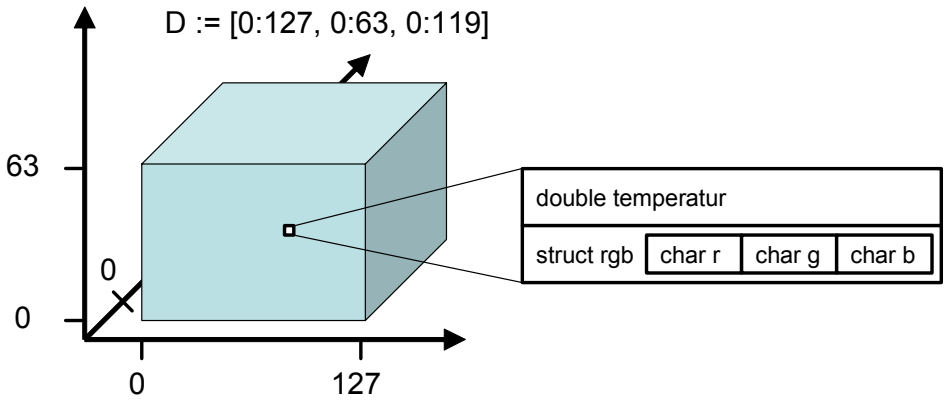


Abbildung 2.1 Beispiel für einen multidimensionalen Datentyp

Ein **Rasterdaten-Objekt** α beliebiger Dimensionalität, im Folgenden auch **MDD** (engl.: Multidimensional Discrete Data, multidimensionales diskretisiertes[6] Objekt) genannt, besitzt einen Datentyp, der die interne Struktur des Objektes beschreibt.

Der **Datentyp M** eines MDD α setzt sich zusammen aus der multidimensionalen Domäne D, d.h. dem multidimensionalen Raum, in dem dieses Objekt positioniert ist und dem Basisdatentyp T der Zellen. In unserem Beispiel in Abbildung 2.1 wird ein 3-dimensionaler Datentyp spezifiziert. Die Ausdehnung in den Dimensionen reicht von 0 bis 127 in der ersten Dimension, von 0 bis 63 in der zweiten und von 0 bis 119 in der dritten. Dieses Intervall wird dargestellt durch die Schreibweise [0:127, 0:63, 0:119]. Der Basisdatentyp T setzt sich zusammen aus einem Fließkommawert doppelter Genauigkeit (*double*) zur Speicherung eines Temperaturwertes und einem Rot-Grün-Blau-(RGB)-Wert, der durch je einen Datentypen *char* für die einzelnen Komponenten gekennzeichnet ist, also T = {double; {char; char; char}}[7].

Ein **Datenobjekt** dieses Typs hat folglich 983.040 Zellen (128 in der ersten Dimension multipliziert mit 64 in der zweiten Dimension multipliziert mit 120 in der dritten Dimension) und speichert 2 Variablen (wobei die zweite Variable selbst aus 3 Variablen besteht) für jede dieser Zellen. Eine atomare (nicht weiter strukturierte) Variable wird auch häufig **Kanal** (engl.: channel) genannt. Da in Rasterdaten jeder Datenpunkt explizit gespeichert wird (auch wenn der Wert 0 ist), veranschlagt also ein Datenobjekt dieses Typs unkomprimiert ca. 10,8 MB (8 Byte für double, 1 Byte pro char, also 11 Byte pro Zelle des Objekts). Bei Daten mit großer Dimensionalität zeigt sich bald der „Fluch der Dimensionalität“, d.h. jede weitere Dimension erhöht die Anzahl der Zellen und somit den Speicherbedarf enorm. Die Erweiterung des Beispielobjekts um eine Dimension für 64 Höhenstufen (ein Objekt wäre dann also 4-dimensional und beinhaltet Länge, Breite, Höhe und Zeit) ergäbe etwa einen Speicherbedarf von 692 MB pro Objekt.

Die Visualisierung von multidimensionalen Objekten ist weiterhin Gegenstand von Forschung (siehe etwa [DB00] und [SA02]). Abbildung 2.2 zeigt mögliche Darstellungen unseres Datenobjekts als Datenwürfel (sichtbar ist die Oberfläche), als Tabelle und als orthogonale Schnitte durch den Datenwürfel. Es wird die Variable für Temperatur farblich visualisiert, wobei eine Farbverteilung von blau (RGB 255, 0, 0) für das Minimum der Werte bis rot (RGB 0, 0, 255) für die maximale Temperatur gewählt wurde. Die Darstellung als Datenwürfel links lässt erkennen, dass es sich bei dem Datenobjekt um die Simulation eines Klimamodells handelt, welches die Temperatur auf Meereshöhe im Laufe einer Zeitspanne ermittelt. Die Erdoberfläche befindet sich hier auf der Vorderseite des Würfels, man beachte die hohen (rötlich dargestellten) Temperaturen nahe dem Äquator und die kalten (blauen) Regionen nahe der Pole. In der Zeitdimension, die sich nach hinten erstreckt, kann man gut die periodischen Schwankungen im Jahresverlauf erkennen. Weitere Möglichkeiten zur Visualisierung multidimensionaler Daten beinhalten die Darstellung als Höhenlinien, Histogramm oder als Animation.

Auf ein multidimensionales Objekt α kann eine Reihe von speziellen **multidimensionalen Operationen** angewendet werden. Vor einer kompletten formalen Definition dieser Operati-

[6] „Diskretus“ (lateinisch) bedeutet soviel wie „getrennt“. In der Mathematik wird der Begriff diskrete Menge für eine Menge mit einer Kardinalität kleiner gleich $\aleph_0$ verwendet, d.h. die einzelnen Punkte sind klar voneinander trennbar. Ein diskreter Raum ist in unserem Sinne ein Raum $\mathbb{Z}^d$, somit werden die Daten auf ein d-dimensionales Raster abgebildet und so gespeichert (also diskretisiert).

[7] Die Bezeichnung der Datentypen ist an das rasdaman DBMS angelehnt. Der Datentyp *char* dient in rasdaman ebenso zur Speicherung eines Byte, ein Datentyp *byte* ist nicht vorhanden.

onen in Kapitel 2.2 werden die wichtigsten hier vorgestellt und zur Veranschaulichung auf unser Beispiel übertragen.

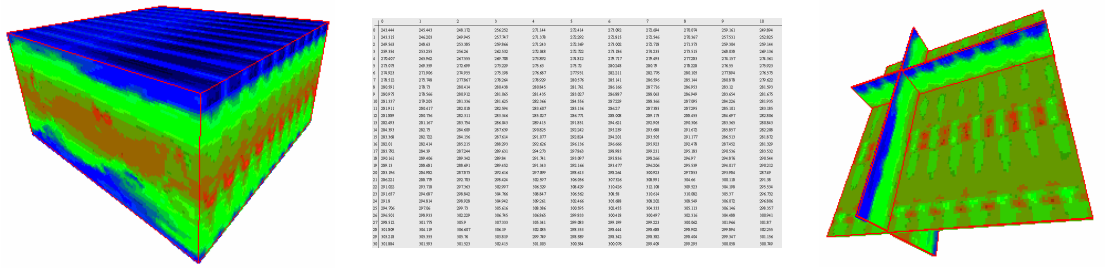


Abbildung 2.2 Beispiel für die Visualisierung eines Datenobjektes

Durch **Projektionen auf dem Datentyp** (auch als **Zelloperationen** oder als **Kanalselektierung** bezeichnet) kann man ein Objekt auf eine Teilmenge der Variablen des Basisdatentyps reduzieren. So kann etwa die Temperatur unseres Beispielobjektes für weitere Operationen selektiert werden mit einer Zellenoperation $\alpha.temperatur$. Das Resultat ist ein Datenwürfel mit gleicher Domäne, der Basisdatentyp ist jedoch auf einen Wert des Typs *double* für die Temperatur eingeschränkt.

Geometrische Operationen spezifizieren die Domäne des Objekts, was entweder in der Einschränkung einer oder mehrerer Dimensionen, in der Fixierung einer Dimension auf einen Wert oder in der Verschiebung der Domäne resultieren kann. In unserem Beispiel schränkt etwa $\alpha[0:127, 0:63, 108:119]$ die dritte Dimension auf die 12 letzten Werte ein, spezifiziert also das letzte Jahr der Simulation. Dagegen ergibt die Operation $\alpha[0:127, 0:63, 119]$ ein 2-dimensionales Objekt, indem die dritte Dimension auf den Wert 119 fixiert wird. Man beachte, dass der Zugriff auf eine konkrete Zelle des Objekts als ein Spezialfall von geometrischer Operation gesehen werden kann.

Induzierte Operationen sind auf einem Basistyp des Arrays definierte Operationen, angewendet auf multidimensionale Arrays. Auf dem Basistyp *double* für die Speicherung der Temperaturwerte ist beispielsweise die Operation „<“ definiert, die einen booleschen Ergebnistyp *true* oder *false* als Resultat hat. Diese Operation angewendet auf ein Array, das durch Projektion auf die Variable Temperatur entstand (d.h. nach vorheriger Zelloperation auf dem Ursprungsobjekt), ergibt ein Objekt mit identischer Domäne und einem booleschen Wert *true* oder *false* in jeder Zelle (bezüglich des Prädikats). So können etwa Bereiche mit markanten Zellwerten des Arrays identifiziert werden. Neben Vergleichsoperationen sind arithmetische Operationen typische Beispiele dieser Klasse, wie etwa Addition, Subtraktion, Multiplikation, Division, Logarithmus, Wurzel, etc.

Aggregationsoperationen auf ein MDD angewendet, ergeben ein MDD mit eingeschränktem multidimensionalem Intervall oder Dimensionalität (meist ein skalarer Wert) als Resultat. Typische Vertreter dieser Klasse von Operationen sind Maximum und Minimum, Durchschnittswert, etc. In unserer Klimasimulation ist der Durchschnittswert der Temperaturen $avg(\alpha)$ in etwa 279 Kelvin, also ungefähr 6° Celsius.

Nach dieser kurzen anschaulichen Einführung werden nun Datenmodell und Operationen formal definiert.

2.2 Logisches Datenmodell

In diesem Kapitel wird das logische Datenmodell, bestehend aus der Trägermenge der multidimensionalen Objekte und den Operationen darauf, vorgestellt. Ein konkretes multidimensionales Objekt, welches eine Abbildung von einem multidimensionalen Intervall (seiner Domäne) auf konkrete Zellwerte vornimmt, wird im Weiteren kurz als MDD bezeichnet (wir verwenden für konkrete Objekte die Notation α, β, γ, etc.). Die Schreibweise $\underline{x}$ stellt einen Punkt des multidimensionalen Raumes dar, auch Vektor oder (multidimensionale) Position genannt. $\underline{x}[k]$ ergibt den Wert des Punktes $\underline{x}$ in Dimension k. Eine Übersicht der verwendeten Notation befindet sich im Anhang.

2.2.1 Trägermenge: Multidimensionale Objekte

Kurz gesagt ordnet ein konkretes multidimensionales Objekt (MDD, Multidimensional Discretized Data) α jedem Vektor $\underline{x}$ seiner multidimensionalen Domäne D einen Wert eines Basisdatentyps T zu. Das MDD wird also bestimmt durch eine Menge von Zellen, das heißt Tupel der Form *Zelle=(Position, Wert)*. Die Definitionsmenge dieser Abbildung ist folglich ein multidimensionales Intervall, die Wertemenge ist per Definition die Menge aller Werte eines gegebenen skalaren (einfachen) oder komplexen (zusammengesetzten) Basisdatentyps. Davon zu unterscheiden ist der Datentyp des MDD selbst, welcher durch die multidimensionale Domäne und den Basisdatentyp spezifiziert wird.

Definition 2.1 Multidimensionales Intervall D

Ein multidimensionales Intervall D der Dimensionalität d über den Begrenzungsvektoren $\underline{l}$ (untere Grenze) und $\underline{h}$ (obere Grenze) mit $\underline{l}, \underline{h} \in \mathbb{Z}^d$ und $l_i \le h_i$ für alle i = 1, …, d ist definiert als

$$D := \bigtimes_{i=1}^{d} \{x \mid l_i \le x \le h_i, x \in \mathbb{Z}\} = [l_1 : h_1] \times \ldots \times [l_d : h_d].$$

Das multidimensionale Intervall D wird also in jeder Dimension durch ein eindimensionales Intervall $[l_i:h_i]$ begrenzt.

Mit anderen Worten ist D die Menge aller Punkte im aufgespannten multidimensionalen Intervall $[l_1:h_1, \ldots, l_d:h_d]$ (vereinfachte Schreibweise). Folgende Funktionen sind hilfreich, um die Domäne eines MDD zu spezifizieren:

- low(D) = $\underline{l}$ ergibt den unteren Grenzpunkt $[l_1, l_2, \ldots, l_d]$ des Intervalls
- high(D) = $\underline{h}$ liefert den oberen Grenzpunkt $[h_1, h_2, \ldots, h_d]$ des Intervalls
- extent(D) = $\underline{h} - \underline{l} + 1$ beschreibt die Ausdehnung des Intervalls, wobei die Differenz zweier Vektoren $\underline{l}$ und $\underline{h}$ definiert sei als $\underline{h} - \underline{l} = [h_1 - l_1, \ldots, h_d - l_d]$ und die Addition eines skalaren Wertes 1 zu einem Vektor $\underline{x}$ als $\underline{x} + 1 = [x_1 + 1, \ldots, x_d + 1]$. Die Kurznotation extent(D) = 1 sei eine Domäne mit Ausdehnung 1 in jeder Dimensionen.
- dim(D) = d gibt die Dimensionalität d des Intervalls zurück

Es folgt die Definition des Basisdatentypen eines MDD, also des Typs eines jeden seiner Zellen.

Definition 2.2 Basisdatentyp T eines multidimensionalen Objektes

Sei S eine Menge von gegebenen skalaren Datentypen ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{B}$, etc.). Die Menge aller Werte (Variablen) eines Basisdatentyps T ist dann definiert als

$$T := \bigtimes_{i=1}^{n} \{t_i \mid t_i \in S \cup T\}; n \in \mathbb{N}; S \in \mathbb{N} \cup \mathbb{Z} \cup \mathbb{Q} \cup \ldots .$$

Folglich kann ein Basisdatentyp mehrere Komponenten besitzen, wobei jede Komponente eine skalare Variable (*integer*, *float*, *double*, etc.) sein kann oder eine Variable, die selbst aus einem zusammengesetzten Datentyp besteht. Um Verwechslungen mit Intervallen und Punktwerten zu vermeiden, werden Typbezeichner in geschweifte Klammern gesetzt: $\{typ_1; \ldots; typ_n\}$. Durch diese rekursive Definition sowie durch das Fehlen einer Obergrenze für die Anzahl der Variablen n lassen sich theoretisch unendlich große Basistypen und somit MDD definieren. In der Praxis stellt das kein Problem dar, da die Größe von Objekten immer begrenzt ist.

Ein multidimensionales Objekt ist die Abbildung eines Wertes des Basistyps zu jedem Vektor seiner Domäne, also des multidimensionalen Intervalls, den es einnimmt.

Definition 2.3 Multidimensionales Objekt (MDD) α

Ein multidimensionales Objekt α mit dem Basistyp T und der Domäne D ist eine Abbildung f: D$\rightarrow$ T

$$\alpha := \{(\underline{x}, v(\underline{x})) \mid v(\underline{x}) \in T, \underline{x} \in D\}$$

$v(\underline{x})$ steht in diesem Zusammenhang für „Wert an der Position $\underline{x}$“ (engl.: value). Eine häufiger benutzte Notation ist $\alpha[\underline{x}]$ für den Zellwert des Arrays α an Position $\underline{x}$. Ein Tupel $(\underline{x}, v(\underline{x}))$ wird eine Zelle genannt, bestehend aus Position und Zellwert.

Die Begriffe multidimensionales Objekt, multidimensionales Array, Rasterdatenobjekt und MDD werden im Folgenden synonym verwendet. Folgende Funktionen sind zur Spezifizierung eines konkreten MDD hilfreich:

- base(α) = T liefert den Basistyp des Objekts
- sdom(α) = D ergibt die Domäne (engl.: spatial domain) des Objekts

Wir benutzen abkürzend low(α) = low(sdom(α)), etc., wobei sdom(α) = D. Ferner besagt die Notation D’ < D, dass $l_i(D) \leq l_i(D')$ und $h_i(D) \geq h_i(D')$ für alle i = 1, …, d. Mit anderen Worten: das Intervall D’ liegt vollständig innerhalb des Intervalls D.

Wir kommen nun zur Definition des Typs eines multidimensionalen Objekts, nicht zu verwechseln mit seinem Basisdatentyp, also dem Datentyp der Zellen.

Definition 2.4 Typ M eines multidimensionalen Objekts

Der Typ eines multidimensionalen Objekts setzt sich zusammen aus seiner Domäne D und seinem Basisdatentyp T. Wir verwenden die Schreibweise type(α) = <D, T>. Die Menge M = <D, T> ist also die Menge aller multidimensionalen Objekte mit Domäne D und Basistyp T.

Zur Veranschaulichung dieser Definitionen betrachte man noch einmal das Beispiel aus Kapitel 2.1. Die Domäne des MDD ist D = [0:127, 0:63, 0:119], mit Dimensionalität dim(D) = 3, Begrenzungsvektoren low(D) = [0, 0, 0], high(D) = [127, 63, 119] und Ausdehnung extent(D) = [128, 64, 120]. Der Basisdatentyp ist T = {double; {char; char; char}}[8] für die Speicherung eines Temperaturwerts als Gleitkommazahl und eines RGB Wertes mit je einem Byte pro Farbkanal. Der Datentyp des MDD ist folglich M = <[0:127, 0:63, 0:119], {double; {char; char; char}}>. Das MDD selbst wird durch eine Abbildung f: D → T spezifiziert, also beispielsweise durch Angabe aller (Vektor, Wert)-Tupel:

- ([0, 0, 0], {-23,567, {0, 0, 255}}),
- ([0, 0, 1], {-23,230, {0, 1, 255}}),
- etc.

2.2.2 Operationen auf multidimensionalen Daten

Operationen auf multidimensionalen Daten lassen sich in Hilfsfunktionen und multidimensionale Operationen unterteilen. Während die Hilfsfunktionen, auf ein MDD angewendet, Meta-Informationen bezüglich Struktur, Lage, Größe, Datentyp, etc. liefern, also mit anderen Worten auf Daten zurückgreifen, die bei der Definition der Datenobjekte im Systemkatalog angelegt wurden, sind multidimensionale Operationen auf der Trägermenge abgeschlossen, liefern also als Resultat ein MDD, möglicherweise jedoch mit anderer Dimensionalität und Zelltyp[9].

Beispiele für die **Hilfsfunktionen**, die bereits im letzten Kapitel gegeben wurden, sind sdom(α), base(α), low(α), high(α), extent(α), dim(α), type(α), etc. Der Rückgabewert ist typischerweise kein MDD, sondern Informationen über die Struktur der Daten (Domäne, Zelltyp, Begrenzungspunkte der Domäne, etc.).

Die **multidimensionalen Operationen**, die Inhalt dieses Kapitels sind, lassen sich in verschiedene Klassen unterteilen, nämlich elementare Basisoperationen, Projektion auf dem Basistyp, geometrische Operationen, induzierte Operationen und Aggregationsoperationen. Es folgen die formalen Definitionen, begleitet von Beispielen für die einzelnen Operationen, basierend auf dem Beispielobjekt aus Kapitel 2.1.

---

[8] Notation angelehnt an das rasdaman DBMS. Der Datentyp *byte* ist in rasdaman nicht vorhanden und wird durch *char* dargestellt.

[9] Die Abgeschlossenheit ist im Array DBMS RasDaMan (Kapitel 2.5) nicht durchgehend umgesetzt.

2.2.2.1 Elementare Array Operationen

Die Basis für Array Operationen bilden zwei **Elementarfunktionen**, ***marray*** als genereller Konstruktor für ein MDD beliebiger Ausdehnung und Dimensionalität und ***condense***, welches ein neues MDD aus einem gegebenen mittels Berechnungsfunktion erzeugt.

Der Konstruktor *marray* spannt ein MDD der Domäne D auf. Die Zellwerte werden durch eine Funktion f($\underline{x}$) berechnet, $\underline{x} \in \mathbb{Z}^d$ ist eine freie Koordinate, die zu diesem Zweck über alle Positionen in D iteriert:

marray $_{D, \underline{x}, f(\underline{x})}$: $\varepsilon \rightarrow <D, T>$, $\varepsilon \rightarrow \alpha$ mit $\alpha = \{ (\underline{x}, f(\underline{x})) \mid x \in D\}$

Im Gegensatz dazu ist die allgemeine Aggregationsfunktion *condense* die Basis für alle aggregierenden Berechnungen [Rit99]. Insbesondere die Definition eines neuen Arrays im Zusammenhang mit einer Berechnungsfunktion, die auf *condense* basiert, ist von Interesse. Das Projekt ESTEDI hat gezeigt, dass diese Funktionen wegen der Komplexität der Notation in der Praxis nicht eingesetzt werden.

2.2.2.2 Projektion auf dem Basistyp

Unter einer **Projektion auf dem Basistyp *cell*** (auch Zelloperation genannt) versteht man die Abbildung eines MDD auf ein neues MDD, welches dieselbe Domäne hat, aber als Basistyp eine Teilmenge der Elemente des ursprünglichen Basistyps enthält. Formal ist eine Zelloperation auf das MDD α bezüglich der Elemente T' also eine Funktion:

cell: $<D, T> \rightarrow <D, T'>$, $\alpha \rightarrow$ cell(α, T'), wobei $T' \subset T$ und $|T'| < |T|$.

Typisch für diese Art Operation ist die Projektion auf eine einzelne Komponente des Basistyps (Kardinalität $|T'| = 1$), man spricht in diesem Fall auch von Selektion eines Kanals (engl.: channel selection). Dies wird durch die Notation *α.temperatur* dargestellt, in diesem Fall wäre das Resultat ein MDD mit Basisdatentyp T' = {double}.

2.2.2.3 Geometrische Operationen

Geometrische Operationen *geom*, auch räumliche Operationen (engl.: spatial operations) genannt, bilden ein MDD auf ein MDD mit gleichem Basisdatentyp aber unterschiedlicher Domäne ab, wobei die ursprünglichen Zellwerte selbst nicht verändert werden. Formal ist das eine Abbildung geom: $<D, T> \rightarrow <D', T>$.
Mit anderen Worten manipulieren diese Operationen nicht die in den Objekten gespeicherten Daten (die Zellwerte), sondern beeinflussen lediglich die Domäne. Je nach Art der Einschränkung bzw. Veränderung der Domäne $D \rightarrow D'$ unterscheidet man

1. **Zuschneiden** der Domäne (engl.: trimming)
   Eine Trimming-Operation *trimm* bedeutet eine Einschränkung der Domäne, wobei die Dimensionalität der Domäne identisch bleibt:

   trimm $_{D'}$: $<D, T> \rightarrow <D', T>$, $\alpha \rightarrow$ trimm(α, D') mit dim(D) = dim(D'), D' < D.
   Die Zellwerte selbst bleiben erhalten: $\forall \underline{x} \in D'$: trimm($\alpha$, D')[$\underline{x}$] = α[$\underline{x}$][10].

   Diese Art der geometrischen Operation wird häufig nach der Art der Anfragen auch als

[10] α[$\underline{x}$] ist der Punktzugriffsoperator (Definition siehe Punkt 2).

Bereichsanfrage (engl.: range query) oder area-of-interest Anfrage bezeichnet. Man beachte, dass eine Punktanfrage kein Spezialfall dieser Klasse ist, sondern unter das Fixieren von Dimensionen (siehe unten) fällt.

2. **Fixieren** einer Dimension (engl.: section)
   Die Dimensionalität der Domäne verringert sich um 1, indem in einer Dimension eine Einschränkung auf einen konstanten Wert stattfindet:

   section $_{d, pos}$: <D, T> → <D', T> mit α → section(α, d, pos)
   mit dim(D') = dim(D) – 1. D'[k] = D[k] für k = 1, …, dim(D), k ≠ dim.
   Die Zellwerte selbst bleiben erhalten.

   Die Operation *section* entspricht dem Ausschneiden einer Hyperebene aus einem multidimensionalen Objekt. Als Parameter werden die zu schneidende Dimension *d* sowie die Position der Schnittebene *pos* angegeben. Eine verkürzende Schreibweise definiert die resultierende Domäne mit der Notation *α[..., pos, ...]*, wobei die Dimension *d* durch einen Wert *pos* statt einem Intervall gekennzeichnet ist, z.B. section$_{3, 0}$ = α[0:127, 0:63, 0]. Ein gleichzeitiges Fixieren mehrerer Dimensionen ist erlaubt und entspricht der Konkatenation der entsprechenden Operationen für die einzelnen Dimensionen. Ein Spezialfall hiervon ist ein **Punktzugriff**, wobei jede Dimension des Arrays fix gesetzt wird, d.h. dim(D') = 0. Die entsprechende Schreibweise ist *α[x]*, z.B. α[0,0,3].

3. **Verschieben** der Domäne (engl.: shift)
   Die Operation bewirkt eine Verschiebung der Domäne um einen gegebenen Punkt $\underline{x}$. Resultat der Operation ist ein MDD mit gleichem Inhalt und einer Domäne mit gleicher Dimensionalität und Ausdehnung. Lediglich die Lage der Domäne im multidimensionalen Raum ist verändert:

   shift$_{\underline{x}}$: <D, T> → <D', T> mit α → shift(α, $\underline{x}$),
   wobei low(D') = low(D) + $\underline{x}$ sowie high(D') = high(D) + $\underline{x}$
   Die Zellwerte selbst bleiben erhalten: $\forall\ \underline{y} \in$ D': shift(α, $\underline{x}$)[$\underline{y}$] = α[$\underline{y}$+$\underline{x}$].

   Diese Operation wird häufig in Zusammenhang mit induzierten Operationen (Kapitel 2.2.2.4) genutzt, da in binären induzierten Operationen (Verknüpfung zweier MDD) die Domänen identisch sein müssen.

In Abbildung 2.3 wird jede dieser drei Arten von geometrischer Operation an unserem Beispieldatenobjekt demonstriert. Aus dem 3-dimensionales Objekt α mit Domäne [0:127, 0:63, 0:119], d.h. extent(α) = [128, 64, 120], wird links in einer Trimming-Operation ein Bereich extrahiert. Die erste und dritte Dimension haben weiterhin die volle Ausdehnung, die zweite Dimension wird auf ein Intervall zwischen 0 und 31 begrenzt, was der Nordhalbkugel der Erdoberfläche entspricht. Folglich hat das Resultat die Ausdehnung extent(α) = [128, 32, 120].
In der Mitte ist das Resultat einer Section-Operation zu sehen, in welcher die zweite Dimension auf den Wert 31 fixiert wurde. Ergebnis ist ein 2-dimensionales MDD mit extent(α) = [128, 120]. Die Visualisierung zeigt, wie sich die Temperatur im Laufe von 10 Jahren (120 Monate) im Bereich des Äquators geändert hat.
Rechts sieht man das Resultat einer Shift-Operation. Das ursprüngliche MDD wurde im 3-D Raum um den Punkt [0, 0, 120] verschoben. Resultat ist ein MDD mit identischen Zellwerten, welches nun an einem anderen Ort im multidimensionalen Raum positioniert ist. Bezogen auf

die Semantik des Beispiels wurde die Zeitspanne des Datensatzes von den ersten 10 Jahren der Simulation (Monate 0 bis 119) in das zweite Jahrzehnt (Monate 120-239) versetzt. Häufigste Anwendung solcher Verschiebungen der Domäne sind induzierte Operationen, die eine Verknüpfung von zwei multidimensionalen Objekten nur erlauben, wenn diese eine identische Domäne haben (siehe Kapitel 2.2.2.4). In unserem Fall ist nun eine Verknüpfung (beispielsweise eine Berechnung von Differenzen) mit Objekten möglich, die das zweite Jahrzehnt einer Klimasimulation darstellen.

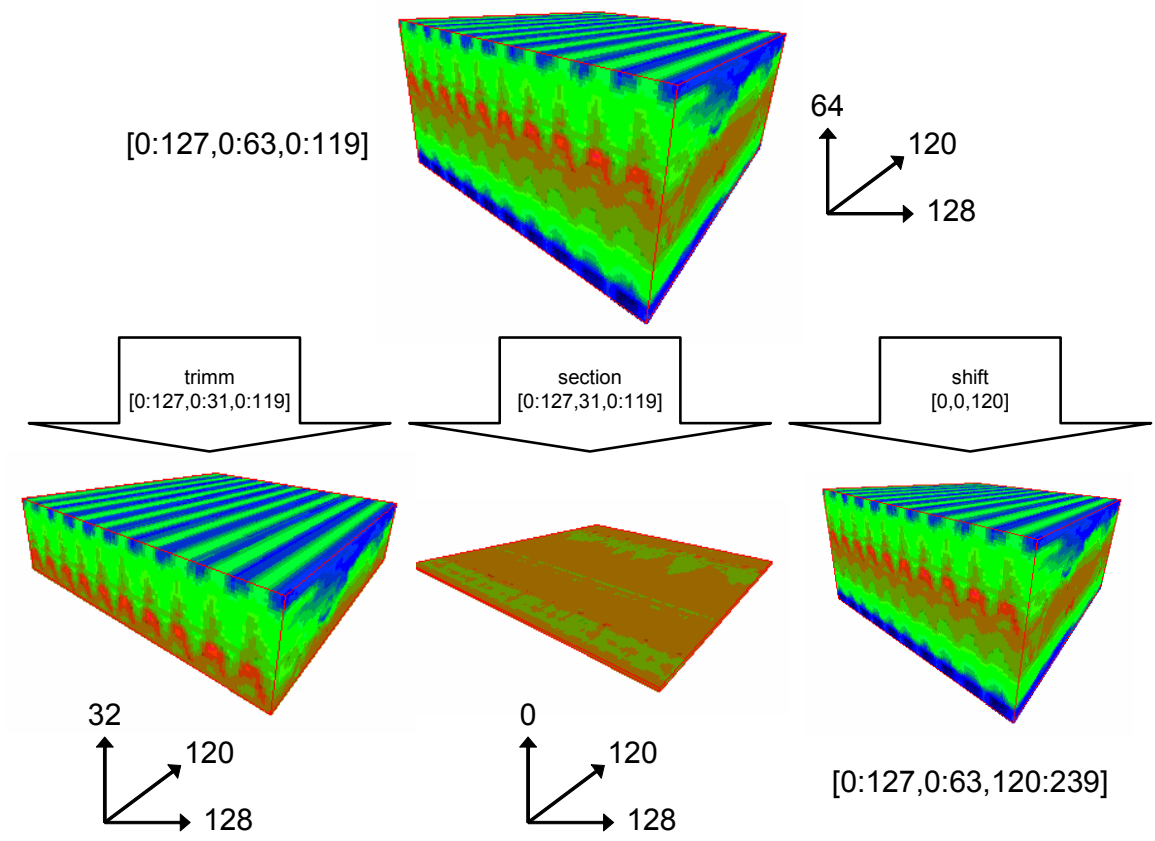


Abbildung 2.3: Beispiel für geometrische Operationen

2.2.2.4 Induzierte Operationen

Eine auf einem Basisdatentyp definierte Operation (unär oder binär) ist auch auf einem multidimensionalen Objekt mit diesem Basisdatentyp definiert. Die Operation wird dabei auf jede Zelle des (der) MDD angewendet, Resultat ist ein MDD mit identischer Domäne. Man unterscheidet zwischen **unär induzierten Operationen** und **binär induzierten Operationen**. Erste Klasse erfordert als Eingabe ein MDD und evtl. eine Konstante (die Operation selbst kann also binär sein). Binär induzierte Operationen hingegen verknüpfen zwei multidimensionale Objekte. Formal gesprochen, lassen sich induzierte Operationen folgendermaßen definieren.

1. **Unär induzierte Operationen**
   Sei $\oplus$ eine unäre oder binäre Operation, die auf jeder Zelle $\underline{x}$ des MDD α definiert ist:
   $\oplus$: $\alpha[\underline{x}] \rightarrow$ T' ist definiert für jede Zelle $\underline{x} \in \text{sdom}(\alpha)$, bzw.
   $\oplus$: $\alpha[\underline{x}] \times s \rightarrow$ T' ist definiert für jede Zelle $\underline{x} \in \text{sdom}(\alpha)$, s sei eine skalare Konstante.
   Dann ist die induzierte Operation auf dem MDD α ebenfalls definiert

$ind_\oplus$: <D, T> → <D, T'>, α → $ind_\oplus(\alpha)$; $\forall \underline{x} \in D$: $ind_\oplus(\alpha)[\underline{x}] = \oplus\, \alpha[\underline{x}]$ bzw.
$ind_\oplus$: <D, T> × s → <D, T'>, $\alpha \times s$ → $ind_\oplus(\alpha, s)$;
$\forall \underline{x} \in D$: $ind_\oplus(\alpha, s)[\underline{x}] = \alpha[\underline{x}] \oplus s$

Die Operation wird dabei auf jede Zelle des MDD α ausgeführt. Man beachte, dass die induzierte Operation nur definiert ist, wenn sie auf jeder Zelle definiert ist (nicht nur auf dem Basistyp). Beispielsweise ist die induzierte Operation *sqrt*, das heißt die Berechnung der Quadratwurzel, nicht auf einem MDD definiert, welches mindestens eine Zelle mit einem negativen Wert beinhaltet.

2. **Binär induzierte Operationen**
   Sei $\oplus$ eine binäre Operation, die auf den Zellen des MDD α und den entsprechenden Zellen des MDD β definiert ist:
   $\oplus$: $\alpha[\underline{x}] \oplus \beta[\underline{x}]$ → T' ist definiert für jede Zelle $\underline{x} \in sdom(\alpha) = sdom(\beta)$.
   Dann ist die binär induzierte Operation $\alpha \oplus \beta$ ebenfalls definiert:

   $ind_\oplus$: <D, T_1> × <D, T_2> → <D, T'>, $\alpha \times \beta$ → $ind_\oplus(\alpha, \beta)$;
   $\forall \underline{x} \in D$: $ind_\oplus(\alpha, \beta)[\underline{x}] = \alpha[\underline{x}] \oplus \beta[\underline{x}]$

   Die Operation wird dabei jeweils auf die Zellen mit identischen Positionen von α und β angewendet. Die Domänen D von α, β und dem Ergebnisarray sind identisch.

Beispiele für induzierte Operationen auf α, β mit $sdom(\alpha) = sdom(\beta) = [0:127, 0:63, 0:119]$ sind:

- unär ohne Konstante: Quadratwurzel
  α → sqrt(α): <D, T> → <D, T>[11]
- unär mit Konstante: Umwandlung einer Temperatur von Kelvin in ° Celsius
  α → α - 273.15: <D, T> × s → <D, T>
- binär: Differenz zwischen zwei Datensätzen α und β
  $\alpha \times \beta$ → α - β: <D, T> × <D, T> → <D, T>

mit jeweils D = [0:127, 0:63, 0:119], T = {float}. Es ist anzumerken, dass induzierte Operationen nicht auf Arrays mit strukturiertem Zelltyp definiert sind. Diese Arrays müssen zuvor mit einer Projektion auf dem Basistyp auf ein Array mit skalarem Zelltypen abgebildet werden.

2.2.2.5 Aggregationsoperationen

Die Klasse von **Aggregationsoperationen *agg*** aggregiert ein MDD zu einem MDD mit verringerter Domäne. Die Abgrenzung zu den geometrischen Operationen, welche ebenfalls die Domäne beeinflussen, besteht darin, dass Aggregationsoperationen die Zellwerte des Eingabearrays zur Berechnung des Resultats benötigen. Bei geometrischen Operationen dagegen werden die Zellwerte selbst immer unverändert übernommen, lediglich die Domäne wird verändert. Die Aggregationsoperationen lassen sich je nach Form des Resultats in Klassen einteilen:

[11] Die induzierte Funktion ist nur definiert, falls für alle $\underline{x} \in \alpha$ gilt: $\alpha[\underline{x}] \geq 0$

1. Aggregation eines Arrays zu einem **skalaren Wert**[12]

   agg: <D, T> → <D', T'>, mit dim(D') = 0.
   agg(α) = f(W), W = {α[x] | x ∈ α}.

   D.h. *agg* basiert auf einer Funktion, die als Eingabe die Zellwerte von α erhält. Die Aggregation eines Arrays ist definiert, wenn die Funktion f(W) auf allen Zellwerten W definiert ist[13]. Die Reihenfolge der Auswertung ist für die hier definierten Aggregationen nicht von Bedeutung, da für die Funktionen f(W) Kommutativität und Assoziativität gelten. In diese Klasse fallen folgende Funktionen: Minimum *min(α)*, Maximum *max(α)*, Summe *sum(α)*, arithmetischer Durchschnitt *avg(α)*, Anzahl der Zellen *count(α)*, Existenzquantor *exist(α)* und Allquantor *all(α)*.

2. Aggregation eines Arrays zu einem **Array der gleichen Dimensionalität und einer Ausdehnung von 1** in jeder Dimension

   agg: <D, T> → <D', T'>, mit dim(D') = dim(D) und extent(D') = 1.
   agg(α) = f(α).

   Diese Form der Aggregation wird benötigt, um nicht nur einen Extremwert zu erhalten, sondern auch dessen Position in der Domäne von α, also *max_pos(α) und min_pos(α)*.

3. Aggregation eines Arrays zu einem **Array der gleichen Dimensionalität und geringerer Ausdehnung**

   agg: <D, T> → <D', T>, mit dim(D') = dim(D) und extent(D') < extent(D).
   agg(α) = f(α).

   Darunter fällt die Skalierung *scale(α, s)* von multidimensionalen Arrays um den Faktor s, wobei ein Punkt des Ergebnisarrays durch eine Zusammenfassung von Punkten des Quellarrays berechnet wird.

4. Aggregation eines Arrays zu einem **Array niedrigerer Dimensionalität**

   agg: <D, T> → <D', T'>, mit dim(D') < dim(D).
   agg(α) = f(α).

   Adaption des Datenformats (für die Ausgabe) lassen sich so beschreiben. Diese erzeugen in der Regel ein 1-dimensionales Array, welches die Formatierung in das Ausgabeformat realisiert. Bekannte Konvertierungsfunktionen sind *jpeg(α)*, *bmp(α)*, *hdf(α)*, etc. Das Ausgabeformat erzwingt mitunter, eine bestimmte Dimensionalität des Eingabearrays, etwas *jpeg(α)* → dim(α) = 2.

Aggregationsoperationen sind typisch für die Analyse von großen Rasterdaten, da hiermit Daten effizient untersucht werden können, bevor das komplette Objekt (oft über ein langsames Netzwerk) zum Client übertragen wird.

[12] Ein skalarer Wert kann in einem abgeschlossenen multidimensionalen Datenmodell als Array mit Dimensionalität 0 gesehen werden. Wir werden im Folgenden diese Bezeichnungen als synonym betrachten.

[13] So ist beispielsweise max(α) nicht definiert, wenn der Datentyp zusammengesetzt ist (z.B. base(α) = {float, float}), da ein Maximum im Sinne eines dominierenden Wertes nicht vorhanden sein muss.

2.3 Einbettung in das relationale Modell

Ein Array DBMS erweitert das relationale Modell um den Datentyp MDD bzw. die Anfragesprache SQL um Operationen für multidimensionale Objekte. Mit anderen Worten wird die Semantik für multidimensionale Objekte direkt in das Datenbanksystem und die Anfragesprache integriert. Auswirkungen auf das Datenmodell und die Erweiterung der Anfragesprache werden in Kapitel 2.3.1 behandelt. Eine nahtlose Integration in das relationale Modell ist im Array DBMS rasdaman indes nicht gelungen. Auf Ursachen und Gründe dafür wird in Kapitel 2.3.2 näher eingegangen.

2.3.1 Der Datentyp MDD für multidimensionale Arrays

Eine Relation R ist definiert als eine Teilmenge des kartesischen Produkts (Kreuzprodukt) von n Domänen: $R \subseteq D_1 \times \ldots \times D_n$. Die Attribute dieser Domänen enthalten atomare Werte, das heißt Mengen sind als Wert nicht erlaubt. Die Erweiterung des relationalen Modells um multidimensionale Array Daten erlaubt neben üblichen Domänen wie Zahlen, Zeichenketten, etc. auch die Speicherung von multidimensionalen Objekten. Der Wertebereich für diese Domäne entspricht dabei dem Typ der MDD und beinhaltet das multidimensionale Intervall und den Basistyp. Die Umsetzung der Speicherung geschieht wie für andere Daten auch auf der physischen Ebene des DBMS und ist dem Benutzer in der Regel verborgen, muss jedoch intern vom DBMS unterstützt werden. Das DBMS muss somit um folgende Module erweitert werden:

1. Effiziente Speicherung der Array Daten und effizienter Zugriff
   Eine effiziente Speicherung von Rasterdaten erfolgt meist durch so genannte Kachelung engl.: tiling bzw. chunking) [Fur99] [SS94] [Ols92]. Die Daten werden hierzu in multidimensionale Kacheln (engl.: tiles) aufgeteilt, die zusammen das komplette Objekt bilden. Der Zugriff auf Teilbereiche der Daten (und somit auf Kacheln) erfolgt intern mit Hilfe eines multidimensionalen Index, beispielsweise über einen R-Baum [GG98]. Das physikalische Datenmodell von rasdaman wird in Kapitel 2.4 näher beschrieben.
2. Erweiterung von Datendefinitionssprache (DDL) und Datenmanipulationssprache (DML)
   Die Sprache muss um die Definition neuer Basistypen, multidimensionaler Intervalle und somit MDD erweitert werden [Bau99]. Neue multidimensionale Operationen wie in Kapitel 2.2.2 beschrieben müssen in die Anfragesprache integriert werden. Ferner müssen relationale Operationen dahingehend untersucht werden, ob ihr Einsatz für multidimensionale Objekte Sinn macht. Beispielsweise ist eine (relationale) Selektion von MDD aus einer Menge von MDD durchaus anwendbar während etwa ein Join von MDD näher zu untersuchen ist.
3. Optimierung der Anfrageausführung für multidimensionale Operationen
   Die Anfrageoptimierung muss um eine spezielle multidimensionale Optimierung erweitert werden. rasdaman beinhaltet eine Anfrageoptimierung auf Basis von Heuristiken. Eine detaillierte Beschreibung hierzu findet sich in [Rit99].

In rasdaman implementiert sind lediglich Relationen von MDD mit einem einzigen Attribut multidimensionalen Typs und keinen relationalen Attributen: $R \subseteq D_1$ mit $D_1 = <D, T>$. Wir sprechen in diesem Fall von einer Kollektion von MDD.

Definition 2.5 Kollektion C von MDD

Eine Kollektion C ist eine Relation mit einem Attribut des Typs MDD. Somit ist C eine ungeordnete Menge von MDD mit identischem Intervall D und Zelltyp T:
$C \subset \{\alpha \mid type(\alpha) = <D, T>\}$.

Wiederum werden abkürzend verwendet: sdom(C) = sdom(α) mit $\alpha \in C$, type(C) = type(α) mit $\alpha \in C$, etc.

Das Array DBMS **rasdaman** definiert vier (quasi-)relationale Operationen auf einer Kollektion von multidimensionalen Objekten:

1. **Zugriff ω** auf eine Kollektion
   Ergibt alle Elemente einer Kollektion C: $\omega(C) := \{ \alpha \mid \alpha \in R \}$. Diese Operation entspricht dem relationalen *scan* Operator.
2. **Repartitionierung ϕ** führt zu einer Umstrukturierung der Kollektion. Typischerweise wird dieser Operator auf ein einzelnes MDD α einer Kollektion C angewandt, um daraus eine Menge von i disjunkten MDD β_i mit Teilintervallen der Ausdehnung D' zu erstellen[14]:
   $\phi(C, \alpha, D') := \{ \beta_i \mid extent(sdom(\beta_i)) = extent(D') \wedge \beta_i[\underline{x}] = \alpha[\underline{x}]$ für alle $\underline{x} \in \beta_i\}$
   mit $\cap(D_i') = \varnothing$. $\cup(D_i') \neq sdom(\alpha)$, da „Randbereiche" nicht in $\phi(C, \alpha, D')$ liegen.
3. **Kreuzprodukt ×** liefert das kartesische Produkt von Kollektionen. Im binären Fall ist dies $\times(C_1,C_2) := \{ (\alpha_i,\beta_j) \mid \alpha_i \in C_1 \wedge \beta_j \in C_2 \}$
4. **Selektion σ_{cond}** überprüft ein Prädikat *cond* auf alle Objekte einer Kollektion C. Nur die MDD α, für die das Prädikat erfüllt ist, werden zurückgegeben: $\sigma_{cond}(C) := \{ \alpha \mid \alpha \in C, cond(\alpha) \}$
5. **Applikation α_{op}**. Eine Applikation α, bestehend aus Array Operationen, wird auf alle Elemente α der Kollektion C ausgeführt[15] und liefert ein MDD Ergebnis β oder ein skalares Ergebnis s:
   $\alpha_{op}(C) := \{ \beta \mid \beta = op(\alpha), \alpha \in C \} \vee \{ s \mid s = op(\alpha), s \in S, \alpha \in C \}$

Zur Verdeutlichung zeigen wir jeweils ein Beispiel: die Kollektion C_1 besitze vier 3D Objekte, wobei jedes Objekt einem Jahrzehnt einer Klimasimulation entspricht (siehe Kapitel 2.1). Die Dimensionen entsprechen geografischer Länge, geografischer Breite und Zeit (120 Monate, das heißt 10 Jahre). Die Kollektion C_2 beinhaltet zwei 3D Objekte, beispielsweise Vergleichswerte zur Verknüpfung. Die Datentypen seien
$type(C_1) = type(C_2) = <[0:127, 0:63, 0:119], \{float; \{int; int; int\}\}>$.

- $\omega(C_1)$ ergibt vier MDD: $\alpha_1, \ldots, \alpha_4$
- $\omega(C_2)$ liefert zwei MDD: β_1, β_2

[14] Im Prinzip dürfen die Domänen D' der Teilintervalle verschieden sein. Für die Implementierung in **rasdaman** werden jedoch nur identische D' zugelassen, das heißt Randbereiche mit kleinerer Domäne werden fallengelassen. Grund hierfür ist einerseits, dass einerseits nur so binär induzierte Operationen auf den resultierenden Arrays möglich sind und andererseits Aggregationsoperationen auf den Arrays vergleichbar sind.

[15] Das MDD α und die Operation α_{op} verwenden lediglich aus historischen Gründen eine ähnliche Notation.

- $\phi(C_1, \alpha_2, D'=[0:127, 0:63, 0:11])$ repartitioniert Objekt α_2 der Kollektion C_1 in zehn Objekte der Ausdehnung D'[16]. In diesem Fall entsprechen die Partitionen den Jahren der Klimasimulation. Diese Jahresdaten können so weiter analysiert werden.
- $\times(C_1,C_2)$ resultiert in insgesamt acht 2-Tupel von MDD aus den Kollektionen C_1 und C_2. Ein Kreuzprodukt kann dazu genutzt werden, Daten aus verschiedenen Kollektionen zu verknüpfen. In diesem Fall können beispielsweise Differenzen von C_1 zu den zwei Vergleichsobjekten der Kollektion C_2 mittels induziertem Minus analysiert werden.
- $\sigma_{cond}(C_1)$ mit cond = (avg(α.temperatur) > 20) selektiert aus C_1 die Objekte, deren Kanal „Temperatur" einen Durchschnittswert größer als 20 hat.
- $\alpha_{op}(C_1)$ mit op = (jpeg(α.temperatur[*:*,*:*,0])) liefert aus jeder der vier Objekte von C_1 die Temperaturwerte des ersten Monats (Einschränkung der dritten Dimension auf den Wert 0 ergibt vier 2D Objekte) und wandelt die resultierenden MDD jeweils in ein JPEG Bild um.

2.3.2 Einschränkungen und Probleme

rasdaman, die einzig uns bekannte Implementierung eines Array DBMS für multidimensionale Daten beliebiger Dimensionalität und Größe, wurde nicht als Erweiterung eines bestehenden relationalen DBMS entwickelt, sondern ist eine eigenständige Implementierung, die ein relationales DBMS zur Datenhaltung und zur Transaktionssicherung nutzt. Die Trennung von bestehendem RDBMS (es werden Oracle 8i und 9i, IBM DB2 sowie Informix unterstützt, die Portierung auf weitere Systeme ist einfach realisierbar) und multidimensionaler Schicht (**rasdaman** Server) bietet hierbei den Vorteil, dass vorhandene Datenbanksysteme einfach für eine Speicherung und Analyse von Array Daten erweitert werden können. Diese Trennung resultiert aber auch in vielen Problemen und Einschränkungen, vor allem des logischen Datenmodells, aber auch der Anfragesprache:

1. Die Trägermenge des Datenmodells beschränkt sich auf Mengen von multidimensionalen Objekten (und wenige skalare Datentypen). Relationale Daten (im Gegensatz zu multidimensionalen Objekten) in der Datenbasis können nicht direkt geschrieben oder angefragt werden. Mit anderen Worten unterstützt **rasdaman** nur Kollektionen von MDD und nicht den Datentyp MDD in Relationen generell.
2. Relationale Operationen werden nur rudimentär unterstützt. Lediglich Kreuzprodukt, relationale Selektion, also hier eine Selektion von multidimensionalen Objekten aus einer Menge von solchen, und eine an multidimensionale Objekte angepasste Projektion, sowie die Repartitionierung werden angeboten. Weitere relationale Operationen wie Sortierung, Gruppierung, etc. werden nicht unterstützt.

In der Praxis ergeben sich dadurch vor allem Probleme, falls die multidimensionalen Daten durch Metadaten ergänzt werden sollen. Speicherungsdatum des Objekts, Beschreibung, verantwortliche Person, Zusatzdaten wie Durchschnittswerte, Maximalwerte, etc, können nicht direkt mittels **rasdaman** bzw. Anfragesprache manipuliert werden.

Im Weiteren konzentrieren wir uns auf das in Kapitel 2.2 beschriebene multidimensionale Datenmodell und die eingeschränkte relationale Einbettung aus Kapitel 2.3.1, das heißt die Speicherung und Manipulation von Mengen von multidimensionalen Objekten, und werden die konzeptionelle Erweiterung des relationalen Modells nicht weiter ins Auge fassen.

[16] Die Domäne D' der Resultate bezieht sich immer auf die Domäne des Eingabearrays. In diesem Fall bildet D' = [0:127, 0:63, 0:11] aus dem Eingabearray mit D = [0:127, 0:63, 0:119] Ergebnisse mit Domänen [0:127, 0:64, 0:11] für das erste Jahr, [0:127, 0:63, 12:23] für das zweite Jahr, etc.

2.4 Physikalisches Datenmodell

Die physikalische Ebene eines DBMS sichert (dem Benutzer verborgen) die effektive Speicherung und das effektive Lokalisieren und Laden von Daten. Die Speicherung von multidimensionalen Daten in relationalen Datenbanksystemen wurde insbesondere für Data Warehouse Systeme eingehend untersucht. In diesem Fall werden dünn besetzte multidimensionale Datenräume (engl.: data cubes) in vektorisierter Form gespeichert, d.h. unter Speicherung von Koordinaten und Datenwerten in einem relationalen DBMS abgelegt. Das DBMS wird lediglich zur Datenspeicherung eingesetzt, das multidimensionale Datenmodell selbst ist dem RDBMS unbekannt. Multidimensionale Operationen (OLAP Operationen wie dicing, slicing, roll-up, drill-down, etc.) werden von einer OLAP Schicht interpretiert und die benötigten Daten mittels SQL aus der Datenbasis geladen und gegebenenfalls grafisch aufbereitet.

c1	c2	c3	temperatur	R	G	B
0	0	0	264,86452	55	205	12
0	0	1	264,84633	55	204	12
0	0	2	263,89762	54	205	11
0	0	3	264,63541	53	205	12
0	0	4	265,27384	53	205	12
0	0	5	264,19776	59	200	10
0	0	6	264,63747	60	200	10
0	0	7	266,56545	52	203	17

Koordinaten Werte

Abbildung 2.4: Speicherung von 3D Daten in einem RDBMS als Vektordaten

Multidimensionale Rasterdaten hingegen werden in der Regel nicht als Vektoren gespeichert, da für dicht besetzte Räume der Speicherbedarf für Koordinaten einen Großteil des gesamten Speicherbedarfs einnimmt (Abbildung 2.4), sondern als **multidimensionale Arrays**, das heißt die Position eines Punktes im Objekt wird bestimmt durch eine Linearisierungsfunktion [Deh02]. Eine effektive Speicherung und vor allem effektives Laden von Sekundärspeicher bzw. Tertiärspeicher erfordert darüber hinaus das Aufteilen großer Objekte in so genannten Kacheln (engl. Tiles), womit ein selektives Laden der Daten bei Bereichsanfragen gesichert wird.

Definition 2.6: Kachelung eines MDD α

Die Kachelung eines MDD α ist definiert als eine Menge von MDD τ_i mit folgenden Eigenschaften:

1. $\bigcup_{i=1}^{n} \text{sdom}(\tau_i) = \text{sdom}(\alpha)$
2. $\text{sdom}(\tau_i) \cap \text{sdom}(\tau_j) = \varnothing$ für alle $i, j \in \{1, \ldots, n\}, i \neq j$

Die Teilbereiche τ_i des MDD α werden als Kacheln (engl.: tiles) bezeichnet.

Mit anderen Worten ergeben alle Kacheln das komplette Objekt, wobei keine Bereiche doppelt gespeichert werden. Kacheln sind selbst wiederum MDD, das heißt multidimensionale Quader.

Abbildung 2.5 zeigt ein Beispiel für ein 2D Objekt α mit sdom(α) = [0:67, 0:65], das in 49 Kacheln gespeichert wird. Die Kacheln haben eine Ausdehnung von 10 in jeder der zwei Dimensionen (beinhalten also 100 Zellen), lediglich am Rand des Objekts verbleiben Kacheln von geringerer Ausdehnung. Eine Bereichsanfrage, etwa für den Bereich [16:58,18:52] lädt alle Kacheln, die Daten aus dem angefragten Bereich enthalten, im Beispiel sind das 25. Typischerweise existiert ein Verschnitt, das heißt die Kacheln geben die Granularität für Ladeoperationen vor, Teilbereiche von Kacheln können nicht geladen werden.

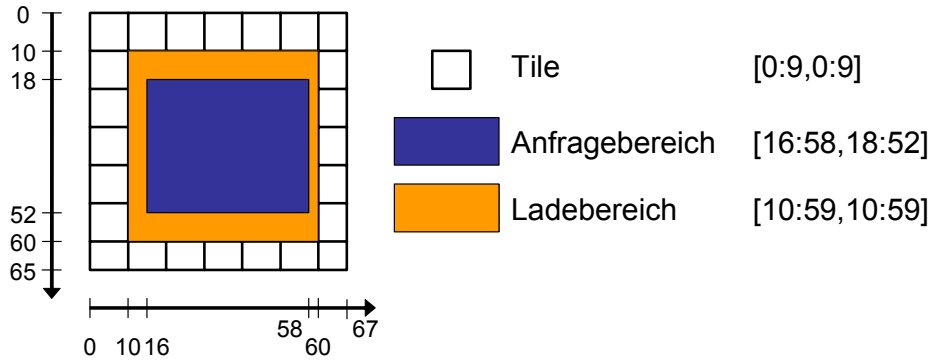


Abbildung 2.5: Speicherung von 2D Daten als Array

Größe und Lage der einzelnen Kacheln τ im Objekt, also die **Kachelungsmethode**, lassen sich variieren, meist erfolgt eine manuelle Anpassung an die zu erwartenden Anfragen, in seltenen Fällen eine automatische Anpassung.

Definition 2.7: Kachelungsmethode eines MDD

1. arbiträr: für die Kachelung gelten die Eigenschaften aus Definition 2.6
2. ausgerichtet: zusätzlich zu 1. gilt extent(τ_i) = extent(τ_j) für i, j $\in$ {1, …, n}. Ausnahmen sind möglich, falls high(τ_i)[k] = high(α) [k], k $\in$ {1, …, d}, d.h. für Kacheln am „oberen Rand“ der Domäne des MDD α[17].
3. regulär: zusätzlich zu 2. gilt: extent(τ_i) [k] = extent(τ_i) [l], für alle i $\in$ {1, …, n}, für alle k, l $\in$ {1, …, d}. Auch hier gilt die Ausnahme für Randbereiche aus 2.

In Abbildung 2.6, links, ist **reguläre Kachelung** skizziert. Die Kacheln haben in jeder Dimension dieselbe Ausdehnung (außer Randkacheln). Diese Art der Kachelung ist sinnvoll, falls Anfragen keine Dimension bevorzugen bzw. wenn keine Aussagen über zu erwartende

[17] Gilt etwa in Abbildung 2.5 die Kachel mit sdom(τ) = [60:67,0:9]. extent(τ) $\neq$ [10, 10] ist erlaubt, da obere Begrenzung von Kachel und Array in der ersten Dimension identisch sind: high(τ) [0] = high(α) [0] = 67.

Anfragen gemacht werden können. Die Größe der Kacheln wird im Hinblick auf typische Anfragebereiche gewählt. Je größer die Kacheln, desto schneller erfolgen in der Regel Zugriffe auf große Anfragebereiche, wohingegen der Verschnitt zunimmt.

Das mittlere Objekt aus Abbildung 2.6 zeigt eine **ausgerichtete Kachelungsstrategie**. Werden Anfragen erwartet, welche die Dimensionen unterschiedlich einschränken, können mit dieser Kachelung Verschnitt reduziert sowie Performanz gesteigert werden. Als Beispiel seien hier Satellitenbilder genannt, die zusätzlich mit einer Zeitdimension als 3D Objekt modelliert werden. Werden zu einem Großteil Anfragen erwartet, die eine Einschränkung auf einen bestimmten Zeitpunkt vornehmen, sollte die Kachelung in der Zeitdimension eine geringe Ausdehnung zeigen. Nichtsdestotrotz bleiben Auswertungen, die eine zeitliche Analyse vornehmen, das heißt die Zeitdimension nicht einschränken, weiterhin effektiv möglich.

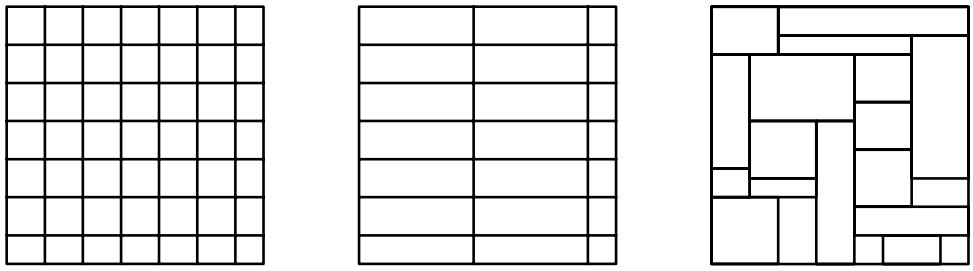


Abbildung 2.6: Kachelung von MDD: regulär, ausgerichtet, arbiträr

Eine **arbiträre Kachelung** (Abbildung 2.6, rechts) erlaubt Kacheln verschiedener Größe. Dies erfolgt meist dann, wenn eine automatisierte Anpassung der Kachelung an die Anfragen gewährleistet werden soll. Hierbei resultieren kleine Anfragebereiche in Kacheln geringer Größe während große Anfragebereiche diese Kacheln gegebenenfalls wieder zusammenfassen.

Eine Zwischenform zwischen arbiträrer und ausgerichteter Kachelung ist die **irreguläre Kachelung**. Die Kacheln sind hierbei nicht von gleicher Größe, jedoch in den Dimensionen ausgerichtet:

$\forall\ \tau_i, \tau_j \in \alpha$: $low(\tau_i[d]) = low(\tau_j[d]) \Rightarrow extent(\tau_i[d]) = extent(\tau_j[d]), d \in 1, \ldots, dim(\alpha), i \neq j$.

Kachelungsstrategien, deren Implementierung und Performanz für Array Daten, sind detailliert in [Fur99] beschrieben. In der Praxis kann die Speicherung der Kacheln beispielsweise als Binärobjekte in einem relationalen DBMS erfolgen (vgl. Kapitel 2.5) und der effiziente Zugriff mittels eines multidimensionalen Index [GG98] wie etwa dem UB-Baum [Bay97] oder dem R^+-Baum [SRF87].

Abschließend sei der Speicherort für Kacheln definiert. Kacheln können dauerhaft auf Sekundärspeicher gespeichert werden, oder insbesondere für Anfragen in den Hauptspeicher geladen und evtl. verändert werden.

Definition 2.8: Persistente und transiente Kacheln

Eine Kachel ist persistent (dauerhaft), wenn sie auf Sekundärspeicher (RDBMS) gesichert wurde. Befindet sich die Kachel dagegen im Hauptspeicher, wird sie transient genannt. Kacheln können transient sein, wenn sie noch nicht auf Sekundärspeicher geschrieben wurden, oder wenn sie im Rahmen von Anfragen in den Hauptspeicher geladen wurden (Anwendung von Operatoren oder Vorbereitung zum Transfer).

2.5 Das Array DBMS rasdaman

Das Array DBMS **rasdaman** ist das einzig uns bekannte DBMS für Rasterdaten beliebiger Dimensionalität und Größe. Es wird von der rasdaman GmbH vertrieben und wurde im Rahmen des EU Projektes ESTEDI (siehe Kapitel 1.4) bezüglich paralleler Anfrageverarbeitung erweitert. Somit bildet **rasdaman** in Version 5.0 die Basis für die Implementierung der in dieser Arbeit beschriebenen Algorithmen.
Nach einer Beschreibung der **rasdaman** Architektur und der Komponenten im nächsten Kapitel werden Datenmodell, Anfragesprache sowie die Umsetzung der Anfragen mittels Anfragebaum und Anfrageverarbeitung skizziert.

2.5.1 Architektur

rasdaman ist ein Client-Server System und zeigt sich als 4-Schichten Architektur (Abbildung 2.7)[18]: die unterste Schicht dient der Speicherung und Transaktionsverwaltung (engl.: *Storage, Transaction*, TA). Hierzu dient ein konventionelles relationales DBMS, derzeit werden IBM DB2, Oracle oder Informix unterstützt. Die Verwendung dieser ausgereiften Technologie erlaubt die effiziente Speicherung von großen Datenvolumina, die auf Sekundärspeicher (Festplatten) oder Tertiärspeicher (Magnetbänder, DVD, etc.) abgelegt sein können. Transaktionen in **rasdaman** werden direkt auf die entsprechende Transaktionsverwaltung der relationalen Datenbank abgebildet.

Die zweite Schicht bildet der **rasdaman** Server, selbst in Schichten aufgeteilt und bestehend aus einer Vielzahl von Modulen, welche Funktionalitäten kapseln. Eine Schnittstelle zur relationalen Datenbank (engl.: *Base DBMS Interface*) erledigt die Kommunikation zwischen **rasdaman** und RDBMS mittels EQSL. Die Kapselung dieser Schnittstelle erlaubt die schnelle Integration neuer RDBMS in die **rasdaman** Architektur. Darüber liegen Module für die Kompression der Datenbasis (engl.: *Storage Compression*) und für den Zugriff auf Tertiärspeicher (engl.: *Tertiary Storage Manager*). Multidimensionale Indizierung (engl.: *Index*), Verwaltung von Kacheln (engl.: *Tile Manager*) sowie Kataloginformationen (engl.: *Catalog*) zur Verwaltung von Metadaten wie etwa Typinformationen liegen in einer weiteren Schicht. Die Anfrageverarbeitung wird durch einen Parser für die Anfragesprache (engl.: *Query Language Parser*), einen Optimierer (engl.: *Optimizer*) und ein Modul zur Ausführung der Anfragen (engl.: *Executor*) realisiert. Die parallele Verarbeitung von Anfragen wird ebenfalls in dieser Schicht realisiert. Ein Parallelisierungsmodul (engl.: *Parallelizer*) adaptiert die Anfragebäume und bereitet so die parallele Verarbeitung vor. Das Ausführungsmodul (engl.: *Executor*) muss für die Parallelverarbeitung angepasst werden. Die Schnittstelle zu den Clients bilden ein Modul zur Kommunikation mit den Client Applikationen (engl.: *Server Communication*) mittels

[18] In der Abbildung werden die englischen Namen für die Module verwendet, da dies eine bessere Identifikation in der Implementierung ermöglicht. Die englischen Begriffe werden im Folgenden ins Deutsche übersetzt.

RPC oder HTTP sowie ein Modul, das die Kompression der zu übertragenden Daten vornehmen kann (engl.: *Transfer Compression*). Ferner kommuniziert der Server mit einem rasdaman Manager (engl.: *Manager Communication*).

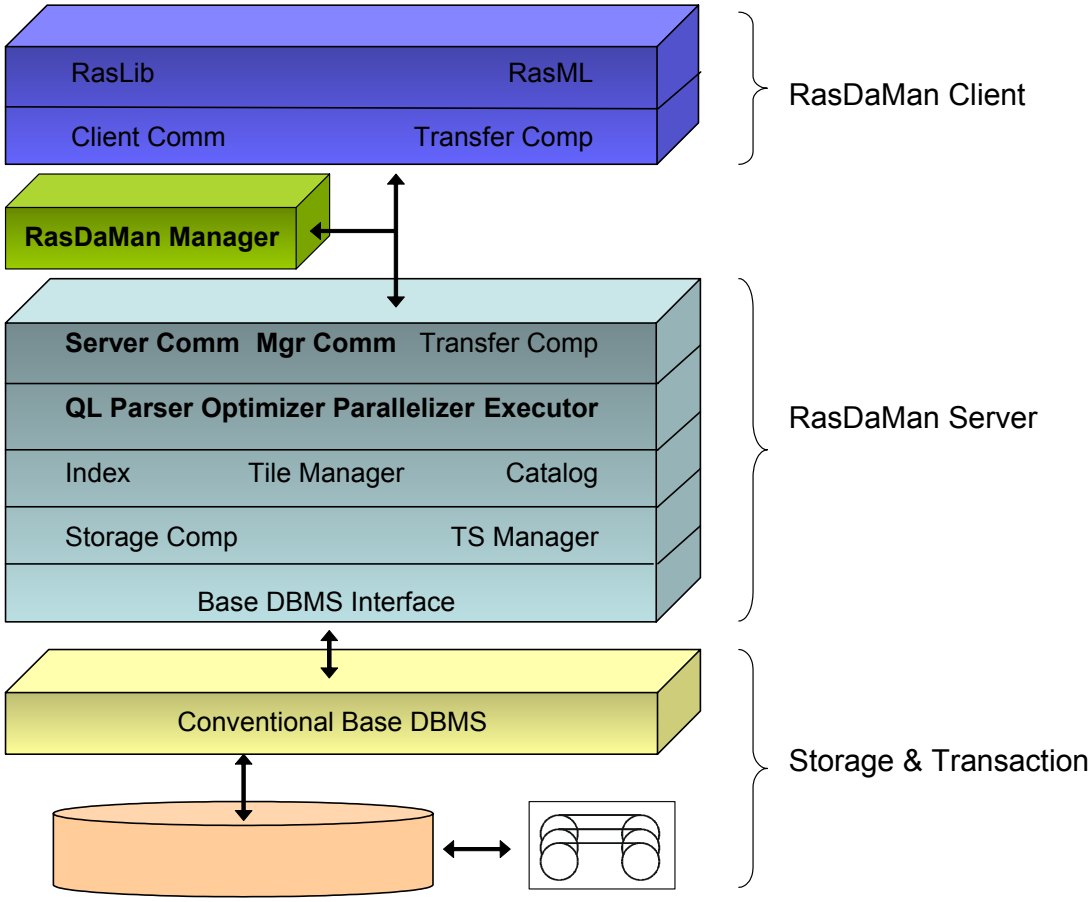


Abbildung 2.7: Architektur des rasdaman DBMS

Dieser rasdaman Manager bildet eine Art Zwischenschicht zwischen Client und Server. Aufgaben wie Sicherheitsfunktionen, Authentifizierung, Mehrbenutzerbetrieb durch Zuteilung von Anfragen auf verschiedene rasdaman Server werden hier gesichert. Eine Applikation wendet sich anfangs stets an den rasdaman Manager, der dann eine Kommunikation zwischen einem Server und der Applikation etabliert.

Die vierte Schicht schließlich ist die Client Schicht, das heißt die Schicht für Anwendungen. Neben Modulen zur Kommunikation mit dem Server bzw. dem Manager (engl.: *Client Communication*) und der (De-)Kompression (engl.: *Transfer Compression*) der empfangenen Daten finden sich hier Programmierschnittstellen (rasdaman Library *RasLib*, rasdaman Modification Language *RasML*).

2.5.2 Anfragesprache, Anfragebaum, Anfrageverarbeitung

Die in rasdaman verwendete Anfragesprache RasQL basiert auf SQL92 und wurde speziell für multidimensionale Operationen entwickelt [Bau99]. Befehle zur Datendefinition (engl.: DDL, Database Definition Language) erlauben die Erzeugung eigener Basistypen, eigener Array Typen sowie Kollektionen. Das Einfügen und Modifizieren der Daten (engl.: DML, Database Manipulation Language) ist prinzipiell möglich, wird aber aufgrund der Komplexi-

tät der Daten meist nicht per Anfragesprache, sondern mit speziellen Methoden der API durch eigens entwickelte Einfügeprogramme realisiert. Für die Datenanalyse stehen eine Vielzahl von Sprachkonstrukten bereit, die sich an den multidimensionalen Operationen in Kapitel 2.2.2 orientieren. Die relationalen Operationen für die Analyse von Kollektionen wurden bereits in Kapitel 2.3 vorgestellt.

Die Umsetzung der Anfragesprache auf einen Anfragebaum sowie dessen Optimierung und Ausführung wird hier an einem Beispiel kurz erläutert. Eine detaillierte Sprachbeschreibung ist in [Bau99] sowie [AK01a] zu finden. Anfragebaum, Optimierung und Anfrageverarbeitung werden in aller Tiefe in [Rit99] dargestellt. Eine RasQL Anfrage ist prinzipiell von folgender Form:

```
SELECT <operation>
FROM <collection> [ AS <name> ]
[ WHERE <condition> ]
```

Ähnlich wie in SQL spezifiziert hier der FROM Teil die Kollektionen für die Anfrage. Es können beliebig viele Kollektionen referenziert werden und einzelne Kollektionen können repartitioniert werden. Im optionalen WHERE Teil kann eine Bedingung (engl.: condition) für eine Datenselektion angegeben werden. Dieses Prädikat besteht aus multidimensionalen Operationen und muss als Ergebnis einen booleschen Wert ergeben. Mittels SELECT können die Daten weiteren Operationen unterzogen werden.

Die Analyse einer Anfrage beurteilt häufig die Kardinalität, d.h. welche Anzahl an Arrays müssen für die Verarbeitung einer Anfrage in den Speicher geladen werden:

Definition 2.9: Kardinalität einer Anfrage

Die Kardinalität einer Anfrage spezifiziert die Anzahl der multidimensionalen Objekte, die für die Anfrage verarbeitet werden. Hierbei werden logische MDD (das heißt auch während der Anfrage generierte) betrachtet, die in einen transienten Zustand übergehen, d.h. deren Kacheln in den Hauptspeicher geladen wurden.

Sei n eine Anzahl der in der Anfrage referenzierten Kollektionen mit i = 1, …, n:

1. Wird eine einzelne Kollektion C_i nicht durch die Operationen aus Punkt 2 und 3 verändert, ist die Kardinalität der Kollektion unverändert: $card(C_i) = |C_i|$.

2. Wird eine referenzierte Kollektion per OID (engl.: object identifier, Identifikator für MDD) in der Selektion eingeschränkt, so wird auf ein einzelnes Objekt zugegriffen. Die Kardinalität der Kollektion ist somit $card(C_i) = 1$.

3. Ein Repartitionierungsoperator $\phi(C_i, \alpha, D')$ verändert die Kardinalität der Kollektion C_i. Es wird ein MDD α aus der Kollektion selektiert und aus diesem wird eine Anzahl neuer MDD geschaffen. Sei $sdom(\alpha) = D$ die Ausdehnung der Objekte in der Kollektion C, so ergibt sich als Kardinalität nach einer Repartitionierung des selek-

tierten MDD

$$\mathrm{card}(C_i) = \prod_{k=0}^{d} \lfloor \mathrm{extent}(D)[k] / \mathrm{extent}(D')[k] \rfloor$$ [19].

Die Kardinalität der Anfrage Q gesamt ist $\mathrm{card}(Q) = \prod_{i=1}^{n} \mathrm{card}(C_i)$.

Es sei angemerkt, dass die Kardinalität der Anfrage nicht mit der Kardinalität der Ergebnismenge übereinstimmen muss.

Wir demonstrieren eine typische Anfrage anhand eines einfachen Beispiels:

```
SELECT jpg(m * a[20:420, *:*,sdom(a)[2].low])
FROM mask400_200 AS m, SLAY(mpim3d,1025) SLICES(2:1) AS a
WHERE avg_cells(a) > 25.0
```

Diese Anfrage wird intern in den in Abbildung 2.8 skizzierten Anfragebaum aufgelöst. Aus der Kollektion mpim3d, welche eine Menge von 3-dimensionalen Arrays mit Domäne [0:127, 0:63, 0:119] enthält, wird durch den Repartitionierungsoperator ϕ (engl.: slay: schlachten; engl.: slices: Scheiben) ein eindeutiges Objekt (mit dem Objektidentifikator 1025) selektiert und eine Menge von 3D Objekten zurückgegeben, indem Scheiben der Ausdehnung 1 orthogonal zur Dimension 2 geschnitten werden [20]. Das Resultat der Repartitionierung sind folglich Arrays mit Domäne [0:127, 0:63, 0:0], [0:127, 0:63, 1:1], etc[21]. Aus diesen 3D Objekten werden jetzt jene selektiert, welche einen durchschnittlichen Zellwert größer als 25.0 besitzen. Die resultierenden Objekte werden geometrisch eingeschränkt (Trimming und Fixierung der 3. Dimension auf unteren Grenzwert der Domäne), mit einer Maske durch induzierte Multiplikation verknüpft und in ein JPEG Bild umgewandelt.

Der Anfragebaum zerfällt in zwei grundlegende Arten von Operatoren, nämlich (quasi-) relationalen Iteratoren (α, σ, $\times$, ϕ, ω), welche eine Menge von multidimensionalen Objekten oder Tupel von diesen iterieren, und Array Operationen, die auf multidimensionalen Objekten definiert sind. Die Verarbeitungsstrategie erfolgt bei den Iteratoren, indem die Elemente nacheinander durch den Baum gereicht werden. Mit dieser Strategie kann der Ressourcenverbrauch, insbesondere Hauptspeicher, minimiert werden [Gra93]. Die multidimensionalen Operatoren (in unserem Beispiel *avg*, >, *, *jpg*) folgen nicht diesem Iteratorkonzept, sondern geben Ergebnisse komplett zurück. Die Teilbäume, die diese Array Operationen beinhalten, werden als Operatorbäume (oder Prädikatbäume) bezeichnet (in Abbildung 2.8 hinterlegt dargestellt). Insbesondere zwei Iteratoren besitzen komplexe Operatorbäume, nämlich Selektion σ und Applikation α.

rasdaman unterstützt eine Kachelung der multidimensionalen Daten wie in Kapitel 2.4 beschrieben. Die einzelnen Kacheln werden als BLOB (engl.: binary large objects) im relationalen DBMS abgelegt, wobei eine Kachel typischerweise eine Größe von mindestens 256K hat.

---

[19] Die untere Schranke resultiert aus dem Ignorieren von „unvollständigen Randbereichen“ bei der Repartitionierung.

[20] Dimension 2 ist die dritte Dimension, da der Index für Dimensionen in RasDaMan bei 0 beginnt

[21] Ein Resultat mit gleicher Dimensionalität ist häufig zur Identifikation der Scheiben, etwa nach einer Selektion, notwendig. Erhält man etwa als Resultat [0:127, 0:63, 13:13] ist der 13. Monat selektiert worden. Bei einer Reduktion der Dimensionalität im Repartitionierungsoperator würde diese Information prinzipiell verloren gehen.

Das effiziente Laden der Daten bei Bereichsanfragen erfolgt mit Hilfe eines R^+-Baum Index, der ebenso wie die Array Daten selbst im RDBMS gespeichert wird.

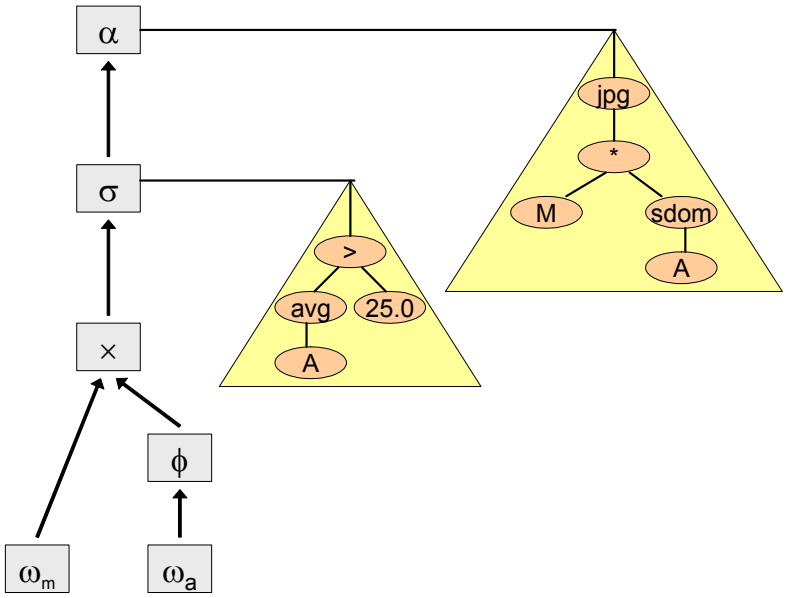


Abbildung 2.8: Beispiel für Anfragebaum

2.6 Zusammenfassung

Dieses Kapitel beschreibt multidimensionale Array Daten und deren Verarbeitung in einem Array DBMS. Es bildet die Basis für die Definition der parallelen Verarbeitung von Operationen auf diesen Daten. Es sei darauf hingewiesen, dass das Kapitel keine Beschreibung eines Datenmodells bzw. eines Array DBMS ist, wie es vor Beginn der vorliegenden Arbeit existierte. Vielmehr wurden Weiterentwicklungen, welche im Rahmen des Projekts ESTEDI entstanden, jedoch nicht direkt mit der parallelen Verarbeitung dieser Daten zu tun haben, bereits in die Formalisierung integriert. Im Detail wurden folgende Punkte behandelt:

1. Formalisierung des logischen Datenmodells für multidimensionale Array Daten, genauer gesagt die Trägerobjekte sowie die darauf definierten Operationen. In den folgenden Kapiteln dieser Arbeit kann auf dieser Grundlage die Eignung der Operatoren auf Parallelverarbeitung formal analysiert werden.
2. Beschreibung des physischen Datenmodells. Dazu wurde die Abbildung der logischen Datenelemente auf die physikalische Speicherebene untersucht. Dies ist vor allem für eine spätere Bewertung der parallelen Algorithmen bezüglich des (zusätzlich) zu ladenden Datenvolumens nötig.
3. Integration von multidimensionalen Arrays und darauf definierter Operationen in das relationale Datenmodell sowie eine Analyse der daraus entstehenden Probleme. Mengenoperationen auf Arrays sind nicht identisch mit den relationalen Operatoren, somit kann von einer vollständigen Integration keine Rede sein.
4. Präsentation einer Implementierung für ein Array DBMS am Beispiel des rasdaman DBMS. Dieses Array DBMS wurde als Implementierungsbasis für die in dieser Arbeit entwickelten Konzepte und Algorithmen ausgewählt.

Kapitel 3 Parallelität in relationalen Datenbanksystemen

Alles selbst machen zu wollen,
ist das Zeichen des Unbegabten.
(Richard von Schaukal, österr. Lyriker)

Dieses Kapitel beinhaltet eine kurze Zusammenfassung von Parallelität in relationalen Datenbanksystemen. Es bildet damit den „State of the Art“ (übersetzt in etwa: Stand der Forschung) für die vorliegende Dissertation. In Ermangelung von wissenschaftlichen Ergebnissen bezüglich der parallelen Analyse speziell von multidimensionalen Rasterdaten bildete die umfassende Analyse von Parallelität in relationalen Systemen ein „Sprungbrett“ für die in Kapitel 4 vorgestellten Algorithmen. Neben einer Neubewertung und Adaption relationaler Parallelisierungsalgorithmen werden dort jedoch auch neue Methoden speziell für große multidimensionale Rasterdaten entwickelt, die bisher keine Entsprechung in der relationalen Welt haben.

Der Begriff „Datenbanken“ wird häufig als Synonym für relationale Datenbanksysteme (RDBMS) verwendet. Das kommt sicher daher, dass diese Art von DBMS mit weitem Abstand die am weitesten verbreitete und am besten wissenschaftlich untersuchte Ausprägung ist. RDBMS haben mit Hilfe von Erweiterungen konkurrierende Datenmodelle quasi integriert, so sind etwa objektorientierte DBMS heute ein Nischenprodukt während relationale Systeme zu objektrelationalen mutierten. Ein ähnliches Schicksal scheint aktuell auch XML Datenbanken zu ereilen, aktuelle RDBMS bieten Erweiterungen für effiziente Speicherung und Zugriff auf XML Daten. Aufgrund dieser Dominanz relationaler Datenbanksysteme in Forschung und auf dem Markt konzentrieren wir uns im Folgenden auf parallele Anfrageverarbeitung für diese Systeme als Basis von und Vergleichsmaßstab zu dieser Arbeit. Kapitel 3 zeigt den aktuellen Stand von Forschung und eine Übersicht bezüglich Parallelität von kommerziellen relationalen DBMS. Parallelität in (reinen) objektorientierten DBMS (OODBMS) ist in nur wenigen Produkten implementiert, zumeist Forschungsprototypen. Lediglich in einem einzigen kommerziell vertriebenen OODBMS wird Parallelität zur Steigerung der Performanz genutzt, nämlich Objectivity [Obj99]. Eine gute Zusammenfassung von parallelen OODBMS findet sich in [Nor00]. Untersuchungen zu paralleler Anfrageverarbeitung für XML Datenbanken oder anderen Arten von DBMS sind uns nicht bekannt.

Parallele Anfrageverarbeitung für relationale Datenbanksysteme hatte in der Welt der Datenbankforschung seine Blütezeit Ende der 80er und Anfang der 90er Jahre. Die zentralen Mechanismen und Algorithmen von RDBMS waren zu dieser Zeit wissenschaftlich gesehen weitgehend „ausgetretene Pfade“, neue Trends waren vor allem Objektorientierung und eben Parallelität. Die Grundkonzepte einer parallelen Verarbeitung relationaler Daten sowie die Unterscheidung der prinzipiell unterscheidbaren Architekturen waren bereits sehr früh zu erkennen, die folgenden Jahre wurden vor allem Optimierungen der Algorithmen und Anpassung relevanter Datenbankmodule wie der Anfrageoptimierung vorgestellt. Bis Ende der 90er Jahre hatten alle namhaften Datenbankhersteller parallele Anfrageverarbeitung, insbesondere die (technisch gesehen komplexere) Parallelverarbeitung von Einzelanfragen, in ihr Produkt integriert. Aktuell rücken parallele RDBMS wieder verstärkt in den Blickpunkt, insbesondere im Zusammenhang mit Hochleistungsnetzwerken wie etwa InfiniBand [Inf] oder Myrinet [Myr], die eine verbesserte Skalierbarkeit in lose gekoppelten Architekturen ermöglichen.

Dieses Kapitel soll einen kurzen Abriss der Forschung der letzten 2 Jahrzehnte aufzeigen. Insbesondere die Gütekriterien für Parallelität, parallele Architekturen, Parallelisierungsstrategien und Arten der Inter-Prozess Kommunikation werden kurz beleuchtet und aus heutiger Sicht bewertet. Ferner wird aufgezeigt, inwiefern die theoretischen Erkenntnisse in aktuelle parallele RDBMS integriert wurden. In Kapitel 4 wird dann die parallele Verarbeitung von Array Daten, d.h. Entscheidungen zu Architektur, Adaption von Algorithmen sowie die Entwicklung neuer Methoden, in diesen Kontext integriert.

3.1 Skalierbarkeit und deren Grenzen

Es gibt eine Reihe von Vorteilen eines parallelen DBMS gegenüber einem ohne Parallelverarbeitung, von dem nur einer die Verbesserung der Performanz ist. Gleichzeitig sollen Randbedingungen gewahrt werden, wie etwa einfache Administrierbarkeit, Transparenz der Datenverteilung bzw. Verteilung der Anfrage für den Endbenutzer, etc. Die Vorteile im Einzelnen sind vor allem:

1. **Verfügbarkeit / Ausfallsicherheit**
   Szenarien, die eine extrem hohe Ausfallsicherheit voraussetzen, sind wohl der häufigste Einsatzbereich für parallele DBMS. Parallele Instanzen des DBMS werden hierbei redundant eingesetzt, so dass der Ausfall einer Instanz (z.B. ein Rechner) durch eine andere Instanz ohne Verzögerung übernommen werden kann. Ein markantes Beispiel für diesen Einsatzzweck sind E-Commerce Anwendungen, bei denen Kunden über das WWW eine Produktdatenbank nutzen. Kommerzielle Anbieter wie Oracle haben diesen Bedarf an ausfallsicheren DBMS erkannt und bieten zum Teil vorgefertigte und einfach zu administrierende Lösungen an.
2. **Performanz / Skalierbarkeit**
   Ein paralleles DBMS kann Anfragen schneller verarbeiten. Hierbei werden die parallele Verarbeitung einer Menge von Anfragen (Inter-Query) bzw. Transaktionen und eine parallele Verarbeitung von Einzelanfragen (Intra-Query) unterschieden. Unter Skalierbarkeit versteht man - einfach gesagt – die Fähigkeit eines parallelen DBMS, die Erweiterung der parallelen Hardware, also etwa eine neue CPU oder einen neuen Rechner, zu nutzen. Wir werden Skalierbarkeit sowie deren Maßzahlen unten näher definieren.
3. **Erweiterbarkeit**
   Ein häufig gesehener Fehler ist die Verwechslung von Skalierbarkeit und Erweiterbarkeit. Während Skalierbarkeit ausdrückt, wie viel Nutzen aus neuen physikalischen parallelen Instanzen gezogen werden kann, beschreibt Erweiterbarkeit, wie einfach bzw. kosten-

günstig dies realisiert werden kann. So sind etwa Multiprozessorrechner typischerweise schlechter erweiterbar als ein Verbund von parallel arbeitenden Rechnern. Entgegen der oft gesehenen Behauptung sind jedoch bezüglich Skalierbarkeit Multiprozessorrechner überlegen.

