Institut für Informatik der Technischen Universität
München

Evaluierung und Tuning von parallelen Datenbanksystemen

Stephan Zimmermann

Institut für Informatik der Technischen Universität
München

Evaluierung und Tuning von parallelen Datenbanksystemen

Stephan Zimmermann

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. Dr. h.c. W. Brauer
Prüfer der Dissertation:	1.	Univ.-Prof. R. Bayer, Ph.D.
	2.	Univ.-Prof. Dr. B. Mitschang, Universität Stuttgart

Die Dissertation wurde am 31.5.2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.11.2000 angenommen.

Kurzfassung

MIDAS ist ein Prototyp eines parallelen Shared-Disk-Datenbanksystems, welches aus dem sequentiellen Datenbanksystem TransBase entstand. Bei der Entwicklung von MIDAS mußten viele Komponenten des sequentiellen TransBase Systems ersetzt und neu implementiert werden. Dabei wurde insbesondere Wert gelegt auf die Integration von Konzepten und Algorithmen, die in anderen Arbeiten im Rahmen der Forschung an parallelen Datenbanksystemen in der Datenbankforschungsgruppe unter Leitung von Prof. Bayer und Prof. Mitschang entstanden sind.

Diese Arbeit befaßt sich zum einen mit einem Modell zur Fehlerbehandlung für Datenbanksysteme mit einem parallelen Verarbeitungsmodell, wie es MIDAS besitzt. Nachdem die Charakteristika des Ausführungsmodells genau analysiert und spezifiziert sind, wird der Entwurf eines Modells für die Fehlerbehandlung vorgestellt. Dabei werden unter anderem die zu betrachtenden Fehlerfälle beschrieben und die Möglichkeiten, Fehlertoleranz bereitzustellen erörtert. Es wird dargestellt, wie durch eine feinere Strukturierung der Transaktionen Vorteile im Fehlerfall zu erzielen sind. Ein Vergleich mit bereits existierenden Konzepten zeigt die zusätzlichen Möglichkeiten dieses Modells auf. An Hand von Beispielanfragen aus MIDAS wird die Fehlerbehandlung demonstriert und evaluiert.

Zum anderen beschreibt diese Arbeit die Entwicklung und jetzige Architektur von MIDAS und stellt Evaluierungen bezüglich der Effektivität und Effizienz der verwendeten Algorithmen und Konzepte vor. Die Entscheidungen bei der Implementierung der gewählten Architekturalternativen werden durch Messungen bestätigt. Bereiche, auf die sich diese Arbeit insbesondere konzentriert, sind das Laufzeitsystem mit Cacheverwaltung, Ein- und Ausgabe, Kommunikation und Skalierbarkeit.

Abstract

The MIDAS prototype is a parallel database system derived from the sequential database system TransBase. With the MIDAS system many components from the sequential TransBase system were replaced and newly implemented. The main focus was the integration of concepts and algorithms that were investigated by the research group for database systems led by Prof. Bayer and Prof. Mitschang.

This thesis presents an error handling model for database systems that have a parallel processing schema similar to MIDAS. First the characteristics of MIDAS' parallel processing schema are introduced. The concept of the error handling model is then presented, whereby the relevant fault situations and possibilities to provide fault tolerance are discussed. It is then demonstrated how to profit from a fine grained transaction structure in case of a system fault. The comparison with other existing transaction models shows the advantages of this new model. Using MIDAS sample queries this new model demonstrates and evaluates its error handling features.

The second part of this thesis describes the development and the present architecture of the MIDAS parallel database system. It shows several investigations of the effectiveness and efficiency of the integrated algorithms and concepts of the system. The implementation decisions are validated with detailed performance measurements. The main focus of this thesis is the runtime system with cache management, input and output, communication as well as scalability.

Danksagung

Mein herzlichster Dank gilt allen, die an der Entstehung und dem Abschluß dieser Arbeit beteiligt waren.

Prof. Bayer und Prof. Mitschang ermöglichten und betreuten meine Arbeit.

Giannis Bozas, Angelika Reiser und die gesamte Datenbankgruppe unterstützten durch die intensive fachliche Zusammenarbeit.

Giannis Bozas, Angelika Reiser, Sigrid Lanzl und Michael Jaedicke lasen meine Dissertation ausgiebig korrektur und brachten Verbesserungsvorschläge an.

Bei meinen Eltern Ingrid und Axel bedanke ich mich für ihre Unterstützung sowie die mehrfache Analyse der kryptischen Texte.

Durch umfangreiche Arbeiten am System und den Konzepten trugen desweiteren Michael Fleischhauer, Karsten Pries, Rolf Drügh, Matthias Tschaffler, Franz Brandmayer, Steffen Rost, Michaela Schult, Anastasia Voinou, Sabine Perathoner, Marcello Mariucci, Markus Fehr und Clara Nippl zum Gelingen dieser Arbeit bei.

Für tägliche Aufmunterungen und gute Stimmung sorgten insbesondere Giannis Bozas und Aiko Frank, wodurch speziell die schwierigen Zeiten während der Fertigstellung der Arbeit überbrückt werden konnten.

Anne Einenkel und Helmar Göttsch gelang es immer, die notwendige Systemumgebung für das MIDAS Projekt bereitzustellen.

Schließlich bedanke ich mich noch bei Herrn Kuss für seine aufmunternden Worte und die hervorragende Organisation aller vertraglichen Angelegenheiten.

Nochmals vielen Dank allen, speziell Sigrid und meinen Eltern, die stets ihr Vertrauen in mich setzten und unendliche Geduld mit mir hatten.

Inhaltsverzeichnis

1 Einführung **13**

2 Das parallele Datenbanksystem MIDAS **17**

2.1 Überblick 17

2.2 Von TransBase zu MIDAS 18

2.2.1 Das Datenbanksystem TransBase 18

2.2.2 Erste Schritte der Parallelisierung 21

2.3 Das relationale, parallele Datenbanksystem MIDAS 23

2.3.1 Das Anfragesystem 25

2.3.2 Das Ausführungssystem 28

2.3.3 Der MIDAS-Administrations-Server 29

2.3.4 Kommunikation im MIDAS-Server 30

2.4 Die Komponenten von MIDAS 30

2.4.1 Katalogmanager 30

2.4.2 Compiler 30

2.4.3 Optimierer 31

2.4.4 Parallelisierer 31

2.4.5 Scheduler 32

2.4.6 Interpreter 32

2.4.7 Pufferverwaltung 33

2.4.7.1 Ortstransparente Cacheverwaltung 33

2.4.7.2 Synchronisation und Sperrverwaltung 35

2.4.8 Ein- und Ausgabe 36

2.4.9 Kommunikationssegmente 36

2.5 Wiederverwendung von Software 39
2.6 Portierbarkeit von Anwendungen 40
2.7 Werkzeuge 41
2.8 Zusammenfassung 42

3 Implementierungsaspekte des Ausführungssystems 43

3.1 Portierbarkeit – Entwicklung von TransBase zu MIDAS 44
3.2 Prozeßdesign und Parallelität im System 46
3.2.1 Architekturumstellung 46
3.2.1.1 Das alte Architekturmodell 46
3.2.1.2 Das neue Architekturmodell 48
3.2.1.3 Bewertung der Architekturumstellung 50
3.2.2 Threads im Ausführungssystem 51
3.2.3 Synchronisation 52
3.2.3.1 Semaphore und Latches 52
3.2.3.2 Auswirkungen auf die parallele Architektur . . . 54
3.3 Kommunikation – PVM 56
3.3.1 Kommunikationscharakteristika von MIDAS 57
3.3.2 PVM-spezifische Optimierungen 59
3.3.3 PVM auf Mehrprozessorsystemen 61
3.3.4 Der Einsatz von PVM 62
3.4 Kosten der Kommunikationssegmente 63
3.5 Ein- und Ausgabe 65
3.5.1 Physische Datenverteilung 66
3.5.1.1 Aufteilung der Segmente – Partitionierung 66
3.5.1.2 Leistungsanalyse 68
3.5.1.3 Replikation 70
3.5.1.4 Zusammenfassung 71
3.5.2 Cache- und NFS-Problematik 72
3.5.3 NFS und Datenkohärenz 75
3.6 Pufferverwaltung – Sperrverwaltung und Kohärenzkontrolle 75
3.7 Cachebereiche – Rahmenkontingente 77

3.8 Zusammenfassung 77

4 Leistungsanalysen 79

4.1 Vergleich von MIDAS mit TransBase 79
4.1.1 Meßbedingungen 80
4.1.2 OLTP-Last 81
4.1.3 TPC–C-Last 83
4.1.4 Schreibende Anfragen 84
4.1.5 Zusammenfassung 85
4.2 Skalierbarkeit 86
4.2.1 Verteiltes Mehrrechnersystem 86
4.2.1.1 OLTP-Last 86
4.2.1.2 TPC–C-Last 87
4.2.2 Mehrprozessorsystem 88
4.2.2.1 OLTP-Last 89
4.2.2.2 TPC–C-Last 90
4.2.3 Intra-Transaktionsparallelität 91
4.2.4 Zusammenfassung 91
4.3 Ein- und Ausgabe 92
4.3.1 Zugriff auf Daten - Probleme durch NFS 92
4.3.1.1 OLTP-Last 93
4.3.1.2 TPC–C-Last 94
4.3.2 Zusammenfassung 95
4.3.3 Seiten und Subseiten 95
4.3.3.1 OLTP-Last 95
4.3.3.2 TPC–C-Last 97
4.3.3.3 Zusammenfassung 98
4.3.4 Datenbankcache und Festplattenzugriffe 98
4.3.4.1 OLTP-Last 99
4.3.4.2 TPC–C-Last 101
4.3.4.3 Zusammenfassung 102
4.4 Leistungsgrenzen 102

4.4.1 Maximaler Parallelitätsgrad 102
4.4.2 Mehrprozessorsystem 103
4.4.3 Mehrrechnersystem 104
4.4.4 Lokalitätsbasiertes Transaktionsrouting 105
4.5 Zusammenfassung 107

5 Aktivitätskontrolle und Modell zur Fehlerbehandlung 109

5.1 Motivation 109
5.2 Fehlerfälle und existierende Konzepte für Transaktionsmodelle . . 110
5.2.1 Fehlerklassen 111
5.2.2 Flache Transaktionen 111
5.2.3 Geschachtelte Transaktionen 113
5.2.4 Mehrebenentransaktionen 117
5.2.4.1 Einführung 117
5.2.4.2 Das Modell der Mehrebenentransaktionen 117
5.2.5 Sicherungspunkte 119
5.3 Ausführungsmodell in MIDAS – Aktivitätskontrolle und Fehlerbehandlung 121
5.3.1 Charakteristika des MIDAS-Ausführungsmodells 121
5.3.2 Aktivitätskontrolle 123
5.3.3 Fehlerbehandlung 124
5.4 Anforderungen an das Modell zur Fehlerbehandlung 125
5.5 Evaluierung der Modelle 126
5.5.1 Sicherungspunkte 126
5.5.2 Geschachtelte Transaktionen 127
5.5.3 Mehrebenentransaktionen 129
5.6 Änderungen am Modell der geschachtelten Transaktionen 130
5.7 Das Modell zur Fehlerbehandlung – FPT-Modell 132
5.7.1 Subtransaktionskonzept 133
5.7.1.1 Subtransaktionen 133
5.7.1.2 Datenflüsse 134
5.7.2 Fehler in Transaktionskomponenten 137

5.7.3 Die Fehlerbehandlung . . . 139
5.7.3.1 Reproduktion von Komponenten . . . 139
5.7.3.2 Die Fehlerbehandlung . . . 140
5.7.3.3 Fehlererkennung . . . 143
5.7.3.4 Algorithmus zur Fehlerbehandlung . . . 150
5.7.4 Bewertung des Modells . . . 151
5.7.5 Fallbeispiel . . . 153
5.7.6 Zusammenfassung . . . 158

6 Abbildung des Modells zur Fehlerbehandlung auf MIDAS 161

6.1 Abbildung der Transaktionsstruktur auf MIDAS . . . 161
6.1.1 Subtransaktionen . . . 162
6.1.2 Datenflüsse . . . 164
6.1.3 Transaktionsstruktur einer Beispielanfrage . . . 166
6.2 Fehlerbehandlung in MIDAS . . . 167
6.2.1 Fehlererkennung . . . 167
6.2.1.1 Transaktionsfehler . . . 167
6.2.1.2 Reproduktionsbedingte Fehler . . . 168
6.2.2 Fehlerbeseitigung . . . 170
6.3 Bewertung . . . 170
6.3.1 Ausbreitung von Datenfehlern . . . 170
6.3.2 Ausbreitung reproduktionsbedingter Fehler . . . 172
6.3.3 Betrachtung midasspezifischer Situationen . . . 176
6.3.4 Eingabeverfügbarkeit . . . 178
6.3.5 Maßnahmen zur Leistungssteigerung . . . 179
6.3.5.1 Kompensierende Datenflüsse . . . 179
6.3.5.2 Vermeidung von nichtdeterministischen Operatoren 180
6.4 Transaktionen mit mehreren Anfragen und ändernde Anfragen . . 181
6.4.1 Transaktionen mit mehreren Anfragen . . . 181
6.4.2 Ändernde Anfragen . . . 182
6.5 Leistungsanalysen . . . 183
6.5.1 Implementierung . . . 183

6.5.2 Fallbeispiel 184
6.6 Zusammenfassung 189

7 Zusammenfassung der Arbeit 191
7.1 Ergebnisse 191
7.1.1 Der Datenbankprototyp MIDAS 191
7.1.2 Das FPT-Modell 193
7.2 Ausblick 194

A Rechnerkonfigurationen 195

B TPC Benchmarks 197
B.1 TPC–A – TPC Benchmark A 197
B.2 TPC–C – TPC Benchmark C 198
B.3 TPC–D – TPC Benchmark D 201

C Datenbanken 203

D Transaktionen 205

E Abkürzungsverzeichnis 207

F Begriffsfestlegungen und Definitionen 209

Literaturverzeichnis 211

Abbildungsverzeichnis

2.1 Prozeßstruktur von TransBase 19
2.2 Aufbau des TransBase-Kernels 20
2.3 MIDAS–Architektur 24
2.4 Komponenten des Anfragesystems 26
2.5 Funktionsweise des Applikationsservers 27
2.6 Komponenten des Ausführungssystems 28
2.7 Virtueller DatenBank-Cache – VDBC 34

3.1 Aufbau eines Segments 45
3.2 Das alte Architekturmodell 47
3.3 Das neue Architekturmodell 49
3.4 Semaphore versus Latches 54
3.5 Gesperrte Segmentschicht 55
3.6 Varianten der Kommunikationssegmente 63
3.7 Verteilung eines Segments auf mehrere Verzeichnisse 68
3.8 Laufzeitveränderungen bei Verteilung auf mehrere Festplatten . . 69
3.9 Cacheeffekte bei steigender Rahmenzahl 74
3.10 Nachrichten der Kohärenzkontrolle und Sperrverwaltung 76

4.1 TransBase, MIDAS – Laufzeiten und Durchsatz unter TPC–A^{RO} . 81
4.2 TransBase, MIDAS – Laufzeiten und Durchsatz unter TPC–C^{S+O} 83
4.3 Durchschnittliche Zeiten für eine Transaktion TPC–A 85
4.4 Mehrrechnersystem – Skalierbarkeit unter OLTP-Last 86
4.5 Mehrrechnersystem – Skalierbarkeit unter TPC–C-Last 88
4.6 Mehrprozessorsystem – Skalierbarkeit unter OLTP-Last 89

4.7 Mehrprozessorsystem – Skalierbarkeit unter TPC–C-Last 90
4.8 Gewinn durch Replikation – OLTP-Last 93
4.9 Gewinn durch Replikation – TPC–C-Last 94
4.10 Verschiedene Seitengrößen unter OLTP-Last 96
4.11 Verschiedene Seitengrößen unter TPC–C-Last 97
4.12 Anzahl Festplattenzugriffe und mittlere Lesezeit TPC–A^{RO} 99
4.13 Anzahl Festplattenzugriffe und mittlere Lesezeit TPC–C 101
4.14 Maximaler Durchsatz auf Mehrprozessorsystem 103
4.15 Maximaler Durchsatz auf Mehrrechnersystem 104
4.16 Lokalitätsbasiertes Transaktionsrouting 106

5.1 Baumstruktur einer geschachtelten Transaktion 114
5.2 Mehrebenentransaktionen 119
5.3 Intra-Transaktionsparallelität 122
5.4 Parallelisierung und Transaktionen 124
5.5 Paralleler Ausführungsplan und Prozeßkontrollstruktur 128
5.6 Das neue Rücksetzverhalten 131
5.7 Transaktion als Baum von Subtransaktionen 134
5.8 Datenflüsse in Transaktionsstruktur 135
5.9 Zustandsübergänge von Datenflüssen 137
5.10 Ausbreitung von Datenfehlern 144
5.11 Reproduktion mit Verlust von Eingabedaten 146
5.12 Störung des Informationsflusses 148
5.13 Reproduktionsfehler bei nichtdeterministischen Komponenten . . 149
5.14 Bedeutung der Position nichtdeterministischer Komponenten . . . 152
5.15 Beispiel der Fehlerbehandlung 154
5.16 Beispiel 2 – potentiell vollständige Ausgabe von G 158

6.1 Ausführungseinheiten und Subtransaktionen 162
6.2 Abbildung von Subtransaktionen auf Ausführungseinheiten 163
6.3 Nichtdeterministische und blockierende Subtransaktionen 164
6.4 Kommunikationsformen 165

6.5 Transaktionsstruktur von TPC–D Anfrage 3 166
6.6 Fehlerausbreitung bei Pipelining und Mischen 171
6.7 Fehlerausbreitung bei Replikation 171
6.8 Fehlerausbreitung bei Partitionierung 172
6.9 Reproduktionsbedingte Fehler bei Pipelining 173
6.10 Reproduktionsbedingte Fehler bei einer Partitionierung 174
6.11 Reproduktionsbedingte Fehler beim Mischen 175
6.12 Verzweigung von Operatorbäumen und blockierende Operatoren . 177
6.13 Subtransaktionsbaum der TPC–D-Anfrage 9 185
6.14 Laufzeitverhalten der TPC–D-Anfrage 9 186
6.15 TPC–D-Anfrage 9 – Potentielle Vollständigkeit 187
6.16 TPC–D-Anfrage 9 – Kompensierende Datenflüsse 188

B.1 Logische Struktur der TPC–C Datenbank 198
B.2 Schema der TPC–D Datenbank 201

Tabellenverzeichnis

2.1 Wiederverwendung der TransBase-Quellen 39

3.1 Nachrichtenaufkommen bei alter und neuer Architektur 51
3.2 Nachrichtendurchsatz in MIDAS 59
3.3 PVM Parameter *RouteDirect*, *DataDefault* und *DataRaw* 60
3.4 Laufzeiten mit verschiedenen PVM-Bibliotheken 61
3.5 Kosten der Kommunikationssegmente 65
3.6 Durchsatz bei Verteilung der Segmente 70

4.1 Gegenüberstellung der Meßbedingungen in TransBase und MIDAS 80
4.2 Seitenzugriffsstatistik verschiedener Cachegrößen, TPC–A^{RO} . . . 100
4.3 Seitenzugriffsstatistik verschiedener Cachegrößen, TPC–C 101

B.1 Transaktionsmix TPC-Benchmark C 200

C.1 TPC–A-Datenbanken 203
C.2 TPC–C-Datenbank 204
C.3 TPC–D-Datenbank 204

Kapitel 1

Einführung

Datenbanksysteme sehen sich mit immer weiter anwachsenden Datenmengen und immer komplexer werdenden Anwendungen konfrontiert. Insbesondere in neuen Anwendungsgebieten wie der Dokumentenverwaltung, den Multimediaanwendungen oder bei Data Warehouses sind die Grenzen eines Datenbanksystems hinsichtlich Datenkapazität, gewünschter Transaktionsraten und erforderlicher Antwortzeiten schnell erreicht.

Eine Möglichkeit, diesen immer höher werdenden Leistungsanforderungen gerecht zu werden, besteht in der Einführung von Parallelität in den Systemen. Die entstehenden parallelen Datenbanksysteme werden auf mehreren Rechnern oder auf Mehrprozessorsystemen eingesetzt. Die dabei verwandte Software-Architektur verspricht, zusammen mit geeigneten Parallelisierungstechniken, sowohl einen höheren Gesamtdurchsatz als auch verkürzte Antwortzeiten für einzelne Transaktionen.

Vor diesem Hintergrund wurden an der Technischen Universität München, im Sonderforschungsbereich SFB 342 am Lehrstuhl von Prof. Bayer, im Teilprojekt B2, Konzepte zur Parallelisierung von Datenbanksystemen entwickelt und an dem Datenbankprototyp MIDAS erprobt und bewertet.

MIDAS ist der Prototyp eines relationalen, parallelen Shared-Disk-Datenbanksystems. In MIDAS sollten insbesondere verschiedene Formen von Parallelität, wie Inter-Transaktionsparallelität und Intra-Transaktionsparallelität, eingeführt werden. Bei dem Parallelisierungsansatz der Inter-Transaktionsparallelität werden Transaktionen nicht mehr nur quasiparallel auf einem Prozessor ausgeführt. Vielmehr wird durch Prozeßreplizierung auf mehrere Prozessoren eine echte Parallelausführung der einzelnen Transaktionen erreicht. Daraus resultiert eine Steigerung der Transaktionsrate. Ziel des Parallelisierungsansatzes der Intra-Transaktionsparallelität ist es, den Ausführungsplan einer einzelnen Datenbankanfrage parallel abzuarbeiten. Hierbei wird auf Daten- und Pipeline-Parallelität zurückgegriffen, wodurch eine Verkürzung der Antwortzeit gewonnen wird.

Bei der Entwicklung von MIDAS wurde ein evolutionärer Ansatz verfolgt. MIDAS wurde nicht komplett neu implementiert, sondern es wurden große Teile des Codes und der Systemkomponenten des sequentiellen, relationalen Datenbanksystems TransBase wiederverwendet. Durch schrittweises Erweitern, Ergänzen und Umstrukturieren konnten die verschiedenen Formen der Parallelität sukzessive eingebaut werden. Dabei wurde insbesondere Wert gelegt auf die Integration von Konzepten und Algorithmen, die in anderen Arbeiten im Rahmen der Forschung an parallelen Datenbanksystemen am Lehrstuhl entstanden sind. Auf diese Weise konnte sehr schnell ein lauffähiger Prototyp entwickelt werden, der als Implementierungs- und Testplattform für verschiedene parallele Datenbanktechnologien dient. An ihm werden die entwickelten Konzepte und Algorithmen erprobt und evaluiert.

Zu den Gebieten, die bei der Entwicklung von MIDAS von besonderem Interesse waren, zählen die Optimierung und Parallelisierung von Anfragen [Nippl 00], die Erweiterung der Datenbankfunktionalität um Konzepte wie benutzerdefinierte Funktionen [Jaedicke 99], der Entwurf von Algorithmen zur Pufferverwaltung, Kohärenzkontrolle und Synchronisation [Listl 96, Bozas 98] und die Integration dieser Konzepte in einer gemeinsamen Systemarchitektur.

Der erste Teil dieser Arbeit beinhaltet die Realisierung der Systemarchitektur von MIDAS. Besonderer Wert wird auf die Gestaltung der Systemarchitektur gelegt, die die verschiedenen Formen der Parallelität ermöglicht. Schwerpunkte liegen zusätzlich beim Laufzeitsystem mit Cacheverwaltung, der Ein- und Ausgabekomponente sowie den Kommunikationsmechanismen im System. Weiter werden Untersuchungen zum Umfang der Wiederverwendung von Software, der erreichbaren Skalierbarkeit des Shared-Disk-Datenbanksystems MIDAS und der Effizienz von Pufferverwaltung und Kommunikation durchgeführt.

Der zweite Teil der Arbeit beschäftigt sich mit einem Modell zur Fehlerbehandlung, das in einem System mit paralleler Anfrageverarbeitung, wie MIDAS, eingesetzt werden kann. Es wird erörtert, wie durch die Strukturierung von Transaktionen im Fehlerfall Teile der bereits verrichteten Arbeit gerettet und beim Wiederaufsetzen der Transaktion erneut verwendet werden können.

Die Arbeit ist im einzelnen wie folgt gegliedert:

Kapitel 2 stellt das parallele Datenbanksystem MIDAS vor. Zu Beginn steht die Beschreibung der Entwicklung von TransBase zu MIDAS. Anschließend wird der derzeitige Stand der Implementierung von MIDAS dokumentiert sowie eine Analyse des Ansatzes der Wiederverwendung von Software vorgenommen.

Kapitel 3 präsentiert speziell ausgewählte Aspekte der Implementierung des Laufzeitsystems von MIDAS. Insbesondere beschäftigt es sich mit Implementierungsdetails, die für die Effizienz des Laufzeitsystems von Bedeutung sind. Es wird beschrieben, wie die Portierung von Anwendungen durch das Design der Puffer-

verwaltung unterstützt wird, welche Kommunikationscharakteristika das System auszeichnen, wie effizient diese realisiert wurden und welche Konsequenzen sich daraus ergeben. Daran schließt sich die Beschreibung der Datenverteilung auf Festplatten an. Ferner wird die Problematik erläutert, die sich aus den Wechselwirkungen des Datenbanksystems mit dem Betriebssystem sowie dem Einsatz des Network File Systems (NFS) ergibt. Die Entscheidungen bei der Implementierung der gewählten Architekturalternativen werden durch Messungen bestätigt.

In Kapitel 4 werden umfangreiche Leistungsanalysen des Systems vorgestellt und es wird der Übergang von TransBase zu MIDAS bewertet. Anschließend erfolgt die Evaluierung der Skalierbarkeit und des Ein- und Ausgabeverhaltens. Die Systemleistungsgrenzen werden bestimmt und weitere Optimierungsmöglichkeiten aufgezeigt.

Mit Kapitel 5 beginnt der zweite Teil der Arbeit. Es wird das im Rahmen dieser Arbeit entworfene FPT-Modell, ein Modell zur **F**ehlerbehandlung in **p**arallelen **T**ransaktionen, vorgestellt. Es ist für die Fehlerbehandlung in Datenbanksystemen mit einem parallelen Verarbeitungsmodell, wie es MIDAS besitzt, konzipiert. Nachdem die Charakteristika des Verarbeitungsmodells von MIDAS genau analysiert und spezifiziert sind, wird der daran angepaßte Entwurf des Modells für die Fehlerbehandlung beschrieben. Dabei werden unter anderem die zu betrachtenden Fehlerfälle aufgeführt und die Möglichkeiten vorgestellt, Fehlertoleranz bereitzustellen. Es wird erörtert, wie durch eine feinere Strukturierung der Transaktionen Vorteile im Fehlerfall zu erzielen sind. Ein Vergleich mit bereits existierenden Konzepten zeigt die zusätzlichen Möglichkeiten des FPT-Modells auf.

Kapitel 6 behandelt die Abbildung des FPT-Modells auf das reale System MIDAS. Es werden spezielle Situationen und Konsequenzen aufgezeigt, die sich beim Einsatz in MIDAS ergeben. An Hand von Beispielanfragen wird die Fehlerbehandlung demonstriert und bewertet.

Die Arbeit wird in Kapitel 7 durch eine Zusammenfassung und einen kurzen Ausblick abgeschlossen.

Kapitel 2

Das parallele Datenbanksystem MIDAS

2.1 Überblick

In diesem Kapitel wird das parallele Datenbanksystem MIDAS (**M**un**I**ch parallel **DA**tabase **S**ystem) vorgestellt.

MIDAS wurde im Rahmen des Sonderforschungsbereiches SFB 342, Teilprojekt B2, entwickelt. MIDAS ist der Prototyp eines relationalen, parallelen Datenbanksystems. Bei der Entwicklung von MIDAS wurde keine komplette Neuentwicklung vollzogen, vielmehr stellt MIDAS eine Weiterentwicklung des sequentiellen, relationalen Datenbanksystems TransBase [TransBaseS 88] dar.

TransBase ist ein sequentielles, relationales Datenbanksystem, das durch die Firma *TransAction Software GmbH* entwickelt und vertrieben wird. Es ist die kommerzielle Version des Datenbanksystems Merkur [ElKiLeSe 87], eines ebenfalls an der Technischen Universität München entworfenen Prototypen.

Die TransBase-Version 3.3 diente als Codebasis für MIDAS. Ausgehend von dieser Codebasis entstand MIDAS schrittweise durch Ersetzen, Hinzufügen und Anpassen von Komponenten [BJLMRZ 96a, BJLMRZ 96b].

MIDAS besteht aus zwei großen Subsystemen, dem Anfragesystem und dem Ausführungssystem. Das Anfragesystem umfaßt SQL-Compiler, Optimierer und Parallelisierer und unterstützt im wesentlichen Inter-Transaktionsparallelität. Das Ausführungssystem ist für die Ausführung von Anfrageplänen zuständig und ermöglicht Intra-Transaktionsparallelität bis hin zur Intra-Operator-Parallelität. Die detailliertere Prozeßstruktur ist eng an das Ausgangssystem TransBase angelehnt.

Bei der Implementierung des Prototyps zeigte sich, daß sehr große Teile von

TransBase fast unverändert übernommen werden konnten. Es konnte durch die Fortentwicklung eines kommerziellen Datenbanksystems sehr schnell und mit verringertem Arbeitsaufwand ein Prototyp entwickelt werden, bei dem außerdem große Subsysteme von vornherein stabil und ausgereift sind.

In [Listl 96] wurden Untersuchungen über eine parallele, verteilte Datenbankcacheverwaltung auf einer Shared-Disk-Architektur vorgestellt. Dieser verteilte Datenbankcache (VDBC) wurde in MIDAS realisiert. Um zu einem parallelen Shared-Disk-Datenbanksystem zu gelangen, mußten an TransBase im Bereich der Cacheverwaltung wesentliche Erweiterungen vorgenommen werden. Außerdem mußte die Möglichkeit geschaffen werden, in einem Workstation-Netz, das im Prinzip eine Shared-Nothing-Architektur ist, durch eine aufgesetzte Softwareschicht eine Shared-Disk-Architektur zu realisieren. Zusätzlich sollten die Datenbanken über mehrere Festplatten verteilt werden.

Für die Intra-Transaktionsparallelität wurden ein Parallelisierer, ein Scheduler und ein an die Parallelverarbeitung angepaßter Optimierer entworfen [Nippl 00]. Im Ausführungssystem mußten die Operatoren der Ausführungspläne um Kommunikationsoperatoren erweitert werden, um Zwischenergebnisse zwischen partiellen Ausführungsplänen auch über Rechnergrenzen hinweg austauschen zu können. Zusätzlich dazu mußten mehrere Partitionierungsverfahren und Mischtechniken entwickelt und in Form von Operatorknoten realisiert werden.

In Abschnitt 2.2 werden zuerst das Datenbanksystem TransBase sowie erste Versuche, daraus ein paralleles System zu gewinnen, beschrieben. Anschließend wird in Abschnitt 2.3 das Datenbanksystem MIDAS vorgestellt. Es werden die Subsysteme, das Anfragesystem (Abschnitt 2.3.1) und das Ausführungssystem (Abschnitt 2.3.2) sowie die Kommunikation im System beschrieben. Zusätzlich erfolgt in Abschnitt 2.4 eine Beschreibung wichtiger Systemkomponenten von MIDAS. Am Ende des Kapitels folgen eine Analyse der Wiederverwendung von Software bei der Entwicklung von MIDAS (Abschnitt 2.5), sowie eine Beschreibung der Portierbarbarkeit von Anwendungen (Abschnitt 2.6) und der zusätzlich entwickelten Werkzeuge (Abschnitt 2.7).

2.2 Von TransBase zu MIDAS

2.2.1 Das Datenbanksystem TransBase

Die Systemarchitektur des relationalen, sequentiellen Datenbanksystems TransBase wurde gemäß den gängigen Datenbankarchitekturprinzipien konzipiert und orientiert sich dabei an der Fünf-Schichten Architektur für relationale Datenbanksysteme [LoiObePaw 91, Stonebraker 94, HärRah 99].

TransBase ist ein relationales Mehrbenutzerdatenbanksystem. Es erlaubt Re-

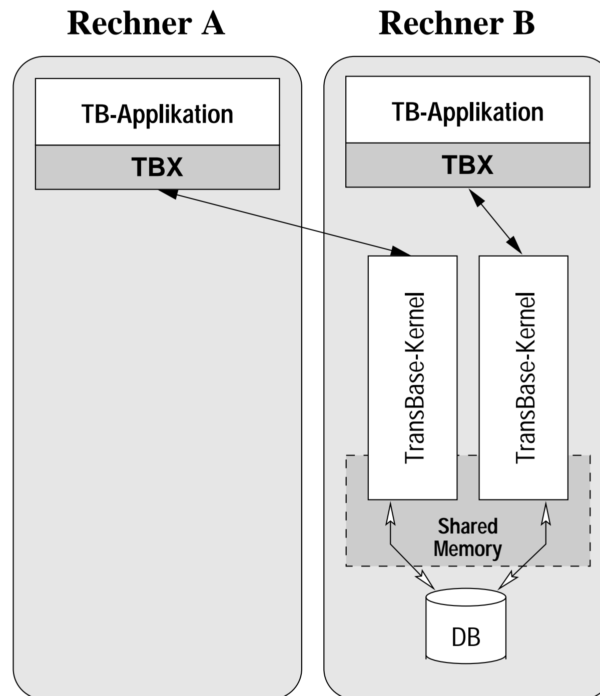

Abbildung 2.1: Prozeßstruktur von TransBase

mote-Zugriffe auf Datenbanken sowie den gleichzeitigen Zugriff auf mehrere Datenbanken. Die allgemeine Prozeßstruktur von TransBase ist in Abbildung 2.1 dargestellt. Die von der Datenbank verwalteten Daten werden auf externen Speichermedien abgelegt. In der verwendeten Version werden Festplatten benutzt, auf denen ein UNIX-Dateisystem installiert ist. Jede Relation des Datenbanksystems wird dabei in einer eigenen Datei abgelegt. Auf dem Rechner, auf dem eine TransBase-Datenbank hochgefahren wird, wird ein Shared-Memory-Speicherbereich im Hauptspeicher eingerichtet. Dieser wird von allen auf dem Rechner befindlichen TransBase-Kernel-Prozessen zur Ablage gemeinsamer Daten verwendet. Er dient in erster Linie als Datenbankcache, in dem gemeinsame Datenseiten gepuffert werden. Der Shared-Memory-Bereich wird aber auch für Daten der Sperr- und Transaktionsverwaltung genutzt.

Für jede Datenbankapplikation, die auf eine TransBase-Datenbank zugreift, wird auf dem Rechner, auf dem sich die Datenbank befindet, ein eigener Prozeß erzeugt. Dieser Prozeß, genannt TransBase-Kernel-Prozeß, ist dieser Applikation exklusiv zugeordnet und steuert den Datenbankzugriff für die Applikation. Der Zugriff auf diesen Kernel-Prozeß findet über die Kommunikationsschnittstelle TBX (TransBase-Exchange) von TransBase statt. TBX wird in Form einer Programmbibliothek zur Applikation gebunden [TransBaseP 89].

Abbildung 2.2 zeigt den Aufbau eines Kernel-Prozesses. Der Kernel ist

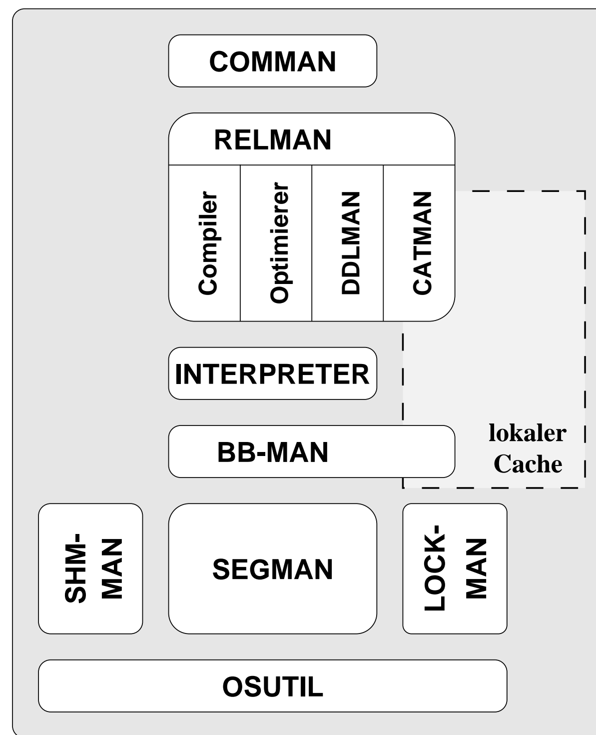

Abbildung 2.2: Aufbau des TransBase-Kernels

in mehrere hierarchisch angeordnete Module aufgeteilt. Das oberste Modul, der Kommunikations-Manager (COMMAN), ist das Gegenstück zur TBX-Schnittstelle auf seiten der Applikation. Er nimmt die Anfragen der Applikation entgegen und liefert die Ergebnisse nach Ausführung der Anfragen zur Applikation zurück. Vom Kommunikations-Manager empfangene Anfragen werden an das nächst tiefere Modul, den Relationen-Manager (RELMAN), übergeben. Dort werden sie, entsprechend dem Typ der Anfrage, an eines seiner Untermodule weitergeleitet.

DML-Anfragen[1] werden vom Compiler-Modul in einen Operatorbaum übersetzt. Der Operatorbaum ist ein Ausdruck einer erweiterten relationalen Algebra und repräsentiert die gestellte Anfrage. Dieser Operatorbaum wird durch das Optimierer-Modul in einen äquivalenten Baum transformiert, der eine effizientere Ausführung verspricht.

DDL-Anfragen[2], wie das Anlegen und Löschen von Relationen, Sichten oder Indexstrukturen, werden direkt vom DDL-Manager im Applikationsserver aus-

[1] Anfragen der DatenManipulations-Sprache

[2] Anfragen der DatenDefinitions-Sprache

geführt.

Das Katalog-Manager-Modul verwaltet und puffert Metadaten, wie den Aufbau von Relationen, Attributen, Indexstrukturen oder Sichten, und stellt diese dem Compiler und Optimierer zur Verfügung.

Die vom Optimierer erzeugten Operatorbäume werden vom Interpreter-Modul ausgewertet. Die dabei berechneten Tupel werden einzeln durch die Knoten des Operatorbaumes bis zu dessen Wurzel gereicht, wo das Ergebnis der Anfrage gesammelt wird. Der Zugriff auf die Daten der Datenbank findet in den Blättern der Operatorbäume statt und wird durch das B-Baum-Manager-Modul (BB-MAN) realisiert. Der B-Baum-Manager stellt über eine tupelorientierte Zugriffsschnittstelle Präfix-B*-Bäume sowie ein Sortier-Modul für Relationen zur Verfügung, die als Präfix-B*-Baum oder sequentiell organisierte Datei vorliegen.

Unter dem B-Baum-Manager liegt das Segment-Manager-Modul (SEGMAN). Dieses verwaltet Segmente als einen in Seiten unterteilten linearen Adreßraum. Relationen werden dabei vom B-Baum-Manager auf Segmente abgebildet, wobei die Tupel der Relationen in den einzelnen Seiten der Segmente plaziert werden. Der Segment-Manager bildet Segmente auf Dateien ab, führt die Cacheverwaltung sowie physische Ein- und Ausgaben durch und nimmt die Recovery vor.

Das unterste Modul (OSUTIL) kapselt die betriebssystemabhängigen Funktionen des Datenbanksystems. Dies erleichtert die Portierung von TransBase auf andere Betriebssysteme, da nur Anpassungen in diesem Modul vollzogen werden müssen.

Parallel zu dieser Modulhierarchie existieren noch das Sperrverwaltungsmodul (LOCKMAN) zur Synchronisation parallel ablaufender Transaktionen sowie das Shared-Memory-Manager-Modul (SHMMAN), das das Anlegen sowie den Zugriff auf gemeinsamen Speicher verwaltet.

Zusätzlich zu diesen Programmodulen besitzt jeder Kernel-Prozeß seinen eigenen, lokalen Speicherbereich, einen prozeßlokalen Cache. Dieser hat eine fest vorgegeben Größe und dient zur Pufferung der Daten des Katalog-Managers sowie als Speicher für Zwischenergebnisse, die bei der Auswertung der Operatorbäume in Sortierknoten entstehen.

2.2.2 Erste Schritte der Parallelisierung

Die ersten Versuche, ein paralleles Datenbanksystem zu implementieren zielten darauf ab, das sequentielle Datenbanksystem TransBase auf eine parallele Hardware zu portieren. Als Plattform hierfür standen Shared-Disk-Multiprozessorrechner ohne gemeinsamen Speicher vom Typ iPSC/860 und iPSC/2 [Intel 89] zur Verfügung. Letzterer besaß 32 Prozessoren, die alle mit lokalem Arbeitsspeicher ausgestattet waren und durch ein schnelles Hypercube-Netzwerk verbunden wurden. Als Betriebssystem existierte MMK (Multiprozes-

sor Multitasking Kernel) [BemLud 90], bei dem Kommunikation zwischen Komponenten über Nachrichtenaustausch stattfindet. MMK ist ein Aufsatz auf das Intel-Betriebssystem NX/2. Die wichtigste Änderung, die an TransBase vorgenommen werden mußte, um Anfragen auf verschiedenen Prozessoren einer Maschine ohne gemeinsamen Speicher auszuführen, war die Implementierung einer verteilten Cacheverwaltung. Die Portierung von TransBase auf die Multiprozessorrechner wurde vollzogen, allerdings zeigte sich diese portierte TransBase-Version als sehr instabil und leistungsschwach, was im wesentlichen auf Hard- und Software-Fehler der iPSC/2 zurückzuführen war.

Ein weiterer Ansatz zur Implementierung eines parallelen Datenbanksystems war die Verwendung eines speichergekoppelten Parallelrechners, der ALLIANT FX 2800. Mit diesem Modell wurde zum einen ein paralleles, externes Sortierverfahren für Mehrprozessorsysteme mit gemeinsamen Speicher [Menzel 91] untersucht, und zum anderen eine Portierung von TransBase auf die ALLIANT FX 2800 angegangen. Allerdings stellte sich die Portierung von TransBase auf die ALLIANT FX 2800 wegen unzureichender Software- und Hardwarewartung (die Firma ALLIANT meldete Konkurs an) als zu kompliziert und zeitintensiv heraus, so daß dieser Ansatz nicht weiter verfolgt wurde.

Nach den schlechten Erfahrungen mit Parallelrechnern auf Basis von Spezial-Hardware, fand ein Umstieg des Projektes auf ein Netz gewöhnlicher Workstation-Rechner statt. MMK wurde ebenfalls auf diese Umgebung portiert und hieß dort MMK/X. TransBase wurde auf MMK/X portiert. Dabei wurden isolierte Implementierungen für parallel arbeitende Komponenten eines Datenbanksystems erstellt [LisPaw 92]. Aber auch in dieser Systemkonfiguration gab es Probleme mit dem proprietären MMK/X, so daß das Projekt schließlich komplett auf spezielle, parallele Hardware und proprietäre Systemumgebungen verzichtete. Die Untersuchungen verschiedener Parallelitätsformen beschränkten sich seitdem auf die Parallelität in den Software-Komponenten.

Als Umgebung für die Entwicklung eines Prototypen des parallelen Datenbanksystems MIDAS wurden ab diesem Zeitpunkt Netze gewöhnlicher Workstation-Rechner und Workstation-Multiprozessorrechner auf dem Betriebssystem UNIX verwendet. Ein wichtiges Ziel bei der Entwicklung war es unter anderem, einen möglichst systemunabhängigen Prototypen zu entwickeln, was erfolgreich gelungen ist. So kann MIDAS beispielsweise auch auf PCs mit dem Betriebssystem Linux eingesetzt werden. Portierungen auf andere UNIX-Systeme als die der verwendeten Sun-Rechner sollten ohne Probleme möglich sein.

2.3 Das relationale, parallele Datenbanksystem MIDAS

MIDAS ist ein Prototyp eines relationalen, parallelen Shared-Disk-Datenbanksystems. Zielarchitektur für MIDAS ist ein Netz von Workstations. MIDAS besitzt eine Client-Server-Architektur. Die Datenbankanwendungen (Applikationen) sind die Clients. Sie sind normale, sequentielle Programme, die über eine SQL-Schnittstelle auf das Datenbanksystem, den MIDAS-Server, zugreifen und diesen mit Transaktionen beauftragen. Diese Schnittstelle ist die in Abschnitt 2.2.1 erwähnte TBX-Schnittstelle. Sie wurde unverändert von TransBase übernommen, so daß alle TransBase-Applikationen ohne Neuübersetzung, nur durch Binden mit der neuen MIDAS-TBX-Bibliothek, auf MIDAS portiert werden können. Eine MIDAS-Applikation kann auf jedem Rechner, der Zugriff auf den MIDAS-Server besitzt, ausgeführt werden. Sie muß sich nicht auf einem der Rechner des MIDAS-Servers befinden.

Als MIDAS-Server wird eine Menge von Prozessen, die auf verschiedenen Rechnern laufen, bezeichnet. Diese Rechner können sowohl normale Einprozessorsysteme als auch Shared-Memory-Mehrprozessorsysteme sein. Der MIDAS-Server bietet zum effizienten Zugriff auf die Daten der Datenbank eine SQL-Schnittstelle, die TBX-Schnittstelle, an. Die Datenbank selbst ist auf Sekundärspeichermedien (Festplatten) untergebracht, auf die der Zugriff durch ein vom Betriebssystem bereitgestelltes Dateisystem erfolgt. Eine schematische Darstellung der Architektur von MIDAS findet sich in Abbildung 2.3.

Die Prozesse des MIDAS-Servers sind in zwei Klassen eingeteilt, die des Anfragesystems und die des Ausführungssystems. Die Prozesse des Anfragesystems, die Applikationsserver, sind jeweils einer Applikation exklusiv zugeordnet. Sie sind zuständig für den Empfang der SQL-Anweisungen sowie deren Übersetzung, Optimierung, Parallelisierung und Scheduling. Als Ergebnis produzieren die Applikationsserver Ausführungspläne (QEPs, Query Execution Plan), die durch die Prozesse der zweiten Schicht, des Ausführungssystems, ausgewertet werden. Die Prozesse des Ausführungssystems sind zuständig für die Interpretation der Ausführungspläne sowie die verteilte Cache- und Sperrverwaltung.

Die gewählte Prozeßstruktur ermöglicht es, im MIDAS-Ausführungsmodell zwei Parallelitätsarten zu unterstützen.

- **Inter-Transaktionsparallelität:**
  Inter-Transaktionsparallelität wird durch die Verteilung ganzer Transaktionen auf mehrere Applikationsserver auf verschiedenen Rechnern erreicht. Jeder Applikationsserver arbeitet zusammen mit den ihm zugeordneten Prozessen des Ausführungssystems echt parallel zu anderen Applikationsservern auf anderen Rechnern. Die Parallelität ist damit nicht mehr auf

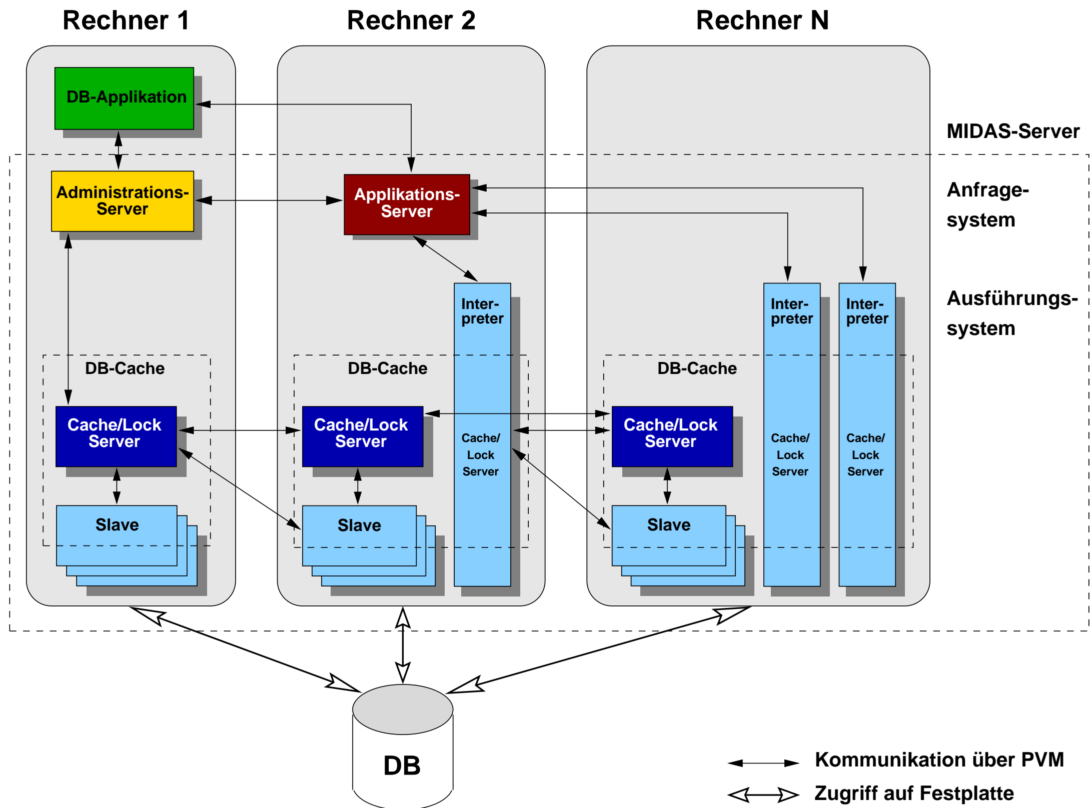

Abbildung 2.3: MIDAS–Architektur

eine Pseudoparallelausführung auf einer Einprozessormaschine oder auf eine nur durch das Betriebssystem gesteuerte Parallelausführung auf einer Multiprozessormaschine beschränkt.