Wir werden uns im Folgenden auf Performanz als Ziel unserer Arbeit konzentrieren. Ausfallsicherheit ist technisch einfach realisierbar, wurde beispielsweise konkret in das **rasdaman** Array DBMS durch die Möglichkeit von redundanten DBMS Server und Datenquellen bereits integriert. Erweiterbarkeit spielt für unsere Arbeit insofern eine Rolle, dass wir uns nicht auf eine parallele Architektur beschränken, sondern in dieser Hinsicht flexibel bleiben und somit eine große Anzahl verschiedener Architekturen und somit auch Architekturen mit einfacher Erweiterbarkeit unterstützen.

Das am häufigsten genutzte Qualitätskriterium für Parallelität ist der so genannte Speed-Up, also die Beschleunigung der Ausführung:

Definition 3.1: Speed-Up

Speed-Up beschreibt den Performanzvorteil, genauer die Verbesserung der Ausführungszeit eines parallelen Systems gegenüber einem sequentiellen System bei ansonsten fixen Parametern:

$$\text{Speed-Up} = \frac{\text{Zeit (sequentielles System)}}{\text{Zeit (paralleles System)}}$$

Verarbeitet ein paralleles DBMS eine Anfrage beispielsweise in 10 Sekunden statt in 40 Sekunden, ergibt sich ein Speed-Up von 4. Ein ähnliches Maß für parallele Performanz ist **Scale-Up**. Hierbei wird die Größe eines Problems (etwa die Menge der verarbeiteten Daten) in Relation gesetzt zum Parallelitätsgrad. Ein linearer Scale-Up wird durch den Wert 1 charakterisiert, d.h. ein paralleles System mit Grad n kann ein Problem mit Faktor n in identischer Zeit lösen.

Der mögliche Speed-Up nimmt mit Parallelitätsgrad tendenziell ab, wir zeigen das an einem Beispiel:

Beispiel 3.1: Grenzen der Skalierbarkeit

Betrachtet wird ein Problem mit einem parallelisierbarem Anteil b_p von 98%, d.h. 2% (b_s) der Ausführung sind nicht parallel ausführbar (also ein sehr gut parallelisierbares Problem). Es ergeben sich bei linearem Speed-Up folgende Grenzen der Skalierbarkeit.

$$\text{Zeit für parallele Ausführung (Grad n): } t_p = \frac{b_p}{n} + b_s$$

$$\text{Speed-Up: } \frac{1}{\frac{b_p}{n} + b_s}$$

$$\text{Grenze der Skalierbarkeit: } \lim_{n\to\infty} \frac{1}{\frac{b_p}{n} + b_s} = \lim_{n\to\infty} \frac{1}{\frac{0.98}{n} + 0.02} = 50$$

Diese Grenze wurde bereits 1967 von Amdahl aufgezeigt [Amd67]. Die Formel wird als Amdahls Gesetz bezeichnet und zeigt, dass Skalierbarkeit stark abhängig ist vom zu lösenden Problem, oder genauer vom Anteil des nicht parallelisierenden Bereichs b_s.

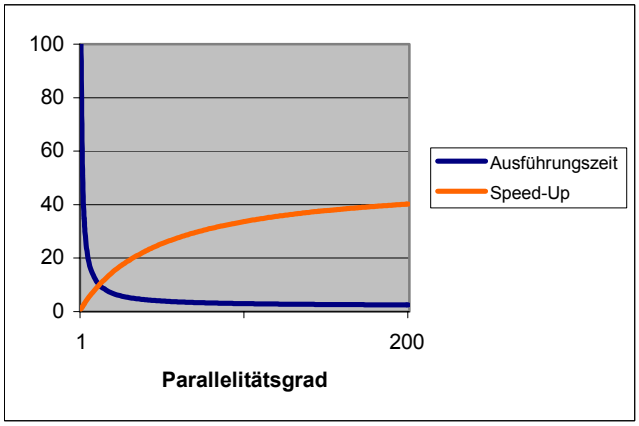


Abbildung 3.1: Beispiel für Amdahls Gesetz

Grafisch veranschaulicht sieht dies wie in Abbildung 3.1 aus (die Ausführungszeit sei in diesem Beispiel 100 Sekunden). Die Ordinate gibt die Ausführungszeit in Sekunden bzw. den Faktor des Speed-Up an. Die Speed-Up Kurve nähert sich asymptotisch dem Grenzwert von 50 an. Umso höher also der Parallelitätsgrad, umso geringer gestaltet sich der Zuwachs an Performanz.

Man kann Speed-Up als Güte eines konkreten parallelen Systems verstehen. Erreicht etwa ein paralleles DBMS auf einer parallelen Plattform mit 4 Instanzen (etwa einem Multiprozessorrechner mit 4 CPU) für typische Anfragen einen Speed-Up von nahezu 4, spricht man von einem **linearem Speed-Up**. Ein linearer Speed-Up ist das Optimum und Ziel von parallelen Systemen und kann nur durch Ausnutzen von Randbedingungen übertroffen werden[22]. Das Maß des Speed-Up allein sagt jedoch nur wenig über die Qualität eines parallelen Algorithmus aus. Erst der Speed-Up in Relation zu der Anzahl der parallelen Instanzen ergibt die Skalierbarkeit eines parallelen Systems oder eines Algorithmus. Man sieht an obigem Beispiel, dass selbst bei perfekten Rahmenbedingungen die Skalierbarkeit primär vom zu lösenden Problem, also im Kontext eines DBMS von der Anfrage abhängig ist.

[22] Beispielsweise wird oft ein überlinearer Speed-Up erreicht, wenn ein Problem bei einem sequentiellen System bereits an Ressourcengrenzen stößt, die durch ein paralleles System entschärft werden. Wenn etwa der physikalische Hauptspeicher für eine Berechnung im sequentiellen System nicht vollständig ausreicht, jedoch bei einem Verbund von Rechnern, so ist ein deutlich überlinearer Speed-Up möglich. Auch Replikation von Quelldaten und ein damit verbundenes Ausschalten von I/O Problemen ist ein häufig gesehenes Vorgehen, um einen überlinearen Speed-Up zu erreichen.

In der Praxis ist die Skalierbarkeit nicht nur begrenzt, vielmehr sinkt die Performanz vor allem wegen der Kosten für Kommunikation ab einer gewissen Grenze mit steigendem Parallelitätsgrad. Insgesamt lassen sich in der Praxis drei große Probleme identifizieren, die eine Skalierbarkeit jenseits eines Parallelitätsgrades von 100 praktisch unmöglich machen.

1. Kosten für die **Initialisierung** der parallelen Verarbeitung (engl.: startup costs)
   Hierzu zählen Kosten für die Initialisierung der Prozesse, der Kommunikation, etc. Diese Kosten werden jedoch meist durch die Initialisierung eines parallelen Prozess-Pools beim Start des DBMS von der Anfragebearbeitung abgezogen (sind also auch bei Speed-Up Berechnungen nicht mehr relevant). Kosten für das Parallelisierungsmodul, das einen Anfragebaum für sequentielle Ausführung in einen adaptierten für parallele Ausführung transferiert sind jedoch unter Umständen extrem hoch. Der Suchraum, der bei der Suche des optimalen parallelen Plans entsteht, ist in der Regel sehr groß und kann nur mittels Pruning[23] effizient durchsucht werden.
2. Kosten für die **Inter-Prozess Kommunikation** (engl.: interference costs)
   In dieser Kategorie lassen sich Kosten für die Synchronisation von Prozessen und für die Übermittlung von Zwischenergebnissen unterscheiden. Insbesondere der Transfer großer Datenmengen zwischen Prozessen, vor allem über Netzwerk, ist ein primärer Grund für schlechte parallele Skalierbarkeit.
3. Kosten durch **Verteilungsprobleme** (engl.: skew costs)
   Eine Ungleichverteilung von Datenvolumina oder der Auslastung führt zu Verringerung des parallelen Speed-Up. So genannte Skew Probleme wurden in der wissenschaftlichen Literatur ausführlich untersucht und klassifiziert (eine gute Abhandlung ist [Mär01]). Die wichtigste Strategie zur Vermeidung von Skew ist die dynamische (Um-)Verteilung der Last, im Englischen bezeichnet als dynamic load balancing.

Eine Vermeidung dieser Probleme ist Basis für eine gut skalierbare Implementierung eines parallelen DBMS. Wir werden bei der Analyse von Methoden für die Parallelverarbeitung von Array Daten in Kapitel 4 und der Beschreibung der Implementierung in Kapitel 5 immer wieder den Fokus auf diese Probleme legen.

3.2 Parallele Architekturen

Die Klassifizierung paralleler Architekturen für DBMS hat sich in den letzten 2 Jahrzehnten nicht geändert, lediglich die praktische Relevanz und die Einschätzungen bezüglich der „optimalen“ Architektur haben sich gewandelt. Grundlegend kann man zwei große Klassen von Architekturen unterscheiden, nämlich **eng gekoppelte Architekturen** (engl.: Shared-Everything, Teilen aller Ressourcen) und **lose gekoppelte Architekturen** (engl.: Shared-Nothing, kein Teilen von Ressourcen). Eine häufig genutzte hybride Lösung sind lose gekoppelte Systeme mit **gemeinsamen Sekundärspeicher** (engl.: Shared-Disk, gemeinsame Nutzung von Festplattenspeicher).

1. **Shared-Everything**
   Auch bekannt als Shared-Memory Architektur, symmetrische Mehrprozessorsysteme (engl.: Symmetric Multiprocessing, SMP) oder einfach Multiprozessorrechner. In der Praxis sind das meist Rechner mit 2 bis 32 Prozessoren. Den primären Vorteilen der effizienten Kommunikation über Hauptspeicher und einfachen Lastverteilung stehen schlechte Erweiterbarkeit, das Problem der Cache Kohärenz und schlechte Fehlerisolation gegen-

[23] engl.: pruning, wörtlich übersetzt „Gehölzschnitt“. Technik zur Baumevaluierung, bei der Teilbäume mit schlechter Bewertung frühzeitig für die weitere Evaluierung ausgeschlossen werden.

über. Aktuelle technologische Entwicklungen in diesem Bereich konzentrieren sich auf die Entwicklung neuer Bussysteme, um einen bekannten Engpass, den Datentransfer zwischen Prozessoren und Hauptspeicher, zu entschärfen (siehe etwa [PCIX]). Dieser Engpass, der nach Untersuchungen bereits ab acht CPU bemerkbar ist, wird meist durch so genannte **NUMA** Architekturen (engl.: Non Uniform Memory Access, ungleichförmiger Speicherzugriff) umgangen. Hierbei teilen sich jeweils wenige CPU, in der Praxis meist 4 oder 8, den Zugriff auf Hauptspeicher über einen herkömmlichen PCI Bus. Diese kleinen SMP Cluster sind dann über spezielle Bussysteme miteinander verbunden. Somit entsteht ein differenzierter Speicherzugriff, er erfolgt entweder im eigenen Speicher oder in einem entfernten Speichersegment. Programmiertechnisch wird jedoch der Speicher gleichförmig adressiert, weshalb man NUMA als Spezialfall von SMP begreifen kann, d.h. als spezielle Implementierung. NUMA Architekturen beinhalten im Gegensatz zu reinen SMP Systemen oft über 100 Prozessoren (solche Systeme werden etwa von SUN Microsystems angeboten [SUN]).

2. **Shared-Nothing**
   Diese Systeme realisieren einen Verbund von parallel arbeitenden Rechnern[24] und wurden auch als Workstation Cluster oder MPP (engl.: Massively Parallel Processing) Architekturen bekannt. So genannte Beowulf Cluster bezeichnen eine Cluster von PC mit Linux Betriebssystem, also eine sehr kostengünstige Variante von Shared-Nothing. Es hat sich gezeigt, dass solche Architekturen für viele Probleme eine ähnliche Leistung wie große SMP Rechner zu einem Bruchteil des Preises liefern. Vorteile als Plattform für parallele DBMS sind geringe Kosten, einfachere Erweiterbarkeit und Fehlerisolation. Primärer Nachteil ist die Kommunikation über Netzwerk, die deutlich kostenintensiver ist als Kommunikation in SMP Architekturen. Programmiertechnisch sind diese Systeme deutlich zu unterscheiden von SMP, bei denen Kommunikation zwischen Prozessen über einen durchgehend adressierten Speicher stattfindet. Die Inter-Prozess Kommunikation in Shared-Nothing erfolgt über Nachrichten, die über Netzwerke versendet werden. Beispiele für diesbezügliche Standards sind Parallel Virtual Machine PVM [PVM] oder das aktuellere Message Passing Interface MPI, welches mittlerweile in Version 2 standardisiert ist [MPI].
3. **Shared-Disk**
   Genau genommen sind das Shared-Nothing Systeme mit gemeinsamen Sekundärspeicher, der meist als Festplatten RAID (engl.: Redundant Array of Inexpensive Disks) realisiert wird. Die Nutzung einer gemeinsamen Datenbasis hat viele Vorteile für parallele DBMS, folglich ist diese Architektur als typisch anzusehen. Mit der aktuellen Entwicklung neuer Netzwerkstandards (etwa Myrinet [Myr], InfiniBand [Inf]) ist der gemeinsame Zugriff auf die Datenbasis auch nicht mehr notgedrungen ein Engpass.

Diese verschiedenen Architekturen für parallele DBMS sind in Abbildung 3.2 skizziert. Die gestrichelte Umrandung in der Abbildung entspricht hierbei einem autonomen Rechner. Shared-Nothing und Shared-Disk sind auf der rechten Seite als verwandte Architektur gemeinsam abgebildet, in einem Shared-Nothing System besitzt jeder Knoten seinen eigenen Sekundärspeicher.

Historisch gesehen war die erste große Welle von parallelen DBMS auf Shared-Everything Architekturen beschränkt, knapp gefolgt von Implementierungen für Shared-Nothing bzw. Shared-Disk Architekturen. DBMS der ersteren Klasse waren XPRS (eine Erweiterung des Postgres DBMS) von Stonebraker et al. 1988 [SKPO88], Volcano von Graefe 1990 [Gra90] und DBS3 von Bergsten et. al. 1991 [BCV91]. [Gra90] ist speziell für die Kapselung paralleler Funktionalität in speziellen Operatoren, die dem Iterator Modell folgen, bekannt. Diese

[24] diese Instanzen des Clusters sind häufig selbst Multiprozessor Rechner, also SMP Systeme

Kapselung in parallelen Operatoren (Bezeichnungen reichen von *send/receive* über *split/merge* bis *exchange*) haben Einzug gefunden in alle wichtigen Implementierungen von parallelen DBMS (und wurden auch von uns übernommen, siehe Kapitel 4). Die wichtigsten Shared-Nothing DBMS waren Gamma von Dewitt et. al. 1990 [DGSB+90], Bubba von Boral et. al. 1990 [BACC+90] und Prisma von Apers et. al. 1992 [AVFG+92]. Kommerzielle DBMS folgten diesem Trend mit einer Verzögerung von einigen Jahren. Shared-Everything Implementierungen wie etwa IBM DB2 auf einer IBM3090 als erstes DBMS auf SMP wurden gefolgt von Shared-Nothing DBMS, beispielsweise Teradata's DBC, Tandem NonStopSQL und IBM DB2 Parallel Edition. Oracle konzentriert sich hingegen bis heute auf Shared-Disk.

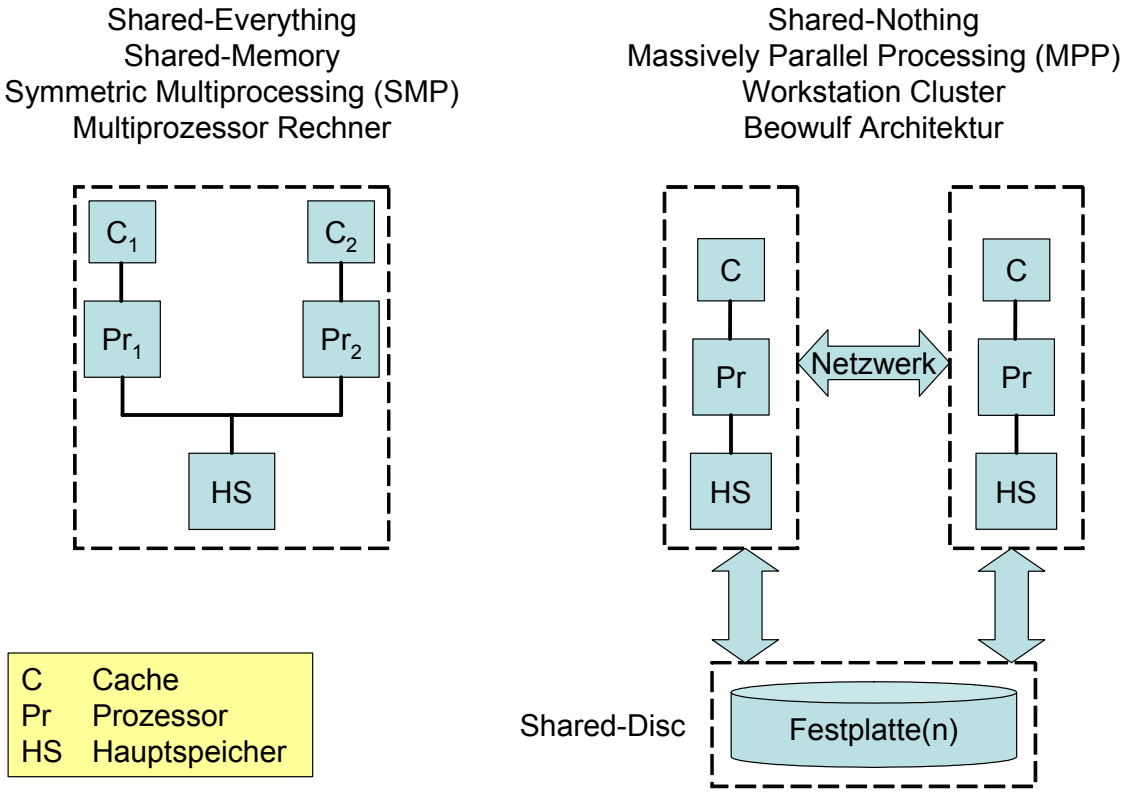


Abbildung 3.2: Parallele Architekturen für DBMS

Obwohl schon 1990 von David DeWitt und Jim Gray prophezeit [DG90], haben sich Shared-Nothing Architekturen für DBMS in der Praxis bisher nicht komplett durchgesetzt. In allen Preissegmenten rangieren heute nach wie vor SMP Systeme vor Cluster Lösungen (siehe etwa [TPC]), lediglich in hochverfügbaren Anwendungen werden Cluster eingesetzt, allerdings weniger zur parallelen Verarbeitung von Einzelanfragen, sondern für die parallele Verarbeitung einer Menge von Transaktionen.

Bezüglich unserer Implementierung eines parallelen Array DBMS wollten wird jede Einschränkung der Architekture vermeiden. Im Projekt ESTEDI (Kapitel 1.4) wurde **rasdaman** anfangs auf SMP mit maximal 4 CPUs eingesetzt, jedoch wünschten die Projektpartner explizit die Option, **rasdaman** bei Bedarf auch auf Rechnercluster einsetzen zu können. Als Folge dieser Anforderungen kann das parallele **rasdaman** DBMS aktuell auf Multiprozessorrechnern (inklusive NUMA Architekturen) und Rechnerclustern (inklusive Shared-Disk Architekturen) eingesetzt werden.

3.3 Arten der Parallelität

Parallelität in relationalen DBMS wird bezüglich der folgenden Merkmale klassifiziert (siehe etwa [Rah94] [OV99]):

- Was wird parallelisiert? Das parallele Laden bzw. Schreiben der Daten auf Sekundärspeicher (bzw. Tertiärspeicher) wird als **I/O Parallelität** (engl.: In & Out) oder auch **E/A Parallelität** bezeichnet. Dem gegenüber steht die **Verarbeitungsparallelität** der Daten, also vor allem die Verteilung der CPU Last auf parallele Instanzen.
- Wie wird parallelisiert? Man unterscheidet die Konzepte **Pipeline Parallelität** und **Datenparallelität**. Erste Methode nutzt die Tatsache, dass Knoten im Anfragebaum Teile ihres Resultats weitergeben können während die Berechnung noch im Gange ist. Der Vaterknoten kann somit schon Berechnungen auf diesen Teilergebnissen anstellen. Die zweite Methode der Datenparallelität verteilt die Daten auf verschiedene parallele Instanzen, um dann Operationen auf diesen Datenfragmenten gleichzeitig ausführen zu können.
- Wie fein ist die parallele Verarbeitung? Man unterscheidet hier die Parallelverarbeitung auf Basis von Transaktionen (Inter-, Intra-**Transaktionsparallelität**), Anfragen (Inter-, Intra-**Query-Parallelität**), Operatoren (Inter-, Intra-**Operator-Parallelität**).

3.3.1 E/A Parallelität und Verarbeitungsparallelität

Das Laden der Daten von bzw. das Schreiben auf Sekundärspeicher ist nach wie vor ein Flaschenhals für relationale Anfrageverarbeitung, insbesondere für Datenbanken mit großen Datenvolumina. Dieser Flaschenhals kann entschärft werden durch

- Low-level Lösungen. **RAID** (engl.: redundant array of inexpensive discs) Systeme bündeln mehrere Festplatten zu einer logischen Platte. Parallele Lese- und Schreibvorgänge erhöhen die Datentransferrate. Eine ähnliche Methode ist das so genannte **Striping** (siehe beispielsweise [SWZ98]), also das Verteilen von kleinen Fragmente fester Größe mit einem Round-Robin Algorithmus auf verschiedene Platten. Im Gegensatz zu den unten genannten Strategien erfolgt hier die Datenverteilung ohne jede Semantikerhaltung.
- **Partitionierung** der Datenbasis ist eine logische Unterteilung der Tabellen, die zum Beispiel vom DBMS verwendet werden kann, um diese Partitionen auf verschiedene Platten (innerhalb eines Rechners) abzulegen und die Leserate für Daten so zu erhöhen. Der Speed-Up hängt bei dieser Technik primär von der Anfrage ab, ist in der Regel keineswegs linear, da die Daten ungleich verteilt sind.
- **Replikation**. Die redundante Speicherung von Daten kann Lesevorgänge praktisch mit einem linearen Speed-Up parallelisieren, allerdings müssen Schreibvorgänge entweder synchron oder asynchron auf replizierte Daten durchgeschrieben werden. Während Replikation bei Systemen mit vielen Transaktionen große Nachteile impliziert, kann sie bei Systemen mit primärem Lesezugriff (etwa Data Warehouse oder eben Rasterdaten) den E/A Flaschenhals zum Großteil entschärfen.
- **Fragmentierung** oder Positionierung der Daten (engl.: data placement). Eine Fragmentierung der Daten bedeutet das Speichern auf Platten von verschiedenen Instanzen in einem Shared-Nothing System (also auf verschiedenen Rechnern). Während sie den E/A Flaschenhals entschärfen kann, wird in der Regel im gleichen Maße das Netzwerk zum neuen Flaschenhals, da die parallel gelesenen Daten bei einer Anfrage zu entfernten parallelen Instanzen über Netzwerk übertragen werden müssen.[25]

[25] Fragentierung bezeichnet meist eine Verteilung von Tupel einer Relation auf mehrere Rechner in einem verteilten DBMS. Partitionierung hingegen verteilt in der Regel Relationen komplett auf die Partitionen, die auf verschiedene Platten positioniert sind. Partitionierung wird etwa in Oracle durch die Definition mehrerer Tablespaces für eine Datenbank realisiert.

Wissenschaftliche Arbeiten beschäftigen sich insbesondere mit Fragmentierung von relationalen Daten. In [MD97] werden allgemeine Faktoren für eine optimale Verteilungsstrategie vorgeschlagen, einen Überblick über praktische Verfahren gibt etwa [Rah94]. Grundsätzlich lassen sich statische und dynamische Strategien erkennen. Dynamische Verteilungsstrategien versuchen, die Datenfragmentierung bezüglich der Anfragen zu reorganisieren, um somit implizit Probleme der Lastverteilung zu umgehen[26] [LKOT+00]. Statische Verfahren lassen sich klassifizieren bezüglich der der zu verteilenden Fragmente und des Verteilungsalgorithmus selbst. Eine horizontale Fragmentierung ergibt eine Verteilung von kompletten Tupel einer Relation auf verschiedene Instanzen, vertikale Fragmentierung dagegen resultiert in einer Aufteilung von verschiedenen Attributen der Relation auf die Instanzen, hybride Fragmentierung stellt eine Zwischenform dar. Algorithmen für die Verteilung selbst sind Round-Robin (eine alternierende Verteilung der einzelnen Elemente), basieren auf Hashing oder weisen den Instanzen feste Datenbereiche (engl.: range partitioning) zu.

Im Rahmen der vorliegenden Arbeit werden wir jedoch auf E/A Parallelität nur am Rande eingehen. Erfahrungen aus dem ESTEDI Projekt lassen erkennen, dass das typische Einsatzszenario für multidimensionale Rasterdaten ähnlich wie bei Data Warehouse Anwendungen einer Vielzahl von Lesezugriffen nach einem einmaligen Einfügen der Daten entspricht. Dies resultiert hauptsächlich aus der Tatsache, dass das Einfügen von multidimensionalen Rasterdaten komplex ist – es müssen eigens angepasste Einfügeprogramme geschrieben werden – und andererseits sehr zeitintensiv ist. Folglich lassen Array DBMS bezüglich des Datenzugriffs eine Verwandtschaft zu OLAP und weniger zu OLTP erkennen. Außerdem beschränken sich Anfragen an Rasterdaten meist auf eine Kollektion oder ein MDD, Verknüpfungen sind eher selten. Somit ist der Zugriff auf Datenmengen gut vorhersehbar. Aus diesen Gründen ist Datenfragmentierung einfach realisierbar, da die primären Herausforderungen der Transaktionskontrolle und der aus Verteilung entstehenden Lastbalancierung kaum in Erscheinung treten. Wir werden uns im Folgenden auf Verarbeitungsparallelität konzentrieren.

3.3.2 Datenparallelität und Pipeline-Parallelität

Berechnungen auf großen Datenmengen folgen dem Iteratorkonzept [Gra93]. Eine klassische „bottom-up“ Verarbeitung des Operatorbaumes, bei der das komplette Ergebnis von unten nach oben durch den Operatorbaum gereicht wird, ist für solch große Datenmengen ineffizient, da die Größe der Zwischenergebnisse die Größe des Hauptspeichers deutlich übersteigt und folglich auf Sekundärspeicher ausgelagert werden muss. Das Iteratorkonzept (auch *open-next-close* Verarbeitung), berechnet nach einem Initialisieren der Ressourcen (durch die Methode *open*) nach und nach die Ergebnisse in feiner Granularität (Methode *next*), bevor die Ressourcen wieder freigegeben werden (Methode *close*). Die Granularität der Zwischenergebnisse ist in RDBMS dabei meist eine Plattenseite (engl.: page), d.h. in der Praxis häufig 4K oder 8K. Diese Strategie vermeidet das Auslagern und Einlagern von Zwischenergebnissen auf bzw. von Sekundärspeicher.
Das Konzept der **Pipeline-Parallelität** nutzt diese Verarbeitungsstrategie, indem im Operatorbaum Bereiche parallel ausgeführt werden, das heißt Iteratoren werden (fast) unabhängig voneinander auf diese Teilbereiche des Baumes angewandt. Dies teilt den Baum in Bereiche von Datenerzeugern und Datenkonsumenten, die Übergabe der Zwischenergebnisse erfolgt durch so genannte Pipelines. Diese Pipelines sind nichts anderes als Puffer für die Zwischenergebnisse, in Shared-Everything Systemen typischerweise gemeinsam genutzter Hauptspeicher, in Shared-Nothing Nachrichtenpuffer. Eine Synchronisation zwischen den parallelen

[26] Die Verteilung der Daten impliziert die Verteilung der Berechnungsarbeit, da typischerweise Operatoren der Instanz zugewiesen wird, auf der die Daten liegen.

Instanzen muss nur bei Pufferüberlauf erfolgen. In diesem Fall wartet der Datenerzeuger bis der Verbraucher einen leeren Puffer signalisiert oder er beginnt, die Zwischenergebnisse auf Sekundärspeicher auszulagern.
In der Praxis wird Pipeline-Parallelität sekundär eingesetzt. Das Hauptproblem besteht darin, dass es nur schwer möglich ist, Bereiche des Operatorbaumes zu selektieren, in welchen die Arbeitslast fast identisch ist. Eine ungleiche Verteilung der Last resultiert in leeren oder komplett gefüllten Pipelines, was wiederum zu Auslagerung auf Sekundärspeicher oder Warten von Prozessen führt. Beides beeinträchtigt den Speed-Up deutlich.

Datenparallelität verteilt die Last auf parallele Instanzen, indem die Datenmenge getrennt und die Operationen auf diesen Partitionen separat verarbeitet werden. Datenparallelität zeigt hervorragende Performanz, wenn die Daten komplett partitioniert und unabhängig voneinander verarbeitet werden können. Dies ist allerdings nicht immer der Fall. Als Mechanismen für den Datenfluss zwischen parallelen Instanzen unterscheidet man eine 1:1 Pipeline, Replikation, Partitionierung nach gegebenen Algorithmus (round-robin, hash, range partitioning), sowie das Zusammenführen (engl.: merging) nach bestimmten Algorithmen (merge, sequential, asynchron) und das Repartitionieren von Daten.

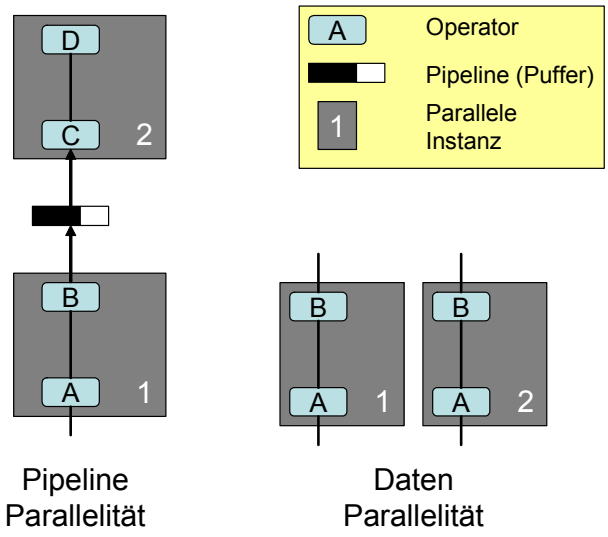


Abbildung 3.3: Pipeline- und Datenparallelität

Abbildung 3.3 skizziert ein Beispiel für Pipeline-Parallelität und Datenparallelität. Bei Pipeline-Parallelität links führt ein erzeugender Prozess die Operatoren A und B aus, ein konsumierender Prozess nimmt dessen Ergebnisse aus der Pipeline und führt die Operatoren C und D aus. Auf der rechten Seite werden die Operatoren B und C parallel auf den Instanzen 1 und 2 ausgeführt, jeweils auf einem Teilbereich der Daten.

Pipeline-Parallelität ist für Array Daten aus diversen Gründen äußerst problematisch, während Datenparallelität aufgrund der Daten und der Operationen darauf hervorragend geeignet ist. Eine Analyse hierüber erfolgt in Kapitel 4.

3.3.3 Granularität der parallelen Verarbeitung

Parallelität in relationalen DBMS kann klassifiziert werden bezüglich der Granularität der parallelen Verarbeitungsschritte (Transaktion, Anfrage, Operator). Abbildung 3.4 zeigt eine

aus [Rah94] entnommene Klassifizierung von Parallelitätsarten, wobei die Granularität von oben nach unten geringer wird.

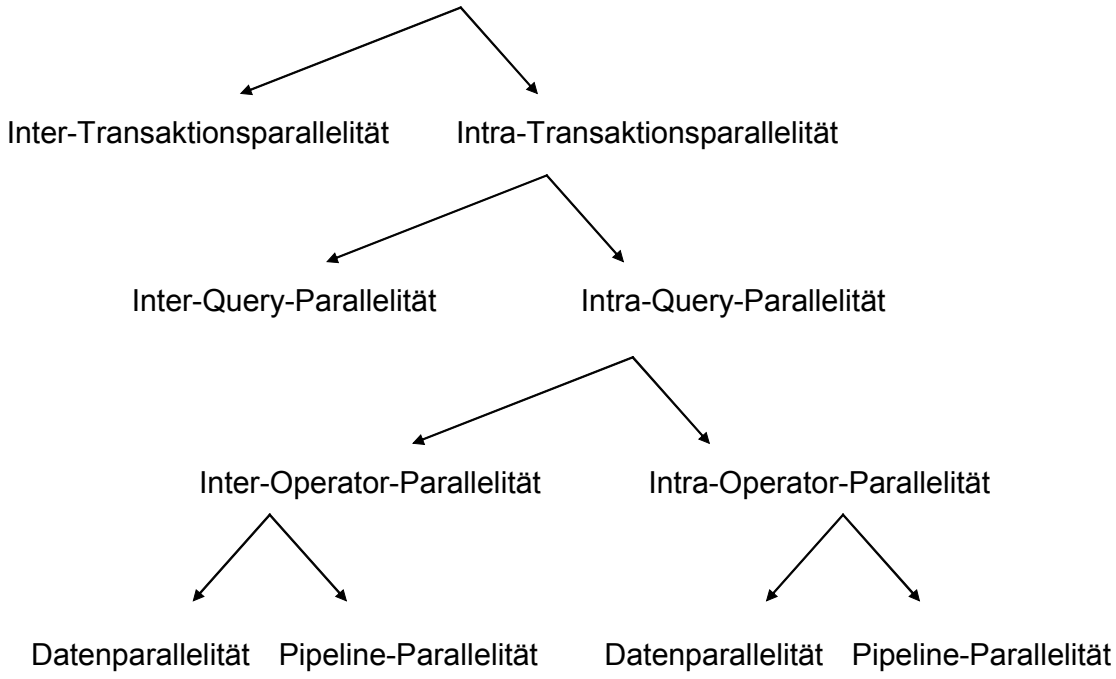


Abbildung 3.4: Klassifizierung von Parallelität in RDBMS

Die einfachste Art der Parallelität ist **Inter-Transaktionsparallelität**, das Verteilen kompletter Transaktionen auf parallele Instanzen. Inter-Transaktionsparallelität ist somit Voraussetzung für parallelen Mehrbenutzerbetrieb. Haupteinsatzgebiet dieser Art von Parallelität ist die Skalierung eines OLTP (engl.: Online Transaction Processing, Transaktionsverwaltung) Systems, um eine größere Anzahl von Transaktionen pro Sekunde abwickeln zu können. Elementare Probleme, die insbesondere in Shared-Nothing Architekturen eine Herausforderung bedeuten, sind:

- Koordination von Sperrverfahren (engl.: Locking) und Protokollierung (eng.: Logging).
- Sicherstellen von Cache Kohärenz. Bei Benutzung von zwischengespeicherten Daten (engl.: Cache, Speicher) muss sichergestellt sein, dass auf aktuelle Daten zugegriffen wird. Hat ein anderer Prozess diese Daten geändert, können inkonsistente Daten entstehen. Dieses Problem wurde bei Shared-Everything Architekturen als Problem der Cache Kohärenz bekannt (gemeint ist hier der Prozessorcache bzw. ein evtl. vorhandener Sekundärcache), ist jedoch für Shared-Nothing Systeme ebenso relevant, wenn Daten im Hauptspeicher der einzelnen Parallelknoten zwischengespeichert werden, um die Performanz zu erhöhen.

Bei der **Inter-Query Parallelität** erfolgt die parallele Verarbeitung auf der Granularität von Anfragen (engl.: Query) statt Transaktionen. Die Konzepte sind der Inter-Transaktionsparallelität ansonsten sehr ähnlich.

Mit den oben vorgestellten Inter-Transaktionsparallelität und Inter-Query-Parallelität können Einzelanfragen nicht parallel verarbeitet werden. Solch eine parallele Verarbeitung von Einzelanfragen ist in bestimmten Szenarien jedoch erforderlich. Speziell Anwendungen, die extrem komplexe Einzelanfragen stellen, etwa Data Warehouse oder in unserem Fall die Analyse von Arrays, verlangen nach **Intra-Query-Parallelität**. Bei der **Inter-Operator Parallelisierung** werden hierbei verschiedene Operatoren des Anfragebaumes parallel verarbeitet. Dies geschieht entweder als Pipeline Parallelisierung (vertikale Parallelisierung) oder als Datenparallelisierung (horizontale Parallelisierung von verzweigten Bäumen, engl.: bushy trees). Die parallele Berechnung von Einzeloperatoren, also **Intra-Operator Parallelität**, wird in der Praxis fast ausschließlich durch Datenparallelisierung realisiert.

In Kapitel 4 wird Intra-Query Parallelität für Array Daten ausführlich beschrieben. Inter-Operator Parallelität als erste Ausprägung wird dort analysiert und als nicht adäquate Strategie zur Parallelisierung von Analysen auf Array Daten verworfen. Intra-Operator Parallelität wird vor allem in Form von Datenparallelität in diesem Kapitel eingehend beschrieben. Ferner wird eine Strategie der Parallelisierung neu entwickelt, die in Abbildung 3.4 nicht enthalten ist: Datenparallelität durch Partitionierung von Einzelobjekten (statt einer Menge). Diese Parallelität wird von uns als Intra-Objekt Parallelisierung deklariert.

3.4 Parallele Ausführung einer Anfrage

Bisher wurden Ziele, Grenzen, Architekturen und Methoden von Parallelität vorgestellt. In diesem Kapitel wollen wir Parallelität in die Anfrageverarbeitung in einem DBMS einbeziehen und auf Herausforderungen in diesem Sinne eingehen, d.h. die Integration in ein bestehendes DBMS wird beleuchtet.

Abbildung 3.5 ordnet die Parallelisierung einer Anfrage in die komplette Anfrageverarbeitung ein. Das neue **Parallelisierungsmodul** wird typischerweise nach einer Optimierungsphase vor der Ausführung der Anfrage angesprochen. Die evtl. erforderliche Adaption von Modulen ist in der Abbildung gestreift markiert. Grundsätzlich kann man zwei Ansätze für die Integration des Parallelisierungsmoduls unterscheiden. Der in Abbildung 3.5 gezeigte Ansatz nennt sich Zwei-Phasen Strategie. Optimierung und Parallelisierung sind hier zwei getrennte Module, typischerweise sind jedoch auch im Optimierer Anpassungen nötig. In der Praxis zeigt sich häufig, dass die durch den Optimierer vorgenommenen Veränderungen des Anfragebaumes die Parallelisierung erschweren oder zu suboptimalen parallelen Anfragebäumen führen. Ein klassisches Beispiel hierfür ist die Linearisierung von Bäumen (engl.: left-deep tree, right-deep tree) während der Optimierung, die der Parallelisierung insofern widerspricht, als dass die parallele Verarbeitung von verzweigten Bäumen (engl.: bushy-tree) dadurch erschwert wird. Einen Ausweg aus diesem Konflikt bietet die so genannte Ein-Phasen Strategie. Optimierung und Parallelisierung verschmelzen hier zu einem Modul, welches das ursprüngliche Optimierungsmodul komplett ersetzt.

Die Generierung eines **parallelen Ausführungsplanes**, also eines adaptierten Anfragebaumes, kann mittels Heuristiken oder kostenbasiert durch Analyse des Suchraumes erfolgen. Beide Methoden garantieren keinen optimalen parallelen Plan. Prägnant formuliert kann man sagen, dass das Parallelisieren mittels Heuristiken schneller ist, während die Analyse mittels Kostenmodell oft bessere parallele Ausführungspläne erzeugt. Das Anwenden von Heuristiken geschieht regelbasiert, d.h. der Anfragebaum wird nach bestimmten Mustern durchsucht und nach einem vorgegebenen Algorithmus werden Knoten für die parallele Funktionalität eingefügt. Liegt für die einzelnen Operatoren des Baumes ein Kostenmodell vor, welches beispielsweise bereits bei der Optimierung genutzt wird, kann auch eine Suche nach einem

optimalen parallelen Plan durch Berechnung der Kosten geschehen. Techniken hierfür sind ein Durchsuchen des kompletten Suchraumes (engl.: exhaustive search), ein Durchsuchen nach Zufallsprinzip (engl.: randomized search) oder bottom-up und top-down Algorithmen. Der Suchraum ist typischerweise extrem groß, deshalb kann eine Suche meist nur mit Hilfe von Pruning (engl.: pruning, abschneiden) realisiert werden, d.h. Teilpläne, die wahrscheinlich nicht zu einem optimalen parallelen Komplettplan führen, werden nicht weiter analysiert.

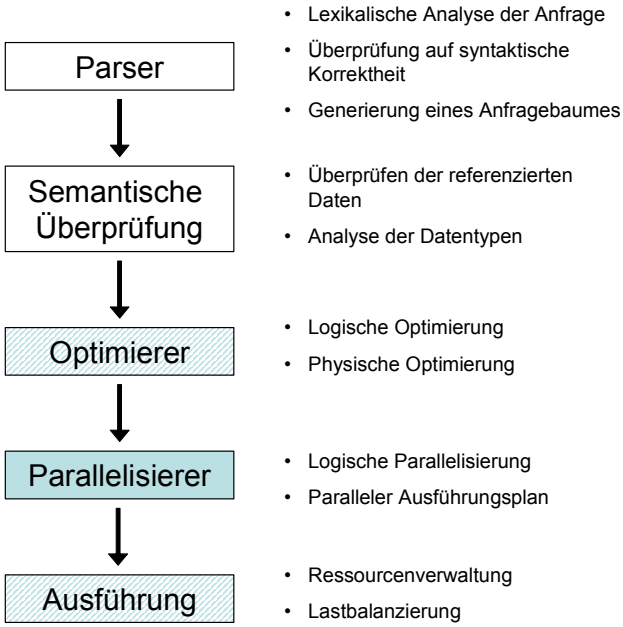


Abbildung 3.5: Anfrageverarbeitung in einem parallelen DBMS

Abbildung 3.6 zeigt die Phasen der Parallelisierung noch einmal beispielhaft. Der skizzierte Baum links sei der sequentielle Anfragebaum, d.h. die Ausgabe des Optimierungsmoduls. In [Nip00] werden vier Phasen für die Parallelisierung vorgeschlagen:

1. Inter-Operator Parallelisierung. Dies wird in der Abbildung als zweiter Baum dargestellt. Der Anfragebaum wird in 4 Blöcke aufgeteilt, welche jeweils durch verschiedene Prozesse verarbeitet werden können und über Pipelines miteinander Zwischenergebnisse austauschen (im Anfragebaum sind das spezielle Knoten, die die Inter-Prozess Kommunikation übernehmen). Wir erhalten somit eine erste logische Sicht der Ausführung.
2. Intra-Operator Parallelisierung für Operatoren mit hohen Kosten. Operatoren, die als Operatoren mit hohen Kosten identifiziert werden (entweder basierend auf Heuristiken oder auf einem Kostenmodell) werden datenparallel ausgeführt. Der Parallelitätsgrad ist hierbei meist vorgegeben (durch globale Einstellungen, welche die zugrunde liegende Hardware repräsentieren).
3. Blockerweiterung und Parallelisierung von Operatoren mit geringen Kosten. Dies führt dazu, dass einzelne Blöcke, die in vorigem Schritt entstanden, expandiert und von mehreren Prozessen auf verschiedenen Datenfragmenten ausgeführt werden. In der Abbildung ist dieser resultierende Ausführungsplan rechts skizziert. Die Inter-Operator Blöcke werden jeweils von unten nach oben in zwei, drei, zwei und einer parallelen Instanz verarbeitet.

4. Kombination von Blöcken für die Vermeidung von parallelen Kosten. Der in den bisherigen Schritten gewonnene parallele Ausführungsplan zeigt in der Regel einen Parallelitätsgrad, der den tatsächlichen Parallelitätsgrad der Hardware deutlich übersteigt. Dies würde lediglich zu Zusatzkosten für die Kommunikation der parallelen Instanzen führen. Folglich werden in diesem letzten Schritt parallele Blöcke des Anfragebaumes zusammengefasst, um den Parallelitätsgrad anzupassen. Diese Zusammenführung kann entweder auf Basis von Inter- oder Intra-Operator Parallelität erfolgen.

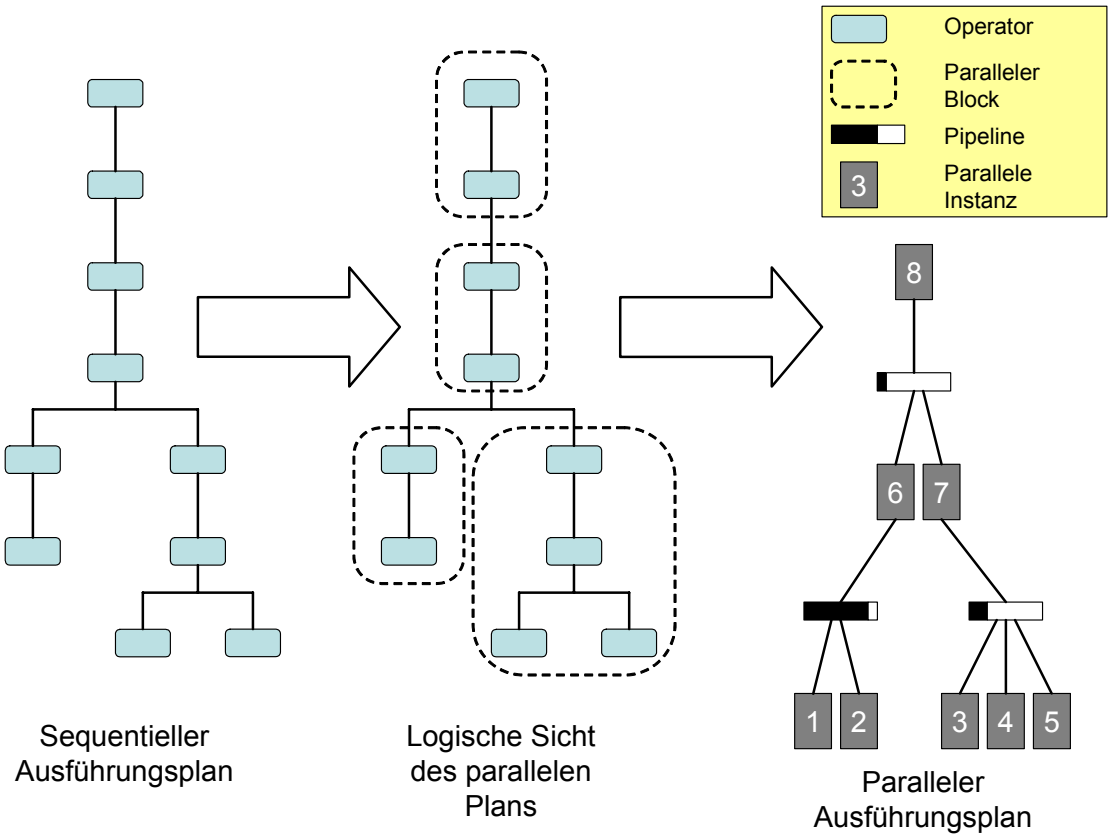


Abbildung 3.6: Phasen der Parallelisierung eines Anfragebaumes

Bezüglich der Integration von paralleler Anfrageverarbeitung in ein existierendes RDBMS sei auf das MIDAS Projekt verwiesen [Nip00] [Jae99] [Zim00] [CJMN+97] [BJLM+96]. Im Rahmen dieses Projekts wurde TransBase, ein kommerziell erfolgreiches RDBMS, um Parallelitätsfunktionalität erweitert.

In Kapitel 4 wird aufgezeigt, dass die Integration des Parallelisierungsmoduls in die Anfrageverarbeitung für Array Daten nicht das primäre Problem war. Typische Optimierungen auf Array Daten bieten sogar eine hervorragende Ausgangsposition für die Parallelisierung, der Optimierer musste folglich nicht angepasst werden. Die Auswirkungen der Parallelisierung auf das Ausführungsmodul waren schwerwiegender, konnte jedoch auch hier durch Definition neuer Knoten im Anfragebaum fast komplett gekapselt werden.

Der in Kapitel 4 vorgestellte Algorithmus zur Parallelisierung des Anfragebaumes ist nicht kostenbasiert, sondern beruht auf Heuristiken. Er ist somit sehr effizient und resultiert in ebenso guten parallelen Ausführungsplänen. Die Definition eines Kostenmodells und eine er-

schöpfende Suche nach einem optimalen parallelen Plan ist für die statischen und einfach strukturierten Anfragebäume nicht ratsam.

3.5 Zusammenfassung

Dieses Kapitel gibt einen komprimierten „State of the Art“ über Parallelität in relationalen DBMS. Mangels Forschungsergebnissen bezüglich der parallelen Verarbeitung von multidimensionalen Array Daten kann in späteren Kapiteln meist keine Abgrenzung der eigenen Forschungsergebnisse zu verwandten Ansätzen vorgenommen werden. Der einzige Vergleichsmaßstab bleibt die parallele Verarbeitung von relationalen Daten. Dieses Kapitel diente folglich dazu, eine Basis zu erhalten für das Verständnis von Gemeinsamkeiten, von nötigen Adaptionen der Algorithmen sowie von grundlegenden Unterschieden. Im Detail wurden folgende Fragen gestellt:

1. **Wie misst man die Qualität** der Parallelität einer bestehenden Implementierung? Wo liegen die Grenzen der Performanzsteigerung durch Parallelität? Die Qualitätskriterien für parallele Performanz werden insbesondere in Kapitel 6 als Bewertungsmaßstab für die Implementierung der parallelen Analyse von Array Daten benutzt.
2. **Welche Rechnerarchitekturen** werden für parallele Datenverarbeitung durch RDBMS eingesetzt? Welche Vor- und Nachteile haben diese Architekturen? Welche Relevanz haben sie in der Praxis? Die parallele Verarbeitung von Array Daten wird hier einerseits keinerlei Einschränkungen bezüglich der Architektur unterliegen. Jedoch wurden bekannte Nachteile der Architekturen bereits bei der Entwicklung der Algorithmen berücksichtigt.
3. **Welche Methoden** der parallelen Anfrageausführung gibt es in RDBMS? Was wird parallelisiert, wie wird parallelisiert? Die vorliegende Arbeit konzentriert sich auf die Parallelisierung der Arbeitslast, weniger auf parallele E/A oder Datenfragmentierung. Schließlich wurde mangelnde Effizienz der Analyse von Array Daten in der Praxis als Problem identifiziert und ist somit die Motivation für diese Dissertation. Die Methoden für die Verteilung der Arbeitslast werden in Kapitel 4 auf Basis der in diesem Kapitel vorgestellten Algorithmen adaptiert oder komplett neu entwickelt, wenn bekannte Konzepte nicht greifen.
4. **Wie werden konkret parallele Ausführungspläne erzeugt** und abgearbeitet? Wie wird Parallelität in existierende Anfrageverarbeitung integriert? Was waren die primären Herausforderungen der Adaptierung existierender DBMS? Bezüglich unserer Arbeit stellt sich die Frage: Sind die Herausforderungen für die Parallelverarbeitung von Array Daten ähnlich? Die eindeutige Antwort ist „nein“. Viele klassische Probleme der relationalen Parallelisierung, etwa das Pruning von Suchräumen, treten für Array Daten nicht auf. Viele Strategien, die dagegen durch diese Arbeit entwickelt werden, finden in der Literatur zu relationaler Parallelität keine Entsprechung. Ein Beispiel hierfür ist ein optimaler Algorithmus für die Aufteilung eines einzelnen Arrays.

Kapitel 4 Parallele Verarbeitung von Array Daten in einem DBMS

Performance is your reality. Forget everything else.
(Harold Geneen)

Parallele Anfrageverarbeitung wurde in relationalen DBMS eingehend wissenschaftlich untersucht und viele Konzepte sind in relationalen DBMS bereits integriert. Die Erfahrung der letzten zwei Jahrzehnte hat jedoch auch gezeigt, dass es keine klaren Gewinner der Parallelisierung gibt, sei es bezüglich Architekturen oder Algorithmen. Für die Wahl der optimalen Architektur müssen sowohl die Anforderungen als auch verfügbare Umgebung (etwa das vorhandene Netzwerk) in Betracht gezogen werden. Aus Sicht der DBMS Hersteller wird eine Option auf den Einsatz von Rechnercluster, also Shared-Nothing Architekturen, insbesondere wegen der deutlich besseren Erweiterbarkeit, immer wichtiger, aber in der Praxis werden weiterhin Multiprozessorrechner als Datenbankserver häufig eingesetzt (siehe etwa [TPC]). Diese Anforderung, also Einsatz auf einem Multiprozessor-Server mit Option auf den Einsatz auf Rechnerclustern, zeigte sich auch bezüglich der parallelen Analyse von Array Daten im Projekt ESTEDI (siehe Kapitel 1.4). Kurz und mittelfristig wurde von den beteiligten Projektpartnern der Einsatz eines Multiprozessor-Datenbankserver realisiert, die Möglichkeit, das DBMS mittel- und langfristig auf einem Cluster besser skalieren zu können, wurde jedoch ausdrücklich erwünscht.

Während sich also bezüglich der Architektur die Situation für relationale und Array DBMS gleicht, zeigen sich die Voraussetzungen für die Parallelisierungsalgorithmen unterschiedlich: die besondere Struktur und Granularität von Array Daten sowie die besondere Anfrageverarbeitung erfordern eine Neubewertung der Algorithmen. Somit ist auch die Struktur des folgenden Kapitels vorgegeben: nach einer **Analyse von Array Daten in Kapitel 4.1** und der daraus resultierenden Besonderheiten gegenüber relationalen Daten werden in Kapitel 4.2 und 4.3 die wichtigsten relationalen Parallelisierungsmethoden auf ihren Einsatz für Array Daten untersucht.

Inter-Operator Parallelität (Kapitel 4.2), also die Verarbeitung verschiedener Operatoren im Anfragebaum von verschiedenen Prozessen, wird in relationalen DBMS vor allem wegen Problemen der Lastverteilung meist nur sekundär verwendet ([Hal97] [IBM98]). Für Array Daten entstehen mit diesem Verfahren weitere Nachteile, etwa durch die Komplexität der Daten und deren Übertragung.

Die **Intra-Operator Parallelität (Kapitel 4.3)** bzw. Datenpartitionierung, bereits in relationalen DBMS die primär benutzte Methode, zeigt für Array Daten weitere Vorteile, zum Beispiel sind dynamische Verteilungsstrategien aufgrund der Größe der Datenelemente sehr effizient.

Intra-Objekt Parallelität (Kapitel 4.4), also das Aufteilen eines einzelnen Datenelements zur parallelen Bearbeitung, ist für relationale Daten in den seltensten Fällen sinnvoll. In der Regel ist ein relationales Tupel so klein, dass eine parallele Bearbeitung keinen Gewinn bringt. Lediglich der Spezialfall einer parallelen Verarbeitung von großen binären Objekten in Relationen (LOB, engl.: Large Objects, große Objekte) wurde untersucht (siehe etwa [JM98] [JM99] [CO98] [OCL99]). Gegen eine parallele Verarbeitung spricht, dass die Verarbeitung dieser Daten meist mit UDF (engl.: user defined functions, Benutzerdefinierte Funktionen) geschieht, wobei Datenabhängigkeit nur schwer zu bestimmen ist, was eine Datenpartitionierung mitunter unmöglich macht. Da für Array Daten die (teuren) Operationen und die Eingabeströme exakt klassifizierbar sind, ist eine solche Intra-Objekt Parallelität weitaus einfacher zu realisieren. Die Analyse der Operatoren bezüglich dieser Parallelisierungsstrategie sowie von Partitionierungs- und Vereinigungsstrategien für einzelne Datenobjekte wird in Kapitel 4.4 behandelt.

Die grundlegende Strategie für die parallele Verarbeitung einer Anfrage an Array Daten wird in Abbildung 4.1 skizziert. Eine Analyse der Anfrage bzw. die Analyse der darin referenzierten Kollektion(en) und der Operationen zeigt, ob eine Menge von Arrays oder ein einzelnes Array verarbeitet wird. Im ersten Fall werden die in Kapitel 4.3 beschriebenen Konzepte zur Parallelisierung angewandt. Wird nur ein Array verarbeitet, was etwa durch Referenzierung einer einzelnen Kollektion mit einem einzelnen Objekt oder durch Einschränkung in der Selektion gegeben sein kann, werden die in Kapitel 4.4 beschriebenen Techniken für die Parallelisierung eingesetzt. Prinzipiell ist das dort gezeigte Aufteilen von Arrays auch für jedes Objekt einer Menge von Arrays möglich, jedoch sichert das Verteilen von kompletten Objekten eine unabhängige Verarbeitung und ist daher vorzuziehen.

Die in Abbildung 4.1 skizzierten Algorithmen zur weiteren Parallelisierung innerhalb dieser zwei Klassen werden in den entsprechenden Kapiteln erläutert. Diese in den Kapiteln 4.3 und 4.4 vorgestellten Algorithmen für parallele Anfrageverarbeitung in einem Array DBMS sind als Heuristiken zu sehen. Die einfache Struktur der Anfragen[27] macht eine Parallelisierung auf Basis eines Kostenmodells fragwürdig. Bereits bei der Realisierung eines Optimierers für das rasdaman Array DBMS wurde aus diesen Erwägungen heraus auf ein kostenbasiertes Optimierungsmodul verzichtet [Rit99]. Die beschriebenen Heuristiken zur Parallelisierung basieren vor allem auf den in Kapitel 4.1 diskutierten Eigenheiten von Array Daten und der daraus resultierenden Verarbeitung dieser Daten.

[27] Vor allem durch das weitgehende Fehlen von geschachtelten Anfragen, siehe 4.3.2

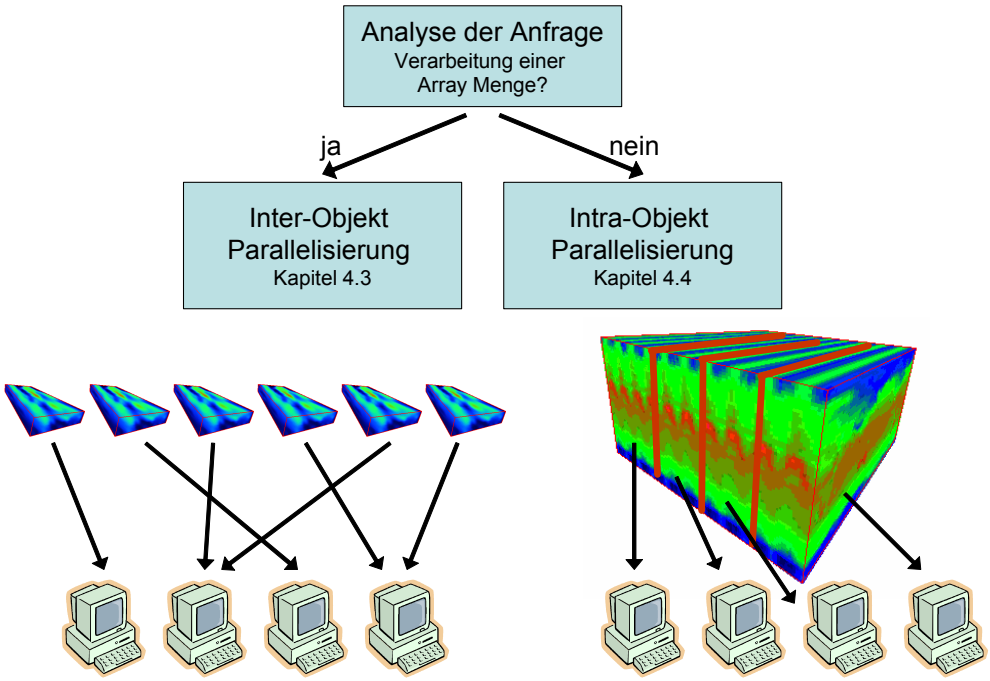


Abbildung 4.1: Generelle Unterscheidung der Parallelisierungsstrategien

Der allgemeine **Algorithmus für die Wahl der Parallelisierungsstrategie** wird in **Kapitel 4.5** vorgestellt. Zu diesem Zweck erfolgt eine Analyse der zu erwartenden parallelen Performanz bezüglich Anfrage und Parallelitätsgrad. Ferner wird eine Kombination verschiedener Parallelisierungsstrategien diskutiert.

4.1 Besonderheiten gegenüber paralleler Verarbeitung von relationalen Daten

Unterschiede in der Verarbeitung von relationalen und Array Daten und somit auch in der parallelen Verarbeitung sind vor allem in den **Unterschieden der Daten** selbst begründet. Tabelle 4.1 skizziert die Gegensätze in typischer Datenstruktur und Datengröße, Lade- und Übertragungsgranularität sowie die unterschiedlichen Operationen und den daraus resultierenden Anfragebäumen[28].

Relationale Daten sind in der Regel in ihrer Struktur einfach. Eine Relation ist eine (per se ungeordnete) Menge von strukturell gleichartigen (da durch Typ der Domänen bestimmten) Datenelementen, genannt Tupel. Jedes Tupel besitzt eine durch die Definition der Relation vorgegebene Menge von Attributen gegebener Datentypen (bzw. NULL). Die Struktur ist bei Array Daten deutlich komplexer (ein detailliertes Beispiel folgt in Kapitel 4.1.1).

Bei relationalen Daten besitzt ein logisches Datenelement, also ein Tupel, eine typische Größe von 20 Byte bis zu 1 KB (eine Ausnahme bilden LOBs). Die Granularität des physikalischen Zugriffs, also die Größe der vom Sekundärspeicher zu ladenden Daten, ist ein Vielfaches davon, nämlich eine Plattenseite, meist 4, 8 oder 16KB groß. Die Übertragung der Daten zwischen parallelen Prozessen erfolgt meist in Seiten bzw. dem Bruchteil einer Seite. Dem gegenüber steht bei Array Daten ein logisches Datenobjekt MDD mit einer typischen Größe

[28] Die angegebenen Werte sind Erfahrungswerte, die für spezielle Anforderungen unter- bzw. überschritten werden können

von 1MB bis 20GB[29] bei einer physikalischen Granularität von 128KB bis 16MB. Die Übertragung der Daten erfolgt als MDD bzw. partielles MDD.

	Relationale Daten	**Array Daten**
Logisches Datenelement	Relationales Tupel	MDD
Typische Größe eines Elements	20 Byte – 1KB	1MB – 20GB
Struktur der Datenelemente	Einfach	Komplex
Physikalische Zugriffsgranularität[30]	Seite: 4 – 16KB	Kachel: 128KB – 16MB
Übertragungsgranularität[31]	Seite bzw. ½, ¼ Seite	(Partielles) MDD
Kardinalität der Mengen	Meist sehr groß ($>10^6$)	Klein (<100)
Anfragebaum	Komplex, viele Operatoren	Einfach
Prädikatbaum (eines Operators)	Meist einfacher Aufbau	Komplexer Aufbau

Tabelle 4.1: Daten und Operatoren: Vergleich von Relationen und Arrays

Im Gegensatz zu relationalen Operatoren besitzen Array Operatoren und damit ihre Prädikatbäume eine deutlich komplexere Struktur, andererseits sind relationale Anfragebäume typischerweise sehr viel höher und verzweigter.

4.1.1 Komplexität der Daten und der Datenübertragung

Multidimensionale Array Daten haben in der Regel eine **deutlich komplexere und weitaus dynamischere Struktur als relationale Daten**:

1. Eine Kollektion enthält Arrays gleichen Typs (Domäne und Zelltyp)[32].
2. Die Domäne einer Kollektion ist dabei völlig beliebig definierbar, sowohl in der Dimensionalität wie auch in der räumlichen Lage und Ausdehnung.
3. Der Zelltyp kann ebenso frei gewählt werden. Auch ein beliebig tief strukturierter Zelltyp ist erlaubt (jede Variable kann einen strukturierten Zelltyp enthalten, welcher seinerseits strukturierte Zelltypen enthält).
4. Jedes MDD besteht aus einer beliebigen Menge von räumlich disjunkten, unterschiedlich großen Kacheln, welche bereits in den Hauptspeicher geladen sein können (transient) oder noch unverändert in der relationalen Datenbank auf Sekundär- oder Tertiärspeicher liegen (persistent). Die Größe der Kacheln kann hierbei auch in den jeweiligen MDD unterschiedlich sein. Ferner kann sich Lage und Größe der Kacheln während der Anfragebearbeitung oder auch durch eine vom Benutzer angestoßene Neukachelung ändern.
5. Jede Kachel kann Daten auch in komprimierter Form enthalten, d.h. die Daten wurden durch Datenkompressionsalgorithmen bezüglich des Speicherbedarfs reduziert. Die verwendete Methode muss hierbei innerhalb eines MDD nicht identisch sein, jede Kachel des Objekts kann theoretisch einen anderen Kompressionsalgorithmus (etwa ZLib, RLE, Wavelet, etc.) und hierfür eigene Kompressionsparameter nutzen.

[29] Theoretisch ohne Größenbeschränkung nach oben. In der Praxis beschränkt der zur Verfügung stehende virtuelle Speicher die Größe eines MDD. Für derzeit aktuelle DBMS Server scheint der Wert sinnvoll.

[30] Kleinste Einheit, die von Sekundärspeicher geladen werden kann

[31] Übertragung von Daten zwischen parallelen Prozessen

[32] Im Array DBMS rasdaman wird die Einschränkung einer identischen Domäne aufgehoben, vor allem zur Realisierung eines inkrementellen Ladens der Daten. In der Praxis führt diese Aufweichung des Datenmodells zu Problemen in der Anfrageverarbeitung, so sind viele Operationen auf den resultierenden Arraymengen nicht mehr definiert (induzierte Operationen, geometrische Operationen) oder es werden verfälschte Ergebnisse erzeugt (Aggregationsoperatoren).

Punkt 1 bis 3 erlaubt eine komplexe Datendefinition, jedoch sind die so definierten Objekte dann statisch, das heißt Größe und Struktur ändert sich nach der Definition nicht mehr. Die Punkte 4 und 5 führen jedoch dazu, dass Arrays in ihrer Struktur (auch während der Verarbeitung) dynamisch sein können. Abbildung 4.2 zeigt ein Beispiel für die Struktur eines MDD. Man erkennt ein bereits in den Hauptspeicher geladenes Objekt mit zusammengesetztem Zelltyp (*StructTyp*) und mehreren Kacheln. Die Abbildung skizziert die Implementierung eines solchen komplexen Objektes mit Verweisen (Pointer, hier in C Notation als *) auf weitere dynamische Objekte. Die Abbildung soll lediglich einen Eindruck von der Komplexität eines Datenobjekts vermitteln, die Bedeutung der einzelnen Komponenten ist an dieser Stelle nicht von Bedeutung.

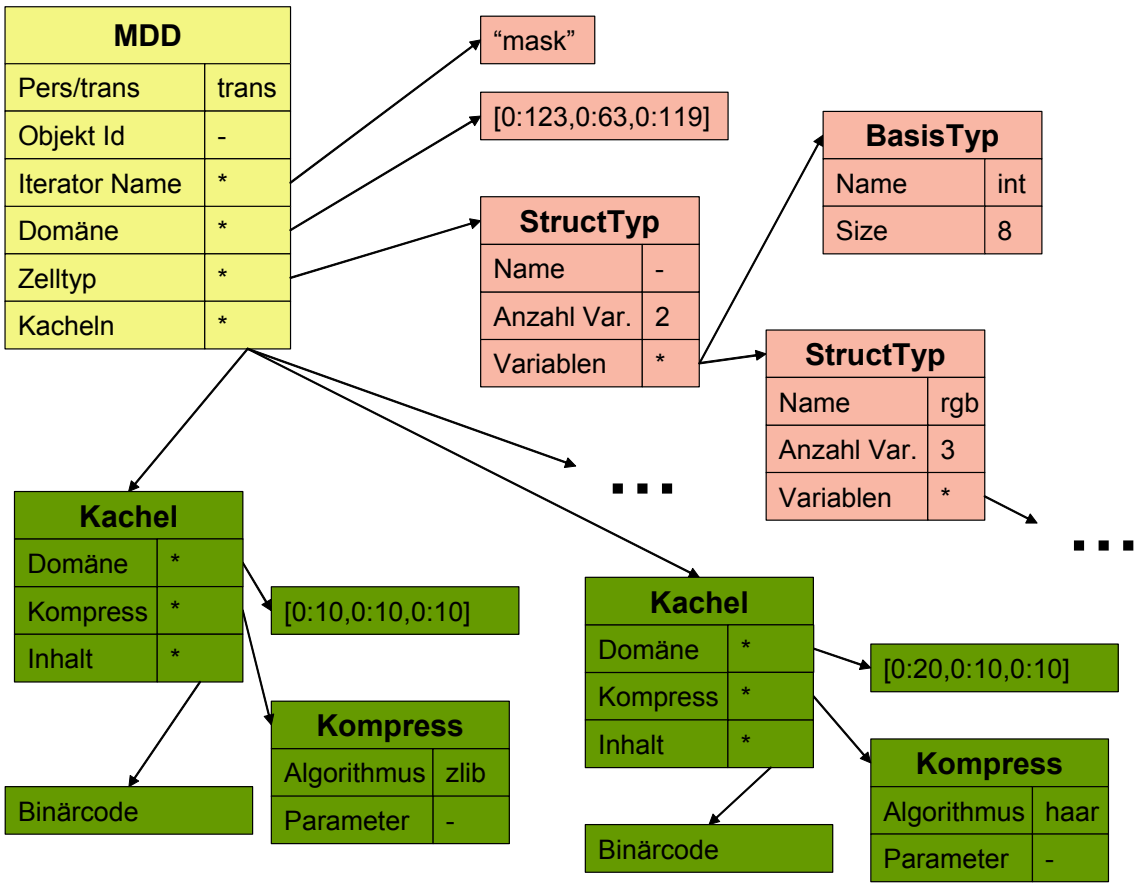


Abbildung 4.2: Beispiel für die komplexe Struktur von Array Daten

Diese äußerst komplexen, oftmals sich dynamisch ändernden Objekte zwischen Prozessen zu übertragen, erweist sich zum einen in der Implementierung als aufwändig und fehleranfällig, zum anderen beeinträchtigt das Präparieren des Objektes vor dem Versenden bzw. die Rekonstruktion nach dem Empfang durch einen anderen Prozess die parallele Performanz der Implementierung spürbar. Die inhärente Komplexität von Array Daten führt folglich dazu, dass die **Übertragung solcher Daten**, insbesondere aber von transienten Objekten, also Arrays, die sich bereits komplett im Hauptspeicher befinden, wenn möglich **vermieden werden muss**. Der Transfer von Objekten, die nur referenziert, aber noch nicht in den Speicher geladen wurden, wirkt sich auf die Performanz weniger deutlich aus.

Diese erste Analyse der Array Daten führt zu folgender Prämisse, die im Folgenden Auswirkungen auf die Entwicklung der Parallelisierungsalgorithmen und deren Implementierung hat.

Prämisse 1	Vermeide den Transfer von Arrays Der Transfer von transienten multidimensionalen Objekten muss vermieden werden. Neben den Transferkosten selbst ist vor allem das Präparieren für den Transfer und das Auspacken und Rekonstruieren des Objekts sehr aufwändig und damit inperformant.

Eine erste **Folgerung dieser Prämisse** besteht darin, dass Parallelisierungsalgorithmen, die einen geringen Datentransfer benötigen, gegenüber Methoden mit hohem Transfervolumen vorzuziehen sind. So ist etwa die Methode der Pipeline Parallelität, welche Zwischenresultate prinzipiell transferiert und in einem Puffer zwischenspeichert, für Array Daten ungeeignet (Kapitel 4.2).

Aber auch Algorithmen, die im Vergleich einen deutlich geringeren Datentransfer gewährleisten, müssen bezüglich der zu transferierenden Daten entwickelt und bewertet werden. Diese zweite Folgerung führt zu Algorithmen in den Kapiteln 4.3 und 4.4, die bevorzugt eine Verteilung der Arbeitslast vor dem eigentlichen Laden der Daten realisiert. Ebenso werden Resultate von parallelen Instanzen wenn möglich dann verwertet, wenn eine Reduktion des Datenvolumens stattgefunden hat.

4.1.2 Heterogenität der Operatorbäume

Die Anzahl der definierten Operatoren ist für Array Mengen geringer als bei relationalen Daten[33]. Des Weiteren sind verschachtelte Anfragen für Array Daten untypisch (siehe Kapitel 4.3.2.3). Dies führt zu **Anfragebäumen, die sehr einfach strukturiert sind**. Die Höhe ist auf die Anzahl der Mengenoperatoren beschränkt. Das Fehlen eines binären Operators hoher Komplexität wie der relationale Join Operator verhindert ferner breite Bäume. Einen typischen Anfragebaum für Array Daten zeigt Abbildung 4.4.

Jedoch sind die einzelnen **Operatoren deutlich komplexer** als im relationalen Fall. Die Tatsache, dass ein Operator selbst einen Baum von Operationen, also einen Prädikatbaum, abarbeitet, wird bei relationalen Operatoren meist nur sekundär wahrgenommen, da dieser meist wenig komplex ist. Beispielsweise beinhaltet eine relationale Selektion $\sigma_{Id=99}$ einen Prädikatbaum, der aus 3 Knoten besteht (Attribut, Gleichheitsoperator und Konstante). Prädikate von Array Operatoren sind dagegen deutlich komplexer. Ein Beispiel für diesen Sachverhalt ist in Abbildung 4.3 skizziert. Auf der linken Seite ist ein relationaler Selektionsoperator mit besagtem Prädikatbaum dargestellt, rechts sieht man einen typischen Prädikatbaum für Array Daten, welcher multidimensionale Operationen beinhaltet, nämlich Basistypprojektion *dot*, induzierte Quadratwurzel *sqr*, induzierte Addition +, geometrische Einschränkung der Domäne *sdom*, Aggregation auf Durchschnittswert *avg* und induzierter Vergleich >.

Operatorbäume für die Verarbeitung von Array Daten gliedern sich also in einen (quasi-) relationalen Teil, der auf Kollektionen und in einen Bereich mit Array Operationen, der aus einem oder mehreren Prädikatbäumen besteht (siehe auch Abbildung 4.4). Der relationale Bereich zeigt sich hierbei gegenüber relationalen Anfragenbäumen als einfach strukturiert während die Operationen eine deutlich komplexere Struktur besitzen.

Als **erste Folgerung** dieser Analyse hat Inter-Operator Parallelität im Gegensatz zu relationalen DBMS eine untergeordnete Bedeutung (Kapitel 4.2). Die Realisierung einer Lastvertei-

[33] Da auf Mengen von multidimensionalen Objekten keine Ordnungsrelation definiert ist, fehlen vor allem Operatoren, die auf solch einer Ordnung basieren, wie beispielsweise Sortierung, Gruppierung, Join, etc.

lung ist in einfach strukturierten Bäumen mit wenigen Operatoren nicht zu verwirklichen. Intra-Operator Parallelität hingegen ist für Array Daten hervorragend geeignet und kann wegen der Größe der Datenobjekte und der Komplexität der Operatoren nicht nur auf Granularität von gesamten Arrays, sondern auf Teilbereichen von diesen basieren (Kapitel 4.3 bzw. 4.4).

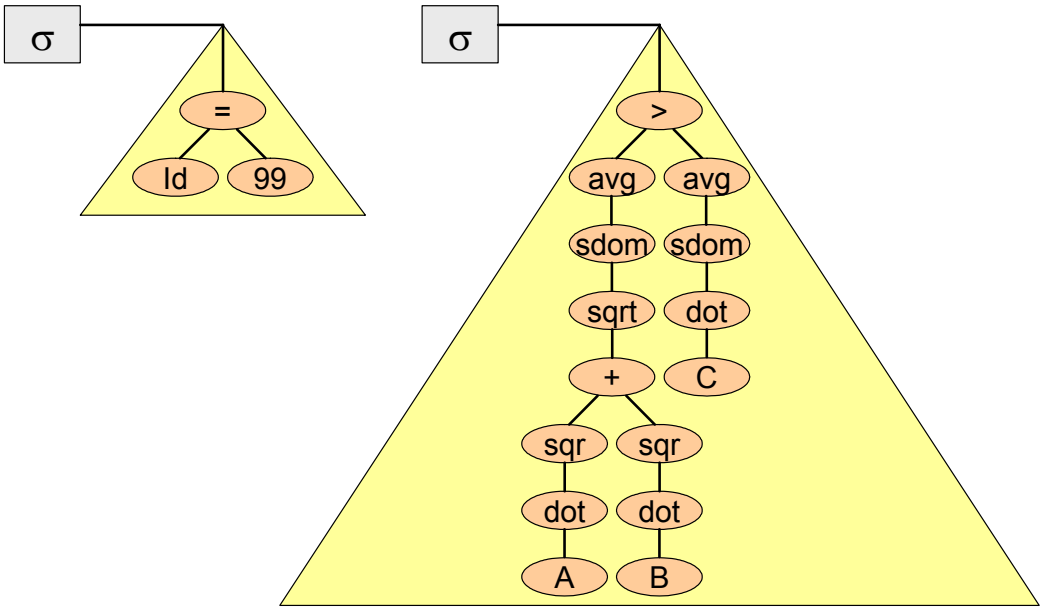


Abbildung 4.3: Vergleich von Prädikatbaum des Selektionsprädikats für relationale Selektion und Selektion von Arrays

Eine zweite Tatsache führt zu einer steigenden Bedeutung der Prädikatbäume. Die **Kardinalität von Array Mengen** ist, nicht zuletzt wegen der Größe der Elemente, geringer als die Kardinalität von relationalen Mengen. So waren etwa im Projekt ESTEDI (Kapitel 1.4) Array Mengen der Kardinalität 100 die obere Schranke. Werden wenige oder sogar nur ein einzelnes Array analysiert, verlieren die mengenbasierten Operatoren als Grundlage der Parallelisierung ihre Bedeutung, da sie die parallele Hardware nicht effizient nutzen können. Als **Folge** muss eine Parallelisierung von Einzelobjekten realisiert werden, was in einer parallelen Verarbeitung von Prädikatbäumen resultiert.

Diese Analyse führt dazu, dass für die parallele Verarbeitung von Array Daten besonderes Augenmerk auf die Verarbeitung von Prädikatbäumen und somit von Einzelobjekten gelegt wird.

Prämisse 2	Forderung nach objektbasierter Parallelisierung Parallele Verarbeitung von Mengen von Arrays kann sich nicht allein auf mengenbasierte Parallelisierungsstrategien stützen. Die Größe von Objekten, eine geringe Kardinalität von Array Mengen und komplexe Array Operationen erfordern Intra-Objekt Parallelität, welche Operationen auf einem Einzelobjekt parallel verarbeitet.

4.1.3 Granularität der Datenverarbeitung und des Datenflusses

Die Wahl der Parallelisierungsstrategie als auch die detaillierte Implementierung der gewählten Strategie, etwa bezüglich der Adaption des Anfragebaumes zur Verteilung auf parallele Instanzen, erfordert eine Analyse der Datenverarbeitung und damit auch des Datenflusses innerhalb des Anfragebaumes. Die hier beschriebene Anfrageausführung ist auf **rasdaman** DBMS bezogen, lässt sich jedoch beliebig auf Operatorbäume für die Verarbeitung von Mengen extrem großer Objekte verallgemeinern.

Wie bereits gezeigt, unterteilen sich Anfragebäume, die die Analyse einer Menge von Objekten vornehmen, in der Regel in Operatoren, die über dieser Menge iterieren (daher der Name Iteratoren) und Operatoren, die jeweils dieses gerade referenzierte Objekt bearbeiten. Logisch gesehen verarbeiten Iteratoren eine Menge von Eingabeobjekten und ergeben eine Menge von Ausgabeobjekten. Implementierungstechnisch referenzieren die Iteratoren zu jedem Zeitpunkt genau ein Objekt. Der **Datenfluss bei Iteratoren** erfolgt hierbei typischerweise, indem die Resultate eins nach dem anderen von den Blättern des Baumes zu der Wurzel gereicht werden. Das zwischen den Iteratoren übergebene Objekt ist in unserem Fall ein MDD oder ein durch ein Kreuzprodukt erzeugtes Tupel von MDD.

Die Objekte, die hierbei durch den Anfragebaum von unten nach oben gereicht werden, durchlaufen hierbei in der Regel verschiedene Phasen der **Referenzierung durch den Iterator**. Solange ein Array nicht für eine Berechnung (oder für einen Transfer zum Client) im Speicher materialisiert werden muss, spricht man von persistenten Daten[34]. Die Existenz der Daten im RDBMS wird durch den Zugriffsiterator ω sichergestellt, allerdings wird lediglich eine Referenz, also ein persistentes Array, im Anfragebaum weitergereicht. Diese Optimierung wird vorgenommen, da ein komplettes Laden der Daten für die Anfrage in den meisten Fällen nicht nötig ist. In Abbildung 4.2 entsprechen die mit „Binärcode“ bezeichneten Strukturen dem Inhalt der multidimensionalen Arrays. Alle zusätzlichen Informationen bezüglich Größe, Kompression, etc., die nur einen Bruchteil der Größe ausmachen, bilden ein persistentes Objekt. Auch die Iteratoren ϕ und $\times$ arbeiten auf persistenten Daten. In Abbildung 4.4 ist die Übergabe von persistenten Arrays zwischen Iteratoren als Pfeile mit unterbrochener Linie skizziert.

Die Arrays werden (eventuell teilweise) im Hauptspeicher benötigt, sobald induzierte oder aggregierende Array Operationen darauf ausgeführt werden oder aber zur Vorbereitung des Datentransfers zum Client. Sie werden aus dem RDBMS geladen und gehen in einen transienten Zustand über. Hierbei wird immer eine Kopie der ursprünglich geladenen Daten in einem internen Speicherbereich (engl.: Cache) gehalten, auch wenn die Daten durch Berechnungen verändert werden. Diese Optimierung basiert auf der Tatsache, dass in der Selektion analysierte Bereiche von Arrays meist in der Applikation weiter für einen Transfer aufbereitet werden. So wird etwa in Abbildung 4.4 in der Selektion das komplette Array geladen und in der Applikation ein Bereich dieses Arrays weiter analysiert.

In der Applikation α erfolgt in den meisten Fällen eine Reduktion des Datenvolumens, entweder durch Formatkonvertierung, Skalierung oder Aggregationsfunktion, um einen effektiven Transfer zum Client zu ermöglichen.

Effektive Parallelisierungsalgorithmen, die auf diesem Bereich des Anfragebaumes arbeiten, also eine Menge parallel verarbeiten, müssen den Datenfluß und die Datenreferenzierung be-

[34] Ein persistentes („dauerhaftes“) MDD hat noch keine Kacheln in den Hauptspeicher geladen (siehe Definition 2.8), ist also persistent im RDBMS gespeichert.

achten. Wie in Kapitel 4.3 gezeigt wird, kann die parallele Verarbeitung und die dadurch nötige Inter-Prozess Kommunikation sehr gut in das Iteratorkonzept integriert werden. Die Zuordnung von Iteratoren zu parallelen Instanzen muss jedoch so erfolgen, dass eine Inter-Prozess Kommunikation zwischen Iteratoren mit transienten Datenfluss vermieden wird. Die in Kapitel 4.3 vorgestellte Parallelisierungsstrategie nutzt vielmehr die typische Datenreferenzierung während der Anfrageverarbeitung aus und transferiert Daten zwischen parallelen Instanzen möglichst vor einer Materialisierung oder nach einer Reduktion des Volumens. Ferner darf eine Aufteilung von Iteratoren im Anfragebaum auf parallele Instanzen keine Caching Effekte zerstören, da dies die Performanz beeinträchtigt.