- **Intra-Transaktionsparallelität:**

  Intra-Transaktionsparallelität ist in MIDAS identisch mit Intra-Anfrageparalleliät, da MIDAS keinen Mechanismus (beispielsweise parallele Konstrukte in der Anfragesprache) bietet, der Parallelität zwischen den Anfragen einer Transaktion erlaubt.

  Intra-Transaktionsparallelität entsteht dadurch, daß von einem Prozeß des Anfragesystems der aus einer Anfrage erzeugte Ausführungsplan in eine Menge von Teilplänen zerlegt wird. Diese Teilpläne werden anschließend parallel zueinander durch mehrere Prozesse des Ausführungssystems, die Interpreter, ausgewertet. Die Interpreter arbeiten gleichzeitig auf verschiedenen Teilplänen und reichen Ergebnisse einander weiter, wodurch Pipeline-Parallelität entsteht. Zusätzlich können einzelne Teilpläne auch repliziert auf mehreren Prozessen bearbeitet werden; dadurch entsteht Datenpar-

allelität. Auf diese Weise werden teure Operatoren parallel auf mehreren Prozessoren ausgeführt (Intra-Operator-Parallelität). Die Produzenten und Konsumenten arbeiten in dem so entstehenden Netz von einander zuarbeitenden Prozessen zur Evaluierung einer Anfrage asynchron.

Die MIDAS-Architektur entspricht dabei der Fünf-Schichten Architektur für relationale Datenbanksysteme [Stonebraker 94, HärRah 99]. Die Applikationsserver des Anfragesystems realisieren die Schicht des deklarativen Datenbankzugriffs. Das Ausführungssystem umfaßt die Schichten des navigierenden Datenbankzugriffs, des physischen Satzzugriffs und des Systempuffers. Die Externspeicherverwaltung wird dem Dateisystem des zugrundeliegenden Betriebssystems überlassen.

Neben der Unterscheidung der Prozesse des MIDAS-Servers in Anfrage- und Ausführungssystem, lassen sich die Prozesse auch durch ihre Eigenschaft als statische oder dynamische Systemkomponenten unterscheiden. Die statischen Komponenten werden einmal beim Hochfahren des MIDAS-Servers gestartet und ihre Anzahl bleibt während der Restlaufzeit des Servers konstant. Im Gegensatz dazu ist die Anzahl der dynamischen Komponenten nicht konstant. Ihre Anzahl kann bei Bedarf erhöht werden, um sich veränderten Lastbedingungen anpassen zu können. Die dynamischen Komponenten werden von den statischen Komponenten in sogenannten Pools verwaltet. Werden während der Laufzeit neue, dynamische Komponenten benötigt, so werden diese aus dem Pool heraus vergeben. Sind in den Pools keine freien Komponenten mehr vorhanden, so werden neue gestartet. Dynamische Komponenten, die ihre Aufgaben erfüllt haben, werden wieder in die Pools aufgenommen, anstatt sie zu beenden. Dadurch entstehen bei Neuanforderung dynamischer Komponenten deutlich geringere Zeitverzögerungen, da selten teure Prozeßerzeugungen notwendig sind.

2.3.1 Das Anfragesystem

Aufgabe des Anfragesystems ist es, aus SQL-Anweisungen, die es von den Anwendungen erhält, Ausführungspläne zu generieren [ElKiLeSe 87, Mitschang 95]. Diese Ausführungspläne werden auf die Komponenten des Ausführungssystems verteilt und dort ausgewertet. Die Ergebnisse dieser Auswertungen sendet das Anfragesystem an die Anwendungen zurück.

Die Applikationsserver bilden das MIDAS-Anfragesystem. Jeder Applikationsserver steht jeweils einer Anwendung exklusiv zur Verfügung. Es wird demnach jeder neuen Anwendung ein anderer Applikationsserver zugeordnet. Die Zahl der im System benötigten Applikationsserver ist daher nicht konstant, und die Applikationsserver fallen somit unter die dynamischen Komponenten des Systems. Sie werden, wie oben erwähnt, in einem sogenannten Pool vom MIDAS-Server, der in Abschnitt 2.3.3 näher beschrieben ist, verwaltet.

Jeder Applikationsserver übernimmt für seine Anwendung die oben beschriebenen Aufgaben des Empfangens von SQL-Anweisungen, deren Übersetzung, Optimierung, Parallelisierung sowie schließlich die Koordination der Auswertung der Anfrage durch die Komponenten des Ausführungssystems.

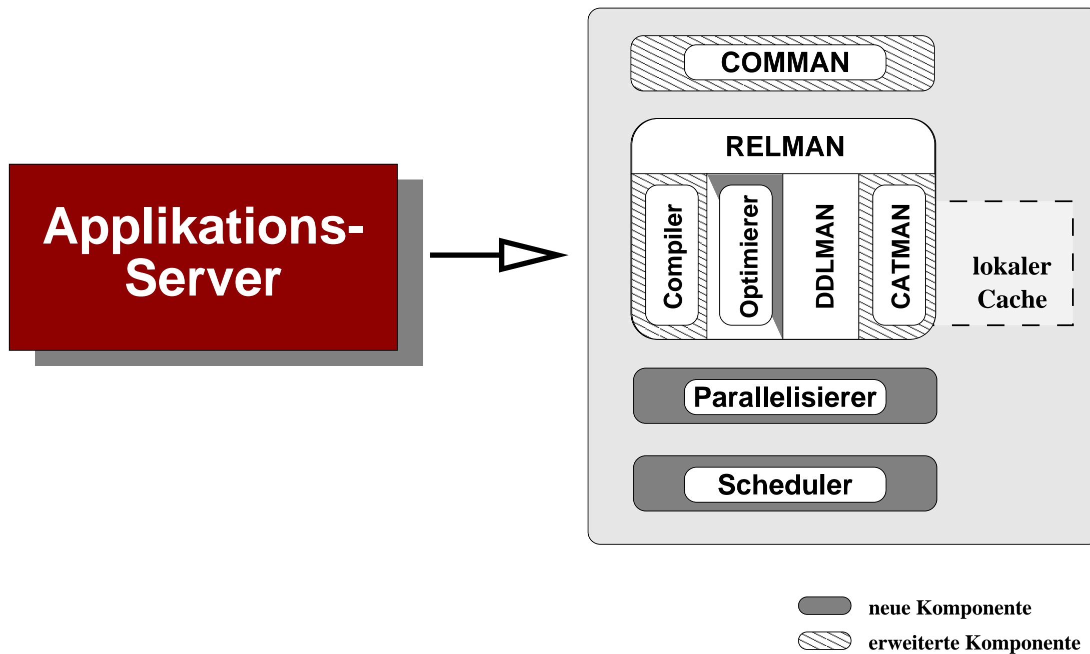

Abbildung 2.4: Komponenten des Anfragesystems

Um diese Aufgaben erfüllen zu können, ist der Applikationsserver in die acht Module Kommunikations-Manager, Relationen-Manager, Compiler, Optimierer, DDL-Manager, Katalog-Manager, Parallelisierer und Scheduler aufgeteilt. Diese Unterteilung des Applikationsservers ist in Abbildung 2.4 dargestellt. Sie ist sehr ähnlich der in Abschnitt 2.2.1 vorgestellten Strukturierung der oberen Schichten des TransBase-Kernels, was sich aus der Entwicklung von MIDAS aus TransBase ergab. Die Abbildung 2.4 zeigt zusätzlich die unverändert von TransBase übernommenen, die erweiterten und die komplett neuen Komponenten. Nach der Beschreibung der allgemeinen Funktionsweise der Module wird detaillierter auf die veränderten und neuen Komponenten eingegangen.

Abbildung 2.5 stellt die Verarbeitung einer SQL-DML-Anfrage durch den Applikationsserver dar. Nach einer Vorverarbeitung durch Parser und Scanner erzeugt der Compiler aus einer DML-Anfrage unter Verwendung der durch den Katalog-Manager verwalteten Systemtabellen einen sequentiellen Operatorbaum. Dieser und alle im folgenden entstehenden Operatorbäume sind die interne Repräsenta-

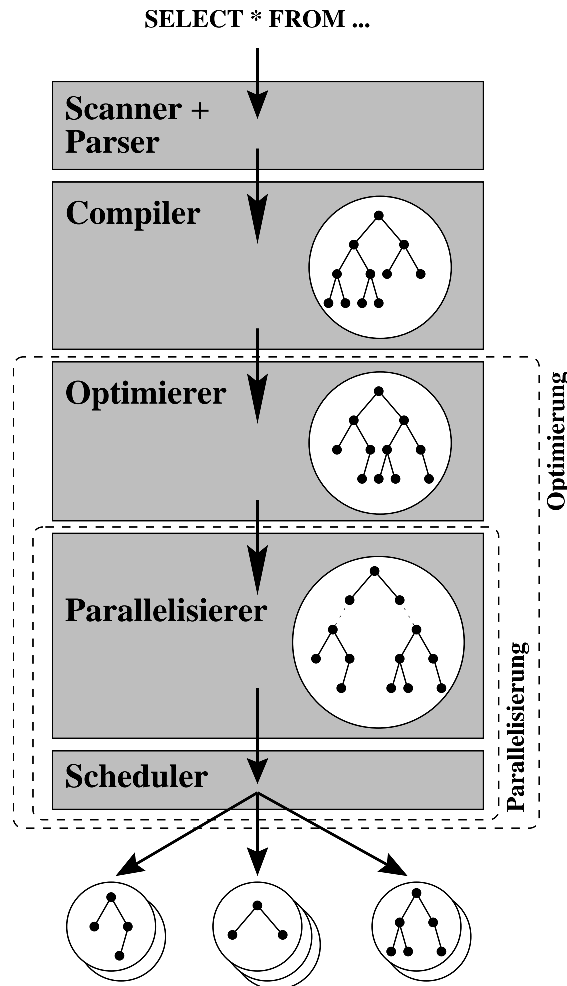

Abbildung 2.5: Funktionsweise des Applikationsservers

tionsform der SQL-Anfrage. Die Knoten dieser Operatorbäume werden als Operatoren bezeichnet und repräsentieren entweder mengenwertige Operationen der relationalen Algebra oder die Verknüpfung skalarer Werte. Beispiele für mengenwertige Operatoren sind beispielsweise Joins (Sort-Merge-Join, Nested-Loop-Join, Hash-Join), Sortierungen, Projektion oder Selektion. Skalare Operatoren sind unter anderem arithmetische oder boolsche Verknüpfungen.

Der sequentielle Operatorbaum wird anschließend vom Parallelisierer in einen parallelen Ausführungsplan (PQEP) umgewandelt, der aus mehreren Teilplänen besteht, die parallel ausgeführt werden können.

Die generierten Teilpläne werden vom Scheduler auf verschiedene Interpreter des Ausführungssystems verteilt und von diesen ausgewertet.

2.3.2 Das Ausführungssystem

Das Ausführungssystem besteht aus einem Cache/Lock-Server und jeweils einigen Slaves und Interpretern pro Rechner.

Die Interpreter stellen die oberste Schicht des Ausführungssystems dar. Sie werten die ihnen vom Applikationsserver zugesandten Teilpläne parallel zu anderen Interpretern aus. Der Interpreter, der den Wurzelteilbaum des parallelen Ausführungsplans auswertet, kombiniert die Ergebnisse aller an der Auswertung beteiligten Interpreter, und sendet das Gesamtergebnis an den Applikationsserver zurück.

Abbildung 2.6 zeigt die Zuordnung der Module zu den einzelnen Prozessen des Ausführungssystems. Der Interpreter beinhaltet dabei das Interpreter- sowie das B-Baum-Manager-Modul aus TransBase.

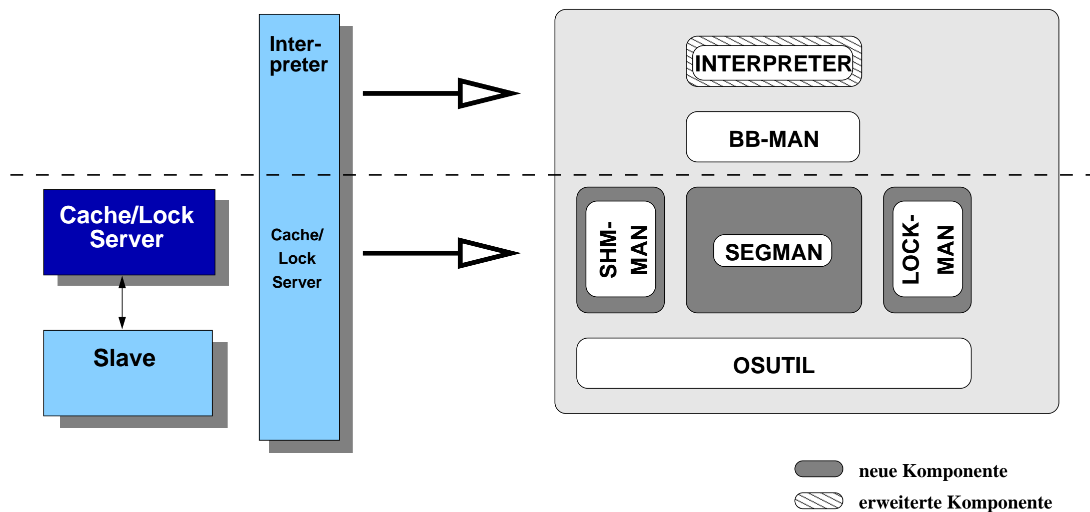

Abbildung 2.6: Komponenten des Ausführungssystems

Bei der Auswertung der Teilpläne generieren die Interpreter Seitenanforderungen an die Pufferverwaltung. Alle Module der Pufferverwaltung befinden sich auch im Interpreterprozeß (siehe auch Abschnitt 3.2.1).

Zusätzlich zu den Interpretern existiert auf jedem Rechner ein Cache/Lock-Server mit seinen Slaves. Die Cache/Lock-Server realisieren die Pufferverwaltung, den Datenbankcache mit zugehöriger Sperrverwaltung und Kohärenzkontrolle. Weiter steuern sie die Zugriffe auf den Hintergrundspeicher (Festplatten). Die Pufferverwaltung stellt dabei den Interpreter-Modulen einen effizienten und ortstransparenten Zugriff auf die Datenseiten zur Verfügung.

Auch die Interpreter-Prozesse beinhalten die Module der Pufferverwaltung, so daß sie alle lokalen Aufgaben der Pufferverwaltung selbst übernehmen können. Die Cache/Lock-Server dagegen dienen der Durchführung aller Aufgaben der Pufferverwaltung, die auf entfernten Rechnern entstehen. Aufträge an die Cache/Lock-Server, die nicht sofort beantwortet werden können, werden an einen seiner Slaves weitergeleitet, um ein Blockieren des Cache/Lock-Servers zu verhindern.

Als statische Komponente verwaltet jeder Cache/Lock-Server einen Pool von Slaves und Interpretern. Die Slaves und Interpreter stellen die dynamischen Komponenten des Ausführungssystems dar.

Die eigentliche Datenbank befindet sich auf dem Hintergrundspeicher (Festplatte), auf den mittels der vom Betriebssystem zur Verfügung gestellten Systemroutinen über ein Dateisystem zugegriffen wird. Da MIDAS als Shared-Disk-Architektur konzipiert wurde, ist der Hintergrundspeicher von allen Rechnern des MIDAS-Servers erreichbar. Dies ist über ein Network File System (NFS) realisiert.

2.3.3 Der MIDAS-Administrations-Server

Der MIDAS-Administrations-Server ist eine statische Komponente des MIDAS-Servers. Er besitzt ausschließlich administrative Aufgaben. Dazu gehören das Anlegen, Löschen, Hoch- und Herunterfahren von Datenbanken. Beim Hochfahren einer Datenbank startet er die statischen Komponenten des MIDAS-Servers, die Cache/Lock-Server, auf den der Datenbank zugeordneten Rechnern des Systems. Beim Herunterfahren einer Datenbank werden die Cache/Lock-Server sowie alle Prozesse des Anfragesystems durch den MIDAS-Administrations-Server wieder beendet.

Zusätzlich verwaltet der MIDAS-Administrations-Server die dynamischen Komponenten des Anfragesystems, die Applikationsserver. Bei jedem Aufbau einer Verbindung zwischen einer MIDAS-Applikation und dem MIDAS-Server wird zuerst der MIDAS-Administrations-Server angesprochen. Dieser verwaltet einen Pool von Applikationsservern, die dem Anfragesystem zur Verfügung stehen. Aus diesem Pool wird ein freier Applikationsserver gewählt. Dieser wird der sich anmeldenden Applikation exklusiv zugeordnet. Sollte kein freier Applikationsserver verfügbar sein, so wird vom MIDAS-Administrations-Server ein neuer gestartet und dem Pool hinzugefügt. Jede weitere Kommunikation der Applikation mit dem MIDAS-Server, also insbesondere das Anmelden bei der Datenbank und das Stellen von SQL-Anfragen, findet über diesen Applikationsserver statt.

2.3.4 Kommunikation im MIDAS-Server

Im MIDAS-Server wird zur Kommunikation zwischen den Prozessen die Message-Passing-Bibliothek PVM (Parallel Virtual Machine, [GBDJMS 94a]) verwendet, die sich hierfür als geeignet erwiesen hat [LisSchFri 94, BozFleZim 97].

Zu den Vorteilen von PVM zählen die Unterstützung von Shared-Memory-Bereichen in UNIX sowie die Möglichkeit, in MIDAS die Prozeßstruktur (UNIX Prozesse) von TransBase weitgehend übernehmen zu können. Des weiteren realisiert PVM effiziente Kommunikation, ist mit geringem Aufwand in das System integrierbar und weist eine hohe Portabilität auf. PVM gestattet zusätzlich das dynamische Starten von Prozessen zur Laufzeit, wodurch die Skalierbarkeit des Systems und die verwendeten Prozeß-Pools unterstützt werden. Abschnitt 3.3 beschreibt die genaue Verwendung von PVM in MIDAS sowie verschiedene, auf PVM bezogene Leistungsmessungen.

2.4 Die Komponenten von MIDAS

2.4.1 Katalogmanager

Der Katalogmanager verwaltet die Katalogdaten des Systems. Die Katalogdaten enthalten Informationen über die im System vorhandenen Relationen, Attribute, Sichten, Benutzer, Zugriffsrechte und Indexstrukturen. Diese sind in Form von Systemrelationen im Datenbanksystem abgelegt und werden im Katalogmanager lokal gepuffert [ElKiLeSe 87, TransBaseS 88]. Der Katalogmanager wurde unverändert von TransBase übernommen. Anschließend wurde die Menge der Systemrelationen um Relationen erweitert, die Statistiken über die Daten in anderen Relationen und Metadaten zu benutzerdefinierten Funktionen enthalten [Haas 98, Perathoner 98a, Jaedicke 99].

2.4.2 Compiler

DML-Anfragen werden vom Compiler-Modul in zwei Schritten in einen Operatorbaum übersetzt. Im ersten Schritt wird die Anfrage auf syntaktische und semantische Korrektheit getestet und direkt in einen Operatorbaum übersetzt, ohne dabei Optimierungen vorzunehmen. Im zweiten Schritt werden zusätzliche semantische Kontrollen auf dem Operatorbaum durchgeführt und die Schemata der im Operatorbaum erzeugten Zwischenergebnisse berechnet. DDL-Anfragen, wie das Anlegen und Löschen von Relationen, Sichten oder Indexstrukturen, werden direkt an den DDL-Manager weitergegeben und dort ausgeführt.

Der Anfrage-Compiler wurde von TransBase übernommen. An ihm wurden

Erweiterungen vorgenommen, um benutzerdefinierte Anfragen übersetzen zu können [Haas 98, Jaedicke 99].

2.4.3 Optimierer

Der Optimierer erhält vom Compiler-Modul einen Operatorbaum, der einem Ausdruck einer erweiterten, relationalen Algebra entspricht und die gestellte Anfrage repräsentiert. Dieser Operatorbaum wird durch das Optimierer-Modul in einen äquivalenten Baum transformiert, der eine effizientere Ausführung verspricht.

Es besteht die Alternative, den unverändert von TransBase übernommenen Optimierer oder den neu entwickelten Optimierer Model_M (Model_MIDAS) einzusetzen.

Wird der TransBase-Optimierer verwendet, so wird dieselbe Optimierung wie in TransBase durchgeführt. Insbesondere werden dabei keine Techniken angewandt, die eine spätere Parallelisierung unterstützen.

Der neue Optimierer Model_M führt nur noch wenige Schritte des alten TransBase-Optimierers durch. Diese überführen den Operatorbaum beispielsweise in eine normalisierte Form, In- und Exists-Unteranfragen werden in Joins umgewandelt und Selektionen und Projektionen werden soweit als möglich im Operatorbaum nach unten geschoben.

Im Unterschied zum TransBase-Optimierer trifft Model_M auch Optimierungsentscheidungen im Hinblick auf eine nachfolgende Parallelisierung. So achtet Model_M z. B. darauf, möglichst keine *links-* oder *rechts-tiefen Bäume* zu erzeugen, sondern *stark verzweigte, buschige*[3] *Bäume*. Letztere haben die Eigenschaft, daß sie ausgeglichen verzweigt sind und sich daher gut parallelisieren lassen. Gerade diese Eigenschaft ist bei den vom TransBase-Compiler und -Optimierer erzeugten Bäumen nicht gegeben, da nur links-tiefe Bäume erzeugt werden.

Für die Implementierung von Model_M wurde das Werkzeug Cascades [Graefe 95] zur Entwicklung regel- und kostenbasierter Optimierer verwendet [Hilbig 98, Krü-Bar 98, Nippl 00].

2.4.4 Parallelisierer

Der Parallelisierer TOPAZ wurde, wie der Optimierer Model_M, über Cascades [Graefe 95] realisiert. Der Parallelisierer hat die Aufgabe, den optimierten sequentiellen Operatorbaum in Teilbäume aufzuspalten, die parallel ausgeführt werden können. Ziel der Parallelisierung ist, die Antwortzeit der Anfrage dadurch zu ver-

[3] Sogenannte "bushy trees".

ringern, daß die Gesamtarbeit in kleinere Teilaufgaben aufgeteilt wird und diese Teilaufgaben dann auf mehrere Prozessoren verteilt werden.

Die Parallelisierung wird dabei in mehrere Phasen unterteilt, um den Suchraum für den optimalen Plan einzugrenzen. Die einzelnen Phasen fokussieren sich dabei auf verschiedene Aspekte der Anfrageausführung. Sie führen unter anderem Inter- und Intra-Operator-Parallelität in Form von Pipeline- und Datenparallelität ein.

Die Parallelisierung findet kostenbasiert statt. Dazu wurde ein Kostenmodell entworfen, das die Ressourcen Rechenzeit, E/A-Kosten, Kommunikationskosten und Speicherverbrauch mit einbezieht.

Als Ausgabe produziert der Parallelisierer einen parallelen, parametrisierten Ausführungsplan. Die Teilbäume in dem parallelen Plan sind durch SEND- und RECEIVE-Operatoren abgegrenzt. Über diese findet bei der späteren Ausführung des Plans die Kommunikation zwischen den Interpretern statt. Die Parametrisierung erlaubt nachträgliche Anpassungen des Plans zur Laufzeit. Zu Beginn der Ausführung können damit noch Größen wie der Parallelitätsgrad des Plans oder die Speicherzuteilung zu Operatoren, abhängig vom momentanen Systemzustand, getroffen werden [Fleischhauer 97, NipMit 98a, Nippl 00].

2.4.5 Scheduler

Aufgabe des Schedulers ist es, für Lastbalancierung im System zu sorgen. Dazu werden von den Cache/Lock-Servern auf jedem Rechner Systemzustandsinformationen zur Verfügung gestellt. Diese werden vom Scheduler verwendet, um die parallelen Ausführungspläne in Parallelitätsgrad und Ressourcenverbrauch anzupassen, um dadurch eine gleichmäßige Lastverteilung auf allen Rechnern zu erzielen. Anschließend werden die einzelnen Teilpläne vom Scheduler auf die Interpreter des Ausführungssystems verteilt und dort zur Ausführung gebracht [Perathoner 98b, Nippl 00].

2.4.6 Interpreter

Das Interpretermodul konnte komplett von TransBase übernommen werden. Seine Aufgabe besteht darin, Ausführungspläne von Anfragen auszuführen. Dazu gehören SELECT- und UPDATE-Anfragen der DML sowie Kataloganfragen von Compiler, Optimierer oder DDL-Modul. Der Interpreter arbeitet nach dem Open-Next-Close-Prinzip, beginnend mit der Wurzel des ihm zugewiesenen Teilbaumes (Top-Down). Die berechneten Ergebnisse werden Tupel für Tupel im Baum nach oben gereicht, bis sie an der Wurzel entweder an andere Interpreter oder an den Applikationsserver weitergeleitet werden.