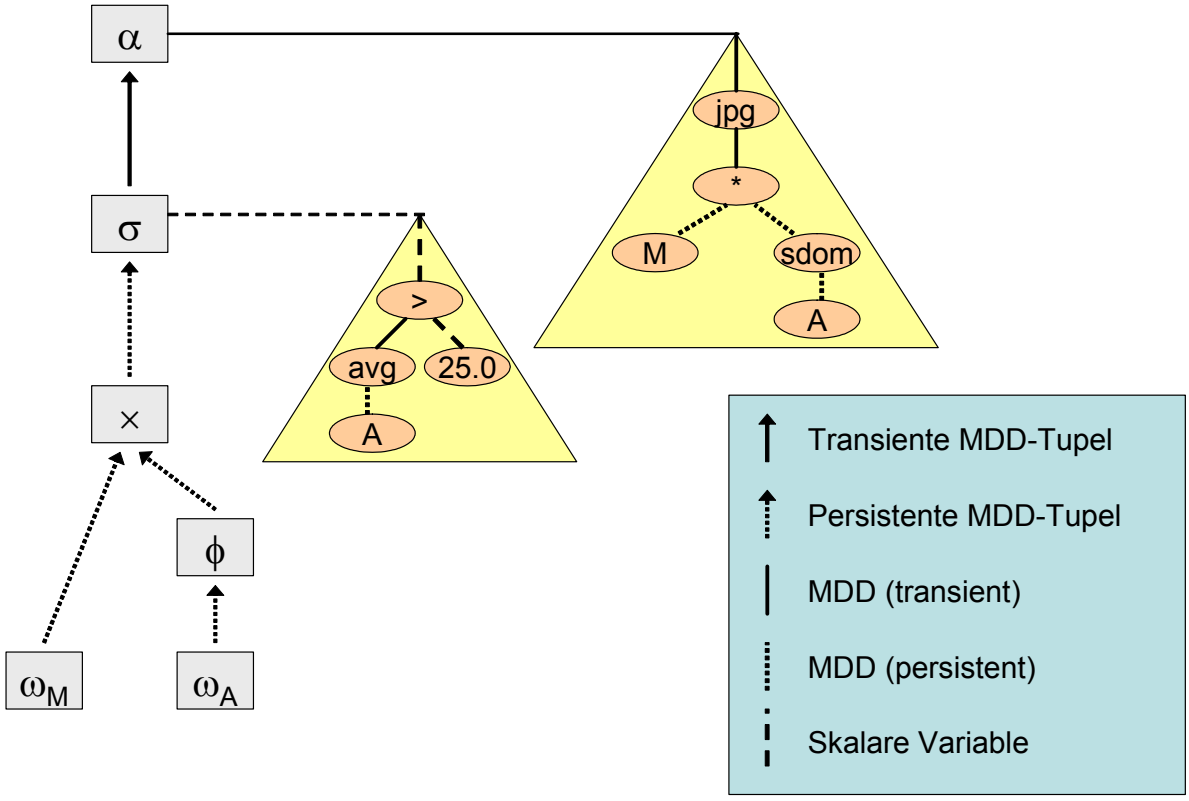


Abbildung 4.4: Datenfluss im Anfragebaum

Der **Datenfluss bei Array Operatoren** (rechte Seite des Anfragebaumes in Abbildung 4.4) erfolgt nicht nach dem Iteratorkonzept. Wird ein Operator aufgerufen, holt er sich von seinem Kindknoten das komplette Zwischenergebnis und verarbeitet es in einem Schritt. Man kann auch diesen Teil des Anfragebaumes in transiente, persistente und skalare Bereiche unterscheiden. Solange keine Dateninhalte benötigt werden, sind nur persistente Daten referenziert. In Abbildung 4.4 realisieren die Zugriffsoperatoren A und M eine Bindung an die vom Iterator gelieferten aktuelle persistenten MDD. Geometrische Operatoren, wie im Beispiel die Einschränkung der Domäne durch *sdom*, laden keine Dateninhalte. Erst induzierte oder aggregierende Operatoren arbeiten auf dem Inhalt und liefern als Ergebnis ein transientes MDD oder einen skalaren Wert. In unserem Beispiel kann die induzierte Multiplikation auf transiente Daten aus dem Cache zurückgreifen.

Auch die **Datenreferenzierung in den Prädikatbäumen** mit Array Operationen erfolgt ähnlich wie bei den Iteratoren: solange wie möglich wird mit persistenten Daten, d.h. auf den

Metadaten, gearbeitet. Nach einer Materialisierung für Berechnungen erfolgt in den meisten Fällen eine Datenreduktion (in der Selektion σ zu einem booleschen Skalar).

Die in Kapitel 4.4 beschriebene Parallelisierungsstrategie, die Array Operationen durch Partitionierung des Eingabearrays parallelisiert, analysiert den Datenfluss in diesem Bereich des Anfragebaumes. Vereinfacht ausgedrückt ist es unvorteilhaft, ein großes Objekt in einer parallelen Instanz zu laden, um es dann sofort an eine weitere parallele Instanz zu versenden. Folglich werden die Partitionen für die Datenparallelität gebildet und an Prozesse transferiert, bevor die Objekte in einen transienten Zustand übergehen. Ebenso werden Teilresultate von parallelen Instanzen nach einer Datenreduktion gesammelt.

Prämisse 3	Beachte Optimierungen bei der Parallelisierung Die Analyse von Datenflüssen und der Objektreferenzierung vor der Umsetzung der Parallelisierungsstrategie reduziert E/A durch Vermeiden von Transferkosten und bewahrt Konzepte wie Caching.

Die Granularität der Verarbeitung ist auch für den Bereich der Array Operatoren ein Array Objekt. Diese relativ grobe Granularität der Verarbeitung führt zu weitreichenden Konsequenzen, wie etwa die Einschränkung, dass nur Arrays verarbeitet werden können, die nicht größer als der verfügbare Speicher sind. Zur Vermeidung solcher Einschränkungen bietet sich ein Iteratorkonzept auf Basis von Kacheln an. Die Abarbeitung eines Prädikatbaumes geschieht dann mittels Iteration über der Menge der Kacheln. Ein Nachteil dieses Konzeptes besteht darin, dass ein Iterator auf solch kleiner Granularität eine Vielzahl an Funktionsaufrufen verursacht (jede Anforderung der nächsten Kachel vom Kindknoten ist ein Funktionsaufruf) und somit die Performanz verschlechtert. Derartige Konzepte wurden in der Vergangenheit für **rasdaman** in Betracht gezogen[Rit99], wurden allerdings nicht implementiert. In Kapitel 4.4 werden die Auswirkungen eines kachelbasierten Iteratorkonzepts auf parallele Verarbeitung aufgegriffen und diskutiert.

Die Einteilung der Operatoren in Iteratoren, die eine Menge elementweise abarbeiten, und Operationen, die auf den einzelnen Elementen definiert sind, sowie die Klassifizierung in persistente, transiente und skalare Bereiche bezüglich des Datenflusses scheint auf den ersten Blick nahe an die Implementierung von **rasdaman** gebunden. Bei näherer Betrachtung erkennt man jedoch, dass die Verarbeitung von großen Datenmengen zwangsweise einem Iteratorkonzept folgen muss, da andernfalls die Nutzung der Ressourcen ineffizient erfolgt. Ebenso ist der beschriebene Datenfluss, d.h. das Konzept der Verzögerung des Ladens, typisch, da nur so Ressourcenvergeudung vermieden wird. Die in den folgenden Kapiteln herausgearbeiteten Schlussfolgerungen und Algorithmen sind also nicht auf Array Daten oder sogar das **rasdaman** DBMS eingeschränkt, sondern adressieren ebenso ähnlich strukturierte Datenmengen, auf die komplexe Berechnungen durchzuführen sind.

In diesem Kapitel wurden drei Prämissen für die parallele Verarbeitung von Array Daten vorgestellt. Die ersten zwei ergeben sich aus den typischen Problemen der Parallelität (Kapitel 3.1). Die Vermeidung des Transfers von Arrays vermeidet hohe Kommunikationskosten, die Forderung nach objektbasierter Parallelisierung vermeidet ungleiche Lastbalancierung für typische Array Anfragen. Die dritte Forderung besagt, dass die Umstellung auf parallele Anfrageverarbeitung bisherige Optimierungen nicht zerstören darf.

Diese Analyse beruht teils auf der Definition des Datenmodells, teils auf mehrjähriger Erfahrung bezüglich der Speicherung und Verarbeitung von Array Daten im rasdaman DBMS im Rahmen des Projekts ESTEDI. Basierend auf dieser Analyse werden nun in den Kapiteln 4.2 und 4.3 relationale Parallelisierungsstrategien evaluiert und gegebenenfalls für Array Daten adaptiert und umgesetzt. In Kapitel 4.4 wird die parallele Verarbeitung von Einzelobjekten untersucht, wie durch Prämisse 2 gefordert. Die dort vorgestellten Algorithmen zur parallelen Verarbeitung haben keine Entsprechung für relationale DBMS.

4.2 Inter-Operator Parallelisierung

Inter-Operator Parallelisierung[35] verarbeitet verschiedene Operatoren des Anfragebaumes parallel. In der Regel werden hierbei nicht einzelne Operatoren parallelen Instanzen zugeordnet sondern der Anfragebaum wird in Blöcke aufgeteilt, die möglichst ähnliche Verarbeitungsgeschwindigkeiten aufzeigen und somit mit völliger Auslastung parallel verarbeitet werden können. Man unterscheidet hierbei zwischen vertikaler und horizontaler Inter-Operator Parallelisierung. Erstere Strategie bildet vertikal angeordnete Blöcke im Anfragebaum, die einen Erzeuger und einen Verbraucher (von Zwischenergebnissen) selektieren, welche gleichzeitig ausgeführt werden. Die Übergabe der Zwischenergebnisse erfolgt mittels Nachrichten und einen Puffer, genannt Pipeline. Horizontale Parallelisierung verarbeitet Unterbäume eines binären Operators (in RDBMS ist das typischerweise ein Join Operator) parallel.

Im Folgenden werden diese Strategien auf ihre Eignung für Array Daten untersucht. Bezogen auf unseren konkreten Anfragebaum für Array Daten, siehe etwa Abbildung 4.4, bewegen wir uns im Bereich der mengenbasierten Operatoren, also der Iteratoren Applikation α, Selektion σ, Kreuzprodukt $\times$, Repartitionierung ϕ und Zugriffsoperator ω. Eine Parallelisierung der Array Operationen, die die Prädikatbäume bilden, in der rechten Seite des Anfragebaumes entspricht der in Kapitel 4.4 beschriebenen Intra-Operator Parallelisierung.

4.2.1 Pipeline Parallelität (Vertikale Inter-Operator Parallelisierung)

Diese Parallelisierungsstrategie basiert auf der Bildung von vertikal angeordneten Operatorblöcken im Anfragebaum mit einer möglichst identischen Verarbeitungsgeschwindigkeit. Diese Blöcke werden Prozessen zugeordnet, die mittels Nachrichten unter Nutzung einer Pipeline Teilergebnissen austauschen. Pipeline Parallelität hat zwei inhärente Probleme (diese entsprechen zwei der drei großen Parallelisierungsprobleme aus Kapitel 3.1):

1. Pipeline Parallelisierung erfordert einen hohen Anteil an Inter-Prozess Kommunikation, da Zwischenergebnisse transferiert werden. Kommunikation zwischen Prozessen ist aus zwei Gründen negativ für die Performanz. Erstens sind dies Zusatzkosten für Parallelisierung, die im Vergleich bei einem sequentiellen System nicht anfallen. Zweitens fallen Kommunikationskosten in den nicht-parallel ausführbaren (sequentiellen) Bereich eines Problems (Amdahls Gesetz, Beispiel 3.1). Die Skalierbarkeit eines Problems mit hohem sequentiellem Anteil ist prinzipiell begrenzt.
2. Lastverteilung ist in Pipeline Parallelität schwer zu realisieren. Das Ideal von Erzeuger und Verbraucher, die in jeder Phase einer Anfrage eine identische Auslastung zeigen, ist nicht erreichbar. Ist der Erzeuger schneller als der Verbraucher, müssen Zwischenergebnisse wegen gefüllter Pipeline auf Sekundärspeicher ausgelagert oder der Erzeuger in seiner Arbeit unterbrochen werden, bis die Pipeline wieder aufnahmebereit ist. Ist der Verbraucher schneller als der Erzeuger, muss der Verbraucher die Bearbeitung mangels

[35] Die Notation ist angelehnt an die parallele Anfrageverarbeitung für relationale Daten. Als Operatoren werden hierbei die mengenbasierten Iteratoren gesehen. Folglich sind in den Kapiteln 4.2 und 4.3 diese Iteratoren gemeint, wenn von Operatoren gesprochen wird. Auf die Parallelverarbeitung von Array Operatoren wird erst in Kapitel 4.4 eingegangen.

Eingabedaten unterbrechen. Eine völlige Auslastung der Ressourcen und damit ein linearer Speed-Up kann unter diesen Umständen nicht garantiert werden.

Beide Nachteile fallen für Array Daten noch deutlicher ins Gewicht als im Falle einer relationalen Anfrageverarbeitung. Da nach Prämisse 1 der Transfer von Arrays vermieden werden soll, werden im Folgenden Operatoren bezüglich der Ein- und Ausgabe klassifiziert. Ferner werden die Operatoren bezüglich ihrer Komplexität analysiert, um die Möglichkeit einer Lastbalancierung zu untersuchen.

Eine Zusammenfassung unserer mengenbasierten Array Operatoren bezüglich Eingabe- und Ausgabetyp, Komplexität, sowie ein Beispiel für einen typischen Zeitbedarf ist in Tabelle 4.2 gegeben. Der konkrete Zeitbedarf hängt natürlich von vielen Faktoren ab, die Abschätzung soll nur beispielhaft eine Vorstellung der Unterschiede in der Laufzeit geben.

	Eingabe	**Ausgabe**	**(Berechnungs-) Komplexität**	**Typischer Zeitbedarf**
Applikation	Tupel persistent Tupel transient	MDD persistent MDD transient Skalar	**O(n*o)**	>100 s
Selektion	Tupel persistent	Tupel transient	**≤ O(n*o)**[36]	>100 s
Kreuzprodukt	i MDD, pers.	Tupel persistent	$O(\prod n_i)$	~10 ms
Repartitionierung	MDD persistent	MDD persistent	O(p)	~10 ms
Kollektionszugriff	-	MDD persistent	O(n)	~500 ms

Tabelle 4.2: Analyse der (quasi-relationalen) Operatoren auf Array Mengen

n = |C| ist für den jeweiligen Array Operator die Kardinalität einer Eingabekollektion von MDD $\{\alpha \mid \alpha \in C\}$ bzw. die Kardinalität $|C| = |C_1|*\ldots* |C_i|$ eines Eingabe-Tupel von MDD $\{\alpha_1 \times \ldots \times \alpha_i \mid \alpha_1 \in C_1, \ldots, \alpha_i \in C_i\}$, $i \in \mathbb{N}$.

o ist die Komplexität der Operation selbst, welche bei induzierten und Aggregationsoperationen von der Anzahl der Zellen c der beteiligten MDD abhängen (diese ist für MDD einer Kollektion konstant).

p ist im Falle der Repartitionierung die Anzahle der resultierenden Partitionen.

Primärer Indikator für Komplexität sind die Kosten der Operation o, da nur in diesem Fall eventuell eine Iteration über alle Zellwerte nötig ist. In Tabelle 4.2 sind Applikation und Selektion markiert, da nur hier die teuren Operationen o auftreten können. Teure Operationen in α und σ sind speziell induzierte und Aggregationsoperationen, da die Komplexität $O(o) = O(o_a*c)$ ist, mit c als Anzahl der Zellen im MDD und o_a als arithmetischer Einzeloperation. Das heißt, induzierte und Aggregationsoperationen bedürfen (meist) einer kompletten Iteration aller Zellwerte für das Ergebnis. Die große Anzahl der Zellwerte c für Array Daten ist folglich die Ursache für die hohe Komplexität dieser Operationen. Die Berechnung einer geometrischen Operation hingegen kann über Metadaten erfolgen und ist somit O(1).

[36] Manche Array Operationen (Quantoren) führen zu einem Resultat, bevor das komplette Array untersucht wurde.

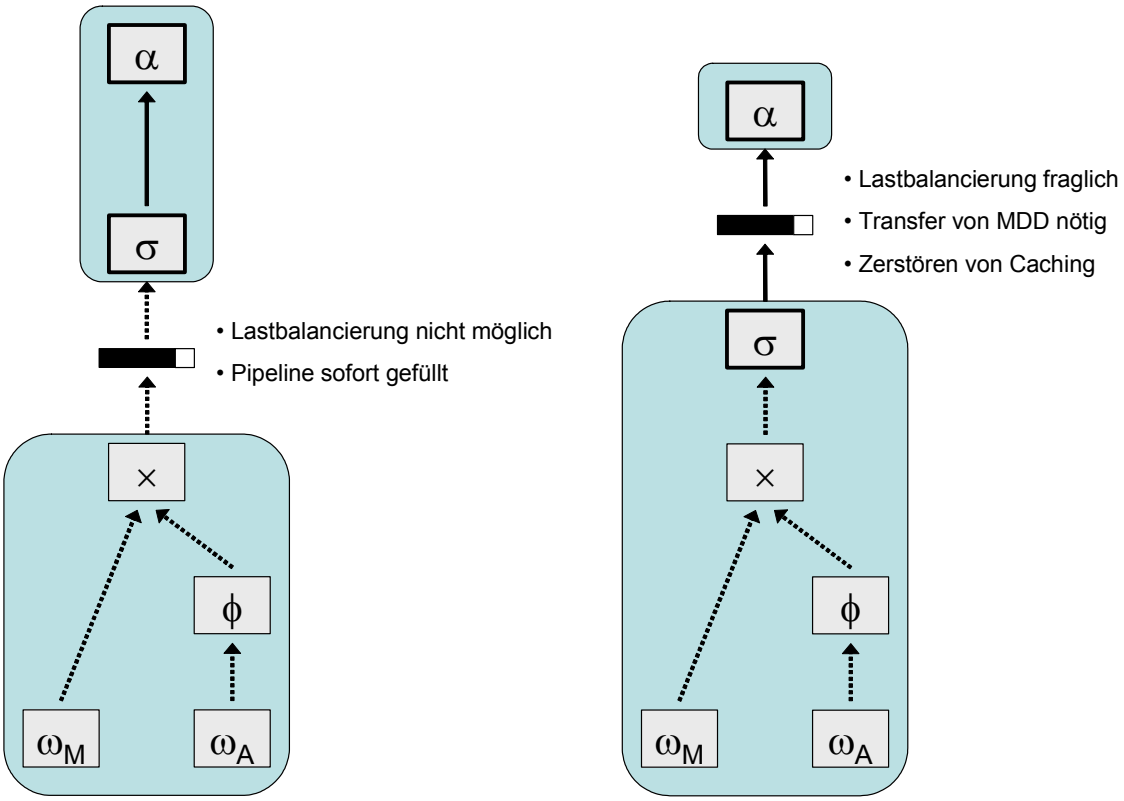


Abbildung 4.5: Probleme der Pipeline Parallelität bei Anfragen auf Array Daten[37]

Für die Ein- und Ausgabe zeigt sich zusammenfassend, dass frühestens bei einer Selektion σ die Daten von einem persistenten in einen transienten Zustand übergehen, also wirklich aus der Datenbank geladen werden (siehe auch Abbildung 4.4)[38]. Die Applikation α erhält als Eingabe ein Tupel von transienten MDD, wenn eine Selektion als Eingabeknoten fungiert. Die Ausgabe der Applikation ist abhängig von der Array Operationen ein Skalar oder ein transientes MDD. Die Einschränkung auf eine Menge von Einzel-MDD statt einer Menge von Tupel von MDD als Resultat von α ist in der Implementierung[39] begründet. Werden in der Applikation lediglich Array Operationen ausgeführt, die die Daten nicht in einen transienten Zustand überführen, etwa geometrische Operationen oder Operationen auf Metadaten, ergibt sich bei gleichzeitigem Fehlen der Selektion ein persistentes MDD als Resultat. Die Daten werden in diesem Falle erst für den Transfer zum Client geladen. Bei der Analyse der Komplexitäten muss beachtet werden, dass die Kardinalität von Kollektionen n in der Realität gering ist. In der Praxis sind Kollektionen mit einer Kardinalität größer als 100 sehr selten. Die Anzahl der Zellen c ist dagegen typischerweise sehr groß, Millionen von Zellen in einem MDD sind keineswegs selten. Die Komplexität o_a der Array Operationen auf diesen Zellen hängt natürlich von der Operation selbst ab, ist aber in der Regel gering.

Problem Nummer eins, die Kommunikationskosten durch Transfer von Arrays, kann mit zwei Methoden vermieden werden. Entweder wird die Pipeline in einen Bereich des Anfragebau-

[37] Aus diesem Grund wird Pipeline Parallelisierung bei rasdaman nicht eingesetzt, sondern auf andere Techniken gesetzt.
[38] Bei verschachtelten Anfragen gehen Daten mit der untersten Selektion oder Applikation im Anfragebaum in einen transienten Zustand. Verschachtelte Anfragen werden in Kapitel 4.3 diskutiert.
[39] Einerseits vereinfacht sich so die Client-Server Kommunikation und somit auch die Programmierschnittstelle der RasDaMan Client Library. Andererseits wird das Einfügen durch eine geschachtelte Anfrage so ermöglicht.

mes integriert, in welchem noch keine transienten Datenflüsse erfolgen, wie etwa in Abbildung 4.5 links skizziert zwischen Kreuzprodukt und Selektion. Eine andere Methode nutzt die Tatsache, dass zwischen Selektion und Applikation ebenso ein persistentes Objekt übergeben werden kann, da die Selektion immer ein unverändertes Array weiterreicht. Die Pipeline wird folglich zwischen Selektion und Applikation positioniert, allerdings werden persistente Objekte transferiert (Abbildung 4.5 rechts). Die Aufteilung von Selektion und Applikation auf unterschiedliche parallele Instanzen mit Transfer von persistenten Objekten zerstört jedoch den Caching Mechanismus und resultiert in einer spürbaren Reduktion der Performanz. Folglich scheidet Methode 2 aus.

Das Problem Nummer zwei, die Lastbalancierung, kann nur durch Aufteilung der Operationen α und σ auf verschiedene Prozesse erfolgen, da nur diese Operatoren eine vergleichsweise hohe Komplexität besitzen. Dies führt jedoch wie gesagt zu Performanzeinbussen in der Verarbeitung, da die Zwischenspeicherung von geladenen Daten zunichte gemacht wird. Die Positionierung der Pipeline zwischen Kreuzprodukt und Selektion kann in keinem Fall auch nur annähernd eine Lastbalancierung zwischen diesen Prozessen erreichen. Abbildung 4.5 skizziert diese Problematik am Anfragebaum. Trennt die Pipeline zwei Blöcke unterschiedlicher Komplexität, ist keine Lastbalancierung möglich. Dieser Fall ist in der Abbildung links zu sehen. Der untere Block füllt die Pipeline sofort, die teuren Operatoren α und σ (in der Abbildung durch eine dicke Umrandung gekennzeichnet) entnehmen dann die Elemente aus diesem Puffer. Auf der rechten Seite der Abbildung werden die teuren Operationen α und σ parallel ausgeführt.

Zusammenfassend ergeben sich bei Pipeline Parallelität für Array Daten folgende Hindernisse:

- Eine gute parallele Auslastung für typische (nicht verschachtelte) Anfragen kann nur erfolgen, wenn α und σ eine annähernd identische Komplexität aufzeigen. Das ist in der Praxis jedoch unwahrscheinlich.
- Durch Aufteilung von α und σ werden Caching Konzepte zerstört. Eine Verarbeitung von α und σ innerhalb eines Knotens ermöglichst die Nutzung der von Selektion σ geladenen Daten für die Applikation α. Dieser Mechanismus kann bei paralleler Verarbeitung nicht genutzt werden.
- Der Datentransfer zwischen den parallelen Instanzen kann auf zwei Arten erfolgen:
    a. Transfer des (teilweise) materialisierten Objekts (transientes MDD)
    Wie in Kapitel 4.1 gezeigt, wirkt sich der Transfer von Arrays negativ auf die Performanz aus und ist zu vermeiden.
    b. Transfer eine Verweises auf das Objekt (persistentes MDD)
    Die Selektion σ verändert das Array nicht, sondern gibt eine Teilmenge der Kollektion an α weiter. Dadurch ist es möglich, dass lediglich eine Referenz auf das Objekt, genauer ein persistentes MDD, transferiert wird. Hier entsteht der Nachteil, dass für α das MDD erneut geladen werden muss, was wiederum zu Performanzeinbussen führt.

Vor allem in Shared-Nothing Architekturen mit Netzwerkverbindungen zwischen den parallelen Instanzen ist der ineffiziente Transfer das Kriterium gegen Pipeline Parallelität. Jedoch zeigt sich auch bei Shared-Everything Architekturen mit einer Kommunikation, die auf gemeinsame Speicherbereichen basiert, dass der Transfer von Arrays die Performanz beeinträchtigt (zum Beispiel wegen Synchronisation oder Überlastung des Bussystems). Dies führt zu zwei Schlussfolgerungen:

1. Pipeline Parallelität in der eigentlichen Form ist für die Verarbeitung von Array Daten nicht sinnvoll, da eine Lastbalancierung nicht realisiert werden kann. Ferner werden Mechanismen der Zwischenspeicherung zerstört.
2. Wie im nächsten Kapitel dargestellt, können Methoden der Pipeline Parallelität genutzt werden, um Funktionalitäten im Anfragebaum zu kapseln. Eine Bildung von Blöcken dient dann nicht der Lastbalancierung, sondern unterstützt Datenparallelität. Eine Beschreibung dieser Konzepte folgt in Kapitel 4.3.

Heuristik 1	Pipeline Parallelisierung
	Pipeline Parallelisierung ist aufgrund der Struktur der Anfragebäume sowie der Größe der zu übertragenden Arrays und deren aufwändigem Transfer für Array Daten nicht sinnvoll.[40]

Pipeline Parallelisierung ist also vor allem aufgrund der einfachen Struktur der Anfragebäumen nicht sinnvoll. Es stellt sich die Frage, ob diese Anfragebäume durch Optimierung oder Erweiterungen der Anfragesprache eine komplexere Struktur annehmen können. In Kapitel 4.3.2 wird das im Rahmen der Datenparallelität diskutiert. Für Pipeline Parallelität soll diese Diskussion hier nicht weiter geführt werden. Durch komplexere Anfragebäume ergibt sich zwar eine größere Wahrscheinlichkeit der Lastbalancierung, alle weiteren Nachteile, insbesondere der ineffiziente Transfer von Arrays, bleiben dagegen bestehen.

4.2.2 Bushy-Tree Parallelisierung (Horizontale Inter-Operator Parallelisierung)

Eine für relationale Anfrageverarbeitung häufig genutzte Parallelisierungsstrategie ist die Inter-Operator Datenparallelität. Ausgehend von der Beobachtung, dass die binären Verbundoperatoren (engl.: join) selbst komplex sind, jedoch auch die Berechnung der Eingabe für diese Operatoren oft aufwändig ist, wurden die parallele Verarbeitung von Joins und deren Eingabeströme eingehend untersucht. Der erste Punkt, die Parallelisierung der Operatoren selbst, wird als Intra-Operator Parallelisierung klassifiziert, hierbei werden die Eingabeströme als bereits berechnet gesehen. Die horizontale Parallelisierung, die Gegenstand dieses Kapitels ist, berechnet beide Eingaben eines Verbundoperators parallel. Unterschiedliche Operatoren werden auf unterschiedlichen Daten parallel ausgeführt, daher auch die Bezeichnung Inter-Operator Datenparallelität.

Die Erreichung eines hohen Parallelitätsgrades bei Anfragen mit einer Reihe von Verbundoperatoren, wie zum Beispiel für ROLAP Anfragen typisch, fordert Anfragebäume, deren linker und rechter Unterbaum ähnlich komplexe Operationen beinhalten und somit parallel verarbeitbar sind und einen konstanten Datenstrom liefern. Im Gegensatz zum Optimierungsmodul, das meist (links- oder rechts-) tiefe Anfragebäume erzeugt, ist für die horizontale Parallelisierung ein Anfragebaum gewünscht, welcher nicht tief, sondern möglichst ausgeglichen ist. Dies wird in der Literatur als „bushy tree“, wörtlich übersetzt „buschiger Baum“, bezeichnet. Linker und rechter Unterbaum eines Verbundes können so parallel verarbeitet werden, zur Erhöhung des Parallelitätsgrades kann dies bei weiteren Joins in den Unterbäumen ebenso gehandhabt werden (Abbildung 4.6).

[40] Die aufgezeigten Probleme der Inter-Prozess Kommunikation, die vor allem durch die Größe der zu übertragenen Daten und der Performanz der Übertragung entsteht, können durch ein kleingranulares Iteratorkonzept, zum Beispiel auf Basis von Kacheln, sowie durch performanten Datentransfer auf Basis von gemeinsamen Speicher oder schnellen Netzwerken entschärft werden. Trotzdem ist auch unter diesen Voraussetzungen eine Lastverteilung schwierig, vor allem aufgrund der einfachen Struktur von typischen Anfragebäumen der Anfragen auf Array Daten.

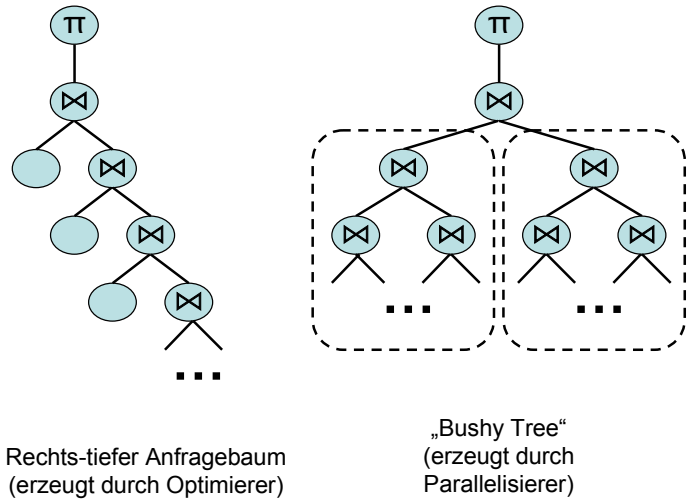


Abbildung 4.6: „Bushy Tree“ - Anfragebaum

Für Array Daten ist diese Form der parallelen Verarbeitung aus folgenden Gründen nicht geeignet:

- Verbundoperationen und verwandte Operationen werten ein Verbundprädikat aus, welches meist auf dem Vergleich von Elementen zweier Mengen beruht [HR01]. Der dabei eingesetzte Vergleichoperator $\ominus$ basiert zumeist auf einer Ordnungsrelation. Für Mengen von multidimensionalen Arrays ist keine Ordnungsrelation definiert. Die Identität selbst ist zwar definiert[41], ist jedoch in diesem Zusammenhang nicht sinnvoll einsetzbar. Auch Prädikate, die auf geografische Relationen beruhen, etwa Inklusion oder Schnitt (vor allem aus GIS bekannt), sind für multidimensionale Arrays nicht definiert. Aus diesem Grund ist im rasdaman DBMS kein Join Operator implementiert. Die einzig uns bekannte binäre Mengenoperation für Array Daten ist das Kreuzprodukt.
- Somit bleibt als Kandidat für horizontale Parallelisierung nur der Kreuzprodukt Operator $\times$ oder ein Vergleich mit aggregierten Werten in der Selektion σ. Wie jedoch in 4.3.2.3 gezeigt wird, sind komplexe Unterbäume durch das Fehlen von verschachtelten Anfragen für Array Daten nur in seltensten Fällen gegeben. Ferner ist eine ungleiche Arbeitslast und somit Skew Effekte bei den entsprechenden Unterbäumen wahrscheinlich. Eine Auslastung der parallelen Ressourcen durch die in Kapitel 4.3 beschriebene Datenparallelität kann eine bessere Lastbalancierung gewährleisten und ist somit effizienter.

Die horizontale Parallelisierung zeigt bereits in der relationalen Anfrageverarbeitung Nachteile gegenüber anderen Parallelisierungsstrategien. Wie in diesem Kapitel analysiert, fallen für Array Daten diese Nachteile noch stärker ins Gewicht. Wir werden diese Methode der Parallelisierung folglich nicht weiter verfolgen.

[41] Identität von MDD kann zweifach definiert sein: erstens die Identität des referenzierten Objektes. Diese ist gegeben, wenn der Objektidentifikator identisch ist. Zweitens existiert ein induzierter Identitätsoperator, der auf dem Basisdatentyp(en) definiert ist. Zwei MDD sind in diesem Fall identisch, wenn alle Zellen identische Werte aufweisen.

Heuristik 2	Bushy-Tree Parallelisierung Horizontale Parallelisierung ist aufgrund der auf Arraymengen definierten Operatoren und aufgrund der einfachen Struktur der Anfragebäume für Array Daten nicht sinnvoll.

Inter-Operator Parallelität zeigt also für die parallele Verarbeitung von Array Daten keine zufrieden stellende Performanz. Das liegt teils an der Struktur der Anfragebäume, die eine Lastbalancierung nur in Ausnahmefällen erlaubt, vor allem aber an der Ineffizienz der Datenübertragung und Zwischenspeicherung, die für dieses Verfahren nötig ist. Die im folgenden Kapitel beschriebene Intra-Operator Parallelisierung schneidet in allen Bereichen besser ab: Lastbalancierung kann sehr effektiv gewährleistet werden, der Transfer von großen Zwischenergebnissen meist vermieden werden. Aufgrund der inhärenten Probleme der Inter-Operator Parallelität wurde diese bei der Implementierung nicht berücksichtigt.

4.3 Intra-Operator Parallelisierung (Datenparallelität)

Im eigentlichen Sinn bedeutet Intra-Operator Parallelität die parallele Verarbeitung eines einzelnen Operators im Anfragebaum. D.h. eine Datenmenge wird aufgeteilt und die Teilmengen an verschiedene parallele Instanzen transferiert, die dann die Operation auf dieser Teilmenge ausführen. Meist werden im Anfragebaum zusammenhängende Operatoren zu Blöcken verbunden, die dann datenparallel verarbeitet werden. Diese Gruppierung von Operanden vermeidet primär Datentransfer, sekundär können Optimierungen wie etwa Caching auf einer Instanz erhalten bleiben.

In Kapitel 4.2 wurde bereits gezeigt, dass in typischen (einfachen) Anfragebäumen Applikation α und Selektion σ die teuren Operationen darstellen. Eine Aufteilung dieser Operationen auf verschiedene parallele Instanzen ist wegen des Caching Mechanismus nicht sinnvoll. Folglich ist ein Block, bestehend aus Applikation und Selektion, ein Kandidat für Intra-Objekt Parallelität.

Vereinfacht beschrieben kann ein typischer Anfragebaum für Array Daten folgendermaßen parallel verarbeitet werden: ein Prozess (Master Prozess) wird für zentrale nicht-parallelisierbare Aufgaben wie etwa Client-Server Kommunikation, Caching und Verteilung der Datenlast, selektiert. Ein weiterer Prozess (Tupel Server Prozess) verteilt die zu verarbeitenden Daten unter Berücksichtigung einer Minimierung der Ein-/Ausgabekosten der Gesamtanfrage. Beide Prozesse verwalten zentrale Aufgaben und werden nicht parallel ausgeführt. Im Anfragebaum (siehe Abbildung 4.7) verarbeitet der Master Prozess einen neu eingefügten Wurzelknoten und der Tupel Server Prozess die Blätter des Baumes, die dem logischen (nicht dem physischen) Datenzugriff entsprechen. Dadurch kann die Datenverteilung zentral optimiert werden.

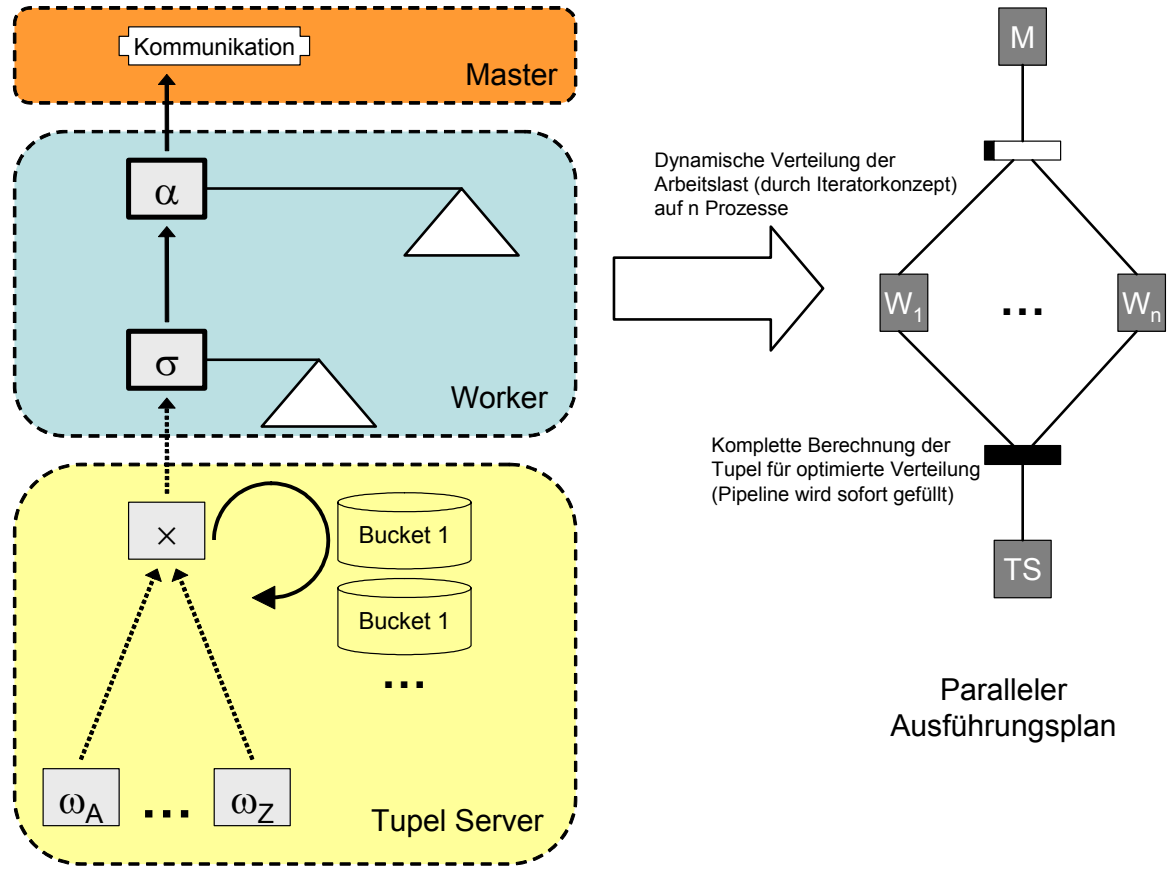


Abbildung 4.7: Anfragebaum und Ausführungsplan (Intra-Operator Parallelität)

Der Bereich des Anfragebaumes, welcher zwischen diesen Bereichen liegt, beinhaltet bei Selektion σ und Applikation α, also die teuren Operatoren einer Anfrage. Eine Aufteilung der Operatoren α und σ auf Prozesse wird nicht vorgenommen, da die zu erwartenden Performanzverluste durch Zerstörung von Caching Mechanismen zu hoch sind. Die Ausführung von σ und α wird parallelisiert und mit einem Parallelitätsgrad von p in das Iteratorkonzept integriert. Der Parallelitätsgrad p ist hierbei vom System, d.h. von der Hardware, abhängig. Auf einer Shared-Memory Architektur entspricht p der Anzahl der Prozessoren, in Shared-Nothing Umgebungen der Anzahl der parallel arbeitenden Rechner[42]. Abbildung 4.7 skizziert die Umsetzung des durch Blockbildung parallelisierten Anfragebaums in einen parallelen Ausführungsplan. Die parallele Last wird durch eine Anzahl von so genannten Worker Prozessen getragen.

Diese kurze Beschreibung der Intra-Operator Parallelität spiegelt das Konzept wider, das in den folgenden Kapiteln entwickelt und analysiert wird. Im Detail behandeln diese Kapitel folgende Fragestellungen:

- Analyse der Iteratoren (Kapitel 4.3.1)

[42] In einem heterogenen Cluster kann der Parallelitätsgrad größer als die Anzahl der Rechner sein. So wird vor allem die Anzahl der CPU eines Rechners im Cluster als Parallelitätsgrad dieses Knoten genutzt. In einem Cluster mit sehr unterschiedlicher Leistung der einzelnen Knoten kann eine Gewichtung der einzelnen Rechner ebenso über den Parallelitätsgrad erfolgen. Dies ist jedoch zumindest bei der in diesem Kapitel beschriebenen Inter-Objekt Parallelisierung überflüssig, da eine dynamische Anpassung an die Leistung der einzelnen Knoten erfolgt.

  - Welche Iteratoren können theoretisch parallel auf einer Teilmenge der Daten verarbeitet werden (Möglichkeit zur Parallelisierung)?
  - Welche Iteratoren haben eine hohe Komplexität und somit ein gutes Potential bezüglich der parallelen Ausführung (Eignung zur Parallelisierung)?
  - Welche Iteratoren werden folglich parallel verarbeitet, welche Folgerungen ergeben sich hieraus (Anwendung der Ergebnisse der Analyse)?
- Fusion der Iteratoren zu parallelen Blöcken (Kapitel 4.3.2)
  - Eine Zusammenfassung von parallel zu verarbeitenden Iteratoren in einen parallelen Block (Ausführung auf einer parallelen Instanz) kann Transferkosten sparen und somit die Performanz erhöhen. Die Analyse wird für typische (einfache) Anfragebäume in Kapitel 4.3.2.1 vorgenommen.
  - Optimierungstechniken beeinflussen den Anfragebaum und wirken sich folglich auch auf die Bildung paralleler Blöcke aus. In Kapitel 4.3.2.2 werden für Array Anfragen bekannte Optimierungstechniken auf ihren negativen Einfluss bezüglich paralleler Verarbeitung untersucht. Falls nötig werden Algorithmen entwickelt, die diese Optimierungen nachbilden, jedoch mit Eignung für parallele Ausführung.
  - Verschachtelte Anfragen (engl.: nested queries) sind für Array Anfragen nur in wenigen Fällen sinnvoll. Das wird die Analyse in Kapitel 4.3.2.3 ergeben. Trotzdem wird eine Auswirkung auf die Bildung von parallelen Blöcken in diesem Kapitel diskutiert.
  - Die Funktion der Datenverteilung wird in einem parallelen Block gekapselt und optimiert. Diese Optimierung der Datenverteilung wird in Kapitel 4.3.2.4 analysiert.
- Adaption des Operatorbaumes (Kapitel 4.3.3)
  - Jegliche parallele Funktionalität wird in neu definierten Iteratoren gekapselt. Dies sichert die Robustheit der Implementierung. Die Funktionalität dieser Iteratoren wird im Detail gezeigt.
  - Ein Algorithmus für die Integration dieser neu definierten Iteratoren in den Anfragebaum wird gegeben.
- Integration in die Anfrageausführung (Kapitel 4.3.4)
  Dieses Kapitel skizziert den Ablauf einer parallelisierten Anfrage durch den angepassten Operatorbaum.
- Analyse bezüglich Anfälligkeit für Skew (Kapitel 4.3.5)
  Das Kapitel zeigt durch eine formale Analyse, dass die mengenbasierte Parallelisierung, das heißt die Verteilung von multidimensionalen Objekten an Prozesse, für viele typische Szenarios nicht ausreichend ist. Diese Analyse bildet die Überleitung zu Kapitel 4.4, welches die parallele Verarbeitung von Einzelobjekten zum Thema hat.

4.3.1 Analyse der Iteratoren

Formal gesehen, können die in Kapitel 2.3 definierten Operatoren α, σ, $\times$, ϕ und ω, die Mengen von Arrays verarbeiten, durch eine Adaption der Algebra parallel verarbeitet werden[43]: wir definieren zwei neue Operatoren für die Partitionierung und die Vereinigung einer Menge von Arrays oder eines Kreuzprodukts von Arraymengen.

[43] Im Folgenden bedeutet die Notation C_1 „erste Kollektion eines Kreuzproduktes“, C^1 hingegen ist „eine Partition der Eingabe, die vom ersten Prozess verarbeitet wird“

Definition 4.1: Partitionierung θ_p einer Arraymenge

Sei C eine Kollektion bzw. R ein Kreuzprodukt von Arrays: $R = C_1 \times \ldots \times C_k$. Eine Partitionierung θ_p teilt die Eingabemenge in p disjunkte Teilmengen auf:

$$\theta_p(C) = \{C^i \mid C^i \subseteq C\};\ \bigcup_{i=1}^{p} C^i = C;\ C^k \cap C^l = \varnothing, C^{k,l} \in C^i$$

$$\theta_p(R) = \{R^i \mid R^i \subseteq R\};\ \bigcup_{i=1}^{p} R^i = R;\ R^k \cap R^l = \varnothing, R^{k,l} \in R^i$$

Wir beschränken uns im Folgenden auf eine Kollektion als Eingabe, die Transformationen können ebenso auf mehrwertige Relationen von Arrays übertragen werden.

Ein einfaches Beispiel für die Partitionierung einer Arraymenge C mit den Identifikatoren 1, 2, 3, 4 ist $\theta_2(\{1, 2, 3, 4\}) = \{1, 4\}, \{2, 3\}$. Der zur Partitionierung gegensätzliche Operator entspricht der Mengenvereinigung. Mit Hilfe dieser zwei Operatoren kann eine Parallelverarbeitung der Operatoren durch Partitionierung der Eingabeströme definiert werden:

1. Parallele Ausführung der Applikation α
   Eine Applikation auf einer Menge erfolgt laut Definition auf den Objekten unabhängig und kann somit parallel auf beliebigen Partitionen ausgeführt werden. Der maximale Parallelitätsgrad ist |C|. Die partiellen Ergebnisse werden anschließend zum Gesamtresultat vereinigt.
   $$\alpha(C) \equiv \bigcup \alpha(\theta(C)) \equiv \bigcup_{i=1}^{p} \alpha(C^i)$$

2. Parallele Ausführung der Selektion σ
   Eine Selektion auf einer Menge erfolgt laut Definition auf den Objekten unabhängig und kann somit parallel auf beliebigen Partitionen ausgeführt werden. Der maximale Parallelitätsgrad ist |C|. Die partiellen Ergebnisse werden anschließend zum Gesamtresultat vereinigt.
   $$\sigma(C) \equiv \bigcup \sigma(\theta(C)) \equiv \bigcup_{i=1}^{p} \sigma(C^i)$$

3. Parallele Ausführung des Kreuzproduktes $\times$
   Ein Kreuzprodukt kann parallel berechnet werden, indem eine Eingabekollektion partitioniert wird. Im Falle eines binären Kreuzprodukts wie folgt:
   $$\times(C_1, C_2) \equiv \bigcup \times(\theta(C_1), C_2) \equiv \bigcup_{i=1}^{p} \times(C_1^i, C_2) \equiv \bigcup \times(C_1, \theta(C_2)) \equiv \bigcup_{i=1}^{p} \times(C_1, C_2^i)$$
   Verallgemeinert auf ein n-näres Kreuzprodukt müssen n-1 Eingaben der n-wertigen Operation für alle Prozesse repliziert werden, lediglich eine Eingabe kann partitioniert werden. Eine parallele Verarbeitung des Kreuzproduktes ist somit neben der geringen Komplexität der Operation auch wegen der nötigen Replizierung der Eingabströme aber nicht sinnvoll.

4. Parallele Ausführung der Repartitionierung ϕ
   Eine Repartitionierung kann zwar parallelisiert werden, ist in der Praxis wegen mangelnder Komplexität der Operation und inhärenter Ungleichverteilung der Arbeitslast nicht sinnvoll.

   $$\phi(C) \equiv \bigcup \phi(\theta(C)) \equiv \bigcup_{i=1}^{p} \phi(C^i)$$

   Wird die Eingabe von ϕ in p parallele Partitionen geteilt, so trägt wegen der dieser Operation inhärenten Beschränkung auf ein einzelnes MDD ein Prozess die komplette Last.

Es sei angemerkt, dass in der Implementierung die Partitionierung ϕ nicht a priori berechnet wird, sondern dynamisch während der Anfrageverarbeitung entsteht. Durch die Bindung der Verteilung an die Verarbeitungsstrategie des Iteratormodells werden Lastungleichheiten vermieden.

Die parallele Verarbeitung einer Verknüpfung dieser Operatoren, also eines Blocks im Anfragebaum, lässt sich algebraisch durch folgende Umformungen berechnen. Sei X ein beliebiger Term der Algebra mit Kollektion als Resultat.

$$\alpha(X) \equiv \bigcup \theta(\alpha(X)) \equiv \bigcup \alpha(\theta(X))$$
$$\sigma(X) \equiv \bigcup \theta(\sigma(X)) \equiv \bigcup \sigma(\theta(X))$$
$$\times(X_1, X_2) \equiv \bigcup \theta(\times(X_1, X_2)) \equiv \bigcup \times(\theta(X_1), X_2) \equiv \bigcup \times(X_1, \theta(X_2))$$
$$\phi(X) \equiv \bigcup \theta(\phi(X)) \equiv \bigcup \phi(\theta(X))$$

Demnach kann etwa ein typischer Anfragebaum, bestehend aus Repartitionierung, Kreuzprodukt, Selektion und Applikation theoretisch folgendermaßen parallelisiert werden[44]:

$$\alpha(\sigma(\times(\phi(C_1), C_2))) \equiv$$
$$\bigcup \theta(\alpha(\sigma(\times(\phi(C_1), C_2)))) \equiv$$
$$\bigcup \alpha(\theta(\sigma(\times(\phi(C_1), C_2)))) \equiv$$
$$\bigcup \alpha(\sigma(\theta(\times(\phi(C_1), C_2)))) \equiv$$
$$\bigcup \alpha(\sigma(\times(\theta(\phi(C_1)), C_2))) \equiv$$
$$\bigcup \alpha(\sigma(\times(\phi(\theta(C_1)), C_2))) \equiv$$
$$\bigcup \alpha(\sigma(\times(\phi(C_1), \theta(C_2))))$$

Wie im nächsten Kapitel detailliert gezeigt, werden wegen der nötigen Replikation von Daten sowie der schlechten Lastverteilung die Regeln zur Partitionierung der Eingaben für Kreuzprodukt und Repartitionierung nicht angewandt. Die Umformung eines Terms, der eine typische Anfrage darstellt, zeigt sich folgendermaßen:

$$\alpha(\sigma(\times(\omega(C_1), ..., \phi(\omega(C_k))))) \equiv \bigcup \alpha(\sigma(\theta(\times(\omega(C_1), ..., \phi(\omega(C_k))))))$$

[44] Der Zugriffsoperator ω wurde aus Gründen der Übersichtlichkeit weggelassen.

D.h. ϕ wird nicht über $\times$ durchgezogen, da ab dieser Operation nur persistente Objekte referenziert werden. Die Partitionierung erfolgt stattdessen nach dem Kreuzprodukt, die Vereinigung der partiellen Ergebnisse wird nach der Applikation an der Wurzel des Anfragebaumes vorgenommen.

Iterator	Parallelisierung möglich	Parallelisierung sinnvoll
Applikation α	Ja	Ja
Selektion σ	Ja	Ja
Kreuzprodukt $\times$	Ja, Redundanz der Eingabe	Nein
Repartitionierung ϕ	Ja, mit Skew	Nein
Kollektionszugriff ω	Ja, jedoch E/A	Nein

Tabelle 4.3: Möglichkeit und Eignung der parallelen Verarbeitung von Array Iteratoren

Eine Zusammenfassung der Möglichkeit und Eignung aller definierten Operationen auf Array Mengen sind in Tabelle 4.3 enthalten. Eine parallele Ausführung des Kollektionszugriffs entspricht einer Realisierung von E/A Parallelität und ist nicht Thema dieser Arbeit.

Heuristik 3	Datenpartitionierung für Mengen von Arrays
	Eine parallele Applikation und Selektion der Objekte einer Menge von Arrays zeigt exzellente Ergebnisse, da diese Arrays unabhängig verarbeitet werden. Eine Kapselung der Operatoren α und σ in einem Prozess bewahrt Caching Effekte und vermeidet größtenteils den Transfer von großen Objekten. Eine parallele Verarbeitung weiterer Operatoren ist nicht sinnvoll.

Die vorgestellte Datenparallelität auf Basis der Verteilung von multidimensionalen Arrays zeigt hervorragende Resultate, insbesondere wegen folgender Faktoren:

- Die Berechnung auf den Daten erfolgt völlig unabhängig voneinander. Eine Kommunikation oder ein Transfer von Zwischenergebnissen ist während der Berechnung von σ und α nicht nötig.
- Die Verteilung der Daten erfolgt vor dem physischen Laden der Daten. Das Übertragen großer Datenelemente mit komplexen Strukturen wird so vermieden.
- Die Ergebnisse der Berechnungen sind meist Daten mit reduziertem Volumen. Eine Aggregation oder Skalierung ist typisch für serverseitige Analysen auf Array Daten (Begrenzung der Datenmenge vor Transfer zum Client). Der Transfer der Zwischenergebnisse zwischen parallelen Instanzen ist in diesen Fällen äußerst effizient.
- Ein Großteil von Anfragen referenziert eine einzelne Kollektion. Hierbei kann die Berechnung auf Datenpartitionen auf Prozesse verteilt werden, wobei redundante Ein-/Ausgabe völlig vermieden wird.
- Die Ausführung der Datenparallelität kann hervorragend in das Iteratorkonzept integriert werden. Die Größe der Objekte erlaubt eine dynamische Balancierung der Arbeitslast (engl.: load balancing) zur Vermeidung von Skew Effekten.

Diese theoretische Analyse und die Schlussfolgerungen bezüglich der Performanz werden durch die in Kapitel 6 präsentierten Messungen bestätigt.

4.3.2 Fusion von Iteratoren zu parallelen Blöcken

Dieses Kapitel behandelt vereinfacht die Frage: „welche Operatoren des Anfragebaumes sollen konkret welchen Prozessen zugeordnet werden?“. Kapitel 4.3.1 hat bereits gezeigt, dass insbesondere Applikation und Selektion Potential für parallele Verarbeitung aufweisen. Eine Kapselung der Operationen in einem Prozess bewahrt eine Optimierung durch Caching. In diesem Kapitel werden nun Anfragen und somit Anfragebäume in diesem Hinblick analysiert. Beginnend bei typischen unverschachtelten Anfragen auf Array Mengen werden im Folgenden Adaptionen des Operatorbaumes durch Optimierungen und komplexere verschachtelte Anfragen betrachtet. Die Bildung möglichst umfangreicher Blöcke als Basis einer Datenparallelität ist aus folgenden Gründen sinnvoll:

- Durch Zusammenfassung von Operatoren können Kommunikationskosten gespart werden. Folglich sind insbesondere Operatoren, zwischen welchen ein Datenfluss mit transienten Array herrscht, Kandidaten für eine solche Fusion.
- Die Kapselung von Funktionalität in einem Block erlaubt dessen Optimierung. Das ist vor allem für Datenverteilung relevant. Aber auch die Selektion eines koordinierenden Prozesses, welcher keine parallele Last trägt und auf spezielle Situationen sofort reagieren kann, fällt unter funktionelle Optimierung.
- Bewährte Konzepte und Optimierungen der Anfrageverarbeitung können gewahrt werden.

4.3.2.1 Parallele Blockbildung in einfachen Anfragebäumen

Das Array DBMS rasdaman kennt keine verschachtelten Anfragen. Dies liegt primär an der Tatsache, dass verschachtelte Anfragen für Mengen von Arrays weit weniger Anwendungsmöglichkeiten aufzeigen als für relationalen Mengen. Wir werden dennoch verschachtelte Anfragen, eine Klassifizierung ihres Einsatzes in relationalen Anfragen und den Sinn für Array Daten, sowie die Auswirkungen auf Parallelität in 4.3.2.3 diskutieren.

Ungeschachtelte Anfragen erzeugen einfach strukturierte Anfragebäume entsprechend Abbildung 4.8. Die kostenintensiven Operatoren α und σ sind durch dicke Umrandung gekennzeichnet, die gestrichelten Linien veranschaulichen die letztendlich realisierte Aufteilung der Teilbereiche des Operatorbaumes auf Prozessklassen, die unten näher erläutert wird.

Die einfachste Art von Anfragen erzeugt den in der Abbildung unter (1) gezeigten Anfragebaum[45]. Weitere zu analysierende Anfragebäume enthalten eine Selektion (2), eine Repartitionierung (3) oder ein Kreuzprodukt (4) für die Verknüpfung mehrerer Kollektionen.

Eine erste Voraussetzung für die parallele Verarbeitung des Operatorbaumes ist, dass ein ausgewiesener Prozess die nicht-parallelisierbare Client-Server Kommunikation abwickelt, d.h. die Ergebnisse sammelt und zum Client Programm überträgt. In Abbildung 4.8 ist das lediglich für die erste Anfrage links skizziert: ein Prozess sammelt die Teilresultate der Anfrage von den parallelen Instanzen (jeweils mit Operator α) und überträgt das komplette Resultat an die Applikation.

[45] Ein Kreuzprodukt-Operator wird im rasdaman DBMS immer erzeugt, auch wenn lediglich eine Eingabekollektion referenziert wird. Ebenso beinhaltet jeder Anfragebaum eine Applikation, bei fehlenden Array Operationen dient dies lediglich der Bindung an die referenzierte(n) Kollektion(en).

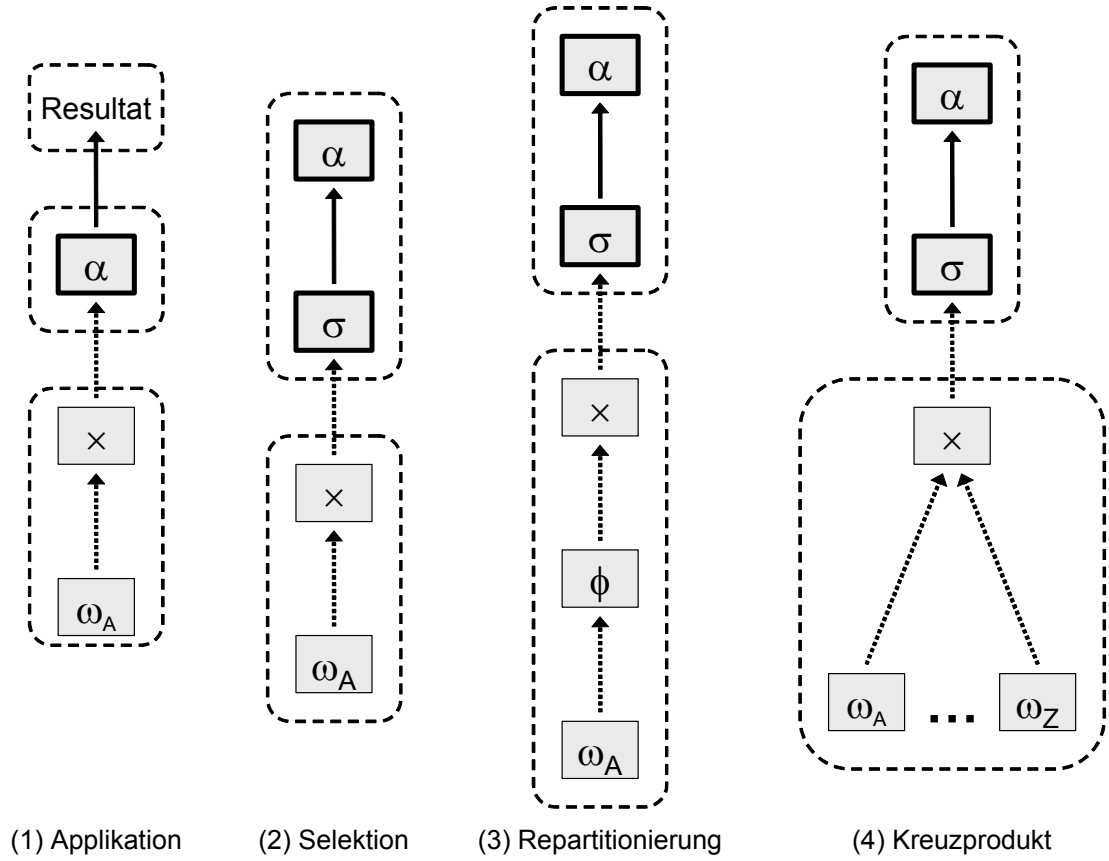


Abbildung 4.8: Klassifizierung von einfachen Anfragen an Array Mengen

Heuristik 4 Client-Server Kommunikation
Positioniere eine Pipeline über dem Wurzelknoten des Anfragebaumes. Somit wird ein Prozess selektiert, der die Client-Server Kommunikation übernimmt.

Eine weitere Zuordnung von Operationen auf Prozesse ergibt sich durch die Trennung von Datenverwaltung und Berechnungen auf diesen Daten im Anfragebaum.

Heuristik 5 Kapselung der Datenverteilung
Positioniere eine Pipeline über dem Kreuzprodukt im Anfragebaum und separiere so die Ausführung des Kreuzprodukts (mit darunter liegenden Operatoren) und der Operatoren α und evtl. σ darüber! Durch Kapselung der Datenverteilung kann ihre Optimierung gewährleistet werden.

Diese vertikale Aufteilung des Operatorbaumes und der Zuteilung zu Prozessen führt wie bereits erwähnt nicht zu Pipeline Parallelität, das heißt, es werden nicht Bereiche gebildet, deren parallele Ausführung die Gesamtarbeitslast aufteilt. Vielmehr hat die das Kreuzprodukt × enthaltende Partition eine deutlich geringere Komplexität und erzeugt somit weniger Last als die Partition mit den Operatoren α bzw. σ. Trotzdem ist es sinnvoll, einen Prozess zu se-

lektieren, der die Verteilung der Daten kapselt. Erstens kann dadurch gegenseitiger Ausschluss bei der Datenvergabe garantiert werden. Zweitens kann die Vergabe an Prozesse so zentral erfolgen und damit optimiert werden. Eine Diskussion hierüber erfolgt in 4.3.2.4.

Eine weitere Regel ergibt sich aus der Tatsache, dass ein Zerstören der Zwischenspeicherung von aus dem RDBMS geladenen Daten durch Parallelisierung des Anfragebaumes vermieden werden sollte.

Heuristik 6 Bewahren von Caching[46] Konzepten Verteile benachbarte Operatoren α und σ nicht auf verschiedene Prozesse! Dies zerstört in den meisten Fällen eine Zwischenspeicherung von bereits geladenen Daten und resultiert in erhöhter E/A, da die Daten erneut geladen oder transferiert werden müssen.

In der Selektion σ referenzierte Daten werden mit hoher Wahrscheinlichkeit auch in der Applikation α benötigt. Ein Caching Mechanismus für bereits geladene Daten ist daher sinnvoll und wird im **rasdaman** Array DBMS verwendet. Hierbei werden einmal in den Hauptspeicher geladene Kacheln des Arrays als solche bis zum Ende der Anfrage unverändert in einem gesonderten Hauptspeicherbereich gehalten. Werden σ und α auf verschiedene Prozesse verteilt, so wird dieser Mechanismus zerstört, was in einem deutlichen Einbruch des Speed-Up resultiert.

Durch Anwendung der Heuristiken 4 bis 6 werden drei Klassen von parallelen Instanzen definiert, für spezialisierte Funktionalität und zur Verarbeitung gesonderter Bereiche des Anfragebaumes:

- Ein **Master** Prozess bildet die Schnittstelle für die Client-Server Kommunikation. Er nimmt die Anfrage entgegen, verteilt die Arbeitslast auf interne Prozesse, verwaltet zentrale Mechanismen (vor allem Cache, siehe 4.3.2.2), sammelt Ergebnisse und übermittelt diese an den entsprechenden Client.
- Ein Pool von internen **Worker** Prozessen (die konkrete Anzahl hängt von der Hardware ab) nimmt vom Master Prozess Aufträge entgegen und arbeitet diese ab. Worker Prozesse verarbeiten die teuren Operatoren α und σ des Anfragebaumes, sie tragen also die Hauptlast.
- Ein **Tupel Server** Prozess[47] verwaltet die Vergabe der Arbeitslast (Operator $\times$ des Anfragebaumes) an die Worker Prozesse. Die Optimierung der Datenvergabe kann in diesem Prozess realisiert werden.

Die so geschaffenen Blöcke werden nicht parallel unter Nutzung einer Pipeline ausgeführt. Vielmehr realisiert diese Partitionierung des Anfragebaumes durch eine Menge von Worker Prozessen eine effiziente Datenparallelität.

[46] Cache, engl.: Speicher, die Strategie des Caching zwischenspeichert besonders häufig benötigte Daten auf einem schnelleren Speicherbereich der Speicherhierarchie. Dies kann von Prozessorcache, der einen schnelleren Zugriff gegenüber Hauptspeicher bietet, bis zu Festplattencache gehen, der Daten von Tertiärspeicher auf Festplatte zwischenspeichert. In unserem Fall werden einmal gelesene Daten von Sekundärspeicher (Festplatte) im Hauptspeicher vorgehalten.

[47] Bei verschachtelten Anfragen (Kapitel 4.3.2.3) existieren evtl. mehrere Tupel Server Prozesse.

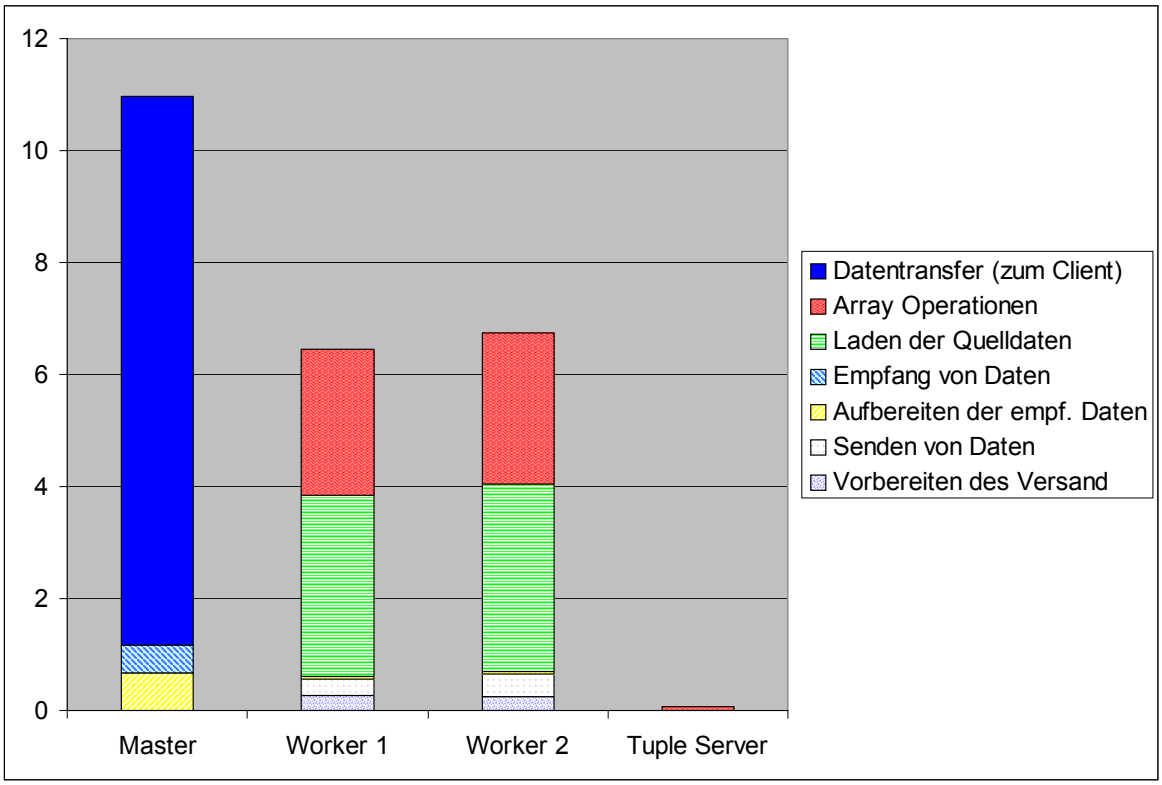


Abbildung 4.9: Verteilung der Arbeitslast auf Prozesse

Um einen ersten Eindruck von der Arbeitslast dieser Prozesse und der beteiligten Faktoren zu bekommen, ist in Abbildung 4.9 der Zeitbedarf der einzelnen Prozesse und Faktoren anhand einer Beispielanfrage skizziert. Die zugrunde liegende Anfrage rechnet ein MDD mit Temperaturwerten in Kelvin in ° Celsius um:

```
SELECT a - 273.15 FROM mpim8 AS a
```

Diese Anfrage fällt in die Klasse von schlecht parallelisierbaren Anfragen, kann jedoch trotzdem durch Parallelität enorm beschleunigt werden. Im Detail ist die Anfrage aus mehreren Gründen problematisch:

- Große Zwischenergebnisse müssen sowohl zwischen parallelen Instanzen als auch zum Client übertragen werden. Das ist für Array Anfragen untypisch.
- Es ist lediglich eine Operation, nämlich die induzierte Subtraktion, beteiligt. Auch das ist für Array Anfragen selten. Der Anteil von E/A Aufwand zu CPU Aufwand ist folglich ausgeglichen. Umso komplexer die Operationen einer Anfrage (d.h. auch umso mehr arithmetische oder aggregierende Operationen), umso besser ist die parallele Performanz, da der Anteil von CPU (parallelisierbar) den Anteil an E/A (nicht parallelisierbar[48]) an der Berechnung überwiegt.
- Das Fehlen einer Selektion und somit weiteren Array Operationen macht sich aus oben genannten Gründen ebenso negativ auf die Performanz bemerkbar.

[48] Wir gehen im Normalfall davon aus, dass Daten nicht parallel geladen werden. I/O Parallelität kann jedoch durch Datenverteilung oder Datenredundanz gewährleistet werden.

Die drei vorgestellten Prozessklassen, Tupel Server, Worker (zwei Prozesse) und Master können in Abbildung 4.9 identifiziert werden.

- Der Tupel Server identifiziert die referenzierten Kollektionen in der relationalen Datenbank, die zur Datenhaltung dient, und verteilt die Tupel von Identifikatoren an die Prozesse. Datenbankzugriff und Verteilungsalgorithmus benötigen kaum Zeit, ebenso das Verpacken der Tupel in einer Nachricht (engl.: message) und das Versenden (die Daten sind noch persistent). Folglich ist der Zeitbedarf minimal, die einzelnen Faktoren „Laden der Quelldaten", „Vorbereiten des Versands durch Linearisieren der Struktur" und der Versand selbst sind in der Abbildung nicht unterscheidbar.
- Die Worker Prozesse empfangen persistente MDD als Zwischenergebnisse vom Tupel Server Prozess. Das Empfangen und die Aufbereitung durch Restrukturierung in Objekte der Programmiersprache sind in der Abbildung kaum zu identifizieren, da die Daten persistent und daher klein sind. Erst bei Ausführung von Array Operationen auf Daten werden diese in einen transienten Zustand überführt, d.h. aus dem RDBMS geladen. Da für diese Anfrage transiente Daten als Resultat an den Client übertragen werden müssen und folglich an den Master Prozess übertragen werden müssen, ist das Linearisieren und der Transfer durchaus prägnant.
- Aus diesem Grund nimmt auch das Empfangen und Restrukturieren beim Master Prozess einen beachtlichen Anteil an der Arbeitslast ein. Den größten Anteil an der Bearbeitungszeit benötigt jedoch der Transfer des Anfrageergebnisses zum Client[49].

Wie erwähnt, ist diese Anfrage extrem einfach, gewinnt jedoch trotzdem durch parallele Verarbeitung an Performanz. Hier zeigt sich ein Unterschied gegenüber der Verarbeitung von relationalen Daten. Die Überraschung über den großen Anteil der Berechnungszeit gegenüber der Zeit für E/A für Anfragen an Array Daten kann man etwa [Rit99] und [Wid00] entnehmen. Die in diesen Arbeiten gewonnenen Erkenntnisse über die Problematik der Analyse von Array Daten führten zu vorliegender Dissertation.

Bei typischen Anfragen ist das Verhältnis von CPU (Array Operationen) und E/A (Laden der Quelldaten) deutlich zugunsten des CPU-Anteils verschoben und somit wird noch deutlich bessere parallele Performanz erreicht. Ebenso ist ein Transfer von kompletten Datensätzen zum Client, wie hier realisiert, eher selten. Vor dem Senden werden Daten meist aggregiert oder zumindest durch Skalierung, Kompression oder Konvertierung in ein komprimierendes Datenformat in ihrer Größe reduziert.

4.3.2.2 Optimierung von Anfragebäumen und Parallelität

Die in Abbildung 4.8 skizzierten einfach strukturierten Anfragebäume können durch zwei Einflüsse in ihrer Struktur komplexer werden:

1. Durch eine **Optimierung** der Verarbeitung von Array Mengen und damit entstehender Verschiebung von Operatoren. Hierunter fallen insbesondere das Zusammenführen gemeinsamer Teilausdrücke (CSE, engl.: common subexpressions) und das Verschieben von Operatoren nach unten (engl.: pushdown). Diese Einflüsse werden in vorliegendem Kapitel analysiert.

[49] Die Client-Server Kommunikation ist in rasdaman 5.0 mittels RPC (engl.: remote procedure call) oder HTTP möglich. Das für diese Grafik genutzte RPC zeigt sich in der Praxis als extrem unperformant und wurde in rasdaman 5.1 überarbeitet.

2. Durch die Möglichkeit **verschachtelter Anfragen**. Dies ist in rasdaman nicht vorgesehen. Wir werden jedoch die Möglichkeiten von solchen Anfragen für Array Daten und die entstehenden Probleme bezüglich Parallelisierung in Kapitel 4.3.2.3 diskutieren.

Optimierung gemeinsamer Teilausdrücke

Abbildung 4.10 zeigt eine Optimierung des Anfragebaumes, welche eine zentrale Berechnung gemeinsamer Teilausdrücke realisiert. Hierbei werden Berechnungen, die mehrfach im Baum auszuführen sind, in einer neuen Applikation zusammengefasst und im Anfragebaum über das Kreuzprodukt geschoben. Alle Bereiche des Baumes, die diesen gemeinsamen Teilausdruck verwendeten, referenzieren jetzt das Resultat der neuen Applikation α' (in der Abbildung als *$0* markiert).

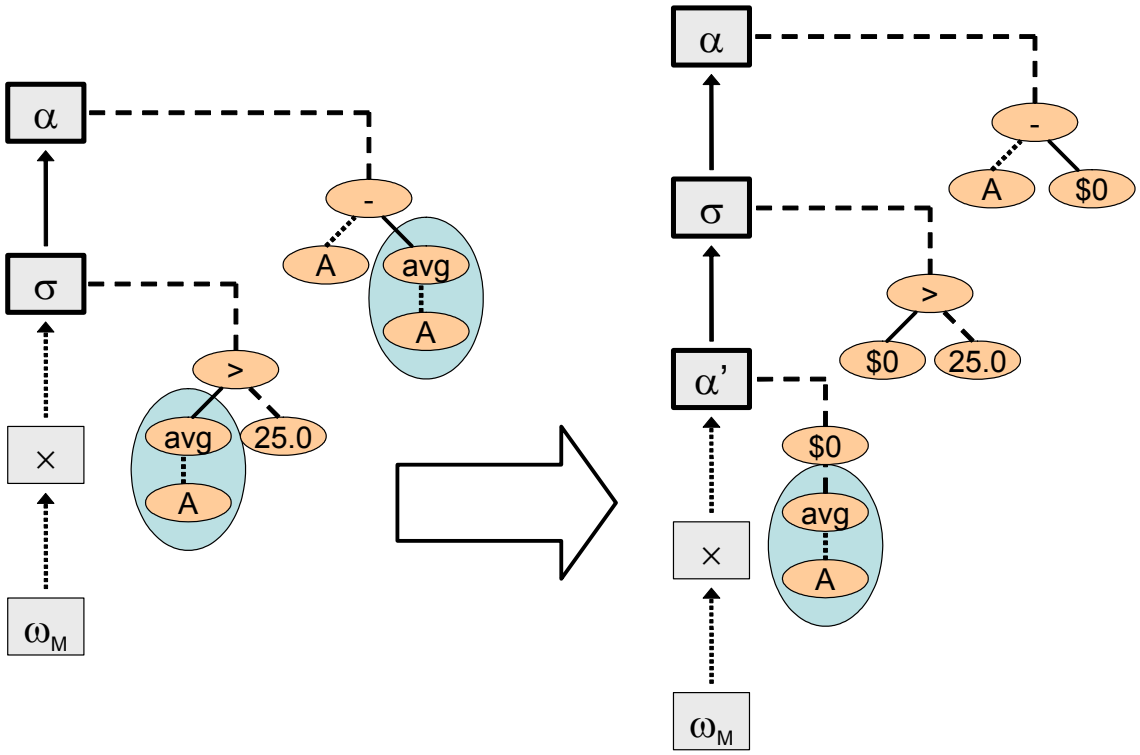


Abbildung 4.10: Optimierung von gemeinsamen Teilausdrücken

Für die in 4.3.2.1 skizzierte Parallelisierung hat die Optimierung von gemeinsamen Teilausdrücken mit diesem Algorithmus keinerlei Auswirkung. Durch Anwendung der Heuristiken erfolgt zwischen den Operatoren × und α' eine Trennung der Operatorblöcke und damit eine Aufteilung auf verschiedene Prozesse, ferner wird ein Kommunikationsprozess über dem Wurzelknoten α eingefügt. Transiente und damit zwischengespeicherte Daten bleiben in einem Prozess gekapselt.

Es muss jedoch erwähnt werden, dass der oben skizzierte und in rasdaman implementierte Algorithmus zur zentralen Berechnung von CSE von Voraussetzungen ausgeht, die nicht immer gegeben sind. Die Optimierung führt somit für bestimmte Fälle zu Einbussen in der Performanz:

- Ein Positionieren von α' unter die Selektion σ ist nur dann sinnvoll, wenn die CSE auch in der Selektion σ genutzt wird (anstatt mehrfach in α)[50]. Andernfalls wird die CSE für Objekte ausgewertet, obwohl die Selektion diese ausfiltert, was zu einer Reduzierung der Performanz führt. Die korrekte Position von α' wäre in diesem Fall zwischen α und σ.
- Werden mehrere Kollektionen referenziert, kann durch ein (anschließendes) Verschieben von α' unter das Kreuzprodukt $\times$ die Auswertung optimiert werden. Dies wird unten als eigene Optimierungstechnik beschrieben.

Im ersten Fall kann die Verarbeitung mühelos parallelisiert werden, das Verschieben von Operatoren unter das Kreuzprodukt $\times$ hat jedoch Auswirkungen für die Parallelisierung.

Heuristik 7 Gemeinsame Teilausdrücke und Parallelisierung
Die Optimierung gemeinsamer Teilausdrücke hat keinen Einfluss auf Parallelisierung. Ein anschließendes Verschieben des gemeinsamen Teilausdruckes nach unten im Operatorbaum (engl.: push-down) ist bei Verknüpfung mehrerer Kollektionen sinnvoll, resultiert jedoch in verminderter paralleler Performanz.

Optimierung durch Verschieben von Operatoren

Im Array DBMS rasdaman war das Verschieben von Applikation bzw. Selektion nach unten zwar angedacht [Rit99], es wurde in der Implementierung jedoch nicht umgesetzt. Im Rahmen dieser Arbeit wurde es aus zwei Gründen durch ein unten beschriebenes verbessertes Verfahren ersetzt:

- Im Zusammenhang mit der Implementierung des Kreuzprodukts als verschachtelte Schleife (engl.: nested loop) ergeben sich Probleme.
- Eine Parallelisierung der entstehenden Bäume erweist sich als problematisch. Insbesondere die Inter-Prozess Kommunikation wird dadurch bezüglich des Datenumfangs erhöht.

Im Folgenden analysieren wir vor allem die „selection pushdown" Optimierung, die erarbeiteten Konzepte können jedoch auf die Verschiebung anderer Operatoren übertragen werden. Das Verschieben der Applikation wird am Ende des Abschnitts kurz diskutiert.

Formal gesehen, wird eine Aufspaltung und Verschiebung der Selektion unter das Kreuzprodukt ausgedrückt durch

$$\sigma_{P(C_1)\wedge\ldots\wedge P(C_n)}(C_1\times\ldots\times C_n)=\sigma_{P(C_1)}(C_1)\times\ldots\times\sigma_{P(C_n)}(C_n)\,.$$

D.h. mehrere Teilprädikate, verknüpft durch ein logisches AND, können über den jeweiligen Zugriffsoperator geschoben werden, der durch das Teilprädikat $P(C_i)$ referenziert wird.

Abbildung 4.11 zeigt ein Beispiel für diese Optimierung durch Verschiebung des Selektionsoperators. Die Anfrage

[50] Beispiel: SELECT avg(a[*:*, *:*, 120]) > 10 AND avg(a[*:*, *:*, 120]) < 20 FROM …

```
SELECT max_cells(A*M)
FROM A, M
WHERE avg_cells(A) > 25.0
```

selektiert MDD aus einer Kollektion A, die einen durchschnittlichen Zellwert größer 25.0 (etwa die Temperatur in ° Celsius) besitzen und verknüpft diese in einer induktiven Multiplikation mit MDD der Kollektion M (die etwa als Bitmaske bestimmte Bereiche so auf einen Nullwert setzt)[51]. Aus den daraus resultierenden MDD wird jeweils das Maximum ermittelt.

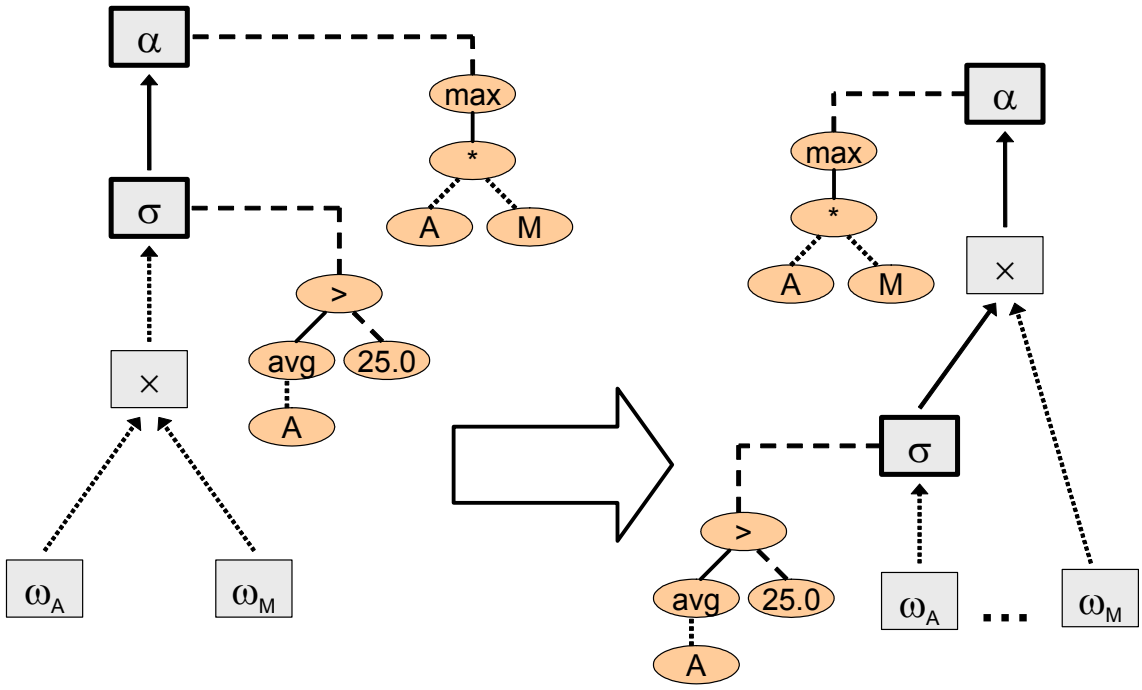


Abbildung 4.11: Optimierung durch Verschieben der Selektion

Das Prädikat der Selektion beinhaltet in diesem Fall lediglich den Verweis auf eine einzige Kollektion, kann also direkt über den entsprechenden Zugriffsoperator ω_A gesetzt werden. Der Vorteil dieser Optimierung besteht darin, dass die Selektion nur |A| mal statt |A*M| mal ausgeführt werden muss. D.h. jedes Objekt der Kollektion A wird nur einmal bezüglich des Prädikats geprüft statt der Prüfung jedes Elements des Kreuzprodukts.

Das erste Problem besteht in der Implementierung des Kreuzprodukts. Die Implementierung arbeitet die Eingabeoperatoren in einer verschachtelten Schleife ab[52]. Hierbei werden für jedes Element der äußeren Schleife alle Elemente der inneren Schleife iteriert. Ist die Selektion in der inneren Schleife referenziert, wird sie wiederum |A*M| mal ausgeführt. In diesem konkreten Beispiel kann das vermieden werden, indem die Selektion in der inneren Schleife ausge-

[51] Die induzierte Multiplikation wird meist als Vorbereitung einer Aggregationsoperation eingesetzt, um Bereiche des MDD „herauszufiltern“. So kann etwa mittels einer Maske M, die außerhalb des relevanten Bereichs den Wert 0 und innerhalb des Bereichs den Wert 1 hat, nur dieser Bereich zur Identifikation des Maximum herangezogen werden

[52] Für relationale Anfragen besteht diese Problematik nicht, da eine Überführung von logischen Operatoren in physikalische Operatoren stattfindet. Damit wird eine Selektion in jedem Fall mit einem vorherigen Zugriff zu einem *index scan* bzw. mit einem nachfolgenden Join zu einem *hash join* o.ä. optimiert.

führt wird (Optimierung der Reihenfolge des Kreuzprodukts). Im Allgemeinen kann × aber nicht dementsprechend optimiert werden, wenn mehr als eine Selektion nach unten verschoben wurden.

Das zweite Problem betrifft die parallele Ausführung. Durch ein Verschieben von Operationen unter das Kreuzprodukt × werden Berechnungen auf denselben Daten auf verschiedene Prozesse verteilt. D.h. es müssen transiente Objekte zwischen Prozessen transferiert werden, was in Extremfällen den Optimierungsvorteil vollständig zunichte macht.

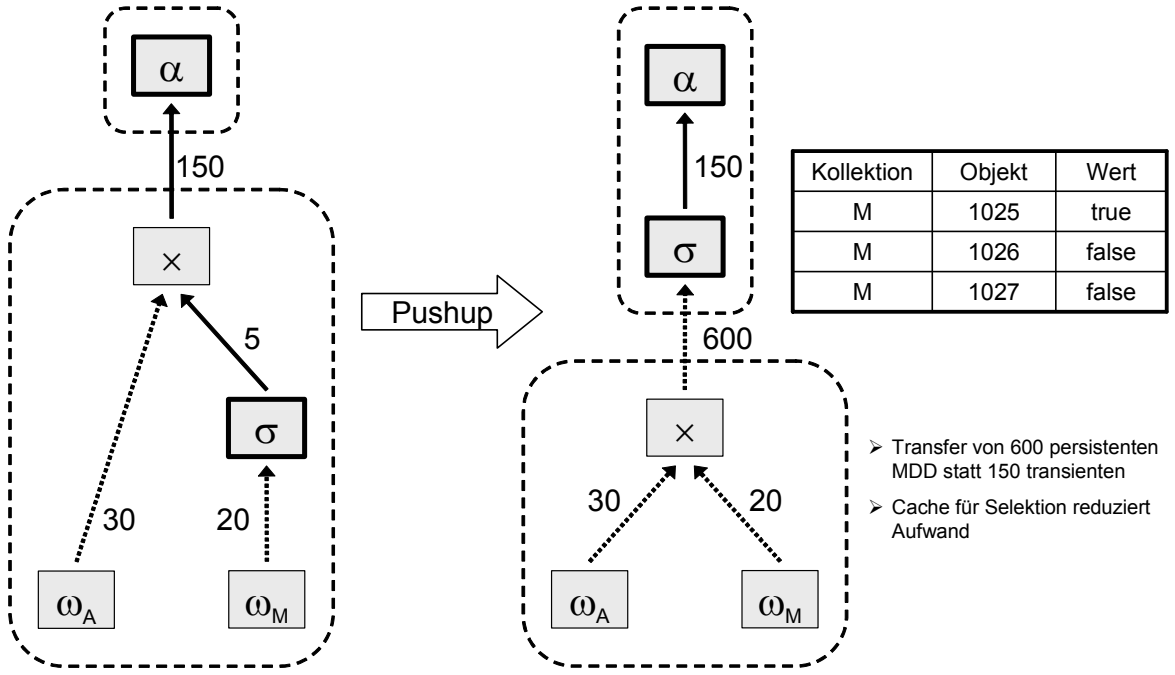


Abbildung 4.12: Cache für Selektionsergebnisse bei paralleler Verarbeitung

Eine Lösung, die beide Probleme vermeidet, besteht in der Integration eines zusätzlichen Cache, der die Ergebnisse der Prädikatauswertung zwischenspeichert. Auf ein Verschieben der Selektion kann dabei verzichtet werden, vielmehr werden die (eindeutigen) Identifikatoren der im Prädikat referenzierten MDD sowie das Ergebnis der Prädikatauswertung gespeichert. Abbildung 4.12 zeigt diese Optimierung an einem Beispiel.

Auf der linken Seite wird die klassische Pushdown Optimierung dargestellt. Kollektion A besteht in diesem Beispiel aus 30 Objekten, Kollektion M aus 20, die Selektivität beträgt 25%. Das Kreuzprodukt besteht also aus 150 Objekten, die bei paralleler Verarbeitung von einem anderen Prozess verarbeitet werden. In diesem Fall können die Daten entweder persistent versendet werden[53], was in einer Zerstörung von Caching resultiert, oder in einem transienten Zustand, was zu hohen Transferkosten führt.

Auf der rechten Seite ist die Optimierung durch den neuen Prädikatcache gezeigt. Es werden zwar 600 Objekte an einen anderen Prozess versandt, diese sind jedoch persistent, d.h. die Transferkosten sind um ein Vielfaches geringer. Die Selektion wird hier in vollem Umfang nur 20-mal ausgeführt, alle weiteren Prädikatauswertungen können durch Prüfen der Cache-

[53] Dies ist speziell für die Selektion der Fall, da die Daten nie in modifizierter Form weitergereicht werden.

einträge abgefangen werden. Die für die Prädikatauswertung geladenen Daten können durch den jeweiligen Prozess weiter für die Applikation verwendet werden. Es ist sinnvoll, den Prädikatcache zentral durch einen Prozess (Master Prozess) zu verwalten, da die Ergebnisse so von verschiedenen Prozessen genutzt werden können.

Am Rande sei angemerkt, dass die Auswertung eines Prädikats in **rasdaman** weiteres Optimierungspotential aufzeigt. Das Konzept des frühen Auswertungsabbruchs (engl.: early termination) kann hier angewendet werden und lässt sich mit dem Konzept eines Prädikatcache hervorragend vereinbaren. So kann beispielsweise bei einem Prädikat, das eine logische *and* Verknüpfung beinhaltet nach einer negativen Evaluierung eines der beiden Unterbäume die Auswertung abgebrochen werden, das Resultat ist *false*. Ebenso kann ein logisches *or* als *true* bewertet werden, wenn eines der Unterbäume bereits als *true* erkannt wurde. Diese Spaltung der Prädikatbäume, verbunden mit einer performanten Auswertung mit frühem Abbruch, ist im Array DBMS **rasdaman** nicht implementiert.

Die Verwaltung eines Cache für die Applikationsoperation zur Vermeidung eines Application Pushdown (insbesondere in Zusammenhang mit gemeinsamen Teilausdrücken verwendet) erweist sich als schwieriger. Das Resultat der Auswertung kann hier ein skalarer Wert (bei einem aggregierenden Operator) oder ein MDD sein. Letzteres erfordert hohen Speicherbedarf für die Zwischenspeicherung und kann bei paralleler Verarbeitung die Performanz durch den nötigen Transfer des Ergebnis MDD zwischen Prozessen merklich beeinträchtigen.

Heuristik 8	Pushdown von Operationen
	Pushdown der Selektion kann für die parallele Verarbeitung durch einen spezialisierten Cache ersetzt werden. Die Prozesse können damit bezüglich Transferkosten und Caching optimiert werden. Pushdown der Applikation führt zu erhöhten Transferkosten, falls das Resultat der Applikation ein MDD ist.

Zusammengefasst können also für Array Daten typische Optimierungstechniken in den meisten Fällen problemlos in eine parallele Verarbeitung integriert werden:

- Gemeinsame Teilausdrücke können zusammengefasst werden. Die eingesetzten Algorithmen zur parallelen Anfrageverarbeitung werden dadurch nicht beeinflusst. Die parallele Performanz bleibt voll erhalten.
- Selektionen werden nicht nach unten verschoben. Stattdessen ist ein zentraler Cache für Selektionsergebnisse nötig, um redundante Berechnungen auf verschiedenen parallelen Instanzen zu vermeiden. Damit wird neben einer allgemeinen Performanzsteigerung auch die parallele Performanz gesichert.
- Applikationen werden nicht nach unten verschoben, falls das Resultat ein skalarer Wert ist. In diesem Fall wird ein zentraler Cache für die Resultate eingesetzt. Ist das Resultat der Applikation ein MDD, ist die Verschiebung nach unten in den meisten Fällen die bessere Alternative. Allerdings muss gesichert werden, dass die Applikation in der äußeren Schleife des Kreuzprodukts liegt. Ferner wird in diesem Fall die parallele Performanz durch den nötigen Transfer von Arrays leiden.

4.3.2.3 Verschachtelte Anfragen für Array Daten

Die relationale Anfragesprache SQL erlaubt das Verschachteln von Anfragen. Das Prinzip der Orthogonalität erlaubt eine Unteranfrage mit einem skalaren Ergebnis oder einer Relation als

Ergebnis an Positionen der Anfrage, an welchen eben ein solch skalarer oder relationaler Wert erwartet wird [Cha98]. Der Standard SQL92 erlaubt lediglich eine Unteranfrage in der Selektion[54], weshalb sich frühe Untersuchungen zur Klassifizierung und Optimierung von verschachtelten Anfragen auf diese Möglichkeit beschränken [Kim82] [GW87].

Im Folgenden werden verschachtelte Anfragen auf Array Daten am Beispiel von RasQL und deren parallele Ausführung untersucht.

1. Unteranfrage im SELECT (in der Applikation α)
   In **rasdaman** sind aus implementierungstechnischen Gründen Tupel als Resultat einer Applikation (als Wurzelknoten) nicht erlaubt. D.h. Ergebnis ist immer eine Menge von Skalaren oder einer Menge von MDD. Ergebnismengen wie $\alpha \times s_1 \times s_2$ mit MDD α und Skalaren s_1, s_2, etc. sind zwar sinnvoll, etwa für die Analyse von skalierten MDD zusammen mit Aggregatswerten, sind in **rasdaman** insbesondere wegen einer Erhöhung der Komplexität in der Client-Server Schnittstelle jedoch nicht implementiert. Allein aus diesem Grund sind Unteranfragen im SELECT Teil einer RasQL Anfrage nicht nötig. Darüber hinaus ist diese Art der Unteranfrage in SQL nur dann sinnvoll, wenn in der Unterabfrage ein impliziter Join mit der äußeren Anfrage stattfindet (korrelierte Subquery). Ein Join ist für Array Mengen nicht vorgesehen, deshalb ist eine Unteranfrage im SELECT für Array Mengen prinzipiell zwar machbar, erhöht aber nicht die Mächtigkeit der Anfragesprache.

2. Unteranfrage im FROM (unter dem Kreuzprodukt $\times$)
   In SQL kann im FROM Teil einer Anfrage eine Unteranfrage gesetzt werden, die als Ergebnis eine Relation liefert. Laut [Cha98] kann dies in bestimmten Fällen sogar die Mächtigkeit von SQL erhöhen, d.h. es können Anfragen gestellt werden, die ohne ein solches Konstrukt nur durch mehrere Anfragen mit Speicherung der Zwischenergebnisse gelöst werden können. Eine detaillierte Analyse zu dieser Aussage ist in [Cha98] nicht zu finden, es findet sich lediglich ein Verweis, dass diese Verbesserung in der Möglichkeit einer Gruppierung in der Unteranfrage begründet sei. Eine Gruppierung ist für Array Mengen wegen einer fehlenden Ordnungsrelation auf diesen Mengen nicht vorgesehen, folglich bringen Unteranfragen in diesem Fall keine Verbesserung der Sprachmächtigkeit. Jedoch kann die Lesbarkeit von RasQL Anfragen durch Unteranfragen im FROM Teil deutlich erhöht werden:

```
SELECT sqrt(sqr(u[0,0:119,*:*,*:*])+sqr(v[0,0:119,*:*,*:*])) * m[0,0:119,*:*,*:*]
FROM mpi_u AS u, mpi_v AS v, mask2 AS m
WHERE avg_cells(sqrt(sqr(u[0,0:119,*:*,*:*]) + sqr(v[0,0:119,*:*,*:*]))) > 10.0
```

entspricht beispielsweise einer verschachtelte Anfrage

```
SELECT sqrt( sqr(u1) + sqr(v1) ) * m1
FROM
  (SELECT mpi_u[0,0:119,*:*,*:*] FROM mpi_u) as u1,
  (SELECT mpi_v[0,0:119,*:*,*:*] FROM mpi_v) as v1,
  (SELECT mask2[0,0:119,*:*,*:*] FROM mask2) as m1
WHERE avg_cells( sqrt( sqr(u1) + sqr(v1) ) ) > 10.0
```

[54] In aktuellen RDBMS und neueren SQL Standards gilt diese Beschränkung nicht mehr.

Die Umsetzung in einen entsprechenden Ausführungsplan ohne Verschachtelung etwa durch Substitution der Variablen kann durch den Optimierer erfolgen.

3. Unteranfrage im WHERE (in der Selektion σ)
   [Kim82] und [GW87] klassifizieren 4 Arten von Unteranfragen im WHERE Teil einer SQL Anfrage, nämlich mit aggregiertem Ergebnis oder einer Relation als Ergebnis, jeweils mit der äußeren Anfrage korreliert oder nicht. Eine korrelierte Unteranfrage ist für Array Daten aufgrund der fehlenden Ordnungsrelation und dem damit nicht definierten Vergleichsoperator nicht sinnvoll. Eine nicht korrelierte Unteranfrage mit einer Menge als Ergebnis wird für Mengenoperationen wie etwa A \ B (SQL: not in) benutzt. Auch diese sind für Array Daten aus oben beschriebenen Grund einer (in diesem Zusammenhang) nicht sinnvollen Identität auf Arrays fragwürdig. Ein wichtiges Konstrukt ist jedoch der Vergleich mit einem aus einer Menge von Arrays aggregierten Wert, etwa:

   ```
   WHERE max_cells(c1) > (SELECT max(max_cells(c2)) FROM c2)
   ```

   In **rasdaman** sind Aggregationsoperationen jedoch nur auf Arrays definiert, d.h. eine Menge von Punkten wird zu einem Skalar oder zu einem Array der Ausdehnung 1 aggregiert. Die Aggregation einer Menge von Arrays (oder Menge von Skalaren) ist nicht möglich. Nicht korrelierte aggregierende Unteranfragen sind also für Anfragen auf Mengen von Arrays relevant, wenn sie auch im aktuellen **rasdaman** Array DBMS nicht unterstützt werden, und sollten somit bei paralleler Verarbeitung generell unterstützt werden.

Die parallele Verarbeitung einer Anfrage mit einer Subquery im FROM Teil kann auf verschiedene Art erfolgen. Ausgehend von der Tatsache, dass eine solche verschachtelte Anfrage die Mächtigkeit der Anfragesprache nicht erhöht, kann man feststellen, dass eine solche Anfrage in einem eigenen Optimierungsschritt, etwa durch Variablensubstitution, in eine adäquate Anfrage ohne Verschachtelung überführt werden kann. In diesem Falle ist die parallele Verarbeitung ohne weiteren Aufwand möglich. Wird solch ein zusätzlicher Optimierungsschritt nicht vorgenommen, wird ein komplexerer Baum erzeugt und in parallele Verarbeitungsblöcke separiert. Die hierbei prinzipiell möglichen Algorithmen werden in Abbildung 4.13 (links) skizziert. Die Ausführung aller Applikationen und Selektionen in einem Block hat den Vorteil, dass ein Transfer von Arrays nicht nötig ist. Das Aufteilen der Applikation / Selektion Blöcke hingegen macht Pipeline Parallelisierung, d.h. die parallele Verarbeitung von teuren Operatorblöcken, möglich. Bei ersterer Variante kann durch Partitionierung der Eingabe Datenparallelität erreicht werden. Gegenüber zweiter Variante fallen dadurch keine Transferkosten und Performanzverluste durch Zerstören von Caching Mechanismen an. Ferner ist Datenparallelität weniger anfällig für eine nicht balancierte Lastverteilung als Pipeline Parallelität, ist also vorzuziehen.

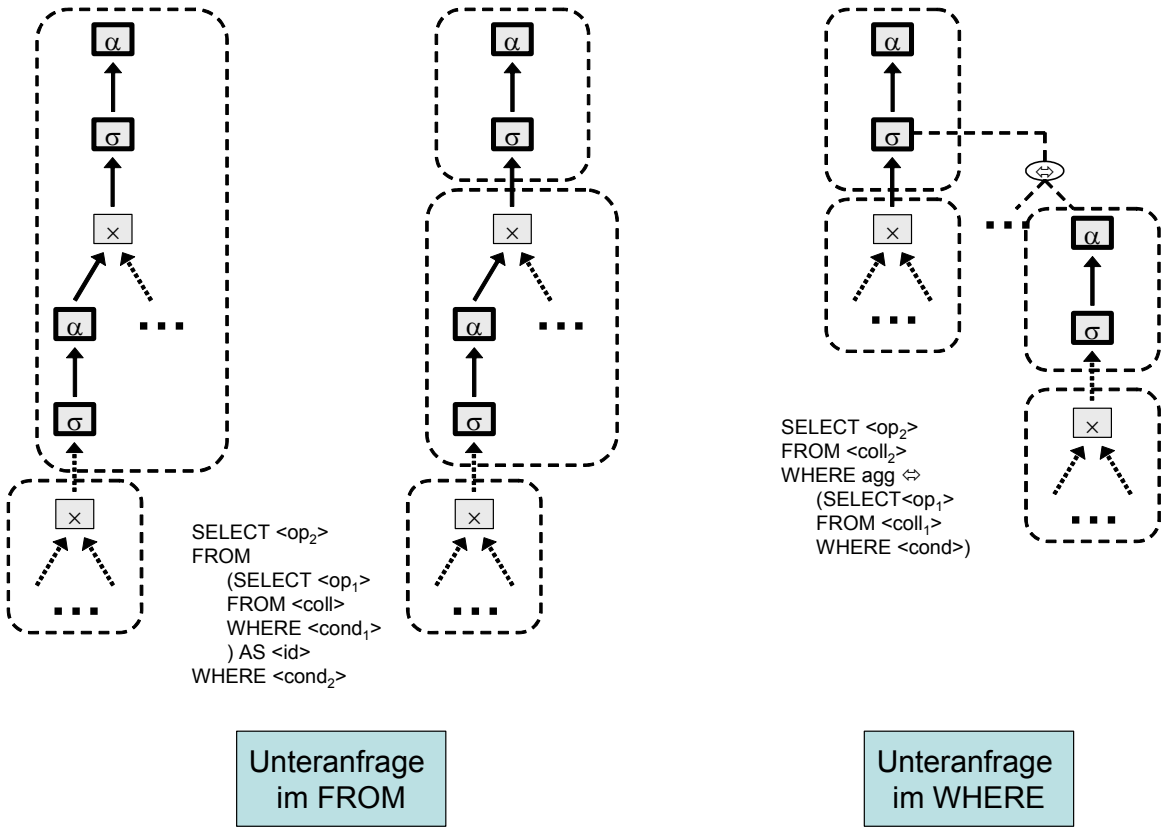


Abbildung 4.13: Parallele Verarbeitung von verschachtelten Anfragen

Bei einer Unteranfrage im WHERE Teil wird diese wie bisher in parallele Blöcke gegliedert: ein Block, der das Kreuzprodukt beinhaltet, kann für eine verbesserte Datenverteilung benutzt werden. Ein Block beinhaltet die teuren Operationen α und σ. Die parallele Instanz für die Server-Client Kommunikation wird für eine Unteranfrage nicht benötigt. An der Schnittstelle zwischen äußerer und innerer Anfrage ist keine Pipeline Parallelität möglich, die Aggregationsoperation ist hier ein so genannter „pipeline breaker", d.h. erst nach der Aggregation kann das Zwischenergebnis weiter genutzt werden. Die so entstehenden Blöcke können nicht weiter zur Bewahrung von Caching Effekten zusammengefasst werden, auch eine Minimierung des Datentransfers ist nicht mehr möglich. Unter Einsatz von Datenparallelität (Kapitel 4.3) kann die Verarbeitung jedoch weiter parallelisiert werden.

Heuristik 9	Parallelisierung verschachtelter Anfragen
	Vermeide Pipeline Parallelität, auch wenn verschachtelte Anfragen eine Lastbalancierung zulassen. Datenparallelität ist auch bei komplexen Anfragebäumen, die durch verschachtelte Anfragen entstehen, die bessere Wahl. Insbesondere die Minimierung von Transferkosten und der Erhalt von Caching Mechanismen sprechen für Intra-Operator Parallelität.

4.3.2.4 Verteilungsstrategien bei mehreren Kollektionen

Wie bereits erwähnt, kann eine Optimierung der Datenverteilung an parallele Prozesse bei Zugriff auf mehrere Kollektionen erreicht werden. Diese Aufgabe übernimmt ein spezieller

interner Prozess, der Tupel Server. Der Name kommt daher, dass dieser Prozess den Kreuzproduktoperator des Anfragebaumes realisiert und somit Tupel von MDD liefert. Das Problem der Datenverteilung auf parallele Prozesse und damit mögliche Redundanzen beim Laden der Objekte werden an folgendem einfachen Beispiel einführend dargestellt.

Beispiel 4.1: Auswirkung von Datenzuteilung bezüglich Ladevolumen

Gegeben seien zwei Kollektionen A und B mit Kardialitäten |A| = 2, |B| = 6. Insgesamt sind folglich wegen der Verknüpfung durch ein Kreuzprodukt 12 Tupel zu verarbeiten: $A_1|B_1$, $A_1|B_2$, …, $A_2|B_6$. Die Verarbeitung wird auf 2 Prozesse verteilt, wobei die Verarbeitungszeit für jedes Tupel beliebig sei. Denkbar sind nun verschiedene dynamische Verteilungsstrategien:

1. Naive Methode
   Die Tupel werden durch das Kreuzprodukt in einer doppelten Schleife generiert. Die Tupel werden in dieser Reihenfolge dynamisch an Prozesse vergeben. Fordert ein Prozess ein neues Tupel an, wird das nächste noch nicht verarbeitete vergeben.
2. Einfache semi-statische Aufteilung mit Repartitionierung
   Die Menge mit der größten Kardinalität wird statisch auf die Prozesse aufgeteilt. Die Menge mit der kleineren Kardinalität muss im Normalfall von allen Prozessen geladen werden. Erst wenn ein Prozess bei Anforderung keine für ihn markierten Tupel erhalten kann, wird das erste noch freie Tupel aus der Liste zugeordnet.
3. Dynamische Analyse zur Vermeidung von redundantem Laden
   Fordert ein Prozess ein Tupel an, wird für alle noch vorhandenen Tupel analysiert, ob der Prozess die Daten neu laden muss oder bereits einen Teil der Daten geladen hat. Muss er die Daten komplett laden, wird möglichst ein Tupel gewählt, das von anderen Prozessen noch nicht (komplett) geladen wurde.

Abbildung 4.14 skizziert die drei Methoden an einem Beispiel. Das Tupel $A_1|B_1$ wurde bereits Prozess 1 zugeteilt, Prozess 2 fordert ein Tupel an.

Die naive Methode gibt das erste freie Tupel $A_1|B_2$ der Liste zurück.

Beim statischen Ansatz wurde die Verteilungsstrategie bereits a priori festgelegt, indem die Menge B mit größerer Kardinalität aufgeteilt wurde. Jedes Tupel mit B_1, B_2, B_3 wird Prozess 1 zugeteilt, der Rest Prozess 2. Mit dieser Strategie bekommt Prozess 2 $A_1|B_4$.

Methode 3 überprüft, ob Prozess 2 ein Tupel zugeteilt werden kann, welches ein Element beinhaltet, das schon einmal diesem Prozess gegeben wurde. Das ist hier nicht der Fall, es müssen prinzipiell alle zwei Objekte neu geladen werden (Attribut L). Folglich wird ein Tupel gesucht, dessen Elemente möglichst noch nicht an einen anderen Prozess gegeben wurden (Attribut S). In unserem Beispiel ist das erste Tupel, welches das erfüllt, $A_2|B_2$.

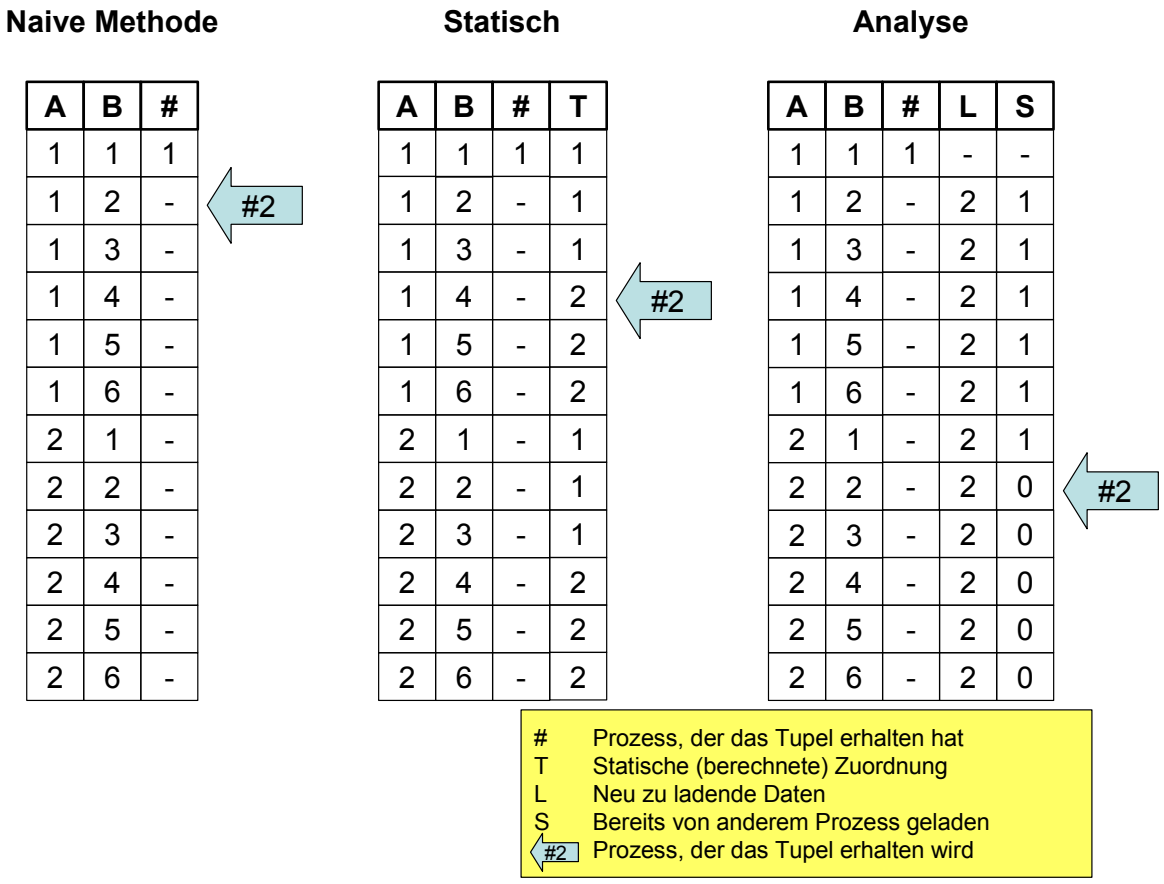


Abbildung 4.14: Beispiel für Zuteilung von Tupel zu Prozessen

Mit allen drei Methoden ist ein redundantes Laden der Quelldaten nicht zu vermeiden. Eine sequentielle Verarbeitung erfordert das Laden aller Objekte aus Kollektion A und aller Objekte der Kollektion B, also A+B, in unserem Beispiel sind das 8 Objekte. Eine zufällige Aufteilung auf Prozesse (Methode 1) kann im Extremfall dazu führen, dass jeder Prozess alle Objekte laden muss, also hier bei zwei Prozessen insgesamt 16 Objekte. Umso größer die Kardinalitäten der Mengen sind, umso wahrscheinlicher wird sich die Anzahle der zu ladenden Objekte an dieses Maximum annähern. Bei der semi-statischen Methode 2 wird der Ladeaufwand durch eine verbesserte Verteilung minimiert. Falls keine Repartitionierung der Mengen erforderlich ist, lädt jeder Prozess 2 Objekte aus Kollektion A und 3 Objekte aus Kollektion B. Insgesamt sind also 10 Objekte zu laden. Die dynamische Verteilung führt wider Erwarten in fast allen Fällen zum Fiasko. Simulationen zeigen, dass hier alle Prozesse immer die kompletten Daten laden. Eine Erklärung für dieses Verhalten wird durch die folgende Analyse geliefert.

Eine Erhöhung von E/A führt in der Regel zu einer schlechten parallelen Performance. Das Ziel ist folglich ein allgemeiner Algorithmus zur Verteilung einer Menge von Tupel auf Prozesse mit minimalem Gesamtladevolumen. Obwohl das Problem einfach erscheint, ist es eine Abwandlung des bekannten BIN PACKING Problems aus der theoretischen Informatik und somit NP-vollständig: die Bestimmung einer Partitionierung und Zuordnung einer Menge von Objekten, so dass eine bestimmte Bedingung erfüllt bzw. eine Zielfunktion minimiert oder maximiert wird. Für diese Klasse von Problemen gibt es im Allgemeinen keine effiziente Lö-

sung, übliche Herangehensweisen sind auf Heuristiken basierende Algorithmen, approximierende Algorithmen oder die Hoffnung, dass eine geringe Kardinalität der Eingabemenge die Effizienz sichert. Wir geben für unser Problem der Partitionierung eine Heuristik an, welche auch für größere Eingabemengen effizient eine sehr gute Lösung liefert.

Das Problem lässt sich zurückführen auf eine Aufteilung (Färbung) eines multidimensionalen Raumes, der durch die referenzierten Kollektionen aufgespannt wird. Sei

d: Anzahl der referenzierten Kollektionen

$C_1, ..., C_d$: Referenzierte Kollektionen

p: Anzahl der Prozesse

s_i: Seitenlänge der Kachel in Dimension i

Dann ist die optimale Seitenlänge für (quadratische) Kacheln, die jeweils einem Prozess zugeteilt werden

$$s_1 = ... = s_d = \left\lfloor \sqrt[d]{\frac{\prod_{i=1}^{d} |C_i|}{p}} \right\rfloor$$

Ist die Kardinalität einer Menge $|C_i|$ kleiner als die optimale Seitenlänge, ergeben sich suboptimale Partitionen. In diesem Fall ist eine gute Approximation an das Optimum, die Dimension mit maximaler Kardinalität durch die Anzahl der Prozesse zu teilen:

1. Sei $m = \{i \mid i = \max(|C_i|)\}$ die Dimension mit maximaler Kardinalität, dann sei die Seitenlänge in dieser Dimension $s_m = |C_m| / p$
2. Die Seitenlängen aller anderen Dimensionen sei $s_i = |C_i|$ für alle $i = 1,\ldots,d$, $d \neq m$

Wir demonstrieren diese Verteilungsstrategie anhand eines einfachen Beispiels.

Beispiel 4.2: Optimale Verteilung eines Kreuzprodukt bezüglich E/A

Die Verteilung lässt sich am besten grafisch veranschaulichen. In Abbildung 4.15 wird das Kreuzprodukt zweier Kollektionen mit jeweils 12 und 16 Objekten auf 4 Prozesse verteilt. Insgesamt sind also 192 2-Tupel auf 4 Prozesse zu verteilen, wobei das Gesamtladevolumen minimiert werden soll.

Grafisch gesehen, muss also ein 2-dimensionaler Raum der Ausdehnung 12*16 in mindestens 4 Partitionen zerteilt werden. Bezüglich Lage und Form der Partitionen für die einzelnen Prozesse lassen sich folgende Tatsachen beobachten:

- Der Raum zerfällt in zusammenhängende Bereiche, also Kacheln für die einzelnen Prozesse. Besteht räumlich gesehen eine Nachbarschaftsbeziehung von Tupel innerhalb einer Partition, bedeutet das, dass (mindestens) ein Ladevorgang gespart werden kann.
- Eine Aufteilung des Raumes in quadratische Kacheln führt gegenüber rechteckigen Kacheln zu einem verbesserten Ladeverhalten. Für eine Kachel ist die Anzahl der zu la-

denden Objekte die Summe der Seitenlängen $s_1 + s_2$, auf den multidimensionalen Fall verallgemeinert $\sum_{i=1}^{d} s_i$.

Bei gegebenem Inhalt der Kachel (entspricht der Anzahl der zu verarbeitenden Tupel) ist die Summe der Seitenlängen bei einem Quadrat, bzw. in einem multidimensionalen Hyperwürfel mit gleichen Seitenlängen, minimal.

- Über die Laufzeit der einzelnen Operationen auf den Tupeln kann keine Aussage gemacht werden, insbesondere sind die Laufzeiten nicht identisch. Eine gute Strategie beinhaltet eine statische Zuteilung zu Prozessen unter Sicherung eines minimalen Ladeverhaltens mit einer dynamischen Repartitionierung bei Skew Effekten. Jedoch soll auch die dynamische Repartitionierung bezüglich E/A eine adäquate Verteilung gewährleisten

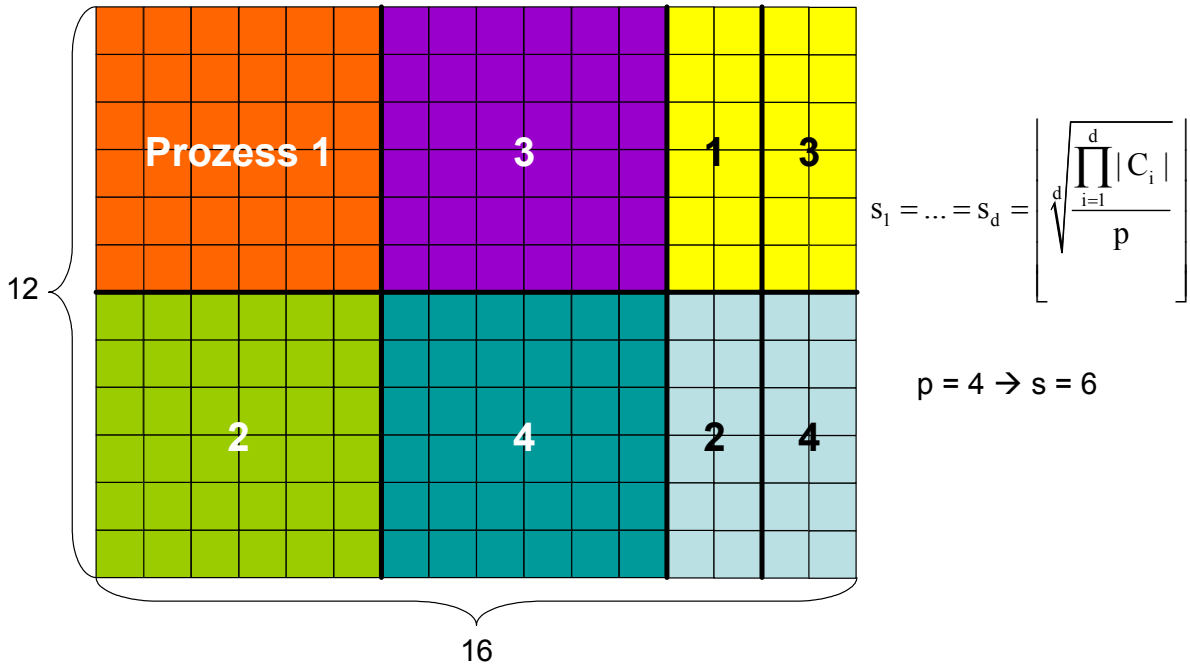


Abbildung 4.15: Beispiel für Zuteilung von 2-Tupeln zu Prozessen

Diese Überlegungen führen zu oben vorgestellter Formel, die den Raum in p quadratische Kacheln und kleinere Restpartitionen aufteilt. Die Menge der Tupel (die Anzahl entspricht dem Produkt der Kardinalitäten der Einzelkollektionen) wird auf p Prozesse aufgeteilt. Durch die Wurzel wird die Seitenlänge von d-dimensionale Quadraten bestimmt (nach unten abgerundet). Im Beispiel aus Abbildung 4.15 ergibt sich eine Seitenlänge von 6. Die 4 Prozesse erhalten nun Partitionen von je 36 Objekten, die ein minimales E/A (von 12 Objekten pro Prozess) gewährleisten. Die restlichen Tupel werden ebenso in (nicht quadratische) Partitionen aufgeteilt, die dynamisch verteilt werden, um Skew Effekte durch variable Berechnungszeiten zu vermeiden. In unserem Beispiel lädt jeder der 4 Prozesse 12 Objekte aus seiner statischen Partition und etwa 2 Objekte aus der dynamischen Partition (falls für die weitere Vergabe bereits geladene Daten berücksichtigt werden). Folglich sind insgesamt etwa 56 Objekte zu laden. Eine sequentielle Verarbeitung (ohne Aufteilung auf parallele Instanzen) erfordert dem gegenüber das Laden von 28 Objekten, nämlich $|C_1| + |C_2|$.

Eine nicht optimierte zufällige Verteilung benötigt in diesem Beispiel bis zu 112 Ladevorgänge. Dieses schlechte Szenario wird - wie bereits erwähnt – durch die dynamische Verteilungsmethode aus Beispiel 4.1 erreicht. Grafisch gesehen zieht diese Zuteilung eine Diagonale durch den multidimensionalen Raum, welche iterierend von allen Prozessen belegt ist. Eine effiziente Verteilung wird dadurch bereits bei Verteilung der ersten Tupel verhindert.

Eine Optimierung der Datenverteilung führt zu einer Reduzierung von E/A und somit zu einer verbesserten parallelen Performanz. Diese Optimierung wird durch den Tupel Server Prozess gewährleistet, der die Operationen des Datenzugriffs und Kreuzprodukt im Anfragebaum verwaltet und die Verteilung zu den Prozessen realisiert.

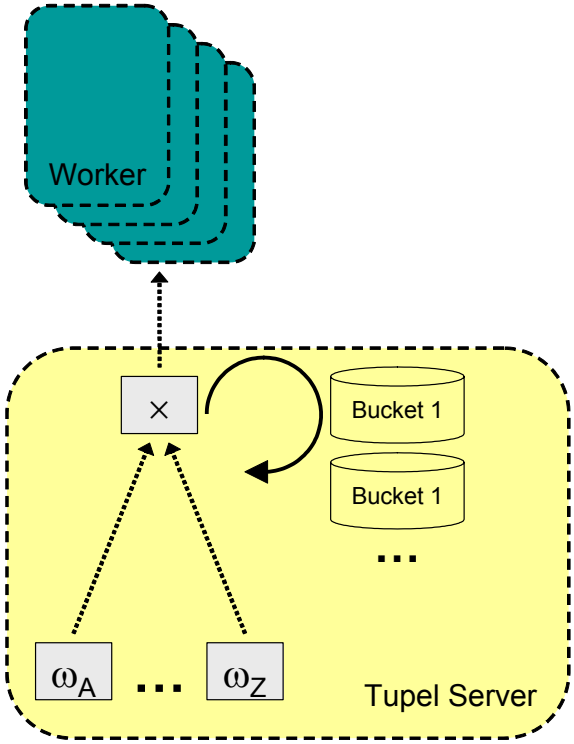


Abbildung 4.16: Optimierung der Datenzuteilung durch den Tupel Server Prozess

Abbildung 4.16 skizziert den Tupel Server Prozess, der das Kreuzprodukt des Anfragebaums bildet und die Tupel in Partitionen für die Prozesse (engl. bucket, Behälter) einteilt. Zu diesem Zweck wird bereits bei Initialisierung durch *open* die komplette Eingabemenge iteriert. Bei jeder Datenanforderung eines Worker Prozesses mittels Aufruf der Iterator Methode *next* wird diesem dann ein Tupel aus dem entsprechenden Bucket gegeben. Erst falls das entsprechende Bucket leer ist, wird dynamisch ein neues Bucket (primär aus der Restmenge, sekundär ein Bucket eines anderen Prozesses) zugeteilt.

Diese statische Zuteilung muss bereits zu Beginn einer Anfrageverarbeitung berechnet werden, d.h. die komplette Funktionalität des Tupel Server Prozess (bis auf das Versenden der Objekte selbst) wird zu Beginn der Anfrage blockierend ausgeführt. Dies macht sich jedoch wegen der geringen Gesamtkomplexität dieses Operators (siehe etwa Abbildung 4.9) nicht negative auf die Performanz bemerkbar.

Heuristik 10	Partitionierung der Datenmenge für Parallelverarbeitung Eine Analyse der zu verarbeitenden Mengen und die Vorberechnung einer statischen Verteilung mit dynamischer Repartitionierung bei Skew Effekten kann E/A Kosten in großen Umfang sparen und somit die parallele Performanz der Anfrage verbessern.

4.3.3 Adaption des Operatorbaumes

Die Adaption eines Anfragebaumes für eine parallele Verarbeitung wird durch das Einfügen spezieller Knoten realisiert, welche die Funktionalitäten für Parallelverarbeitung, etwa das Versenden und Empfangen von Zwischenergebnissen, kapseln. Die Parallelisierung eines Anfragebaumes durch spezielle Iteratoren wurde erstmals von Götz Graefe [Gra93] vorgestellt. Aktuelle parallele RDBMS nutzen ausschließlich diese Technik, wobei der Name für diese Iteratoren variiert. Häufig benutzte Notationen sind:

- *send / receive* (engl.: send, senden, engl.: receive, empfangen) bei IBM DB2 UDB [JMP97],
- *split / merge* (engl.: split, aufteilen, engl.: merge, mischen) in Gamma [DGSB+90] und
- *exchange* (engl.: exchange, austauschen) in Informix Dynamic Server [Zou97] und Volcano [Gra90].

Wir verwenden die Notation analog zu DB2, also *send* und *receive*[55]. Diese Knoten treten im Anfragebaum immer paarweise auf und realisieren das Versenden und Empfangen von Anforderungen (engl.: requests) und Zwischenergebnissen (siehe Abbildung 4.17). Die Notation ist hier insofern ungenau, als dass Anforderungen, genauer die Aufrufmethoden für Iteratoren *open*, *next*, *close* und *reset* im Anfragebaum von oben nach unten gesendet werden, während Zwischenergebnisse von unten nach oben transferiert werden. Die Semantik für *send* und *receive* bezieht sich in unserem Fall auf die Zwischenergebnisse:

- *receive* bzw. *recv*: Iterator im Anfragebaum, der die Methoden *open*, *next*, *close*, *reset* des Iteratorkonzept als Nachricht an einen zuvor spezifizierten Prozess sendet (statt als Methodenaufruf). Zwischenresultate, die durch diese Aufrufe erzeugt werden, werden von diesem Prozess empfangen und weitergegeben.
- *send*: Iterator im Anfragebaum, der die Anfragen *open*, *next*, *close*, *reset* des Iteratorkonzept als Nachricht erwartet. Diese Anfragen werden nach unten weitergereicht und die erhaltenen Zwischenergebnisse an den aufrufenden Prozess gesendet.

[55] Die Parallelisierung des Prädikatbaums einer Selektion σ und des Operatorbaums einer Applikation α, also die in Kapitel 4.4 beschriebene Intra-Objekt Parallelisierung, fügt ebenfalls spezielle Knoten zur Kapselung der parallelen Funktionalität in diesen Teil des Anfragebaums ein. Zur besseren Unterscheidung werden diese Array Operationen, die keine Iteratoren sind, *split* (engl.: Teilen) und *merge* (engl.: Zusammenfügen) genannt, da sie im Wesentlichen genau das für Arrays realisieren.

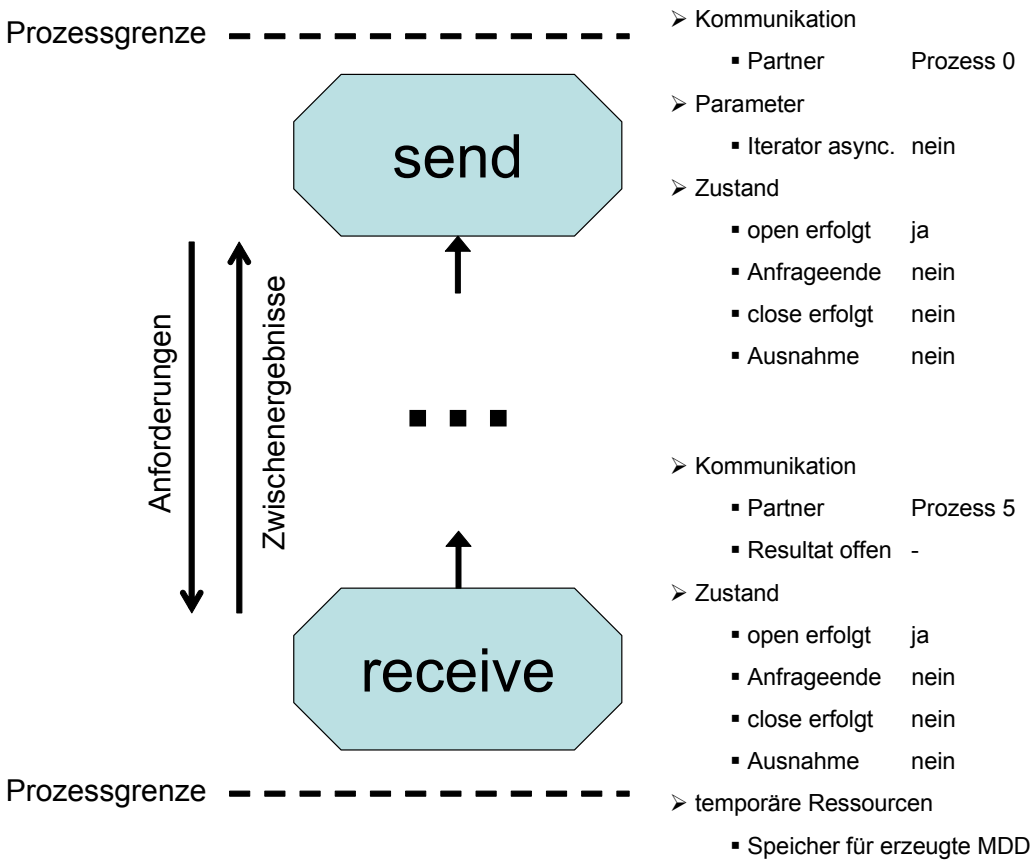


Abbildung 4.17: Funktionalität der Iteratoren send / receive

Darüber hinaus werden jedoch noch weitere durch das Parallelisierungsmodul gesetzte Parameter und Zustände der Parallelverarbeitung in den Knoten gespeichert:

1. **Kommunikationsschema:** Identifizierung der Prozesse, an die die Anforderungen gesendet werden können bzw. von welchen Anforderungen empfangen werden können. Als Kommunikationsschema ist theoretisch eine n:n Verbindung möglich, d.h. jeder Prozess kann beliebig viele Kommunikationspartner haben. Die Kommunikationsstruktur wird durch das Parallelisierungsmodul gesetzt.
2. **Prozessverwaltung:** Identifikation des Zustands der Kommunikationspartner. Beispielsweise enthält ein *receive* eine Verwaltung von arbeitenden (Anforderung versandt, Ergebnis noch nicht erhalten) und freien Partnerprozessen.
3. **Optimierung der parallelen Verarbeitung:** für eine Verbesserung der parallelen Performanz kann vom strengen Iteratorkonzept abgegangen werden (in Abbildung 4.17 als asynchron bezeichnet). So können etwa im *send* Knoten durch sofortiges Iterieren Optimierungen der Datenvergabe erreicht werden. Dies wurde für die aus einem Kreuzprodukt entstehende Menge von Tupeln in Kapitel 4.3.2.4 gezeigt.
4. **Zustandsverwaltung (prozessintern oder global):** Folgende Zustände der Anfrageverarbeitung beeinflussen die Reaktion auf Anfragen bzw. auf den Empfang von Ergebnissen:
   - *open* wurde aufgerufen: ein *receive* sendet einen Aufruf zur Initialisierung der Ressourcen an alle Kommunikationspartner. Ein *send* hingegen darf diese Anforderung nur beim ersten Empfang an seine(n) Eingabeknoten weiterleiten, um eine unnötige Ressourcenbelegung zu vermeiden.

- *Verarbeitung beendet*. Liefert der Eingabeknoten bei einem *next* Aufruf ein leeres Resultat (NULL), so wurde die Eingabemenge komplett iteriert. Ein *send* muss sich diesen Zustand merken, da im Gegensatz zu einer sequentiellen Verarbeitung auch danach *next* Aufrufe von parallel arbeitenden Prozessen eintreffen können. Befindet sich der Prozess in diesem Zustand, wird bis zu einem *close* für jeden *next* Nachricht sofort ein leeres Resultat gesendet (ohne Aufruf von *next* auf den Eingabeknoten).
- *close* wurde aufgerufen: auch hier leitet ein *receive* die Anforderung an alle Kommunikationspartner weiter. Ein *send* darf auf die Anforderung nur das erste Mal reagieren, da der Versuch bereits freigegebene Ressourcen wiederholt freizugeben, zu Fehlern führen kann.
- eine *Ausnahme (engl.: exception)* ist aufgetreten: Fehler in der Verarbeitung einer Anfrage führen nicht zum Absturz des Servers. Stattdessen wird ein Fehlercode durch den Anfragebaum nach oben gereicht und letztlich an das Client Programm zurückgegeben. Hierbei werden im gesamten Baum die belegten Ressourcen wieder freigegeben. Bei Parallelverarbeitung führt eine Ausnahme in einem Prozess zu einer globalen Ausnahme, die von allen Prozessen (genau ein Mal) realisiert werden muss. Dies erfordert Synchronisationsaufwand, der zu geringen Performanzeinbussen im Fehlerfall führen kann.
- Prozessinternes *Ressourcenmanagement*: durch die Aufteilung der Verarbeitung des Anfragebaumes in parallele Prozesse greifen die meisten Mechanismen zur Ressourcenverwaltung und insbesondere Ressourcenfreigabe nicht mehr. So werden etwa im Hauptspeicher befindliche (potentiell an den Client zu übertragene) Objekte erst nach Verarbeitung der kompletten Anfrage im Server-Kommunikationsmodul wieder freigegeben. Da die internen Prozesse dies nicht ausführen, resultiert dieser Mechanismus in Speicherlecks (engl.: memory leaks), die schon nach wenigen Anfragen zu enormen Speicherbedarf für den Server führen und typischerweise zum Programmabsturz führen. Deshalb werden Ressourcenanforderungen in den *receive* Knoten (durch Erzeugen neuer Objekte aus der empfangenen Nachrichten) protokolliert und Ressourcenfreigabe in den *send* Knoten (Löschen versendeter Objekte, da das globale Management nicht greift) eingefügt.

Heuristik 11	Kapselung der parallelen Funktionalität in Iteratoren
	Die parallele Funktionalität kann durch spezielle Iteratoren *send* und *receive* komplett gekapselt werden. Neben den Methoden des Iteratorkonzept werden in diesen alle nötigen Verwaltungsinformationen zur parallelen Verarbeitung gespeichert. Eine Änderung oder Neudefinition von Iteratoren, deren Implementierung oder Anordnung im Baum wirkt sich so nicht auf die Parallelverarbeitung aus.

Abbildung 4.18 skizziert die Adaption eines typischen Anfragebaums für die parallele Verarbeitung, indem Paare von *send / receive* Operatoren eingefügt wurden. Die paarweise auftretenden Parallelisierungsknoten trennen den Baum für die Verarbeitung durch die in Kapitel 4.3.2 beschriebenen Prozessklassen Master, Tupel Server und Worker und realisieren deren Inter-Prozess Kommunikation.

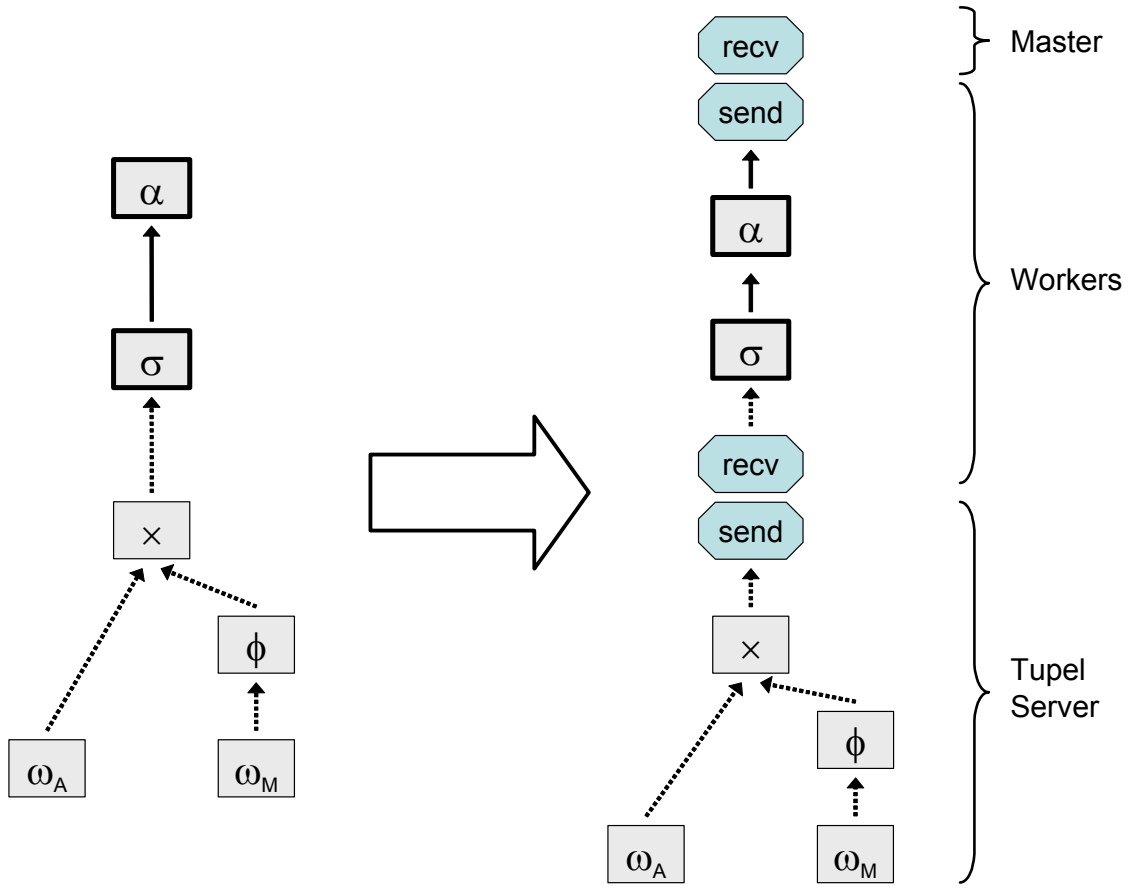


Abbildung 4.18: Adaption eines typischen Anfragebaumes

Über dem Operator × realisieren *send/receive* die Kommunikation zwischen einem Tupel Server Prozess, der × und den Unterbaum verarbeitet, und einer Anzahl von Worker Prozessen, welche α und σ ausführen. An der Wurzel des Anfragebaumes wird die Kommunikation zwischen Worker Prozessen und Master Prozess als *send/receive* eingefügt.

Der generelle Algorithmus für die Adaption des Anfragebaumes berücksichtigt die in Kapitel 4.2.1 analysierten möglichen Anpassungen der Struktur der Bäume. Diese können, wie gesehen, durch Optimierungen, die die Position der Iteratoren verändern, etwa gemeinsame Teilausdrücke oder Pushdown, oder durch verschachtelte Anfragen entstehen. Es sei ein gewünschter Parallelitätsgrad vorgegeben, dieser hängt primär von der eingesetzten Architektur ab[56]. Bei Shared-Everything Architekturen entspricht das der Anzahl der Prozessoren, bei Shared-Disk oder Shared-Nothing der Anzahl der Rechner im Verbund. Ein höherer Parallelitätsgrad der Anfrage führt zu unerwünschten Effekten wie etwa häufige Prozesswechsel bei Shared-Everything Systemen, welche zu einem Performanzverlust führen. Ein zu geringer Parallelitätsgrad der Anfrage führt zu einer suboptimalen Auslastung des Systems.

[56] Sekundär kann etwa bei einem ausgelasteten System (Mehrbenutzerbetrieb) eine Reduzierung des Parallelitätsgrades sinnvoll sein. Wir haben in unserer Implementierung auf das Testen der Systemressourcen bezüglich Last verzichtet, da (wie das ESTEDI Projekt gezeigt hat) ein Mehrbenutzerszenario für ein Array DBMS untypisch ist.

Algorithmus 4.1: Adaption des Anfragebaumes für Intra-Operator Parallelisierung

1. Suche das Kreuzprodukt, das sich im Anfragebaum am weitesten unten befindet (bei verschachtelten Anfragen sind diverse Operatoren × möglich)! Füge ein *send/receive* über diesem Operator ein!

2. Füge über dem Wurzeloperator α ein *send/receive* ein!

3. Für die parallele Ausführung seien p Prozesse mit Rang 0 bis (p-1)[57] vorhanden. Die Verteilung des parallelisierten Anfragebaumes wird folgendermaßen vorgenommen:
   a. Der Prozess mit Rang 0 wird als Master Prozess ausgewiesen. Der Wurzelknoten ist der neu eingefügte *receive* Knoten ganz oben im Anfragebaum. Obwohl der Prozess nur diesen einen Knoten des Baumes ausführt und so die Verarbeitung steuert, ist ein Abtrennen des Baumes unterhalb des Knotens nicht nötig und auch nicht erwünscht. Interne Prozesse wie etwa Optimierung oder Debugging benötigen den kompletten Anfragebaum und sollen aus Performanzgründen auch nicht auf andere Prozesse ausgelagert werden. Die Funktionen zur Anfrageausführung selbst werden nicht wie bisher als Prozeduraufrufe weitergegeben, sondern als Nachrichten an Kommunikationspartner gesendet. Als Kommunikationspartner werden im *receive* Knoten die Prozesse 1 bis (p-2) eingetragen.
   b. Die Prozesse mit Rang 1 bis (p-2) sind Worker Prozesse. Jeder dieser Prozesse erhält als neuen Wurzelknoten den im Baum am weitesten oben befindlichen *send* Iterator. Als Kommunikationspartner wird jeden dieser Knoten jeweils Prozess 0, also der Master Prozess eingetragen. Die Prozesse gehen in einen Zustand der internen Kommunikationsverarbeitung und warten auf Aufrufe des ONC[58] Protokolls, die per Nachricht vom Master Prozess gesendet werden. In den receive Knoten oberhalb von ×, die die Grenze zum Tupel Server Prozess markieren, wird dieser als Kommunikationspartner ausgewiesen.
   c. Der Prozess mit Rang (p-1) ist der Tupel Server Prozess. Wurzelknoten diese Prozesses ist der *send* Knoten über ×, Kommunikationspartner sind respektive die Prozesse 1 bis (p-2).

Die Anzahl der Prozesse p wird in der Implementierung bezüglich des Parallelitätsgrades der Hardware um zwei erhöht. Da Tupel Server und Master nur geringe Ressourcen benötigen und die Ausführung alternierend zu den Worker Prozessen bewerkstelligt werden kann (siehe Kapitel 4.3.4), wird so eine optimale Auslastung des Systems gewährleistet.

4.3.4 Integration in die Anfrageausführung

Die Adaption des Anfragebaumes und der Wurzelknoten für die verschiedenen Prozessklassen bereitet eine parallele Ausführung der Anfrage vor. Die *send* und *receive* Iteratoren kapseln die parallele Verarbeitung, indem die Methoden des ONC Protokolls nicht mehr als Methodenaufruf realisiert werden, sondern als Nachricht an den durch das Parallelisierungsmodul gewählten Kommunikationspartner versendet werden. Die empfangenen Zwischenergeb-

[57] Die Prozesse werden entsprechend der Nomenklatur des für die Implementierung der Inter-Prozess Kommunikation genutzten LAM-MPI von Rang 0 bis Rang (p-1) durchnummeriert.

[58] ONC Protokoll (engl.: *open-next-close*), alternative Bezeichnung für das Iteratorkonzept. Der Name bezieht sich auf die primären Methoden *open*, *next* und *close*

nisse werden aus dem Datenstrom in ursprünglicher Form restauriert und an den aufrufenden Iterator nach oben im Anfragebaum weitergegeben. Methoden der Iteratoren, die nicht zur Anfrageausführung gehören, beziehen sich allein auf Metadaten und werden aus Performanzgründen wie bisher als Methodenaufruf lokal verarbeitet. Dazu gehören vor allem Methoden für Optimierungen des Baumes (Rückgabe aller Kindknoten, Identifikation von Knoten, etc.), aber auch Debugging Methoden wie z.B. die Ausgabe des kompletten Anfragebaumes. Somit besitzt jeder Prozess exakt denselben Anfragebaum, wobei jedoch verschiedene Bereiche durch die jeweiligen Prozesse verarbeitet werden.

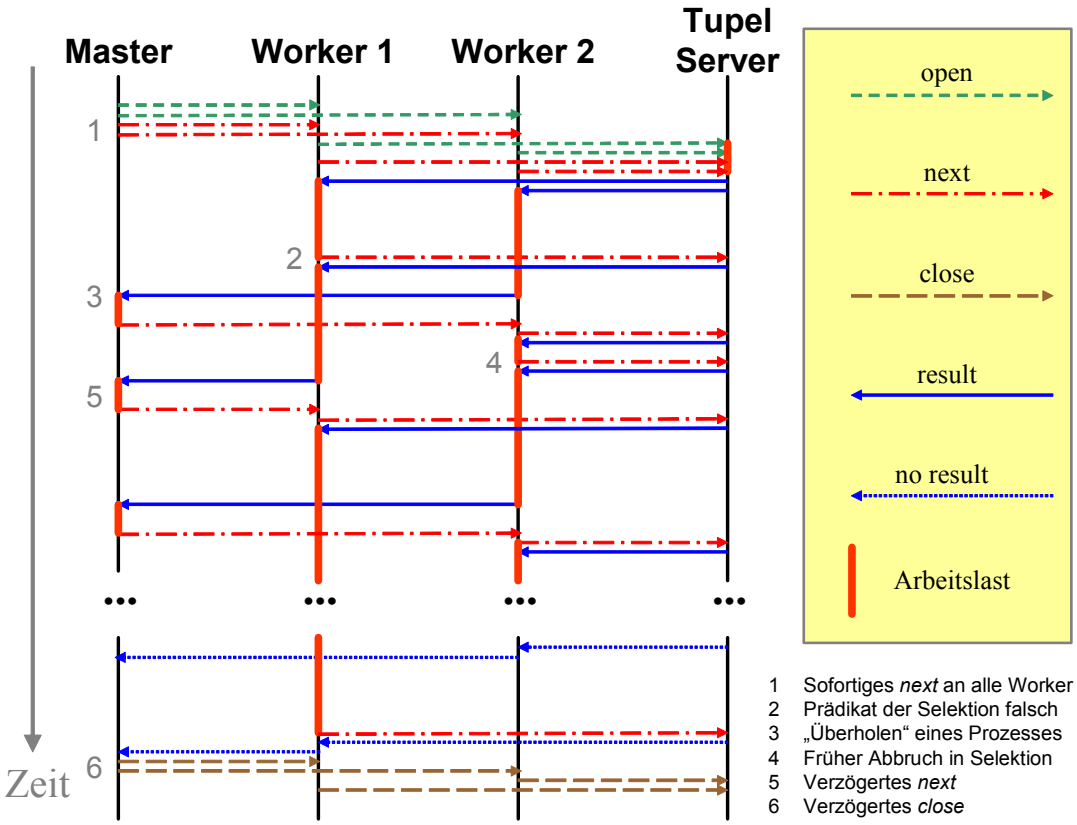


Abbildung 4.19: Integration einer parallelisierten Anfrage in das Iterator-Modell

Abbildung 4.19 zeigt ein Beispiel für eine parallel ausgeführte Anfrage. Entlang der Abszisse sind die Prozesse markiert, zu einer besseren Übersicht beschränken wir uns auf zwei Worker Prozesse (der Parallelitätsgrad der Anfrage wäre hier also 2). Die Ordinate entspricht von oben nach unten dem zeitlichen Verlauf. Der Zeitbedarf der Einzelaktionen ist hier in ihrem Verhältnis nicht absolut zu sehen, vielmehr soll der Nachrichtenaustausch skizziert werden. Ein Pfeil zwischen Prozessen entspricht einer Nachricht, was je nach Architektur durch Nachrichtenaustausch (engl.: message passing) über Netzwerk (etwa mit TCP/IP Protokoll) oder durch (von Threads) gemeinsam genutzte Speicherbereiche (engl.: shared memory) realisiert werden kann. Die hier gezeigten Nachrichten des Iteratorkonzepts sind *open*, *next*, *close*, sowie der Transfer eines Zwischenergebnisses (engl.: *result*) bzw. eines leeres Resultats (programmiertechnisch ein *NULL* Zeiger), um aufzuzeigen, dass die komplette Menge iteriert wurde. Ein markierter (roter) Bereich indiziert, dass der jeweilige Prozess für die auszuführende Aktion blockiert ist, d.h. der Prozess arbeitet unter Nutzung von CPU oder E/A. Die Abbildung zeigt eine korrekt verarbeitete Anfrage, d.h. Fehler während der Anfrage, die durch eine Ausnahmebehandlung realisiert werden und bei paralleler Verarbeitung besondere Sorgfalt benötigen, werden hier nicht berücksichtigt.

Bei sequentieller Verarbeitung wird auf dem Wurzelknoten ein einmaliges *open* aufgerufen, welches durch den kompletten Anfragebaum nach unten gereicht wird und alle erforderlichen Ressourcen reserviert. Dann wird solange mittels *next* ein Element angefordert und von unten nach oben im Baum von den einzelnen Iterator Knoten verarbeitet, bis eine komplette Iteration stattgefunden hat, aufgezeigt durch ein leeres Ergebnis (NULL Zeiger) als Ergebnis eines *next* Aufrufs. Ein einmaliges *close* gibt schließlich im gesamten Baum die Ressourcen wieder frei. Bei Verteilung dieser Arbeit auf verschiedene Prozesse muss insbesondere sichergestellt sein, dass das Sichern und die Freigabe von Ressourcen für jeden Prozess genau einmal passiert. Darüber hinaus muss vor allem berücksichtigt werden, dass eine sequentielle Folge der ONC Aufrufe nicht mehr als gegeben vorausgesetzt werden kann. Einfach ausgedrückt, kann jederzeit ein Prozess den anderen „überholen". In Abbildung 4.19 sind einige solche besonders häufig zu beobachtenden Fälle skizziert.

Die parallele Anfrageverarbeitung wird im Folgenden anhand von Abbildung 4.19 beispielhaft skizziert: auch bei paralleler Verarbeitung wird die Anfrageverarbeitung am Wurzelknoten, also vom Master Prozess aus, gesteuert. Ein open im *receive* Iterator des Master Prozesses wird an alle Kommunikationspartner, also alle Worker Prozesse, gesendet. Dieses *open* wird von allen Workern an den Tupel Server Prozess weitergereicht, dieser darf jedoch nur auf das erste *open* reagieren und die Ressourcen anfordern. Alle weiteren *open* werden ignoriert, da sich der Tupel Server bereits im Zustand *open* befindet[59]. Der Master sendet nun sofort ein *next* an alle Worker (1), die diese Anforderung ihrerseits an den Tupel Server weiterreichen. Durch einen zentralen Prozess für die Datenvergabe ist ein gegenseitiger Ausschluss durch das Konzept des Nachrichtenaustauschs gewährleistet. Ein gleichzeitiger Zugriff kann nicht erfolgen, da dieser zentrale Prozess die Anforderungen in der Reihenfolge bearbeitet, in welcher die Nachrichten eintreffen.

Die Worker erhalten so Tupel von MDD, für die evtl. eine Selektion σ geprüft und dann eine Applikation α ausgeführt wird. Wird das Prädikat der Selektion für das Tupel als falsch evaluiert, wird per *next* sofort ein neues Tupel vom Tupel Server Prozess angefordert (2). Über Laufzeiten der Operationen auf den verschiedenen Prozessen können keinerlei Aussagen gemacht werden. Durch Schwankungen in den Auslastungen einzelner paralleler Instanzen, aber auch durch die Operationen selbst, können Anforderungen andere „überholen". Ein solcher Fall ist (3): durch das Prüfen eines Tupel, das das Prädikat der Selektion nicht erfüllt, wird Worker 1 durch Worker 2 überholt, d.h. Worker 2 liefert ein Resultat vor Worker 1, obwohl der *next* Aufruf erst später erfolgte. In diesem Fall zeigen sich keine Konsequenzen, jedoch kann beispielsweise auch ein leeres Resultat ein gültiges Ergebnis überholen, siehe (6). In diesem Fall darf der Master die Verarbeitung nicht abschließen, bevor er sicher ist, dass alle Prozesse die Anfrage abgeschlossen haben, andernfalls droht der Verlust einer Teilmenge des Ergebnisses. Die Laufzeiten der jeweiligen Operatoren müssen ebenfalls nicht identisch sein. So erlauben etwa Quantoren (Allquantor, Existenzquantor) einen sofortigen Abbruch, falls das Resultat der Operation feststeht (exemplarisch durch die verkürzte Verarbeitungszeit in (4) dargestellt).

Ein interessantes Detail bezüglich der Steuerung der parallelen Auslastung ist in (5) gezeigt: der interne Parallelitätsgrad der Anfrageverarbeitung kann erhöht werden, indem der Master nach Erhalten eines gültigen Zwischenergebnisses sofort ein *next* an den jeweiligen Prozess sendet und dann (parallel) mit dem Restaurieren und Präparieren für den Transfer zum Client fortfährt. Eine solche Implementierung führt jedoch zu einem Parallelitätsgrad der Anfrage

[59] Eine Wiederholung von *open* führt zur Reinitialisierung und kann zum Verlust von Ergebnissen führen.

die über dem tatsächlichen Grad (der Hardware) liegt, was in einer Reduktion der Performanz resultiert. Wird ein *next* erst im Anschluss an diese Aufgabe gesendet, wird eine bessere Auslastung erreicht. Ferner pendelt sich so eine alternierende Auslastung durch Master (komplett CPU Last) und Worker (teils E/A Last) ein. Eine Koppelung der parallelen Verarbeitung mit dem Iteratorkonzept bei Ausführung von p+2 Prozessen (p sei der Parallelitätsgrad der Hardware) zeigt optimale Ergebnisse, da der interne Parallelitätsgrad p beträgt.

Heuristik 12	Intra-Operator Parallelität und Anfrageausführung
	Die Integration von Intra-Operator Parallelität in die Anfrageausführung durch spezielle Iteratoren muss besonders darauf abgestimmt sein, dass eine Ordnung der Ergebnisse bezüglich der Anforderungen verloren gehen kann.

4.3.5 Analyse bezüglich Anfälligkeit für Skew

In Kapitel 3.1 wurden als die drei großen Probleme der Parallelität Initialisierung, Kommunikation und Skew genannt. Die in diesem Kapitel vorgestellte Intra-Operator Parallelität vermeidet die ersten zwei Probleme so weit wie möglich. Initialisierungskosten werden durch Bereitstellung eines Pools von Prozessen und durch einen effektiven Parallelisierungsalgorithmus vermieden. Kommunikationskosten werden durch adäquate Positionierung der Prozessgrenzen im Anfragebaum minimiert. Das dritte Problem, die ungleiche Verteilung der parallelen Last auf die Instanzen, wird in diesem Kapitel näher analysiert.

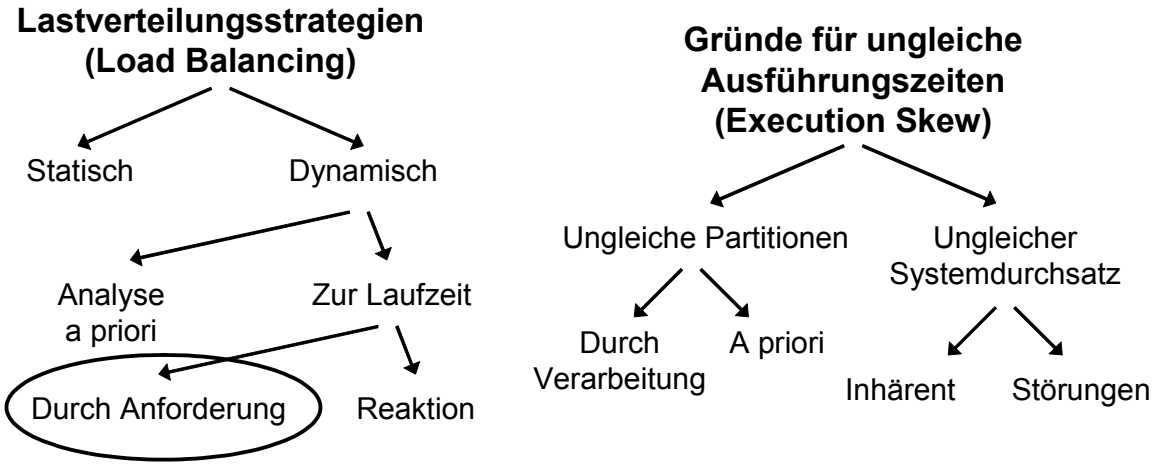


Abbildung 4.20: Klassifizierung der Intra-Operator Parallelisierung bezüglich Verteilungsstrategien

Eine optimale parallele Performanz, oder anders ausgedrückt, ein annähernd lineares Speed-Up, wird vor allem durch eine gute Lastbalancierung (engl.: load balancing) auf die parallelen Instanzen sichergestellt. Der primäre Grund für eine suboptimale parallele Ausführungszeit (siehe Abbildung 4.20, rechts) in der Realität ist eine Ungleichverteilung der Last (engl.: skew effects). Diese tritt meist als Ungleichverteilung der zu verarbeitenden Daten (engl.: data skew) auf und kann auch während der Verarbeitung (engl.: query skew) etwa durch verschiedene Laufzeiten der Operatoren entstehen. Eine Störung der Lastverteilung kann jedoch auch eintreten, wenn die beteiligten parallelen Instanzen einen ungleichen Durchsatz zeigen.

Das kann entweder inhärent sein, etwa bei einer heterogenen Shared-Nothing Architektur oder durch äußere Einflüsse bedingt sein.

Die im vorigen Kapitel beschriebene Einbettung der Parallelverarbeitung in das Iteratorkonzept vermeidet eine Ungleichverteilung der Last größtenteils. Von der dort beschriebenen einfachen aber höchst effektiven Strategie zur Verteilung der Last auf p Prozesse wird lediglich für eine zentrale Berechnung zur optimierten Datenvergabe durch den Tupel Server bei Referenzierung mehrerer Kollektionen (siehe Kapitel 4.3.2.4) abgegangen. Jedoch ist auch diese Zuteilung nicht statisch, ist ein „Behälter“ leer, wird ein freier Prozess aus einem anderen bedient. In der aus [Mär01] entliehenen Klassifizierung der Lastverteilungsstrategien in Abbildung 4.20 (links) ist unsere Verteilungsstrategie einzuordnen als dynamische Verteilung zur Laufzeit durch Anforderung. Die Lastverteilung wird durch die Anfrageverarbeitung dynamisch gesteuert. Werden durch interne oder externe Einflüsse einzelne Prozesse in ihrer Verarbeitung behindert, wird die Arbeitslast dynamisch auf andere Prozesse gelegt.

Grundlegende Probleme der Lastverteilung können mit der vorgestellten parallelen Verarbeitung nur auftreten, wenn in einem Stadium der Anfrageverarbeitung eine Menge von Objekten kleiner als der Parallelitätsgrad p verarbeitet werden muss. Das kann in folgenden Fällen auftreten:

1. Die Kardinalität der Anfrage ist 1. Das ist etwa der Fall, wenn alle referenzierten Kollektionen genau ein MDD beinhalten bzw. wenn in der Selektion eine Einschränkung auf ein Objekt stattfindet. Die diskutierte mengenbasierte Parallelisierung greift hier nicht, da die Verarbeitung innerhalb von Objekten nicht parallel erfolgt.
2. Die Kardinalität der Anfrage ist n mit $n < p$. Der tatsächlich mögliche Parallelisierungsgrad kann hier nicht voll ausgenutzt werden, da eine parallele Instanz auch hier keine kleinere Einheit als ein MDD (-Tupel) verarbeiten kann.
3. Die Kardinalität der Anfrage ist $n > p$. Vor Abschluss der Anfrage tritt jedoch in der Regel der Fall ein, dass die Menge der noch zu verarbeitenden Objekte kleiner ist als der Parallelitätsgrad p. Für diese Phase der Anfrage ist eine Einbusse der parallelen Performanz sicher, was sich je nach Kardinalität der Gesamtanfrage in der parallelen Performanz bemerkbar machen kann.

Bei relationaler paralleler Anfrageverarbeitung werden diese Fälle nicht in Betracht gezogen, da die Kardinalität der Anfrage, also die Anzahl der zu verarbeitenden Tupel, sehr groß ist. Tendenziell tritt eine Beeinflussung der Gesamtperformanz ein, wenn die Kardinalität n sehr klein ist oder der mögliche Parallelitätsgrad p sehr groß. Mengen von Arrays, also in unserer Notation Kollektionen, haben nach den Erfahrungen des ESTEDI Projektes jedoch selten mehr als 100 Objekte. Ganz im Gegenteil, der Extremfall einer Speicherung von einem MDD pro Kollektion ist keine Seltenheit. Die Möglichkeit der Repartitionierung von Arrays (Operator ϕ), macht die Speicherung als Einzelobjekt sogar attraktiver[60]. Diese geringe Kardinalität von Array Mengen führt zu einer schlechten parallelen Performanz für typische Modellierungsszenarien und gängige Anfragen. Wir wollen dies an einem Beispiel demonstrieren.

[60] Applikation und Selektion arbeiten immer auf Granularität von Objekten einer Menge. Die Speicherung als Einzelobjekt mit Repartitionierung während der Anfrage erlaubt diese Operationen auf beliebiger Granularität. So können etwa Bereiche (z.B. Jahre einer Zeitdimension) aus einem MDD selektiert werden, die einem Prädikat genügen.

Beispiel 4.3: Einbußen von paralleler Performanz bei mengenbasierter Parallelität für Anfragen mit geringer Kardinalität

Die Dauer für die Verarbeitung eines einzelnen Objekts spaltet sich – vereinfacht gesehen - auf in eine Zeitspanne t_{io} für das Laden von Sekundärspeicher und t_{cpu} für die Berechnung. Zur Vereinfachung nehmen wir für das Beispiel an, dass diese Dauer der Verarbeitung für alle Objekte identisch ist. Ebenso vereinfachend nehmen wir an, dass bei paralleler Verarbeitung alle Komponenten einer Anfrage voll parallelisierbar sind.

Die nicht-parallele Verarbeitung eines Objekts benötigt also $t_{obj} = t_{io} + t_{cpu}$, die parallele Verarbeitung $t_{obj} = t_{io} + t_{cpu} + t_{par}$. Die Zeitkomponente für Parallelisierung t_{par} wird z.B. für den Transfer des Objekts zwischen Prozessen benötigt.

Die nicht-parallele Verarbeitung einer Anfrage der Kardinalität n bedarf $t_{query} = \sum_{i=1}^{n} (t_{io} + t_{cpu})$, d.h. die Summe der Kosten aller Einzelobjekte. Eine parallele Verarbeitung der Anfrage lässt sich folgendermaßen abschätzen:

$$t_{query} = \left[\left\lfloor \frac{n}{p} \right\rfloor + \mathrm{sgn}(n \bmod p) \right] * \sum_{i=1}^{n} (t_{io} + t_{cpu} + t_{par}) .$$

Mit anderen Worten können fast alle Objekte parallel verarbeitet werden, lediglich ein „Rest“ *n modulo p*, kleiner als der Parallelitätsgrad, ist bezüglich Parallelität evtl. suboptimal.

Es ergeben sich folgende Berechnungen für Speed-Up und prozentualen Anteil des Skew an der kompletten Berechnungszeit:

Speed Up: $\dfrac{n}{\left\lfloor \frac{n}{p} \right\rfloor + \mathrm{sgn}(n \bmod p)}$

Anteil der Anfrage mit Skew: $\dfrac{\mathrm{sgn}(n \bmod p) * t_{obj}}{t_{query}}$

Der Anteil der Anfrage, der nicht voll parallel ausgeführt werden kann, der also data skew aufzeigt, hängt von der Anfragekardinalität n und dem Parallelitätsgrad p ab. Der optimale Wert 0 wird erreicht, wenn *n modulo p* null ist. Abbildung 4.21 zeigt den (theoretisch erreichbaren) Speed-Up einer Anfrage in Abhängigkeit von der Kardinalität für Parallelitätsgrade 4 und 32.

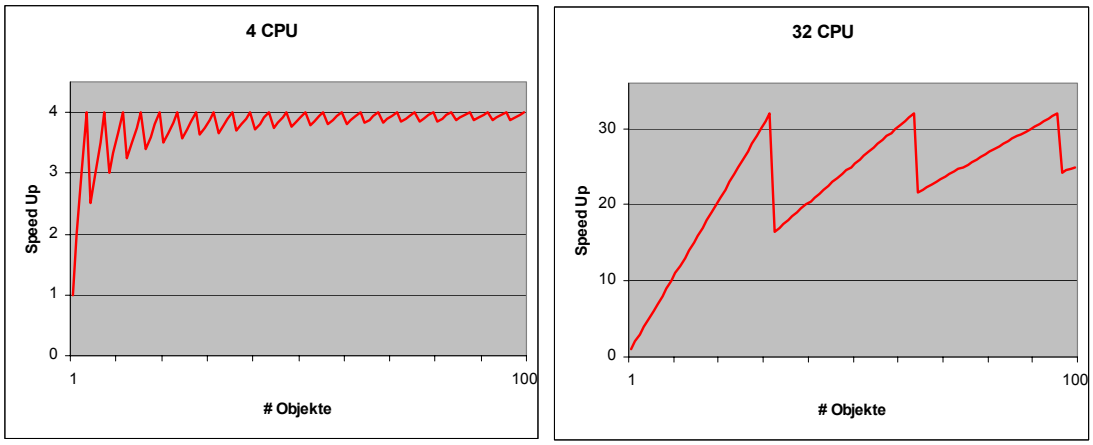


Abbildung 4.21: Speed Up bei Fixierung des Parallelisierungsgrad und Skalierung der Objektanzahl

Insbesondere bei Parallelitätsgrad 32 ist zu erkennen, dass bei für Array Daten typischer geringer Anfragekardinalität ein Einbruch des Speed-Up von bis zu p/2 zu verzeichnen ist. So können 32 Objekte voll parallel mit einem optimalen Speed-Up von 32 verarbeitet werden. Beträgt die Kardinalität 33, muss das letzte Objekt sequentiell verarbeitet werden. Erst ab einer Kardinalität von 450 erreicht der Speed-Up ein dauerhaftes Niveau von über 30. Dazu ist zu erwähnen, dass eine so große Kollektion beispielsweise im Projekt ESTEDI nie beobachtet wurde.

Das Beispiel hat die Anfälligkeit typischer Array Anfragen für Ungleichverteilung der Daten aufgrund geringer Kardinalität aufgezeigt. Das Problem resultiert nicht aus schlechten Verteilungsstrategien, sondern ist inhärent in der parallelen Verarbeitung von Mengen mit geringer Kardinalität. Die Notwendigkeit der Parallelisierung auf einer Granularität kleiner als ein Einzelarray, die im folgenden Kapitel 4.4 gezeigt wird, entsprang nicht nur diesen theoretischen Beobachtungen. Im Projekt ESTEDI zeigte sich sehr bald, dass eine Reihe von Anfragen durch mengenbasierte Parallelität nicht oder suboptimal parallelisiert werden konnte.

4.4 Intra-Objekt Parallelisierung

Intra-Objekt Parallelisierung realisiert die parallele Verarbeitung innerhalb eines einzelnen multidimensionalen Arrays auf Basis von Datenparallelität. Die Daten des Objekts werden zu diesem Zweck in geeigneter Form aufgeteilt und zur weiteren Verarbeitung an parallele Prozesse gesandt. Diese Art der Parallelisierung ist vor allem nötig für Anfragen mit einer Kardinalität von 1, das heißt die Anfragen referenzieren explizit (Kollektion hat nur ein Objekt) oder implizit (durch Operatoren, die die Kardinalität beeinflussen) nur ein Einzelobjekt. Jedoch hat die Analyse in 4.3.5 gezeigt, dass auch für weitere Szenarien eine rein mengenbasierte problematisch ist. Der Algorithmus für die Wahl der Parallelisierungsstrategie, das heißt mengenbasierte Parallelisierung nach 4.3 oder objektbasierte Parallelisierung nach 4.4, folgt in Kapitel 4.5.

Prinzipiell ist die Intra-Objekt Parallelisierung jedoch nur als Unterstützung für die mengenbasierte Datenparallelität aus Kapitel 4.3 zu sehen. Bei Verteilung von kompletten Arrays an

Prozesse ist eine Unabhängigkeit der Datenverarbeitung gewährleistet, die innerhalb von Arrays nicht immer gegeben ist, sondern von den Operationen abhängt. Des Weiteren ist bei Parallelisierung innerhalb von Arrays immer entweder eine Redundanz der von den Prozessen zu ladenden Daten gegebenen oder alternativ ein inhärentes Ungleichgewicht der Datenverteilung.

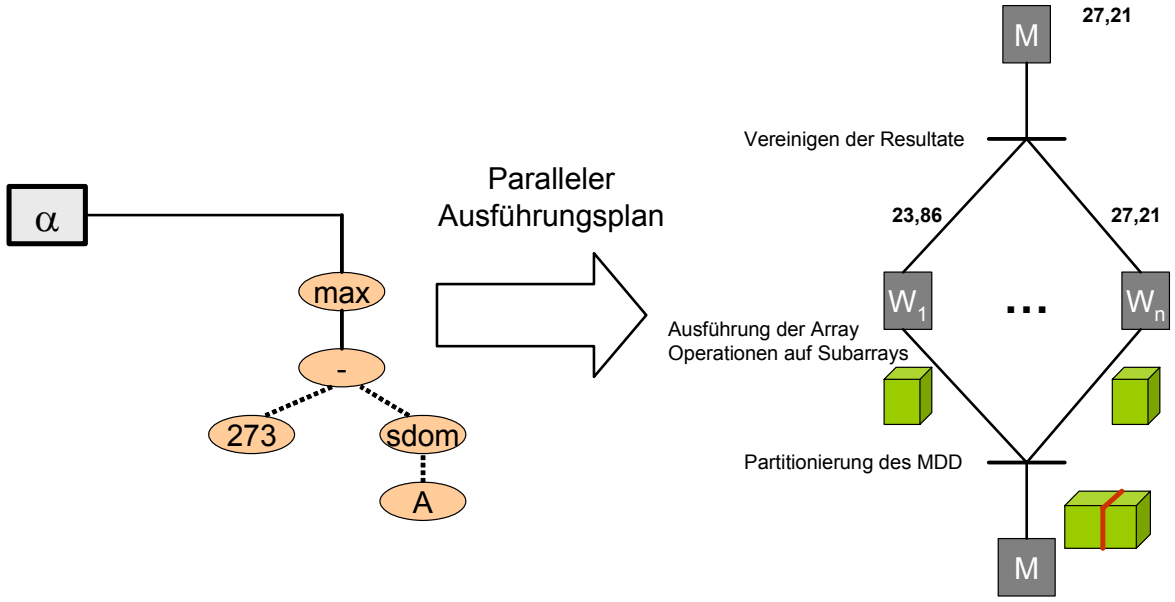


Abbildung 4.22: Anfragebaum und Ausführungsplan (Intra-Objekt Parallelität)

Intra-Objekt Parallelisierung realisiert eine parallele Ausführung von teuren Array Operationen auf Partitionen eines Datenobjekts. Abbildung 4.22 skizziert dies innerhalb einer Applikation α, die eine induzierte Subtraktion und eine Aggregation bezüglich Maximum als Operationen beinhaltet. Der Ausführungsplan zeigt einen Master Prozess (M), der das Eingabearray partitioniert und als kleinere Subarrays an Worker Prozesse (W_1,…,W_n) versendet. Die Operationen können unabhängig und parallel auf diesen Partitionen ausgeführt werden. Die Resultate, in diesem Fall skalare Werte, die den Maxima der Partitionen entsprechen, werden wiederum vom Master Prozess zu einem Gesamtresultat kombiniert.