Das Interpretermodul wurde für MIDAS um einige Operatoren erweitert. Dies

sind der Hash-Join, der die Menge der verfügbaren Joins ergänzt. Bisher waren Nested-Loop- und Sort-Merge-Joins vorhanden. Weiter wurden der SEND- und der RECEIVE-Operator eingeführt, die den Austausch von Zwischenergebnissen zwischen Interpretern und somit Pipeline-Parallelität ermöglichen. Schließlich gibt es noch neue Operatoren, die parallele Relationen-Scans durchführen, und den UDTO-Operator, der benutzerdefinierte Tabellenoperatoren implementiert [ElKiLeSe 87, Fleischhauer 94, Brandmayer 97, Heupel 98, Jaedicke 99, Nippl 00].

2.4.7 Pufferverwaltung

2.4.7.1 Ortstransparente Cacheverwaltung

MIDAS verfügt als Shared-Disk-Datenbanksystem über einen eigenen Pufferbereich auf jedem Rechner, der als rechnerlokaler Cache verwendet wird. Dieser Pufferbereich ist als Shared-Memory-Region implementiert und wird von den lokalen Cache/Lock-Servern angelegt. Er beinhaltet die Datenstrukturen und Seitenrahmen der Pufferverwaltung. Alle Komponenten des Ausführungssystems auf diesem Rechner haben darauf Zugriff.

Die Cache/Lock-Server dienen der Durchführung aller Aufgaben der Pufferverwaltung, die auf entfernten Rechnern entstehen. Um Blockierungen zu vermeiden, werden alle Aufträge an die Cache/Lock-Server, die nicht sofort beantwortet werden können, an einen lokalen Slave weitergeleitet. Alle lokalen Anfragen an die Pufferverwaltung werden von den Interpreter-Prozessen selbst beantwortet. Die Interpreter-Prozesse besitzen dazu ebenfalls die Module der Pufferverwaltung, so daß sie die lokalen Aufgaben der Pufferverwaltung übernehmen können. Dadurch wird lokale Kommunikation eingespart (siehe Abschnitt 3.2.1).

Jede Datenseite wird zur Bearbeitung von der Festplatte in den lokalen Cache des Rechners geladen (vertikale Replikation). Dabei kann aufgrund der Parallelverarbeitung im System dieselbe Seite zur gleichen Zeit bei mehreren Rechnern in verschiedenen, zum Teil veralteten Versionen geladen sein (horizontale Replikation). Es muß jedoch sichergestellt werden, daß jede Transaktion nur auf gültige, d. h. die jüngsten, transaktionskonsistenten Seitenversionen zugreift. Die Verwaltung und Erkennung gültiger oder veralteter Seiten wird von der Kohärenzkontrolle durchgeführt.

Im sequentiellen Datenbanksystem TransBase mußte die Pufferverwaltung nur Datenbankseiten auf einem Rechner in einem lokalen Cache bereitstellen. Auf Aspekte der Verteilung brauchte keinerlei Rücksicht genommen zu werden. Somit mußte diese Komponente für MIDAS fast komplett neu geschrieben werden.

In der Dissertation von Andreas Listl [Listl 96] wurde zur Realisierung der Pufferverwaltung eine Architektur namens VDBC (Virtueller DatenBank-Cache) für

einen ortstransparenten Datenbankcache vorgeschlagen. Der VDBC stellt einen virtuellen Cache dar, der die lokalen Cachebereiche aller Rechner umfaßt und den Interpretermodulen auf jedem Rechner erlaubt, auf die Seiten im Cache zuzugreifen, ohne zu wissen, wo sich diese Seiten physisch befinden (siehe Abbildung 2.7). Die angefragten Seiten werden gegebenenfalls durch die Pufferverwaltung auf die Rechner transferiert, auf denen sie benötigt werden.

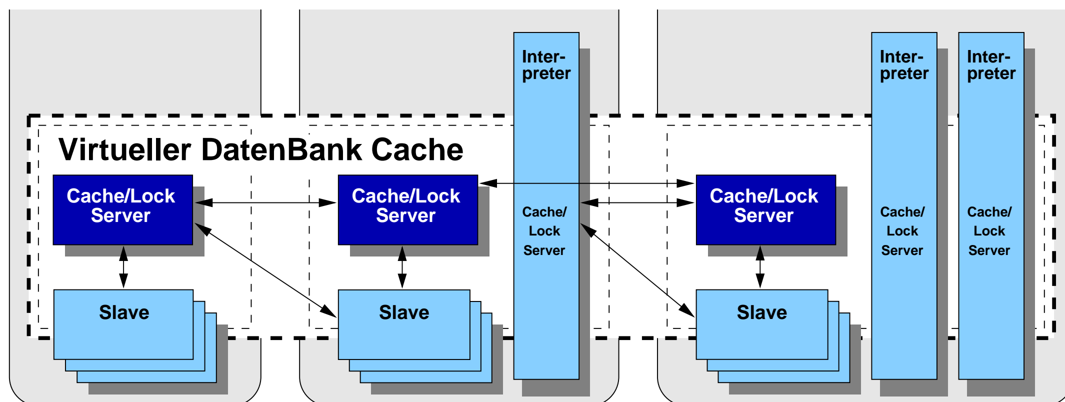

Abbildung 2.7: Virtueller DatenBank-Cache – VDBC

Ein wesentlicher Punkt der in [Listl 96] vorgeschlagenen Architektur ist die Möglichkeit, die Cachekohärenz auf einer feineren Granularität als der Seite zu gewährleisten. Jede Datenbankseite wird in eine Anzahl von Subseiten fester Länge unterteilt. Die Kohärenzkontrolle erfolgt dann auf der Ebene der Subseiten [Listl 94a, Listl 94b, BozLis 95, Listl 96, BozLis 97]. Daraus ergeben sich folgende Vorteile:

- Das für die Protokolle zur Kohärenz notwendige Nachrichtenvolumen wird reduziert, da oft Subseiten statt Seiten ausgetauscht werden.
- Die Möglichkeit, ein feineres, subseitenbasiertes Sperrverfahren einzusetzen, das sich ohne zusätzlichen Aufwand mit der Kohärenzkontrolle kombinieren läßt.

Die in [Listl 96] beschriebenen Algorithmen führen den Begriff des Besitzers von Seiten und Subseiten ein. Der Besitzer ist sowohl für die horizontale Propagierung der gültigen Objektversion zwischen den beteiligten Rechnerknoten als auch für die vertikale Propagierung zu den Festplatten verantwortlich. Das Besitzrecht für die Objekte der physischen Datenbasis wird unter den beteiligten Knoten dynamisch verteilt und kann an Hand von Verdrängungen migrieren.

Für die Behandlung von Pufferinvalidierungen wurde in [Listl 96] ein Multicast-Invalidierungsverfahren vorgeschlagen. Bei diesem Verfahren werden im Rahmen des Commits einer Änderungstransaktion Invalidierungsnachrichten an alle Knoten geschickt, die eine Kopie der geänderten Seiten besitzen. Die Empfänger invalidieren ihre Seitenkopien und benachrichtigen den Rechner, der das Commit koordiniert, der dann die Freigabe der Sperren veranlassen kann. Diese Multicast-Invalidierung verursacht eine sehr hohe Anzahl an Nachrichten, die mit steigender Anzahl von Rechnern und damit Seitenkopien wächst.

Die entstehenden Kommunikationskosten während der Commit-Phase können deutlich verringert werden, indem die Invalidierung in Kombination mit der Sperrvergabe erfolgt. Diese Methode wird als On-Request-Invalidierungsverfahren bezeichnet [Rahm 91, Rahm 94]. Messungen mit dem Simulationsprogramm DBSIM [Bozas 98] haben gezeigt, daß damit eine deutliche Durchsatzsteigerung erzielt werden kann. Deshalb wurde in MIDAS ein On-Request-Invalidierungsverfahren implementiert. Die gleichen Simulationen haben weiterhin gezeigt, daß eine dynamische Verteilung des Besitzrechtes nur bei bestimmten Transaktionslasten einen Gewinn gegenüber einer statischen Verteilung garantiert. Da die Implementierung für die statische Verteilung wesentlich einfacher ist, wurde in MIDAS eine statische Verteilung des Besitzrechtes gewählt.

Die Besitzrechtsverteilung entspricht gleichzeitig der statischen Verteilung der Sperrautorität. Dies bedeutet, daß jeder Knoten gleichzeitig auch Besitzer einer Seite ist, für die er die Sperrautorität besitzt. Dieses Verfahren ermöglicht das Zusammenlegen von Nachrichten der Kohärenzkontrolle und der Sperrverwaltung, wodurch das Nachrichtenvolumen und somit die Netzlast weiter reduziert werden kann [Bozas 98].

2.4.7.2 Synchronisation und Sperrverwaltung

Der MIDAS Lock-Manager realisiert ein hierarchisches RX-Sperrprotokoll. Er wurde komplett neu entwickelt. Die Sperrautorität wird statisch vergeben. Die Verteilung der Zuständigkeit erfolgt mit der gleichen Hashfunktion, die bei der Verteilung des Besitzrechts einer Seite verwendet wird, um, wie im vorigen Abschnitt beschrieben, die Anzahl zu versendender Nachrichten zu reduzieren. Es werden Seiten und Subseiten als Sperrgranulate angeboten. Die Entscheidung, welches Sperrgranulat verwendet wird, wird momentan intern vom Lock-Manager getroffen. An Hand eines Schalters ist es möglich, für jede Transaktion zu bestimmen, auf welcher Ebene sie Sperren anfordert. Deeskalierungen und Eskalierungen von Sperren sind nicht implementiert [Mariucci 97, Bozas 98].

2.4.8 Ein- und Ausgabe

TransBase verwendet zur physischen Speicherung der Daten pro Datenbank jeweils nur eine Festplatte. Dabei bildet die Pufferverwaltung jedes Segment, das einer Relation entspricht, auf eine Datei im Dateisystem ab[4].

Bei einem parallelen Datenbanksystem wie MIDAS wird die Verwendung einer einzigen Festplatte bei den ohnehin schon sehr teuren Plattenoperationen sehr schnell zum Engpaß im System. Deshalb erwies es sich als notwendig, an dieser Stelle eine Unterstützung für die Verwendung mehrerer Festplatten pro Datenbank bereitzustellen.

Als weiterer Nachteil erwies es sich, daß es bei der Verwendung einer einzigen Festplatte schwer ist, eine symmetrische Architektur bezüglich der Plattenzugriffe zu schaffen. MIDAS, das als Shared-Disk-System konzipiert ist, geht somit davon aus, daß alle beteiligten Rechnerknoten Zugriff auf alle Platten haben. MIDAS nutzt das vom Betriebssystem zur Verfügung gestellte NFS, um diesen Zugriff zu gewährleisten[5]. Allerdings entsteht bei der Verwendung einer einzigen Festplatte sofort eine Asymmetrie in der Architektur, da der Rechnerknoten, an dem die Festplatte lokal angeschlossen ist, immer deutliche Zugriffsgeschwindigkeitsvorteile gegenüber den anderen Rechnern besitzt.

Beide Nachteile wurden durch eine Verteilung der Datenbanksegmentdateien auf mehrere Festplatten aufgehoben [Pries 97]. Bei der Konfiguration der Datenbank können mehrere Festplatten (z. B. eine pro teilnehmendem Rechnerknoten) angegeben werden, auf die die Seiten eines Segments momentan nach einer einfachen Round-Robin-Strategie verteilt werden (siehe auch Abschnitt 3.5.1). Damit können zum einen beliebig viele Festplatten die Datenbank aufnehmen, zum anderen haben alle Rechner, bei symmetrischer Verteilung der Daten, gleiche Zugriffscharakteristika auf die Daten.

Durch diese Erweiterung der E/A-Schicht konnten, wie in Abschnitt 3.5.1.2 gezeigt wird, je nach Transaktionstyp zum Teil deutliche Leistungsverbesserungen erzielt werden.

2.4.9 Kommunikationssegmente

Durch die Verteilung der Teilpläne eines parallelen Ausführungsplans auf mehrere Interpreter wurde es notwendig, einen Kommunikationsmechanismus zu imple-

---

[4] Wie diese Abbildung in MIDAS in Kombination mit Seiten und Subseiten realisiert wurde wird in Abschnitt 3.1 beschrieben.

[5] Durch die Verwendung von NFS entstehen weitere Nachteile im System. Diese sind in den Abschnitten 3.5.2 und 3.5.3 beschrieben.

mentieren, der den Interpretern den Austausch von Zwischenergebnissen erlaubt.

Dazu wurde erstmals in [Usner 94, LiPaReBoLe 95] vorgeschlagen, diese Kommunikation über Segmente der Pufferverwaltung zu realisieren.

Nach Analyse des derzeit vorhandenen Systems wurde in Zusammenarbeit mit Giannis Bozas über die Implementierung der Kommunikation zwischen Interpretern entschieden. Als Alternative zur Kommunikation über Segmente stand die Realisierung von direkten Kommunikationsverbindungen über PVM zwischen den Interpretern zur Diskussion. Es wurde entschieden, die Kommunikation über spezielle Segmente der Pufferverwaltung stattfinden zu lassen.

Diese Kommunikationssegmente stellen spezielle, temporäre Relationen der Pufferverwaltung dar und können von den Interpretern wie andere Segmente behandelt werden. Der Zugriff auf Kommunikationssegmente kann identisch zum Zugriff auf andere Segmente der Pufferverwaltung erfolgen. Dadurch ergab sich der Vorteil, daß keine neue Schnittstelle im Interpretermodul für Kommunikationsverbindungen zu schaffen war. Weiter kann sich dieser Kommunikationsmechanismus, wenn er in der Pufferverwaltung realisiert wird, auf zahlreiche Verfahren der Pufferverwaltung, wie den Seitenaustausch zwischen Rechnern, abstützen. Schließlich besteht in der Pufferverwaltung eine einfache Möglichkeit, über die E/A-Schnittstelle die über die Kommunikationssegmente ausgetauschten Zwischenergebnisse auf Festplatte zu materialisieren. Dies kann im Rahmen der Parallelisierung beim Austausch großer Zwischenergebnisse genutzt werden, wenn die Zwischenergebnisse nicht im Puffer Platz finden und der Produzent nicht warten soll, oder für den Fall, daß diese Zwischenergebnisse mehrfach gelesen werden sollen. Gleichzeitig ist im Rahmen der Fehlertoleranz die Möglichkeit der Materialisierung von Zwischenergebnissen von entscheidender Bedeutung beim Neustart fehlerhafter Transaktionen, wie in den Kapiteln 5.7 und 6 über das FPT-Modell, ein Modell zur Fehlerbehandlung in parallelen Transaktionen, gezeigt wird.

Zusammen mit der Entscheidung für die Kommunikationssegmente wurden deren Funktionalität und deren Parameter festgelegt.

Zu diesen Parametern gehören die Wahl

- der Koordination des Datenaustauschs. Es sollte möglich sein, die beim Produzent anfallenden Daten an den Konsumenten weiterzuleiten, wenn jeweils eine Subseite, Seite oder ein ganzes Segment vollständig beschrieben wurde.

- der Kommunikationsform. Dabei wird bestimmt, ob die produzierten Daten vom Produzenten automatisch an den Konsumenten gesendet werden sollen, oder ob dieser jede Dateneinheit anfordern muß.

- der Persistenz der Daten. Es sollte möglich sein, die transferierten Daten nur einmal zu lesen (ReadOnce). Einmal gelesene Daten können dann sofort

verworfen werden, ohne weiteren Speicherplatz zu belegen.

- der Verdrängungsstrategie. Überschreitet ein Konsument das ihm zugewiesene Speicherkontingent, so bestehen folgende Alternativen:
  - Der Produzent wird angehalten, bis erneut Daten zum Konsumenten transferiert werden, Seiten werden nicht verdrängt (WAIT).
  - Vom Produzenten bereits produzierte Seiten werden auf Festplatte ausgelagert. Die neuesten Seiten werden zuerst ausgelagert, da die ältesten Seiten zuerst vom Konsumenten gelesen werden. Dabei kann entschieden werden, ob dies auf die lokale Festplatte des Produzenten, die des Konsumenten oder auf alle Festplatten des Systems geschieht (WRITEOUT).
  - Die Seite wird der normalen LRU-Verdrängungsstrategie der Pufferverwaltung unterworfen (NOBUF). Die Beschränkung durch das Speicherkontingent wird dadurch aufgehoben. Der gesamte Cache kann genutzt werden, wobei allerdings die Konkurrenz durch parallele Anfragen zu beachten ist wie auch der Umstand, daß bei der LRU-Verdrängung die ältesten Seiten zuerst verdrängt werden, obwohl diese zuerst vom Konsumenten gelesen werden.

Nach der vorgenommenen Spezifikation wurde in [Brandmayer 97] in Zusammenarbeit mit Clara Nippl die Implementierung der Kommunikationssegmente vorgenommen.

Zusätzlich wurde die Menge der Operatoren um den SEND- und den RECEIVE-Operator erweitert. Diese bilden die Schnittstelle des Interpretermoduls zu den Kommunikationssegmenten. Das Schreiben beziehungsweise das Lesen der Kommunikationssegmente durch die Operatoren unterscheidet sich dabei für diese Operatoren nicht vom Zugriff auf andere Segmenttypen, da der Zugriff auf die Kommunikationssegmente über dieselbe, seitenorientierte Schnittstelle der Pufferverwaltung abgewickelt wird.

Mit Hilfe der Kommunikationssegmente lassen sich m:n-Kommunikationsverbindungen zwischen Interpretern realisieren. Dabei werden $m * n$ Kommunikationssegmente zu einem Kommunikationskanal zusammengefaßt, über den sich für die Parallelverarbeitung wichtige Kommunikationsformen wie Pipelining, Replikation, Partitionierung, Mischen und Repartionierung realisieren lassen [Nippl 00, Graefe 90] (vergleiche auch Abschnitt 6.1.2).

Über die Kommunikationssegmente wird in MIDAS Intra-Transaktionsparallelität ermöglicht. Diese kann sowohl in Form von Inter- als auch in Form von Intra-Operator-Parallelität eingesetzt werden [Brandmayer 97, Fleischhauer 97, Nippl 00].

Beim Einsatz der WAIT-Strategie kann es in der Parallelverarbeitung zu zyklischen Wartesituationen und damit zu Verklemmungen kommen [NipMit 98b]. Diese können über präventiven Verzicht einiger WAIT-Einstellungen vermieden werden. Als Alternative dazu könnte im Verklemmungsfall zur Laufzeit die WAIT-Einstellung aufgehoben werden.

Weitere Arbeiten [NipZimMit 99, Nippl 00] haben gezeigt, wie die Kommunikationssegmente und deren Parameter am besten eingesetzt werden (siehe auch Abschnitt 3.4).

2.5 Wiederverwendung von Software

Wie schon in den obigen Abschnitten erwähnt, wurden bei der Entwicklung der MIDAS-Komponenten große Teile des Codes des sequentiellen Datenbanksystems TransBase als Codebasis übernommen.

Dieses Vorgehen war einer der grundlegenden Entwicklungsansätze bei der Implementierung des Datenbankprototypen MIDAS. Durch einen möglichst hohen Wiederverwendungsanteil von Software sollte möglichst schnell ein funktionierender Datenbankprototyp entwickelt werden [LiPaReBoLe 95, BJLMRZ 96a, BJLMRZ 96b, SpeZimCla 97]. Durch schrittweises Erweitern, Ergänzen und Umstrukturieren des Quellcodes wurde MIDAS um die verschiedenen Möglichkeiten der Parallelverarbeitung erweitert.

Modul	Zeilen C-Code TransBase	(MIDAS)	unverändert	angepaßt	nicht benutzt
Anfragesystem	47000	(110000)	81 %	7 %	12 %
Interpreter	15000	(30000)	95 %	0 %	5 %
Cache/Lock-Server	18000	(60000)	20 %	2 %	78 %
alle Komponenten	90000	(200000)	75 %	5 %	20 %

Tabelle 2.1: Wiederverwendung der TransBase-Quellen

Die als Codebasis dienende sequentielle TransBase-Version 3.3 hat einen Codeumfang von ca. 90000 Zeilen C-Code. Um die genaue Menge der wiederverwendeten Codeteile zu zeigen, wurde die Tabelle 2.1 erstellt. In dieser Tabelle wird in jeder Zeile eine Teilkomponente (Modul) des Gesamtsystems beschrieben. Die erste Spalte gibt die Größe des jeweiligen Moduls in TransBase an. In Klammern sind

noch zusätzlich die momentanen Größen der entsprechenden Module in MIDAS vermerkt. Die weiteren Spalten geben die Prozentsätze an, in denen die Original-TransBase-Quellen in MIDAS wiederverwendet werden. Unterschieden wird dabei in Codeteile, die unverändert benutzt, Teile, die an MIDAS angepaßt und Teile, die nicht wieder benutzt werden konnten.

Besonders erwähnenswert hierbei sind die hohen Prozentsätze in Anfragesystem und Interpreter, die wiederverwendet werden konnten. Bei beiden Komponenten mußten fast ausschließlich Erweiterungen wie Kommunikationsmechanismen (PVM) zwischen den Komponenten ergänzt werden. Eine grundlegende Umstrukturierung der Funktionsweisen war nicht notwendig. Die Tatsache, daß beide Komponenten in MIDAS deutlich größer als in TransBase sind, liegt daran, daß beim Applikationsserver neue Module, wie Optimierer (17000), Parallelisierer (32000) oder Scheduler (2000), und beim Interpreter neue Operatoren (insgesamt 10000 Zeilen durch Hash-Join, parallele Relationen-Scans, SEND, RECEIVE und UDTO) hinzugefügt wurden.

Auf der Ebene der Cache- und Sperrverwaltung fiel der Wiederverwendungsanteil gering aus. Dies war allerdings zu erwarten, da auf diesen Ebenen eine verteilte Puffer- und Sperrverwaltung implementiert werden mußte, die in einer solchen Form in TransBase nicht vorhanden ist.

Wie sich aus der Tabelle 2.1 erschließen läßt, wurden bei der Implementierung des Prototypen 128000 Zeilen Code neu entwickelt[6], 67500 Zeilen unverändert und 4500 verändert von TransBase übernommen.

Insgesamt ließ sich feststellen, daß die TransBase-Quellen als Basisgerüst gut verwendbar waren, und daß der auf Wiederverwendung basierende Entwicklungsansatz erfolgreich war und sehr schnell zu einen lauffähigen Basissystem geführt hat.

2.6 Portierbarkeit von Anwendungen

In MIDAS wurde neben der Wiederverwendung von TransBase-Software zusätzlich darauf Wert gelegt, alte TransBase-Anwendungen auch auf MIDAS ausführen zu können. Dazu waren zwei Maßnahmen erforderlich.

Zum ersten wurde die Schnittstelle für die Anwendungsprogramme (TBX) unverändert von TransBase übernommen, so daß alle TransBase-Anwendungen ohne Neuübersetzung, nur durch Binden mit der neuen MIDAS-TBX-Bibliothek, auf MIDAS portiert werden können.

[6] Dies ergibt sich aus dem Umfang des Codes von MIDAS minus 75 % + 5 % des TransBase-Codes, die unverändert beziehungsweise angepaßt wiederverwendet wurden.
$200000 - (67500 + 4500) = 128000$.

Auf diese Weise konnten TransBase-Anwendungen wie *tbi*, *ufi* [TransBaseS 88, TransBaseR 88] oder *xufi* erfolgreich auf MIDAS portiert werden.

Zum zweiten wurde die Pufferverwaltung so konzipiert, daß die von TransBase-Anwendungen erzeugten Datenbanken mit sehr geringem Aufwand in Datenbanken für MIDAS konvertiert werden können (siehe Abschnitt 3.1).

Somit konnten auch Anwendungen, die eigene Datenbanken besitzen, portiert werden. Als Beispiel für eine solche Anwendung dient OMNIS/Myriad, ein System zur Verwaltung von Multimedia-Dokumenten in Bibliotheken und im Büro. Es ist auf einem Netz von Workstations als Client-Server Architektur realisiert und unterstützt die Archivierung, die Volltext-Recherche und die Ausleihe von Dokumenten [ClaVogWie 95, Bayer 95, CJMNRZ 97]. OMNIS wurde in [Schult 97] erfolgreich auf MIDAS portiert.

2.7 Werkzeuge

Im Rahmen einer Diplomarbeit [Pries 97] wurde ein umfangreiches Tracingsystem in MIDAS integriert, mit dem die meisten der in dieser Arbeit aufgeführten Meßergebnisse ermittelt wurden. Dieses System erlaubt eine genauere Analyse der verwendeten Algorithmen und der internen Abläufe. Es umfaßt Lastgeneratoren, eine Bibliothek mit Funktionen zum Messen von Ereignissen, die in MIDAS an vielen Stellen integriert wurde, sowie Auswertungsprogramme für die angefallenen Daten.