Diese kurze Beschreibung der Intra-Objekt Parallelisierung spiegelt das Konzept wider, das in den folgenden Kapiteln beschrieben und analysiert wird. Im Detail behandeln diese Kapitel folgende Fragestellungen:

- Analyse der Array Operationen (Kapitel 4.4.1)
  - Welche Array Operationen können theoretisch parallel auf einer Partition eines Array verarbeitet werden (Möglichkeit zur Parallelisierung)?
  - Welche Array Operationen haben eine hohe Komplexität und somit ein gutes Potential bezüglich der parallelen Ausführung (Eignung zur Parallelisierung)?
  - Welche Operatoren werden als Resultat der Analyse parallel verarbeitet?
- Parallele Verarbeitung von Verknüpfungen von Operationen (Kapitel 4.4.2)
  Verknüpfungen von Operationen können durch eine parallele Instanz verarbeitet werden. Eine solche Bildung von Blöcken im Operatorbaum erhöht die Performanz, da der Transfer von Zwischenergebnissen vermieden wird.
- Adaption des Operatorbaumes (Kapitel 4.4.3)

  - Analog zur mengenbasierten Parallelität wird die parallele Verarbeitung und Inter-Prozess Kommunikation in speziellen Knoten gekapselt, die nun jedoch nicht Array Objekte einer Menge (Kollektion) selektieren, sondern Einzelobjekte aufteilen und zum Zwecke einer parallelen Verarbeitung versenden. Zur Unterscheidung zu den vorgestellten mengenbasierten Parallelisierungsknoten werden diese Knoten *split* (engl.: Aufteilen) und *merge* (engl.: Zusammenfügen) genannt.
  - Ein Algorithmus für die Anreicherung des Baumes mit den neu definierten parallelen Operatoren zur Vorbereitung der parallelen Ausführung wird vorgestellt.
- Partitionierung von multidimensionalen Arrays (Kapitel 4.4.4)
  - Eine Partitionierung von multidimensionalen Arrays kann auf verschiedener Granularität und mit unterschiedlichen Algorithmen geschehen. In diesem Kapitel werden Strategien der Aufteilung in Subarrays oder alternativ in Kachelmengen diskutiert. Die für die Implementierung gewählte Vergabe von Subarrays kann unter den Gesichtpunkten der Reduktion von E/A oder Skew optimiert werden.
  - Ein Algorithmus für die Berechnung einer bezüglich Skew und E/A optimierten Partitionierung wird vorgestellt.
- Fusion von Zwischenergebnissen paralleler Instanzen (Kapitel 4.4.5)
  - Die Fusion von Subarrays oder skalaren Werten, die das Ergebnis von parallel verarbeiteten Operationen enthalten, bedarf mitunter eines finalen Berechnungsschritts. Eine Analyse der Operatoren sowie deren Resultate und der Generierung eines Endresultats aus den Resultaten der Subarrays wird in diesem Kapitel vorgenommen.
  - Ein allgemeiner Algorithmus für die Fusion wird vorgestellt.
  - Bei der Fusion von Zwischenergebnissen kann eine komplette Trennung von Array Operatoren und parallelen Operatoren nicht mehr aufrechterhalten werden. Dies wird an zwei Bespielen gezeigt, die in der Implementierung eine besondere Herausforderung boten, nämlich an der Fusion von Durchschnittswerten und Quantoren.

4.4.1 Analyse der Operatoren

In diesem Kapitel werden die in Kapitel 2.2.2 definierten Array Operationen bezüglich ihrer Eignung für eine parallele Verarbeitung analysiert. Wie gezeigt, lassen sich Array Operationen in 5 Klassen gliedern:

1. Elementare Array Operationen: hierunter fällt einerseits die Basisoperationen *marray* zur Definition eines MDD, das unter Verwendung eines Iterators und einer Berechnungsfunktion beliebig gefüllt werden kann. Zweitens der allgemeine Aggregationsoperator *condense*. Dieser bildet die Grundlage für die Aggregation von MDD durch die Definition einer beliebigen Aggregationsfunktion. Während die fest definierten Aggregationsfunktionen (siehe 5.) auf einen skalaren Wert abbilden, kann *condense* bezüglich einer Dimension aggregieren und so mit Hilfe des Konstruktor *marray* ein MDD als Resultat ergeben.
2. Projektionen auf dem Basistyp: ein MDD mit einem strukturierten Basis wird durch *cell* auf ein MDD mit eingeschränktem Basistyp abgebildet. Man spricht auch von Kanalselektion.
3. Geometrische Operationen: die Domäne eines MDD wird verändert. Dies kann eine Einschränkung der Ausdehnung (Operation *trimm*), die Fixierung einer Dimension (Operation *section*) oder das Verschieben der Domäne (Operation *shift*) bedeuten.
4. Induzierte Operationen *ind*: auf einem Basistyp definierte Operationen sind auch auf einem MDD definiert. Hierbei wird die Operation auf jeder Zelle eines MDD indu-

ziert. Man unterscheidet unär und binär induzierte Operationen, je nachdem ob ein MDD oder zwei MDD als Eingabe fungieren.

5. Aggregierende Operationen *agg*: fest definierte Aggregationsfunktionen sind ein Spezialfall der allgemeinen Aggregation. Die gesonderte Definition von häufig genutzten Operationen wie Maximum, Minimum, arithmetischer Durchschnitt, etc. erlauben eine optimierte Implementierung und eine vereinfachte Benutzung durch Integration in die Anfragesprache.

Eine parallele Verarbeitung dieser Operatoren durch Datenparallelität bedarf der **Partitionierung der Eingabemenge**.

Definition 4.2: Partitionierung θ_p eines Array

$P = \{(\underline{x}, v(x))| \underline{x} \in \alpha, v(\underline{x}) = \alpha[\underline{x}]\}$ sei die Menge aller Zellen eines zu partitionierenden MDD α. Die Partitionierung θ_p zerlegt diese Menge von Zellen des MDD α in p Teilmengen, die zusammen das komplette MDD α ergeben:

$$\theta_p(\alpha) = \{P^i = \{(\underline{x}, v[\underline{x}])\} \mid \forall(\underline{x}, v[\underline{x}]) \in P^i \Rightarrow \underline{x} \in \alpha, v(\underline{x}) = \alpha[\underline{x}]\};$$

$$\bigcup_{i=1}^{p} P^i = \alpha; \bigcap_{i=1}^{p} P^i = \varnothing$$

Die Zerlegung ist disjunkt und komplett.

Diese allgemeine Zerlegung eines MDD α in eine Menge von Zellen ist Grundlage der Untersuchung der Operatoren bezüglich paralleler Verarbeitung. In der Implementierung ist eine solch allgemeine Zerlegung nicht effizient machbar. In Kapitel 4.4.4 werden Strategien zur Partitionierung vorgestellt. Diese lassen sich grob unterscheiden in die Partitionierung in eine Menge von Subarrays oder in eine Menge von Kacheln.

Die **Fusion von Teilresultaten**, die auf den Partitionen berechnet wurden, lässt sich in zwei Klassen gliedern, je nach beteiligten Array Operationen:

- Die Vereinigung $\cup$ kombiniert (nach paralleler Verarbeitung der Partitionen) Zellmengen zu einem Array. Diese Fusion kommt zum Einsatz, wenn das finale Resultat ein Array ist, welches aus den Subarrays gebildet wird.
- Eine Funktion f() berechnet aus einer Menge von skalaren Werten oder von Teil-Arrays ein finales Resultat. Diese Fusion wird für die parallele Berechnung von Aggregaten benötigt.

Das Potential der einzelnen Operationsklassen für eine parallele Verarbeitung zeigt sich wie folgt.

4.4.1.1 Parallele Ausführung von elementaren Operationen

Der einzig praxisrelevante Einsatz der Elementaroperationen *marray* und *condense* besteht in der Definition eines neuen Array mittels *marray*, welches durch die Funktion *condense*, also durch ein gegebenes Array und Berechnungsfunktion, gebildet wird. Mathematisch betrachtet, ist dies eine Relation zwischen zwei Punktmengen (und keine Abbildung). Technisch ge-

sehen kann bei paralleler Ausführung die Menge der Eingabepunkte im Allgemeinen nicht partitioniert werden, was zu einer schlechten Performanz führt.

$$\text{marray}(\text{cond}(\alpha)) = \bigcup \text{marray}(\theta(\text{cond}(\alpha)))$$

$$\text{Beachte: } \text{marray}(\text{cond}(\alpha)) \neq \bigcup \text{marray}(\text{cond}(\theta(\alpha)))$$

Aus diesem Grund und wegen der geringen Praxisrelevanz wurde die parallele Ausführung von elementaren Operationen nicht implementiert.

4.4.1.2 Parallele Ausführung von Projektionen auf dem Basistyp

Logisch gesehen entspricht die Operation der Abbildung eines Arrays auf ein Array mit reduziertem Basistyp. Im Array DBMS **rasdaman** ist die Basistypprojektion als Eintrag in den Metadaten (Adaption des persistenten MDD) realisiert, das Laden der Daten und die Einschränkung auf den entsprechenden Typ erfolgt erst, wenn die Zellwerte selbst für Operationen oder für die Übertragung benötigt werden. Somit verschmilzt die eigentliche Operation mit einer Operation, die die Zellinhalte für Berechnungen lädt. Eine Parallelisierung der Basistypprojektion selbst ist aus diesen Gründen nicht sinnvoll.

4.4.1.3 Parallele Ausführung von geometrischen Operationen

Ähnlich wie bei Basistypprojektionen muss bei geometrischen Operationen logische Operation und Implementierung getrennt gesehen werden. Eine geometrische Operation verändert die Domäne, was durch Indexzugriff sehr effektiv durchführbar ist. In der Implementierung im Array DBMS **rasdaman** wird sogar gänzlich auf das Laden der Daten in der jeweiligen Operation verzichtet, es wird lediglich die Domäne des Arrays in den Metadaten angepasst. Ein Laden der Daten erfolgt immer zum spättestmöglichen Zeitpunkt, d.h. wenn auf Zellen zugegriffen wird. Die parallele Verarbeitung von geometrischen Operationen ist folglich nicht sinnvoll. Wie später erläutert wird, stört darüber hinaus eine geometrische Operation innerhalb der parallelen Verarbeitung die Lastbalancierung.

4.4.1.4 Parallele Ausführung von induzierten Operationen

Induzierte Operationen sind perfekte Kandidaten für eine parallele Verarbeitung. Die auf den Zelltypen definierte Operation wird auf Arrays induziert. Aus der Definition der induzierten Operationen ergibt sich, dass die Operationen auf den einzelnen Zellen vollkommen unabhängig von allen anderen Zellen sind. Unär und binär induzierte Operationen lassen sich demnach wie folgt parallelisieren:

$$\text{ind}(\alpha) \equiv \bigcup \text{ind}(\theta(\alpha)) \equiv \bigcup_{i=1}^{p} \text{ind}_i(P^i)$$

$$\text{ind}(\alpha,\beta) \equiv \bigcup \text{ind}(\theta(\alpha),\theta(\beta)) \equiv \bigcup_{i=1}^{p} \text{ind}_i(P^i_\alpha, P^i_\beta)$$

Der mögliche Parallelisierungsgrad entspricht theoretisch der Anzahl der Zellen im Array. Es sei angemerkt, dass die Partitionierung im Falle der binären Operation identisch sein muss, das heißt, für die jeweiligen Partitionen eine identische Vektormenge enthalten muss[61].

[61] Die Identität von Partitionen kann durch Angabe entsprechender Parameter für θ sichergestellt werden. Für die letztendlich realisierte Partitionierung in Subarrays (siehe Kapitel 4.4.4) entsprechen diese Parameter einer Menge von Domänen der resultierenden Partitionen, also Subarrays.

4.4.1.5 Parallele Ausführung von aggregierenden Operationen

Die definierten Operationen zur Aggregation von Arrays sind ebenfalls sehr gut parallel zu verarbeiten, indem die Aggregatsfunktion auf die einzelnen Partitionen angewandt wird.

$$agg(\alpha) \equiv f(agg(\theta(\alpha))) \equiv f(\sum_{i=1}^{p}(agg_i(P^i)))$$

Die Berechnung des finalen Aggregats wird durch eine zusätzliche Funktion f() vorgenommen. Diese berechnet aus den Ergebnissen der Partitionen das Endresultat. Diese Funktion wird für die verschiedenen Aggregate im Speziellen folgendermaßen realisiert.

1. Aggregation zu einem skalaren Wert: <D, T> → <D', T'> mit dim(D') = 0

$$max(\alpha) \equiv max'(max(\theta(\alpha)))$$
$$min(\alpha) \equiv min'(min(\theta(\alpha)))$$
$$sum(\alpha) \equiv sum'(sum(\theta(\alpha)))$$
$$count(\alpha) \equiv sum'(count(\theta(\alpha)))$$
$$all(\alpha) \equiv and'(all(\theta(\alpha)))$$
$$some(\alpha) \equiv or'(some(\theta(\alpha)))$$
$$avg(\alpha) \equiv \frac{sum(\alpha)}{count(\alpha)} \equiv \frac{sum'(sum(\theta(\alpha)))}{sum'(count(\theta(\alpha)))}$$

   Für Maximum, Minimum und Summe entspricht diese Funktion f() den auf den Arrays definierten Aggregatsfunktionen. Sie erhalten jedoch als Eingabe nicht ein Array, sondern eine Menge von Skalaren, die das Ergebnis der Operation auf den Partitionen repräsentieren. Zur Unterscheidung werden sie als *agg'* (also etwa *max'*) bezeichnet. Das Ergebnis der Quantoren *all* (Allquantor) und *some* (Existenzquantor) wird durch ein logisches AND bzw. OR auf den booleschen Ergebnissen der Partitionen berechnet. Eine spezielle Behandlung erfordert die Berechnung des Durchschnitts *avg*. Aus den Ergebnissen der Durchschnittsoperation für die Partitionen lässt sich das Endresultat nicht ermitteln[62]. Die Berechnung erfordert die Summen *sum* und Zellenanzahl *count* aller Partitionen (ene detaillierte Analyse erfolgt in Kapitel 4.4.5.2).

2. Aggregation zu einem Array der Ausdehnung 1:
   <D, T> → <D', T'> mit dim(D) = dim(D') und extent(D') = 1

   Die Aggregationsfunktionen *max_pos* und *min_pos* werden zur Ermittlung der Menge von Positionen verwendet, an denen das Extremum auftritt[63].

$$max\_pos(\alpha) \equiv \{max\_pos(\theta(\alpha)) \mid v(max\_pos(\theta(\alpha))) = max'(v(max\_pos(\theta(\alpha))))\}$$
$$min\_pos(\alpha) \equiv \{min\_pos(\theta(\alpha)) \mid v(min\_pos(\theta(\alpha))) = min'(v(min\_pos(\theta(\alpha))))\}$$

[62] Partitionen müssen (und können in der Regel) nicht gleich groß sein: folglich gilt $avg(\alpha) \neq avg'(avg(\theta(\alpha)))$

[63] In rasdaman besteht das Problem der fehlenden Zuordnung der Resultatmenge (der Maxima) zu den jeweiligen Arrays der referenzierten Kollektion. Aus diesem Grund wurden in rasdaman zwei Ausprägungen dieser Funktion implementiert: die erste ignoriert dieses Problem der Zuordnung. Eine alternative Funktion liefert ein Array mit booleschen Werten, das für jede Position des Maximums ein *true* enthält. Der Nachteil der zweiten Funktion besteht in der zu übertragenden Datenmenge (komplettes Array).

Die Funktion *v()* steht für die Extraktion des Zellwertes (engl.: value, Wert) aus dem Array der Ausdehnung 1. Es wird also das Array aus der Ergebnismenge als Resultat selektiert, dessen Zellwert den maximalen bzw. minimalen Wert aufweist.

3. Aggregation zu einem Array gleicher Dimensionalität und geringerer Ausdehnung:
<D, T> → <D', T'> mit dim(D) = dim(D') und extent(D') < extent(D)

   Die Operation *scale(α, s)* bewirkt eine Skalierung eines Array α um den Faktor s, also die Berechnung eines in der Größe veränderten Arrays, welches jedoch „ähnlich" sein soll und die Charakteristika des Ursprungsarrays beibehält. Die Funktion ist vor allem wegen einer Reduktion des Datenvolumens vor einer Übertragung zu Client Anwendungen von großer praktischer Relevanz. Eine parallele Ausführung ist durch Partitionierung und Vereinigung der partiellen Ergebnisarrays möglich:

$$\text{scale}(\alpha, s) \equiv \bigcup \text{scale}(\theta(\alpha, s)) \equiv \bigcup_{i=1}^{p} \text{scale}_i(P^i, s)$$

   Diese einfache und effektive parallele Verarbeitung einer Skalierungsfunktion ist nicht prinzipiell möglich, sondern basiert auf der Implementierung der Funktion und ist in unserem Fall für **rasdaman** möglich. Eine Diskussion über Skalierungsfunktionen und deren Eignung für parallele Verarbeitung erfolgt weiter unten.

4. Aggregation zu einem Array geringerer Dimensionalität:
<D, T> → <D', T'> mit dim(D) < dim(D')

   In diese Klasse von Aggregationsoperationen fallen Konvertierungen in ein anderes Datenformat. In der Praxis am wichtigsten dürfte wohl die Funktion *jpeg* sein, die ein generisches 2-dimensionales Array, welches einen nicht strukturierten Zelltyp haben muss, in ein 1-dimensionales Array überführt, das dem Datenformat JPEG[64] entspricht. Hierzu erfolgt eine Abbildung auf Farbwerte, die im Bereich blau (RGB: 0, 0, 255) für das Minimum bis rot (RGB: 255, 0, 0) für das Maximum liegen. Neben einer vereinfachten Visualisierung der Daten durch das standardisierte Datenformat, etwa innerhalb einer HTML Seite, ist die Konvertierung wegen der inhärenten Datenkompression interessant und praxisrelevant. Weitere wichtige Konvertierungsoperatoren sind *gif*, *bmp*, *png*, *hdf*. Eine parallele Verarbeitung dieser Konvertierungen wird in der wissenschaftlichen Literatur eingehend untersucht (siehe etwa [CD94]). In unserer Implementierung, also im parallelen DBMS **rasdaman**, werden Formatkonvertierungen aus externen Bibliotheken genutzt, entziehen sich somit der direkten Einflussnahme. Eine parallele Verarbeitung dieser Funktionen bedarf der Nutzung von neuen Bibliotheken, die dies unterstützen oder alternativ deren Reimplementierung.

[64] JPEG (Joint Photographic Experts Group), GIF (Graphic InterFace), BMP (bit-mapped file format), PNG (Portable Network Graphics) sind bekannte Datenformate für Bilddateien. HDF (Hierarchical Data Format, siehe http://hdf.ncsa.uiuc.edu/) ist ein Format für multidimensionale Daten, das von NCSA (National Center for Supercomputing Applications) entwickelt wurde.

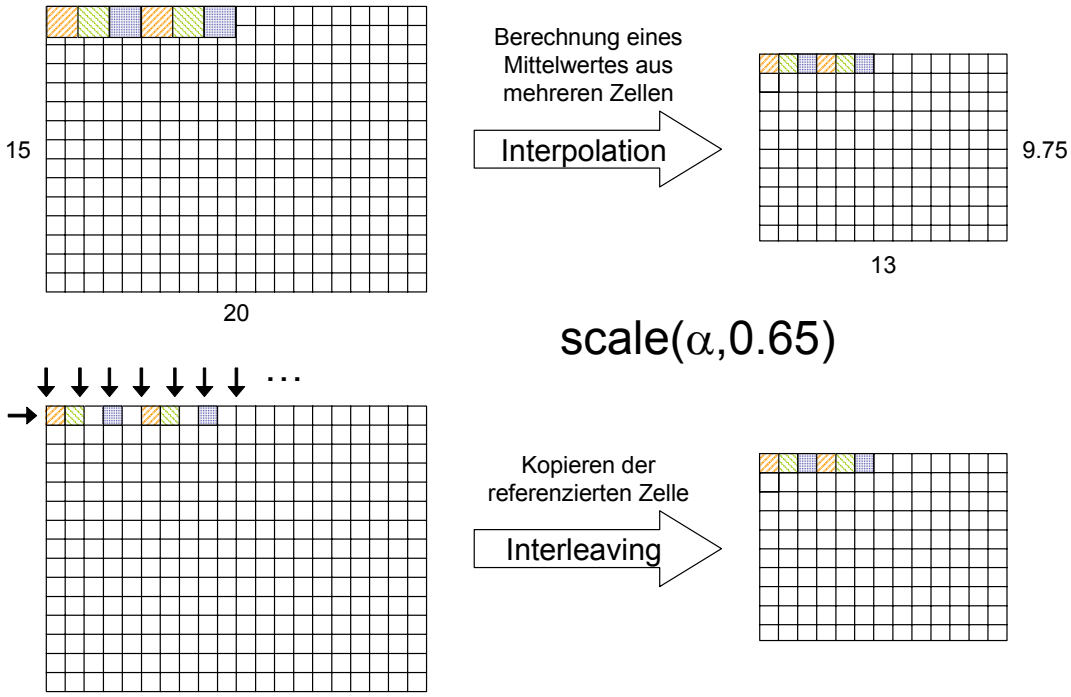


Abbildung 4.23: Skalierung multidimensionaler Arrays

Eine offene Frage bleibt zu diskutieren: unter welchen Umständen kann eine Skalierungsfunktion parallel ausgeführt werden? Eine Funktion scale(α, s) → β bildet ein Quellarray α auf ein Zielarray β ab mit dim(β) = dim(α) und ∀ 0 ≤ i < dim(α): extent(β[i]) = ⌊s * extent(α[i])⌋. Ein Beispiel ist in Abbildung 4.23 gegeben: hier wird durch *scale(α, 0.65)* ein Array mit Ausdehnung [20; 15] in ein Array der Ausdehnung [13; 9.75] skaliert. Die ist kein diskretisiertes Array, muss folglich durch Abrundung in ein Array der Ausdehnung [13; 9] umgewandelt werden. Die Berechnung eines Zellwertes des Zielarrays ist nun auf zwei verschiedene Arten möglich:

- Interpolation ist der in der Bildbearbeitung primär eingesetzte Algorithmus zur Änderung der Größe eines Rasterbildes. Der Wert des Zielpunktes wird hierbei berechnet, indem erstens der Bereich des Quellarrays berechnet wird, welcher zur Berechnung herangezogen wird. In einem zweiten Schritt wird ein Mittelwert aus den beteiligten Zellwerten gebildet unter Gewichtung des Anteils. In unserem Beispiel in Abbildung 4.23 oben wird der Zellwert an Position [0, 0] des Zielarrays aus den Werten der Positionen [0, 0], [0, 1], [1, 0] und [1, 1] im Quellarray gebildet.
- Interleaving überträgt lediglich selektierte Zellwerte des Quellarrays unverändert in das Zielarray und lässt alle weiteren außer Betracht. Hierbei wird für jede Zelle des Zielarrays die entsprechende Position im Quellarray berechnet und der Inhalt dieser Zelle kopiert. Dies ist in Abbildung 4.23 unten skizziert.

Eine Interpolation ist keine injektive Abbildung, die Berechnung eines Zielpunktes bedarf in der Regel mehrerer Quellpunkte. Bei paralleler Verarbeitung wäre somit eine Partitionierung des Quellarrays in nicht disjunkte „überlappende“ Bereiche nötig.

In rasdaman wird die Methode des Interleaving eingesetzt, um die Komplexität der Operation und den Ressourcenverbrauch der Implementierung in Grenzen zu halten. Diese Abbildung zwischen zwei Punktmengen ist bijektiv, das heißt, jede Zelle des Zielarrays wird aus genau einer Zelle des Quellarrays gebildet. Jede Zelle kann folglich aus genau einer Partition des Quellarrays gebildet werden, eine Parallelisierung ist problemlos möglich: jeder Prozess prüft für alle Punkte der Zieldomäne, ob seine Partition den zugrunde liegenden Punkt der Quelldomäne enthält und fügt ihn in diesem Fall in die Ergebnismenge ein.

Als Ergebnis der Analyse von Array Operationen bezüglich der Eignung auf parallele Verarbeitung lässt sich festhalten:

Heuristik 13 Parallele Verarbeitung von Array Operationen
Eine parallele Verarbeitung von Array Operationen ist für die Klasse der induzierten und der aggregierenden Operationen sinnvoll. Alle Operatoren dieser Klassen außer Formatkonvertierungen lassen sich hervorragend parallel berechnen.

4.4.2 Parallele Ausführung von Verknüpfungen von Operatoren

Bisher wurde analysiert, in welchem Umfang die einzelnen Array Operationen parallel auf Partitionen der Eingabe ausgeführt werden können. In diesem Kapitel wird die parallele Verarbeitung einer Folge solcher Operationen, also einer Verknüpfung analysiert. Eine Ausführung mehrerer Operationen auf den gebildeten Partitionen P^i, verbunden mit einer möglichst verzögerten Fusion, sichert eine effiziente parallele Performanz durch Vermeidung des Transfers von Zwischenergebnissen.

Die parallele Verarbeitung einer Verknüpfung dieser Array Operationen ist durch folgende Umformungen realisierbar:

$$\theta(\bigcup P) \equiv P$$

Eine Partition P kann hierbei durch direkte Bindung an ein Array entstehen oder das Resultat einer Operation mit Ergebnistyp Array sein. Entsteht also durch die Parallelisierung von Einzeloperatoren eine Fusion direkt gefolgt von einer (Re-) Partitionierung, so kann dies einfach weggelassen werden[65]. Grafisch kann das auch als „Verschieben“ der split Operation unter eine parallelisierbare Operation gesehen werden.

Beispiel 4.4: Intra-Objekt Parallelisierung einer Anfrage

Wir skizzieren die Adaption einer Anfrage durch Partitionierung und Fusion anhand eines Beispiels, indem der Anfragebaum und die algebraischen Umformungen präsentiert werden.

```
SELECT avg(a[0:10,0:10].x - shift(b[10:20,10:20].y,[-10,-10]))
FROM collA AS a, collB AS b
```

[65] Identische Parameter für die Partitionierung innerhalb einer Anfrage werden durch den Parallelisierungsalgorithmus (Algorithmus 4.2) und den Partitionierungsalgorithmus (Algorithmus 4.3) sichergestellt.

Durch diese Anfrage wird die durchschnittliche Differenz eines Kanals in eingeschränkten Bereichen zweier MDD ermittelt. Die beteiligten Array Operatoren sind geometrische Einschränkung (hier in Anlehnung an die Implementierung als *sdom* bezeichnet), Basistypprojektion *dot*, Verschieben der Domäne *shift*, binär induzierte Subtraktion und Durchschnittsbildung als aggregierende Operation. Durch die gezeigten algebraischen Umformungen ergibt sich folgender Ausdruck:

$$\text{avg}(\text{minus}(\alpha[0:10,0:10,0].\text{x},\text{shift}(\beta[10:20,10:20].\text{y},[-10,-10]))) \equiv$$
$$\frac{\text{sum}(\theta(\bigcup\text{minus}(\theta(\alpha[0:10,0:10,0].\text{x}),\theta(\text{shift}(\beta[10:20,10:20].\text{y},[-10,-10])))))}{\text{count}(\theta(\bigcup\text{minus}(\theta(\alpha[0:10,0:10,0].\text{x}),\theta(\text{shift}(\beta[10:20,10:20].\text{y},[-10,-10])))))} \equiv$$
$$\frac{\text{sum}(\text{minus}(\theta(\alpha[0:10,0:10,0].\text{x}),\theta(\text{shift}(\beta[10:20,10:20].\text{y},[-10,-10]))))}{\text{count}(\text{minus}(\theta(\alpha[0:10,0:10,0].\text{x}),\theta(\text{shift}(\beta[10:20,10:20].\text{y},[-10,-10]))))}$$

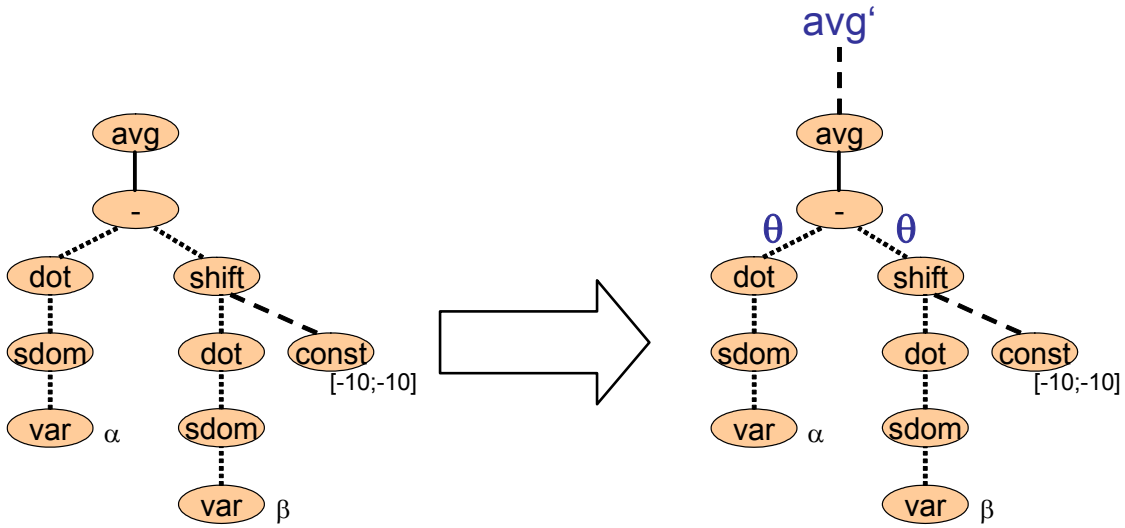


Abbildung 4.24: Partitionierung und Berechnung des finalen Resultats für Operatorverknüpfungen

In Abbildung 4.24 ist die entsprechende Umformung des Operatorbaumes skizziert: möglichst weit oben im Operatorbaum wird die Partitionierung mit anschließender Fusion eingefügt. Die Umformungen des algebraischen Ausdrucks entsprechen einer Verschiebung der Partition nach unten im Baum. Die gut parallelisierbaren Operatorklassen Induktion und Aggregation werden somit durch Partitionierung und Fusion gekapselt und können ohne eine Repartitionierung der Daten parallel ausgeführt werden. Ein Verschieben der Partitionierung unter geometrische Operationen findet nicht statt, eine ungleiche Lastverteilung wird so vermieden.

Die parallele Ausführung aufeinender folgender Array Operationen in einer Instanz ist also problemlos möglich. Die Bildung möglichst großer Blöcke innerhalb der Operatorbäume minimiert Inter-Prozess Kommunikation und wird folglich vorgenommen.

Heuristik 14	Parallele Ausführung mehrerer Array Operationen Sind Array Operationen parallelisierbar, so ist auch deren Verknüpfung parallelisierbar. Die Identifikation möglichst langer Verknüpfungsketten und deren parallele Ausführung minimieren den Transfer von Zwischenergebnissen und sichern somit Performanz.

4.4.3 Adaption des Operatorbaumes

In diesem Kapitel werden Operatoren zur Partitionierung von Arrays und zur Fusion der Resultate, welche aber auch die Inter-Prozess Kommunikation übernehmen, vorgestellt. Darüber hinaus werden der Algorithmus zur Adaption des Anfragebaumes sowie die sich ergebende Anfrageausführung präsentiert. Auf Algorithmen für das eigentliche Partitionieren und Zusammenführen von Objekten, mit anderen Worten die Realisierung der primären Aufgaben der Operatoren, wird in den folgenden Kapiteln eingegangen.

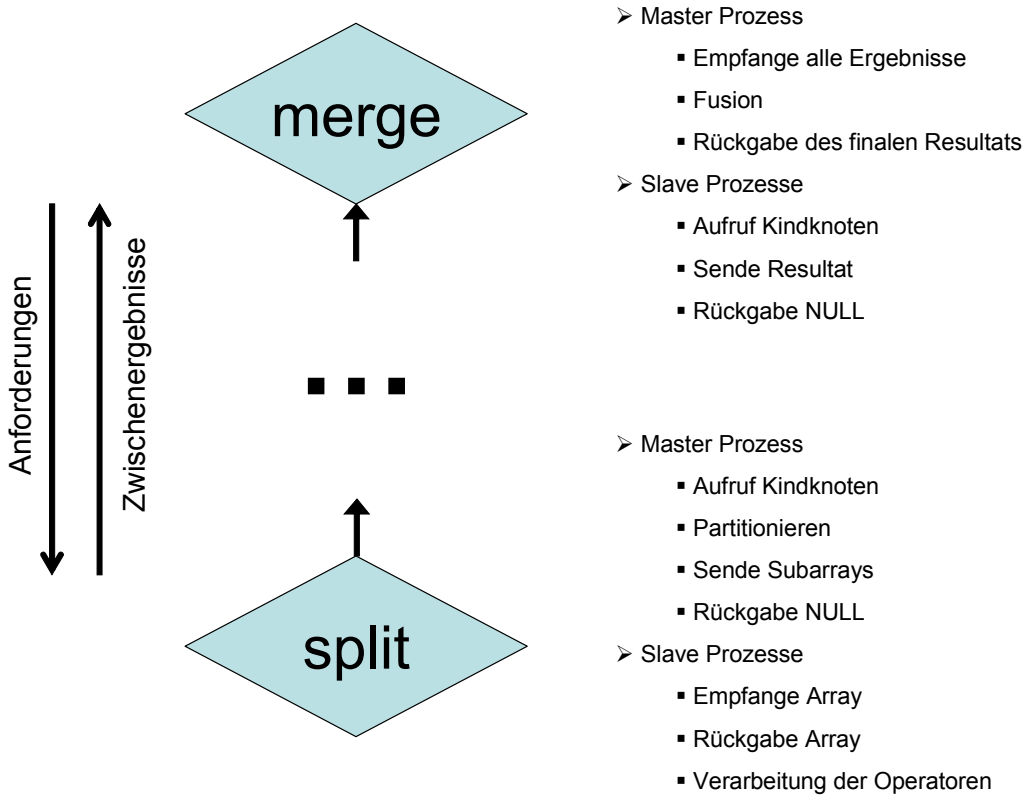


Abbildung 4.25: Funktionalität der Operatoren split und merge

Analog zur Inter-Objekt Parallelisierung, welche die Verteilung von Objekten und die Zusammenführung von Resultaten sowie die damit einhergehende Inter-Prozess Kommunikation in eigens dafür realisierten Iteratoren kapselt, wird die parallele Funktionalität auch für die Intra-Objekt Parallelität in neuen Operatorknoten umgesetzt. Um eine Verwechslung mit den Iteratoren *send* und *receive* zu vermeiden, werden diese Array Operatoren *split* (engl.: Aufteilen, Partitionieren) und *merge* (engl.: Zusammenführen, Fusion) genannt.

Diese Knoten werden in die Operatorbäume eingefügt, die das Prädikat einer Selektion σ oder die Funktion einer Applikation α bilden. Diese Operatoren verarbeiten Arrays und sind nicht als Iteratoren implementiert, das heißt ein Aufruf wird im Baum von oben nach unten durchgereicht (Methode *evaluate*) und das Resultat komplett berechnet, indem das Zwischenergebnis von unten nach oben gegeben werden.

Die an der Verarbeitung beteiligten Prozesse fallen wiederum in Klassen. Sie orientieren sich hierbei streng am Master (engl.: master, Meister) Slave (engl.: slave, Sklave) Paradigma [GLS99], wobei ein ausgezeichneter Master Prozess die Koordination übernimmt und die Arbeitslast in geeigneter Weise auf mehrere Slave Prozesse verteilt. Diese effektive Methode der Koordination von parallelen Prozessen kann eingesetzt werden, da zwischen den Slave Prozessen keinerlei Kommunikation erfolgen muss.

Die Operatoren und ihre Funktionalität sind in Abbildung 4.25 dargestellt, ihre Arbeitsweise wird zum besseren Verständnis für Master und Slave gesondert erklärt:

1. Master Prozess
   Im Operatorbaum wird von oben nach unten die Anforderung nach der Berechnung eines Resultats (Methode *evaluate*) gesendet. Der Master Prozess gibt diesen Methodenaufruf sowohl im *merge* wie auch im *split* an den entsprechenden Kindknoten weiter. *Split* erhält ein Array als Rückgabewert dieses Methodenaufrufs zurück (durch den Algorithmus zur Baumadaption gegeben), dieses wird unter Verwendung eines geeigneten Algorithmus partitioniert und die Teilobjekte an Slave Prozesse gesendet. Die Partitionierungsstrategie muss hierbei für alle *split* im Operatorbaum identisch sein, da die einzelnen Slave Prozesse Punktmengen verschiedener Arrays mit gleichen Koordinaten benötigen. Nach erfolgreichem Transfer wird durch die Rückgabe des leeren Resultats NULL die Anwendung der folgenden Operatoren unterdrückt. Ein *merge* stellt einen Puffer für den Empfang der partiellen Ergebnisse der Slave Prozesse zur Verfügung und kombiniert diese zu einem finalen Resultat. Eine gesonderte Behandlung erfordern unvorhergesehene Ausnahmen in der Programmausführung (engl.: exception) und Quantoren als Kindknoten. Eine Ausnahme resultiert in einer sofortigen Nachricht an alle Slave Prozesse mit der Aufforderung nach einem Abbruch der Verarbeitung. Für Quantoren können Teilergebnisse bereits das finale Resultat bestimmen, in diesem Fall wird die Forderung nach vorzeitigem Abbruch (engl.: early termination) an die Slave Prozesse gesandt und das Resultat an den Vaterknoten zurückgegeben (siehe auch Kapitel 4.4.5.2).

2. Slave Prozess
   Die Anforderung eines Resultats durch Aufruf der Methode *evaluate* auf dem jeweiligen Kindknoten wird in *merge* ausgeführt, jedoch nicht in *split*. Stattdessen reservieren die Prozesse einen Puffer für den Empfang eines Teilobjektes vom Master Prozess. Nach dem Empfang wird dieses Objekt an den aufrufenden Vaterknoten im Baum weitergegeben, die folgenden Array Operationen werden dann auf diesem Objekt ausgeführt. Erhält ein merge das Resultat des *evaluate* Aufrufs, wird dieses an den Master Prozess gesendet. Die Verarbeitung dieses Objekts ist damit für den Slave Prozess abgeschlossen (zumindest in dem gerade zu verarbeitenden Prädikatbaum).

Parallele Blöcke im Operatorbaum werden durch Operatoren gebildet, die sich in einem Pfad von einem *merge* bis zu einem *split* bzw. einem Blattknoten befinden. Im Folgenden sind immer maximale parallele Blöcke gemeint, das heißt, der Block endet erst mit dem *merge*, das sich am weitesten oben im Pfad befindet. Ziel der Baumadaption ist die Bildung solcher

maximaler parallelen Blöcke, da somit unnötige Transfers von Zwischenergebnissen verhindert werden. Die parallelen Blöcke selbst benötigen untereinander keine Kommunikation. Diese Tatsache folgt aus der Definition der parallelisierbaren Operationen. Einzige Ausnahme ist die parallele Ausführung von Quantoren. Bei einem vorzeitigen Abbruch werden die anderen Prozesse zur Vermeidung unnötiger Auslastung informiert. Das Ergebnis der Operation wird jedoch nicht beeinflusst.

Operation	Abk.	parallel	Eignung / Besonderheit
Elementar	*marray / condense*	nein	• Keine Abbildung, d.h. Bestimmung der Definitionsmenge unklar
Aggregation	*agg*	ja	• Hohe Komplexität, da Zelloperation • Sofortige finale Aggregationsfunktion
Induziert	*ind*	ja	• Hohe Komplexität, da Zelloperation • Innerhalb parallelen Blocks möglich
Konstanten	*const*	ja/nein	• Geringe Komplexität, immer Baumblatt • Darf in parallelem Block liegen[66]
Basistypprojektion	*dot*	nein/ja	• Geringe Komplexität, da kein Zellzugriff • Durch Optimierung immer Folgeoperation von *geom* oder *var*, folglich nicht parallelisiert
Geometrisch	*geom*	nein	• Geringe Komplexität, da kein Zellzugriff • Führt zu Störung der Lastbalancierung, deshalb in parallelem Block nicht erlaubt
Arithmetisch	*arith*	nein	• Geringe Komplexität, da keine „echte“ Array Operation • Soll nicht in parallelem Block liegen[67]
Array Variable	*var*	nein	• Geringe Komplexität, da kein Zellzugriff; Baumblatt

Tabelle 4.4: Liste der Knoten im Operatorbaum und Eignung für Parallelisierung

Die Array Operationen wurden bereits in Kapitel 4.4.1 auf ihre Eignung für Parallelisierung untersucht. Zur Vervollständigung der Liste der Knoten im Operatorbaum, die sich innerhalb eines parallelen Blocks befinden können, beziehen wir die Operationen *arith* (arithmetische Operation zwischen zwei skalaren Werten; Sonderfall einer induzierten Operation), *const* (Zugriff auf konstanten Wert aus der Anfrage) und *var* (Bindung an ein Array) in die Analyse ein. Die Verarbeitung von Konstanten *const* bildet hier einen Sonderfall: sie kann innerhalb eines parallelen Blocks geschehen, etwa als Eingabe einer unär induzierten Operation, aber auch außerhalb eines parallelen Blocks, etwa als Variable einer geometrischen Operation.

Das Einfügen der *split* und *merge* Knoten in den kompletten Anfragebaum erfolgt nach folgendem Algorithmus:

Algorithmus 4.2: Adaption des Anfragebaums für Intra-Objekt Parallelisierung

[66] Zur Vermeidung von Datentransfer

[67] Führt ansonsten zu weiterem Datentransfer und somit unnötiger Zusatzlast.

1. Parallelisiere den Prädikatbaum jeder Selektion σ und den Operatorbaum jeder Applikation α im Anfragebaum, das heißt, führe Punkt 2 für den Wurzeloperator jedes Iterators α und σ aus!

2. Prüfe den aktuellen Knoten:
   a. der Operator ist parallelisierbar: füge ein *merge* über dem aktuellen Knoten ein, gehe zu Punkt 3.
   b. der Operator ist nicht parallelisierbar: führe Punkt 2 für allen Kindknoten aus.

3. Ein *merge* wurde bereits eingefügt. Prüfe den aktuellen Knoten:
   a. der Operator ist eine Aggregation und parallelisierbar: setze ein *merge* und ein *split* über den Knoten. Führe Punkt 3 auf allen Kindknoten aus[68].
   b. der Operator ist parallelisierbar: führe Punkt 3 für allen Kindknoten aus.
   c. der Operator ist nicht parallelisierbar: füge ein *split* über dem aktuellen Knoten ein, wenn das Resultat der Operation ein Array ist. Führe Punkt 2 auf dem Knoten aus.

Der Algorithmus für die Adaption eines Operatorbaumes (Punkt 2 und 3) folgt prinzipiell der in Kapitel 4.4.2 vorgestellten Umformung des algebraischen Terms. Wir werden diesen Algorithmus anhand des Operatorbaums aus Beispiel 4.4 demonstrieren. Anschließend wird durch Analyse der Optimierungsregeln für Array Operationen aufgezeigt, dass dieser Algorithmus für optimierte Operatorbäume die teuren Operationen in einem Block einschließt und parallelisiert, bei gleichzeitiger Vermeidung einer ungleichen Arbeitslast und des Transfers großer Datenmengen.

Abbildung 4.26 zeigt die Adaption des Operatorbaumes aus Beispiel 4.4 durch *split* und *merge* Knoten. Die Traversierung des Baumes beginnt am Knoten *avg*, der die Aggregation eines Arrays zu einem Durchschnittswert realisiert. Da dieser Operator parallel ausgeführt werden kann, wird ein *merge* darüber eingefügt. Die Traversierung wird fortgesetzt, indem Algorithmus 4.2 rekursiv auf dem Kindknoten (mit der Information, dass *merge* bereits eingefügt wurde) aufgerufen wird. Die binäre Subtraktion ist ein parallel auszuführender Operator. Da *merge* bereits eingefügt wurde, liegt dieser Knoten in einem parallelisierbaren Block. Der Subtraktionsoperator ruft Algorithmus 4.2 folglich rekursiv für seine Kindknoten auf. Die Basistypprojektion *dot* und die geometrischen Operation *shift* fallen nicht in die parallel auszuführenden Operatoren, folglich wird ein *split* darüber eingefügt.

[68] Dieser Fall bedeutet eine mehrstufige Aggregation in einem Operatorbaum. Einerseits kann das auftreten, falls das Ergebnis einer Aggregation die skalare Eingabe für einen unär induzierten Operator ist. Eine weitere Möglichkeit bildet die Aggregation durch Skalierung *scale*, die ein Array als Ausgabe liefert und theoretisch eine weitere Aggregation nach sich ziehen kann.

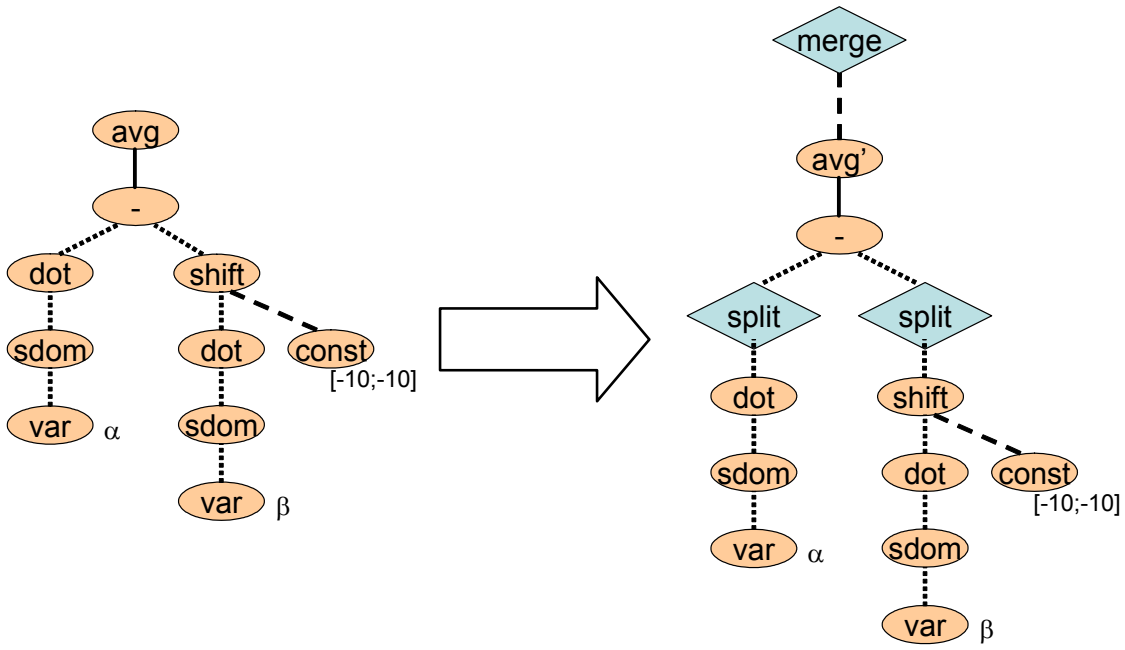


Abbildung 4.26: Adaption des Anfragebaumes für Intra-Objekt Parallelisierung

Dieser Algorithmus zur Parallelisierung führt für typische Array Anfragen zu hervorragend parallelisierbaren Anfragebäumen. Dazu sei auf die Optimierung von Array Operationen verwiesen (siehe [Rit99]) sowie auf typische Eigenheiten von Array Anfragen. Die optimierten Operatorbäume weisen folgende Eigenarten auf, die sich direkt auf die parallele Verarbeitung des Baumes auswirken:

- Geometrische Operationen werden soweit wie möglich nach unten verschoben. Insbesondere können die Operationen *trimm* und *section* unter induzierte Operationen geschoben werden. Bei aggregierenden Operationen sind geometrische Einschränkungen des Arrays zuvor sinnvoll, Einschränkungen nach einer Aggregation sind nicht sinnvoll und finden sich extrem selten.
- Aggregierende Operationen sind meist an der Wurzel der Operatorbäume zu finden. Eine Reduzierung des Datenvolumens als finale Operation vor der Übertragung eines Anfrageresultats zu einer Client Applikation ist in den meisten Fällen unumgänglich.
- Arrays werden erst für induzierte oder aggregierende Operationen geladen[69]. Andere Operationen beeinflussen lediglich Metadaten.

Als Ergebnis dieser Beobachtungen ergeben sich in der Praxis Anfragebäume, in welchen geometrische Operationen unten gefolgt werden von induzierten Operationen, die eine Datenanalyse realisieren. Abschließend finden sich weiter oben im Anfragebaum aggregierende Operatoren für eine Begrenzung des zu übertragenden Datenvolumens. Der vorgestellte Algorithmus minimiert das zwischen den Prozessen zu transferierende Datenvolumen, da das Partitionieren vor einem Laden der Zellinhalte, das Übertragen der Zwischenergebnisse möglichst nach einer Aggregation geschieht. Aggregierende Operationen, die logisch gesehen durch die Notwendigkeit einer speziellen finalen Aggregation gegenüber induzierten Operati-

[69] Eine Ausnahme ist die elementare Operation *condense*. Diese wird aber nicht parallelisiert und in unserer Betrachtung außer Acht gelassen.

onen im Nachteil sind, zeigen aufgrund der resultierenden Reduktion des Datentransfers hervorragendes Potential für einen linearen Speed-Up.

Heuristik 15 Parallelisierung von typischen Operatorbäumen
Der vorgestellte Algorithmus zur Parallelisierung von Array Operatorbäumen minimiert die Inter-Prozess Kommunikation für typische Array Anfragen. Die Partitionierung und der resultierende Transfer der Daten erfolgt meist rein auf Metadaten, Zellinhalte werden nicht übertragen. Die Fusion der Zwischenergebnisse und der dazu nötige Datentransfer werden (wenn möglich) mit aggregierten Resultaten realisiert.

4.4.4 Partitionierungsstrategien

Eine Partitionierung von multidimensionalen Arrays bedeutet theoretisch die Aufteilung einer Menge von Zellen (bestehend aus Vektor und Zellinhalt) in mehrere Teilmengen. Wie die Analyse der Operatoren ergeben hat, muss für unsere Zwecke diese Zerlegung disjunkt und komplett sein. In der Praxis kann ein multidimensionales Array aus folgenden Gründen nicht willkürlich in Mengen von Zellen aufgeteilt werden:

- Eine gute Partitionierungsstrategie beinhaltet nicht nur eine Lastbalancierung durch eine geeignete Aufteilung, sondern auch Vermeidung von E/A. Die Granularität der physischen Datenhaltung, in unserem Fall in Kacheln, erfordert unter Umständen, dass Daten mehrfach geladen (oder transferiert) werden. Das tritt auf, wenn Daten, die auf derselben physischen Speichereinheit liegen, von mehreren Prozessen verarbeitet werden. Dieses redundante Laden von identischen Quelldaten beeinträchtigt die Performanz.
- Durch die bestehende Implementierung der Datenverarbeitung sind Rahmenbedingungen für die parallele Verarbeitung zu beachten. Werden durch die Partitionierung wichtige Prämissen bisheriger (sequentieller) Verarbeitung verletzt, muss schlimmstenfalls eine komplette Neuentwicklung der Anfrageausführung in Betracht gezogen werden.

Grundsätzlich lassen sich drei verschiedene Strategien der Partitionierung von multidimensionalen Arrays unterscheiden. Jede Klasse ist hierbei als Spezialisierung der vorhergehenden Klasse zu sehen, wobei durch Integration des logischen und physikalischen Datenmodells die Partitionierung zusätzliche Eigenschaften erhält:

1. Allgemeine Partitionierung von multidimensionalen Arrays in Mengen von Zellen. Die Zerlegung ist disjunkt und komplett.
2. Logische Partitionierung in eine Menge von Arrays: Jede aus der Partitionierung resultierende Zellmenge bildet selbst wieder ein multidimensionales Array nach Definition.
3. Physikalische Partitionierung in eine Menge von Kacheln: zum einen eine Spezialisierung von 2, da jede Kachel nach Definition ein multidimensionales Array ist. Zusätzlich gilt jedoch, dass Zellen einer Kachel in derselben Partition liegen müssen:

$$(\underline{x}, v[\underline{x}]) \in \tau_i, (\underline{y}, v[\underline{y}]) \in \tau_j, (\underline{x}, v[\underline{x}]) \in P^k, (\underline{y}, v[\underline{y}]) \in P^l$$
$$\forall x, y : i = j \Rightarrow k = l$$

Abbildung 4.27 zeigt die drei Klassen anhand eines Beispiels, es werden drei Partitionen gebildet. Im ersten Fall ist die Zuteilung der Zellen willkürlich (und in diesem Beispiel noch

nicht komplett). Wird das logische Datenmodell als Grundlage für die Ergebnismenge herangezogen, können multidimensionale Arrays gebildet werden, grafisch gesehen multidimensionale Hyperquader, im 2-dimensionalen Fall also Rechtecke. Dient das physikalische Datenmodell als Basis der Zuteilung, ergeben sich Kachelmengen als Partitionen. Zellen einer Kachel werden hierbei immer in einer Partition untergebracht. Das Beispiel zeigt ein Array mit arbiträrer Kachelung, dessen Kachelmenge in drei Partitionen aufgeteilt wird.

Die willkürliche Zuteilung von Zellen zu den parallel zu verarbeitenden Partitionen verursacht die bereits erwähnten Probleme: E/A wird zum einen unnötig stark gefordert, zum anderen ist die Integration in die bestehende Anfrageverarbeitung praktisch unmöglich. Da die Anfrageverarbeitung typischerweise (auch in **rasdaman**) in Granularität von logischen oder physikalischen Objekten und nicht von Zellen erfolgt, lässt sich diese Partitionierung nicht in ein existierendes System integrieren und wird in der folgenden Analyse außer Acht gelassen. Wir werden somit die folgenden zwei Kapitel die Partitionierung gemäß physikalischen und logischen Datenmodells, sowie Vor- und Nachteile bezüglich der Integration, analysieren.

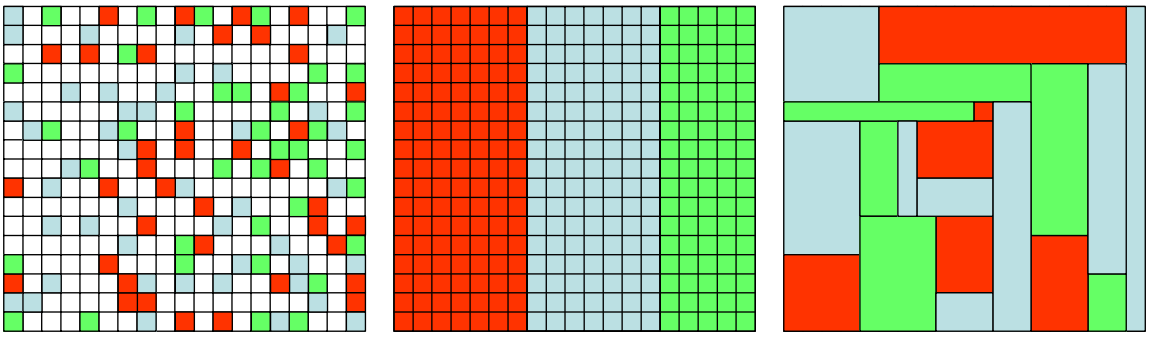


Abbildung 4.27: Partitionierung von multidimensionalen Arrays

Tabelle 4.5 zeigt eine Übersicht der Partitionierungsstrategien unter Beachtung des Datenflusses zwischen Array Operatoren. Der Datenfluss, das heißt, die Datenübergabe von Arrays zwischen Array Operatoren, kann durch zwei Konzepte implementiert werden:

1. Iteratorkonzept mit Übergabe einzelner Kacheln. Mittels eines Iterators werden alle Kacheln nacheinander verarbeitet. Vor allem der Speicherbedarf wird gemindert, da Speicher für Zwischenresultate zu einem sehr frühen Zeitpunkt wieder freigegeben werden kann. Durch die Anzahl der Kacheln in einem typischen MDD und die damit verbundenen Methodenaufrufe wird jedoch die Performanz gemindert. Ferner zerstören Operationen, welche einer nicht injektiven Funktion mit Array als Resultat entsprechen[70], dieses Konzept wieder. Das heißt, es müssen so lange Kacheln angefordert werden, bis alle zur Berechnung nötigen Zellen vorliegen. Aus diesen Gründen wird diese Strategie eines zweiten Iteratorkonzepts mit kleinerer Granularität in **rasdaman** nicht verfolgt.

[70] Vor allem *marray* in Verbindung mit einer Berechnungsfunktion, die durch *condense* realisiert wird und Formatkonvertierungen sind injektive Abbildungen, für die nur sehr schwer Zwischenergebnisse aus partiellen Eingaben gebildet werden können.

2. Übergabe von kompletten Arrays zwischen den Array Operatoren. Dies geschieht in rasdaman durch den Datentyp MDD, der Verweise auf die entsprechenden Kacheln beinhaltet. Der primäre Vorteil ist die Performanz, der Nachteil besteht in einem erhöhten Ressourcenbedarf.

Die folgenden Kapitel 4.4.4.1 und 4.4.4.2 diskutieren diese zwei Partitionierungsstrategien sowie Vor- und Nachteile im Detail. In Kapitel 4.4.4.3 wird abschließend auf Algorithmen zur Partitionierung mehrerer Arrays mit unterschiedlicher Kachelung eingegangen. Dieser Sonderfall tritt insbesondere bei einem oder mehreren binär induzierten Operatoren in einer Anfrage auf.

4.4.4.1 Physikalische Partitionierung in eine Menge von Kacheln

Eine Partitionierung der parallel zu verarbeitenden Datenmenge gemäß physikalischer Speicherung ist eine hervorragende Strategie, da so in den meisten Fällen E/A, genauer das Laden des gleichen physikalischen Datenobjekts von mehreren Prozessen, vermieden werden kann. In relationalen DBMS werden in der Regel Seiten (engl.: pages) als physikalische Speichereinheit den einzelnen Prozessen zugeteilt. Die Verarbeitung von Relationen mittels Iteratorkonzept sichert darüber hinaus eine Lastbalancierung. Für Arrays gestaltet sich die Datenzuteilung etwas komplexer: je nachdem, welche Mechanismen für die Verarbeitung von Arrays und deren Übergabe im DBMS gewählt wird, kann eine parallele Verarbeitung und damit zusammenhängend eine Partitionierung der Arrays, mehr oder weniger einfach integriert werden.

Durch die Nutzung eines **zweiten Iteratorkonzepts** auf Basis von Kacheln wird die Verteilung der Daten an einen Iterator geknüpft. Jeder Prozess holt sich per *next* Aufruf dynamisch die nächste Kachel. In diesem Fall können die *split* und *merge* Operatoren wie in Kapitel 4.3.4 beschrieben in die Anfrageausführung integriert werden. Bezüglich des Ressourcenbedarfs und der paralleler Verarbeitung hat diese Implementierung die bekannten Vorteile des Iteratorkonzepts: Speicher für Zwischenresultate kann früher freigegeben werden, eine gute parallele Lastbalancierung wird durch eine dynamische Verteilung an parallele Instanzen gewährleistet. Jedoch leidet die Gesamtperformanz durch die Größe der zu iterierenden Menge, also der Kachelmenge, und der damit verbundenen häufigen Aufrufe der *next* Funktion. Die Größe der Kacheln unterliegt auch keinen Regeln wie in RDBMS (Plattenseiten), wird eine kleine Kachelung gewählt, sinkt bei Nutzung eines Iteratorkonzepts die Performanz spürbar. Ferner werden bei binär induzierten Operationen auf Arrays mit unterschiedlicher Kachlung deutliche Performanzeinbussen spürbar. Dies wird detailliert in Kapitel 4.4.4.3 diskutiert.

Für die Implementierung der parallelen Verarbeitung von Arrays in rasdaman wurde dieses Konzept nicht verfolgt. Primärer Grund ist der zu erwartende Verlust an Gesamtperformanz, der durch eine parallele Verarbeitung zwar mehr als aufgefangen würde, aber einen linearen Speed-Up nicht zulässt. Dieser und die weiteren erwähnten Nachteile sprechen klar gegen ein zweites Iteratorkonzept, welches nebenbei bemerkt einer kompletten Reimplementierung der Anfrageverarbeitung in rasdaman bedürfte.

		Datenfluss zwischen Operatoren	
		Kachel Iterator	**Array (rasdaman)**
Partitionierungsstrategie	**Kachelmengen (4.4.4.1)**	**Parallelisierung auf Basis eines zweiten Iteratorkonzepts** • Adaption der gesamten Anfrageausführung nötig (rasdaman) • Meist sehr gute parallele Performanz (Lastverteilung) • Gute Ressourcenverwaltung • Einbussen bei Gesamtperformanz • Probleme mit binär induzierten Operationen auf Arrays mit unterschiedlicher Kachelungsstrategie (4.4.4.3)	**Übergabe von teilweise besetzten Arrays** • Adaption der Array Operatoren und deren Fehlerbehandlung nötig (rasdaman) • Meist sehr gute parallele Performanz • Probleme mit binär induzierten Operationen auf Arrays mit unterschiedlicher Kachelungsstrategie (4.4.4.3)
	Subarrays (4.4.4.2)	**Nicht sinnvoll**	**Naive Partitionierung in Subarrays** • Sehr gute Integrierbarkeit in bestehende Anfrageausführung (rasdaman) • Hervorragende Lastbalancierung • Schlechte parallele Performanz • Pathologische Fälle mit Einbrüchen der Performanz durch große Redundanz bei E/A
			Partitionierung in Subarrays mit Analyse der Kachelung • Relativ gute Integrierbarkeit in bestehende Anfrageausführung (rasdaman) • Meist sehr gute parallele Performanz • Hervorragende Lastbalancierung • Meist sehr gute Resultate bei binär induzierten Operationen (4.4.4.3) • Geringe Redundanz bei E/A

Tabelle 4.5: Übersicht der Partitionierungsstrategien unter Beachtung des Datenflusses zwischen Array Operatoren

Die Partitionierung in Kachelmengen bei einer Implementierung der Datenübergabe als Arrays zwischen Operatoren lässt sich durch **teilweise besetzte Arrays** realisieren. Die Partitionen bestehen hierbei aus Datenobjekten des Typs MDD. Dieser Datentyp beinhaltet eine Menge der Kacheln, die das Array ausfüllen. Füllt die Menge der Kacheln das MDD nicht komplett aus, ist es nach unserer Definition kein gültiges Array. Nichtsdestotrotz können die meisten Array Operationen auf einem solchen unvollständigen Datenobjekt korrekte Zwischenergebnisse liefern[71]. Abbildung 4.28 zeigt eine Partitionierung in drei Kachelmengen anhand eines Beispiels.

Der Vorteil dieser Partitionierungsstrategie besteht in einer guten parallelen Performanz, da zumindest für unäre parallele Operationen keinerlei Daten von verschiedenen Prozessen redundant geladen werden. Für binär induzierte Operationen gilt das jedoch nicht: besitzen die Eingabearrays eine unterschiedliche Kachelung, kann zusätzliche E/A nicht vermieden werden und nur mit sehr großem Aufwand minimiert werden. Eine Analyse darüber folgt in Kapitel 4.4.4.3.

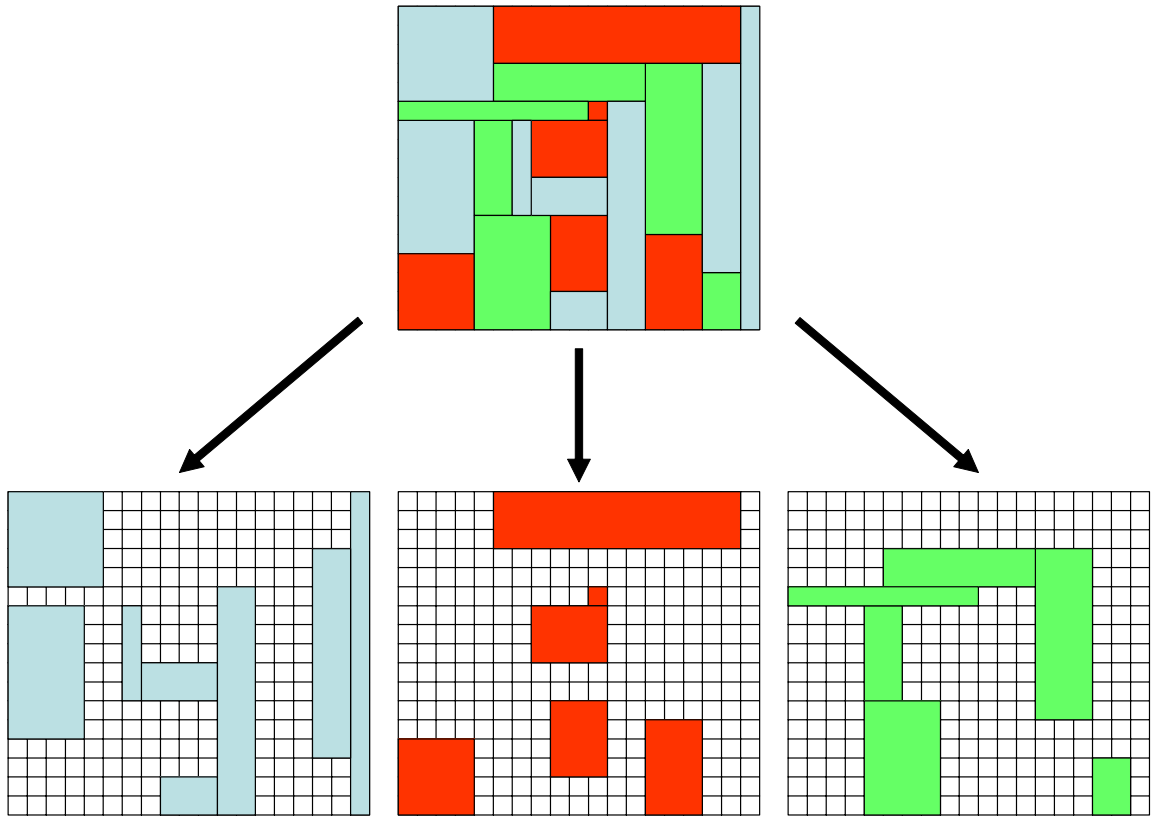


Abbildung 4.28: Partitionierung in Kachelmengen bei Übergabe von Arrays

Dieses Verfahren wurde in rasdaman nicht implementiert. Ein in Kapitel 4.4.4.2 vorgestellter Algorithmus zur Partitionierung zeigt eine fast identische Performanz für die parallele Verarbeitung bei gleichzeitiger Vermeidung der Schwächen bei binär induzierten Operationen. Des Weiteren müssten Array Operationen eine Verarbeitung von teilweise besetzten Arrays er-

[71] Genauer gesagt, können insbesondere Array Operationen, die laut Analyse parallel auf einer Partition verarbeitet werden können, auch auf einem unvollständigen Array ausgeführt werden.

möglichen, was in rasdaman nicht der Fall ist[72]. Eine solche umfangreiche Reimplementierung der Anfrageverarbeitung von rasdaman ist also nicht sinnvoll.

Zusammenfassend ist festzustellen, dass eine Partitionierung von Arrays allein auf Basis der physikalischen Datenhaltung Probleme bereitet: insbesondere binär induzierte Operationen auf Objekten mit verschiedener Kachelung bereiten Probleme. Außerdem ist die Integration in die bestehende Anfrageverarbeitung von rasdaman unmöglich.

4.4.4.2 Logische Partitionierung in Arrays

Eine Aufteilung eines multidimensionalen Arrays in p Partitionen, die jeweils wieder ein Array nach Definition ergeben, muss als Basis für eine gute parallele Verarbeitung folgende (konkurrierende) Zielfunktionen bedienen:

1. Optimierung der Lastbalancierung
   Die Lastbalancierung muss bereits durch die Partitionierung sichergestellt sein, da eine dynamische Lastverteilung wie etwa bei Iteratoren nicht automatisch erfolgt. Die Anzahl der Zellen in den Subarrays muss hierzu möglichst gleichmäßig sein. Da Operationen auf den Zellen eine identische Komplexität aufzeigen, wird unter Vernachlässigung externer Einflüsse die Verarbeitungszeit durch die Partition mit maximaler Größe bestimmt. Ein einfaches Gütekriterium für die Partitionierung ist also: das Maximum der Zellenanzahl der Subarrays muss möglichst klein sein.
2. Minimierung von E/A
   Berücksichtigt man für die Partitionierung die physikalische Datenhaltung, können Ladevorgänge minimiert werden. Im schlimmsten Fall schneidet jede Partition alle Kacheln eines multidimensionalen Arrays, das heißt jeder Prozess muss das komplette Objekt laden. Bei einer aus dieser Sicht optimalen Partitionierung deckt sich jede Partitionsgrenze mit den Kachelgrenzen. In diesem Fall lädt jeder Prozess ausschließlich die von ihm benötigten Daten.

Die Analyse aller möglichen Partitionierungen eines multidimensionalen Arrays unter Berücksichtigung dieser zwei meist konkurrierender Parameter ist ein Sonderfall eines klassischen Problems der theoretischen Informatik, nämlich BIN PACKING und ist somit NP-vollständig. Mit anderen Worten ist im Allgemeinen eine Lösung dieses Problems nicht effizient berechenbar. Wir präsentieren ein heuristisches Verfahren, welches ein Array immer in einer ausgewählten Dimension partitioniert und in der Praxis immer eine gute Lösung mit hervorragender Effizienz liefert.

Abbildung 4.29 skizziert die Partitionierung entlang einer Dimension (links) im Gegensatz zu einer „komplizierten“ Aufteilung des Arrays (rechts). Der vorgestellte Algorithmus partitioniert ein Array gemäß der ersten Art. Der Grund für diese Einschränkung liegt in Erfahrungen des Projektes ESTEDI: in der Praxis wird meist eine ausgerichtete Kachelungsstrategie gewählt. Dies resultiert aus der Tatsache, dass gewisse Dimensionen häufiger durch geometrische Operationen eingeschränkt werden[73]. Wie später gezeigt wird, führt die Reduktion von E/A bei ausgerichteter Kachelung meist zu einer Partitionierung entlang einer Dimension.

[72] Eine Array Operation ausgeführt auf ein teilweise besetztes MDD führt in rasdaman in der Regel zu Ausnahmebehandlungen und zu einem Abbruch der Anfrage.

[73] So wird beispielsweise in einer Zeitreihe von Satellitenbildern häufig ein bestimmter Zeitpunkt selektiert. Die Selektion eines ausgewählten Höhen- oder Breitengrades ist dagegen nur für sehr spezielle Analysen von Bedeutung. Eine Kachelung mit geringer Ausdehnung in der Zeitdimension reduziert somit den „Verschnitt“ und damit E/A für typische Anfragen und führt zu einer Verbesserung der Performanz.

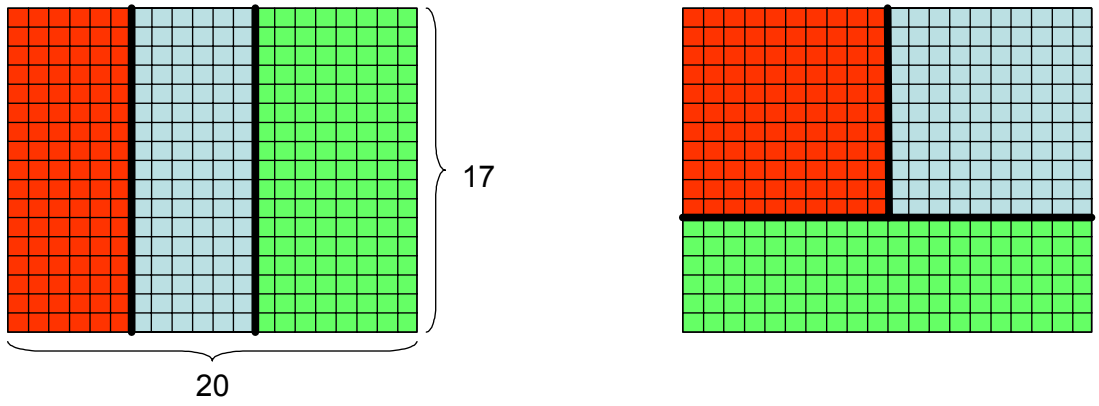


Abbildung 4.29: Partitionierung in Subarrays

Der folgende Algorithmus erzeugt in jeder Dimension eine Partitionierung mit guter Lastverteilung und analysiert diese bezüglich des Gütekriteriums Minimierung von E/A:

Algorithmus 4.3: Logische Partitionierung eines multidimensionalen Arrays unter Sicherung von Lastbalancierung und Minimierung von E/A

1. Analysiere die Partitionierung entlang jeder Dimensionen!

   a. Optimierung der Lastbalancierung
   Bilde p Subarrays δ^i (i = 1, ..., p) aus dem Array α. Die Ausdehnung in der zur Partitionierung gewählten Dimension d ist:

      i. Das Array lässt sich ohne Rest in p Partitionen teilen:
      $$extent(\alpha[d]) \bmod p = 0 \Rightarrow extent(\delta^i[d]) = \frac{extent(\alpha[d])}{p},\ i = 1,\ldots,p$$

      ii. Die Partitionierung ist nur ungleich möglich. Eine optimale Lastbalancierung für p-1 Prozesse wird durch folgende Partitionierung erreicht:
      $$extent(\alpha[d]) \bmod p \neq 0 \Rightarrow extent(\delta^i[d]) = \left\lfloor \frac{extent(\alpha[d])}{p} \right\rfloor + 1,\ i = 1,\ldots,p\text{-}1.$$
      Die letzte Partition[74] erhält den Rest, für welchen gilt:
      $$\left\lfloor \frac{extent(\alpha[d])}{p} \right\rfloor - (p-1) < extent(\delta^i[d]) < \left\lfloor \frac{extent(\alpha[d])}{p} \right\rfloor, i = p\,.$$

   In allen anderen Dimensionen ist die Ausdehnung identisch zu α. Merke die Anzahl der Zellen in der größten Partition.

[74] Theoretisch können mehrere Partitionen leer sein. Dies tritt auf, wenn extent(δ^i [d])*(p-1) ≥ extent(α[d]). In der Praxis ist dieser Fall extrem selten, da typischerweise extent(α[d]) >> p. Folglich lassen wir diesen Sonderfall außer Acht.

b. Optimierung von E/A:
Analysiere die Zellenanzahl der durch diese Partitionierung zu ladenden Kacheln! Die zu ladenden Kacheln werden durch eine Methode *intersect* ermittelt, welche die Kacheln ermittelt, die Daten im Schnittbereich zweier Arrays (jeweils die Partitionen δ^i mit Array α) beinhaltet.

2. Wähle die Dimension mit den geringsten Ladevolumen! Die in Punkt 1 erzeugten Partitionierungen zeigen nur geringe Abweichungen bezüglich Lastbalancierung, E/A kann sich jedoch maßgeblich unterscheiden. Sind zwei Dimensionen bezüglich E/A identisch, wähle aus diesen die Partitionierung mit optimaler Lastbalancierung!

Dieser Algorithmus erzeugt zwar praktisch immer eine „gute" Partitionierung der Quelldaten, jedoch wird nicht immer die „optimale" Partitionierung erreicht:

- Die Partitionierung wird entlang einer Dimension vollzogen. Dies erzeugt eine Menge von Subarrays, die das Array analog zu einer ausgerichteten Kachelung auskleiden, wobei in der gewählten Dimension die Ausdehnung der Subarrays der Ausdehnung des Arrays entspricht. Theoretisch können arbiträre Partitionierungen in manchen Fällen bessere Resultate ergeben, sowohl für die Lastverteilung, als auch bezüglich E/A Minimierung. Die Analyse aller möglichen Partitionierungen eines multidimensionalen Arrays bezüglich dieser zwei Kriterien ist jedoch für große Eingabemengen nicht effizient durchführbar.
- Der vorgestellte Algorithmus analysiert auch nur eine Teilmenge der möglichen Partitionierungen entlang einer Dimension. Obwohl in typischen Szenarien praktisch immer eine sehr gute Partitionierung erreicht wird, kann in manchen Fällen bei ausgerichteter und regulärer Kachelung durch Akzeptanz von leichten Lastungleichheiten ein minimiertes E/A Ladevolumen und somit eine bessere Performanz erreicht werden. Einfach ausgedrückt: verschiebt man die Partitionsgrenze auf eine Kachelgrenze, wird eine ungleiche Last aber auch eine Reduktion von E/A erreicht. Der gewählte Algorithmus deckt diese Sonderfälle nicht ab, da sich in der Praxis für typische Arrays und deren Kachelungsmethoden gezeigt hat, dass eine geringe Redundanz bezüglich E/A keine signifikanten Auswirkungen auf die Performanz hat.

Näher betrachtet ist das Problem der optimalen Partitionierung eines multidimensionalen Arrays ein typisches Optimierungsproblem, welches in der Praxis nur durch bekannte Methoden wie Anwendung von Heuristiken oder Pruning (engl.: pruning, Abschneiden) des Suchraumes effizient gelöst werden kann. In diesem Zusammenhang kann der vorgestellte Algorithmus als Heuristik gesehen werden.

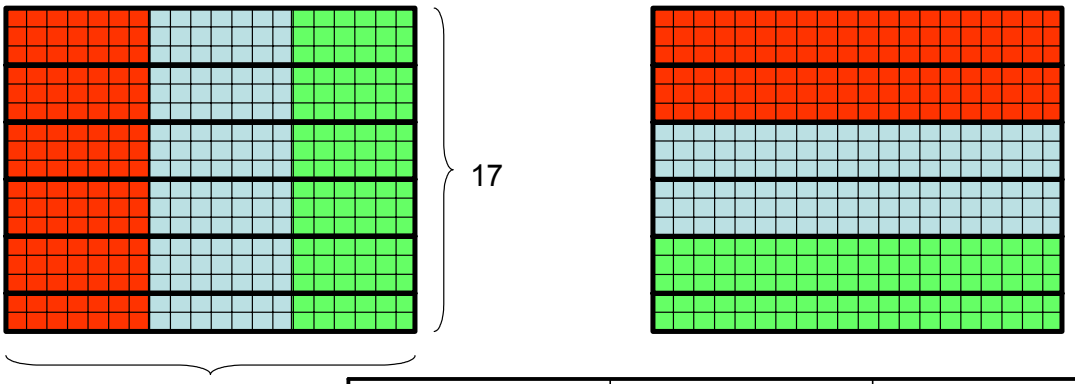
Ein einfaches Beispiel (auf Basis von Abbildung 4.29) soll den Algorithmus veranschaulichen.

Beispiel 4.5: Partitionierung eines Arrays in Subarrays

Eine optimale Lastbalancierung liegt dann vor, wenn alle gebildeten Partitionen genau die gleiche Anzahl an Zellen beinhalten. Dies kann nur erreicht werden, wenn die Ausdehnung der gewählten Dimension durch die Partitionsanzahl ohne Rest dividiert werden kann. Hat etwa eine Dimension die Ausdehnung 200 und soll in 4 Partitionen geteilt werden, hat jede Partition eine Ausdehnung von 50. Ist die Dimension nicht ohne Rest in p Partitionen zu teilen, muss für mindestens eine Partition gelten, das sie eine größere Ausdehnung hat, als der (abgerundete) Quotient.

In Abbildung 4.30 hat die erste Dimension eine Ausdehnung von 20 und kann nicht restfrei auf 3 Partitionen verteilt werden, somit muss mindestens ein Prozess ein Array mit einer Ausdehnung größer 6 bewältigen. Die gesamte Laufzeit hängt von der maximalen Partition ab. Die Bildung von Partitionen mit einer Ausdehnung von 7 mit Zuteilung einer kleineren Restpartition an den letzten Prozess sichert eine Laufzeit, die nicht über dem theoretischen Optimum liegt. Die Partitionierung in drei Bereiche, die aus Punkt 1 des Algorithmus 4.3 für die zwei Dimensionen resultiert, ist in Abbildung 4.30 farbig gekennzeichnet.

In der Praxis hat eine Optimierung von E/A einen großen Einfluss auf die Gesamtperformanz. Die durch dieses Verfahren gebildeten Partitionierungen werden folglich auf die damit zu ladenden Datenvolumina geprüft. Da die Partitionen an unabhängige Prozesse zur parallelen Verarbeitung gegeben werden und somit von diesen teilweise redundant geladen werden, ist das zu Ladevolumen in der Regel größer als die Quell-



	Partitionierung Dimension 1	Partitionierung Dimension 2
Lastbalancierung	max(119, 119, 102) → 119 Zellen	max(120,120,100) → 120 Zellen
Ladevolumen	340+340+340 → 1020 Zellen	120+120+100 → 340 Zellen

Abbildung 4.30: Optimierung von E/A bei Partitionierung

Abbildung 4.30 zeigt dies anhand eines Arrays mit besonders extrem ausgerichteter Kachelung mit extent(τ) = [20, 3], welche die komplette Ausdehnung der ersten Dimension

in jeweils eine Kachel einschließt (die Kachelgrenzen sind in der Abbildung als dicke Linien markiert). Bezüglich Lastbalancierung bringt die Partitionierung in Dimension 1 geringe Vorteile, die jedoch durch die zu ladenden Datenvolumina mehr als wettgemacht werden: Die zu ladenden Datenbereiche sind:

$intersect(\alpha[0:6,0:16], \alpha) + intersect(\alpha[7:13,0:16], \alpha) + intersect(\alpha[14:19,0:16], \alpha) = \alpha[0:16,0:19] + \alpha[0:16,0:19] + \alpha[0:16,0:19] = 3\alpha$

Die Partitionierung entlang Dimension 1 resultiert also darin, dass jeder Prozess die komplette Datenbasis des Arrays laden muss. Der beschriebene Algorithmus wählt hier eine Partitionierung entlang der zweiten Dimension aufgrund des geringeren Ladevolumens:

$intersect(\alpha[0:19,0:5], \alpha) + intersect(\alpha[0:19,6:11], \alpha) + intersect(\alpha[0:19,11:16], \alpha) = \alpha[0:19,0:5] + \alpha[0:19,6:11] + \alpha[0:19,11:16] = \alpha$

Diese zwei Fälle stellen die obere und untere Schranke des zu ladenden Datenvolumens dar. In der Praxis sind die Unterschiede selten so deutlich, dennoch sind Differenzen in der Performanz meist deutlich auszumachen.

Die beschriebene Partitionierung von Arrays zum Zwecke einer parallelen Verarbeitung führt logisch gesehen zu Bereichsanfragen, die jeder beteiligte Prozess ausführt. Eine Analyse hat ergeben, dass Bereichsanfragen von **rasdaman** nicht optimal unterstützt werden, was vor allem durch fehlendes multidimensionales Clustering der Kacheln und die Verarbeitungsstrategie verursacht wird.

1. Clustering der Kacheln
   Der in **rasdaman** implementierte multidimensionale Index eines R^+-Baumes ist nicht clusternd, das heißt, eine Nachbarschaft der Kacheln im multidimensionalen Raum wird für die Speicherung auf Platte nicht beachtet. Die Reihenfolge der Speicherung wird stattdessen durch die Einfügereihenfolge bestimmt, der Index dient lediglich zum Lokalisieren von Kacheln (und damit BLOBs), die einen angefragten Bereich schneiden.

2. Verarbeitungsstrategie eines alternierenden Ladens und Verarbeitens von Kacheln
   Die Verarbeitung einer Anfrage in **rasdaman** erfolgt durch Iteration über die Menge der zu verarbeitenden Kacheln. Ein Ladevorgang erfolgt immer dann, wenn die Verarbeitung (z. B. Array Operation) einer Kachel abgeschlossen ist. Eine Bereichsanfrage wird somit nicht als eine logische E/A Operation gesehen, vielmehr wird jeweils eine zu verarbeitende Kachel durch Index lokalisiert und als eine Anforderung an das RDBMS weitergegeben.

Diese Punkte führen bereits bei nicht-paralleler Verarbeitung zu einer suboptimalen Performanz von Bereichsanfragen. Messungen haben gezeigt, dass für typische Festplatten die Leserate bei Bereichsanfragen um den Faktor 6 bis 8 langsamer ist als ein sequentielles Lesen[75].

[75] Der Faktor ist stark abhängig von der Größe der Kacheln. BLOBs können in der Regel sequentiell gelesen werden, somit führen große Kacheln zu einer Verbesserung der Leserate.

Wird der Ladeauftrag zusammengefasst und an das RDBMS gegeben, das heißt, wird eine nicht alternierende Verarbeitung simuliert, reduziert sich dieser Faktor auf etwa 4[76].

Bei paralleler Verarbeitung führen einerseits die oben beschriebenen Punkte, andererseits auch die alternierende Anforderung der Kacheln von verschiedenen Prozessen zu einem zufälligen Zugriffsmuster. Diese zufälligen Zugriffsmuster resultieren in einem potentiellen Flaschenhals bezüglich E/A bei paralleler Ausführung.

Eine Optimierung von E/A bei paralleler Verarbeitung bedarf eines Clustering und einer Anpassung der Verarbeitungsstrategie. Das Clustering kann durch Linearisierung des multidimensionalen Raumes mittels einer raumfüllenden Kurve realisiert werden. Der multidimensionale Index eines UB-Baums [Bay97] nutzt hierfür eine Z-Kurve, eine andere Möglichkeit ist die Hilbert-Kurve. Im Gegensatz zum UB-Baum werden bei unserem Ansatz jedoch nicht Datenbankseiten, sondern Kacheln referenziert. Das Ausnutzen des Clusterings muss weiterhin dadurch geschehen, dass eine Bereichsanfrage als eine logische E/A Operation realisiert wird.

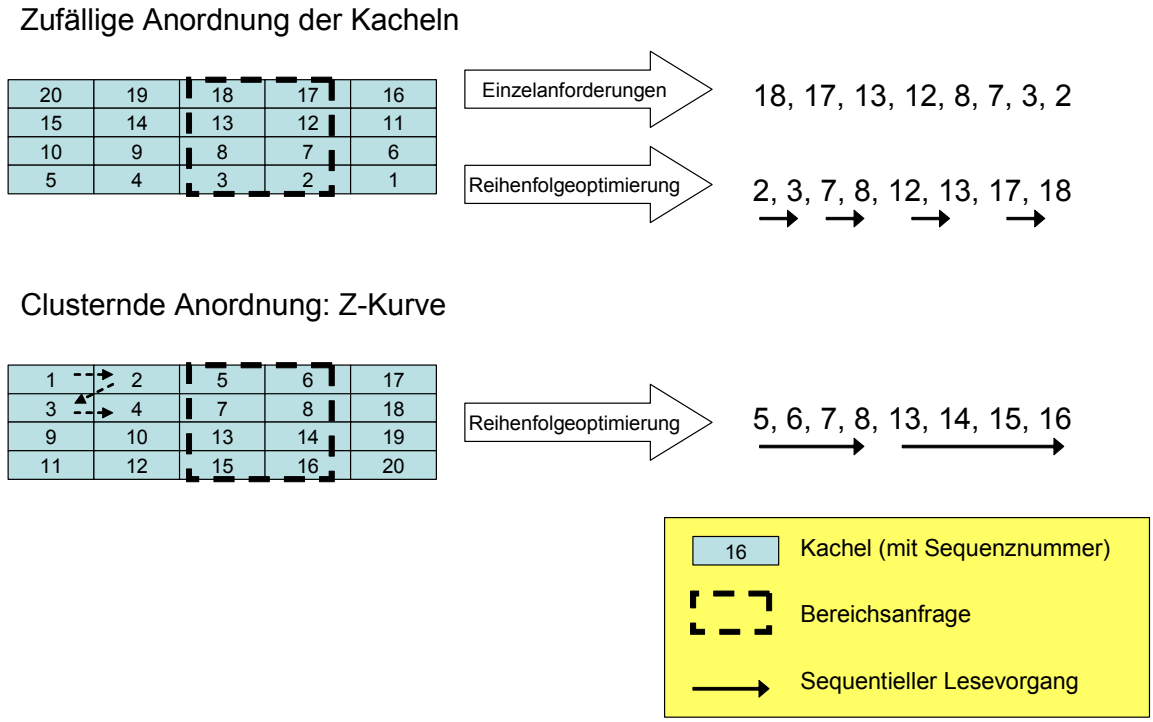


Abbildung 4.31: Optimierung von E/A bei paralleler Verarbeitung durch Clustering mittels Z-Kurve

Abbildung 4.31 skizziert den Ladevorgang für eine Bereichsanfrage. Im Beispiel oben wurden die Kacheln in der Reihenfolge des Einfügens auf Platte geschrieben. Ein alternierendes Laden und Verarbeiten der Kacheln im angefragten Bereich (wie in rasdaman implementiert) führt zu Einzelanfragen und führt in allen Fällen zu einer komplett nicht-sequentiellen Reihenfolge der Lesezugriffe. Die Methode einer einzelnen logischen E/A Operation mit Opti-

[76] Eine Zusammenfassung der Lesevorgänge hat keine negativen Auswirkungen auf Array Operationen mit frühzeitigen Abbruch (Quantoren). Die Auswertung von Quantoren erfolgt, indem alle Daten komplett gelesen und auf das Prädikat getestet werden, was in einem Array mit booleschem Datentyp resultiert. Der frühzeitige Abbruch kann dann nur während der Analyse dieses Arrays vollzogen werden, die Daten werden also in jedem Fall komplett gelesen.

mierung der Zugriffsreihenfolge (ohne Clustering der Kacheln) wird darunter skizziert. Jedoch erlaubt auch dies nur kurze sequentielle Ketten, die überdies mit steigender Dimensionalität der Basisdaten immer unwahrscheinlicher werden. In der Abbildung unten ist eine durch die Z-Ordnung gegebene Kachelreihenfolge skizziert. Bereichanfragen führen zu Ketten von zusammenhängenden und damit sequentiell zu lesenden Kacheln.

Die Implementierung dieser Optimierung kann nur mit großem Aufwand realisiert werden, indem beim Einfügen der Arrays eine Z-Ordnung der Kacheln auf einen Primärschlüssel der Tabelle im RDBMS abgebildet wird. Ferner muss die Anfrageverarbeitung von **rasdaman** komplett überarbeitet werden. Bereichsanfragen sind als eine Einzelanforderung an das RDBMS zu geben, damit die multidimensionale Ordnung für ein sequentielles Lesen genutzt werden kann.

Wir haben bisher einen Algorithmus zur Partitionierung eines Arrays präsentiert, der Lastbalancierung sichert und E/A minimiert. Dieser Algorithmus erzeugt nicht in jedem Fall eine optimale Partitionierung. Wir werden im Folgenden zeigen, dass er jedoch für die meisten Fälle eine sehr gute Partitionierung erzeugt. Zu diesem Zweck werden wir den Algorithmus bezüglich typischer Kachelungsmethoden der Eingabearrays analysieren.

Die im Projekt ESTEDI durch das Array DBMS **rasdaman** verwalteten multidimensionalen Rasterdaten zeigen bezüglich der Kachelung folgende drei Ausprägungen:

1. In sehr wenigen Fällen wurde eine reguläre Kachelung eingesetzt. Eine identische Ausdehnung der Kacheln in jeder Dimension wird genutzt, wenn die zu erwartenden Anfragen keine Präferenzen bezüglich Dimension zeigen. Im Projekt ESTEDI war das für 2-dimensionale Satellitenbilder und 3-dimensionale (räumliche) Simulationen der Fall, die nicht als Zeitreihen modelliert waren.
2. In der Praxis wurde fast ausschließlich eine ausgerichtete Kachelungsstrategie eingesetzt. Bei den meisten Analyseszenarien war die Fixierung einer Dimension eine primär zu unterstützende Anfrage. So ist etwa in räumlichen Daten zu Klimasimulationen einer Einschränkung der Höhenstufe typischer als die Fixierung parallele zu Langen- oder Breitengrad. Darüber hinaus liegen fast alle Daten als Zeitreihen vor, da die Analysefähigkeiten eines Array DBMS (im Gegensatz zu bisherigen Systemen) dynamische Zeitreihenanalysen ermöglichen. Die Modellierung als Zeitdimension soll aber (wie erwähnt) nicht dazu führen, dass „bisher übliche“ Anfragen, das heißt die einfache Extraktion eines Datensatzes bezüglich eines fixen Zeitstempels, spürbare Performanzeinbussen zeigen. Folglich wird die Kachelung für die Zeitdimension sehr klein gewählt. Eine Ausdehnung von 1 ermöglicht hier die beste Performanz für eine *section* Operation in der Zeitdimension ohne Einschränkung der Domäne in allen anderen Dimensionen. Wird eine bessere Performanz für Zeitreihenanalysen gewünscht, wird die Zeitdimension mit einer Ausdehnung von 2 bis 4 gekachelt. Eine gröbere Kachelung in dieser Dimension resultiert in spürbar schlechterer Performanz in den „üblichen“ Anfragen. Dies wird von den Benutzern meist nicht toleriert.
3. Eine arbiträre Kachelungsstrategie ist zu kompliziert, erfordert eine genaue Analyse der Daten und eine Adaption der Einfügeprogramme und wurde in ESTEDI von keinem Partner verwendet.

Die Betrachtung unseres Algorithmus bezüglich Effizienz für typische Kachelungsstrategien muss sich somit auf die am häufigsten eingesetzte ausgerichtete Kachelung konzentrieren. Der Anteil der redundant geladenen Daten ist entweder null, falls Partitionsgrenzen und Kachelgrenzen zusammenfallen oder lässt sich durch folgende Formel berechnen:

$$\text{Redundanz } \Delta_{E/A} = \frac{(p-1)\text{extent}(\tau[i])}{\text{extent}(\zeta[i])}$$

$$\text{extent}(\zeta[i]) \gg \text{extent}(\tau[i]) \gg p \Rightarrow \Delta_{E/A} \succ 0$$

Die Partitionierung mit Grad p bedarf (p-1) Schnittflächen, um die Partitionen zu trennen. Die Ausdehnung einer Kachel in der gewählten Dimension i multipliziert mit der Anzahl von Schnittflächen in Relation zur Ausdehnung des Gesamtobjekts ergibt den Anteil der redundant zu ladenden Daten. Der Algorithmus wählt in der Regel eine Dimension mit minimaler Kachelausdehnung, Ausnahmen erfolgen nur beim Zusammenfallen von Kachel- und Partitionsgrenze. Unter der Annahme, dass die Ausdehnung eines Array deutlich größer ist als die Ausdehnung der Kachelung in dieser Dimension ergibt sich nur geringe Redundanz bezüglich E/A.

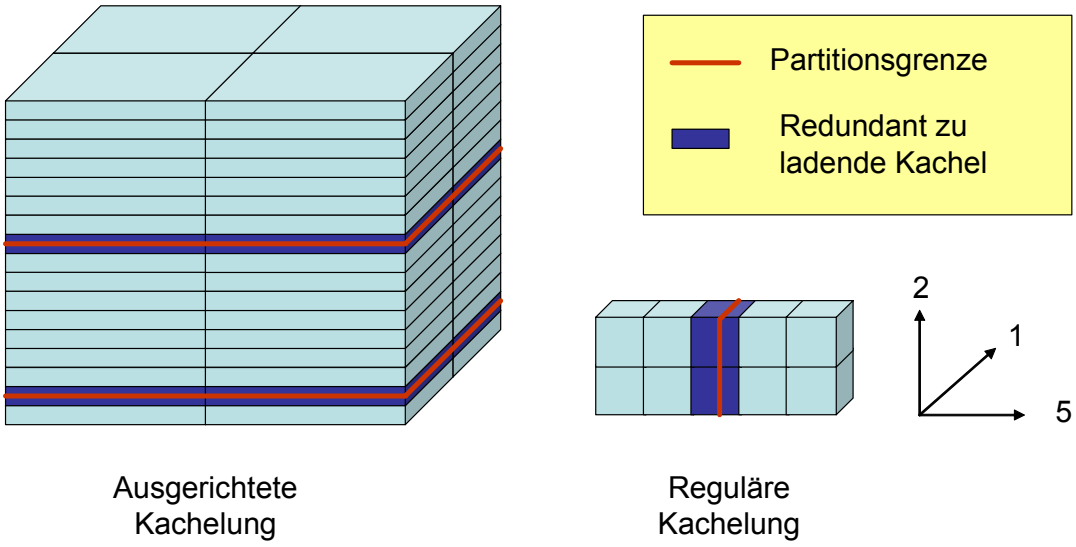


Abbildung 4.32: Redundanz von E/A beim Partitionieren von Arrays

Abbildung 4.32 skizziert links die zu erwartende Partitionierung eines typischen Array mit ausgerichteter Kachelung und einer Dimension mit geringer Kachelausdehnung. Auf der rechten Seite der Abbildung ist die Partitionierung eines Array mit regulärer Kachelung gezeigt. Da der Quotient zwischen Ausdehnung einer Kachel (hier fix) und Ausdehnung des Array minimal sein soll, wird die Dimension mit maximaler Arrayausdehnung gewählt.

Tabelle 4.6 zeigt zusammenfassend die Wirkung des vorgestellten Algorithmus für die diversen Kachelungsstrategien. Der vorgestellte Algorithmus bewirkt für allen Kachelungsstrategien eine sehr gute Partitionierung mit geringer Redundanz von E/A.

		Kachelung		
		regulär	**ausgerichtet**	**arbiträr**
Problem	**Ungleiche Last**	In den meisten Fällen. Jedoch sehr gering, da in der Praxis: $\forall d=1,\ldots,\dim(\alpha)$: extent($\alpha$[d]) >> p.		
	E/A Redundanz	1. Nein, falls Grenzen der Partitionen mit Kachelgrenzen zusammenfallen. 2. Ja, falls nicht 1 gilt. Der Algorithmus wählt die Dimension mit größter Ausdehnung für die Partitionierung.	1. Nein, falls Grenzen der Partitionen mit Kachelgrenzen zusammenfallen. 2. Ja, falls nicht 1 gilt. Der Algorithmus wählt die Dimension mit geringster Kachelausdehnung für die Partitionierung.	Ja, aber durch Analyse und Minimierung des Ladevolumens gering

Tabelle 4.6: Verhalten des Partitionierungsalgorithmus für die verschiedenen Kachelungsstrategien

4.4.4.3 Partitionierung bei binär induzierten Operatoren

Die bisherige Analyse der Partitionierung von Arrays zum Zwecke einer parallelen Verarbeitung hat sich auf die Partitionierung eines einzelnen Objekts beschränkt. Wir werden in diesem Kapitel die Partitionierung von zwei Arrays als Vorbereitung einer parallelen binär induzierten Operation analysieren. Abbildung 4.33 zeigt ein Beispiel für eine binär induzierte Subtraktion. Das Beispiel basiert auf einer vom Max-Planck Institut teils gemessenen, teils simulierten Klimaänderung, genauer gesagt der Veränderung der äquatorialen Windgeschwindigkeiten. Ein Meteorologe kann aus der Differenzen in einem Zeitraum von 200 Jahren sofort die Katastrophen erahnen, die sich für die nächsten Jahrzehnte abzeichnen.

Wird die Berechnung von binär induzierten Operationen parallelisiert, müssen zu diesem Zweck beide Arrays in identische Partitionen aufgeteilt werden. Je nach Kachelungsstrategie und Lage der Kacheln bedarf die parallele Verarbeitung einer Redundanz beim Laden. Grundsätzlich lassen sich zwei Fälle unterscheiden:

1. Die Kachelung der beiden Arrays ist vollkommen identisch.
   Dieser Fall ist nicht so selten wie es bei erster Betrachtung erscheint. Häufig werden Daten derselben Kollektion oder sogar Ausschnitte eines Arrays verknüpft. In diesen Fällen decken sich die Kachelgrenzen beider Arrays meist.
2. Die Kachelung der beiden Arrays ist ähnlich.
   Dieser Fall tritt auf, wenn die Größe der Kachelung, aber nicht deren Lage, identisch ist. Dieser Fall ist typisch bei ähnlicher physischer Modellierung von Kollektionen oder einer Operation auf Ausschnitten eines Arrays.
3. Die Kachelung ist konträr.
   Dieser Fall ist zwar selten, soll aber nicht zu pathologischen Fällen bezüglich der Partitionierung und der daraus resultierenden E/A führen.

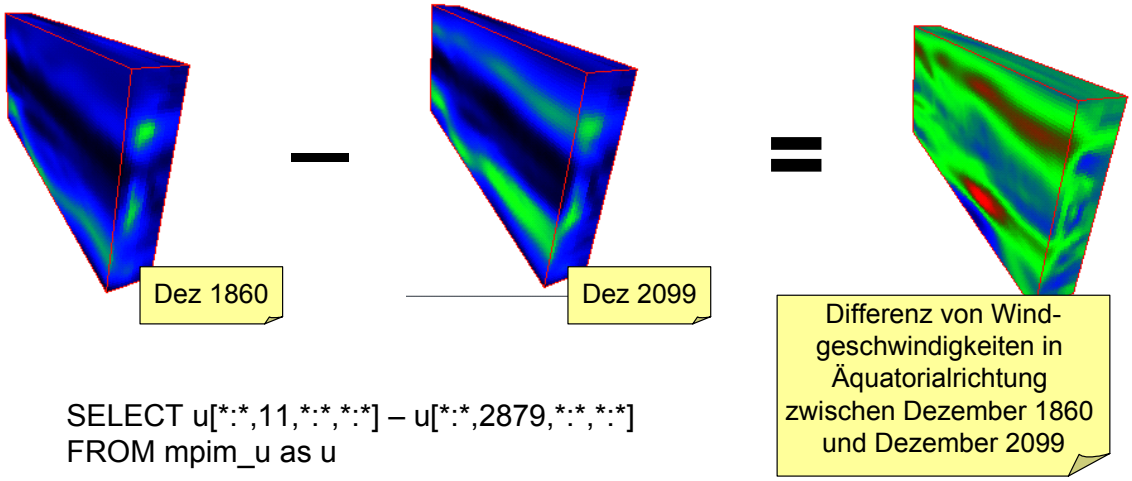


Abbildung 4.33: Beispiel für Anwendung einer binär induzierten Operation

Fall 1 bedarf keinen weiteren Überlegungen, hier sind die Analysen für Partitionierung von Einzelarrays zutreffend. Fall 2 und 3 zeigen unterschiedliche Auswirkungen auf E/A, je nachdem, ob die Strategie einer Partitionierung in Kachelmengen (4.4.4.1) oder einer Partitionierung in Subarrays (4.4.4.2) verfolgt wurde. Tabelle 4.7 gibt einen Überblick von E/A bezüglich Kachelungs- und Partitionierungsstrategie.

		Kachelung		
		identisch	**ähnlich**	**konträr**
Partitionierungsstrategie	**Bindung an Iterator**	Keinerlei Redundanz von E/A	Große Redundanz, nicht optimierbar.	
	Partiell besetzte Arrays		Große Redundanz. Optimierung schwierig.	
	Subarrays (Algorithmus 4.3)	Gering (siehe Analyse in Kapitel 4.4.4.2)	Optimierung durch Abschätzung bzgl. Dimension	Durch Algorithmus meist nahe am Optimum.

Tabelle 4.7: Analyse von E/A bezüglich Kachelungs- und Partitionierungsstrategie

Abbildung 4.34 zeigt eine Partitionierung von zwei MDD mit konträrer Kachelung, welche die Operanden für eine induzierte Differenz bilden. Oben ist die Zuteilung zu einem ersten Prozess zu sehen, wenn ein Iteratorkonzept auf Kacheln oder die Verteilung von teilweise besetzten Arrays realisiert wird. Der Prozess muss hierbei fast das komplette zweite MDD laden. Eine Optimierung dieses Verhaltens kann nur mit großem Aufwand erfolgen. Im zweiten Fall wurde eine logische Optimierung durch Schnitt in der ersten Dimension erreicht. Das zu ladende Datenvolumen ist hier aufgrund der Analyse deutlich geringer.

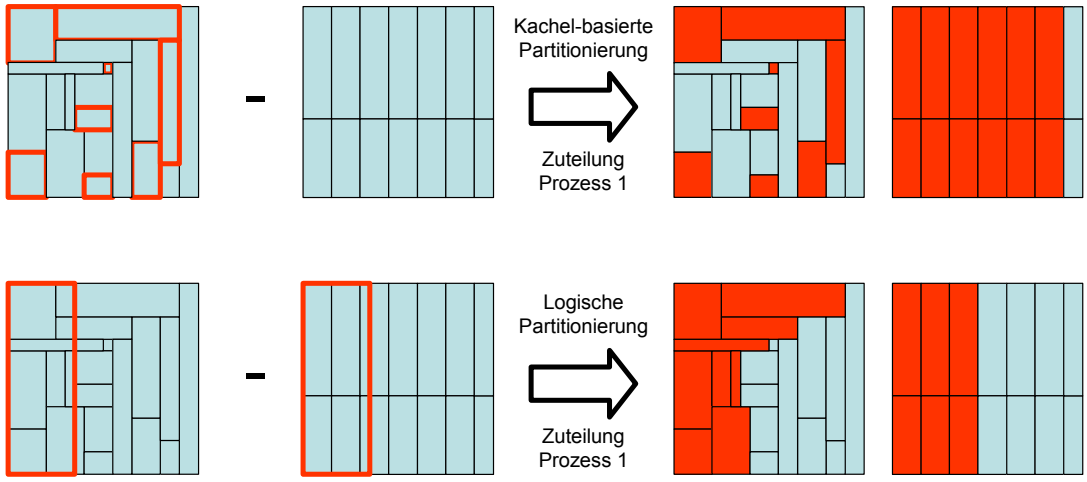


Abbildung 4.34: Konträre Partitionierung für binär induzierte Operationen

Eine parallelisierte binär induzierte Operation auf den Arrays α und β mit ähnlicher oder konträrer Kachelung bedeutet für die in Kapitel 4.4.4 vorgestellten Partitionierungsstrategien im Detail:

1. Iterator auf Basis von Kacheln
   Hier besteht einerseits die Möglichkeit, das Iteratorkonzept in ursprünglicher Form, das heißt, ohne Optimierung der Kachelreihenfolge und Vergabe an Prozesse, zu übernehmen. Alternativ dazu kann die Zuteilung a priori optimiert werden.
   a. Striktes Iteratorkonzept
      Die Verarbeitung einer Kachel des Array α benötigt das Laden aller Kacheln aus β, die diesen Bereich schneiden. Die parallele Verarbeitung zeigt eine (nicht optimierte und daher enorme) Redundanz beim Laden.
   b. Iteratorkonzept mit Analyse der Prozesszuteilung
      Die Redundanz beim Laden kann durch eine a priori Analyse optimiert werden. Dies ist eine Abkehr vom reinen Iteratorkonzept und führt im Wesentlichen zu optimierten Partitionierungen gemäß Punkt 2.
2. Übergabe partiell besetzter Arrays
   Für jedes an einen Prozess vergebene teilweise besetzte Array muss die Überdeckung mit dem zweiten Array analysiert werden. Eine Optimierung ist aufwändig, da die Überdeckung für jede Partitionierung des Array α ermittelt werden muss. In der Praxis ist eine umfassende Analyse aufgrund der Menge der Kacheln nicht effizient durchführbar.
3. Partitionierung in Subarrays mit Analyse der Kachelung gemäß Algorithmus 4.3
   Diese Strategie zeigt mit geringem Zusatzaufwand auch für binär induzierte Operationen hervorragende Ergebnisse. Da bei diesen Operationen die Domäne der Eingabearrays identisch sein muss, kann die Partitionierung zur Lastbalancierung übernommen werden, sie gilt für beide Eingabearrays gleichermaßen. Die Analyse zur Minimierung von E/A muss lediglich die zu ladenden Kacheln für beide MDD summieren.

Anfragen mit mehr als einem binär induzierten Operator verschärfen die gezeigten Probleme für auf physikalischen Objekten basierenden Partitionierungsstrategien noch, während die logische Partitionierung auch hier einfach zu berechnen ist:

1. Naive Verteilungen auf Kachelebene vermeiden Redundanz der Ladevorgänge nur für das erste Array, alle weiteren werden bezüglich Ladevolumen nicht optimiert.
2. Optimierungen werden mit der Anzahl der Eingabearrays immer komplexer, da für alle Kachelmengen die Überdeckung mit allen anderen Arrays berechnet werden muss.
3. Die Optimierung von E/A für die Partitionierung in einer Dimension bedarf der Summe der zu ladender Kacheln aller MDD[77]. Dies bedeutet keine Steigerung der Komplexität, diese ist durch den gewählten Algorithmus an die Anzahl der Dimensionen gebunden und nicht an die Anzahl der Eingabeobjekte.

Heuristik 16 Partitionierung von Arrays zur parallelen Verarbeitung
Die Partitionierung von Arrays zum Zwecke der parallelen Verarbeitung ist theoretisch ein rechenintensives Optimierungsproblem. Durch Verendung eines einfachen Algorithmus basierend auf Heuristiken, welcher die E/A Last für gute Lastbalancierungen evaluiert, wurde hervorragende Performanz mit geringem Optimierungsaufwand erreicht. Der Algorithmus hat eine Komplexität abhängig zur Dimensionalität der Eingabearrays O(dim(α)).

4.4.5 Fusion von Zwischenergebnissen

Die Partitionierung von multidimensionalen Arrays dient als Basis für die Verteilung der Daten zum Zwecke der parallelen Verarbeitung durch verschiedene Prozesse. In diesem Kapitel wird detailliert auf die Fusion der dadurch gewonnenen Teilresultate zu einem Endresultat eingegangen. Dies ist die Hauptaufgabe des *merge* Operators: die Worker Prozesse senden das Ergebnis ihrer Berechnung zu einem Master Prozess, der das finale Resultat bildet und im Anfragebaum nach oben weiterreicht.

Die Fusion von Zwischenergebnissen kann bezüglich Funktionalität in zwei Klassen unterteilt werden:

1. Fusion von Arrays zu einem kompletten Array durch (multidimensionale) Konkatenation. Dies tritt insbesondere dann auf wenn die Fusion einer induzierten Operation folgt. Lediglich eine aggregierende Funktion fällt in diese Klasse, nämlich die Skalierung *scale*.
2. Berechnung eines finalen Resultates aus einer Menge von Elementen durch eine Berechnungsfunktion. Zumeist werden skalare Werte fusioniert. Einen Sonderfall bilden die aggregierenden Operationen *max_pos*, *min_pos*, hier ist die Eingabe eine Menge von Arrays.

Algorithmus 4.4 skizziert den Algorithmus zur Fusion insgesamt.

Algorithmus 4.4: Fusion von Teilresultaten

1. Empfange das erste Teilresultat und restauriere es. Ist es vom Typ MDD, gehe zu Punkt 2. Ist es ein Skalar, gehe zu Punkt 3.

[77] Die Berechnung der Überdeckung bzw. der zu ladenden Kacheln wird in **rasdaman** durch die Methode *intersect* realisiert. Da diese lediglich auf Metadaten operiert, ist sie sehr effektiv (typische Ausführungszeiten liegen im Bereich von Millisekunden).

2. Das empfangene und restaurierte Objekt ist vom Typ MDD. Das Ergebnis des merge Opertors wird ein MDD sein.
   a. Prüfe den Kindknoten des *merge* Operators. Ist er vom Typ *max_pos* oder *min_pos*, wird das Resultat durch Funktion berechnet (Punkt 5).
   b. Konkateniere die partiellen Arrays (Punkt 3).
   c. Gib das Objekt zurück.

3. Das empfangene und restaurierte Objekt ist ein skalarer Wert.
   a. Führe für die empfangenen Teilresultate die Berechnungsfunktion der jeweiligen Aggregation (Kindknoten des *merge* Operators) durch (Punkt 5).
   b. Gib das Objekt zurück.

4. Multidimensionale Konkatenation von Arrays (Kapitel 4.4.5.1)
   a. Das erste partielle MDD bestimmt Dimensionalität und Zelltyp und ist Trägerobjekt für das Ergebnisarray.
   b. Aus anderen partiellen MDD werden die Kacheln extrahiert und in das Trägerobjekt integriert.
   c. Nach Einfügen des letzten partiellen MDD wird die Domäne des Trägerobjekts adaptiert.

5. Finalisieren einer Aggregationsfunktion (Kapitel 4.4.5.2)
   a. Wenn die Aggregationsfunktion einen frühzeitigen Abbruch erlaubt, muss eine Unterbrechung der Inter-Prozess Kommunikation initialisiert werden.
   b. Führe die Aggregationsfunktion mit Teilergebnissen aus!
   c. Prüfe einen sofortigen Abbruch, propagiere diesen gegebenenfalls an alle Prozesse!

Wir werden im Folgenden beide Fälle (in Bezug zur gewählten Partitionierungsstrategie) betrachten.

4.4.5.1 Fusion von Arrays

Eine Vereinigung von Arrays wird vollzogen, wenn im Operatorbaum zwischen *merge* Operator und dem darunter liegenden Array Operator ein Datenfluss vom Typ MDD herrscht und somit die parallelen Prozesse Kacheln oder Subarrays als Teilresultate senden (Ausnahme *max_pos*, *min_pos*). Für alle vorgestellten Partitionierungsstrategien ist eine solche Fusion problemlos möglich.

1. Bei Verwendung eines Iteratorkonzepts auf Basis von Kacheln werden von den Prozessen per *next* solange Kacheln angefordert bis das zu restaurierende Objekt mit diesen voll ausgefüllt ist. Das Parallelisierungsmodul muss hierzu die Domäne des resultierenden Arrays speichern. Dies ist nötig, um ein Abbruchkriterium für den Iterator bestimmen zu können[78].
2. Sowohl teilweise besetzte Arrays als auch Subarrays werden als Teilmengen des Resultatarrays gesehen. Die Definition eines noch nicht vollständig von Kacheln besetzten Datenelements des Typ MDD ist problemlos möglich[79], solange keine Array Operationen auf diesen nur teilweise definierten Arrays stattfinden. Die Fusion ist folglich in diesem Fall

[78] Eine andere Implementierung wartet auf ein Terminierungssignal (NULL) von jedem beteiligten Prozess.
[79] Bei der Definition eines neuen Objekts werden Metadaten (z.B. Zelltyp) aus dem ersten empfangenen MDD extrahiert und genutzt.

blockierend, da nur komplette Arrays im Operatorbaum weitergegeben werden dürfen. Die Bestimmung eines vollständig definierten (komplett fusionierten) Arrays geschieht nicht über die Definition des Array, sondern durch Bestimmung der Anzahl der Teilergebnisse: wurden p Partitionen gebildet, werden p Teilresultate erwartet.

Die von uns realisierte Implementierung ist in Algorithmus 4.4 beschrieben. Das erste empfangene MDD fungiert als Träger des finalen Resultats, Dimensionalität sowie Zelltyp sind bereits korrekt. Von allen weiteren empfangenen Subarrays werden die enthaltenen Kacheln in das Trägerobjekt eingefügt. Abschließend muss die Domäne des MDD entsprechend der eingefügten Kacheln adaptiert werden[80].

Die für die Implementierung gewählte blockierende Fusion kann in wenigen Fällen Probleme der Lastverteilung bewirken. Dies wirkt sich insbesondere bei hoher Last oder heterogenen Shared-Nothing Architekturen aus. Auf dieses Problem wird in Kapitel 4.4.6 näher eingegangen.

4.4.5.2 Fusion von Teilergebnissen von Aggregationsoperationen

Die Fusion von Teilergebnissen von aggregierenden Array Operationen, die meist skalare Werte als Resultat einer parallel ausgeführten Aggregation zu einem finalen Resultat zusammenfasst, ist nicht so problemlos möglich wie die Fusion von Arrays. In der Tat ist diese Art der Fusion die einzige Ausnahme bezüglich einer kompletten Trennung von paralleler und Berechnungsfunktionalität im Anfragebaum. Genauer gesagt, muss die Fusion Funktionalität der darunter liegenden Array Operation übernehmen und umgekehrt, einzelne Array Operationen müssen für eine anschließende parallele Fusion adaptiert werden. Im Detail wirkt sich das folgendermaßen aus:

1. Die parallele Operation *merge* übernimmt teilweise die Funktion der Array Operation.
    a. Dies impliziert, dass die *merge* Operation Kenntnis über die Aggregationsfunktion sowie deren exakte Implementierung haben muss. Die Berechnung des Aggregates aus den Teilergebnissen in *merge* muss dem Algorithmus in den Array Operationen entsprechen, um eine identische Berechnung und somit identische Gesamtergebnisse zu gewährleisten.
    b. Ebenso ergibt sich die Folgerung, dass die Definition und Implementierung neuer Aggregationsfunktionen bzw. eine Änderung der Implementierung die Adaption des *merge* Operators nach sich ziehen muss.

2. Die Funktionalität der bisherigen Aggregationsfunktion ist in wenigen Fällen nicht hinreichend für eine Fusion parallel berechneter Teilergebnisse. Im Fall unserer Implementierung eines parallelen **rasdaman** Array DBMS war dies insbesondere für die Funktion zur Ermittlung des arithmetischen Durchschnitts *avg* der Fall.

Die beschriebene Problematik der nicht möglichen Trennung von paralleler und Berechnungsfunktionalität wird im Folgenden am Beispiel der Aggregationen *some* und *avg* gezeigt.

Beispiel 4.6: Auswirkungen von Array Funktionen auf den *merge* Operator am Beispiel des Existenzquantors *some*

---

[80] Die Domäne eines MDD ist eigentlich implizit durch die Hülle der Domänen der enthaltenen Kacheln gegeben. In **rasdaman** wird aus Gründen der Effizienz die Domäne redundant in Objekten des Typ MDD gespeichert.

Der Existenzquantor ∃ wird in unserem Datenmodell durch die Aggregationsfunktion *some* dargestellt:

some: <D, 𝔹> → <D', 𝔹> mit dim(D') = 0. ∃ α[x] ∈ α | α[x] = *true* ↔ *some*(α) = *true*.

Die Fusion einer Menge von Teilresultaten des *some* Operators erfolgt logisch gesehen durch die Funktion:

some': $\{b_i \mid b_i \in \mathbb{B}\} \rightarrow \mathbb{B}$. *or*($b_i$) → *some'*($\{b_i\}$) = *true*.

Neben der logischen Funktionalität muss jedoch auch die Implementierung des Array Operators übernommen werden. In vielen Fällen kann nur so ein identisches Ergebnis von sequentieller und paralleler Berechnung gewährleistet werden. So etwa bei Konvertierungen des Datenformats, um numerische Probleme zu umgehen. In unserem Beispiel kann durch Übernahme der Implementierung der *some* Funktion eine identische Performanz erreicht werden.

Der (nicht parallele) Algorithmus für die some Funktion erlaubt einen frühzeitigen Abbruch (engl.: early termination): sobald die erste Zelle mit Inhalt *true* gefunden wird, steht das Resultat *true* fest und das Array wird nicht weiter geprüft. Die Funktion *merge* kann diesen frühzeitigen Abbruch ebenso durchführen. Sobald der erste Prozess ein *true* sendet, wird durch eine Nachricht an alle Prozesse, deren Ergebnis noch nicht vorliegt, der sofortige Abbruch signalisiert. Anschließend wird das Resultat im Anfragebaum sofort weitergereicht. Die Fusion spiegelt also folgende Verarbeitung wider:

some': $\{b_i \mid b_i \in \mathbb{B}\} \rightarrow \mathbb{B}$. ∃ $b_i \mid b_i$ = *true* → *some'*($\{b_i\}$) = *true*.

Implementierungstechnisch kann dieser sofortige Abbruch durch zwei Konzepte realisiert werden:

1. Der Operator *some* wird nicht adaptiert, die Realisierung des Abbruchs als Reaktion einer Early Termination in einem Prozess bleibt im parallelen Operator *merge* gekapselt. Der Ablauf einer Early Termination für diese Implementierung ist in Abbildung 4.35 skizziert: Der Master Prozess sendet als Reaktion auf ein Early Termination ein Abbruchsignal an die anderen Slave Prozesse, gibt das Resultat zurück und kann folglich keine weiteren Zwischenergebnisse mehr empfangen. Die Slave Prozesse empfangen das Abbruchsignal erst im *merge* Knoten und verwerfen als Reaktion den Transfer des Resultats. Dieses Konzept sichert ein sehr schnelles Ergebnis der *some* Operation ohne Adaption des Array Operators. Jedoch erzeugen die Slave Prozesse weiterhin Last, obwohl das Resultat bereits vorliegt.
2. Eine Adaption des *some* Operators wird durchgeführt. Während der Verarbeitung wird auf ein Abbruchsignal vom Master geprüft. In diesem Fall wird die Verarbeitung sofort gestoppt und kein Zwischenresultat transferiert. In Abbildung 4.35 würden die Slave Prozesse 1 und 3 mit diesem Konzept sofort nach Empfang des Abbruchsignals keine Last mehr tragen.

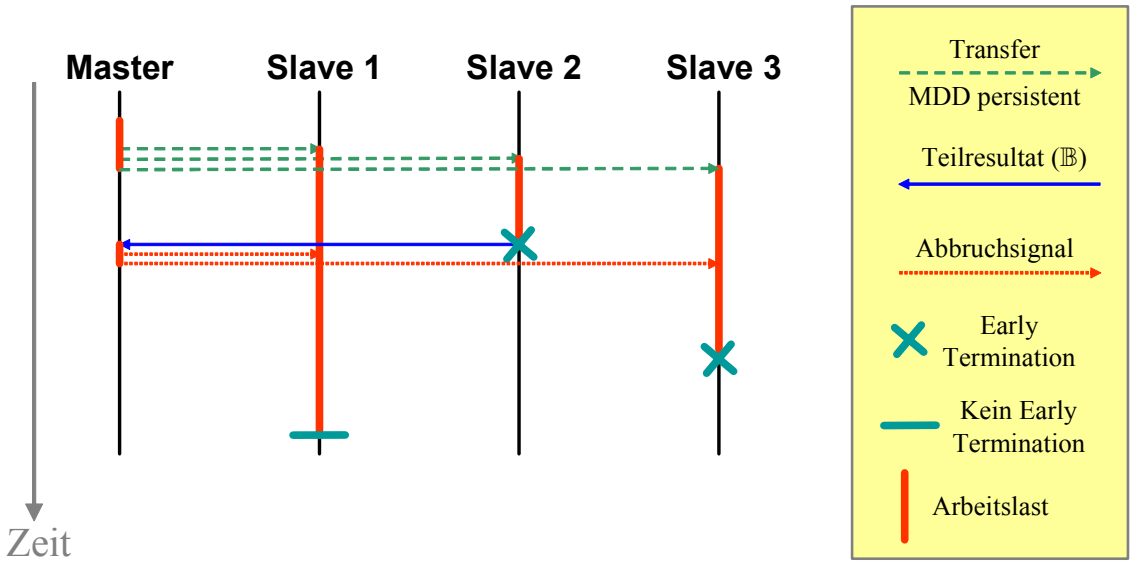


Abbildung 4.35: Ablaufdiagramm für Early Termination bei Quantoren

Das erste Konzept zeigt eine etwas bessere Performanz, da die Verarbeitung nicht durch Prüfen auf Abbruch unterbrochen wird. Das zweite Konzept wird insbesondere für ausgelastete Systeme (z.B. Mehrbenutzerbetrieb) die bessere Wahl sein.

In obigem Beispiel muss nicht nur Funktionalität, sondern auch die Implementierung der Array Operation im *merge* Operator nachvollzogen werden. Die Nachbildung einer Array Optimierung sichert hier eine identische Performanz, in vielen anderen Beispielen müssen numerisch notwendige Datenkonvertierungen kopiert werden, um ein korrektes Ergebnis zu sichern. Wir geben ein weiteres Beispiel, in welchem eine parallele Verarbeitung Auswirkungen auf eine Array Operation hat (in obigem Beispiel konnte das noch vermieden werden).

Beispiel 4.7: Auswirkungen von Parallelität auf Array Operationen am Beispiel des arithmetischen Durchschnitts *avg*

Die Berechnung des arithmetischen Durchschnitts aller Zellwerte eines Array erfolgt im originalen rasdaman Server, d.h. ohne Hinblick auf parallele Verarbeitung, durch die Funktion:

avg: <D, T> → <D', T'> mit dim(D') = 0 und T' = {*double*}. $\text{avg}(\alpha) = \frac{\text{sum}(\alpha)}{\text{count}(\alpha)}$.

Im Falle der Parallelverarbeitung kann eine Fusion der Durchschnittswerte der Partitionen zu einem finalen Resultat nicht allein aus diesen Werten erfolgen:

$$avg' \neq \frac{\sum_{i=1}^{p} avg_i}{p}$$

Dies rührt daher, dass die Teilergebnisse auf unterschiedlich großen Partitionen berechnet wurden (eine identische Größe der Partitionen kann nicht garantiert werden), aber die Information über die Größe der Partition nicht vorliegt. Wird diese Information über die Größe der Partition nachgereicht, kann die Berechnung folgendermaßen durchgeführt werden:

$$avg': \{avg_i \times count_i \mid avg_i \in \mathbb{Q}, count_i \in \mathbb{N}\} \rightarrow \mathbb{Q}.\ avg' = \frac{\sum_{i=1}^{p} count_i * avg_i}{\sum_{i=1}^{p} count_i}.$$

Logisch gesehen ist dieses Ergebnis korrekt, jedoch können in seltenen Fällen durch numerische Fehler nicht korrekte[81] Ergebnisse entstehen. Die Rückberechnung der Summe mittels Produkt des Durchschnitts und Zellenanzahl kann durch einen numerischen Fehler bei der Durchschnittsbildung fehlerhaft sein, d.h.

$$avg_i = \frac{sum_i}{count_i} \Rightarrow sum_i = avg_i * count_i \text{ gilt nicht.}$$

Ein zur sequentiellen Verarbeitung identisches Ergebnis wird allgemein nur erzielt, wenn die sequentielle Verarbeitung kopiert wird. Folglich benötigt der merge Operator zur Bildung des Durchschnitts die Summe und die Zellanzahl der Partitionen:

$$avg': \{avg_i \times sum_i \mid sum_i \in \mathbb{Q}, count_i \in \mathbb{N}\} \rightarrow \mathbb{Q}.\ avg' = \frac{\sum_{i=1}^{p} sum_i}{\sum_{i=1}^{p} count_i}.$$

Für die Implementierung bedeutet das eine Adaption des *avg* Operators und des *merge* Operators:

1. Der *avg* Operator muss die für Summe der Zellwerte und die Anzahl der Zellen gespeichert halten. Dies war bisher nicht nötig, da ein Zugriff auf diese Werte nach Berechnung des Durchschnitts nicht erfolgte
2. Der *merge* Operator muss prüfen, ob er auf einen *avg* Operator folgt. Ist dies der Fall, wird das Ergebnis des Aufrufs von *avg* (der Durchschnittswert) verworfen. Stattdessen werden die im avg Operator gespeicherten Werte bezüglich Summe und Zellanzahl angefordert und übertragen.

[81] Nicht korrekt bedeutet in diesem Zusammenhang abweichend vom Resultat einer nicht-parallelen Berechnung in **rasdaman**.

Wir haben diese zwei Beispiel zur Veranschaulichung gewählt, da sie in der Praxis bezüglich der Implementierung am meisten Aufwand bedeutet haben. Im Falle der Quantoren mit Optimierung durch frühzeitigen Abbruch war insbesondere die Inter-Prozess Kommunikation eine Herausforderung. Erst die Nachbildung dieser Optimierung für parallele Verarbeitung führte zu einem reinen Master Slave Paradigma, da hier erstmals die Notwendigkeit eines koordinierenden Prozesses bestand[82]. Im Falle des arithmetischen Durchschnitts wurde die Fehlerhaftigkeit des intuitiven Ansatzes, die Werte ungewichtet zu kombinieren, erst durch Tests aufgezeigt. Der Fehler war in der Praxis jedoch aufgrund von fast identischen Partitionen, die sich aus dem Algorithmus zur Partitionierung ergeben, immer im Bereich von maximal 10^{-6}. Dies legte anfangs den Verdacht nahe, dass das Problem rein numerischer Natur sei.

Bezüglich der Fusion von parallel berechneten Zwischenergebnissen zu einem finalen Resultat lässt sich zusammenfassen:

Heuristik 17 Fusion von Zwischenergebnissen paralleler Prozesse
Die Fusion zu einem Array mittels Konkatenation von Teilarrays ist problemlos möglich, allerdings eine blockierende Operation. Die Berechnung eines finalen Resultats aus skalaren Werten tritt meistens als Folge einer Aggregation auf. Hier zeigen sich nicht auflösbare Unschärfen bei der Trennung von paralleler Funktionalität und Array Operationen.

4.4.6 Analyse bezüglich Anfälligkeit für Skew

Der vorgestellte Algorithmus zur Partitionierung von multidimensionalen Arrays ist als statische Lastverteilungsstrategie einzuordnen (Abbildung 4.36). Die Last wird a priori auf zwei Klassen von Prozessen aufgeteilt:

1. Ein Master Prozess erhält keinerlei Last. Dieser Prozess muss jederzeit während der Verarbeitung erreichbar sein, um Aufgaben wie Fehlerbehandlung oder frühzeitiger Abbruch ohne Verzögerung übernehmen zu können.
2. Slave Prozesse tragen jegliche Last. Die Gesamtlast wird hierbei durch Partitionierung des Arrays in möglichst gleichmäßige Teilbereiche aufgebrochen. Die Anzahl der Slave Prozesse wird durch die Anzahl der Prozessoren in einer Shared-Everything Architektur bzw. die Anzahl der Rechner in einer Shared-Nothing Architektur fest vorgegeben.

Der große Vorteil dieser einfachen Strategie ist eine hervorragende Effizienz, welche aus einer minimalen Kommunikation und geringem Initialisierungsaufwand rührt. Zwei der drei großen Parallelisierungsprobleme aus Kapitel 3.1 wurden also vermieden. Ferner kann dieses Verfahren sehr einfach in die bestehende Anfrageausführung in **rasdaman** integriert werden.

[82] Ursprünglich war für den Master Prozess eine Verarbeitung des letzten, kleinsten Partition implementiert.

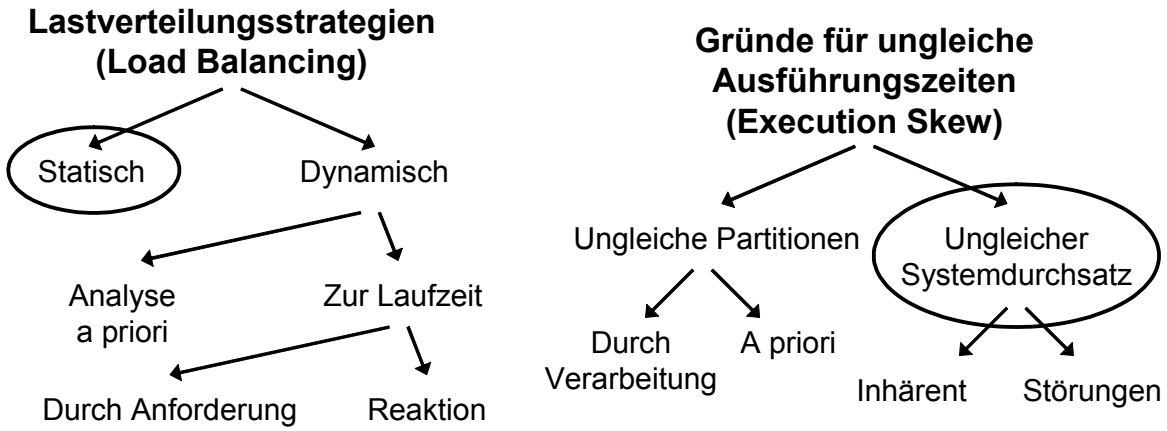


Abbildung 4.36: Klassifizierung der Intra-Objekt Parallelisierung bezüglich Verteilungsstrategien

Bekannter Nachteil einer statischen Lastverteilungsstrategie ist ihre Anfälligkeit gegen Laststörungen. Hierbei lassen sich verarbeitungsspezifische und systemspezifische Gründe unterscheiden:

1. Ungleiche Verteilung der Daten durch Verarbeitung
    a. Eine ungleichmäßige Partitionierung a priori ist durch den verwendeten Algorithmus ausgeschlossen. Leichte Abweichungen der Partitionsgrößen können (auch bei dynamischen Verfahren) nicht verhindert werden.
    b. Eine Verschiebung der Last während der Verarbeitung eines Array ist nur durch Quantoren möglich, das heißt ein Prozess führt einen frühzeitigen Abbruch der Operation durch. In diesem Fall wird jedoch durch ein Propagieren des Abbruchs der negative Effekt von Skew verhindert[83].
2. Ungleicher Durchsatz der beteiligten parallelen Instanzen
    a. Dieser Zustand kann in der verwendeten Architektur begründet sein. Vor allem in heterogenen Shared-Nothing Architekturen haben die parallelen Instanzen immer eine unterschiedliche Leistungsfähigkeit. Ein Lösungsansatz ist unten skizziert.
    b. Externe Störungen einer parallelen Instanz und damit einhergehende Verminderungen des Durchsatzes wirken sich insbesondere auf Shared-Nothing Systeme aus. In Shared-Everything wird durch dieser Effekt durch die Prozessverwaltung (und die Zuteilung zu Prozessoren) in der Regel ausgeglichen.

Der vorgestellte statische Algorithmus zur Partitionierung von Arrays ist gegen ungleichen Systemdurchsatz nicht geschützt. Im Falle einer ungleichen Ausführungszeit der Prozesse wird die Gesamtausführungszeit durch das Maximum aller Prozesse bestimmt (siehe Abbildung 4.37). Jedoch kann der Algorithmus einfach adaptiert werden, um inhärente Unterschiede im Durchsatz der parallelen Instanzen auszugleichen. Durch eine a priori Analyse während der Phase der Partitionierung in Algorithmus 4.3 kann die Performanz der parallelen Instanzen ermittelt und die Größe der Partitionen entsprechend angepasst werden. Die Evaluierung bezüglich E/A erfolgt dann auf den so ermittelten Teilarrays für alle Dimensionen.

[83] Das eigentliche Problem von Skew besteht darin, dass die Ausführungszeit durch die Zeit des am längsten arbeitenden Prozesses bestimmt wird. Das ist durch das Propagieren des Abbruchs anders: der Abbruch des ersten Prozesses bestimmt die Ausführungszeit.

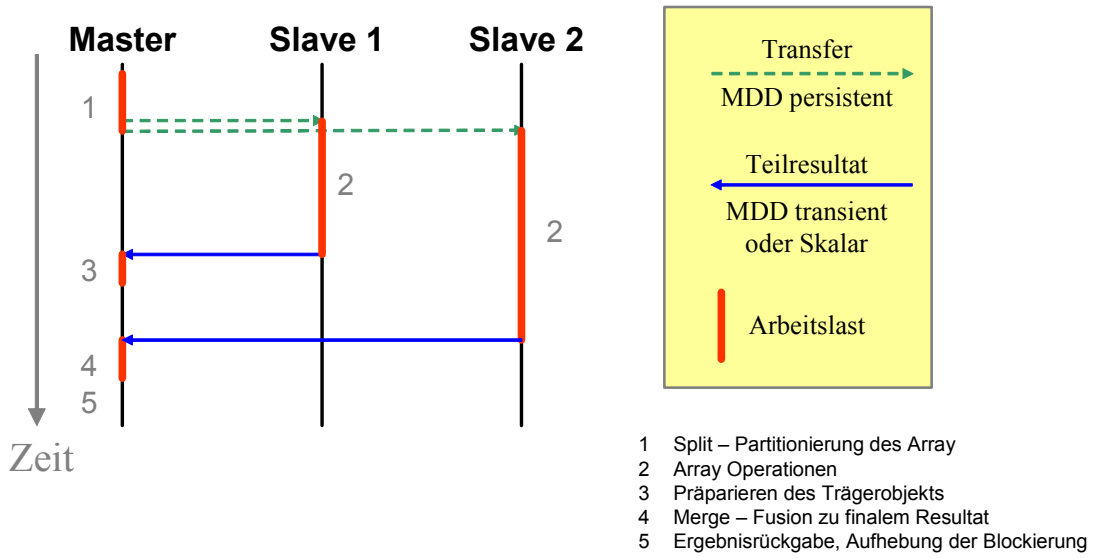


Abbildung 4.37: Ablaufdiagramm für Inter-Objekt Parallelisierung

Eine Robustheit gegen äußere Störungen ist deutlich schwieriger zu integrieren. Eine dynamische Ablaufsteuerung (durch Anforderung) wie etwa beim Iteratorkonzept ist bei vorliegender Implementierung der Übergabe von Arrays nicht inhärent. Während der Laufzeit kann somit eine Reaktion auf Laststörungen nur durch Evaluierung des aktuellen Berechnungszustandes und darauf reagierend durch eine Umverteilung der Last erfolgen.

Man muss jedoch wissen, dass jede Optimierung immer typische Szenarien adressiert und eine Reduzierung von Performanz in anderen Szenarien mit sich zieht. Eine dynamische Reaktion auf Störungen des Systems kostet Performanz für den Normalfall. Die im Projekt ESTEDI eingesetzte Plattformen für das parallele Array DBMS rasdaman sowie Diskussionen über Möglichkeiten weiterer Architekturen haben gezeigt, dass Multiprozessor Server und homogene Cluster, verbunden über ein Hochleistungsnetzwerk, die primären Plattformen für das Projekt waren. Diese Architekturen standen primär rasdaman zur Verfügung, eine Beeinflussung der angebotenen Dienste durch andere rechenintensive Programme ist nicht erwünscht. Die parallele Verarbeitung wurde diesbezüglich optimiert und zeigt unter diesen Voraussetzungen hervorragende Resultate.

4.5 Wahl der Parallelisierungsstrategie

Im Kapitel 4.3 wurde mit der Intra-Operator Parallelisierung ein Verfahren zur Parallelverarbeitung einer Menge von multidimensionalen Arrays vorgestellt. Ist die Kardinalität einer Anfrage deutlich größer als der Parallelitätsgrad, zeigt diese Methode eine hervorragende Performanz. Die Kardinalität von Array Mengen ist jedoch in der Regel deutlich geringer als die Kardinalität von Relationen, folglich auch häufig die Kardinalität der Anfragen. Ferner sind Analysen auf einem Objekt meist so komplex und rechenintensiv, dass eine parallele Verarbeitung auch hier Sinn macht. Diese Anfragen können durch Intra-Operator Parallelisierung nicht oder nur unzureichend unterstützt werden. Diese Beobachtung führte zur Entwicklung der in Kapitel 4.4 beschriebenen Intra-Objekt Parallelisierung durch Partitionierung von Arrays.

Bisher ist jedoch eine Frage noch unbeantwortet: „Wann ist welche Art der parallelen Verarbeitung die bessere Wahl?“ Um diese Frage zu beantworten, werden wir den zu erwartenden parallelen Speed-Up in Bezug zu Anfragekardinalität und Parallelitätsgrad analysieren(Kapitel 4.5.1). Darauf basierend wird in Kapitel 4.5.2 ein allgemeiner Algorithmus für die Wahl der Parallelisierungsstrategie vorgestellt. Eine Diskussion über die Kombination der Verfahren folgt in Kapitel 4.5.3.

4.5.1 Analyse der parallelen Effizienz

Grafisch lässt sich der Zusammenhang zwischen Anfragekardinalität, Parallelitätsgrad und Parallelisierungsstrategie wie in Abbildung 4.38 darstellen.

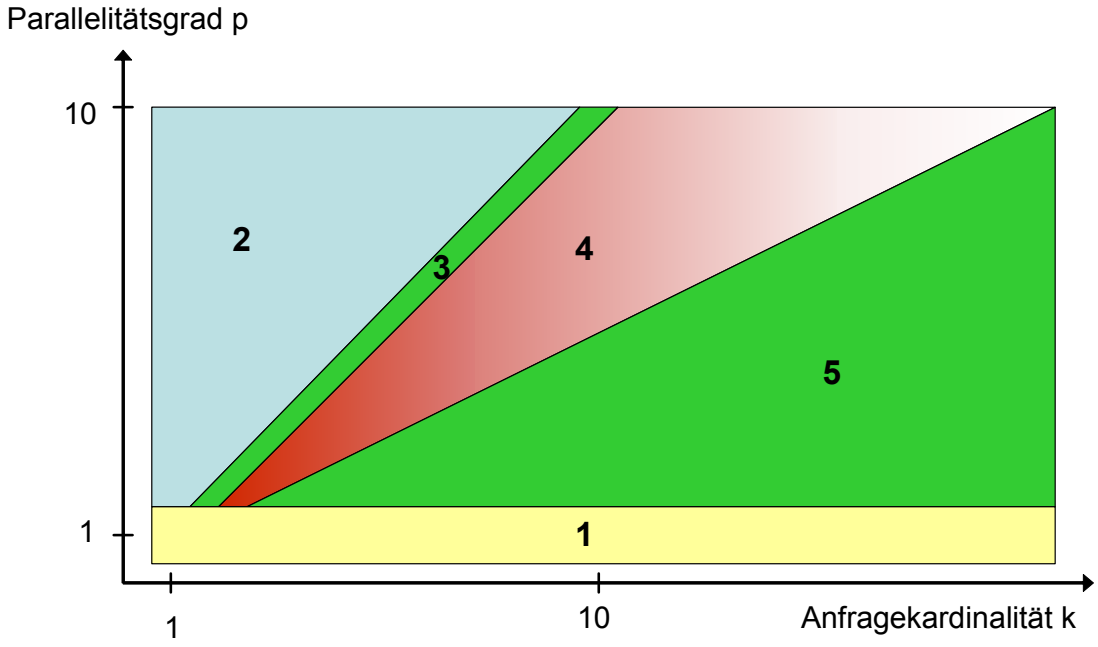


Abbildung 4.38: Abhängigkeit der Parallelisierungsstrategie von Kardinalität der zu verarbeitenden Menge und dem Parallelitätsgrad

Die Abszisse beschreibt die Anfragekardinalität nach Definition 2.9 (Anzahl der verarbeiteten MDD Objekte in der Anfrage), die Ordinate entspricht dem Parallelitätsgrad der eingesetzten Hardware. Man kann folgende Bereiche bezüglich der eingesetzten Parallelisierungsstrategie und deren Effizienz unterscheiden:

- Bereich 1: Sequentielle Verarbeitung, die eingesetzte Hardware erlaubt keine Parallelisierung
- Bereich 2: Die Anfragekardinalität ist geringer als der Parallelitätsgrad.
    a. k = 1: eine mengenbasierte Parallelisierung ist nicht möglich. In diesem Bereich kommt Intra-Objekt Parallelisierung zum Einsatz.
    b. k > 1: mengenbasierte Parallelisierung kann die zur Verfügung stehende Hardware nicht voll auslasten. In diesem Fall ist eine Intra-Objekt Parallelisierung die bessere Wahl[84].

[84] Für k = p – 1 und hohen Parallelitätsgrad, verbunden mit einer Anfrage ohne Selektivität oder frühzeitigem Abbruch einer Qauntorenoperation kann das in wenigen Einzelfällen nicht gelten. Wir lassen diesen Sonderfall außer Acht.

- Bereich 3: Die Anfragekardinalität entspricht dem Parallelitätsgrad. In diesem Fall wird Inter-Objekt Parallelisierung eingesetzt. Skew Probleme treten nur in Einzelfällen auf. Diese können durch folgende Faktoren verursacht werden:
    a. Selektion: Die Verarbeitung einer Teilmenge der Objekte endet bereits nach einer negativ evaluierten Selektion σ. Da jedoch meist ein Großteil der Ausführungszeit in der Selektion verbracht wird, hält sich der Skew in Grenzen.
    b. Quantoren: Durch frühzeitigen Abbruch dauert die Verarbeitung der Objekte unterschiedlich lange. Quantoren werden für eine Selektion benötigt und prüfen ein Prädikat. Die Laufzeitunterschiede sind in der Praxis gering, da unabhängig von der Quantoroperation selbst ein Laden der Quelldaten und eine Verarbeitung des Prädikats immer erfolgen muss.
    c. äußere Einflüsse: geringe Störungen in der Verarbeitung (z. B. Netzwerklast)
- Bereich 4: Die Anfragekardinalität liegt nur gering über dem Parallelitätsgrad. Die Verarbeitung erfolgt unter Einsatz von Inter-Objekt Parallelisierung. Der Speed-Up leidet jedoch durch die nicht-parallele Verarbeitung weniger Objekte am Ende der Anfrage.
- Bereich 5: Effiziente Inter-Objekt Parallelisierung. Eine nicht-parallele Phase am Ende der Anfrage fällt für die meisten Anfragen durch die große Anfragekardinalität nicht ins Gewicht.

Im folgenden Kapitel 4.5.2 wird der Algorithmus zur Wahl der Parallelisierungsstrategie gegeben. Dieser Algorithmus wird durch das Parallelisierungsmodul im Anschluss an die Optimierung des Anfragebaumes aufgerufen. Je nach Ergebnis der Analyse werden Algorithmus 4.1 (Intra-Operator Parallelisierung) oder Algorithmus 4.2 (Intra-Objekt Parallelisierung) für eine Adaption des Anfragebaumes angewandt.

Durch Wahl einer mengenbasierten Parallelisierungsstrategie geht eventuell im Bereich 4 Potential zur Parallelisierung verloren. Einfach ausgedrückt entspricht dieser Bereich einem Szenario wie diesem: bei Parallelisierungsgrad 4 wird eine Menge von 5 Arrays durch Inter-Objekt Parallelisierung verarbeitet. 4 Arrays werden parallel verarbeitet, für das fünfte ist eine parallele Verarbeitung nicht möglich. Eine Verbesserung des Speed-Up für diese Klasse von Anfragen ist möglicherweise mit einer Adaption der Parallelisierungsstrategie während der Ausführung zu realisieren. Kapitel 4.5.3 untersucht im Detail, inwiefern eine Adaption der Parallelisierungsstrategie während der Anfrageausführung sinnvoll ist.

4.5.2 Algorithmus zur Wahl der Parallelisierungsmethode

Das Kriterium für die Methode der Parallelisierung lässt sich einfach beschreiben: nutze die bessere Strategie, falls dadurch eine Auslastung der parallelen Instanzen gewährleistet werden kann. Intra-Operator Parallelisierung ist in der Regel performanter als Intra-Objekt Parallelisierung. Dies liegt zum einen an einer besseren Datenunabhängigkeit, zum anderen an einer durch das Iteratorkonzept möglichen dynamischen Lastbalancierung. Der Algorithmus zur Wahl der Parallelisierungsstrategie bevorzugt folglich die Parallelisierung durch Verteilung kompletter multidimensionaler Arrays.

Der Bereich 1 aus Abbildung 4.38, also ein Parallelitätsgrad von 1, wird durch den Algorithmus außer Acht gelassen. Ferner wird der Bereich 4, also eine Anfragekardinalität, die nur gering über Parallelitätsgrad liegt, durch Intra-Operator Parallelität (ohne Adaption der Strategie) verarbeitet. Wie in Kapitel 4.5.3 analysiert wird, ist die Adaption problematisch und garantiert keine verbesserte Performanz.

Algorithmus 4.5: Wahl der Parallelisierungsstrategie

1. Analysiere die Kardinalität k der Anfrage nach Definition 2.9
2. Sei p ein (durch die Hardware) vorgegebener Parallelitätsgrad.
   a. $k < p \rightarrow$ Intra-Objekt Parallelisierung (Kapitel 4.4)
   b. $k \geq p \rightarrow$ Inter-Operator Parallelisierung (Kapitel 4.3)

Adaptiere den Operatorbaum entsprechend mit den vorgestellten Algorithmen.

Dieser Algorithmus wählt die Parallelisierungsmethode aufgrund einer Analyse des durch den Optimierer aufgebauten Anfragebaum. Dieser Algorithmus traversiert zur Bestimmung der Anfragekardinalität den Anfragebaum und nutzt ausschließlich Metadaten. Er zeigt somit eine hervorragende Effizienz, die dominiert wird von der Zugriffszeit auf das RDBMS.

4.5.3 Kombination der Parallelisierungsstrategien

Die Adaption der Ausführungsstrategie kann in einer verbesserten Performanz resultieren, allerdings ist das nicht immer gewährleistet. Zu einem gegebenen Zeitpunkt t muss aufgrund gesammelter Ausführungsdaten entschieden werden, ob die Strategie der Inter-Objekt Parallelisierung beibehalten wird oder mit einem Grad $p_{neu} \leq p$ zu Intra-Objekt Parallelisierung gewechselt wird. Dieses Szenario ist in Abbildung 4.39 skizziert. Im ersten Fall (oben) wird die bisherige Strategie beibehalten, Prozess 1 verarbeitet das letzte Objekt komplett. Im zweiten Fall wird $p_{neu} = 2$ gewählt. Da in diesem Bereich des Anfragebaumes kein Iteratorkonzept mit Reaktion auf Anforderungen realisiert ist, kann ein *split* erst erfolgen, wenn die beteiligten Prozesse identifiziert sind. Die Verarbeitung kann somit erst fortgesetzt werden, wenn p_{neu} Prozesse verfügbar sind. Im dritten Fall betragt $p_{neu} = 3$, die Verarbeitung wird fortgesetzt, sobald 3 Prozesse frei sind.

Eine Optimierung des Parallelitätsgrades p_{neu} für die Intra-Objekt Parallelisierung kann nur durch Schätzung der Ausführungszeit für die Bearbeitung eines Objekts vorgenommen werden. Auf Basis dieser Daten kann der Zeitpunkt geschätzt werden, zu welchem die einzelnen Prozesse wieder verfügbar werden.

Diese Schätzung ist jedoch relativ aufwändig. Gegeben seien folgende durch die bisherige Anfrage gewonnenen Daten:

- noch zu verarbeitende Objekte *rest(k)*
- durchschnittliche Ausführungszeit von Selektion $t(\sigma)$
- durchschnittliche Ausführungszeit der Applikation $t(\alpha)$
- Anfrageselektivität *sel*
- Zeitpunkt des Beginns der Evaluierung des letzten Objekts (für jeden Prozess) $t_{begin}(i)$

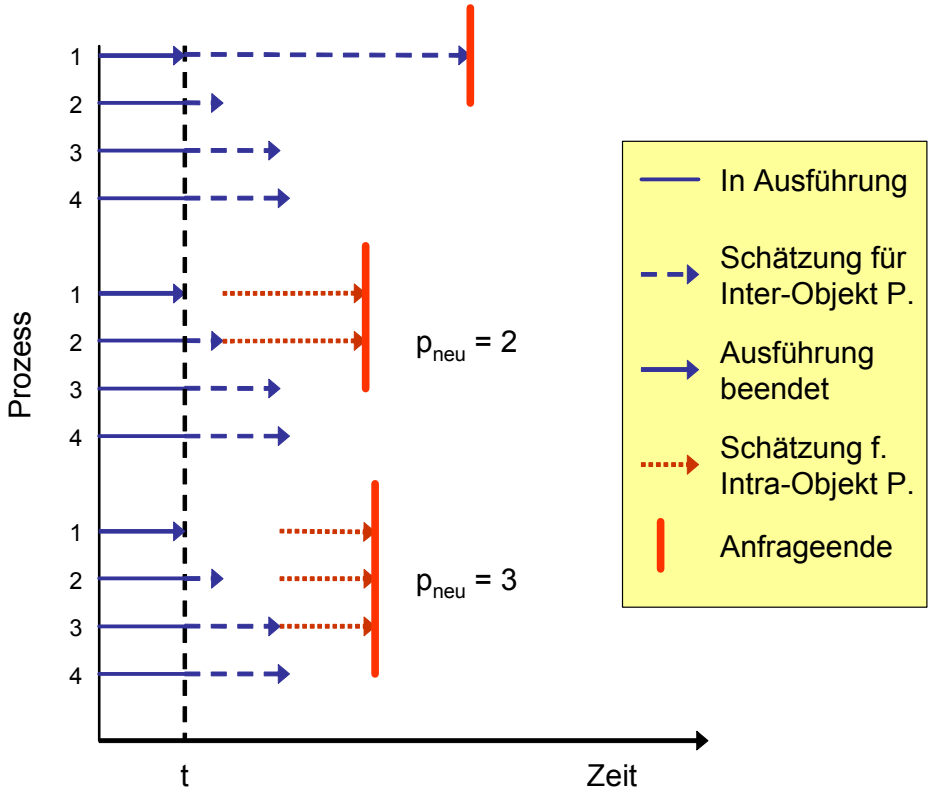


Abbildung 4.39: Adaption der Parallelisierungsstrategie während der Ausführung

Eine gewichtete Schätzung bezüglich der Evaluierung eines Objekts ist $t_{obj} = t(\sigma) + sel * t(\alpha)$. Der voraussichtliche Zeitpunkt zu dem die einzelnen Prozesse i verfügbar werden ist folglich $t_{est}(i) = t_{begin}(i) + t_{obj}$. Quantoren werden hierbei vernachlässigt, da einerseits ein frühzeitiger Abbruch nur schwer zu prognostizieren ist, andererseits die Quantoroperation gegenüber E/A und der Operation für das Quantorprädikat einen geringen Anteil (maximal 30%) hat. Die Analyse, welcher neue Parallelisierungsgrad eine optimale Performanz aufzeigt, ergibt sich folgendermaßen:

1. Berechne für alle Prozesse eine Schätzung für die Verfügbarkeit:
   $t_{est}(i) = t_{begin}(i) + t_{obj}$ für $i = 1,\ldots,p$
2. Sortiere diese Zeiten aufsteigend in $t_1,\ldots,t_p$. Diese Liste enthält nun die aufsteigend geordnet die Zeitpunkte, zu denen die Prozesse die Verarbeitung ihres gerade evaluierten Objekts voraussichtlich beendet haben.
3. Berechne für jeden möglichen neuen Parallelitätsgrad $p_{neu} = 1,\ldots,p$ eine geschätzte Ausführungszeit t_{end}: $t_{end}(p_{neu}) = t_{begin}(p_{neu}) + (t_{obj} / p_{neu} * rest(k))$. Das geschätzte Ende der Verarbeitung berechnet sich aus einer Wartezeit $t_{begin}(p_{neu})$, bis Prozesse für den entsprechenden Parallelitätsgrad frei sind und einer Ausführungszeit $(t_{obj} / p_{neu} * rest(k))$ für die noch zu verarbeitenden Objekte.
4. Wähle das p_{neu}, für das t_{end} minimal ist. Bleibe bei Inter-Objekt Parallelisierung, wenn $p_{neu} = 1$. Andernfalls wird auf Intra-Objekt Parallelisierung mit Grad p_{neu} umgestellt.

Eine Implementierung der Adaption der Parallelisierungsstrategie in das Array DBMS **rasdaman** ist aus folgenden Gründen nicht anzuraten:

1. Im Array DBMS rasdaman werden aus Performanzgründen im Normalfall keine Metadaten bezüglich der Ausführung gesammelt. Die Protokollierung von Zeiten während der Anfrage ist nur für Benchmarking zur Entwicklung vorgesehen und somit nicht auf Effizienz ausgelegt. Eine Analyse der Selektivität basierend auf Sampling wurde im Rahmen des ESTEDI Projektes zur Abschätzung der Dauer von Anfragen implementiert, jedoch ist auch dieses Verfahren Performanz hemmend. Eine Analyse der einzelnen Anteile einer Anfrage ist folglich nur sehr ineffizient möglich und wird durch den Gewinn einer Adaption nicht wettgemacht.
2. Das negative Szenario, welches durch Adaption vermieden werde soll, ist ein einzelner ausgelasteter Prozess bei gleichzeitig vielen verfügbaren Prozessen. Dieser Fall tritt ein, wenn alle Prozesse die Verarbeitung eines Objekts zu fast identischem Zeitpunkt abschließen und nur noch ein Objekt verarbeitet werden muss. Messungen haben gezeigt, dass dieser Fall in der Praxis fast nie eintritt. Die gleichzeitige Anforderung von Daten durch die Prozesse und damit einhergehende kurzfristige Engpässe bei E/A führen zu alternierender Ausführung der Prozesse: ein Teil der Prozesse fordert E/A an während der andere Teil Berechnungen auf den Daten durchführt. Diese Verschiebungen der Ausführungszeiten führen dazu, dass ein gleichzeitiges Beenden der Verarbeitung eines Objekts nur durch Zufall zustande kommt, selbst wenn logisch gesehen die Evaluierung eines Objekts ein identischer Aufwand ist.

In der Implementierung eines parallelen Array DBMS rasdaman wurde folglich auf eine Adaption der Parallelisierungsstrategie verzichtet. Eine Analyse zu Beginn der Anfrage entscheidet über die Methode der Parallelisierung.

4.6 Zusammenfassung

In diesem Kapitel wurden ein umfassendes Konzept und darauf basierend Algorithmen für die effiziente parallele Analyse von multidimensionalen Array Daten entwickelt. Eine Analyse von multidimensionalen Array Daten und deren Verarbeitung in **Kapitel 4.1** zeigte folgende Besonderheiten auf, die sich für die Entwicklung der Konzepte als wichtige Rahmenbedingungen erweisen:

1. Materialisierte Arrays sind extrem groß und komplex. Ein Transfer dieser Objekte zwischen parallelen Instanzen reduziert die Performanz und soll möglichst vermieden werden.
2. Array Anfragen haben häufig eine geringe Kardinalität. Auch für diese Anfragen muss eine effiziente parallele Verarbeitung gesichert sein.
3. Optimierungen von Array Anfragen müssen beachtet werden. Dies vermeidet den Transfer materialisierter Arrays und die Zerstörung von Optimierungen wie etwa Caching.

Kapitel 4.2 analysierte die aus der relationalen Anfrageverarbeitung stammende Inter-Operator Parallelität. Beide Ausprägungen, Pipeline Parallelität sowie Bushy-Tree Parallelität, wurden aufgrund der Spezifika von Array Daten und deren Verarbeitung als Parallelisierungsmethoden verworfen.

Die in parallelen RDBMS primär verwendete Parallelisierungsstrategie der Intra-Operator Parallelisierung wurde in **Kapitel 4.3** umfassend auf ihre Eignung für Array Daten untersucht und als hervorragend geeignet befunden. Zur Realisierung der Intra-Operator Parallelität wurden folgende Fragen geklärt:

1. Welche Operatoren können parallel verarbeitet werden, welche eignen sich hierfür? Eine Analyse der Operatoren zeigte für Applikation α und Selektion σ ein sehr gutes Potential zur parallelen Verarbeitung. Für weitere Operatoren ist Parallelität zwar theoretisch möglich, allerdings aufgrund fehlender Komplexität nicht anzuraten.
2. Wie kombiniert man die parallele Verarbeitung mehrerer Operatoren? Die Ausführung von Operatoren auf einer parallelen Instanz entspricht im Anfragebaum der Bildung von parallelen Blöcken. Diese Strategie minimiert den Transfer von materialisierten Arrays und kann Optimierungen zwischen Operatoren, insbesondere Caching Mechanismen zwischen α und σ, bewahren. Die Fusion von Operatoren zu parallelen Blöcken wurde für typische Anfragebäume analysiert, des Weiteren wurden komplexere Anfragebäume betrachtet, die durch Optimierungen und verschachtelte Anfragen entstehen.
3. Wie wird parallele Funktionalität, wie etwa der Datentransfer zwischen parallelen Instanzen, in die Anfrageverarbeitung integriert? Zur Verwirklichung der parallelen Ausführung dieser Blöcke im Anfragebaum wurden neue Operatoren definiert, welche die parallele Funktionalität kapseln. Diese Operatoren werden durch einen neu entwickelten Algorithmus in die Anfragebäume eingefügt.
4. Das Iteratorkonzept der Anfrageausführung basiert nun nicht mehr allein auf Methodenaufrufen, sondern auf Nachrichtenaustausch. Die Kommunikation zwischen parallelen Instanzen erfolgt durch Nachrichten, welche Aufrufe und Zwischenergebnisse transferieren.
5. Eine theoretische Analyse des mit dieser Parallelisierungsstrategie zu erwartenden Speed-Up schließt das Kapitel. Die Analyse ergab, dass Intra-Operator Parallelität für wenige typische Anfragen an Array Daten keine oder unzureichende Performanzsteigerung erreicht. Eine parallele Verarbeitung dieser Anfragen wird jedoch in Kapitel 4.1 gefordert.

Die parallele Analyse eines einzelnen Arrays ist eine häufige Anfrage und wird durch die Ergebnisse aus Kapitel 4.1 ausdrücklich gefordert. Dies bedarf einer Aufteilung eines Arrays zur Verarbeitung in verschiedenen parallelen Instanzen und einer späteren Fusion der Teilergebnisse. Dieses Konzept wurde in **Kapitel 4.4** als Intra-Objekt Parallelisierung beschrieben. Die Realisierung der Intra-Objekt Parallelisierung impliziert im Detail folgender Fragestellungen:

1. Welche Array Operationen können parallel ausgeführt werden? Welche Operatoren sind aufgrund ihrer Komplexität geeignete Kandidaten? Die Analyse ergab, dass eine Parallelisierung für alle induzierten Operationen und einen Großteil der Aggregationsoperationen möglich und sinnvoll ist.
2. Die parallele Verarbeitung kann auf Verknüpfungen von Operationen ausgedehnt werden, um Datentransfer zu vermeiden.
3. Wie wird parallele Funktionalität in den Anfragebaum integriert? Zur Realisierung wurden zwei neue Operatoren definiert und ein Algorithmus zur Baumadaptierung vorgestellt.
4. Welche Methoden gibt es für die Partitionierung von multidimensionalen Arrays? Die Strategien wurden in Relation zu mehreren möglichen Implementierungen untersucht. Für die Anfrageverarbeitung in **rasdaman** wurde ein effektiver Algorithmus basierend auf Heuristiken vorgestellt.
5. Wie erfolgt die Zusammenfassung von partiellen Ergebnissen zu einem finalen Resultat? Die Fusion kann eine multidimensionale Konkatenation von Arrays bedeuten oder einen finalen Berechnungsschritt zur Bildung eines aggregierten Wertes bedeuten. Ein Algorithmus zur Fusion wurde entwickelt und an besonders problematischen Fällen beispielhaft vorgestellt.
6. Abschließend erfolgt eine Analyse von Skew Effekten, die mit dieser Parallelisierungsstrategie auftreten können.

Kapitel 4.5 analysierte Anfragen bezüglich ihrer Kardinalität und dem Parallelitätsgrad und der zu erwartenden Performanzsteigerung durch Parallelität. Ergebnis dieser Analyse ist ein Algorithmus zur Wahl der Parallelisierungsstrategie. Eine Kombination der Verfahren wird diskutiert, jedoch bezüglich der Integration in rasdaman insbesondere aufgrund von Restriktionen von rasdaman und mangels zu erwartender Performanzsteigerung verworfen. Folglich wird für eine parallele Verarbeitung von Array Daten entweder Intra-Operator Parallelität oder Intra-Objekt Parallelität eingesetzt. Dies wird am Beginn der Anfrage vom Parallelisierungsmodul nach einer Analyse der Anfragekardinalität und des Parallelitätsgrades entschieden.

Kapitel 5 Implementierung: rasdaman 5.0 PE

All programmers are playwrights
and all computers are lousy actors.
(Unknown)

5.1 Prämissen der Implementierung

Die Implementierung der in Kapitel 4 entwickelten Konzepte zur parallelen Analyse von Arrays ist kein „proof of concept“ (übersetzt in etwa: Beweis der entwickelten Konzepte), wie er für wissenschaftliche Arbeiten häufig üblich ist. Diese Implementierungen haben als Ziel die Untermauerung von wissenschaftlichen Thesen und neigen dazu, dass sie nur von den Wissenschaftlern selbst benutzt werden können. Instabilität, schlechte Bedienbarkeit, Einschränkung der Funktionalität sind die häufig gesehenen Begleiterscheinungen von Software, die nicht wirklich für Anwender geschrieben wurde. Funktioniert das Programm gut genug, um reproduzierbare Messergebnisse zu erzeugen, hat es seinen Zweck erfüllt.

Ein Ergebnis dieser Dissertation ist das parallele Array DBMS **rasdaman 5.0 PE** (Parallel Edition), welches größtenteils während des EU Projektes ESTEDI realisiert wurde. In diesem Rahmen wurden die **rasdaman** Versionen 3.5 und 5.0 um parallele Anfrageverarbeitung erweitert. Version 5.0 wurde von den Projektpartnern erfolgreich eingesetzt und evaluiert. Hierzu war es nötig, **rasdaman PE** in Funktionalität, Benutzerfreundlichkeit und Administrierbarkeit dem ursprünglichen **rasdaman** gleichwertig zu machen. Nebenbei sollte es bezüglich Anfragezeiten natürlich deutlich schneller und für keine Anfrage spürbar langsamer sein. Vor allem die Zusatzanforderungen, welche in der Implementierung viel Aufwand bedeuten, waren die Basis für die Akzeptanz von **rasdaman PE** im Projekt ESTEDI.

Darüber hinaus war ein erklärtes Ziel von ESTEDI die Integration von paralleler Anfrageverarbeitung in **rasdaman** als erfolgreich seit 1998 kommerziell vertriebenes Array DBMS. Mehr noch als die Partner im Projekt ESTEDI müssen die kommerziellen Anwender den Zusatznutzen der Parallelität ohne großen Aufwand abschöpfen können. Wichtige parallele Pa-

rameter wie Anzahl der CPU oder Nutzung proprietärer Netzwerke wird einmalig von einem Administrator bei Installation vorgegeben, die Anwender selbst bemerken von Parallelität nichts außer einer verbesserten Performanz. Im Detail sind die Anforderungen an die Implementierung wie folgt:

1. Stabilität
   Die Anfrageverarbeitung muss ebenso stabil erfolgen wie im nicht-parallelen Server. Diese Anforderung erscheint auf den ersten Blick einfach, stellt sich in der Praxis jedoch als extrem schwierig zu realisieren heraus. Einerseits muss die korrekte Funktionalität auch bei „exotischen“ und extrem komplexen Anfragen sichergestellt sein. Auf der anderen Seite erweist sich insbesondere die Inter-Prozess Kommunikation und die damit verbundene Linearisierung und Restaurierung von multidimensionalen Arrays als neue potentielle Fehlerquelle. Eine Sicherstellung der Stabilität des parallelen rasdaman wurde durch umfangreiche Tests gewährleistet, die sowohl alle Operationen als auch alle Arten von zu übertragenden Daten ausgiebig testen.
2. Transparenz für die Benutzer
   Die Benutzer des parallelen Array DBMS rasdaman bemerken keinerlei Unterschied zu der nicht-parallelen Version außer natürlich einer verbesserten Performanz. Die parallele Anfrageverarbeitung erfolgt automatisch auf Seite des Servers und bedarf keiner Interaktion des Anwenders.
3. Benutzerfreundlichkeit
   Hiermit ist insbesondere die Benutzerfreundlichkeit für einen Datenbankadministrator gemeint. Parallelität zeigt sich bei der Installation von rasdaman aber auch jederzeit während des Betriebs einfach administrierbar. Hierzu erfolgte eine Integration von parallelen Parametern wie etwa Parallelitätsgrad und Kommunikationsprotokoll in das zentrale rasdaman Administrationswerkzeug, den rasdaman Manager. Die Mechanismen zur Inter-Prozess Kommunikation selbst sind in der MPI Implementierung LAM enthalten. Diese wird vorkompiliert in die rasdaman Distribution integriert.
4. Integration von Parallelität in Dokumentation und Hilfe
   Die Dokumentation von rasdaman wurde um Handbücher für die Administration eines parallelen Servers erweitert. Ebenso wurden die Hilfetexte, insbesondere die Online Hilfe des rasdaman Managers, um entsprechende Passagen ergänzt.
5. Bewahrung und Adaption der Fehlerbehandlung
   Interne Fehler in der Anfrageausführung können entweder als normales Abbruchkriterium in der Anfrageverarbeitung auftreten (z.B. wenn eine Operation auf den Quelldaten nicht definiert ist) oder auf Programmfehler hindeuten. In beiden Fällen soll ein Absturz des Servers vermieden werden. Stattdessen muss eine aussagekräftige Fehlermeldung an den Benutzer weitergeleitet werden. Diese kann zu einer Anpassung der Anfrage oder einer detaillierten Fehlerbeschreibung für die Entwickler führen. Für den parallelen rasdaman Server wurde dieser Mechanismus vollständig bewahrt und um spezielle Abbruchkriterien für Probleme bei paralleler Verarbeitung ergänzt.

Eine detaillierte Dokumentation der Implementierung ist an dieser Stelle nicht vorgesehen, wir beschränken uns auf zwei Punkte: erstens werden in Kapitel 5.2 Entscheidungen zum Design des parallelen rasdaman Server, sowie der unterstützten parallelen Architektur und des Inter-Prozess Protokolls diskutiert, die vor Implementierungsbeginn getroffen wurden. Diese Entscheidungen sind letztlich entscheidend für die Architektur des parallelen rasdaman PE Servers, die in Kapitel 5.3 vorgestellt wird. Abschließend werden zwei Beispiele für die Interaktion der parallelen Prozesse kurz erläutert.

5.2 Designentscheidungen

Zu Beginn des Projektes ESTEDI und somit vor Beginn der Implementierungsarbeiten zur Integration von Parallelität in rasdaman mussten Architektur- und Designentscheidungen getroffen werden, die die Erscheinung von rasdaman PE bis heute bestimmen. Prägnant ausgedrückt, waren folgende Fragen zu klären:

1. Wie wird Parallelität in rasdaman integriert? Implementiert man parallele Anfrageverarbeitung als unabhängige Weiterentwicklung oder als Option?
2. Welche parallelen Architekturen sollen unterstützt werden?
3. Welches Protokoll (bzw. welche Protokolle) wird für die Inter-Prozess Kommunikation gewählt?

Diese Fragen werden die folgenden Kapitel im Detail diskutiert und die getroffene Entscheidung erklärt.

5.2.1 Adaptionsstrategie

Eine Integration von paralleler Anfrageverarbeitung in rasdaman kann eine unabhängige Implementierung oder eine Adaption durch Module für die Parallelität bedeuten. Eine Reimplementierung bietet den Vorteil, dass keine Kompromisse bei der Umsetzung der erarbeiteten Konzepte nötig sind. Probleme oder Inkompatibilitäten zwischen dem ursprünglichen rasdaman Array DBMS und paralleler Verarbeitung können so einfach beseitigt werden, indem inkompatible Module angepasst oder gegebenenfalls ausgetauscht werden. Der Nachteil einer solchen Vorgehensweise besteht vor allem in Schwierigkeiten für eine spätere Zusammenführung der Entwicklungsstränge. Im Rahmen des ESTEDI Projektes arbeiteten mindestens drei verschiedene Entwicklungsteams parallel, neben der parallelen Anfrageverarbeitung wurde eine automatisierte Anbindung an Tertiärspeichersysteme [Rei05] bei FORWISS entwickelt, Active Knowledge GmbH integrierte neue Funktionalität wie benutzerdefinierte Funktionen und Mehrbenutzerbetrieb. Auf den ersten Blick erscheinen diese Funktionen unabhängig, bei näherer Betrachtung beinhaltet eine parallele Implementierung jedoch enormes Potential für Quereffekte und Inkompatibilitäten im rasdaman Datenbankkern.