Aufbauend auf die vom Tracingsystem generierten Daten, konnte eine Visualisierung von zeitlichen Abläufen im System vorgenommen werden. Diese Abläufe umfassen unter anderem Transaktions- und Interpreterlaufzeiten, Festplattenzugriffe, Cachetrefferraten und Cacheinhalte. Dazu wurde das in [Fehr 97] entwickelte Visualisierungswerkzeug für DBSIM [Bozas 98] auf MIDAS portiert [Tschaffler 98]. Zusätzlich zur Portierung der graphischen Oberfläche und der Funktionalitätserweiterung um Aspekte wie Intra-Transaktionsparallelität, die in DBSIM nicht analysiert werden, wurde auch eine Reihe von Statistiken angefertigt (z. B. Nachrichten- oder Festplattenzugriffsstatistiken), welche auch dazu dienten, das Verhalten von MIDAS zu evaluieren (siehe Meßergebnisse in Abschnitt 3.5 und Kapitel 4).

Schließlich wurde noch ein Werkzeug zur parallelen Übersetzung der MIDAS-Quellen entworfen, um so die Entwicklungszeiten des Systems verkürzen zu können.

2.8 Zusammenfassung

Dieses Kapitel stellte das relationale, parallele Datenbanksystem MIDAS vor. Es wurde eine Übersicht über die Komponenten des Systems präsentiert und viele an der Entwicklungen des Systems beteiligten Arbeiten aufgeführt.

Der Übergang von TransBase zu MIDAS konnte erfolgreich vollzogen werden. Große Teile des TransBase-Systems konnten in MIDAS wiederverwendet werden und halfen bei der schnellen Entwicklung eines funktionstüchtigen Systems. MIDAS stellt sich inzwischen als überzeugender Datenbankprototyp vor, mit dem es möglich ist, reale Anwendungen, wie z. B. das OMNIS System oder Data-Warehouse-Szenarien, auf MIDAS zu portieren und zu parallelisieren.

Kapitel 3

Implementierungsaspekte des Ausführungssystems

In der Praxis ist es wichtig festzustellen, wie die vorgeschlagenen und in MIDAS umgesetzten Konzepte das System beeinflussen. So wurden beispielsweise Mechanismen zur Synchronisation auf Datenstrukturen von TransBase übernommen und es entstanden neue Aspekte wie die zu entwickelnden Kommunikationsmechanismen oder die Verteilung des Systems auf mehrere Rechner, deren effiziente Implementierung entscheidend für die Güte des Laufzeitsystems ist.

In diesem Kapitel werden die interessantesten Aspekte der Implementierung des Ausführungssystems vorgestellt und gezeigt, welchen Einfluß sie auf die Leistungsfähigkeit von MIDAS haben.

In Abschnitt 3.1 wird dargestellt, wie die einfache Portierung alter TransBase-Anwendungen und -Datenbanken durch das Design des Ausführungssystems unterstützt wird. Abschnitt 3.2 beschreibt die Architektur des Ausführungssystems und wie durch Synchronisationsverfahren die Parallelität beeinflußt wird. Danach folgen in Abschnitt 3.3 und 3.4 Analysen der im Ausführungssystem vorkommenden Kommunikationsmechanismen, der Nachrichtenbibliothek PVM und der Kommunikationssegmente.

Die verwendeten Datenverteilungen auf Festplatte und die Probleme, die durch Caching des Betriebssystems entstehen, werden in Abschnitt 3.5 dargestellt.

Schließlich werden in Abschnitt 3.6 der verwendete Algorithmus zur Kohärenzkontrolle und die Synchronisation beschrieben. Abschnitt 3.7 stellt eine dynamische Verteilung von Cacherahmen vor.

3.1 Portierbarkeit – Entwicklung von TransBase zu MIDAS

Bei der Entwicklung von TransBase zu MIDAS wurden viele Codeteile unverändert übernommen, um schnell zu einem funktionstüchtigen Prototypen zu gelangen (siehe Abschnitt 2.5). Des weiteren sollte MIDAS kompatibel zu TransBase sein, d. h. Anwendungen, die für TransBase programmiert wurden, sollten mit geringem Aufwand auch mit MIDAS als Datenbanksystem verwendet werden können (siehe Abschnitt 2.6).

Um diese Kompatibilität zu gewährleisten wurde MIDAS so konzipiert, daß die Applikationen leicht portiert und auch die von ihnen verwendeten Datenbanken nach MIDAS transferiert werden können.

Die Portierbarkeit der Anwendungen wurde dadurch erzielt, daß die Schnittstelle der Programmbibliothek für die Anwendungen identisch übernommen wurde (TBX-Schnittstelle, vergleiche Abschnitt 2.3). Alle TransBase-Anwendungen können somit ohne Neuübersetzung, nur durch Binden mit der neuen MIDAS-Bibliothek auf MIDAS portiert werden.

Die zweite zu lösende Aufgabe war die Portierung der Datenbanken der TransBase-Anwendungen. In MIDAS sollte das in [Listl 96] vorgestellte Konzept der Subseiten implementiert werden. Dabei werden die Seiten der Pufferverwaltung in Subseiten gleicher und fester Länge unterteilt. Diese Subseiten werden als die Einheiten der Konsistenzkontrolle verwendet. Ziel dieses Konzeptes ist es zum einen, eine einfache Konsistenzkontrolle durch Verwendung von Sperren auf Objekten fester Größe[1] zu erhalten. Zum anderen soll die Anzahl der Sperrkonflikte reduziert werden, indem ein kleineres Granulat als eine ganze Seite zur Synchronisation verwendet wird. Um E/A-Kosten zu sparen, findet die Ein- und Ausgabe weiterhin auf großen Einheiten, den Seiten, statt (siehe Abschnitt 2.4.7.1).

Damit werden an der oberen Schnittstelle der Pufferverwaltung zu den Interpretern Subseiten als kleinste adressierbare Einheiten zur Verfügung gestellt, wohingegen von der E/A-Komponente und in der Pufferverwaltung[2] zwischen den Rechnern ganze Seiten transferiert werden.

Bei TransBase werden an den Schnittstellen der Pufferverwaltung zum Interpreter beziehungsweise zur E/A-Komponente identische Einheiten ausgetauscht. Die Pufferverwaltung wurde komplett neu entwickelt, so daß dort die Algorithmen entsprechend für Seiten und Subseiten ausgelegt werden konnten. Da aber sowohl die Interpreter als auch die verwendeten Datenbanken möglichst unverändert

[1] In diesem Fall sind es Subseitenperren fester Größe, im Gegensatz zu Tupelsperren, die verschiedene große Objekte (Tupel) sperren.

[2] In der Pufferverwaltung werden in manchen Situationen auch Subseiten versendet.

übernommen werden sollten, mußte hierfür eine Lösung gefunden werden.

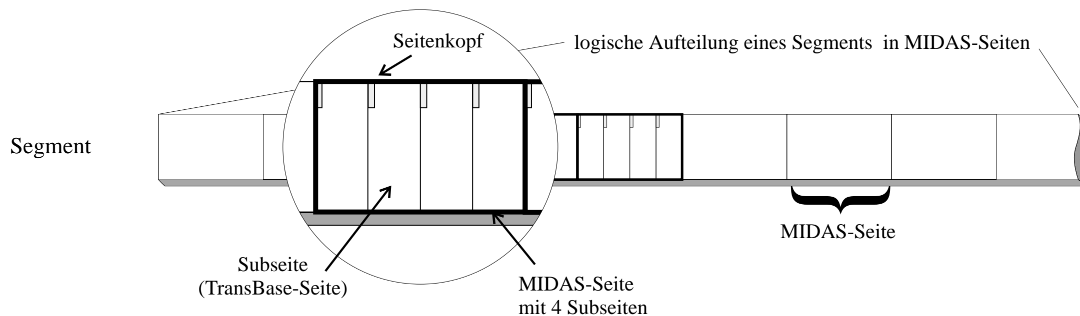

Abbildung 3.1: Aufbau eines Segments

Dazu wurden die Segmente der Pufferverwaltung in *MIDAS-Seiten* unterteilt, die wiederum aus mehreren *Subseiten* bestehen. Die Subseiten entsprechen den von TransBase übernommenen Seiten. Sie können somit, ohne Veränderungen am Interpreter vornehmen zu müssen, an diesen weitergegeben werden. Die Subseiten enthalten damit den bei TransBase-Seiten üblichen Seitenkopf, in dem Verwaltungsinformationen wie die Segment- und Seitennummer enthalten sind.

Um auch die Struktur der auf Festplatte gespeicherten TransBase-Segmente beibehalten zu können, wurden die MIDAS-Seiten als Folge der im TransBase-Segment direkt aufeinanderfolgenden TransBase-Seiten festgelegt. MIDAS-Seiten dürfen keinen Seitenkopf enthalten. Mit diesen Festlegungen sind die TransBase-Segmente, die Folgen von TransBase-Seiten enthalten, gleichzeitig auch Segmente, die Folgen von MIDAS-Seiten enthalten. Somit sind insbesondere Umstrukturierungen der Segmente, die ein aufwendiges Anpassen der Seitenköpfe zur Folge gehabt hätte, vermieden worden. Abbildung 3.1 zeigt den Aufbau eines Segments, wie es derzeit in MIDAS verwendet wird.

Damit mußte nur die E/A-Komponente umgeschrieben werden, so daß sie statt einzelner TransBase-Seiten aufeinanderfolgende Blöcke von TransBase-Seiten liest und diese als MIDAS-Seiten interpretiert. Die einzige Anpassung, die damit an den Segmenten auf Festplatte vorgenommen werden mußte, ist die Ergänzung der Segmente um leere Seiten am Ende der entsprechenden Datei, so daß die Anzahl der Seiten im Segment ein Vielfaches der Subseitenanzahl pro Seite ist. Mit diesem Konzept war die Umstellung auf das Seiten-Subseiten-Konzept in den an die Pufferverwaltung angrenzenden Modulen sehr einfach zu realisieren.

Eine Evaluierung der gesamten Umstellung von TransBase auf MIDAS ist in Abschnitt 4.1 zu finden. Der Einfluß der Subseitenanzahl pro Seite und damit der Seitengröße auf die Leistung des Systems wird in Abschnitt 4.3.3 diskutiert.

3.2 Prozeßdesign und Parallelität im System

Dieser Abschnitt betrachtet Architekturentscheidungen sowie Aspekte der Parallelverarbeitung im Ausführungssystem. Dabei werden Techniken und Entscheidungen vorgestellt, die notwendig waren, um zu einer leistungsfähigen Systemarchitektur zu gelangen.

3.2.1 Architekturumstellung

Dieser Abschnitt beschreibt eine Änderung des ursprünglichen Prozeßmodells von MIDAS. Die Änderung wurde am Ausführungssystem von MIDAS vorgenommen und hatte zum Ziel, die rechnerlokale Kommunikation zu minimieren, da die Anzahl der Nachrichten im System ein stark leistungsbegrenzender Faktor ist, wie auch in Abschnitt 3.3 gesehen werden kann.

Dazu wurden die Interpreterkomponenten mit den Komponenten der Pufferverwaltungsebene zu einem Prozeß verschmolzen [Fleischhauer 97, Pries 97]. Die Architektur vor der Umstellung wird im folgenden als *altes Architekturmodell* bezeichnet, die nach der Umstellung als *neues Architekturmodell*. Das neue Architekturmodell entspricht der derzeit in MIDAS verwendeten Prozeßstruktur, die in Abschnitt 2.3.2 beschrieben wurde.

Während der Abarbeitung eines Operatorbaumes muß der Interpreter Aufträge an die Pufferverwaltungsschicht des Datenbanksystems senden. Der häufigste Auftrag ist dabei die Bereitstellung einer Datenbankseite im Datenbankcache. Bevor der Interpreter auf ein bestimmtes Tupel einer Relation zugreifen kann, muß sich die entsprechende Datenbankseite, die das Tupel enthält, im Datenbankcache befinden. Welche Seite dies ist, entnimmt der Interpreter der Zugriffspfadstruktur. Diese stellt in den meisten Fällen einen Präfix-B-Baum dar. Für die entsprechende Seite wird daraufhin vom Interpreter ein RFIX- oder WFIX-Auftrag an die Pufferverwaltungsschicht gestellt mit der Folge, daß die Seite, sofern sie sich nicht schon im Cache befindet, von der Pufferverwaltungsschicht dorthin geholt und fixiert, d. h. als unverdrängbar gekennzeichnet wird. Der Interpreter kann dann auf diese Seite über Shared-Memory direkt zugreifen und das entsprechende Tupel lesen oder verändern. Wenn die Seite vom Interpreter nicht mehr benötigt wird, sendet er wiederum einen Auftrag an die Pufferverwaltungsschicht (UNFIX), und die Seite wird als verdrängbar eingetragen, damit sie bei Bedarf aus dem Cache ausgelagert werden kann.

3.2.1.1 Das alte Architekturmodell

Im alten Architekturmodell waren Interpreter sowie die Pufferverwaltungsschicht (Cache/Lock-Server-Prozeß) eigene Prozesse. Diese Aufteilung wurde bei den er-

sten Implementierungen des Ausführungssystems so gewählt, da sie der modularen Aufteilung der Systemkomponenten entsprach [LiPaReBoLe 95]. Die Interpreterprozesse wandten sich mit Anfragen an die Cacheebene immer an den lokalen, d. h. auf demselben Rechner laufenden Cache/Lock-Server-Prozeß. Die komplette Kommunikation zwischen Interpreter und Pufferverwaltungsschicht erfolgte somit über rechnerlokale PVM-Kommunikation.

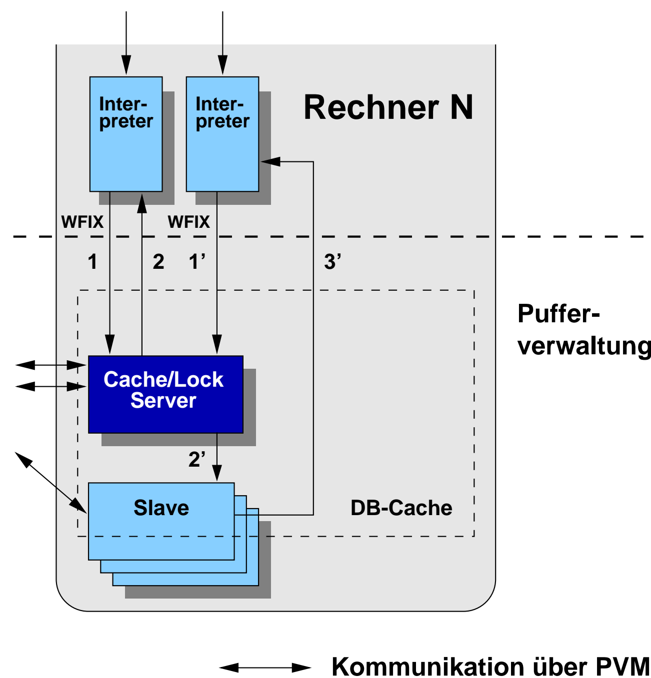

Abbildung 3.2: Das alte Architekturmodell

In Abbildung 3.2 sind die Nachrichtenverläufe für das alte Architekturmodell an Hand eines WFIX-Auftrags graphisch dargestellt. Der Auftrag wird in Form einer PVM-Nachricht an den lokalen Cache/Lock-Server-Prozeß gesendet (Nachricht 1). Der Cache/Lock-Server prüft, ob sich die gewünschte Seite schon im lokalen Datenbankcache befindet. Ist dies der Fall, so ist der Auftrag erledigt, und er kann die Adresse der Seite im Datenbankcache, der als Shared-Memory-Region realisiert ist und mit den Interpretern geteilt wird, direkt an den Interpreter zurücksenden (Nachricht 2). Minimal sind also zwei lokale Nachrichten zur Bearbeitung des Auftrags nötig.

Falls die Seite jedoch noch nicht im lokalen Cache ist, muß sie entweder von Festplatte eingelesen oder von einem anderen Rechner geholt werden. Beides kostet relativ viel Zeit, während der der Cache/Lock-Server keine anderen Aufträge

bearbeiten kann. Außerdem besteht die Gefahr von Verklemmungen. Um dies zu vermeiden, wird der WFIX-Auftrag (Nachricht 1') an einen der lokalen Slaves weitergereicht (Nachricht 2'). Der Cache/Lock-Server steht damit für weitere Aufträge von anderen Interpretern zur Verfügung. Der Slave empfängt den Auftrag und sorgt dafür, daß die Seite in den Cache gelangt, indem er sie z. B. von Festplatte liest. Sobald die Seite im Cache fixiert ist, meldet der Slave die erfolgreiche Ausführung des Auftrags direkt an den Interpreter zurück (Nachricht 3'). Im ungünstigsten Fall werden daher drei lokale Nachrichten pro Auftrag benötigt[3]. Außerdem kommt es durch die Anzahl der beteiligten Prozesse zu mindestens drei Prozeßwechseln.

Da es sich bei diesen Nachrichten um "richtige" Nachrichten handelt, die über Sockets realisiert werden, ist ersichtlich, daß die Schnittstelle zwischen Interpreter und Pufferverwaltungsschicht die Performanz sehr stark beeinträchtigt.

Die Zahl der Aufträge, die vom Interpreter an die Pufferverwaltungsschicht gesendet werden, ist ein kritischer Faktor. Als Beispiel diene eine einfache Anfrage, die nur ein Tupel liest. Sie wird etwa drei Seitenzugriffe zur Auffindung der B-Baum-Blattseite absetzen, d. h. sechs Nachrichten (RFIX und UNFIX). Zusätzlich muß das Segment über zwei Nachrichten geöffnet und geschlossen werden. Selbst bei einer so einfachen Anfrage sind im alten Architekturmodell nur zur Kommunikation zwischen Interpreter und Cache/Lock-Server 8 Nachrichten notwendig.

Da die Performanz dieser Implementierung nicht zufriedenstellend war, wurde über eine Umstellung der Architektur nachgedacht. Als erfolgversprechender Ansatz erwies sich eine Zusammenfassung der Funktionen von Interpreter und Segmentschicht in einem Prozeß mit dem Ziel, alle FIX-Aufträge prozeßlokal als Funktionsaufrufe behandeln zu können.

3.2.1.2 Das neue Architekturmodell

Aufgrund der im vorherigen Abschnitt beschriebenen Leistungsdefizite wurde die Architektur des Ausführungssystems umstrukturiert. Ziel war es, die lokale Kommunikation über Nachrichten zwischen Interpreter und Pufferverwaltungsschicht zu vermeiden. Dazu wurden Interpreter-, Cache/Lock-Server- und Slave-Prozeß zu einem Prozeß zusammengefaßt, wodurch die aktuelle Systemarchitektur entstand, die in den Abschnitten 2.3 und 2.3.2 beschrieben wurde. Der grundlegende Gedanke war, daß der Interpreter-Prozeß, der direkten Zugriff auf den lokalen Datenbankcache hat, selbst dafür sorgen kann, daß eine Seite in den Cache gelangt. Er muß dazu nicht den Cache/Lock-Server beauftragen. Da der Interpreter im

[3]Eventuell werden vom Slave auch externe Nachrichten, d.h. Nachrichten an andere Cache/Lock-Server versendet. Für die Architekturumstellung ist jedoch nur die Betrachtung lokaler Kommunikation wichtig, da diese externe Kommunikation auch im neuen Architekturmodell notwendig ist.

alten Modell ohnehin so lange wartet, bis der Auftrag von Cache/Lock-Server und Slave bearbeitet wird, geht bei dieser Konstruktion auch keine Parallelität zwischen den Prozessen verloren.

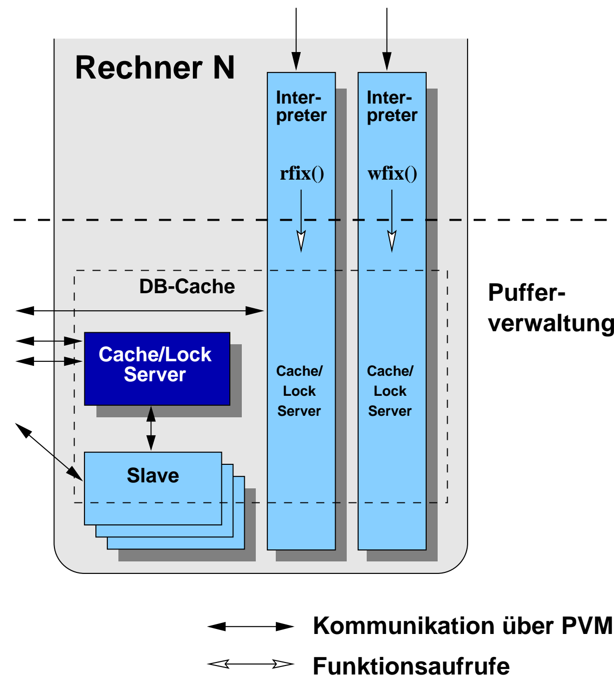

Abbildung 3.3: Das neue Architekturmodell

Abbildung 3.3 zeigt eine detaillierte Darstellung des neuen Architekturmodells, das schon in den Abbildungen 2.3 und 2.7 vorgestellt wurde. Die lokalen Nachrichten zwischen Interpreter und Cache/Lock-Server sind jetzt vollständig entfallen. Cache/Lock-Server und Slaves bearbeiten nur noch externe Aufträge.

Im neuen Modell sind alle Module der Pufferverwaltungsschicht zu den Interpretern hinzugebunden. Dadurch ist der Interpreter zugleich ein vollwertiger Cache/Lock-Server und auch sein eigener Slave. Alle Aufträge an die Pufferverwaltungsschicht sind nun prozeßlokale Funktionsaufrufe und sind keine Nachrichten mehr. Auch ein Weiterleiten von Aufträgen an Slaves, wie dies vorher notwendig war, entfällt, da der Interpreter keine anderen Prozesse in ihrer Parallelverarbeitung behindern kann. Falls er eine Seite laden oder auf die Beantwortung externer Nachrichten warten muß, ist nur er selbst und kein anderer Prozeß betroffen.

Ein möglicher Nachteil des neuen Modells ist, daß durch das Zusammenlegen der Prozesse Parallelverarbeitung zwischen den Komponenten unterbunden wird.

Zum einen könnten Interpreter und Slaves asynchron arbeiten. Das vorhandene Aufrufsystem sieht aber nur synchrone Aufrufe vor, so daß dieser Nachteil nicht zum Tragen kommt. Zum anderen bestünde die Möglichkeit, mehrere Slaves parallel für sich arbeiten zu lassen. Auch diese Variante wurde von der Parallelisierung bis dahin nicht genutzt, daher entstehen keine Einschränkungen.

Der Cache/Lock-Server und seine Slaves sind allerdings nicht komplett entfallen, da es nach wie vor notwendig ist, Sperren- und Seitenanfragen von Prozessen auf anderen Rechnern zu bearbeiten. Der Pool an Slaves ist ebenfalls weiter notwendig, da externe Aufträge eventuell an Slaves weitergegeben werden müssen, um Wartezustände und damit Verklemmungen im Cache/Lock-Server zu verhindern.

In der Implementierung sind Interpreter, Cache/Lock-Server und Slaves absolut identisch, d. h. sie sind alle Instanzen derselben ausführbaren Datei. Dies wurde aus Gründen der Speicherersparnis auf diese Weise implementiert. Zum einen benötigen somit alle auf einem Rechner laufenden Instanzen des Ausführungssystems nur eine Kopie des Objektcodes im Speicher (Code-Sharing). Zum anderen kann dadurch die Anzahl der Prozesse auf einem Rechner klein gehalten werden. Wie in Abschnitt 2.3 beschrieben, werden dynamische Komponenten des Ausführungssystems wie Interpreter und Slaves in Pools gehalten, um die Prozeßstartkosten zu minimieren. Da Interpreter und Slaves identische Prozesse sind, kann dies in einem einzigen Pool geschehen. Die entsprechende Instanz muß nur nach ihrer jeweiligen Verwendung initialisiert werden. Dadurch kann die Anzahl ungenutzter Prozesse im System niedrig gehalten werden.

Im nächsten Abschnitt werden die Vorteile der neuen Architektur an Hand von Messungen belegt.

3.2.1.3 Bewertung der Architekturumstellung

Die Messungen fanden auf dem in Anhang A beschriebenen 4-Prozessorsystem statt. Zu jedem Zeitpunkt war nur eine Transaktion aktiv. Als Lasten wurden die TPC–A^{RO}-, TPC–C^{S+O}- und TPC–C^{S}-Transaktionen verwendet (siehe Anhang D). In Tabelle 3.1 sind die Meßergebnisse gegenübergestellt.

Die Ergebnisse fallen klar zugunsten der neuen Architektur aus. Hier wurden Verkürzungen der Laufzeit von bis zu 74 % gemessen. Besonders bei Transaktionen, die viele Seiten lesen, ist die Einsparung in der Anzahl der PVM-Nachrichten gravierend. Die verbleibenden Nachrichten ergeben sich aus der Kommunikation zwischen Applikationsserver und Interpreter.

Zusammenfassend kann gesagt werden, daß die Umstellung der Architektur eine wesentliche Verbesserung der Leistungsfähigkeit bewirkt hat. Es ging zwar ein theoretisches Potential an Parallelität verloren, dieses wurde aber auch vor der Umstellung nicht genutzt. Durch die Vereinigung der Prozesse der Segment-

Transaktion	Wert pro TA	alte Architektur	neue Architektur
TPC–A^{RO}	Gesamtlaufzeit TA	119 ms	55 ms
	PVM-Nachrichten	102	19
TPC–C^{S+O}	Gesamtlaufzeit TA	6.324 ms	1.642 ms
	PVM-Nachrichten	6.360	105
TPC–C^{S}	Gesamtlaufzeit TA	23.424 ms	6.385 ms
	PVM-Nachrichten	25.190	21

Tabelle 3.1: Nachrichtenaufkommen bei alter und neuer Architektur

schicht mit dem Interpreter entfällt sämtliche für das Lesen von Seiten bisher notwendige Kommunikation. Daraus resultiert eine enorme Beschleunigung der Ausführung von Transaktionen. Diese Vorteile machen sich in jeder Konfiguration des Systems bemerkbar, da die Änderung grundlegender Natur ist. Es zeigt sich erneut, daß besonders in kommunikationsintensiven Anwendungen das Einsparen von Nachrichten eine der wichtigsten Möglichkeiten der Leistungssteigerung ist.

Eine weitere Variante, Nachrichten im System einzusparen, wird mit dem lokalitätsbasierten Transaktionsrouting in Abschnitt 4.4.4 vorgestellt.

3.2.2 Threads im Ausführungssystem

Eine sehr vielversprechende Optimierungsmöglichkeit ist der Einsatz von Threads im Ausführungssystem. Die Architektur des Ausführungssystems würde sich dann so ändern, daß pro Rechner nur noch ein Prozeß existiert, der das gesamte Ausführungssystem darstellt. Alle bisher existierenden Prozesse würden in Threads innerhalb dieses einen Prozesses umgewandelt werden.

Eine derartige Architektur profitiert zum einen von den billigen Kontextwechseln zwischen Threads. Teure Prozeßwechsel sind dann nicht mehr notwendig.

Zum anderen könnten alle Prozeßpools (siehe Abschnitt 2.3) entfallen, da beim Erzeugen von Threads wesentlich weniger Kosten entstehen als bei der Erzeugung eines Prozesses.

Die größten Einsparungen sind bei einer solchen Architektur durch die Reduzierung von Nachrichten zu erwarten. Derzeit wird jeder Auftrag, der einen Cache/Lock-Server-Prozeß erreicht und dessen Ausführung möglicherweise blockierend ist, per Nachricht an einen Slave des Cache/Lock-Servers zur Bearbeitung weitergeleitet. Durch diesen Mechanismus wird das Blockieren der Cache/Lock-Server verhindert. Beim Einsatz von Threads kann diese Nachricht

eingespart werden, da sie dort durch einen einfachen Aufruf eines Threads realisiert werden kann. Auf diese Weise könnten $\frac{1}{4}$ bis $\frac{1}{3}$ der kleinen Kontrollnachrichten in der Pufferverwaltung eingespart werden.

Leider ist PVM nicht thread-sicher. Die Threads eines Prozesses arbeiten auf gemeinsamen globalen Daten. Würde ein Thread-Wechsel beim Empfangen einer Nachricht eintreten, so wäre aufgrund der Implementierung von PVM mit globalen Nachrichtenpuffern nicht sichergestellt, daß der richtige Thread die richtige Nachricht erhält, da die globalen Datenbereiche von PVM unsynchronisiert sind und daher überschrieben werden können.

Eine Realisierung des Ausführungssystems als einen einzigen, in Threads unterteilten Prozeß ist damit nicht möglich, solange die aktuelle PVM-Version im System verwendet wird. Die Entwicklung einer thread-sicheren PVM-Version wurde zwar schon angekündigt, ist aber bisher nicht verwirklicht worden.

3.2.3 Synchronisation

Im Ausführungssystem liegen alle wichtigen Datenstrukturen, wie beispielsweise die Seiten des Datenbankcaches und alle zugehörigen Verwaltungsstrukturen, in einem Shared-Memory-Bereich, der von allen Prozessen des Ausführungssystems geteilt wird. Die Synchronisation konkurrierender Zugriffe auf diese Datenstrukturen stellt einen leistungskritischen Teil der Implementierung des Ausführungssystems dar. Zu restriktive Synchronisationsmaßnahmen behindern die Parallelität im System, wohingegen Synchronisation auf sehr feinen Granulaten zuviel Verwaltungsaufwand erzeugt.

3.2.3.1 Semaphore und Latches

Dieser Abschnitt beschäftigt sich mit der am besten geeigneten Implementierung der Synchronisation. Die zu synchronisierenden Zugriffe auf Datenstrukturen der Pufferverwaltung sind alle von sehr kurzer Dauer. Meist werden nur kurze Listen durchlaufen oder auf Hash-Tabellen zugegriffen. Zudem sind diese Zugriffe sehr häufig. Pro Seitenzugriff wird in der derzeitigen Implementierung auf etwa drei zu synchronisierende Bereiche zugegriffen.

Zur Synchronisation boten sich für MIDAS zwei Mechanismen an:

- **Semaphore des Betriebssystems:**

  Diese Semaphore sind leicht zu verwenden, da sie in Form eines Betriebssystemdienstes angeboten werden. Nachteil der Betriebssystemsemaphore ist, daß jedes Belegen oder Freigeben einen relativ teuren Aufruf des Betriebssystemkerns und zusätzlich einen Prozeßwechsel verursacht.

- **Latches:**

  Latches stellen einen Mechanismus für Kurzzeitsperren dar, der im aktuellen Prozeß ausgeführt werden kann. Dabei wird in einer Warteschleife eine atomare Test-And-Set-Operation auf eine Sperrvariable im gemeinsamen Speicher durchgeführt, bis diese nicht mehr belegt ist. Damit wird ein sogenanntes Busy-Waiting implementiert, d. h. der Prozeß, der versucht die Sperre zu bekommen, ist aktiv und verbraucht CPU-Zeit.

  Vorteil der Latches ist, daß bei Anforderung der Sperre kein Prozeßwechsel und kein teurer Aufruf in den Betriebssystemkern erfolgt. Als Nachteil könnte sich die starke CPU-Belastung herausstellen.

  Um diesen Nachteil gegebenenfalls abzuschwächen, wurden noch zwei weitere Varianten von Latches implementiert. Diese Varianten erzwingen einen Prozeßwechsel, wenn zu viele Wartezyklen durchlaufen wurden, ohne die Sperre bekommen zu haben. In der Implementierung geschieht dies einmal nach kurzem Warten von einem Zyklus und einmal nach vergleichsweise langem Warten von 40 Zyklen. Dadurch sollen andere, momentan nicht aktive Prozesse, die nicht nur Busy-Waiting-Schleifen durchlaufen, die CPU nutzen können.

Die Güte der verschiedenen Mechanismen wurde durch eine Messung bestimmt. Dabei wurden auf dem 4-Prozessorsystem (siehe Anhang A) mehrere Anfragen parallel ausgeführt und sowohl deren Anzahl als auch der Synchronisationsmechanismus variiert. Als Anfragen wurden Scans auf Relationen, die sich komplett im Datenbankcache befanden, verwendet. Dadurch wird eine hohe Aktivität der Pufferverwaltung und damit eine hohe Anzahl benötigter Synchronisationsoperationen erzeugt. Gleichzeitig findet keinerlei Ein- und Ausgabe statt, die die Unterschiede zwischen den Synchronisationsmechanismen überdecken könnte.

Abbildung 3.4 zeigt, wie sich die Ausführungszeiten der verschiedenen Synchronisationsmechanismen bei unterschiedlichen Parallelitätsgraden verhalten.

Betriebssystemsemaphore erweisen sich bei allen getesteten Parallelitätsgraden als der deutlich schlechteste Mechanismus. Bei ihrer Verwendung sind die Ausführungszeiten zwischen vier und sechs mal so lang wie bei der jeweils günstigsten Variante der Latches.

Die Busy-Waiting-Latches weisen bis zum Parallelitätsgrad von 16 die besten Ausführungszeiten auf. Erst bei höherem Parallelitätsgrad ist die Latches-40 Variante die günstigste.

Diese Beobachtungen sind leicht zu erklären. Die Nachteile der Semaphore wurden oben schon erwähnt. Häufige Betriebssystemkernaufrufe und Prozeßwechsel sind zu teuer. Die Busy-Waiting-Latches profitieren gegenüber den anderen Varianten der Latches, da sie keinen Prozeßwechsel erzeugen. Ihr verschwenderischer Umgang mit CPU-Zeit kommt erst bei sehr hohem Parallelitätsgrad zum Tragen,

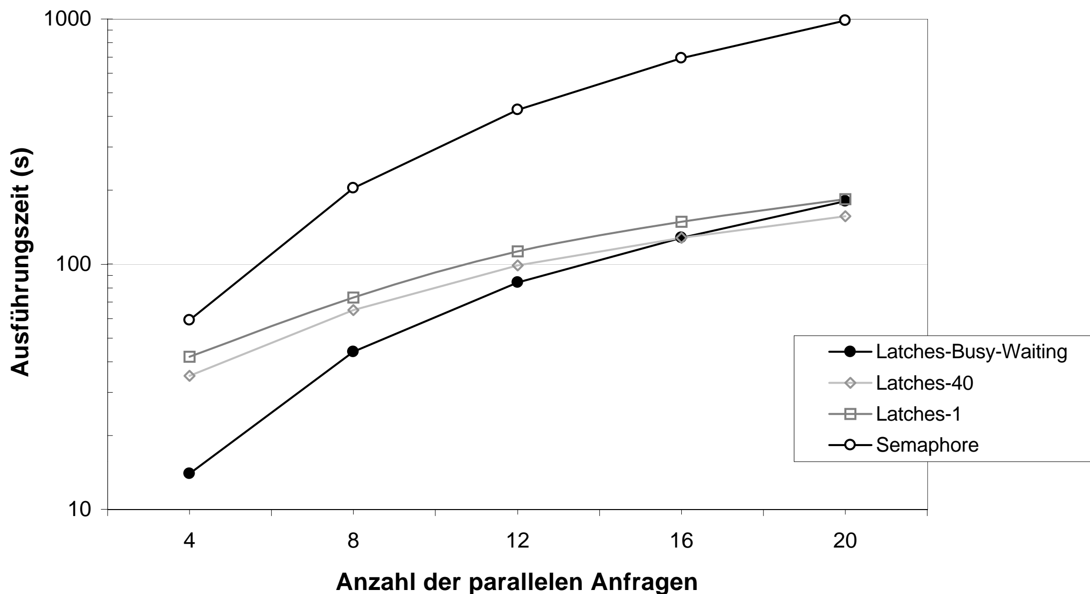

Abbildung 3.4: Semaphore versus Latches

da auf die gesperrten Datenstrukturen immer nur sehr kurz zugegriffen wird. Die Sperren werden schnell wieder freigegeben und somit muß nie lange gewartet werden.

Wie in Abschnitt 4.4.1 gezeigt wird, liegt der Parallelitätsgrad, bei dem der höchste Durchsatz erreicht wird, bei maximal zwei bis drei Prozessen pro zur Verfügung stehendem Prozessor. Auf dem 4-Prozessorsystem ist dies demnach ein Parallelitätsgrad von 8 bis 12. Bis zu diesem Bereich sind die Busy-Waiting-Latches deutlich die beste Variante, so daß sie durchgehend in MIDAS eingesetzt werden.

3.2.3.2 Auswirkungen auf die parallele Architektur

Nachdem der vorangegangene Abschnitt die Implementierung der Synchronisationsmechanismen behandelt hat, wird im folgenden der Einsatz der Synchronisation und seine Auswirkungen auf die Parallelität im System diskutiert. Es stellt sich die Frage, wie fein das Granulat gewählt werden soll, auf dem synchronisiert wird.

In den ersten Implementierungen der Pufferverwaltung wurde fast jede wichtige Datenstruktur separat synchronisiert. Durch dieses sehr fein gewählte Granulat sollte erreicht werden, daß sich Prozesse möglichst wenig behindern, die auf einem Rechner gleichzeitig die Pufferverwaltungsalgorithmen durchlaufen. Dies hatte zur Folge, daß bei einer Seitenanforderung bis zu 10 verschiedene Synchro-

nisationssperren angefordert werden mußten.

Der dabei entstandene Synchronisationsaufwand war sehr hoch im Vergleich zu der dadurch gewonnenen Parallelität.

Daraufhin wurde das System umstrukturiert. Es wurden nur noch wenige Synchronisationssperren verwendet, die jeweils mehrere, häufig gemeinsam angeforderte Datenstrukturen schützen. Pro Seitenanforderung entstehen nun nur noch drei Zugriffe auf Synchronisationssperren. Durch diese Umstrukturierung konnte eine Laufzeithalbierung bei Anfragen erreicht werden, die keine Ein- und Ausgabe benötigen. Daraufhin stellte sich die Frage, ob durch eine nochmalige Reduzierung der Anforderung von Synchronisationssperren eine weitere Verringerung der Anfragelaufzeiten erreicht werden kann.

Dazu wurde in MIDAS testweise ein Mechanismus implementiert, wie er in Trans-Base vorhanden ist. Bei jedem Betreten der Pufferverwaltung wird der gesamte gemeinsame Speicher für alle anderen Prozesse global gesperrt. Dies ist eine sehr restriktive Maßnahme, die aber auch sehr billig durchzuführen ist.

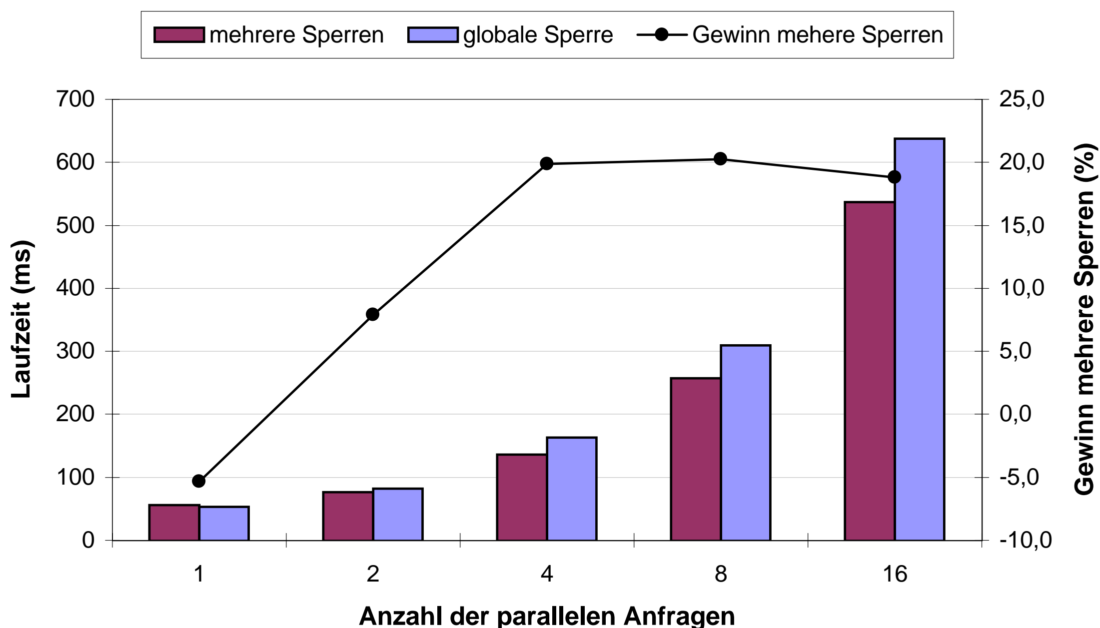

Abbildung 3.5: Gesperrte Segmentschicht

In Abbildung 3.5 sind die Ergebnisse einer vergleichenden Messung auf dem 4-Prozessorsystem dargestellt. Als Anfragen wurden Relationen-Scans auf Relationen verwendet, die sich komplett im Datenbankcache befanden.

Bei einer aktiven Anfrage besitzt die Variante mit global gesperrter Pufferverwaltung einen leichten Vorteil. Aber schon bei nur zwei parallelen Anfragen wird bei dieser Variante die Nebenläufigkeit zwischen den Anfragen derart eingeschränkt, daß die Variante mit mehreren Synchronisationssperren Vorteile besitzt. Bei vier parallelen Anfragen wird der maximale Laufzeitgewinn von 20 % durch die zusätzliche Parallelität im System erzielt.

Ein identischer Effekt tritt bei den Vergleichsmessungen zwischen TransBase und MIDAS in Abschnitt 4.1.2 auf. Je höher der Parallelitätsgrad, desto mehr gewinnt MIDAS gegenüber TransBase, das eine globale Sperrung der Pufferverwaltung benutzt.

Bei MIDAS ist somit ein sehr gutes Verhältnis zwischen billiger, restriktiver und teurer, viel Parallelität erlaubender Synchronisation gefunden worden.

3.3 Kommunikation – PVM

In MIDAS werden drei Formen der Inter-Prozeßkommunikation eingesetzt.

- **Nachrichten:**

  Der größte Teil der Inter-Prozeßkommunikation in MIDAS wird über Nachrichten abgewickelt, speziell jegliche Kommunikation über Rechnergrenzen hinweg. Alle Aufträge an andere Systemkomponenten und Antworten auf solche Aufträge werden in Form von Nachrichten ausgetauscht.

- **Signale:**

  Der Signal-Mechanismus des Betriebssystems wurde nur dort eingesetzt, wo Nachrichten nicht effizient verwendet werden konnten. In MIDAS ist dies immer an den Stellen der Fall, an denen ein Prozeß auf eine momentan nicht zur Verfügung stehende Ressource zugreifen will. In diesem Fall trägt sich der Prozeß in eine Warteliste für diese Ressource ein und legt sich schlafen. Der Prozeß, der die Ressource freigibt beziehungsweise zur Verfügung stellt, weckt dann einen der in der Warteliste schlafenden Prozesse über ein Signal. Dieser Mechanismus wird beispielsweise beim Zugriff auf Seiten im Datenbankcache verwendet.

  Wenn ein Prozeß auf eine Seite im Cache zugreifen will, diese aber nicht im Cache vorhanden ist, und ein anderer Prozeß diese Seite bereits von Festplatte liest oder sie von einem anderen Rechner anfordert, darf der Prozeß sich diese Seite nicht gleichzeitig beschaffen. Er trägt sich in die Warteliste für diese Seite ein und wird über ein Signal geweckt, sobald die Seite im lokalen Cache zur Verfügung steht.

- **Datenaustausch über gemeinsamen Speicher:**
  Inter-Prozeßkommunikation findet zusätzlich implizit über die Veränderung von Datenstrukturen im gemeinsamen Speicher (Shared-Memory) eines Rechners statt. Nur bei der Anforderung von Datenseiten von anderen Prozessen erfolgt ein Datenaustausch über gemeinsamen Speicher. Die Anforderung wird über eine Nachricht übermittelt, die Datenseite wird im Datenbankcache abgelegt und das Eintreffen der Seite wird wieder über eine Nachricht gemeldet.

Zum Versenden von Nachrichten wird in MIDAS ausschließlich die Message-Passing-Bibliothek PVM (Parallel Virtual Machine, Version 3.3.11) eingesetzt [GBDJMS 94a, GBDJMS 94b, BoLeLiPaRe 94]. Diese Bibliothek stellt eine Reihe von Funktionen zur Verfügung, die das Versenden von Nachrichten sehr einfach ermöglicht. PVM faßt dabei ein Netz von Workstations zu einem einzigen virtuellen Parallelrechner mit verteiltem Speicher zusammen. Jeder Prozeß, der über PVM gestartet wird, bekommt eine eindeutige Task-Id zugewiesen, über die er, analog zu den rechnerlokalen UNIX-Prozeß-Ids, identifiziert werden kann. Beim Versenden von Nachrichten an einen bestimmten Prozeß muß nur die entsprechende Task-Id angegeben werden, unabhängig davon, auf welchem Rechner der Prozeß tatsächlich läuft. Darüber hinaus stellt PVM eine dynamische Prozeßstruktur zur Verfügung, die es erlaubt, zur Laufzeit weitere Prozesse auf beliebigen Rechnern zu starten oder Prozesse terminieren zu lassen. Außerdem ist PVM effizient und ließ sich ohne hohen Aufwand in MIDAS integrieren [LisSchFri 94, Gouvedaris 96].

Der folgende Abschnitt beschäftigt sich mit den Kommunikationscharakteristika, die MIDAS aufweist, und zeigt, inwieweit die Leistungsfähigkeit des Systems durch PVM beeinflußt wird. Weiter wird verdeutlicht, wie PVM am besten eingesetzt wird und welche Nachteile die verwendete PVM-Version besitzt.

3.3.1 Kommunikationscharakteristika von MIDAS

Im folgenden wird das Nachrichtenaufkommen beschrieben, das in MIDAS bei der Ausführung von Anfragen entsteht.

Jeder Auftrag, den die Applikation an den ihr zugeordneten Applikationsserver stellt, wird in Form einer Nachricht versendet. Die Rückantwort, wie zum Beispiel Statusmeldungen oder berechnete Ergebnisse, ist ebenfalls wieder eine, gegebenfalls mehrere Nachrichten.

Ist dem Applikationsserver eine Anfrage als Auftrag gesendet worden, so wird diese in einen parallelen Ausführungsplan übersetzt. Die Teilpläne des parallelen Ausführungsplans werden wiederum mit Nachrichten auf verschiedene Interpreter des Ausführungssystems verteilt. Der Interpreter, der den Wurzelknoten des

Ausführungsplans ausgeführt hat, sendet die berechneten Ergebnisse an den Applikationsserver zurück. Die Anzahl der Nachrichten hängt dabei von der Anzahl der verwendeten Teilpläne sowie der Größe des Ergebnisses ab. Es entstehen dabei mindestens vier Nachrichten. Dies sind drei pro eingesetztem Interpreter: das Anfordern, das Schließen und die Übergabe des Teilplans an den Interpreter. Weiterhin werden die berechneten Tupel in einer oder mehreren Nachrichten, deren Größe momentan auf 32 KByte eingestellt ist, zum Applikationsserver zurückgesendet.

Die Interpreter greifen bei der Auswertung ihrer Teilpläne auf Datenbankseiten zu. Die benötigten Seiten sind im lokalen Cache vorhanden, befinden sich in einem entfernten Cache oder auf Festplatte. Um den Zugriff auf die Seite zu gestatten beziehungsweise die Seite in den lokalen Cache zu bekommen, werden die Protokolle der Sperrverwaltung und der Kohärenzkontrolle durchlaufen (siehe Abschnitte 3.6 und 2.4.7). Dazu müssen zwischen zwei und fünf kleine Kontrollnachrichten zwischen den Prozessen des Ausführungssystems verschickt werden. Der anfragende Interpreter bekommt als Resultat entweder die geforderte Seite in einer Nachricht zugesendet. Er erhält das Zugriffsrecht auf eine lokal schon vorhandene Seite oder er bekommt die Genehmigung, die Seite selbst von Festplatte zu laden. Diese Nachrichten sind zwischen ca. 10 Bytes und 200 Bytes groß. Die letzte Nachricht wächst auf 32 KByte oder 64 KByte, falls sie als Antwort eine komplette Seite beinhaltet.

Im folgenden wird das Nachrichtenaufkommen im System unter OLTP- und unter TPC-C-Last analysiert. Die Messungen wurden auf dem Mehrrechnersystem aus Anhang A durchgeführt. Es wurden drei Rechner verwendet, pro Rechner war eine Transaktion aktiv.

Als OLTP-Last wurden die TPC–A^{RO}-Transaktionen (siehe Anhang D) verwendet. Jede der Transaktionen beinhaltet drei Anfragen, die jeweils zwei Datenbankseiten berühren und ein Tupel lesen. Insgesamt wurden 3000 Transaktionen auf dem System ausgeführt. Tabelle 3.2 zeigt die ermittelten Nachrichtenanzahlen und Laufzeiten.

In dieser Konfiguration wurden 922 Nachrichten pro Sekunde zwischen 21 Prozessen des Ausführungssystems auf den drei Rechnern ausgetauscht. Etwa 2,7 Nachrichten pro Sekunde waren große Seitennachrichten, die eine Seite beinhalteten. Wenn zu einem Zeitpunkt nur eine Transaktion im System aktiv ist, sinkt das Nachrichtenaufkommen zwischen 10 Prozessen auf 414 Nachrichten pro Sekunde.

Als zweite Last wurden die TPC–C^{S+O}-Transaktionen verwendet, eine datenintensivere TPC-C-Last (siehe Anhang D). Es wurden 210 Transaktionen auf dem System zur Ausführung gebracht. Jede der Transaktionen greift auf ca. 300 Seiten zu.