Um eine spätere Fusion der Entwicklungsstränge zu ermöglichen, wurde folglich Parallelität als Option in rasdaman integriert:

- Die parallele Funktionalität ist im Quellcode gekapselt (durch eine Präprozessordirektive). Der Quellcode ermöglicht somit die Erstellung eines rasdaman DBMS mit oder ohne parallele Funktionalität. Somit war es auch jederzeit während der Entwicklung möglich, Quereffekte von nötigen Anpassungen auf den ursprünglichen Datenbankkern zu testen und zu vermeiden.
- Vereinzelt wurde bei der Implementierung der Parallelität Nachlässigkeiten oder Fehler im Quellcode von rasdaman entdeckt, die erst bei paralleler Ausführung zutage treten. Diese wurden kommentiert und bereinigt.
- Der Nachteil einer Anpassungsstrategie besteht darin, dass interne Schwachstellen von rasdaman nicht nur beibehalten, sondern teilweise durch parallele Verarbeitung sogar verstärkt werden. Vor allem die inkonsistente Verwaltung von Arrays im Hauptspeicher (verstärkt durch den internen Caching Mechanismus) bereitete für parallele Verarbeitung eine echte Herausforderung. Ein paralleler rasdaman Server ohne so genannte Speicherlecks konnte nur mit großem Aufwand realisiert werden.

Der getroffene Aufwand einer „Rückwärtskompatibilität“ hat sich bei der finalen Integration bezahlt gemacht. Mit dem Sprung von rasdaman Version 3.5 auf 5 konnte parallele Funktionalität mit vertretbarem Aufwand in einen von Active Knowledge GmbH gelieferten massiv veränderten Datenbankkern eingebracht werden.

5.2.2 Parallele Zielplattformen und Kommunikation

Eine wichtige strategische Entscheidung für die Realisierung eines parallelen DBMS ist die Wahl der Zielarchitekturen und damit eng verbunden, die Wahl

1. des parallelen Paradigmas und
2. einer Methode (Protokoll) für die Kommunikation zwischen parallelen Instanzen.

Die Architekturen lassen sich grob in zwei Klassen einteilen, nämlich Multiprozessorrechner und Rechnerverbünde. Die dazu entsprechende Programmierparadigmen sind Threads mit einer Kommunikation über gemeinsamen Speicher und Sperrmechanismen (Semaphore) auf der einen Seite, und Prozesse mit einer Kommunikation über Netzwerke auf der anderen Seite. Ein Protokoll für die Kommunikation basiert entweder auf den entsprechenden Mechanismen von Betriebssystemen (engl.: low-level) oder auf einer Zwischenschicht mit einer vereinfachten Programmierschnittstelle (engl.: high-level). Tabelle 5.1 zeigt wichtigste parallele Kommunikationsmechanismen im Überblick.

	Low-level	**High-level**
Multiprozessor	POSIX Threads Solaris Threads	OpenMP MPI
Rechnerverbünde	Prozesse / TCP RPC	PVM MPI

Tabelle 5.1: Beispiele für Kommunikationsmechanismen

Eine wichtige Entscheidung zu Beginn des ESTEDI Projektes war also die Wahl der einzusetzenden Architektur und der parallelen Kommunikation. Aus Sicht der Anwender, also der Partner im Projekt ESTEDI, war die Option auf jede parallele Architektur, auf eine Reihe von Netzwerken (Ethernet oder neue Standards wie etwa Infiniband) und auf jede Art von Datenquelle erwünscht (siehe Tabelle 5.2). Lediglich eine Verteilung der Datenbasis auf mehreren unterschiedlichen relationalen Datenbanken wurde aufgrund von Sicherheitsbedenken und wegen einer problematischen Administration als nicht nötig und nicht erwünscht eingestuft. Ausdrücklich gefordert wurde jedoch eine einfache nachträgliche Skalierbarkeit bezüglich E/A und Berechnungsressourcen für die Analyse.

Die Wahl des Kommunikationsmechanismus fiel auf MPI, genauer gesagt auf die Implementierung LAM, da hier durch eine Programmierschnittstelle sowohl alle geforderten Betriebssysteme, als auch SMP und MPP effizient unterstützt werden. Je nach vorhandener Architektur und Betriebssystem bildet MPI die Kommunikation auf die entsprechenden low-level Mechanismen ab. Ferner können durch MPI auch heterogene Rechnerverbünde und diverse Netzwerke (bzw. Mechanismen zur Kommunikation über gemeinsamen Speicher) genutzt werden. Darüber hinaus unterstützt die eingesetzte Implementierung LAM alle relevanten neuen Netzwerk Standards wie etwa Infiniband, Myrinet oder Globus.

Die Skalierbarkeit des parallelen rasdaman bezüglich E/A wurde neben Möglichkeiten der Hardware Erweiterung (z.B. RAID Systeme) durch die Möglichkeit eines Einsatzes von Da-

tenbankreplikation sichergestellt. In der Tat kann jeder interne Server (wenn nötig) auf eine eigene replizierte Datenquelle zugreifen. Die einfache Skalierbarkeit bezüglich Berechnungsressourcen wurde einerseits durch die entwickelten Algorithmen, andererseits durch die Möglichkeit einer einfachen Erweiterbarkeit eines Rechnerverbundes realisiert. Die Aufnahme einer neuen Instanz in den Pool der internen Server erfordert lediglich die Definition eines neuen Rechners im rasdaman Manager und des Neustart des parallelen Servers.

Betriebssystem	SUN Solaris ✓		Linux ✓
Architektur	Multiprozessor (SMP) ✓		Cluster (MPP) ✓
Cluster	Homogen ✓		Heterogen ✓
Netzwerk	Ethernet ✓		Neue Standards ✓
Datenquelle	Gemeinsame Platten (Shared-Disk) ✓		Verteilte Platten ✓
Datenverteilung	Fragmentierung ✗	Partitionierung ✗	Replikation ✓

Tabelle 5.2: „Wunschzettel“ der ESTEDI Partner bzgl. paralleler Architekturen

Die Entscheidung für MPI und darüber hinaus für LAM als eine Open Source Implementierung dieses Standards war zu Beginn des Projekts ESTEDI eine gewagte Entscheidung, da der Erfolg dieses neuen Standards unsicher war. Aus heutiger Sicht war die Entscheidung eine hervorragende Wahl.

5.3 Architektur des parallelen Array DBMS rasdaman 5.0 PE

rasdaman 5.0 PE zeigt sich gegenüber der 4-Schichten Architektur des original rasdaman DBMS als Architektur mit 5 Schichten (Abbildung 5.1). Die ursprünglichen Schichten Client, Manager, RDBMS sowie deren Kommunikationsschnittstellen wurden unverändert übernommen. Der rasdaman PE Server besteht nunmehr jedoch aus zwei Schichten: ein rasdaman Server dient als Kommunikationspartner für den Manager und die Clients sowie als Koordinator für die Arbeitszuteilung. In einer zweiten Schicht befinden sich interne Server, die eine parallele Anfrageverarbeitung realisieren. Jeder interne Prozess ist eindeutig einem Koordinator zugewiesen. Die Kommunikation mit dem Server Prozess und untereinander geschieht mittels LAM-MPI, somit können die Prozesse auf einem Rechner positioniert sein (umrandete Linie links in der Abbildung) oder sich auf mehreren Rechnern in einem Netzwerk befinden (rechts).

Jeder Prozess hält eine eigene Verbindung zum RDBMS. Durch diesen Mechanismus kann volle E/A Parallelität einfach verwirklicht werden, indem mehrere replizierte Datenquellen genutzt werden. In diesem Fall nutzt ein Lesezugriff alle Datenbanken parallel, ein Schreibzugriff erfolgt immer auf einer Datenbank und wird synchron oder asynchron an die Replikationsinstanzen propagiert[85]. Der Zugriff auf replizierte Datenbanken wurde speziell für Oracle implementiert und getestet. Der Mechanismus der Bindung von Datenquelle und internem rasdaman Prozess kann hier auf verschiedene Arten erfolgen:

1. Lokale Bindung
   rasdaman Prozessen werden an ein lokales (auf dem Rechner befindliches) RDBMS ge-

[85] Nach unseren Erfahrungen wird ein Array DBMS ausschließlich als Analysewerkzeug und nicht als Transaktionssystem genutzt, d.h. nach einmaligem Einfügen der Daten finden auf diesen nur noch Lesezugriffe statt.

bunden. In diesem Fall muss auf jedem Knoten des Rechnerverbunds eine replizierte Datenbank vorhanden sein.

2. Bindung über Oracle Net8 (Oracle Listener)
   Oracle erlaubt die Bindung an remote Datenbanken durch Namensauflösung per Oracle Net8. Hierzu muss Oracle Client auf den jeweiligen Rechnern installiert sein. Dadurch kann jeder rasdaman Prozess beliebig an eine lokale oder nicht-lokale Oracle Instanz gebunden werden. Dieser Mechanismus wurde erfolgreich implementiert und getestet und kann durch rasdaman 5.0 PE genutzt werden.
3. Explizite Bindung
   Der Zugriff auf eine Oracle Datenbank kann explizit in einem *connect* Befehl spezifiziert werden. Dies bietet jedoch keine weiteren Vorteile außer der Vermeidung einer Oracle Client Installation. Eine Definition der Zugriffsstruktur ist darüber hinaus extrem komplex. Dieser Mechanismus wurde in rasdaman 5.0 PE nicht integriert.

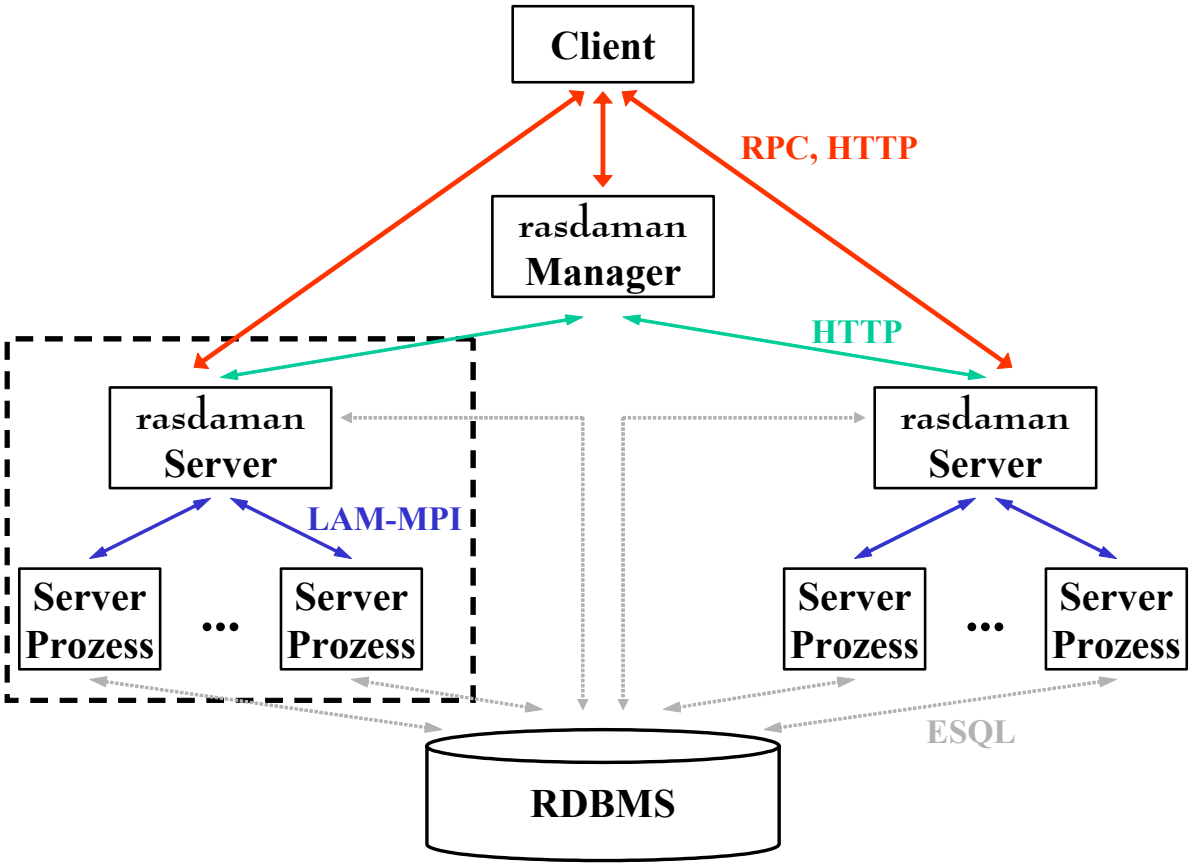


Abbildung 5.1: Architektur des parallelen Array DBMS rasdaman

Im folgenden Kapitel wird die Adaption des rasdaman Servers zum Zwecke der Integration von paralleler Anfrageverarbeitung detailliert erläutert.

5.3.1 Anpassung der Module für rasdaman Server Prozesse

Die Implementierungsstrategie mit Parallelität als Option spiegelt sich in der Architektur des parallelen rasdaman Array DBMS wider (Abbildung 5.1). Die Schichten des original rasdaman wurden soweit möglich unverändert übernommen. Lediglich zwei Schichten wurden verändert:

- der rasdaman Manager wurde für die Administration paralleler Server angepasst. Definition, Änderung, Start und Beenden und Anzeige der Einstellungen eines parallelen Servers wurden adaptiert. Entsprechende Hilfetexte wurden integriert
- der rasdaman Server gliedert sich nunmehr in einen koordinierenden Server Prozess (Master) für zentrale Aufgaben und einen Pool von internen Prozessen, die vom Master gesteuert werden. Diese zwei Klassen von rasdaman Prozessen wurden bezüglich ihrer Aufgaben adaptiert.

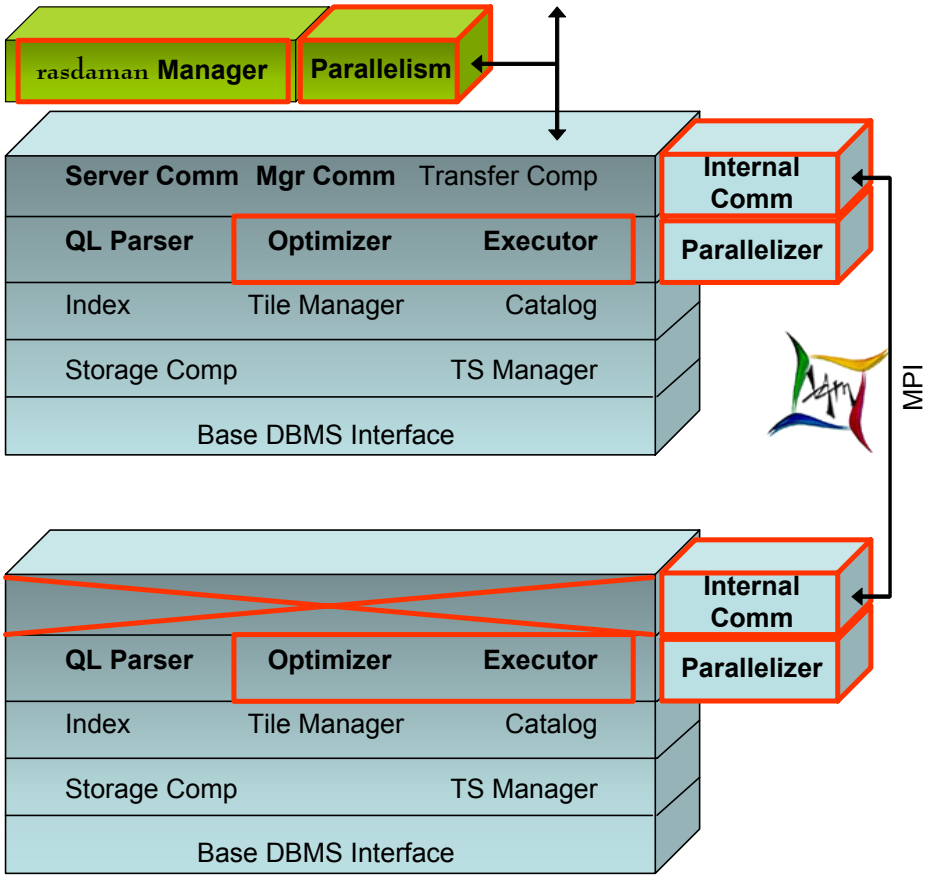


Abbildung 5.2: Modulare Schichtenarchitektur von Master und internen Prozessen

Abbildung 5.2 skizziert die Veränderungen in den Modulen des rasdaman Master Server (oben) und der internen Server Prozesse (unten). Die Kommunikation mit dem rasdaman Manager bzw. den Clients sowie der Datentransfer werden ausschließlich durch den Master realisiert, diese komplette Schicht darf von den internen Prozessen nicht genutzt werden. Die Kommunikation zwischen Master und internen Prozessen mittels LAM-MPI wurde als Modul *Internal Comm* integriert. In der zweiten Schicht, welche das Parsen, die Optimierung und die Ausführung von RasQL realisiert, wurden neben der Integration eines Parallelisierungsmoduls der Optimierer und vor allem das Ausführungsmodul (engl.: Executor) verändert. Insbesondere die internen Prozesse unterscheiden sich hier deutlich: die Anfrageauswertung beginnt nicht prinzipiell am Wurzelknoten, sondern wird vom Parallelisierungsmodul gesondert spezifiziert. Ferner wird die Ausführung nicht extern angestoßen, sondern immer vom Master Prozess initiiert. Neben dieser Integration von Parallelität in die Anfrageausführung mussten auch interne Mechanismen der Ausführung adaptiert werden, wie etwa Ausnahmebehandlung und Speicherverwaltung.

Abbildung 5.2 demonstriert die Implementierungsstrategie einer „Kompatibilität zu parallelen Entwicklungssträngen“. Vorhandene Module wurden sehr selektiv und nur wenn nötig verändert. Die unteren Schichten wurden hingegen bis auf wenige Fehlerbereinigungen komplett unverändert übernommen.

5.3.2 Kommunikation zwischen parallelen Prozessen

In diesem Kapitel wird die Kommunikation zwischen rasdaman Master und den internen Prozessen an zwei Beispielen skizziert. Prinzipiell müssen zwei Klassen von Nachrichten zwischen Master und den internen Server ausgetauscht werden:

1. Propagierung von Anforderungen an rasdaman zur Statusverwaltung
   In diese Klasse gehören etwa ein Starten oder Beenden des rasdaman DBMS, ein Öffnen oder Schließen einer Datenbasis, Transaktionsverwaltung (*commit* bzw. *rollback*), Anstoßen der Anfrageausführung, etc. Mit anderen Worten werden die Methoden der rasdaman Client API an die internen Prozesse (evtl. verändert) durchgereicht.
2. Nachrichten während der parallelen Verarbeitung einer Anfrage
   Diese dienen der Steuerung der parallelen Ausführung (zum Beispiel Nachrichten für die Methoden *open*, *next*, *close*, *reset* des Iteratorkonzepts) oder dem Transfer von Zwischenresultaten. Diese Nachrichten realisieren die parallele Verarbeitung einer Anfrage.

Die Inter-Prozess Kommunikation während der parallelen Anfrageausführung, also Punkt 2, wurde bereits in Kapitel 4 detailliert beschrieben. Im Folgenden werden wir den Start eines parallelen rasdaman Servers und die Initiierung einer parallelen Anfrageausführung als Beispiel für die erste Klasse zeigen.

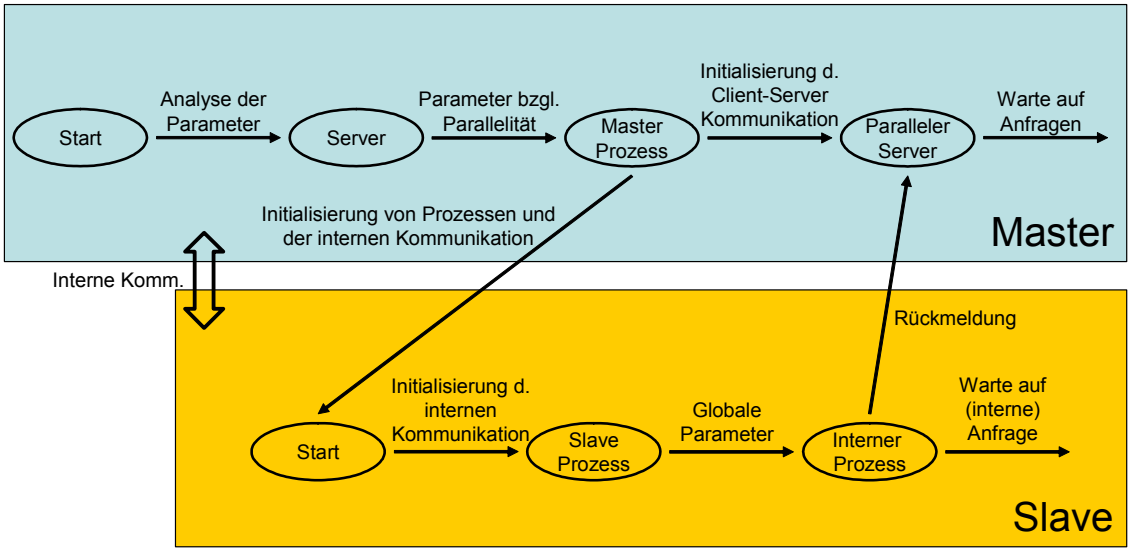


Abbildung 5.3: Start des parallelen rasdaman Server

Abbildung 5.3 zeigt die Zustände sowie die Kommunikation zwischen Master und internen Slave Prozessen während des Starts des parallelen rasdaman DBMS. Der koordinierende Master Prozess ist oben dargestellt, unten ist ein interner Prozess (Slave Prozess) angedeutet. Der Master Prozess erhält das Signal zum Start und wertet neben den üblichen Parametern auch parallele Parameter wie Anzahl der zu startenden Prozesse, das zu verwendende Modul

für die Inter-Prozess Kommunikation, Ausgaben für das Debugging der parallelen Verarbeitung, etc. aus. Nach dieser Analyse werden die internen Prozesse gestartet und die interne Kommunikation initiiert, erst dann initialisiert der Master Prozess eine Kommunikationsschnittstelle für die Verbindung zu Client Programmen. Signalisieren alle gestarteten internen Prozesse einen erfolgreichen Start, so steht der parallele rasdaman Server für Anfragen zur Verfügung.

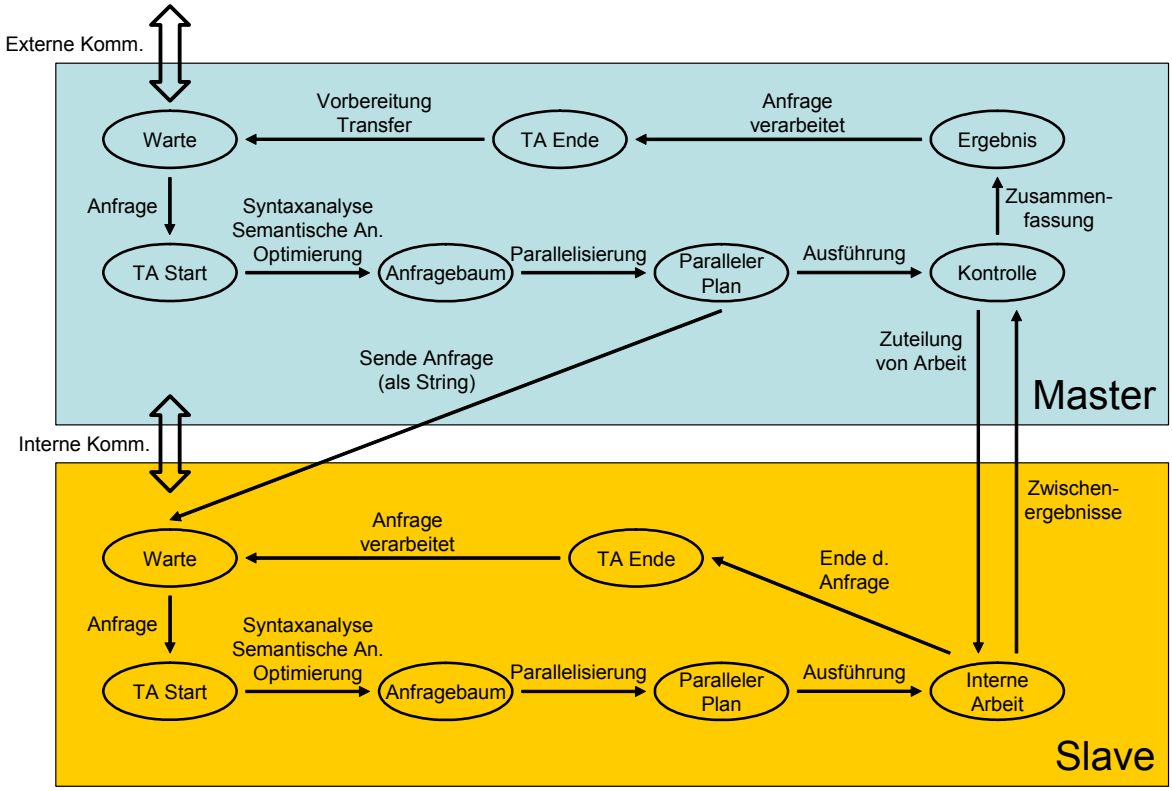


Abbildung 5.4: Anfrageverarbeitung im parallelen rasdaman Server

Die Verarbeitung einer Anfrage, die durch die Methode *execute_query()* der rasdaman API angestoßen wird, ist in Abbildung 5.4 dargestellt. Der Master nimmt die RasQL Anfrage entgegen, startet eine Transaktion, führt eine lexikalische, syntaktische und semantische Analyse sowie die Optimierung und Parallelisierung durch. Falls so ein korrekter paralleler Anfragebaum und Ausführungsplan erzeugt werden konnte, wird die Anfrage an die internen Prozesse transferiert. Hierbei hat sich gezeigt, dass der Transfer der RasQL Anfrage verbunden mit einer erneuten Erstellung des parallelen Plans bei den internen Prozessen deutlich performanter ist als der Transfer des Planes (und damit des parallelisierten Anfragebaumes) selbst[86]. Die parallele Verarbeitung selbst wird vom Master gesteuert, der die Zwischenergebnisse anschließend für einen Transfer zum Client aufbereitet und überträgt.

5.4 Zusammenfassung

In diesem Kapitel wurden Besonderheiten der Implementierung von rasdaman 5.0 PE kurz vorgestellt. Eine Voraussetzung der Implementierung war die Beibehaltung der Qualitäten des

[86] Der Anfragebaum entspricht einer stark verzweigten Struktur von Objekten der Sprache C++. Das „Einpacken“ einer solch komplexen Struktur vor dem Versand an andere Prozesse als Nachricht verursacht einen hohen Aufwand, ebenso die Restaurierung der ursprünglichen Baumstruktur, also das „Entpacken“ der Nachricht. Das Parsen von RasQL und der Aufbau des Anfragebaumes hingegen erfolgt sehr effizient. Oracle 10g nutzt einen identischen Mechanismus zum Transfer von Anfrageplänen, bezeichnet dies jedoch als eine „Zwischensprache SQL“ [CDG04].

ursprünglichen rasdaman DBMS, insbesondere Stabilität und Benutzerfreundlichkeit, bei einer gleichzeitigen Verbesserung der Performanz. Diese in **Kapitel 5.1** näher spezifizierten Prämissen folgen aus der Tatsache, dass rasdaman 5.0 PE im EU Projekt ESTEDI produktiv eingesetzt wurde und die Basis für eine spätere kommerzielle Version bilden soll.

In **Kapitel 5.2** wurden Alternativen des Designs vorgestellt, die sich auf die Methode der Implementierung, die unterstützen Zielplattformen und die Verwendung einer Inter-Prozess Kommunikation auswirken. Für die Implementierung wurde eine Strategie der Kapselung der Parallelität unter Beibehaltung der ursprünglichen Verarbeitungsalgorithmen gewählt, da nur so eine spätere Verschmelzung der unabhängigen Entwicklungsstränge verwirklicht werden konnte. Die Verwendung des Message Passing Interface als Methode der Inter-Prozess Kommunikation ermöglicht den Einsatz von rasdaman 5.0 PE auf einer Vielzahl von parallelen Zielplattformen (Multiprozessor, heterogene und homogene Cluster, NUMA, etc.) unter Nutzung einer Vielzahl von Kommunikationsmodulen (TCP, Shared Memory, Infiniband, Globus, Myrinet, etc.).

Abschließend wurde in **Kapitel 5.3** anhand von Beispielen ein kurzer Eindruck von der Integration interner paralleler Server in die Architektur und die Kommunikation gegeben.

Kapitel 6 Performanz der Implementierung

> It is a capital mistake to theorize before one has data.
> Insensibly one begins to twist facts to suit theories,
> instead of theories to suit facts.
> (Sir Arthur Conan Doyle)

Die Performanz einer parallelen Verarbeitung ist nur zum Teil auf gute Algorithmen und deren gute Implementierung zurückzuführen. Einen wesentlichen Anteil bei der Präsentation eines fast linearen Speed-Up hat die Wahl der richtigen Anfrage und einer für diesen Zweck optimierten Hardware. In diesem Kapitel soll eine „ehrliche“ Analyse zeigen, welche Anfragen und welche Hardwarearchitekturen inwiefern von Parallelität profitieren.

Eine kurze Vorstellung der Testdaten wird in Kapitel 6.1 gegeben. Ein Großteil der Daten stammt vom Max-Planck-Institut für Meteorologie in Hamburg, einem Partner des ESTEDI Projektes. Wir möchten uns für die Überlassung der Daten ausdrücklich bedanken. Die Analysen, die mit diesen Daten möglich sind, welche auf einer Simulationen der globalen Klimaentwicklung basieren, sind faszinierend (und gleichwohl in ihrem Ergebnis erschreckend) und haben die Analysemöglichkeiten von **rasdaman** eindrucksvoll bestätigt.

In Kapitel 6.2 werden die Kosten einer parallelen Verarbeitung von Daten den Kosten einer nicht-parallelen Verarbeitung gegenübergestellt. Allein diese Analyse zeigt die Möglichkeiten und Grenzen paralleler Verarbeitung, die in den folgenden Kapiteln 6.3 und 6.4 mit Performanzmessungen auf Multiprozessorarchitekturen und Rechnercluster bestätigt werden. Eine abschließende Bewertung der analysierten parallelen Rechnerarchitekturen erfolgt in Kapitel 6.5.

6.1 Beschreibung der Testdaten

Vor der Präsentation der Messergebnisse werden in diesem Kapitel die Messdaten in Kürze vorgestellt. Das DBMS **rasdaman** ermöglicht die Speicherung sowie einen effizienten Zugriff und eine effiziente Analyse von Rasterdaten beliebiger Dimensionalität. In der Praxis

– zumindest im Projekt ESTEDI – haben sich jedoch Einschränkungen sowohl der Dimensionalität als auch der Datenvolumina gezeigt, die teils aus menschlichen Präferenzen, teils aus technischen Randbedingungen rühren:

- die Dimensionalität der Arrays wurde im Projekt ESTEDI nie größer als 4 gewählt. Dies basiert auf menschlichen Präferenzen bei der Datenmodellierung: 3 Dimensionen für den Raum zuzüglich einer Dimension für die Zeitachse wird als „natürlich" angesehen. Die Integration weiterer diskretisierbarer Parameter (etwa Frequenzband, Temperaturverteilung, etc.) erfolgt meist nicht als zusätzliche Dimension, sondern als zusätzliche Variable.
- die Größe eines Arrays (Datentyp MDD) wird in der aktuellen Implementierung von **rasdaman** durch den verfügbaren virtuellen Speicher des Servers beschränkt, bei Analyse der Daten sogar auf einen Bruchteil. Alle Daten (Quellarrays und Zwischenresultate) müssen während eines Iterationsschrittes komplett in den Speicher passen, bei einer binär induzierten Operation etwa ist somit die Größe des Arrays auf ein Drittel des verfügbaren Speichers begrenzt. Diese Einschränkung basiert auf der Tatsache, dass Arrays für eine Analyse oder vor deren Übertragung in den Speicher geladen werden müssen und eine partielle Auslagerung auf Sekundärspeicher auf Basis von Kacheln nicht implementiert ist.

Die Testdaten sind aus diesen Gründen auf Dimensionalität 4 und auf eine Größe von etwa 1,4 GByte pro Array beschränkt.

6.1.1 Satellitenbilder (2D)

Ein primärer Einsatzbereich von **rasdaman** liegt in der Verwaltung von Satellitendaten. Oftmals wird hierbei keine Zeitdimension modelliert, da die Satellitenaufnahmen in unregelmäßigen zeitlichen Abständen erfolgen. Stattdessen werden die verschiedenen Aufnahmen als Kollektion gespeichert, der Zeitstempel der 2-dimensionalen Bilder wird über Metadaten im RDBMS an die OID des jeweiligen Datensatzes gebunden. Es sei erwähnt, dass diese Modellierung auf die schnelle Extraktion eines Datensatzes bezüglich eines Zeitstempels optimiert ist, Zeitreihenanalysen sind nur mit großem Aufwand möglich.

Für die Performanzmessungen wurden Satellitendaten des EOS Projekts der NASA [EOS03] in **rasdaman** eingefügt. Abbildung 6.1 zeigt ein 2-dimensionales Rasterbild der eingefügten Kollektion *sat_rgb100*, die insgesamt 100 Arrays beinhaltet. Die Daten zeigen Aufnahmen der Flutkatastrophe des Jahres 2002, die in großen Teilen von Mitteleuropa große Schäden anrichtete. Ein einzelnes Satellitenbild hat eine Domäne von 2550 mal 3300 Rasterpunkten mit jeweils einer Variablen für Rot-, Grün- und Blauwert und ist somit ca. 25 MByte groß, die gesamte Kollektion folglich etwa 2,5 GByte. Die Daten wurden mit einer regulären Kachelung der Ausdehnung [64, 64] gespeichert. Für verschiedene Aspekte der Messungen wurden ferner entsprechende komprimierte Datensätze als Kollektion *sat_rgb_comp100* und mit Basistyp *char* (Graustufenbild) als Kollektion *sat100* integriert.

Auf der linken Seite in Abbildung 6.1 sieht man die Darstellung der Daten mit dem Visualisierungswerkzeug rView, welches in **rasdaman** 5.0 integriert ist. Ein detaillierter Ausschnitt des Überschwemmungsgebiets ist rechts eingefügt.

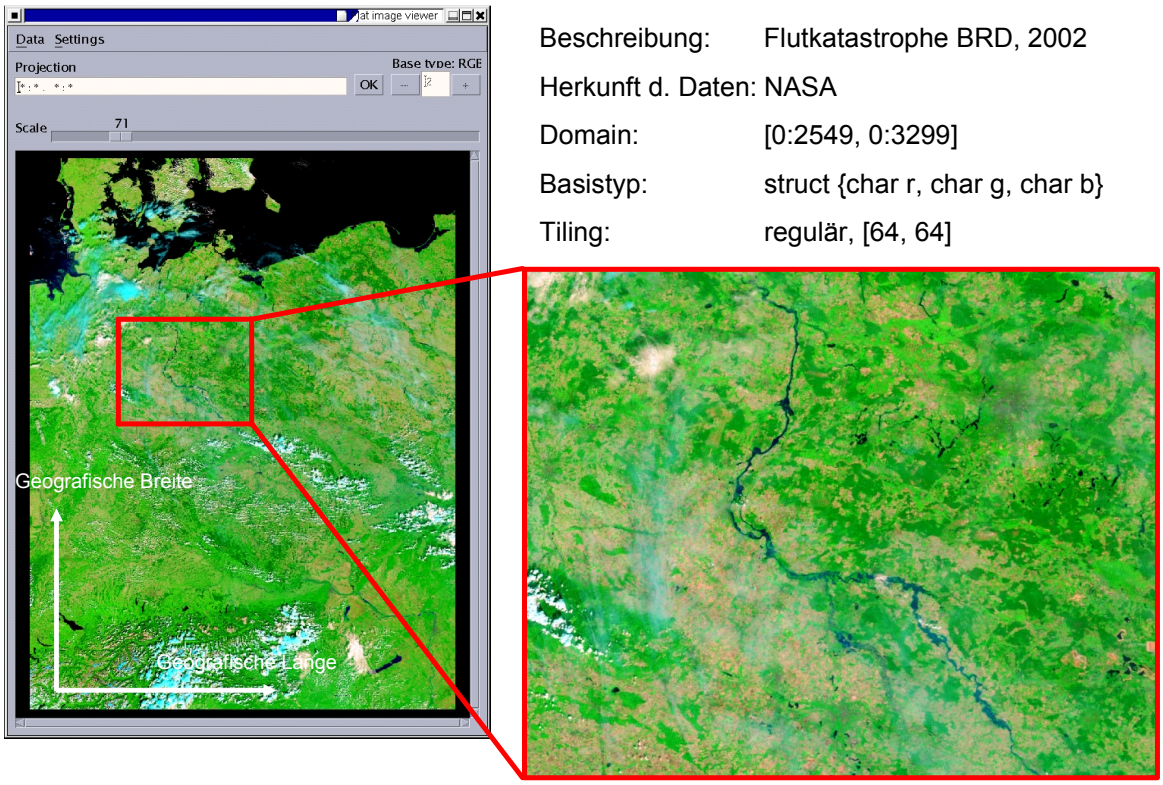


Abbildung 6.1: Visualisierung der 2D Testdaten (Satellitenbilder)

6.1.2 Simulation der globalen Erderwärmung (3D)

Der zweite Satz von Testdaten entstammt einer Klimasimulation des Max-Planck-Instituts für Meteorologie, Hamburg. Die daraus gewonnenen Daten bestätigen die globale Erderwärmung, die aktuell bereits eingesetzt hat und die nach diesen Berechnungen in der Zukunft deutlich zunehmen wird. Die einzelnen Zellwerte entsprechen der atmosphärischen Temperatur 2 Meter über der Erdoberfläche in Grad Kelvin, gespeichert als Gleitkommazahl einfacher Genauigkeit (Datentyp *float*). Die Dimensionen entsprechen der geografischen Länge und Breite diskretisiert auf ein Raster von 128 mal 64 Punkten, sowie der Zeitdimension mit einer Ausdehnung von 2880 Werten, einen Zeitraum von Januar 1860 bis Dezember 2099 abdeckend (also 240 Jahre mit jeweils 12 Monaten). Die Werte bis zum Jahr 1990 sind gemessene Temperaturen, ab 1991 entstammen sie einer Simulation, die teils auf dem IPCC92 Szenario, teils auf dem IPCC96 Szenario (an das Kopenhagen Protokoll angepasstes Klimamodell) basieren.

Abbildung 6.2 zeigt diverse Visualisierungen dieser Daten. Oben links sieht man einen 10-jährigen Teilbereich der Daten (Januar 2000 bis Dezember 2009). Die Zellwerte wurden auf ein Farbspektrum von komplett blau (RGB: 0/0/255) für eine Temperatur von 220°K (ca. -50°C) bis komplett rot (RGB: 255/0/0) für 320°K (ca.50°C) abgebildet. Auf der rechten Seite des 3-dimensionalen Würfels erkennt man die Temperaturverteilung mit heissen Bereichen nahe dem Äquator und kalten Bereichen nahe der Pole. Auf der linken Seite des Würfels lassen sich die jahreszeitlichen Schwankungen der Temperaturen im Laufe von 10 Jahren erkennen. In der Mitte der Abbildung sieht man eine Visualisierung des (gemessenen) Monats Dezember 1860 (Mitte links) und des letzten (simulierten) Monats Dezember 2099 sowie ermittelte Extrem- und Durchschnittswerte. Unten wird der Verlauf der Temperaturentwicklung für

einen ausgewählten Punkt der Erdoberfläche dargestellt (Zeitreihenanalyse durch Projektion durch die Zeitachse).

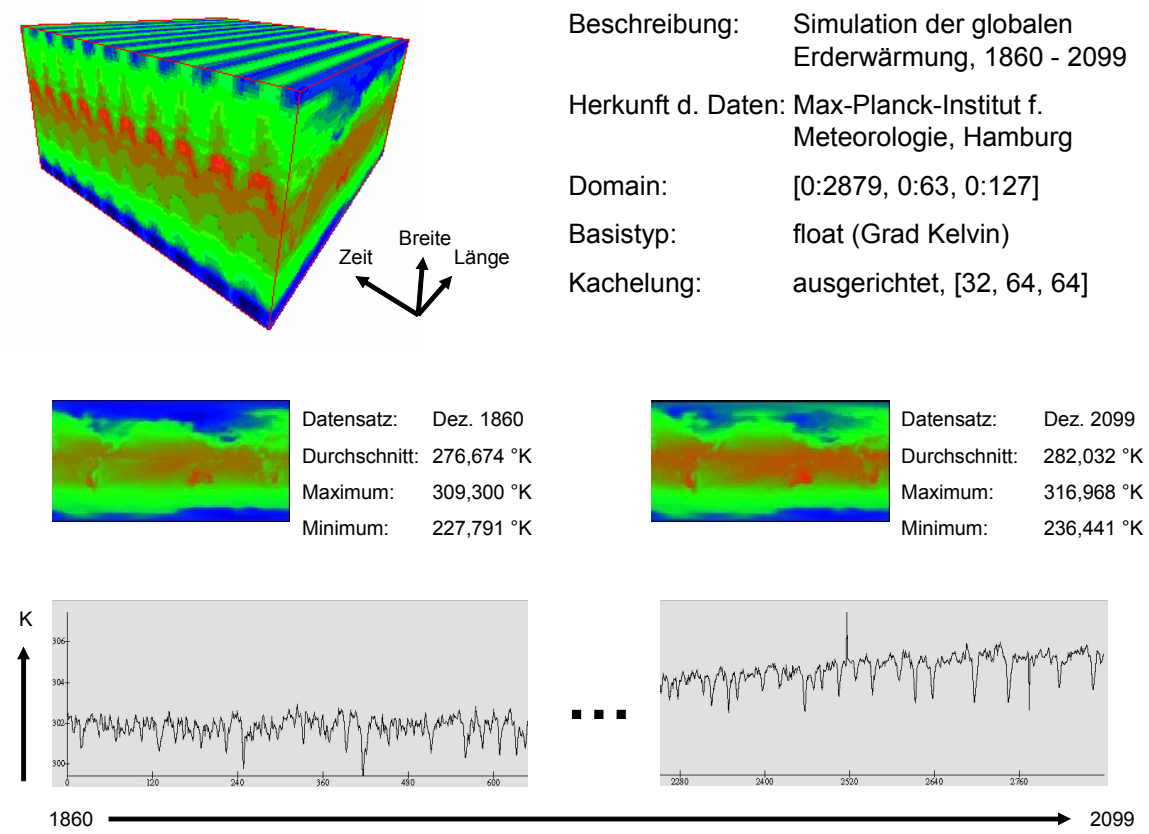


Abbildung 6.2: 3D Testdaten (Simulation der globalen Erderwärmung)

Die Daten sind je nach Messung gespeichert als

- eine einzelne Kollektion (Größe ist ca. 94 MByte),
- eine Kollektion von Objekten, die jeweils ein Jahrzehnt repräsentieren (jedes Array hat eine Größe von knapp 4 MByte),
- Kollektion mit oder ohne Kompression der Datenbasis.

Die Kachelung der Daten im RDBMS entspricht der vom Max-Planck-Institut ursprünglich gewählten Kachelung von 64 mal 64 für die geografischen Dimensionen und 32 für die Zeitdimension. Das Max-Planck-Institut hat für eine Verbesserung von E/A Datenrate bei Extraktion einzelner Monatsdaten diverse Experimente mit geringerer Kachelung entlang der Zeitdimension unternommen.

6.1.3 Simulation von Windverhältnissen nach einem Klimamodell (4D)

Der letzte Satz von Testdaten stammt wiederum vom Max-Planck-Institut für Meteorologie und entspringt der oben diskutierten Klimasimulation. Analysiert wurden hier jedoch nicht Temperaturen, sondern die Entwicklung von Windverhältnissen basierend auf dem oben beschriebenen Klimaszenario. Die vier Dimensionen entsprechen geografischer Länge, geografischer Breite, Höhenstufen und einer Zeitachse. Die Höhenstufen wurden bezüglich von at-

mosphärischem Druck auf 15 Werte diskretisiert (sind also nicht äquidistant), die Zeitachse umfasst die Monate Januar 1860 bis Dezember 2099.

Abbildung 6.3: Repräsentation eines Windvektors

Die einzelnen Werte der Simulation sind technisch gesehen Vektoren, bestehend aus einer Windrichtung und einer Windgeschwindigkeit. Die Repräsentation dieses Vektors erfolgt im Array DBMS rasdaman mittels Speicherung zweier Vektorwerte in orthogonaler Richtung.

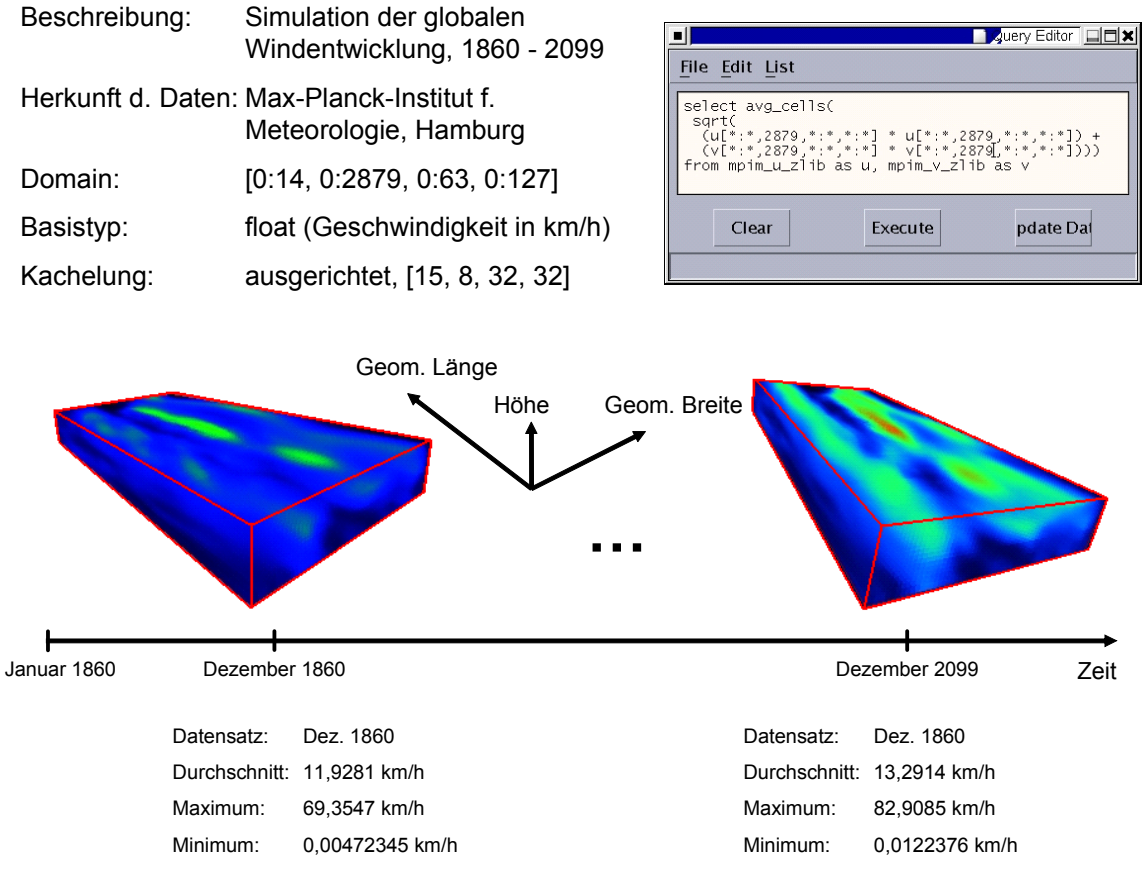


Abbildung 6.4: 4D Testdaten (Simulation der globalen Windgeschwindigkeiten)

Abbildung 6.3 skizziert diese Methode der Speicherung. Der Vektor wird durch zwei Werte u und v repräsentiert, entsprechend der Windgeschwindigkeiten parallel zu Äquator bzw. Meri-

dian. Die absolute Geschwindigkeit kann durch Berechnung nach dem Satz des Pythagoras ermittelt werden.

Die zwei Komponenten der Windvektoren wurden als zwei MDD von etwa 1,4 GByte Größe in zwei Kollektionen gespeichert, da viele Analysen auf nur einem Vektor basieren. Folglich kann durch diese Modellierung E/A reduziert werden. Als Kachelung wählte das Max-Planck-Institut ausgerichtete Kacheln der Größe [15, 8, 32, 32], eine Kachel hat also (in der Regel) die Größe 480KB.

Abbildung 6.4 zeigt eine Visualisierung der Daten. Die absoluten Windgeschwindigkeiten wurden für den ersten und den letzten Dezembermonat des Datensatzes aus den zwei Komponenten berechnet. Sowohl die Visualisierung als 3-dimensionale Würfel als auch die Analyse der Extrem- und Durchschnittswerte zeigt eine deutliche Zunahme von Windgeschwindigkeiten im Jahr 2099. Die Anfrage rechts oben zeigt die RasQL Anfrage zur Berechnung der absoluten Windgeschwindigkeit des Monats Dezember 2099.

6.2 Theoretische Analyse der Kosten einer Parallelverarbeitung

Die Kosten einer Anfrage akkumulieren sich im Falle einer nicht-parallelen Verarbeitung aus den Faktoren:

$$C_{query} = C_p + C_o + C_l + C_e (+C_t)$$

C_{query}	Kosten der Anfrage gesamt
C_p	Kosten für Parsing (syntaktische und semantische Analyse)
C_o	Kosten für Optimierung
C_l	Kosten für Laden der Quelldaten
C_e	Kosten für Evaluierung der Daten
C_t	Kosten für Transfer zum Client

Die Kosten für den Transfer von Datenbankserver zur Client Applikation sind abhängig von der Architektur (Client auf lokalem Rechner oder WWW Browser als Client Applikation), vom Netzwerk und vom eingesetzten Transferprotokoll (RPC, TCP). Für die Analyse der Performanz werden die Transferkosten im Folgenden nicht weiter berücksichtigt, da sie durch den DBMS Server nicht beeinflussbar sind.

Die parallele Verarbeitung einer Anfrage in unserem Sinne konzentriert sich auf eine parallele Datenanalyse, das heißt die Kosten der Evaluierung vermindern sich bei optimalen Algorithmen und Implementierung (d.h. bei steter Auslastung aller Prozesse) um einen Faktor p, der den Parallelitätsgrad darstellt. Die Kostenfaktoren C_p und C_o bleiben konstant, im Sinne von Amdahls Gesetz können sie jedoch als nicht-parallelisierbarer Anteil des zu parallelisierbaren Problems gesehen werden. Die Kosten C_l für E/A bei paralleler Verarbeitung hängen primär von der verwendeten Hardware, sekundär von der Anfrage ab. Bei Nutzung guter Sekundärspeicher (z.B. RAID Systeme) oder Replikation (und schneller Netzwerktechnologie bei nicht-lokalem RDBMS) können im optimalen Fall alle parallelen Instanzen die Daten gleichzeitig ohne gegenseitige Beeinflussung anfordern (folglich sind die Ladekosten C_l /p). In der Praxis wird bei den meisten Systemen eine Komponente C_{lo} hinzukommen, welche die Kosten für die gleichzeitige Benutzung von I/O durch die Prozesse und der damit einhergehenden Wartezeiten repräsentiert. Umso größer der Parallelitätsgrad p, umso größer wird die Wahrscheinlichkeit für eine spürbare Kostenkomponente C_{lo}. Anfragen, welche ein geringes Datenvolumen laden müssen, reduzieren in der Regel C_{lo}.

$$C^{p}_{query} = C_p + C_o + C_{pp} + C_{tp} + \frac{C_l}{p} + C_{lo} + \frac{C_e}{p} + C_{td}(+C_t)$$

C_{pp} Kosten für die Erstellung eines parallelen Planes
C_{tp} Kosten für den Transfer des parallelen Planes zu internen Prozessen
C_{lo} Overhead bei gleichzeitigem I/O mehrerer Prozessen
C_{td} Kosten für den Transfer von Daten zwischen Prozessen

Weitere Kosten, die nur bei paralleler Verarbeitung auftreten, sind Kosten für die Erstellung und für den Transfer eines parallelen Ausführungsplanes (im Sinne von Kapitel 3.1 Initialisierungskosten der Parallelität) sowie Kosten für den Transfer von Daten zwischen parallelen Instanzen (nach Kapitel 3.1 Kommunikationskosten der Parallelität). Ziel einer guten parallelen Implementierung ist es, diese Zusatzkosten gering zu halten und somit einen guten Speed-Up zu ermöglichen. Die in Kapitel 4 entwickelten Algorithmen sichern das durch einen effektiven Algorithmus zur Erstellung eines parallelen Planes und einer durch diesen Plan realisierten Minimierung des Datentransfers.

Messungen bestätigen die Effizienz der entwickelten Algorithmen. Kosten für die Erstellung eines parallelen Plans bewegen sich typischerweise im Bereich von 10 bis 50 ms, unabhängig von der Größe der zu evaluierenden Daten, und sind gegenüber den Analysekosten vernachlässigbar. Die Kosten der Kommunikation zwischen Prozessen hängen von der Anfrage und den Datenvolumen ab, sind jedoch soweit möglich reduziert und für typische Anfragen mit einer finalen Reduktion des Datenvolumens gering.

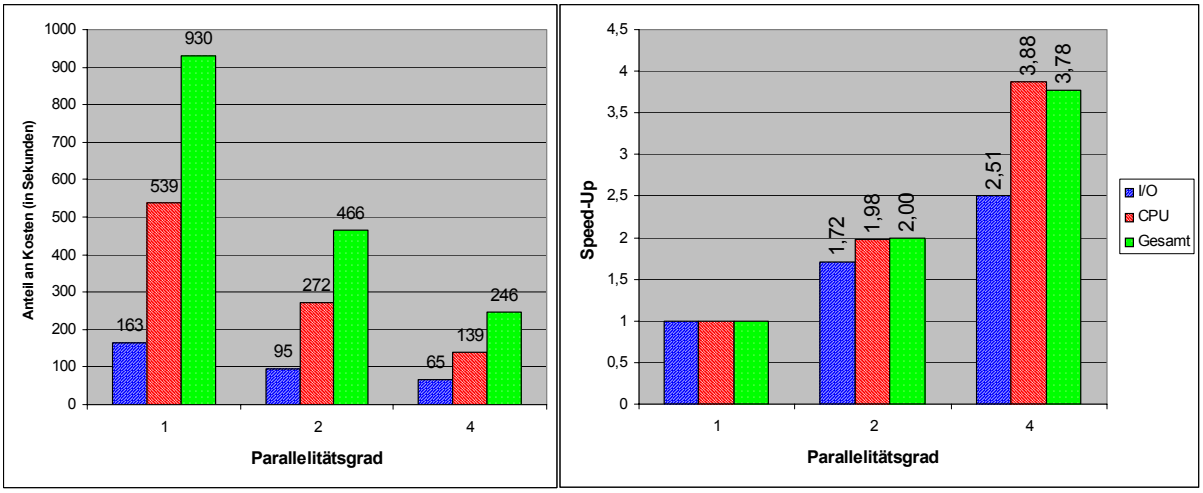


Abbildung 6.5: Parallelisierung der Kosten für E/A und CPU einer Anfrage

Wir wollen diese theoretische Kostenbetrachtung anhand eines Beispiels erläutern. Die Anfrage

```
SELECT (add_cells(a*(a>50))) / (count_cells(a>50))
FROM sat_small100 AS a
WHERE max_cells(a) > 200
```

berechnet nach einer Selektion bezüglich des Maximums den Durchschnittswert aller Zellen, die einen Grenzwert von 200 überschreiten. Für Satellitendaten kann diese Anfrage etwa eine Analyse von Überschwemmungsgebieten realisieren.

Abbildung 6.5 schlüsselt die Kosten der Anfrage nach E/A und CPU auf[87], gemessen auf einem Multiprozessorrechner mit 4 CPU. Der Zugriff auf die Quelldaten im RDBMS erfolgt über Standard 100 MBit Netzwerk. Die Abszisse bezeichnet den Parallelitätsgrad, die Ordinate die Kosten in Sekunden bzw. den Speed-Up. Der Berechnungsanteil (CPU) der Anfrage kann vollständig parallelisiert werden, bei E/A zeigt sich bei Parallelitätsgrad 4 eine eingeschränkte Skalierbarkeit und somit ein sublinearer Speed-Up. Die parallelen Zusatzkosten sind (bis auf C_{lo}) in dieser Grafik nicht darstellbar, da sie zu gering sind. Bei Nutzung eines Parallelitätsgrades von 4 wird der Speed-Up auch insgesamt geringfügig sublinear, da durch den Zugriff von 4 Prozessen auf die Quelldaten eine Reduktion von E/A erfolgt.

In den folgenden Kapiteln werden die Möglichkeiten und Grenzen der Parallelisierung bezüglich der parallelen Architekturen Multiprozessor und Cluster analysiert, jeweils für Inter-Objekt und Intra-Objekt Parallelisierung. Einflussfaktoren wie eine Kompression der Datenbasis, Art der Anfragen und Variationen der Architektur werden in ihrem Einfluss auf Parallelisierung gesondert diskutiert.

6.3 Multiprozessor-Architekturen

Die parallele Verarbeitung von Analysen auf Mehrprozessorrechnern war ein primäres Szenario im Projekt ESTEDI. Diese Architektur ist insbesondere für Server Rechner typisch, die Anzahl der Prozessoren beträgt meist 2,4,8 oder 16. Multiprozessorrechner mit mehr als 16 Prozessoren sind ausschließlich im Hochpreissegment anzutreffen und werden nur von wenigen Herstellern angeboten

Erfahrungen im Projekt ESTEDI haben gezeigt, dass für das rasdaman DBMS nie eine reine Shared-Everything Architektur (im Sinne einer lokalen Datenbasis) gewählt wurde. Die Architektur von rasdaman mit einem eigenen RDBMS für die Datenhaltung hat dazu geführt, dass in allen Fällen ein bereits vorhandener RDBMS Server genutzt wurde. Diese typische Architektur wird durch den Aufbau der Messumgebung im folgenden Kapitel reflektiert.

6.3.1 Messumgebung

Für die Analyse der Performanz von rasdaman 5.0 PE auf Multiprozessor-Architekturen wurde eine SUN Fire 880 mit 8 CPU, jeweils 900MHz, und 16 GB Hauptspeicher gewählt. Dieser Hochleistungsserver bedient eine große Anzahl von SUN Rays (Terminals ohne eigene Rechenleistung), die von den Studenten der Technischen Universität München genutzt werden. Dieser Hochleistungsrechner stand nachts für Messungen zur Verfügung.

Für die Speicherung der Quelldaten in einem RDBMS wurde die im Projekt ESTEDI ausschließlich eingesetzte Architektur eines eigenständigen RDBMS Server realisiert. Als RDBMS Server dient eine SUN Enterprise 450, ausgebaut mit 2 CPU zu je 450 MHz, 1GB Hauptspeicher und einem RAID Level 0 Festplattenverbund. Insbesondere die Verbindung von RDBMS Server und rasdaman Server mittels langsamen 100 MBit Netzwerk stellt einen (wohl kalkulierten) potentiellen Flaschenhals der parallelen Verarbeitung dar. Dieser Flaschenhals wurde jedoch absichtlich beibehalten, um die Skalierbarkeit dieser typischen Archi-

[87] Als CPU Kosten sind hier die Kosten für die Analyse der Arrays durch Array Operationen bezeichnet. Kosten für Indexzugriffe, Transaktionsbehandlung, etc. sind in der nicht in CPU Kosten enthalten, wurden jedoch in den Gesamtkosten berücksichtigt.

tektur zu demonstrieren. Bereits an dieser Stelle sei angemerkt, dass durch eine schnelle Netzwerkverbindung (Gigabit Ethernet, Infiniband) eine deutliche Verbesserung von E/A und damit der Skalierbarkeit erreicht wird.

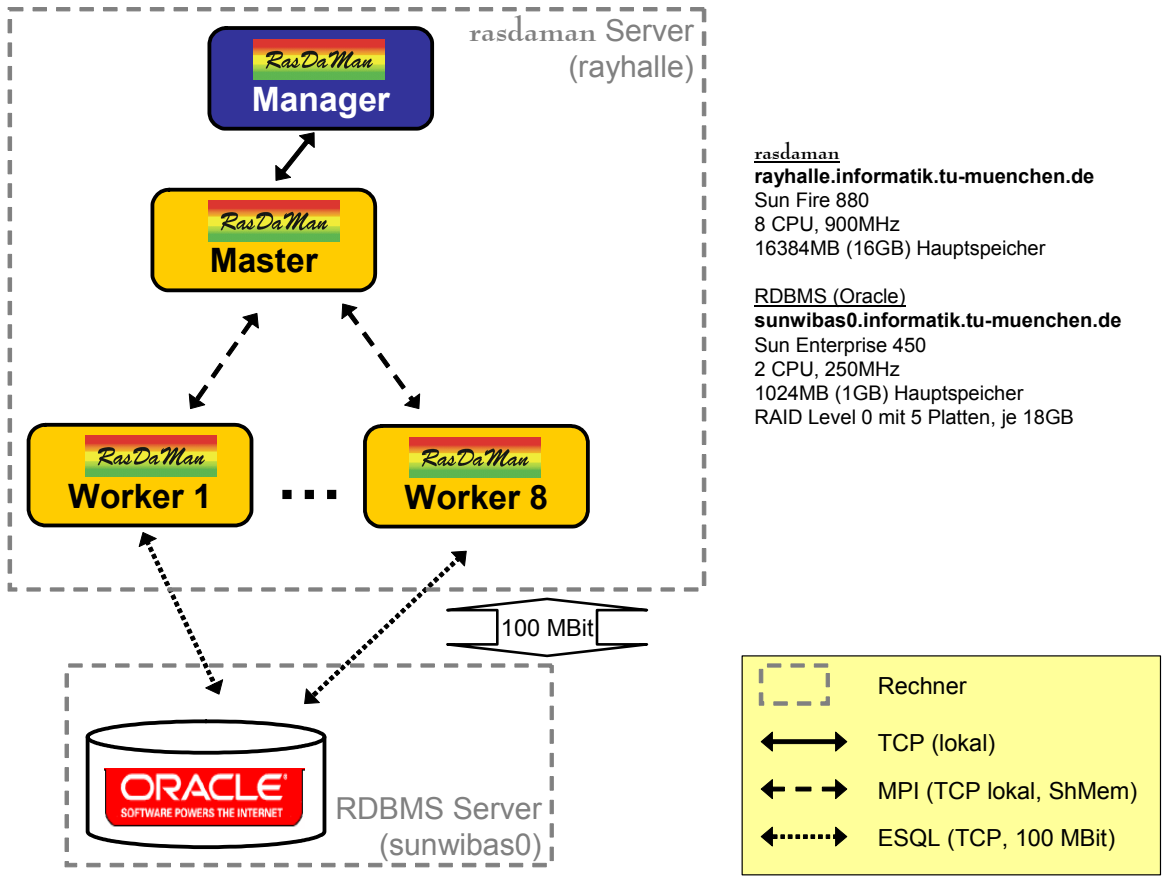


Abbildung 6.6: Shared-Everything Architektur mit nicht-lokalem RDBMS Server (Messumgebung)

Abbildung 6.6 skizziert die Messumgebung für Shared-Everything Architekturen. Der rasdaman Manager und der rasdaman Server, bestehend aus Master Prozess für die Kommunikation und Verteilung der Arbeitslast und Worker Prozessen für die parallele Verarbeitung der zugeteilten Arbeitspakete, sind auf dem 8-Prozessor Rechner lokalisiert. Kommunikation von Manager und Server erfolgt über TCP (lokal), ebenso die Kommunikation zwischen rasdaman Server und Oracle (TCP über 100 MBit Ethernet). Die Kommunikation der internen rasdaman Prozesse wurde durch Nutzung der entsprechenden LAM Module für TCP (lokal) und gemeinsamen Speicher (Shared Memory) gestestet.

6.3.2 Inter-Objekt Parallelisierung

Die erste Messung präsentiert den Speed-Up für eine typische rechenintensive Analyse auf Array Daten. Die Anfrage referenziert die in Kapitel 6.1.1 vorgestellte Kollektion von Satellitendaten einer Überschwemmungskatastrophe, in diesem Fall als Grauwertbilder gespeichert und durch die Methode der Haar-Wavelet Kompression in ihrem Datenvolumen reduziert. Die Kollektion beinhaltet 100 Arrays, folglich wird für die parallele Verarbeitung die Strategie der Inter-Objekt Parallelisierung gewählt. Die RasQL Anfrage lautet im Detail:

```
SELECT (add_cells(a[0:500,0:500]*(a[0:500,0:500]>50))) /
                (count_cells(a[0:500,0:500]>50)
FROM sat100_haar AS a
WHERE max_cells(a[0:500,0:500]) > 100
```

Die Anfrage analysiert einen bestimmten Bereich von Interesse, welcher [0:500, 0:500] umfasst. Die Selektion wählt alle Arrays mit einem Maximum über 100 in diesem Bereich, diese dunklen Gebiete können potentielle Überschwemmungsgebiete kennzeichnen. Die resultierenden Arrays werden auf ihren durchschnittlichen Grauwert geprüft. Die Filterung von hellen Bereichen, die etwa auf Wolken deuten können und somit für die Analyse nicht von Bedeutung sind, erfolgt über eine unär induzierte Operation „>“. Diese Filterung verursacht die Berechnung des Durchschnitts über Summe und Zellenanzahl, statt über die dafür vorgesehene und optimierte Aggregationsfunktion *avg_cells*.

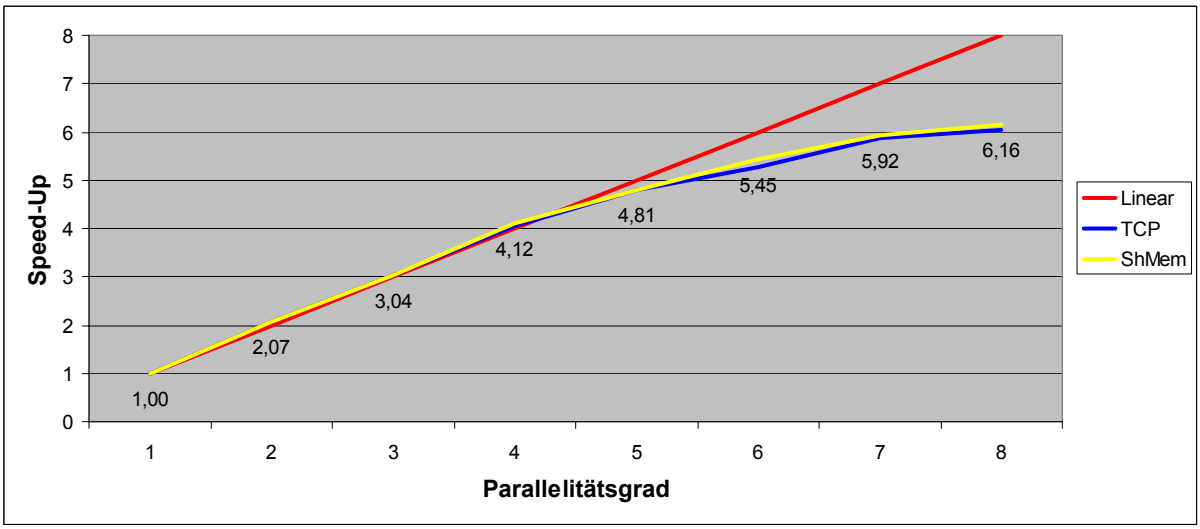


Abbildung 6.7: Speed-Up von rasdaman PE unter Nutzung einer Shared-Everything Architektur mit Zugriff auf RDBMS Server über 100MBit Netzwerk

Abbildung 6.7 präsentiert den theoretisch möglichen und gemessenen Speed-Up dieser Anfrage. Ein optimaler linearer Speed-Up wird bis zu einem Parallelitätsgrad von 5 erreicht[88]. Ab einem Parallelitätsgrad von 5 kann zwar eine weitere deutliche Steigerung der Performanz gemessen werden, jedoch zeigt sich der weitere Anstieg leicht sublinearer. Dies ist durch den Kostenfaktor C_{lo} zu erklären, das heißt, die Anforderung der Quelldaten von 6 und mehr Prozessen parallel lässt das 100 MBit Netzwerk an die Grenzen der Übertragungsrate stoßen. Die Kosten für die Erstellung C_{pp} und den Transfer des parallelen Plans C_{tp} zu den internen Prozessen ist (bedingt durch den Algorithmus) äußerst gering und kann den Performanzverlust nicht erklären. Die Kosten C_{td} für den Transfer von Zwischenergebnissen werden durch den Parallelisierungsalgorithmus minimiert, für die vorgestellte Anfrage werden allein persistente MDD und Skalare, also geringe Datenmengen, zwischen den Prozessen übertragen. Ein Fehlen des Einflusses von C_{td} wird durch die Nutzung zweier LAM Kommunikationsmodule verifiziert. In der Abbildung ist zu erkennen, dass die Kommunikation über TCP (lokal) und gemeinsamen Speicher (ShMem, engl.: shared memory) eine nahezu identische Performanz

[88] Der leicht überlineare Speed-Up kann hier noch mit Messungenauigkeiten begründet werden. Eine allgemeine Analyse des häufig zu beobachtenden Phänomens erfolgt auf Seite 183.

zeigt. Beim Transfer großer Datenmengen bietet die Kommunikation über gemeinsamen Speicher natürlich Vorteile[89].

Ein effektiverer Zugriff auf die Quelldaten, entweder durch ein lokal installiertes RDBMS oder durch eine schnelle Netzwerkanbindung (Infiniband, Gigabit Ethernet), sollte für diese Anfrage auf diesem Rechner einen komplett linearen Speed-Up zeigen.

6.3.2.1 Einflussfaktor Anfragekomplexität

Wir haben bisher demonstriert, dass für eine typische Anfrage auf einem Multiprozessorrechner mit 8 CPU ein fast linearer Speed-Up zu erreichen ist. Eine leicht abnehmende Tendenz ist ab einem Parallelitätsgrad von 6 zu identifizieren, da hier durch die Messumgebung eine effektive Versorgung der Prozesse mit den Quelldaten nicht mehr gewährleistet ist.

Im Folgenden werden Einflussfaktoren analysiert, die den Speed-Up einer Anfrage für diese Architektur positiv oder negativ beeinflussen.

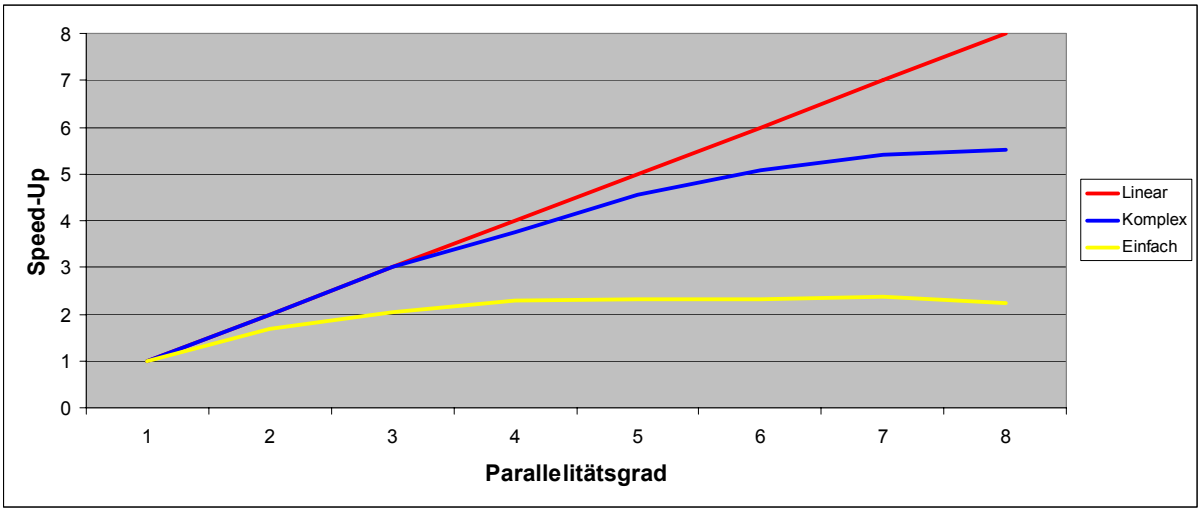


Abbildung 6.8: Einfluss der Anfragekomplexität auf den Speed-Up (Shared-Everything)

Abbildung 6.8 stellt zwei Anfragen mit unterschiedlicher Komplexität gegenüber. Die Analyse ist bis auf eine mittels ZLib komprimierte Datenbasis identisch zu der Abbildung 6.7 zugrunde liegenden Anfrage.

```
SELECT (add_cells(a[0:500,0:500]*(a[0:500,0:500]>50))) /
                (count_cells(a[0:500,0:500]>50))
FROM sat100_c AS a
WHERE max_cells(a[0:500,0:500]) > 100
```

Die unterschiedliche Komplexität wurde simuliert, indem die Selektionsbedingung variiert wurde, so dass eine Selektivität von 0% bzw. 100% gegeben war. Die einfache Anfrage testet

[89] Dennoch wird für die folgenden Messungen (bis auf die Anfrage, die zu Abbildung 6.10 führt) für Shared-Everything Architekturen eine Inter-Prozess Kommunikation über TCP eingesetzt, da hierdurch die Erweiterbarkeit durch Integration weiterer Rechner in optimaler Weise gesichert wird.

somit für jedes MDD lediglich das Selektionsprädikat, während die komplexe Anfrage ebenso jeweils die Applikation ausführt. Im Gegensatz zur Anfrage mit einer Selektivität von 100%, die den bereits präsentierten Speed-Up annähernd erreicht, zeigt sich bei der einfachen Anfrage keine Steigerung der Performanz ab einem Parallelitätsgrad von 4. Die einfache Anfrage wird durch E/A dominiert, der parallelisierbare Kostenfaktor C_e hat keinen entscheidenden Anteil an der Verarbeitung.

6.3.2.2 Einflussfaktor Kommunikationsaufwand

Ein wesentlicher Aspekt der in Kapitel 4 entwickelten Algorithmen war die Vermeidung von Inter-Prozess Kommunikation. Der Transfer von Zwischenergebnissen zwischen Prozessen bedeutet nicht nur für über Netzwerk verbundene Cluster eine potentielle Gefährdung des Speed-Up, sondern auch für Multiprozessorrechner[90]. Typische Anfragen reduzieren in einer finalen Aggregation oder Skalierung das zum Client zu übertragende Datenvolumen, was durch den Parallelisierungsalgorithmus ebenso zu einer Reduktion der Inter-Prozess Kommunikation führt. Manchmal werden jedoch die analysierten Daten auch komplett angefragt, vor allem, wenn die Client Anwendung auf dem Serverrechner installiert ist oder bei Nutzung einer extrem schnellen Netzwerkverbindung in einem LAN. Wir zeigen den Einfluss einer für solche Anfragen nötigen Inter-Prozess Kommunikation mit Transfer großer Datenmengen. Die Anfrage im Detail lautet

```
SELECT a - 273.15
FROM mpim32_c AS a
WHERE max_cells(a - 273.15) > 40
```

bei einem kompletten Datentransfer. Eine Applikation *avg(a – 273.15)* realisiert ein komplett aggregiertes Resultat und *scale((a – 273.15), 0.2)* eine Skalierung der zu übertragenden Datenmenge um den Faktor 5. Die Anfrage besitzt gegenüber bisherigen Beispielen eine deutlich geringere Komplexität.

Aus der bereits bekannten Kollektion der Kardinalität 32, die 3-dimensionale Arrays einer Klimasimulation mit einem Basisdatentyp *float* enthält, werden die Datensätze mit einer maximalen Temperatur über 40°C selektiert[91]. Je nach Anfrage wird das Resultat als unverändertes Array, als skaliertes Array (Dimensionsausdehnungen um den Faktor 5 reduziert) oder nur eine charakteristische Größe des Arrays, hier etwa der Durchschnittswert, ausgegeben. Man beachte, dass sich die Komplexität der Anfrage durch die Array Operationen Aggregation bzw. Skalierung natürlich zusätzlich erhöht.

[90] Vor allem verursacht durch den zusätzlichen Aufwand für Sperrmechanismen und einer potentiellen Überlastung des Systembusses.

[91] Die unär induzierte Subtraktion realisiert eine Umrechnung von Kelvin nach Celsius.

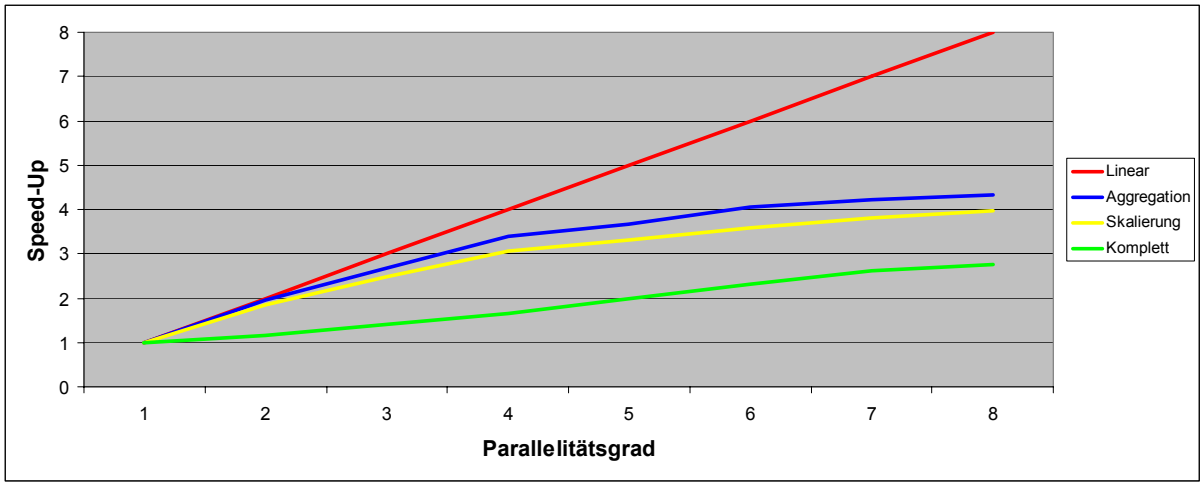


Abbildung 6.9: Einfluss des Datenvolumens der Inter-Prozess-Kommunikation (Shared-Everything)

Abbildung 6.9 zeigt den Speed-Up für Anfragen mit einem aggregierenden oder skalierenden finalem Schritt bzw. bei Transfer der kompletten Daten zur Client Anwendung[92]. Vor allem die Nutzung einer Inter-Prozess Kommunikation basierend auf TCP, die eine Erweiterbarkeit durch Integration weiterer Rechner sehr einfach macht, führt zu einer schlechten Skalierbarkeit, falls komplette Resultate angefragt werden. Die Nutzung von gemeinsamen Speicher als Kommunikationsmethode zeigt einen deutlich besseren Verlauf der Speed-Up Kurve (Abbildung 6.10), jedoch wird auch damit keine annähernd lineare Entwicklung erreicht.

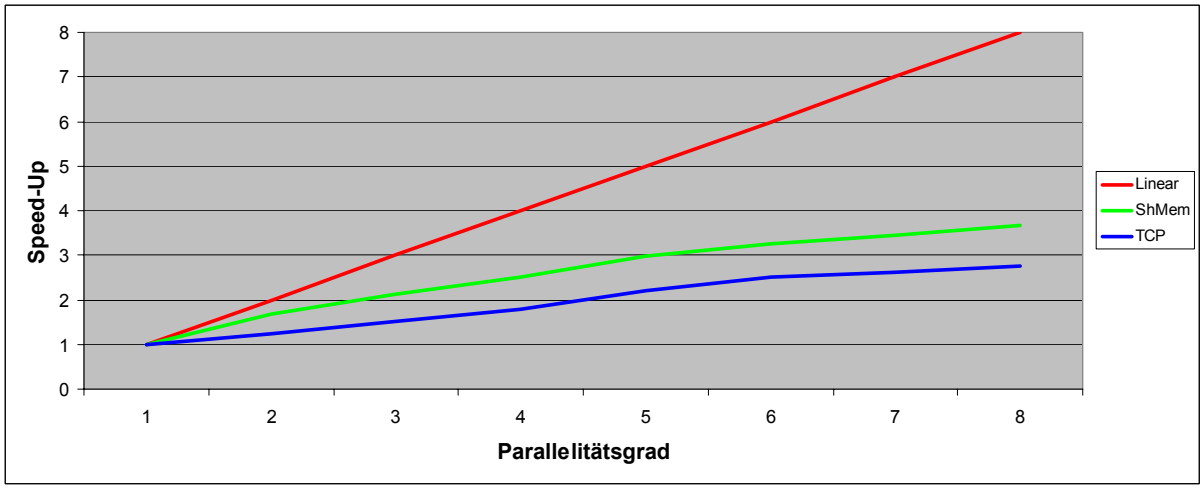


Abbildung 6.10: Einfluss des Kommunikationsmoduls für die Inter-Prozess-Kommunikation (Shared-Everything)

Die Entscheidung, ob auf einem Multiprozessorrechner besser auf TCP oder auf gemeinsamen Speicher basierende Kommunikation eingesetzt wird, wird vor allem durch die Art der Client Anwendungen bestimmt. Erfolgt die Analyse der Daten durch eine Anwendung lokal am Server oder an einem durch Hochleistungs-LAN verbundenen Rechner, sind Auswertun-

[92] Der insgesamt sublineare Speed-Up basiert auf der geringeren Komplexität der Anfrage gegenüber den Beispielen zuvor.

gen mit großen Datenmengen als Resultat möglich und wahrscheinlich. In diesem Fall sichert eine auf gemeinsamen Speicher basierende Kommunikation eine bessere Skalierbarkeit, schränkt jedoch die Erweiterbarkeit deutlich ein. Werden die Daten hingegen über WAN analysiert (z. B. mittels einem Browser über das WWW), ist eine deutliche Reduktion der zu übertragenden Datenmenge unumgänglich. In diesem Falle wird sich eine auf TCP basierende Inter-Prozess Kommunikation weniger deutlich auf die Performanz auswirken. Die Erweiterbarkeit hingegen ist entgegen einer Shared Memory Kommunikation durch Integration weiterer Rechner (entspricht dem Ausbau zu einem Cluster) einfach realisierbar.

In der Implementierung von rasdaman 5.0 PE wurde auf diese verschiedenen Einsatzszenarien besondere Rücksicht genommen. Die Entscheidung für ein Kommunikationsmodul wird bei der Definition des rasdaman Servers getroffen. So können auch beide Arten von rasdaman Server gleichzeitig definiert und gestartet werden. Ein gut skalierbarer Server mit effektiver Inter-Prozess Kommunikation kann somit im Einsatz sein, während ein Server mit optimaler Erweiterbarkeit nur bei Bedarf gestartet wird.

6.3.2.3 Einflussfaktor Datenkompression

Das Array DBMS rasdaman ermöglicht eine effiziente Analyse von extrem großen Datenmengen. Die Speicherung dieser Datenmengen verursacht häufig hohe Kosten, vor allem Kosten für Sekundärspeicher. Um diesen Kostenfaktor zu reduzieren, können zwei Strategien herangezogen werden: entweder werden die Daten durch Kompressionsverfahren in ihrem Speichervolumen reduziert oder die Daten werden auf im Vergleich zu Sekundärspeicher billigerem Tertiärspeicher ausgelagert. Beide Verfahren werden durch rasdaman optimal unterstützt. rasdaman erlaubt eine automatische Kompression der Datenbasis, unterstützte Kompressionsverfahren beinhalten Lauflängencodierung, ZLib und Wavelet Algorithmen [Deh02]. Ebenso erlaubt rasdaman eine direkte Speicherung auf Tertiärspeicher sowie ein optimiertes automatisiertes Laden [Rei05].

Während Tertiärspeicherzugriff die Kosten für E/A deutlich erhöht und somit aus Sicht der parallelen Verarbeitung eher einen negativen Effekt hat, ist die Dekompression der Daten, die vor einer Analyse nötig ist, einerseits eine rechenintensive Aufgabe und wird durch die parallele Verarbeitung voll unterstützt, andererseits reduziert sie E/A und entschärft dadurch den E/A Flaschenhals.

Abbildung 6.11 zeigt den Verlauf des Speed-Up für eine identische (mäßig komplexe) Anfrage. Die referenzierte Kollektion ist bezüglich der Kardinalität und des Dateninhalts identisch, wurde jedoch in unkomprimierter Form, sowie mit den Kompressionsverfahren ZLib und (Haar) Wavelet gespeichert. Laut [Deh02] sind diese Kompressionsverfahren die für multidimensionale Rasterdaten sinnvollsten Varianten für eine Reduktion des Volumens der Datenbasis. Diese Algorithmen gehören zu den verlustfreien Kompressionsmethoden. ZLib zeichnet sich vor allem durch eine hervorragende Kompressionsrate bei einer schnellen Dekompression aus, zeigt jedoch eine Asymmetrie bei Kompressions- und Dekompressionsraten, eine Kompression ist somit sehr teuer. Bei Wavelets ist das Verhältnis hingegen symmetrisch, das heißt die Kompression ist in der Regel schneller als bei ZLib, die Dekompression erfordert mehr Rechenaufwand. Das Projekt ESTEDI hat gezeigt, dass das Einfügen der Datenvolumen in rasdaman teilweise mehrere Wochen und Monate gedauert hat, folglich ist eine schnelle Kompression (durch Wavelets) durchaus eine Option gegenüber ZLib.

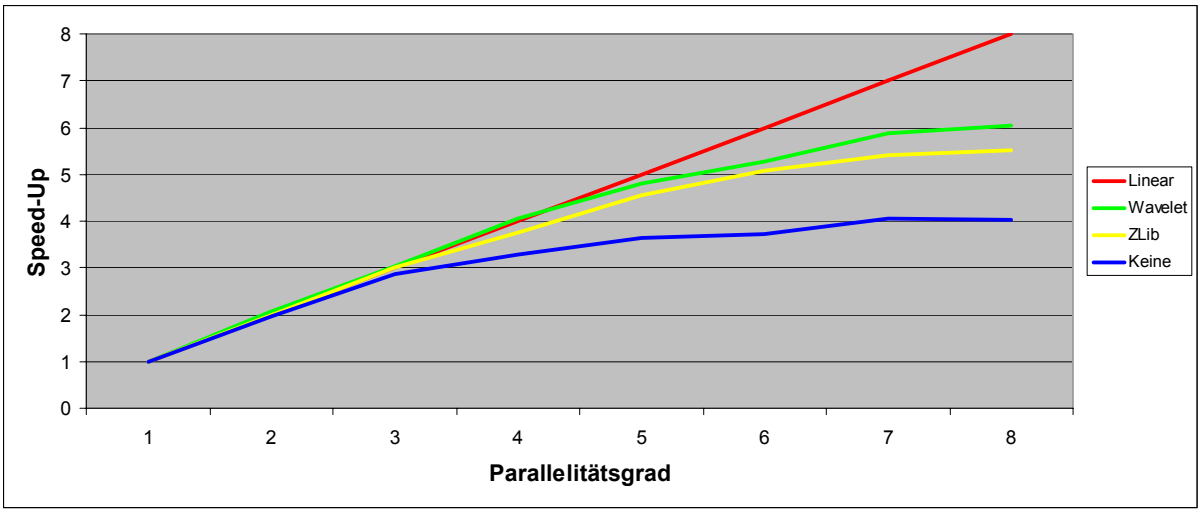


Abbildung 6.11: Einfluss der Datenkompression auf den Speed-Up (Shared-Everything)

Eine Kompression erhöht den Kostenfaktor C_e für die Dekompression. Die Dekompression wird durch die in Kapitel 4 entwickelten Algorithmen vollständig parallel ausgeführt. Folglich führt eine Datenkompression zu einer Verbesserung des Speed-Up, wobei eine komplexe Dekompression besser durch höhere Kosten C_e skaliert.

6.3.3 Intra-Objekt Parallelisierung

Die bisher präsentierten Messergebnisse beschränkten sich auf die primär eingesetzte Parallelisierungsstrategie der Inter-Objekt Parallelisierung, wie in Kapitel 4.3 beschrieben. Das heißt, die Kardinalität der Anfragen war in jedem Fall höher oder gleich dem Parallelisierungsgrad und durch den Parallelisierungsalgorithmus wurde eine parallele Verarbeitung von kompletten Arrays realisiert. Ergibt die Analyse einer Anfrage eine Kardinalität kleiner als dem Parallelisierungsgrad, kann diese Strategie die Ressourcen nicht vollständig ausnutzen, der Algorithmus wählt daher eine parallele Verarbeitung von Teilbereichen der Arrays nach Kapitel 4.4 (Intra-Objekt Parallelisierung).

Die Anfrage referenziert die in Kapitel 6.1.3 beschriebenen 4D Daten, genauer die Geschwindigkeiten der Winde parallel zum Äquator in der nördlichen Hemisphäre in den Jahren 1860 bis 1889. Nach einer Verifikation, dass in diesem Bereich eine maximale Windgeschwindigkeit über 50 km/h existiert[93], wird die Durchschnittsgeschwindigkeit dieser Luftströmungen berechnet.

```
SELECT (add_cells(u[*:*,0:359,0:31,*:*]*(u[*:*,0:359,0:31,*:*]>50))) /
                    (count_cells(u[*:*,0:359,0:31,*:*]>50))
FROM mpim_u AS u
WHERE max_cells(u[*:*,0:359,0:31,*:*]) > 50
```

Eine solche Analyse kann die Charakteristika von extremen Bereichen bestimmen, in diesem Beispiel etwa die Heftigkeit der Jet Streams in der nördlichen Hemisphäre.

[93] Mit dieser Selektion kann eine zeitintensive Analyse in der Applikation verhindert werden, falls solche Geschwindigkeiten gar nicht auftreten.

Die Skalierbarkeit dieser Anfrage, die durch Intra-Objekt Parallelisierung beschleunigt wird, ist in Abbildung 6.12 dargestellt. Bei einem Parallelisierungsgrad von 2 ist ein linearer Speed-Up zu verzeichnen, dieser nimmt ab Grad 3 ab. Ab einem Grad von 4 ist keine signifikante Beschleunigung zu erzielen. Eine Erklärung für diese schlechte Skalierbarkeit bei höherem Parallelitätsgrad ist durch den Algorithmus zur Parallelisierung lieferbar. Während Inter-Objekt Parallelisierung durch die dynamische Verteilungsstrategie, gebunden an das Iteratorkonzept, eine ungleiche Lastverteilung nicht zulässt, bilden bei Intra-Objekt Parallelisierung die *merge* Knoten blockierende Operatoren im Anfragebaum. In obiger Anfrage existiert eine solche Synchronisation, die jeweils auf den letzten und langsamsten Prozess wartet, sowohl in der Selektion wie auch in der Applikation. Die Wahrscheinlichkeit, dass die Verarbeitung gebremst wird durch einen langsamen Prozess (der etwa durch einen externen Hintergrundprozess unterbrochen wurde), steigt sowohl mit der Anzahl der Prozesse, als auch mit der Anzahl der Objekte.

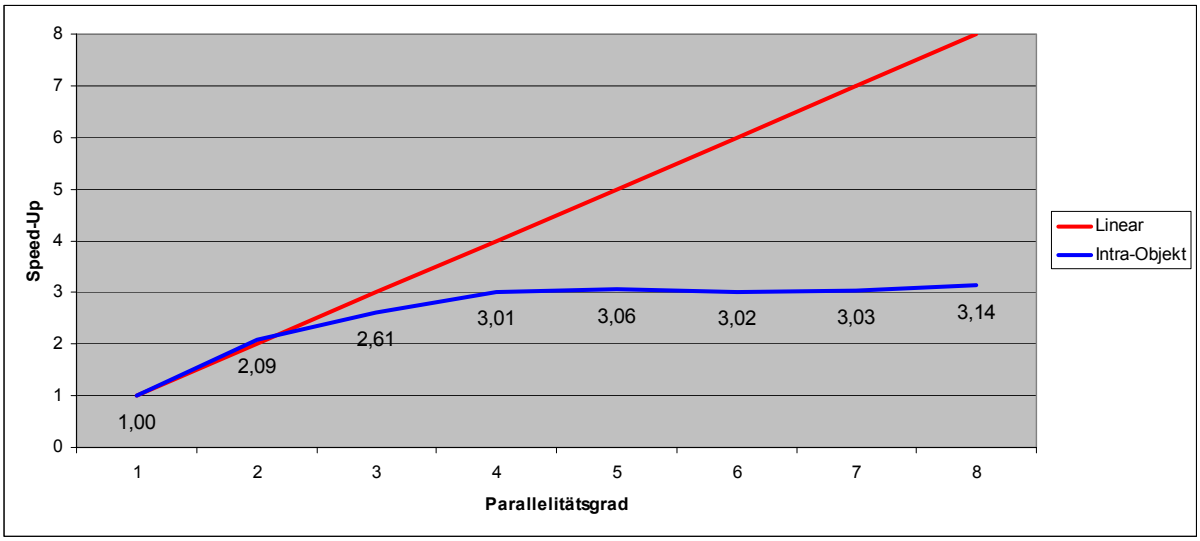


Abbildung 6.12: Speed-Up bei Intra-Objekt Parallelisierung (Shared-Everything)

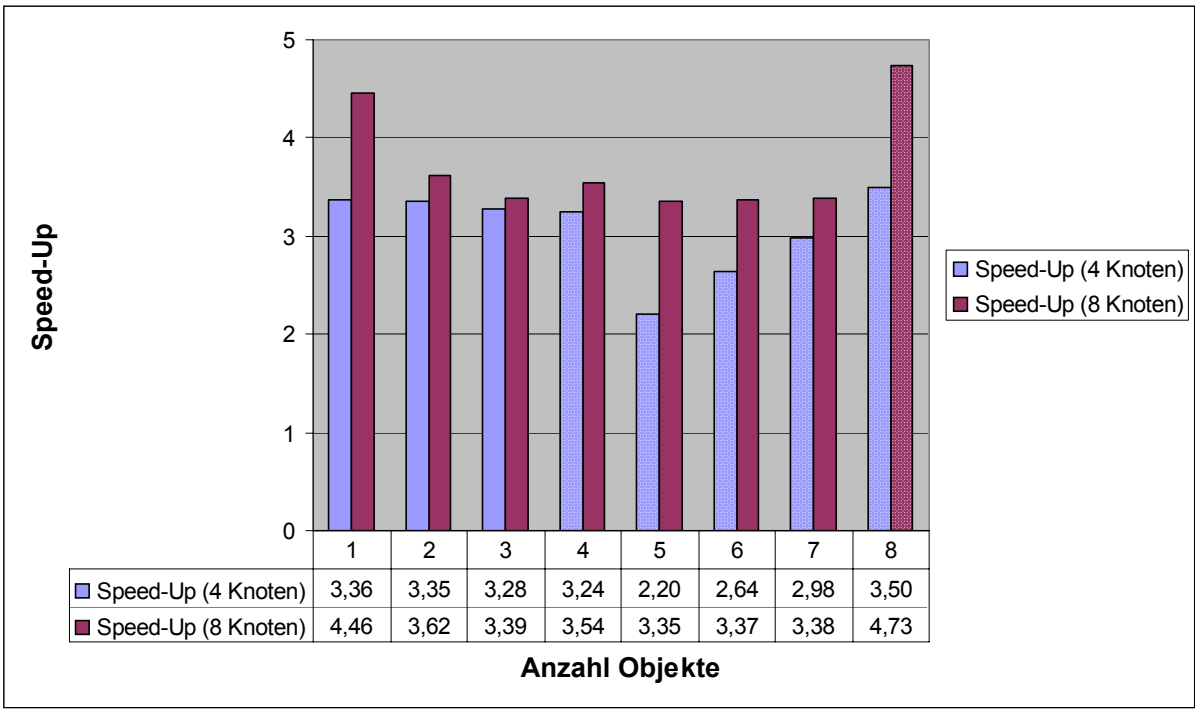
Abbildung 6.13 verdeutlicht dieses Verhalten für einen Parallelitätsgrad von 4 und 8. Die Abszisse zeigt hier die Anzahl der Objekte, die in der Anfrage verarbeitet werden, die Ordinate den erreichten Speed-Up. Ist die Anzahl der Objekte größer oder gleich dem Parallelitätsgrad, wird die Anfrage mittels Inter-Objekt Parallelisierung verarbeitet (in der Abbildung durch gemusterte Balken markiert). Bei einem Parallelitätsgrad von 4 (jeweils der erste Balken) werden Anfragen mit einer Kardinalität bis 3 mittels Intra-Obbjekt Parallelisierung verarbeitet. Der Speed-Up ist hierbei deutlich über einem Faktor von 3, also fast linear. Die Verarbeitung von 4 oder mehr Objekten wird mittels Inter-Objekt Parallelisierung vorgenommen. Während bei 4 Objekten der Speed-Up fast linear ist, zeigt die Verteilung von 5 Objekten auf 4 Prozesse einen Rückgang der Performanz. Mit steigender Anfragekardinalität wird der Speed-Up sich jedoch immer weiter auf einen konstanten Faktor einpendeln.

Die identische Anfrage unter Parallelitätsgrad 8 ausgeführt zeigt bereits die oben beschriebenen Auswirkungen. Der Speed-Up ist insgesamt sublinear, da bei einem zur Prozessoranzahl identischem Parallelitätsgrad die Ausführung sicher verzögert wird[94]. Ferner ist der Trend zu

[94] Externe Hintergrundprozesse oder Betriebssystemprozesse verzögern mindestens einen Prozess.

erkennen, dass sich bei Verarbeitung mehrerer Objekte diese Verzögerungen summieren, was den Speed-Up weiter reduziert.

Intra-Objekt Parallelisierung ist folglich besonders effizient bei einem moderaten Parallelisierungsgrad bzw. bei einer geringen Anfragekardinalität. Dies wird auch durch den Parallelisierungsalgorithmus (siehe Algorithmus 4.5) widergespiegelt, der diese Parallelisierungsstrategie nur für geringe Kardinalität einsetzt.



	1	2	3	4	5	6	7	8
Speed-Up (4 Knoten)	3,36	3,35	3,28	3,24	2,20	2,64	2,98	3,50
Speed-Up (8 Knoten)	4,46	3,62	3,39	3,54	3,35	3,37	3,38	4,73

Abbildung 6.13: Entwicklung des Speed-Up mit Anfragekardinalität (Shared-Everything)

6.3.4 Zusammenfassung der Ergebnisse für Shared-Everything Architekturen

Die Messungen auf für Shared-Everything Architekturen haben die Effizienz der Algorithmen zur Parallelisierung und die Güte der Implementierung verifiziert. Für eine typische Datenspeicherung mit Nutzung von Kompression und typische Anfragen mit Reduktion des Ergebnisvolumens zeigt sich eine fast lineare Skalierbarkeit. Das ist vor allem erstaunlich, da die Messumgebung einen Zugriff auf die Datenbasis über ein 100 MBit Netzwerk vorsieht. Als Hindernis für einen nahezu linearen Speed-Up wurde neben diesem (in der Messumgebung beabsichtigten) E/A Flaschenhals vor allem der Transfer großer Zwischenergebnisse unter den Prozessen ausgemacht, auch bei Nutzung von gemeinsamem Speicher als Kommunikationsmethode. Diese Beobachtung stützt die Prämisse der Vermeidung bzw. Reduktion von Inter-Prozess Kommunikation in den entwickelten Parallelisierungsalgorithmen.

6.4 Rechnercluster

Die Kombination von Standard PC unter Nutzung eines Hochleistungsnetzwerks wie etwa InfiniBand erzeugt eine einfach erweiterbare und im Gegensatz zu großen Multiprozessorrechnern eine extrem kostengünstige parallele Architektur, die überdies bezüglich Rechenkapazität gegenüber großen Hochleistungsrechnern oft kaum nachsteht. Die Technische Univer-

sität München und das Leibniz-Rechenzentrum der bayerischen Akademie der Wissenschaften unterhalten beide große Rechnercluster, die für komplexe parallelisierbare Berechnungen ausgelegt sind. Für unsere Messungen nutzten wir jedoch nicht diese Hochleistungsverbünde, sondern Solaris Workstation Rechner der Technischen Universität München, die über ein einfaches 100MBit Netzwerk verbunden sind. Die Gründe für diese Wahl im Einzelnen sind folgende:

- Die Effizienz der entwickelten Algorithmen soll auch für typische Szenarien, d.h. einfachen Rechnern verbunden über ein typisches Netzwerk, gewährleistet sein, nicht nur für spezialisierte Architekturen.
- Die Architektur von rasdaman mit der Notwendigkeit einer Installation eines relationalen DBMS verhinderte einen Einsatz auf dem Linux Cluster des Leibnitz-Rechenzentrums. Eine Installation von Oracle direkt im Verbund war ebenso wie eine Verbindung zu einem externen RDBMS Server aus Gründen der Sicherheit nicht erwünscht.
- Der Rechnerverbund des Lehrstuhls Bode an der Technischen Universität stand uns für Messungen und Versuche zur Verfügung. Leider hat sich gezeigt, dass die Architektur mit extrem leistungsfähigen Rechnern, verbunden über InfiniBand Netzwerk nur über ein 100 MBit Netzwerk[95] auf Daten zugreift (via NFS). Die Verbindung zur relationalen Datenbasis wird damit (vor allem auch durch die Leistungsfähigkeit der anderen Systemkomponenten) zu einem Flaschenhals, der die Skalierbarkeit praktisch zunichte macht. Trotzdem wollen wir uns an dieser Stelle bei Prof. Bode und Dr. Mairandres für die freundliche Unterstützung bedanken.

6.4.1 Messumgebung

Für die Analyse von rasdaman 5.0 PE auf Rechnercluster-Architekturen wurden bis zu 22 SUN Ultra 60 Workstations, verbunden über ein 100 MBit Netzwerk, als Rechenverbund genutzt. Die Rechner sind in ihrer Ausstattung nahezu identisch[96], neben 2 CPU zu je 300 MHz sind sie mit jeweils 384 MB Hauptspeicher ausgestattet. Der Zugriff auf das RDBMS mit der Datenbasis erfolgt wiederum über Ethernet Netzwerk (100 MBit). Als RDBMS Server mit Oracle 8.1.7 fungiert eine SUN Enterprise 450 mit 2 CPU (je 250 MHz) und 1 GB Hauptspeicher sowie RAID 0 Festplattenverbund.

Die Architektur für die Messumgebung ist in Abbildung 6.14 skizziert. der rasdaman Manager sowie der Master Prozess des Servers sind auf einem Rechner installiert (sunhalle6). Weitere Worker Prozesse werden auf die vorhandenen Rechner des Clusters verteilt, jeweils ein Prozess pro Rechner. Die Kommunikation sowohl zwischen den rasdaman Prozessen als auch der Zugriff auf das RDBMS erfolgt über ein Ethernet LAN mit 100 MBit.

[95] Ein Ausbau auf 1GBit Netzwerk ist bereits geplant, wird aber für unsere Messungen zu spät vollendet.

[96] Einziges Unterscheidungskriterium in der Ausstattung waren wenige Rechner mit nur einer CPU. Diese wurden erst ab einem hohen Parallelisierungsgrad eingesetzt, um die Messergebnisse nicht zu verfälschen. Da für die Messungen pro Rechner nur ein Prozess für rasdaman PE gestartet wurde, ist die einzige Auswirkung hiervon ein evtl. geringfügig schlechterer Speed-Up bei hohem Parallelitätsgrad durch die Beeinflussung von rasdaman durch externe Prozesse auf diesen Rechnern.

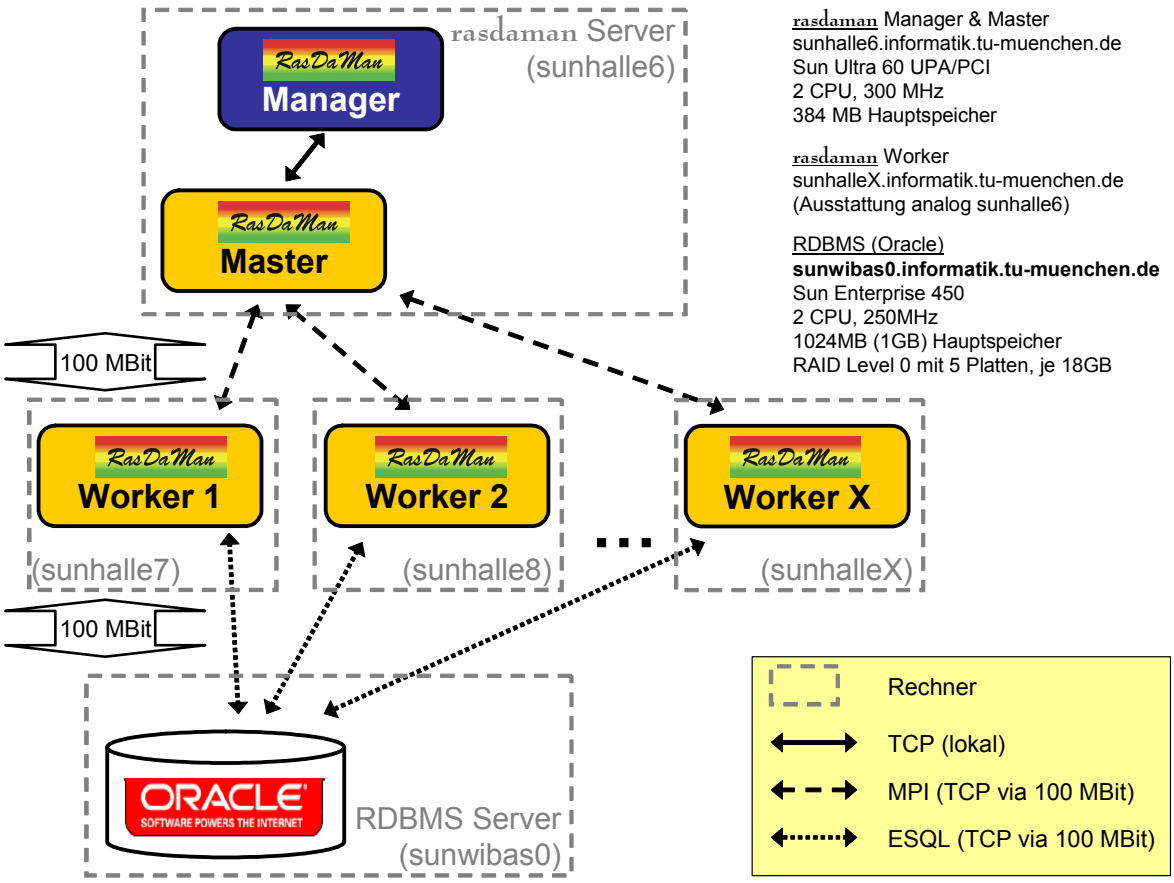


Abbildung 6.14: Shared-Disk Architektur (Messumgebung)

6.4.2 Inter-Objekt Parallelisierung

Die erste Performanzmessung auf unserer Rechnercluster-Architektur demonstriert den Speed-Up einer typischen rechenintensiven Anfrage. Die RasQL Anfrage ist analog zu der ersten vorgestellten Anfrage in Kapitel 6.3.2 und referenziert 2D Satellitenbilder, die komprimiert gespeichert wurden.

Abbildung 6.15 zeigt einen linearen Speed-Up der Anfrage bis zu einem Parallelitätsgrad von 10, darüber hinaus einen leicht sublinearen Verlauf. Der leicht überlineare Verlauf zwischen Grad 2 und 6 ist ein bei Rechnerclustern häufig zu beobachtendes Phänomen. Die Verteilung einer Berechnung auf mehrere parallele Instanzen mit jeweils eigenen Ressourcen entschärft Engpässe der Verarbeitung. Ein nicht ausreichend dimensionierter Hauptspeicher mit Auslagern von Daten auf Festplatte (engl.: swapping) als Folge, welches zu einem Nachteil für die nicht-parallele Berechnung führt, wurde hierbei ausdrücklich ausgeschlossen[97]. Somit bleibt vor allem folgende Erklärungsmöglichkeit für die Tendenz zu leicht überlinearem Speed-Up: auf Seiten des DBMS Servers können Caching Effekte (im Dateisystem oder durch das DBMS selbst, z.B. Indexe) genutzt werden. Bei Zugriff mehrerer Prozesse auf dieselbe Tabelle können etwa durch einen Full Table Scan die Prozesse größtenteils aus dem Cache bedient werden.

[97] Dieser „Trick“ für die Erzielung einer hervorragenden Skalierbarkeit wird häufig eingesetzt. In unseren Messungen wurde dies einerseits durch die genaue Analyse der Anfragen selbst, andererseits durch Nutzung des UNIX Werkzeugs *nice* verhindert, das eine komplette Nutzung des physikalischen Hauptspeichers durch einen einzelnen Prozess verhindern kann.

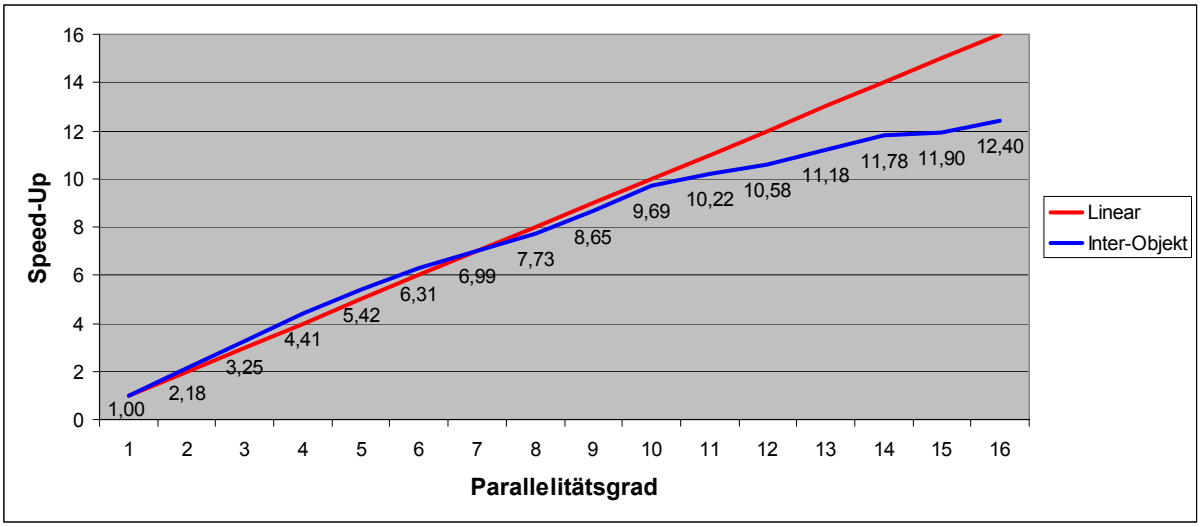


Abbildung 6.15: Speed-Up von rasdaman PE unter Nutzung einer Shared-Disk Architektur mit 100MBit Netzwerk

Der sublineare Verlauf bei hohem Parallelitätsgrad (bei einem Grad von 20 beträgt der Speed-Up 13,6) ist vor allem auf den Kostenfaktor C_{lo} zurückzuführen, das heißt bei vielen parallelen Instanzen kommt es zu Problemen der Datenversorgung über Netzwerk. Werden Verzögerungen bei der Bereitstellung der Quelldaten vermieden, etwa durch Nutzung von Datenbankreplikation, fällt dieser Faktor weg. In diesem Fall kann für Anfragen mit geringem Transfervolumen zwischen Prozessen ein linearer Speed-Up auch bei sehr hohem Parallelitätsgrad erzielt werden.

6.4.2.1 Einflussfaktor Anfragekomplexität

Der Einfluss der Komplexität einer Anfrage stellt sich wie bereits für Shared-Everything Architekturen dar (siehe 6.3.2.1). Die dort beschriebene Anfrage zeigt für unsere Cluster-Architektur folgenden Verlauf des Speed-Up (Abbildung 6.16).

Der im Gegensatz zu Abbildung 6.15 etwas flachere Verlauf der Kurve für die komplexe Anfrage ergibt sich wiederum durch die geringere Komplexität der Anfrage und die Referenzierung einer unkomprimierten Datenbasis. Die einfache Anfrage zeigt einen (fast) linearen Verlauf ab einem Parallelitätsgrad von 5. Die geringe Komplexität einer einzelnen Aggregation in der Selektion (die Applikation wird nicht ausgeführt, da die Selektivität 0% beträgt), verbunden mit einem großen Datenvolumen der referenzierten Kollektion, führt zu einer E/A-lastigen Anfrage. Die Ausführungszeit wird von E/A bestimmt, die somit schnell zum Flaschenhals der Verarbeitung wird.

Einen ähnlichen Effekt auf die Skalierbarkeit zeigten Anfragen mit Referenzierung von Kollektionen unterschiedlichen Basisdatentyps. So zeigen identische Anfragen auf einer Kollektion mit Basisdatentyp *char* und Basisdatentyp *float* einen anderen Verlauf der Speed-Up Kurve. Im zweiten Fall wird die 4-fache Datenmenge (4 Byte statt 1 Byte) geladen, die arithmetische Operation pro Zelle ist jedoch identisch. Folglich ist der Speed-Up für die Anfrage auf der Kollektion mit Basisdatentyp *float* geringfügig flacher. Bezüglich der Datenmodellierung von Array Daten kann man folgern, dass vor allem ein Basisdatentyp mit vielen Variab-

len negative Auswirkungen auf E/A und Parallelität hat, da die kompletten Daten von rasdaman geladen werden, aber nur ein Bruchteil analysiert.

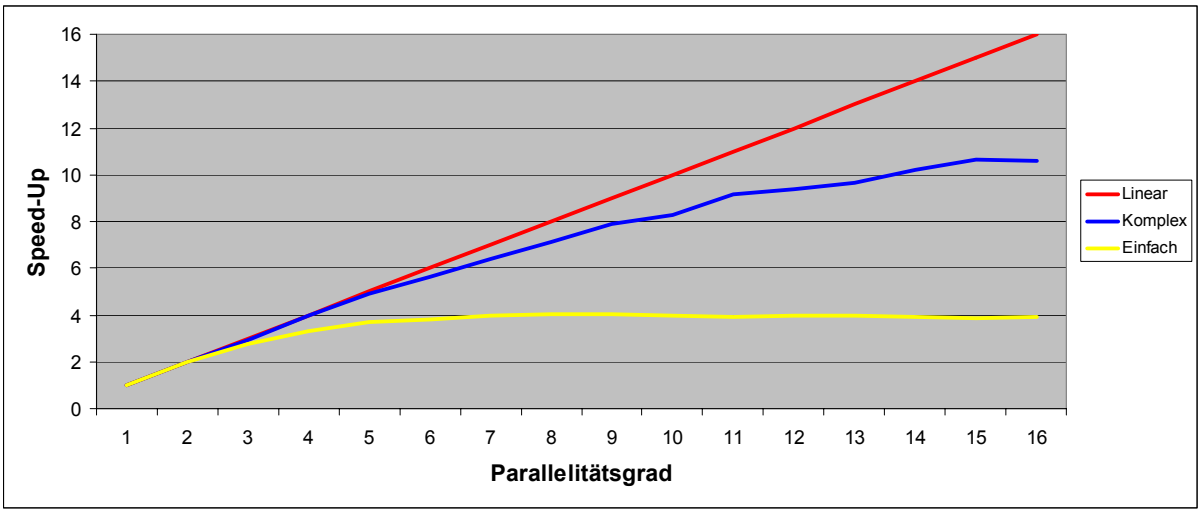


Abbildung 6.16: Einfluss der Anfragekomplexität auf den Speed-Up (Shared-Disk)

6.4.2.2 Einflussfaktor Kommunikationsaufwand und Datenkompression

Der Vollständigkeit halber präsentieren wir den Einfluss der Inter-Prozess Kommunikation und der Datenkompression, welche identisch zu den Auswirkungen bei Multiprozessor-Architekturen sind und bereits in Kapitel 6.3 diskutiert wurden. Die Skalierbarkeit von Anfragen mit unterschiedlichen Datenvolumina des Resultats folgt aus dem Volumen der zwischen den Prozessen zu transferierenden Zwischenergebnissen. Je geringer der Anteil an Inter-Prozess Kommunikation, desto besser die Skalierbarkeit der Anfrage. Abbildung 6.17 zeigt neben diesem Zusammenhang jedoch noch ein weiteres interessantes Detail: da die referenzierte Kollektion 32 Arrays beinhaltet, zeigen die Parallelisierungsgrade 4, 8 und 16 leichte „Spitzen“ in der Kurve. Dies rührt daher, dass hier der Parallelisierungsgrad ein Teiler der Anfragekardinalität ist und meist eine Lastbalancierung bis zum Ende der Anfrage gewährleistet bleibt.

Bei Einsatz von Kompression gewinnt das Kompressionsverfahren „Haar Wavelet“ gegenüber mit ZLib komprimierten und unkomprimierten Daten noch deutlicher als beim Multiprozessorrechner (Kapitel 6.3.2.3). Dies liegt vor allem an der geringeren Leistungsfähigkeit der einzelnen parallelen Knoten (CPU mit 900 MHz auf dem Multiprozessorrechner gegenüber CPU mit 300 MHz auf den Rechnern des Clusters).

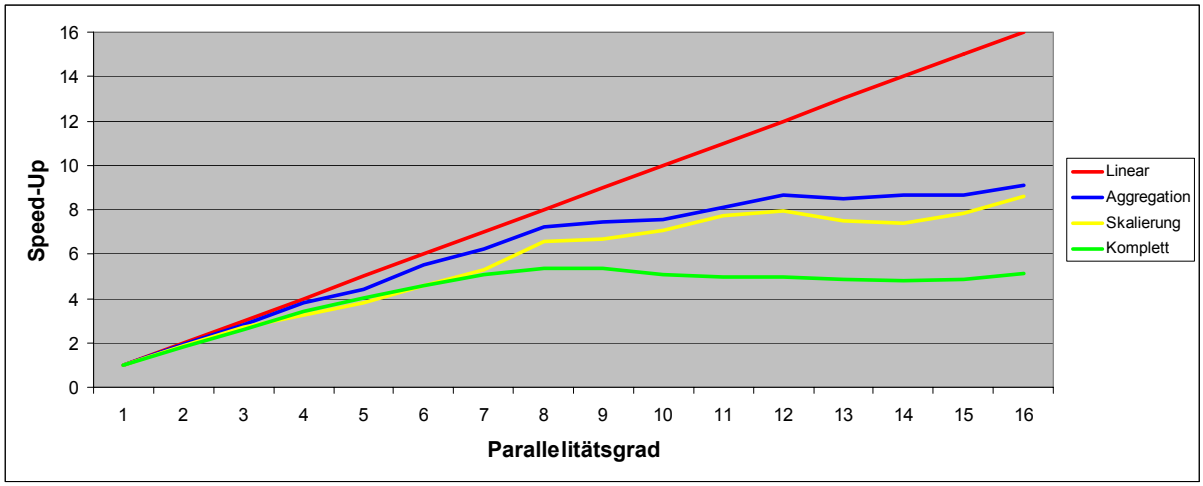


Abbildung 6.17: Einfluss der Inter-Prozess-Kommunikation (Shared-Disk)

Beim Einsatz von stark komprimierenden Verfahren für die Reduktion der Datenbasis zeigen Anfragen bis zu einem hohen Parallelitätsgrad einen fast linearen Anstieg, der erst durch einen Engpass bei E/A limitiert wird.

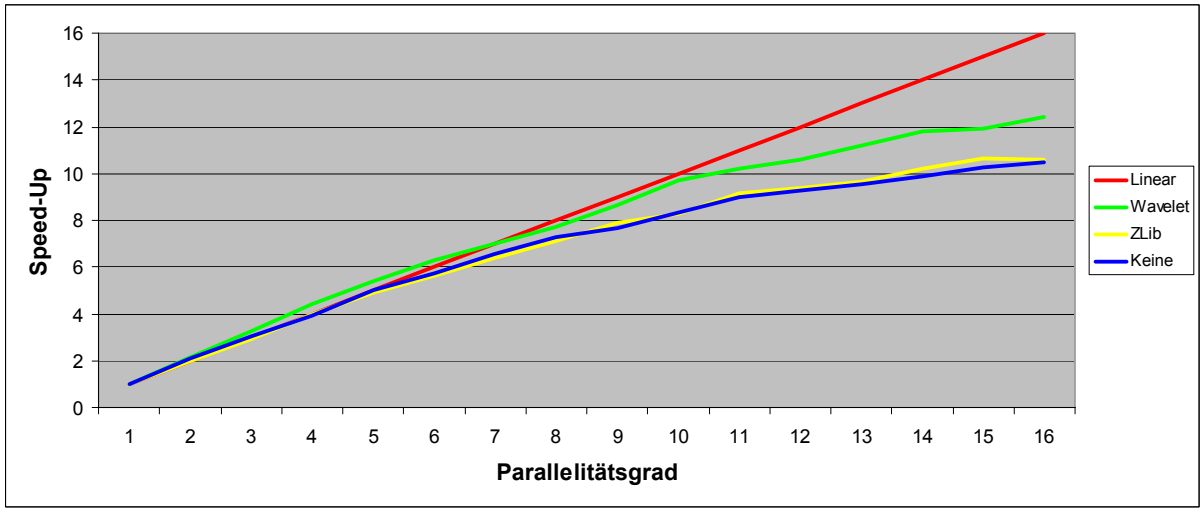


Abbildung 6.18: Einfluss der Datenkompression auf den Speed-Up (Shared-Disk)

6.4.3 Intra-Objekt Parallelisierung

Intra-Objekt Parallelisierung zeigt auch auf dem Rechnercluster das bereits bei Multiprozessorrechnern beobachtete Verhalten: bei geringem Parallelitätsgrad noch ein linearer Speed-Up, schwächt sich die Verbesserung der Performanz bereits ab einem Grad von 4 deutlich ab (Abbildung 6.19).

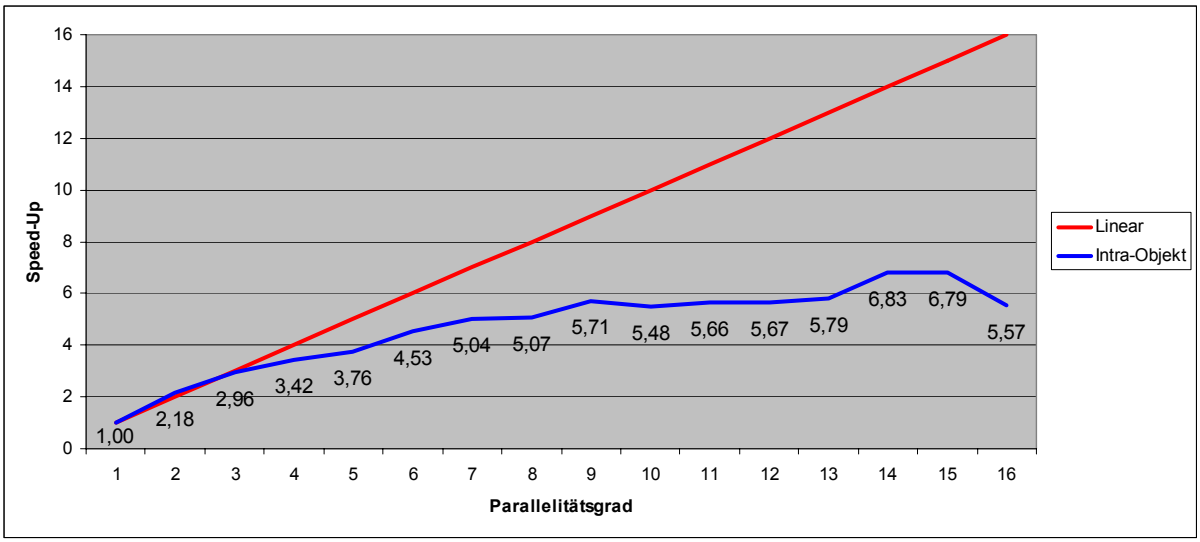


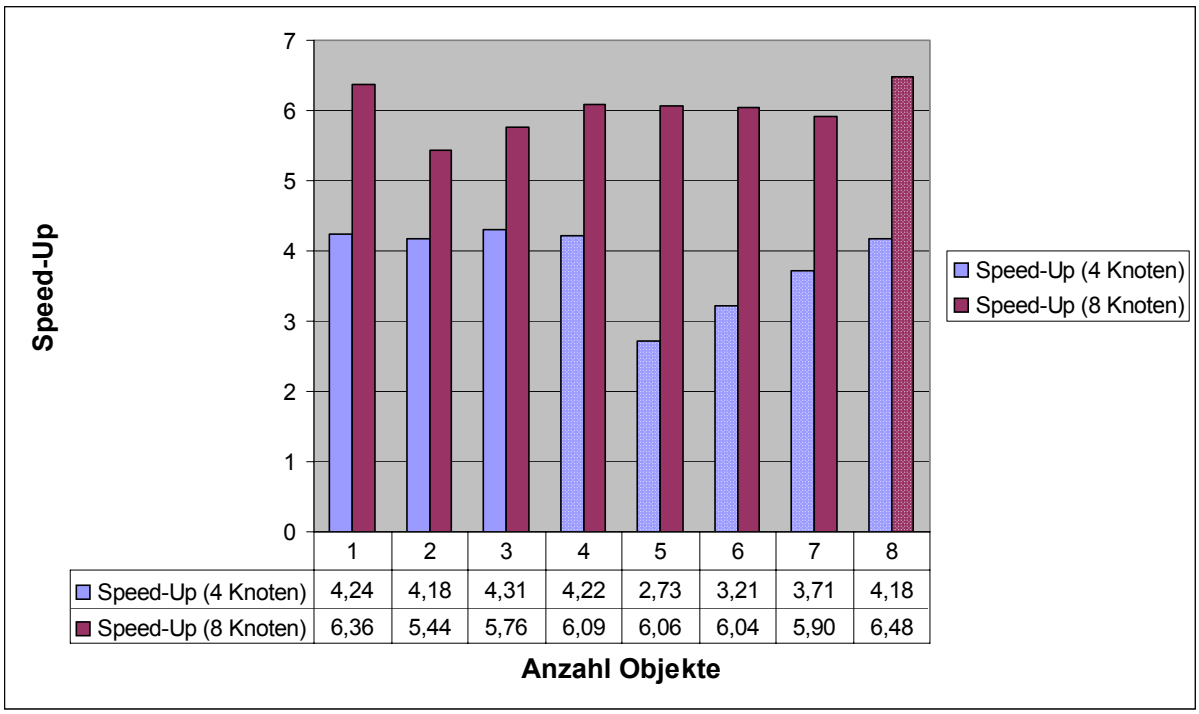
Abbildung 6.19: Speed-Up bei Intra-Objekt Parallelisierung (Shared-Disk)

Im Gegensatz zu den Messungen auf Multiprozessorrechner zeigt sich auf dem Rechnercluster das blockierende Verhalten durch Synchronisationspunkte für die Prozesse während der Ausführung noch deutlicher. Die Messergebnisse zeichnen sich durch eine hohe Schwankungsbreite aus, Differenzen von bis zu 30% bei einer aufeinander folgenden Messung mit identischen Parametern sind keine Seltenheit. Bereits kleine Störungen im Netzwerk[98] oder störende externe Prozesse führen dazu, dass die Ausführung unterbrochen wird, bis der beeinflusste Prozess den Synchronisationspunkt erreicht. Insbesondere bei vielen parallelen Instanzen ist die Wahrscheinlichkeit für ein solches Szenario und damit einer suboptimalen Performanz sehr groß.

In Abbildung 6.19 werden Mittelwerte für den Speed-Up demonstriert, gute Ergebnisse wurden insbesondere nachts bei geringer Auslastung des Netzwerks und der Rechner erreicht. Diese Spitzenwerte näherten sich der der linearen Kurve deutlich an. Bei Auslastung der Rechner durch Studenten wochentags brachen die Ergebnisse dagegen teils deutlich weiter ein.

In Abbildung 6.20 hingegen werden die gemessenen Spitzenwerte für einen Parallelitätsgrad von 4 und 8 präsentiert. Die Wahl des Parallelisierungsalgorithmus ist durch die Musterung dargestellt, bei Parallelitätsgrad 4 erfolgt ab 4 Objekten eine Inter-Objekt Parallelisierung, bei Grad 8 ab 8 Objekten. Der optimale Speed-Up für Intra-Objekt Parallelisierung ist bei geringer Auslastung von Rechnern und Netzwerk deutlich besser, für einen Grad von 4 sogar leicht überlinear, für Grad 8 annähernd linear. Wiederum identifizierbar ist der Einbruch der parallelen Performanz, wenn die Anzahl der Objekte den Parallelitätsgrad nur leicht übersteigt (etwa bei 5 Objekten und Parallelitätsgrad 4).

[98] Verzögerungen des Datentransfers durch das Verwerfen und die Neuanforderung eines TCP Pakets.



	1	2	3	4	5	6	7	8
Speed-Up (4 Knoten)	4,24	4,18	4,31	4,22	2,73	3,21	3,71	4,18
Speed-Up (8 Knoten)	6,36	5,44	5,76	6,09	6,06	6,04	5,90	6,48

Abbildung 6.20: Einfluss der Anfragekardinalität auf den Speed-Up (Shared-Disk)

6.4.4 Zusammenfassung der Ergebnisse für Shared-Disk Architekturen

Die Tatsache, dass Anfragen bei Nutzung einer Shared-Disk Architektur bis zu einem Faktor von 16 und darüber hinaus skalieren, kann durchaus als positive Überraschung bezeichnet werden. Das gilt umso mehr, da die parallele Nutzung der Datenbasis über 100 MBit Netzwerk erfolgte. Ein in der Messumgebung kalkulierter Flaschenhals bei E/A hat sich somit weniger deutlich ausgewirkt als erwartet.
Die Performanz und Skalierbarkeit der Analyse von multidimensionalen Arrays auf Shared-Nothing Architekturen kann insbesondere unter zwei Faktoren leiden:

- E/A Flaschenhals bei der Datenversorgung
- Kosten der Inter-Prozess Kommunikation

Beide Faktoren können einfach durch ein schnelles Netzwerk ausgeschlossen werden, insbesondere Gigabit Ethernet als billige Variante und InfiniBand als Hochleistungsvariante sind hier eine Option. Das Problem eines Flaschenhalses bei der Versorgung mit Basisdaten kann darüber hinaus durch Replikation der Datenbasis gelöst werden.

6.5 Zusammenfassung und Bewertung der parallelen Architekturen

Sowohl für Multiprozessor wie auch für Rechnercluster Architekturen zeigt rasdaman 5.0 PE hervorragende parallele Performanz und Skalierbarkeit. Die gute Skalierbarkeit kann vor allem auf zwei Faktoren zurückgeführt werden. Einerseits auf die Anfragen, die meist komplexe Analysen mit einer finalen Reduktion des zu übertragenden Datenvolumens realisieren. Diese Art von Anfragen ist jedoch typisch und somit ein primärer Kandidat für Performanzsteigerung durch Parallelität. Der zweite Faktor für die gute Skalierbarkeit liegt in den entwickelten Algorithmen für die parallele Verarbeitung begründet. Die Probleme der Parallelität werden konsequent vermieden:

1. Kosten der Initialisierung werden durch einen äußerst schnellen Algorithmus zur Parallelisierung des Anfragebaumes minimiert.
2. Kosten der Inter-Prozess Kommunikation können in vielen –und vor allem den typischen– Anfragen durch geschicktes Positionieren der Kommunikationspunkte vermieden werden.
3. Kosten durch ungleiche Lastverteilung werden durch eine Kopplung der parallelen Verarbeitung mit dem dynamischen Iteratorkonzept bei der primären Parallelisierungsstrategie weitestgehend ausgeschlossen.

Die Grenzen der Skalierbarkeit zeigen sich folglich für die meisten der hier präsentierten Anfragen nicht wegen prinzipieller Probleme wie etwa einer Zunahme der Inter-Prozess Kommunikation, sondern allein durch den Aufbau der Messumgebung. Die Nutzung schneller Netzwerke oder von Datenbankreplikation kann den E/A Flaschenhals entschärfen und die Skalierbarkeit weiter erhöhen.

Einen Sieger unter den zwei parallelen Architekturen auszumachen, fällt dabei schwer, insbesondere da aus technischen Gründen die Multiprozessor Messumgebung keine reine Shared-Everything Architektur war. Bei Realisierung einer reinen Shared-Everything Architektur, das heißt bei Einsatz eines lokal installierten RDBMS kann bei gleichzeitiger Nutzung einer Inter-Prozess Kommunikation basierend auf gemeinsamen Speicher ein linearer Speed-Up für einen Großteil der Anfragen gerechnet werden. Was die pure Leistungsfähigkeit angeht, sind Multiprozessorrechner also als überlegen anzusehen.

Sobald jedoch ein Kostenfaktor bei den Überlegungen berücksichtigt wird, sind Rechnercluster die bessere Wahl. Eine praktisch identische Performanz und Skalierbarkeit für typische Anfragen bei einer deutlich besseren Erweiterbarkeit und einem Bruchteil der Kosten gegen über einem leistungsmäßig vergleichbaren Multiprozessorrechner sprechen zugunsten einer Clusterlösung. Die Nachteile einer teuren Inter-Prozessor Kommunikation und eines schwerer zu entschärfenden E/A Flaschenhalses lassen sich durch Nutzung eines leistungsfähigen Netzwerkes und Datenbankreplikation einfach entschärfen.

Somit ist doch aus Sicht des Praktikers ein Sieger bei der Wahl der Architektur auszumachen: sobald ein Kostenfaktor zu berücksichtigen ist (und wann ist das nicht der Fall), kann mit einem Rechnercluster eine leistungsfähige und gut skalierbare Architektur für **rasdaman 5.0** PE geschaffen werden. Die im Gegensatz zu einem vergleichbaren Multiprozessorrechner gesparten finanziellen Mittel können für E/A (vor allem schnelle RAID Systeme), Netzwerk (Gigabit Ethernet oder InfiniBand), oder weiterer Plattenplatz für replizierte RDBMS eingesetzt werden.

Kapitel 7 Zusammenfassung und Ausblick

Don't fear failure so much
that you refuse to try new things.
The saddest summary of a life
contains three descriptions:
could have, might have, and should have.
(Louis E. Boone)

Multidimensionale diskretisierte Daten sind das primäre Ergebnis wissenschaftlicher Observationen, Versuche und Simulationen. Während aktuell die Speicherung der Datenflut mit Hilfe von Tertiärspeichersystemen ohne größere Probleme bewältigt werden kann, ist eine umfassende Analyse nicht nur problematisch, sondern de facto unmöglich. Ein zentrales Hindernis ist hierbei die fehlende Möglichkeit einer performanten dynamischen Evaluierung der gewonnenen Daten. Neben einer Optimierung der Verarbeitung ist vor allem die Nutzung von parallelen Rechnerarchitekturen der Schlüssel zu einer Steigerung der Performanz. Die Datenbank gestützte parallele Verarbeitung von Mengen sehr großer Datenobjekte im Allgemeinen und von multidimensionalen Array Daten im Speziellen hat bisher noch keine wissenschaftliche Beachtung gefunden. Diese Lücke schließt die vorliegende Arbeit.

Kern dieser Arbeit ist die Verteilung der Rechenlast für die Analyse multidimensionaler Rasterdaten. Zu diesem Zweck wurden im Wesentlichen zwei Methoden der Parallelisierung entwickelt: die Inter-Objekt Parallelisierung stützt sich auf Methoden und Erkenntnisse der relationalen Intra-Operator Datenparallelität, sie wurde jedoch für die Analyse von großen multidimensionalen Array Daten adaptiert und optimiert. Die neu entwickelte Parallelisierungsstrategie der Intra-Objekt Parallelisierung partitioniert Rasterdaten in Teilbereiche, um Datenparallelität zu ermöglichen. Die Algorithmen zur Aufteilung von multidimensionalen Array Daten sind eng verzahnt mit Kachelungsstrategien, welche bei der Speicherung von Rasterdaten mit großen Datenvolumina üblich sind. Die implementierten Algorithmen unterstützen alle uns bekannten Kachelungsstrategien. Werden von anderen Systemen diese nicht vollständig

unterstützt, reduziert sich die Komplexität der Parallelisierung, jedoch in der Regel nicht die parallele Performanz. Maßgeblicher Aspekt für eine effiziente Implementierung dieser Konzepte war die konsequente Vermeidung der „Sünden der Parallelisierung“: Minimierung der Initialisierungskosten, Minimierung der Inter-Prozess Kommunikation und somit des Transfers von Zwischenergebnissen und Vermeidung von ungleicher Lastverteilung.

Alle beschriebenen Methoden der parallelen Anfrageausführung sowie diesbezüglich adaptierte und optimierte Erweiterungen der Analysemöglichkeiten wurden in **rasdaman 5.0 PE** integriert und bereits während des Projekts ESTEDI ausführlich getestet. Die Qualität der Algorithmen und der Implementierung wurde durch umfangreiche Performanzmessungen auf den relevanten parallelen Architekturen demonstriert. Sowohl auf Multiprozessor- als auch auf Rechnercluster-Architekturen zeigte **rasdaman 5.0 PE** eine hervorragende Performanz und Skalierbarkeit. Auf einem 8 CPU Multiprozessorrechner mit einem wohl kalkulierten Flaschenhals einer RDBMS Anbindung über 100MBit Ethernet zeigte **rasdaman 5.0 PE** für typische Anfragen einen nahezu linearen Speed-Up bis zu einem Parallelitätsgrad von 5. Bei Parallelitätsgrad 8, also bei Nutzung aller Prozessoren, lag der Speed-Up immer noch bei einem Wert über 6. Die Messung auf einer Shared-Disk Architektur ergab einen nahezu linearen Speed-Up bis zu einem Parallelitätsgrad von 10, bei Grad 16 noch einen beachtlichen Wert von 12,4. In beiden Messumgebungen ist der Grund für den leicht sublinearen Anstieg bei hohem Parallelitätsgrad in der Messumgebung, konkret der Anbindung der Datenquelle über 100 MBit Ethernet LAN, zu suchen und nicht in inhärenten Problemen der Parallelisierungsalgorithmen. Der Einsatz von neuen leistungsfähigen Netzwerkstandards wie InfiniBand wird die Skalierbarkeit deutlich verbessern.

Die in dieser Arbeit entwickelten Konzepte sind nicht auf das eingesetzte DBMS **rasdaman**, dessen Datenmodell oder multidimensionale Array Daten beschränkt: zentrale Strategien für eine effiziente Analyse von Mengen großer Datenobjekte werden in dieser Arbeit für eine performante Parallelverarbeitung ausgenutzt, folglich sind die entwickelten Algorithmen auf ähnlich strukturierte Datenmengen übertragbar:

- Zur besseren Nutzung der Ressourcen wird die Verarbeitung einer Datenmenge immer durch Iteration über die einzelnen Objekte geschehen. Eine Kopplung der parallelen Verarbeitung an das Iteratorkonzept und damit eine dynamische Verteilungsstrategie, welche eine ungleiche Lastverteilung vermeidet, ist ein allgemein gültiges Konzept.
- Eine Verzögerung des Ladens von Daten kann E/A entlasten, insbesondere wenn Bereichseinschränkungen in der Anfrage enthalten sind oder Operationen einen frühzeitigen Abbruch ohne Evaluierung der gesamten Eingabe erlauben. Die entwickelten Konzepte zur Parallelverarbeitung nutzen diese Mechanismen aus, indem eine Zuteilung von Daten zu parallelen Instanzen nach vorhandenen Bereichseinschränkungen, aber vor dem Laden der Daten erfolgt. Damit wird einerseits eine Lastbalancierung durch optimale Verteilung der Datenbasis und damit der Gesamtlast garantiert. Andererseits wird eine Minimierung der Inter-Prozess Kommunikation durch Vermeidung eines wiederholten Transfers der Datenbasis erreicht: erst die parallelen Instanzen laden die für sie relevanten Quelldaten.
- Anfragen auf große Datenvolumina extrahieren meist charakteristische Werte (berechnet durch aggregierende Funktionen und dargestellt durch skalare Werte) oder liefern ein skaliertes und somit in der Größe reduziertes Resultat. Dies folgt einerseits aus vorhandenen Beschränkungen von Bandbreite für den Datentransfer, andererseits aus der Charakteristik der Evaluierung von großen Datenmengen, die ein Lokalisieren von interessanten Objekten vor eine genaue Analyse der einzelnen Datensätze stellt. Die entwickelten Parallelisierungsstrategien nutzen diese Analysecharakteristik, indem eine finale Fusion der parallel

berechneten Zwischenergebnisse prinzipiell nach einer Reduktion des Datenvolumens erfolgt. Dieses Vorgehen reduziert die Inter-Prozess Kommunikation auf ein Minimum und sichert hervorragende Performanz und Skalierbarkeit.

- Die Analyse von Mengen großer Datenobjekte kann sich nicht allein auf die parallele Verarbeitung von Teilmengen und damit von kompletten Objekten stützen. Das enorme Datenvolumen einzelner Objekte geht zumeist mit einer Reduzierung der Kardinalität der Menge einher. Eine effiziente parallele Verarbeitung von solchen Mengen geringer Kardinalität erfordert zusätzlich zu einer parallelen Verarbeitung von Teilmengen eine parallele Analyse innerhalb von einzelnen Objekten. Die in dieser Arbeit entwickelten Methoden zur Intra-Objekt Parallelisierung können konzeptuell auf ähnliche Datenmodelle übertragen werden, lediglich die im Datenmodell definierten Funktionen bedürfen einer Evaluierung bezüglich ihrer Eignung für eine parallele Ausführung.

Welche neuen Herausforderungen gibt es für die parallele Analyse von multidimensionalen Array Daten? Insbesondere zwei interessante Ideen sorgten während der Entstehung dieser Arbeit für Diskussionen: Methoden zur Steigerung der parallelen Performanz und neue Möglichkeiten der Analyse und der Optimierung für multidimensionale Rasterdaten.

Ideen zur Performanzsteigerung bei Parallelität zielen vor allem auf die Verbesserung des Datenzugriffs, der sich bei hohem Parallelitätsgrad als Flaschenhals entpuppen kann. Kachelungsstrategien verbunden mit einem multidimensionalen Index ermöglichen einen selektiven Zugriff auf Bereiche innerhalb großer Arrays. Jedoch wird in **rasdaman** kein multidimensionales Clustering der einzelnen Kacheln realisiert, durch die Architektur mit unabhängigen RDBMS zur Speicherung der Kacheln als BLOBs ist eine Optimierung hier auch nicht möglich. Verbunden mit einer Verarbeitungsstrategie, die beim Zugriff auf Zellen im multidimensionalen Raum Kacheln einzeln lokalisiert und lädt, zeigt sich insbesondere bei einer geringen Kachelgröße eine suboptimale Leserate für die Daten. Dass eine Optimierung hier möglich wäre, falls Zugriff auf die Algorithmen zur Datenspeicherung besteht, zeigen Versuche mit sehr großen Kacheln: die Leserate für die Daten erhöht sich um den Faktor 10 und mehr, da das eingesetzte RDBMS Oracle ein sequentielles Lesen innerhalb von BLOBs verwirklichen kann. Der Einsatz eines multidimensional clusternden Index für die Speicherung der Kacheln wie etwa durch die Z-Kurve des UB-Baums [Bay97] realisiert, könnte die E/A-Rate erheblich beschleunigen und somit einen primären Flaschenhals der parallelen Analyse entschärfen.

Ein zweiter Punkt für zukünftige Forschung ist die Entwicklung neuer Analyse- und Optimierungsmöglichkeiten für Array Daten. Im Projekt ESTEDI wurden beide Punkte in **rasdaman** bereits entscheidend verbessert [Mil03] [Lav03]. Trotzdem können einige Punkte identifiziert werden, die weiterer Forschungsanstrengungen wert sind:

- Eine generelle Erweiterung der Operationen des Datenmodells, um die Transformation einer Menge von Punkten (Array) zu einer Menge von Objekten (Kollektion) und umgekehrt zu definieren. Ohne diese Transformation ist eine Analyse immer an die Modellierung der Daten gebunden. Will man etwa Simulationsdaten in der Granularität von Jahren vergleichen und verarbeiten, verhindert die Modellierung als Menge von Arrays mit jeweils einem Jahr Ausdehnung in der Zeitdimension eine Zeitreihenanalyse darüber hinaus. Ein erster Ansatz, welcher eine Transformation eines Arrays in eine Kollektion von Arrays mit disjunkten Domänen beschreibt, wurde mit dem Partitionierungsiterator bereits entwickelt und implementiert (siehe Kapitel 2.3). Ein genereller Ansatz, der diesen Ansatz generalisiert und darüber hinaus die Transformation einer Kollektion in ein Array erhöh-

ter Dimensionalität algebraisch beschreibt, würde die Mächtigkeit des Datenmodells und damit die Analysefähigkeiten, enorm verbessern.

- In vielen Bereichen hat sich gezeigt, dass die Trennung der Datenhaltung in einem RDBMS und der Datenanalyse in **rasdaman** zu einem Hindernis für weitere viel versprechende Optimierungen führt. So können etwa eine Optimierung der Datenhaltung (zum Beispiel multidimensionales Clustering, siehe oben), der Aufbau und die Pflege eines Systemkatalogs, die Realisierung eines darauf basierenden Kostenmodells oder die Integration neuer darauf basierender Optimierungsmethoden nicht realisiert werden. Die Aufhebung der Trennung durch Verzicht auf ein RDBMS kann zwar insbesondere zu Problemen mit der Transaktionssicherheit führen, die bisher komplett durch das RDBMS übernommen wird. Nach unserer Erfahrung wird ein Array DBMS jedoch nicht als Transaktionssystem genutzt, vielmehr erfolgt nach einer Einfügephase ein reiner Lesezugriff. Eine konsequente Lösung dieses Problems kann letztlich nur in der Integration des Datentyps MDD und von Array Operationen in den Kern eines objekt-relationalen DBMS bestehen.

Literaturverzeichnis

[AK01a] Active Knowledge GmbH: *RasDaMan, version 5.0: Query Language Guide.* 2001

[Amd67] G. M. Amdahl: *Validity of single-processor approach to achieving large-scale computing capability.* Proceedings of AFIPS Conference, Reston, VA. 1967. pp. 483-485

[AVFG+92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, A. N. Wilschut: *PRISMA/DB: A Parallel Main Memory Relational DBMS.* IEEE Trans. Knowl. Data Eng. 4(6): 541-554 (1992)

[BACC+90] H. Boral, W. Alexander, L. Clay, G. P. Copeland, S. Danforth, M. J. Franklin, B. E. Hart, M. Smith, P. Valduriez: *Prototyping Bubba, A Highly Parallel Database System.* IEEE Trans. Knowl. Data Eng. 2(1): 4-24 (1990)

[Bau99] P. Baumann: *A Database Array Algebra for Spatio-Temporal Data and Beyond*. NGITS 1999: 76-93

[Bay97] R. Bayer: *The Universal B-Tree for Multidimensional Indexing: General Concepts*. WWCA Conference 1997

[BCV91] B. Bergsten, M. Couprie, P. Valduriez: *Prototyping DBS3, a Shared-Memory Parallel Database System.* PDIS 1991: 226-234

[BDFRW98] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann: *The Multidimensional Database System RasDaMan*. SIGMOD Conference 1998: 575-577

[BDFRW99] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, N. Widmann: *Spatio-Temporal Retrieval with RasDaMan*. VLDB 1999: 746-749

[BFRW97] P. Baumann, P. Furtado, R. Ritsch, N. Widmann: *Geo/Environmental and Medical Data Management in the RasDaMan System*. VLDB 1997: 548-552

[BJLM+96] G. Bozas, M. Jaedicke, A. Listl, B. Mitschang, A. Reiser, S. Zimmermann: *On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project.* Euro-Par, Vol. II 1996: 881-886

[CD94] G. W. Cook, E. J. Delp: *An investigation of JPEG image and video compression using parallel processing.* International Conference on Acoustics, Speech, and Signal Processing, 1994, 437-440.

[CDG04] T. Cruanes, B. Dageville, B. Ghosh: *Parallel SQL Execution in Oracle 10g*. SIGMOD Conference 2004: 850-854

[Cha98] D. Chamberlin: *A Complete Guide to DB2 Universal Database.* Morgan Kaufmann 1998

[Che75] P. P. Chen: *The Enity-Relationship Model: Toward a Unified View of Data.* VLDB 1975: 173

[CJMN+97] A. Clausnitzer, M. Jaedicke, B. Mitschang, C. Nippl, A. Reiser, S. Zimmermann: *On the Application of Parallel Database Technology for Large Scale Document Management Systems.* IDEAS 1997: 388-396

[CO98] F. Cariño, W. O'Connell: *Plan-Per-Tuple Optimization Solution - Parallel Execution of Expensive User-Defined Functions.* VLDB 1998: 690-695

[Cod70] E. F. Codd: *A Relational Model of Data for Large Shared Data Banks*. *CACM*, 13(6): 377-387 (1970)

[DB00] A. Dehmel, P. Baumann: *Visualizing Multidimensional Raster Data with rView.* DEXA Workshop 2000: 677-685

[Deh01] A. Dehmel: *Designing a Compression Engine for Multidimensional Raster Data*. DEXA 2001: 470-480

[Deh02] A. Dehmel: *A Compression Engine for Multidimensional Array Database Systems*. PhD Thesis, Technische Universität München, 2002

[DG90] D. J. DeWitt, J. Gray: *Parallel Database Systems: The Future of Database Processing or a Passing Fad?* SIGMOD Record 19(4): 104-112 (1990)

[DGSB+90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, R. Rasmussen: *The Gamma Database Machine Project.* IEEE Trans. Knowl. Data Eng. 2(1): 44-62 (1990)

[EOS03] *The EOS Homepage.* http://eos.gsfc.nasa.gov

[FB99] P. Furtado, P. Baumann: *Storage of Multidimensional Arrays Based on Arbitrary Tiling*. ICDE 1999: 480-489

[Fur99] P.A. Furtado: *Storage Management of Multidimensional Arrays in Database Management Systems*. PhD Thesis, Technische Universität München, 1999

[GG98] V. Gaede, O. Günther: *Multidimensional Access Methods*. ACM Computing Surveys 30(2): 170-231 (1998)

[GLS99] W. Gropp, E. Lusk, A. Skjellum: *Using MPI: Portable Parallel Programming with the Message-Passing Interface, second edition.* Massachusetts Institute of Technology, 1999

[GLT99] W. Gropp, E. Lusk, R. Thakur: *Using MPI-2: Advanced Features of the Message-Passing Interface.* Massachusetts Institute of Technology, 1999

[Gra90] G. Graefe: *Encapsulation of Parallelism in the Volcano Query Processing System.* SIGMOD Conference 1990: 102-111

[Gra93] G. Graefe: *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys 25(2): 73-170 (1993)

[GW87] R. Ganski, H. Wong: *Optimization of Nested SQL Queries Revisited.* SIGMOD Conference 1987: 23-33

[Hal97] G. Hallmark: *Oracle Parallel Warehouse Server*. ICDE 1997: 314-320

[HR01] T. Härder, E. Rahm: *Datenbaksysteme: Konzepte und Techniken der Implementierung.* 2. Auflage. Springer 2001

[HRH03] K. Hahn, B. Reiner, G. Höfling: *Parallel Query Support for Multidimensional Data: Intra-object Parallelism.* DEXA 2003: 212-222

[HRHB02] K. Hahn, B. Reiner, G. Höfling, P. Baumann: *Parallel Query Support for Multidimensional Data: Inter-object Parallelism*. DEXA 2002: 820-830

[IBM98] IBM Corporation: *Meeting Your Scalability Needs with IBM DB2 Universal Database Enterprise – Extended Edition for Solaris Operationg Environment.* IBM White Paper 1998

[Inf] InfiniBand Trade Association, http://www.infinibandta.org

[Inm02] W. H. Inmon: *Building the Data Warehouse*, 3rd Edition, John Wiley & Sons, 2002

[Inm96] W. H. Inmon: *The Data Warehouse and Data Mining.* CACM, 39(11): 49-50 (1996)

[ISO92] ANSI/ISO *SQL92, ISO/IEC 9075:1992(E) Information Technology - Database Languages - SQL*

[Jae99] M. Jaedicke: *New Concepts for Parallel Object-Relational Query Processing.* PhD Thesis, Institut für parallele und Verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart, 1999

[JM98] M. Jaedicke, B. Mitschang: *On Parallel Processing of Aggregate and Scalar Functions in Object-Relational DBMS.* SIGMOD Conference 1998: 379-389

[JM99] M. Jaedicke, B. Mitschang: *User-Defined Table Operators: Enhancing Extensibility for ORDBMS.* VLDB 1999: 494-505

[JMP97] A. Jhingran, T. Malkemus, S. Padmanabhan: *Query Optimization in DB2 Parallel Edition.* IEEE Data Eng. Bull. 20(2): 27-34 (1997)

[KE04] A. Kemper, A. Eickler: *Datenbanksysteme - Eine Einführung.* 5. Auflage. Oldenbourg 2004

[Kim82] W. Kim: *On Optimizing an SQL-like Nested Query.* ACM Trans. Database Syst. 7(3): 443-469 (1982)

[LAM] Local Area Multicomputer, http://www.lam-mpi.org

[Lav03] O. Lavsa: *Entwicklung und Implementierung von Object Framing zum adaptiven Zugriff auf multidimensionale Datenbanken und zur erweiterten Datenanalyse*. Diplomarbeit, Technische Universität München, 2003.

[LKOT+00] M. Lee, M. Kitsuregawa, B. C. Ooi, K. Tan, A. Mondal: *Towards Self-Tuning Data Placement in Parallel Database Systems.* SIGMOD Conference 2000: 225-236

[LMW96] L. Libkin, R. Machlin, L. Wong: *A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques.* SIGMOD Conference 1996: 228-239

[Mär01] H. Märtens: *A Classification of Skew Effects in Parallel Database Systems.* Euro-Par 2001: 291-300

[MD97] M. Mehta, D. J. DeWitt: *Data Placement in Shared-Nothing Parallel Database Systems.* VLDB J. 6(1): 53-72 (1997)

[Mil03] O. Milz: *Top-K Vektor Anfragen im multidimensionalen Datenbanksystem RasDaMan*. Diplomarbeit, Technische Universität München, 2003.

[MPI] Message Passing Interface, http://www.mpi-forum.org

[MS02] A. Marathe, K. Salem: *Query processing techniques for arrays.* VLDB J. 11(1): 68-91 (2002)

[MS97] A. Marathe, K. Salem: *A Language for Manipulating Arrays.* VLDB 1997: 46-55

[MS99] A. Marathe, K. Salem: *Query Processing Techniques for Arrays.* SIGMOD Conference 1999: 323-334

[Myr] Myrinet, http://www.myricom.com/myrinet, American National Standard ANSI/VITA 26-1998

[Nip00] C. Nippl: *Providing efficient, extensible and adaptive intra-query parallelism for advanced applications*. PhD Thesis, Technische Universität München, 2000

[Nor00] K. Nørvåg: *Vagabond – The Design and Analysis of a Temporal Object Management System.* PhD Thesis, Norwegian University of Science and Technology, 2000

[Obj99] *Objectivity technical overview*, version 5, 1999

[OCL99] W. O'Connell, F. Cariño, G. Linderman: *Optimizer and Parallel Engine Extensions for Handling Expensive Methods Based on Large Objects.* ICDE 1999: 304-313

[Ols92] M. A. Olson: *Extending the POSTGRES Database System to Manage Tertiary Storage.* Master Thesis, University of California, Berkeley, USA, 1992

[OMP] http://www.openmp.org

[Ora03] *Achieving Mainframe-Class Performance on Intel Servers Using InfiniBand Building Blocks.* Oracle White Paper, April 2003

[OV99] M. T. Özsu, P. Valduriez: *Principles of Distributed Database Systems, Second Edition.* Prentice-Hall 1999

[PCIX] PCI-X 2.0, http://www.pcisig.com/specifications/pcix_20

[PVM] Parallel Virtual Machine, http://www.csm.ornl.gov/pvm/pvm_home.html

[Rah94] E. Rahm: *Mehrrechner-Datenbanksysteme – Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Addison-Wesley 1994

[Rei05] B. Reiner: *HEAVEN – Eine hierarchische Speicher- und Archivierungsumgebung für multidimensionale Array Datenbankmanagement Systeme.* PhD Thesis, Technische Universität München, 2005

[RH04] B. Reiner, K. Hahn: *Optimized Management of Large-Scale Datasets stored on Tertiary Storage Systems*. IEEE Distributed Systems Online Magazine, March 2004

[RHHB02] B. Reiner, K. Hahn, G. Höfling, P. Baumann: *Hierarchical Storage Support and Management for Large-Scale Multidimensional Array Database Management Systems*. DEXA 2002: 689-700

[Rit99] R. Ritsch: *Optimization and Evaluation of Array Queries in Database Management Systems*. PhD Thesis, Technische Universität München, 1999

[SA02] E. Stolte, G. Alonso: Efficient Exploration of Large Scientific Databases. VLDB 2002: 622-633

[SBHD98] C. Sapia, M. Blaschka, G. Höfling, B. Dinter: *Extending the E/R Model for the Multidimensional Paradigm.* ER Workshops 1998: 105-116

[SETI] SETI Institute: http://www.seti.org/

[SKPO88] M. Stonebraker, R. H. Katz, D. A. Patterson, J. K. Ousterhout: *The Design of XPRS.* VLDB 1988: 318-330

[SRF87] T. Sellis, N. Roussopoulos, C. Faloutsos: *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. VLDB 1987: 507-518

[SS94] S. Sarawagi, M. Stonebraker: *Efficient Organization of Large Multidimensional Arrays.* ICDE 1994: 328-336

[SUN] Sun Microsystems, http://www.sun.com

[SWZ98] P. Scheuermann, G. Weikum, P. Zabback: *Data Partitioning and Load Balancing in Parallel Disk Systems.* VLDB J. 7(1): 48-66 (1998)

[Tan00] A. S. Tanenbaum: *Computernetzwerke.* 3. Auflage, Addison-Wesley, 2000

[TPC] Transaction Processing Performance Council, http://www.tpc.org

[Tur] Alan Turing: http://www.turing.org.uk/

[WB99] N. Widmann, P. Baumann: *Performance Evaluation of Multidimensional Array Storage Techniques in Databases*. IDEAS 1999: 385-389

[Wid00] N. Widmann: *Efficient Operation Execution on Multidimensional Array Data*. PhD Thesis, Technische Universität München, 2000

[Zim00] S. Zimmermann: *Evaluierung und Tuning von parallelen Datenbanksystemen.* PhD Thesis, Technische Universität München, 2000

[Zou97] C. Zou: *XPS: A High Performance Parallel Database Server.* IEEE Data Eng. Bull. 20(2): 21-26 (1997)

Appendix A Notation

Name	Symbol	Schreibweise	Beschreibung
Punkt, Vektor, Position	$\underline{l}, \underline{h}, \underline{x} \in \mathbb{Z}^d$	$[30, 10, 20] \in \mathbb{Z}^3$	Position im d-dimensionalen Raum
Intervall	$D = \{\underline{x} \mid \underline{l} \leq \underline{x} \leq \underline{h}\}$	D = [10:20,10:30]	Intervall im d-dimensionalen Raum, Menge der darin enthaltenen Punkte
Basisdatentyp	T	{float;{int; int; int}}	Menge aller Werte, die diesem Typ entsprechen
Wert des Typs T	$v \in T$	{25,3756; {230;10;40}}	Konkreter Wert v des Basisdatentyps T
MDD	α, β, χ	([0, 10, 20]; 55,175), ([0, 10, 21]; 58,597)	Abbildung f: D → T, die jedem Punkt der Domäne einen Wert zuordnet
Zelle	$(\underline{x}, v(\underline{x}))$	([0, 10, 20]; 55,175)	Zelle, bestehend aus Vektor und Zellwert
Wert einer Zelle	$\alpha[\underline{x}]$	([0, 10, 20]; 55,175)	Abbildung f: $\mathbb{Z}^d$ → T; Wert eines Punktes $\underline{x}$ im MDD α
Domäne des MDD α	$sdom(\alpha) = D$	$sdom(\alpha)$ = [0:100,0:50]	Intervall (spatial domain), auf dem das MDD α definiert ist
Basistyp des MDD α	$base(\alpha) = T$	$base(\alpha)$ = {float}	Datentyp (base type) jeder Zelle des MDD
Untere Begrenzung von α	$low(\alpha) = \underline{l}$ $low(D) = \underline{l}$	$low(\alpha)$ = [0, 0]	Abkürzendes Schreibweise, eigentlich $low(sdom(\alpha))$

Obere Begrenzung von α	$high(\alpha) = \underline{h}$ $high(D) = \underline{h}$	$high(\alpha) = [100, 50]$	Abkürzendes Schreibweise, eigentlich $high(sdom(\alpha))$
Ausdehnung von α	$extent(\alpha) = D$ $extent(D) = D'$	$extent(\alpha) = [101, 51]$	Abkürzendes Schreibweise, eigentlich $extent(sdom(\alpha))$
Dimensionalität von α	$dim(\alpha) = d$ $dim(D) = d$	$dim(\alpha) = 2$	Abkürzendes Schreibweise, eigentlich $dim(sdom(\alpha))$
Anzahl der Punkte in α	$\lvert\alpha\rvert$ $\lvert D\rvert$	$\lvert\alpha\rvert = 5151$	Abkürzendes Schreibweise, eigentlich $\lvert sdom(\alpha)\rvert$
Typ des MDD α	$type(\alpha) =$ <D, T>	<[0:127, 0:63, 0:119], {float; {int; int; int}}>	Typ eines MDD, bestehend aus Domäne und Basistyp
Kachel eines MDD α	τ	$\tau \in \alpha$	Kachel eines MDD, mit $sdom(\tau) < sdom(\alpha)$

Appendix B Abbildungsverzeichnis

Abbildung 1.1: Beispiel für Applikationen aus den Bereichen Data Warehouse, Online Analytical Processing, Geografische Informationssysteme, Array DBMS 3
Abbildung 1.2: 4D Rasterdaten, Simulation eines Klimamodells: Windgeschwindigkeiten in äquatorialer Richtung von Januar 1860 bis Dezember 2099 4
Abbildung 1.3: ESTEDI Applikationsarchitektur 9
Abbildung 1.4: ESTEDI Applikationen 12
Abbildung 2.1 Beispiel für einen multidimensionalen Datentyp 18
Abbildung 2.2 Beispiel für die Visualisierung eines Datenobjektes 20
Abbildung 2.3: Beispiel für geometrische Operationen 26
Abbildung 2.4: Speicherung von 3D Daten in einem RDBMS als Vektordaten 32
Abbildung 2.5: Speicherung von 2D Daten als Array 33
Abbildung 2.6: Kachelung von MDD: regulär, ausgerichtet, arbiträr 34
Abbildung 2.7: Architektur des rasdaman DBMS 36
Abbildung 2.8: Beispiel für Anfragebaum 39
Abbildung 3.1: Beispiel für Amdahls Gesetz 44
Abbildung 3.2: Parallele Architekturen für DBMS 47
Abbildung 3.3: Pipeline- und Datenparallelität 50
Abbildung 3.4: Klassifizierung von Parallelität in RDBMS 51
Abbildung 3.5: Anfrageverarbeitung in einem parallelen DBMS 53
Abbildung 3.6: Phasen der Parallelisierung eines Anfragebaumes 54
Abbildung 4.1: Generelle Unterscheidung der Parallelisierungsstrategien 59
Abbildung 4.2: Beispiel für die komplexe Struktur von Array Daten 61
Abbildung 4.3: Vergleich von Prädikatbaum des Selektionsprädikats für relationale Selektion und Selektion von Arrays 63
Abbildung 4.4: Datenfluss im Anfragebaum 65
Abbildung 4.5: Probleme der Pipeline Parallelität bei Anfragen auf Array Daten 69
Abbildung 4.6: „Bushy Tree“ - Anfragebaum 72
Abbildung 4.7: Anfragebaum und Ausführungsplan (Intra-Operator Parallelität) 74
Abbildung 4.8: Klassifizierung von einfachen Anfragen an Array Mengen 80
Abbildung 4.9: Verteilung der Arbeitslast auf Prozesse 82
Abbildung 4.10: Optimierung von gemeinsamen Teilausdrücken 84
Abbildung 4.11: Optimierung durch Verschieben der Selektion 86
Abbildung 4.12: Cache für Selektionsergebnisse bei paralleler Verarbeitung 87
Abbildung 4.13: Parallele Verarbeitung von verschachtelten Anfragen 91
Abbildung 4.14: Beispiel für Zuteilung von Tupel zu Prozessen 93
Abbildung 4.15: Beispiel für Zuteilung von 2-Tupeln zu Prozessen 95
Abbildung 4.16: Optimierung der Datenzuteilung durch den Tupel Server Prozess 96
Abbildung 4.17: Funktionalität der Iteratoren send / receive 98
Abbildung 4.18: Adaption eines typischen Anfragebaumes 100

Abbildung 4.19: Integration einer parallelisierten Anfrage in das Iterator-Modell 102
Abbildung 4.20: Klassifizierung der Intra-Operator Parallelisierung bezüglich Verteilungsstrategien 104
Abbildung 4.21: Speed Up bei Fixierung des Parallelisierungsgrad und Skalierung der Objektanzahl..... 107
Abbildung 4.22: Anfragebaum und Ausführungsplan (Intra-Objekt Parallelität) 108
Abbildung 4.23: Skalierung multidimensionaler Arrays 114
Abbildung 4.24: Partitionierung und Berechnung des finalen Resultats für Operatorverknüpfungen 116
Abbildung 4.25: Funktionalität der Operatoren split und merge 117
Abbildung 4.26: Adaption des Anfragebaumes für Intra-Objekt Parallelisierung..... 121
Abbildung 4.27: Partitionierung von multidimensionalen Arrays 123
Abbildung 4.28: Partitionierung in Kachelmengen bei Übergabe von Arrays 126
Abbildung 4.29: Partitionierung in Subarrays 128
Abbildung 4.30: Optimierung von E/A bei Partitionierung..... 130
Abbildung 4.31: Optimierung von E/A bei paralleler Verarbeitung durch Clustering mittels Z-Kurve 132
Abbildung 4.32: Redundanz von E/A beim Partitionieren von Arrays 134
Abbildung 4.33: Beispiel für Anwendung einer binär induzierten Operation 136
Abbildung 4.34: Konträre Partitionierung für binär induzierte Operationen..... 137
Abbildung 4.35: Ablaufdiagramm für Early Termination bei Quantoren..... 142
Abbildung 4.36: Klassifizierung der Intra-Objekt Parallelisierung bezüglich Verteilungsstrategien..... 145
Abbildung 4.37: Ablaufdiagramm für Inter-Objekt Parallelisierung..... 146
Abbildung 4.38: Abhängigkeit der Parallelisierungsstrategie von Kardinalität der zu verarbeitenden Menge und dem Parallelitätsgrad 147
Abbildung 4.39: Adaption der Parallelisierungsstrategie während der Ausführung 150
Abbildung 5.1: Architektur des parallelen Array DBMS rasdaman..... 160
Abbildung 5.2: Modulare Schichtenarchitektur von Master und internen Prozessen..... 161
Abbildung 5.3: Start des parallelen rasdaman Server..... 162
Abbildung 5.4: Anfrageverarbeitung im parallelen rasdaman Server 163
Abbildung 6.1: Visualisierung der 2D Testdaten (Satellitenbilder)..... 167
Abbildung 6.2: 3D Testdaten (Simulation der globalen Erderwärmung) 168
Abbildung 6.3: Repräsentation eines Windvektors..... 169
Abbildung 6.4: 4D Testdaten (Simulation der globalen Windgeschwindigkeiten) 169
Abbildung 6.5: Parallelisierung der Kosten für E/A und CPU einer Anfrage 171
Abbildung 6.6: Shared-Everything Architektur mit nicht-lokalem RDBMS Server (Messumgebung) 173
Abbildung 6.7: Speed-Up von rasdaman PE unter Nutzung einer Shared-Everything Architektur mit Zugriff auf RDBMS Server über 100MBit Netzwerk..... 174
Abbildung 6.8: Einfluss der Anfragekomplexität auf den Speed-Up (Shared-Everything) 175
Abbildung 6.9: Einfluss des Datenvolumens der Inter-Prozess-Kommunikation (Shared-Everything)..... 177
Abbildung 6.10: Einfluss des Kommunikationsmoduls für die Inter-Prozess-Kommunikation (Shared-Everything)..... 177
Abbildung 6.11: Einfluss der Datenkompression auf den Speed-Up (Shared-Everything)..... 179
Abbildung 6.12: Speed-Up bei Intra-Objekt Parallelisierung (Shared-Everything)..... 180
Abbildung 6.13: Entwicklung des Speed-Up mit Anfragekardinalität (Shared-Everything) 181
Abbildung 6.14: Shared-Disk Architektur (Messumgebung)..... 183
Abbildung 6.15: Speed-Up von rasdaman PE unter Nutzung einer Shared-Disk Architektur mit 100MBit Netzwerk 184
Abbildung 6.16: Einfluss der Anfragekomplexität auf den Speed-Up (Shared-Disk)..... 185
Abbildung 6.17: Einfluss der Inter-Prozess-Kommunikation (Shared-Disk)..... 186
Abbildung 6.18: Einfluss der Datenkompression auf den Speed-Up (Shared-Disk)..... 186
Abbildung 6.19: Speed-Up bei Intra-Objekt Parallelisierung (Shared-Disk)..... 187
Abbildung 6.20: Einfluss der Anfragekardinalität auf den Speed-Up (Shared-Disk) 188

Appendix C Verzeichnis der Definitionen

Definition 2.1 Multidimensionales Intervall D 21
Definition 2.2 Basisdatentyp T eines multidimensionalen Objektes 22
Definition 2.3 Multidimensionales Objekt (MDD) α 22
Definition 2.4 Typ M eines multidimensionalen Objekts 23
Definition 2.5 Kollektion C von MDD 30
Definition 2.6: Kachelung eines MDD α 32
Definition 2.7: Kachelungsmethode eines MDD 33
Definition 2.8: Persistente und transiente Kacheln 35
Definition 2.9: Kardinalität einer Anfrage 37
Definition 3.1: Speed-Up 43
Definition 4.1: Partitionierung θ_p einer Arraymenge 76
Definition 4.2: Partitionierung θ_p eines Array 110

Appendix D Verzeichnis der Beispiele

Beispiel 3.1: Grenzen der Skalierbarkeit 43
Beispiel 4.1: Auswirkung von Datenzuteilung bezüglich Ladevolumen 92
Beispiel 4.2: Optimale Verteilung eines Kreuzprodukt bezüglich E/A 94
Beispiel 4.3: Einbußen von paralleler Performanz bei mengenbasierter Parallelität für Anfragen mit geringer Kardinalität 106
Beispiel 4.4: Intra-Objekt Parallelisierung einer Anfrage 115
Beispiel 4.5: Partitionierung eines Arrays in Subarrays 130
Beispiel 4.6: Auswirkungen von Array Funktionen auf den *merge* Operator am Beispiel des Existenzquantors *some* 140
Beispiel 4.7: Auswirkungen von Parallelität auf Array Operationen am Beispiel des arithmetischen Durchschnitts *avg* 142