	OLTP-Last		TPC-C-Last	
	1 TA	3 TAs	1 TA	3 TAs
Prozesse	10	21	10	23
Nachrichten gesamt	246162	230450	45498	46929
Seiten-Nachrichten	483	682	1296	4382
Laufzeit (s)	594	250	342	360
Nachrichten/Sekunde	414	922	133	130
Seiten-Nachrichten/Sekunde	0,8	2,7	3,8	12,2

Tabelle 3.2: Nachrichtendurchsatz in MIDAS

Wie in Tabelle 3.2 zu sehen, erzeugt diese Last mit 130 Nachrichten pro Sekunde einen deutlich niedrigeren Nachrichtendurchsatz. Dagegen ist der Anteil großer Nachrichten mit 12,2 Nachrichten pro Sekunde deutlich höher als unter OLTP-Last.

Die oben gezeigten Beispiele mit ihrer hohen Anzahl an Nachrichten und dem hohen Nachrichtendurchsatz machen deutlich, wie wichtig effiziente Kommunikation für MIDAS ist.

Die in MIDAS auftretenden Kommunikationscharakteristika sind hauptsächlich von der Anzahl parallel aktiver Transaktionen, der Anzahl an Rechnern im System und der verwendeten Last abhängig. Allgemein tritt eine sehr hohe Anzahl kleiner Kontrollnachrichten im System auf, die durch die Protokolle von Sperrverwaltung und Kohärenzkontrolle bedingt sind. Der Nachrichtendurchsatz ist konstant hoch und bei gleichbleibender Last ohne größere Schwankungen.

3.3.2 PVM-spezifische Optimierungen

Dieser Abschnitt analysiert die Verwendung einiger Parameter der PVM-Bibliothek und beschreibt, wie durch geeignete Konfiguration die Kommunikationskosten gesenkt werden können.

Eine der Einstellungen, die bei PVM getroffen werden können, ist die Wahl der Verbindungsart. PVM bietet hier zwei Alternativen. Im Standardfall werden alle über PVM versendeten Nachrichten zuerst zu einem lokalen Dämon gesendet. Vor dort aus wird die Nachricht an den Dämon eines anderen Rechners weitergeleitet, falls der Empfänger sich nicht auf demselben Rechner befindet. Von diesem Dämon wird die Nachricht an den eigentlichen Empfänger übermittelt. Diese Kommunikationsvariante unterliegt keinen Einschränkungen, sie bedarf aber

dreier einzelner Nachrichten, um den Empfänger zu erreichen. Alternativ kann die Verbindungsart auf den Wert *RouteDirect* eingestellt werden. Wird diese Einstellung getroffen, so werden direkte Kommunikationsverbindungen zwischen den Kommunikationspartnern erstellt, ohne dabei den Umweg über die Dämonen zu gehen. Diese effizientere Variante ist allerdings beschränkt, da durch limitierte Betriebssystemsressourcen nur eine begrenzte Zahl solcher Verbindungen pro Prozeß geöffnet gehalten werden kann. Bei einem System, das auf viele Rechner skalieren soll, können dann nicht mehr zwischen allen Prozessen der Pufferverwaltung direkte Verbindungen aufgebaut werden.

	OLTP-Last		Kommunikations-
	1 TA	3 TAs	segment-Last
Prozesse	10	21	7
Nachrichten gesamt	246162	230450	28420
Seiten-Nachrichten	483	682	5154
Laufzeit (s)			
RouteDirect & *DataRaw*	594	250	195
DataRaw	666	280	305
DataDefault	684	282	305

Tabelle 3.3: PVM Parameter *RouteDirect*, *DataDefault* und *DataRaw*

In Tabelle 3.3 wird der Leistungsgewinn quantifiziert, der mit direkten Verbindungen zu erzielen ist. Als Abfragen wurden TPC–A^{RO}-Transaktionen als OLTP-Last verwendet, und zusätzlich kam eine besonders kommunikationsintensive Last, bei der zwei Interpreter starken Datentransfer über ein Kommunikationssegment durchführen (siehe Abschnitt 3.4), zum Einsatz.

Zusätzlich zur Verbindungsart wurde auch noch das Verpacken der Nachrichten vor dem Versenden variiert. PVM bietet hier die Einstellungen *DataDefault* und *DataRaw* an. Bei der Einstellung *DataDefault* werden die Daten vor dem Versenden in ein architekturunabhängiges Format gebracht, so daß sie auch auf Rechnern unterschiedlicher Architektur empfangen werden können. Die Einstellung *DataRaw* veranlaßt das direkte Versenden der Nachrichten, ohne Konvertierungen vorzunehmen. Diese Einstellung erlaubt nur Kommunikation zwischen Rechnern derselben Architektur [GBDJMS 94a].

Wie aus Tabelle 3.3 zu entnehmen ist, verschlechtert sich die Laufzeit der Anfra-

gen zwischen 12 % (250:280) und 56 % (195:305), wenn keine direkten Verbindungen verwendet werden. Dagegen ist der Unterschied der Varianten des Verpackens (666:684) der Nachrichten von geringer Bedeutung[4].

3.3.3 PVM auf Mehrprozessorsystemen

PVM bietet eine spezielle Unterstützung für Mehrprozessormaschinen an. Dazu existiert eine PVM-Laufzeitbibliothek, die zu den Prozessen hinzugebunden werden kann. Diese Bibliothek steuert die rechnerlokale Kommunikation auf Mehrprozessormaschinen über Shared-Memory, wodurch Nachrichten eingespart werden.

Bei der Verwendung der Shared-Memory-Bibliothek ist eine Leistungssteigerung des Systems durch geringere Kommunikationskosten zu erwarten. Um dies zu bestätigen, wurden Messungen auf dem 4-Prozessorsystem (siehe Anhang A) mit der Standardbibliothek und der Shared-Memory-Bibliothek durchgeführt. Dabei entsteht ausschließlich lokale Rechnerkommunikation.

Bibliothek	Laufzeit	TPC–A^{RO}	TPC–C^{S+O}	Komm.-Segment
Standard	(s)	13	17	235
Shared-Memory	(s)	20	28	351
Verschlechterung		+ 54 %	+ 65 %	+ 49 %

Tabelle 3.4: Laufzeiten mit verschiedenen PVM-Bibliotheken

Tabelle 3.4 zeigt die Ergebnisse, die unter TPC–A^{RO}- und TPC–C^{S+O}-Last sowie bei reinem Seitenaustausch über ein Kommunikationssegment erzielt wurden[5].

Der erwartete Vorteil der Shared-Memory-Bibliothek konnte nicht bestätigt werden. Die Laufzeiten waren zwischen 49 % und 65 % schlechter als bei Verwendung der Standardbibliothek.

Die genauen Gründe für dieses Verhalten konnten nicht bestimmt werden. Sie liegen aber wahrscheinlich in der Implementierung der PVM-Shared-Memory-Bibliothek.

[4]Da *DataRaw* einen leichten Vorteil gegenüber *DataDefault* aufweist, mußte die Kombination *DataDefault & RouteDirect* nicht getestet werden.

[5]Vergleiche Lasten aus Abschnitt 3.3.1 und Abschnitt 3.3.2

3.3.4 Der Einsatz von PVM

Im Laufe der Implementierung erwies sich die Wahl von PVM als Kommunikationsbibliothek als sehr geeignet.

Als die Implementierung von MIDAS begann, standen außer PVM noch die Programmiersysteme ParMod-C und MMK/X als Alternativen zur Verfügung.

PVM war zu dieser Zeit die beste Wahl [LisSchFri 94], da es sich als Kommunikationsstandard entwickelte, heterogene Architekturen unterstützt und sehr einfach zu integrieren ist. Dies konnte im Laufe der Implementierung bestätigt werden. Ein zweimaliger Wechsel des Betriebssystems wurde problemlos bewältigt[6]. Der Aufwand der Programmierung der Kommunikationsroutinen nahm insgesamt gesehen einen sehr geringen Teil der Arbeiten an MIDAS ein.

Die Entscheidung für PVM wurde nachträglich dadurch bestätigt, daß ParMod-C nicht entscheidend weiterentwickelt wurde und die MMK/X-Architektur komplett verschwunden ist.

Die Hauptnachteile von PVM sind, daß es, wie in Abschnitt 3.2.2 beschrieben, keine Threads unterstützt. Zudem weist PVM einen gewissen Leistungsnachteil auf, beispielsweise gegenüber einer direkten Implementierung der Kommunikation über Sockets.

Eine alternative Implementierung der Kommunikationsroutinen direkt über Sockets würde sich anbieten, um den Overhead auszuschalten, der in der PVM-Implementierung vorhanden ist. Realistisch erscheinen hier 50 % höherer Durchsatz und 50 % kürzere Latenzzeiten [Gouvedaris 96]. Eine Socket-Implementierung ist allerdings als wesentlich aufwendiger einzuschätzen. Zudem ginge dabei die Portierbarkeit des Systems verloren. Die potentiellen Nachteile von PVM gegenüber einer Socket-Implementierung können aber in Kauf genommen werden, da MIDAS nicht als kommerzielles System, sondern als Prototyp zur Evaluierung neuer Techniken entworfen wurde.

Müßte die Entscheidung über die zu verwendende Kommunikationsbibliothek nochmals getroffen werden, so wäre heute unter Umständen aus Sicht der Leistungsfähigkeit des Systems und unter Vernachlässigung heterogener Systeme MPI die bessere Alternative [GeiKohPap 96]. MPI stand zum Zeitpunkt der Wahl des Kommunikationsmodells allerdings noch nicht zur Verfügung.

Zur Leistungssteigerung bietet sich in einem kommunikationsintensiven System wie MIDAS neben billigen Kommunikationsroutinen hauptsächlich das komplette Einsparen möglichst vieler, teurer Nachrichten an. Derartige Ansätze werden in den Abschnitten über die Architekturumstellung (Abschnitt 3.2.1), über Kommunikationssegmente (Abschnitt 3.4), die Verwendung von Threads (Ab-

[6] Portierungen fanden von SunOS auf Solaris und Solaris auf Linux statt.

schnitt 3.2.2) und über lokalitätsbasiertes Transaktionsrouting (Abschnitt 4.4.4) beschrieben.

3.4 Kosten der Kommunikationssegmente

Zur Einführung von Intra-Transaktionsparallelität in MIDAS mußte ein Kommunikationsmechanismus zwischen den Interpretern zum Austausch von Zwischenergebnissen geschaffen werden. Dieser sollte so realisiert werden, daß keine Änderungen oder Erweiterungen der bestehenden Operatoren vorgenommen werden mußten. Durch Einführung des SEND- und des RECEIVE-Operators konnte dies erreicht werden [Brandmayer 97].

Der Datenaustausch zwischen diesen Operatoren wird durch sogenannte Kommunikationssegmente über die Pufferverwaltungsschicht durchgeführt (vergleiche Abschnitt 2.4.9). Dieser Abschnitt stellt eine Bewertung der dabei entstehenden Kosten vor.

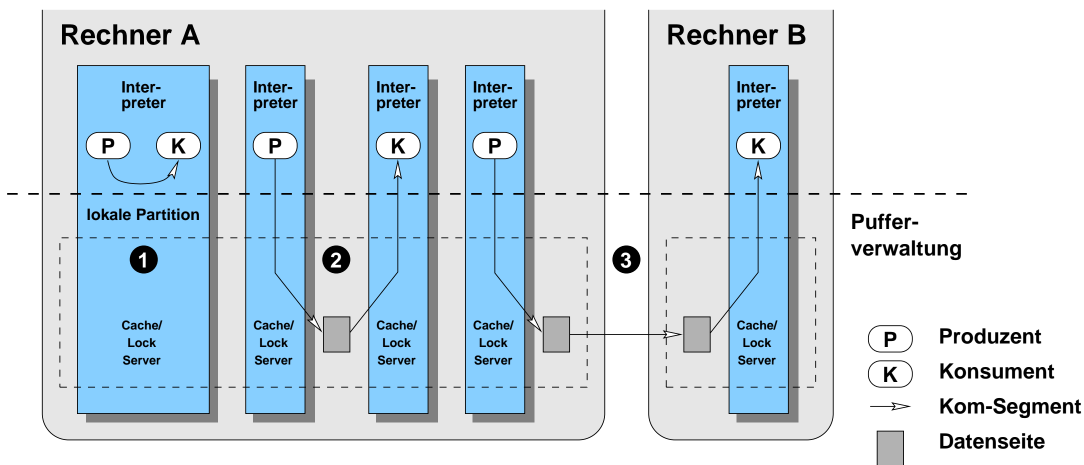

Abbildung 3.6: Varianten der Kommunikationssegmente

Die Kommunikationssegmente verfügen über drei Varianten des Datentransfers. Diese sind in Abbildung 3.6 dargestellt. Bei jeder der drei Varianten werden Daten über ein Kommunikationssegment von einem Produzenten P, der einen SEND-Operator an der Wurzel seines Teilplans besitzt, zu einem Konsumenten K übertragen, der einen RECEIVE-Operator in einem seiner Blattknoten besitzt.

1. **Produzent und Konsument werden im selben Interpreter ausgeführt:**
   Bei dieser Variante werden die Daten Tupel für Tupel direkt zwischen den Teilplänen von Produzent und Konsument übertragen. Diese Variante wird *lokale Partition* genannt und ist identisch mit der Übertragung der Tupel innerhalb eines Operatorbaumes. Bei ihr entstehen die geringsten Kosten, da die Kommunikation innerhalb eines Prozesses stattfindet. Parallelität zwischen Konsument und Produzent ist nicht möglich.

2. **Produzent und Konsument befinden sich auf verschiedenen Interpreter auf demselben Rechner:**
   Bei dieser Variante werden die Daten von den Interpreter-Modulen über normale Aufrufe der Segmentschnittstelle an die Pufferverwaltung übergeben und von dieser gelesen. Die Daten werden lokal im Cache abgelegt und werden subseiten-, seiten- oder segmentweise für den Konsumenten zum Lesen freigegeben. Die Daten müssen hier nicht über Nachrichten verschickt werden.

3. **Produzent und Konsument befinden sich auf verschiedenen Rechnern:**
   Diese Variante arbeitet wie die vorherige. Zusätzlich müssen die Daten aber noch von der Pufferverwaltung über Nachrichten zwischen den Rechnern transferiert werden. Die Daten werden wieder subseiten-, seiten- oder segmentweise übertragen, da kleinere Nachrichten zu einer wesentlich höheren Gesamtzahl an Nachrichten führen würden, was zu teuer wäre.

Variante 3 ist die allgemeinste Variante. Mit ihr können Daten unabhängig von der Lage der Interpreter transferiert werden. Die Varianten 1 und 2 wurden entworfen, um die bei der Variante 3 zu erwartenden hohen Kommunikationskosten zu reduzieren.

Die im folgenden präsentierten Messungen quantifizieren die Kosten der verschiedenen Varianten. Zusätzlich wird noch die eingesetzte Art der Verdrängungsstrategie[7] mit einbezogen.

Die Messungen fanden auf einem beziehungsweise zwei Rechnern des Mehrrechnersystems (siehe Anhang A) statt. Die verwendete Anfrage bestand ausschließlich aus dem Lesen einer 120 MByte großen Relation von Festplatte und anschließender Übertragung durch ein Kommunikationssegment. Tabelle 3.5 zeigt die ermittelten Werte.

Wie erwartet stellte sich die lokale Partition als billigste Kommunikationsvariante heraus. Die Kosten werden im wesentlichen durch das einmalige Lesen der

[7] Wait, WriteOut und NoBuf, siehe Abschnitt 2.4.9.

	Variante	Tupeltransfer (sec)	WAIT (sec)	WRITEOUT (sec)	NOBUF (sec)
1	lokale Partition	50	—	—	—
2	2 Interpreter lokal	—	89	262	112
3	2 Interpreter entfernt	—	130	270	315

Tabelle 3.5: Kosten der Kommunikationssegmente

Daten von Festplatte verursacht. Werden zwei Interpreter eingesetzt, so ist im Fall, daß kein Verdrängen auf Festplatte stattfindet (WAIT), der lokale Datenaustausch (Variante 2) deutlich effizienter (89:130) als der Datenaustausch über die Rechnergrenze (Variante 3), da dort viele Nachrichten anfallen. Wird dagegen auf Festplatte verdrängt (WRITEOUT), so fallen die Kosten für Nachrichten kaum mehr ins Gewicht (262:270). Werden die Seiten der Kommunikationssegmente in die normale Verdrängungsstrategie mit einbezogen (NOBUF), so wird in Variante 2 ein Wert (112) erreicht, der nahe an dem unter WAIT erzielten Wert (89) liegt. Dies ist darauf zurückzuführen, daß hier der gesamte lokale Cache anstelle des sonstigen kleinen Speicherkontingents verwendet wird. In Variante 3 sinkt unter NOBUF die Leistung nochmals ab (315), da durch die LRU-Strategie häufig die Seiten verdrängt werden, die als nächste vom Konsumenten gelesen werden[8].

Die Messungen zeigen, wie teuer Kommunikation im System ist und daß die Reduzierung von Nachrichten eine der wichtigsten Optimierungsmöglichkeiten im System darstellt (siehe auch Abschnitte 3.3.4, 3.2.1 und 4.16). Die Effizienz der implementierten speziellen Varianten, die die Kommunikation reduzieren, wurde belegt. Diese Überlegungen können bei der Parallelisierung in die Kostenberechnung mit einbezogen werden [Nippl 00], um zu effizienteren parallelen Ausführungsplänen zu gelangen.

3.5 Ein- und Ausgabe

Effizientes E/A-Verhalten ist in einem parallelen Datenbanksystem von besonderer Bedeutung. Dieser Abschnitt stellt das in MIDAS verwendete Datenverteilungsverfahren vor und analysiert Probleme, die aus der nicht vorhandenen Kohärenz bei NFS sowie den Wechselwirkungen des Datenbanksystems mit dem Betriebssystemcache entstehen.

[8]Das sind die Seiten, die der Produzent als erstes geschrieben hatte.

3.5.1 Physische Datenverteilung

MIDAS hatte zunächst die E/A-Komponente von TransBase übernommen. Jedes Segment der Cacheverwaltung wird dort auf eine Datei im Dateisystem der lokalen Festplatte abgebildet. Für jede Datenbank sind alle diese Dateien auf einer einzelnen Festplatte gespeichert.

E/A-Anweisungen gehören zu den teuersten Operationen eines Datenbanksystems. Speziell in einem parallelen Datenbanksystem wie MIDAS, in dem Prozesse nebeneinander ablaufen und dabei gleichzeitig E/A-Aufträge absetzen können, stellt sich schnell heraus, daß die Speicherung der Daten auf nur *einer* Festplatte, ein extremer Leistungsengpaß sein kann.

Zudem ist MIDAS als Shared-Disk-Datenbanksystem konzipiert, d. h. es wird davon ausgegangen, daß alle beteiligten Rechner gleiche Zugriffsmöglichkeiten auf externe Daten haben. In "echten" Shared-Disk-Datenbanksystemen wird dies über eine direkte Anbindung eines RAID-Plattensystems an alle Rechnerknoten gewährleistet. Bei der für MIDAS zur Verfügung stehenden Hardware-Ausstattung ist das nicht möglich. Dort werden alle Festplatten als lokale Festplatten an den einzelnen Rechnern betrieben. Der Zugriff auf die Festplatten anderer Rechner erfolgt über NFS. Wird MIDAS in einer Mehrrechnerkonfiguration betrieben, hat die Benutzung einer einzelnen Festplatte den Nachteil, daß der Rechner, auf dessen Festplatte die Datenbank gespeichert ist, nur billige, lokale Festplattenzugriffe tätigen muß, wohingegen alle anderen Rechner die Daten nur durch teure NFS-Zugriffe über das Netzwerk bekommen.

Ausgehend von dieser Überlegung wurde in MIDAS eine Verteilung der physischen Datenbanksegmente auf beliebig viele Festplatten ermöglicht. Dabei bleibt die Verteilung der Daten für alle Schichten des Systems oberhalb der E/A-Komponente transparent.

Im folgenden wird der verwendete Algorithmus für die Verteilung der Daten beschrieben. Es werden Messungen vorgestellt, welche die Vorteile dieser Aufteilung dokumentieren.

3.5.1.1 Aufteilung der Segmente – Partitionierung

In MIDAS werden Segmente auf mehrere Dateien abgebildet, nicht mehr wie in TransBase auf eine einzelne. Jede dieser Dateien wird in einem anderen Verzeichnis abgelegt. Die Verzeichnisse können sich auf verschiedenen lokalen oder entfernten Festplatten befinden, so daß eine Aufteilung auf mehrere Festplatten erzeugt werden kann. Insofern wird im folgenden immer die Rede von einer Aufteilung auf mehrere Verzeichnisse sein. Natürlich kann auch eine Aufteilung auf mehrere Verzeichnisse einer Festplatte erfolgen. Dabei sind allerdings keine Verbesserungen zu erwarten.

Bei der Implementierung dieser Aufteilung wurde im wesentlichen darauf Wert gelegt, daß von allen Rechnern möglichst gleiche Zugriffsbedingungen auf die Daten bestehen, um einem Shared-Disk-Datenbanksystem möglichst nahezukommen. Die Verteilung der Daten sollte also transparent für die oberen Schichten des Systems vorgenommen werden. Die Untersuchung spezieller datenabhängiger Verteilungen wie Partitionierungen, die von den Algorithmen in der Cacheverwaltung oder in der Anfrageverarbeitung benutzt werden können, um möglichst viele lokale Zugriffe zu generieren, sollten hier nicht untersucht werden. Zudem war eine möglichst einfache Implementierung erwünscht.

Die Einheit, die bei einem Aufruf der E/A-Komponente von der Festplatte gelesen oder geschrieben wird, ist eine MIDAS-Seite (siehe auch Abschnitt 3.1). Diese Einheit bot sich als Granulat der Verteilung an, da dies die kleinste Einheit ist, in die Segmente aufgeteilt werden können, ohne massive Änderungen an den E/A-Funktionen vorzunehmen. Bei den verwendeten Datenbanken handelt es sich meist um Blöcke von 32 oder 64 KByte, je nach Anzahl der Subseiten.

Die Aufteilung der Seiten eines Segments erfolgt zyklisch (*round robin*) über die verschiedenen Dateien. Dabei wird mit der ersten Seite jedes Segments ebenfalls zyklisch begonnen, d. h. die ersten Seiten verschiedener Segmente werden der Reihe nach auf verschiedene Verzeichnisse verteilt, um keine Ungleichverteilung zu erzeugen. Wie in Abbildung 3.7 dargestellt, entstehen bei der Verteilung eines Segments auf N Verzeichnisse aus einer Datei (= 1 Segment) N Dateien, sofern das Segment mindestens N MIDAS-Seiten hat. Bei kleineren Segmenten wird nach deren vollständiger Verteilung aufgehört, d. h. es werden keine leeren Dateien angelegt.

Der Zugriff auf eine Seite erfolgt wie bisher über die Angabe der Segmentnummer und der MIDAS-Seitennummer innerhalb des Segments. Zusammen mit der festen Anzahl der Verzeichnisse lassen sich die neue Datei und die Seitennummer in dieser Datei leicht bestimmen:

Das Verzeichnis wird aus der Summe von Seiten- und Segmentnummer modulo der Anzahl der Verzeichnisse ermittelt.

$i = (Seitennummer + Segmentnummer) \bmod AnzahlVerzeichnisse$

Dabei gibt der Wert i an, daß sich die Datei mit der gewünschten Seite im i-ten Verzeichnis befindet. Die Angabe der Segmentnummer in dieser Formel ist notwendig, da über sie sichergestellt wird, daß die ersten Seiten der Segmente zyklisch auf die vorhandenen Verzeichnisse verteilt werden (Round-Robin-Verteilung).

Die Seitennummer innerhalb des neuen Segments wird aus der alten Seitennummer dividiert durch die Anzahl der Verzeichnisse ermittelt:

$SeitennummerInDatei = Seitennummer \operatorname{div} AnzahlVerzeichnisse$

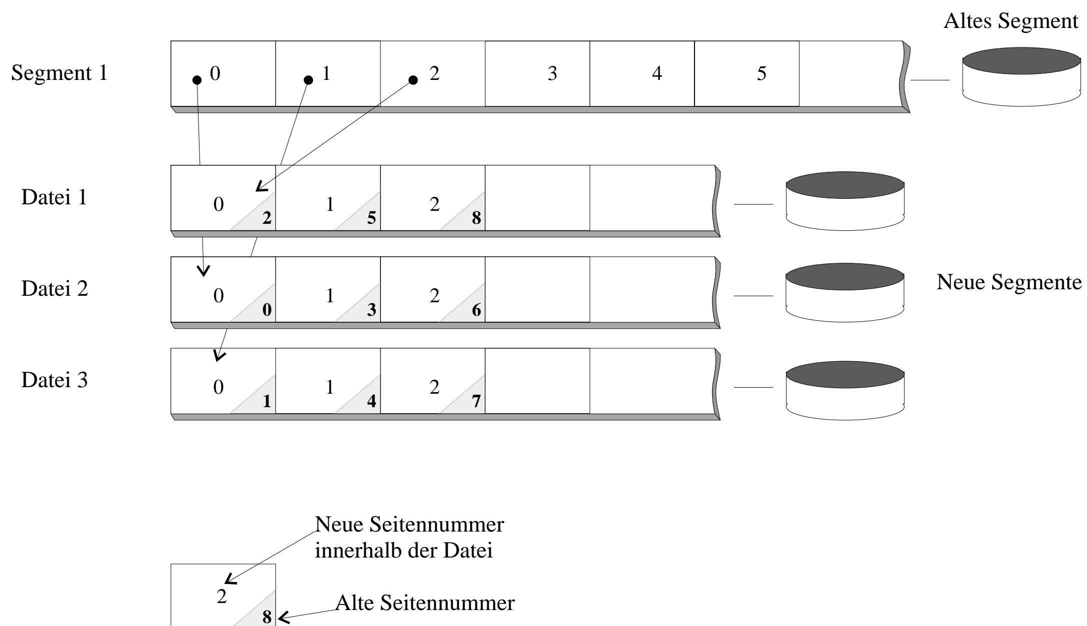

Abbildung 3.7: Verteilung eines Segments auf mehrere Verzeichnisse

Mit diesen Werten wird das Segment angesprochen und die Seite aus der entsprechenden Datei geladen.

3.5.1.2 Leistungsanalyse

Die Messungen wurden auf dem Mehrprozessorsystem durchgeführt (siehe Anhang A). Verwendet wurden die TPC–A- (20 tps, 210 MByte) und die TPC–C-Datenbanken (3 Warehouses, 320 MByte) mit Subseiten von 8 KByte (siehe Anhang C). MIDAS-Seiten waren in 4 Subseiten unterteilt. Da bei diesem Versuch Plattenzugriffe gemessen werden sollten, wurde bei beiden Datenbanken ein recht geringer Cache von 64 MIDAS-Seiten (2 MByte) eingestellt.

Vor Beginn der Messungen wurde mit einem Hilfsprogramm die maximale Transferrate einer Festplatte ermittelt. Beim sequentiellen Lesen beträgt sie ca. 5 MByte/s, bei Random-Zugriff zum Lesen von 64 KByte-Blöcken, wie es etwa in MIDAS geschieht, ca. 1,6 MByte/s.

Alle Messungen wurden mit vier parallelen Applikationen durchgeführt. Variiert wurde die Anzahl der Festplatten und die verwendete Transaktion.

Folgende Ergebnisse wurden bestimmt:

Gemessen wurde jeweils mit 4 parallelen Applikationen und mit auf 1, 2 und 4 Festplatten aufgeteilten Datenbanken. Als Transaktionen wurden die TPC–A^{RO}-

Transaktion sowie die TPC–C^{S+O}-Transaktion verwendet (siehe Anhang D).

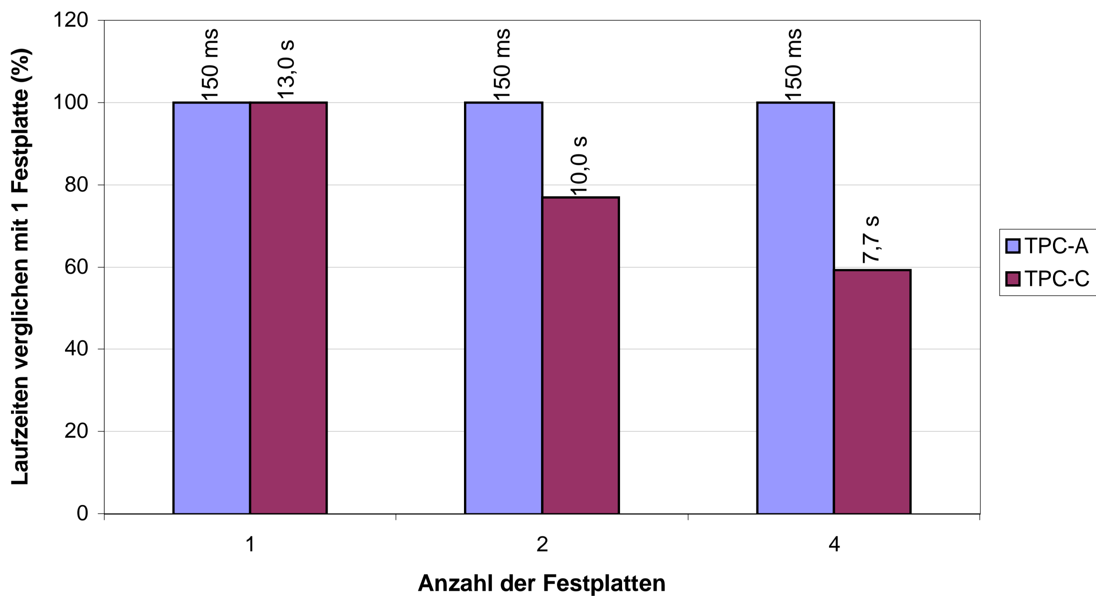

Abbildung 3.8: Laufzeitveränderungen bei Verteilung auf mehrere Festplatten

In Abbildung 3.8 sind die prozentualen Beschleunigungen einer Transaktion bei der Verteilung der Segmente auf mehrere Festplatten dargestellt. Die prozentuale Darstellung wurde gewählt, um beide Transaktionen einander in einem Diagramm gegenüberstellen zu können. Die mit einer Festplatte erreichten Zeiten wurden als 100 %-Marke festgesetzt, die anderen entsprechend umgerechnet.

Deutlich zu sehen sind die bei beiden Transaktionen unterschiedlichen Effekte. TPC–A^{RO}, eine sehr kleine Transaktion, die pro Aufruf eine, selten auch zwei Seiten von der Festplatte liest, kann überhaupt nicht von der Umstellung profitieren. Da die Anforderungen sehr klein sind, treten offensichtlich auch bei nur einer Festplatte keine Wartezeiten auf. Somit kann von der Verteilung nicht profitiert werden. Das deckt sich mit Beobachtungen, die dem Datenvolumen galten, das bei Testmessungen von der Festplatte gelesenen wurde. Während bei einer Festplatte Durchsatzraten von ca. 600 KByte/s beobachtet wurden, sank dieser Wert bei der Verwendung von vier Festplatten auf ca. 180 KByte pro Sekunde und Festplatte. Beides sind Durchsatzraten, die nicht an die Leistungsgrenzen der Festplatten stoßen. Hier wird die Ausführungsgeschwindigkeit einer Transaktion also offensichtlich von der Leistung und Auslastung der CPU und den Grundwerten der Festplatte (Positionierungszeit, Rotationslatenzzeit) bestimmt.

Bei den Transaktionen von TPC–C^{S+O}dagegen ist der Ausführungszeitgewinn

deutlich. Die Verbesserung um 40 % gegenüber einer Festplatte ist auf die großen Datenmengen zurückzuführen, die bei diesen Transaktionen gelesen werden. Hier wird im sequentiellen Fall die Festplatte über ihre Grenzen hinaus gefordert, bei Testmessungen wurde ein Datendurchsatz von nur 2450 KByte/s gemessen. Dabei war die Festplatte, verursacht durch die konkurrierenden Zugriffe auf unterschiedliche Sektoren, offenbar an ihrer Leistungsgrenze angelangt. Vergleichsmessungen mit einem Hilfsprogramm haben gezeigt, daß die maximale Transferrate bei hundertprozentigem Random-Zugriff bei diesen Festplatten bei 1,6 MByte/s liegt. Bei Messungen mit nur einer Applikation konnten Transferraten über 4 MByte/s beobachtet werden, die durch sequentielles Lesen mit Relationen-Scans erreicht wurden. Bei der Verwendung von vier Festplatten lagen die Transferraten immerhin noch bei 780 KByte pro Sekunde und Festplatte, das sind ca. 3,2 MByte/s insgesamt. In Tabelle 3.6 sind diese Werte noch einmal aufgelistet.

Transaktion	**Festplatten**	**Zeit/TA**	**Lesedurchsatz 1 Festplatte**
TPC–A^{RO}	1	150 ms	600 KB/s
	2	150 ms	310 KB/s
	4	150 ms	180 KB/s
TPC–C^{S+O}	1	13.0 s	2.468 KB/s
	2	10.0 s	1.335 KB/s
	4	7.7 s	780 KB/s

Tabelle 3.6: Durchsatz bei Verteilung der Segmente

Die TPC–C^{S+O}-Transaktion hat sich als festplatten-beschränkt herausgestellt. Sie konnte damit deutlich von der verbesserten Festplattenkonfiguration profitieren.

3.5.1.3 Replikation

Zusätzlich zur Partitionierung der Segmente können in MIDAS die Segmente auch repliziert in mehreren Verzeichnissen auf verschiedenen Festplatten gehalten werden. Dabei hat in einer Mehrrechnerkonfiguration jeder Rechner die komplette Datenbank auf seiner lokalen Festplatte zur Verfügung.

Diese Variante der Datenhaltung ist nur für lesende Transaktionen geeignet, da die Daten nicht konsistent gehalten werden. Sie wurde nur zu Testzwecken benutzt, um eine Rechnerkonfiguration zu ermöglichen, in der alle Rechner auf alle

Daten gleichen und schnellen Zugriff besitzen.

3.5.1.4 Zusammenfassung

Die Aufteilung der Segmente kann insgesamt positiv bewertet werden. Bei kleinen Transaktionen ergibt sich zwar kein Vorteil, es können jedoch auch keine negativen Auswirkungen gemessen werden. Bei Transaktionen, die große Datenmengen lesen, ist die Geschwindigkeitssteigerung jedoch erheblich. Hier werden bis zu 40 % bessere Resultate erzielt. Diese Ergebnisse begründen sich in den Leistungsgrenzen der Festplatten, die erreicht werden, wenn große Transaktionen parallel zugreifen. Durch die Aufteilung auf mehrere Festplatten wird ein Engpaß erweitert und ein insgesamt höherer Durchsatz erzielt.

Die Messungen wurden im Mehrbenutzerbetrieb durchgeführt. Es gibt aber auch spezielle Situationen, in denen eine Verteilung auf mehrere Festplatten ungünstig ist. So wurde im Einbenutzerbetrieb mit Transaktionen, die nur einen sequentiellen Scan auf eine große Relation durchführten, eine Verschlechterung um 5 % in der Transaktionslaufzeit festgestellt. Das ist möglicherweise auf die komplizierten Wechselwirkungen zwischen Betriebssystem- und Festplattencache zurückzuführen, die das sequentielle Lesen einer Datei von *einer* Festplatte offenbar besser unterstützen als das sequentielle Lesen mehrerer Dateien auf mehreren Festplatten.

Dagegen wird durch die Verteilung der Segmente ein Ungleichgewicht in der Rechnerkonfiguration bei nur einer verwendeten Festplatte ausgeglichen.

Wenn in einer Konfiguration N Rechner verwendet werden und die Datenbank auf die lokalen Festplatten der Rechner verteilt ist, so haben alle Rechner dieselbe Zugriffscharakteristik auf die permanenten Daten. $\frac{1}{N}$ der Zugriffe erfolgen lokal, die restlichen greifen auf die Festplatten anderer Rechner zu. Diese Verteilung der Datenbank wird im folgenden als **lokale Partitionierung** oder **lokale Verteilung** bezeichnet.

Alternativ können die Daten auf Festplatten verteilt werden, auf denen das Datenbanksystem nicht läuft, womit zwar ein schlechterer Durchsatz, dafür aber ein vollkommen gleichförmiger Zugriff auf die Daten erreicht wird. Diese Verteilung der Datenbank wird im folgenden als **entfernte Partitionierung** oder **entfernte Verteilung** bezeichnet.

Einen ebenfalls vollkommen gleichförmigen Zugriff erhält man mit der Replikation der physischen Daten auf alle lokalen Festplatten des Systems. Diese Art der Verteilung wird im folgenden als **lokale Replikation** bezeichnet. Damit kommt MIDAS den Charakteristika eines Shared-Disk-Datenbanksystems wesentlich näher, bei denen kein Rechner bevorzugten Zugriff auf bestimmte Daten hat.

Messungen zu diesen Rechnerkonfigurationen finden sich in Abschnitt 4.3.1. Auswirkungen, die aus der lokalen oder entfernten Lage der Daten entstehen, werden in Abschnitt 3.5.2 besprochen.

3.5.2 Cache- und NFS-Problematik

In einem "echten" Shared-Disk-System haben alle Rechner zum einen gleichförmigen Zugriff auf alle Daten (siehe auch Abschnitt 3.5.1). Zum anderen sollten lokale Cachezugriffe effizienter als entfernte Cachezugriffe und diese günstiger als Festplattenzugriffe sein. Dies ist in MIDAS nicht immer der Fall.

Wird MIDAS in der in Anhang A beschriebenen Mehrrechnerkonfiguration betrieben, können unerwartete Zeiteffekte beim Laden einer MIDAS-Seite auftreten. Im Gegensatz zu den Anforderungen an ein "echtes" Shared-Disk-System kann es passieren, daß das Laden einer Seite von lokaler Festplatte oder über NFS billiger ist als eine Seite aus einem entfernten Datenbankcache zu transferieren. Die Architektur von MIDAS und alle implementieren Algorithmen entsprechen denen eines Shared-Disk-Systems. Die einzige Abweichung von den Charakteristika eines Shared-Disk-Systems ist das Auftreten dieses Ungleichgewichtes bei den Datenzugriffen.

Das Ungleichgewicht ergab sich aus der unterschiedlichen, teilweise konkurrierenden Funktionsweise von Datenbanksystem und Betriebssystem[9].

Betrachtet wird folgende Situation: Auf einem Rechner des Mehrrechnersystems entsteht eine Seitenanforderung an die Cacheverwaltung. Die Cacheverwaltung stellt fest, daß sich diese Seite nicht im rechnerlokalen Datenbankcache befindet. Es gibt folgende drei Möglichkeiten, die angeforderte Seite in den lokalen Datenbankcache zu laden:

- **Die Seite befindet sich im Datenbankcache eines anderen Rechners:**

  In diesem Fall wird dem anfragenden Rechner die geforderte Seite mittels PVM-Nachricht zugesandt. Dies dauert bei der vorgegebenen Hardwareausstattung im günstigsten Fall ca. 12 ms. Dieser Wert steigt bei hoher Netzbelastung aber schnell auf bis zu 100 ms an. Hohe Netzbelastung kann dabei durch viel E/A über NFS und viele Nachrichten der Cacheverwaltung bei einer hohen Zahl an parallelen Anfragen entstehen. Der Wert von 100 ms ergab sich bei dem 4-Rechnersystem bei 8 parallelen Transaktionen unter OLTP-Last.

[9] Auf einem Mehrprozessorsystem treten diese Effekte nicht auf. Eine Betrachtung des Datenbankcaches auf einem solchen System findet sich in Abschnitt 4.3.4.

- **Die Seite wird von der lokalen Festplatte geladen:**
  Dieser Fall tritt ein, wenn sich die Seite in keinem der Datenbankcaches der beteiligten Rechner, sondern auf der lokalen Platte des anfordernden Rechners befindet. Die Ausführungszeit eines solchen Auftrages liegt, inklusive Aufruf durch die Cacheverwaltung, zwischen 8 ms und 20 ms. Die Schwankungen hängen dabei im wesentlichen von der Art des Zugriffs auf die Festplatte ab. Bei erneutem, schnellem Zugriff auf die gleiche Spur der Festplatte (sequentielles Lesen), bei der keine Neupositionierung des Lesekopfes notwendig ist, können kurze Zeiten erzielt werden. Allerdings kann die Ausführungszeit auch auf unter 2 ms sinken, falls sich die vom Datenbanksystem angeforderten Dateiblöcke noch im Festplattencache des Betriebssystems befinden. Dieser Cache läßt sich nicht ausschalten[10].

- **Die Seite wird über NFS von entfernter Festplatte geladen:**
  Dieser Fall tritt ein, wenn sich die Seite in keinem der Datenbankcaches der beteiligten Rechner und nicht auf der lokalen Platte des anfordernden Rechners befindet. Die Ausführungszeit eines solchen Auftrags liegt, inklusive Aufruf durch die Cacheverwaltung, im günstigen Fall zwischen 20 ms und 30 ms. Bei hoher Netzbelastung steigt dieser Wert entsprechend Fall 1 an. Allerdings kann die Ausführungszeit wie bei der lokalen Festplatte auf unter 2 ms sinken, falls sich die angeforderten Dateiblöcke im lokalen NFS-Cache befinden. Auch dieser Cache läßt sich weder ausschalten noch ist seine Größe einstellbar.

Die Auswirkungen der oben beschriebenen Eigenschaften in einer Mehrrechnerkonfiguration sind in Abbildung 3.9 dargestellt. Sie zeigt die Ergebnisse von Durchsatzmessungen auf einem 2- sowie einem 4-Rechnersystem. Gemessen wurde unter TPC–C-Last, einer Last mit vielen Festplattenzugriffen und in dieser Konfiguration mit einer Cachetrefferrate zwischen 20 % und 50 %. Die Datenbanken wurden dabei alternativ lokal repliziert, entfernt verteilt oder lokal verteilt[11] gehalten.

Der hier zu beobachtende Effekt ist, daß mit steigender Cachegröße der Durchsatz entweder abfällt oder bestenfalls gleich bleibt. Dies betrifft sowohl das 2-Rechner- als auch das 4-Rechnersystem.

Bei den Konfigurationen mit lokal replizierten Daten fällt der Durchsatz wie zu erwarten war besonders stark ab, da die lokalen Festplattenzugriffe billiger sind als die Zugriffe über das verwendete Fast-Ethernet.

---

[10] Dadurch ließe sich eine Situation simulieren, in der lokale Festplattenzugriffe nicht billiger als Netzzugriffe sind.

[11] Bei N Rechnern hält jeder Rechner $\frac{1}{N}$ der Daten lokal. Siehe Abschnitt 3.5.1.

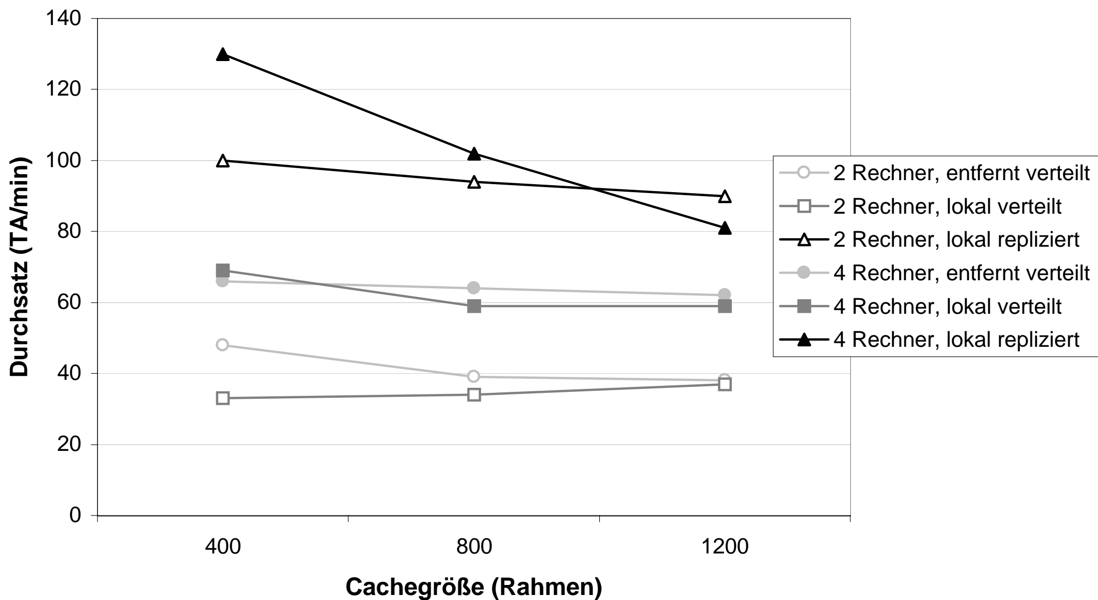

Abbildung 3.9: Cacheeffekte bei steigender Rahmenzahl

Aber auch bei den Konfigurationen, bei denen alle Daten auf entfernten Festplatten liegen und somit über NFS transferiert werden müssen (entfernte Verteilung), steigt der Durchsatz mit steigender Cachegröße nicht an, obwohl die NFS-Zugriffe normalerweise teurer sind als die Seitenübertragung aus einem Datenbankcache eines anderen Rechners. Dieser Effekt kommt durch den NFS-Cache zustande. Durch Treffer im NFS-Cache bei NFS-Festplattenzugriffen wird die durchschnittliche Zugriffszeit von NFS-Zugriffen auf etwa die Zugriffszeit von Seitenübertragungen aus Datenbankcaches gesenkt.

Fazit

Zusammenfassend muß festgestellt werden, daß MIDAS in einer Mehrrechnerkonfiguration mit der zur Verfügung stehenden Hardwareausstattung keine Vorteile aus großen Datenbankcaches ziehen kann[12], da Treffer in entfernten Datenbankcaches oftmals teurer als Festplattenzugriffe sind[13].

Eine "natürlichere" Situation könnte man hier nur über ein schnelleres Netzwerk

[12]Die Datenbankcaches dürfen allerdings nicht besonders klein sein, um ein gutes Funktionieren verschiedener Datenbankoperatoren, die eine gewisse Mindestanzahl an Datenbankcacheseiten benötigen, zu gewährleisten.

[13]Nicht berücksichtigt sind hierbei rechnerlokale Operationen wie Sortierungen, die bei größeren lokalen Datenbankcaches ihre E/A-Operationen reduzieren und somit natürlich von größerem Cache profitieren.

schaffen[14], das in der Lage ist, Datenbankseiten schneller zu transferieren als lokale Festplatten dies tun. Alternativ dazu müßte die Möglichkeit bestehen, den NFS-Cache zu deaktivieren. Damit wären NFS-Zugriffe wieder langsamer als der Seitenaustausch zwischen den Datenbankcaches. Sind dann alle Daten auf entfernten Festplatten gespeichert, entsteht das gewünschte Verhalten im System. Festplattenzugriffe sind nicht billiger als entfernte Cachezugriffe. Allerdings entsteht dabei eine sehr hohe NFS-Netzbelastung.

3.5.3 NFS und Datenkohärenz

Neben den Cacheeffekten und der hohen Netzbelastung, die durch die NFS-Zugriffe verursacht werden, ist noch zu berücksichtigen, daß der Einsatz von NFS keine Datenkohärenz bei verteilten Schreiboperationen garantiert. Es ist beispielsweise nicht sichergestellt, daß nach dem Schreiben einer Datenseite auf Festplatte ein Lesen derselben Datenseite von einem anderen Rechner die neue Version der Seite liefert.

Darin liegt ein weiterer Grund, NFS in Zukunft nicht mehr zu verwenden. Als Alternative zu NFS bietet sich die Verwendung einer von NFS unabhängigen E/A-Komponente an, die wie ein verteiltes Dateisystem wirkt, für die gewünschte Kohärenz sorgt und den Prototypen weiterhin als Shared-Disk-Architektur erscheinen läßt. Als Grundlage für die Implementierung einer solchen E/A-Komponente könnte die Programmbibliothek PFSLib dienen [Lamberts 97]. Allerdings wird durch eine derartige Lösung das Problem der hohen Netzbelastung durch Festplattenzugriffe nicht behoben.

Eine optimale Lösung würde die Verwendung von direkt an alle Rechner angeschlossenen Festplatten bieten, beispielsweise in Form eines RAID-Systems.

3.6 Pufferverwaltung – Sperrverwaltung und Kohärenzkontrolle

Dieser Abschnitt beschreibt kurz den in der Pufferverwaltung verwendeten Algorithmus zur Kohärenzkontrolle, da die dort vorkommenden Begriffe bei späteren Leistungsanalysen in Kapitel 4 benötigt werden.

In [Listl 96] wurde für die Kohärenzkontrolle ein Algorithmus mit Invalidierung zum Transaktionsende und dynamischen Seitenbesitzern vorgeschlagen. Dieser

[14]Die Ursache für die zu langsame Übertragung liegt nur zu einem sehr kleinen Teil an der Verwendung von PVM, da PVM, wie in Abschnitt 3.3 gezeigt, die vorhandenen Netzkapazitäten relativ gut ausnutzt.

Algorithmus wurde in erste Versionen von MIDAS integriert.

Messungen auf dem Simulationssystem DBSIM [Bozas 98] haben gezeigt, daß der Algorithmus zu viel Kommunikation im System hervorruft. Als Alternative, die einfach zu implementieren ist und weniger Kommunikation erzeugt, wurde ein Algorithmus mit On-Request-Invalidierung und festen Seitenbesitzern vorgeschlagen [Bozas 98] und in MIDAS implementiert. Zusätzlich sind bei diesem Algorithmus die Zuständigkeiten der Kohärenzkontrolle und der Sperrverwaltung identisch vergeben, d. h. der Seitenbesitzer ist gleichzeitig auch für die Vergabe von Sperren auf die Seite verantwortlich. Dadurch können die Synchronisationsnachrichten und Nachrichten der Kohärenzkontrolle zusammengelegt werden. Dadurch kann das Nachrichtenvolumen zusätzlich gesenkt werden.

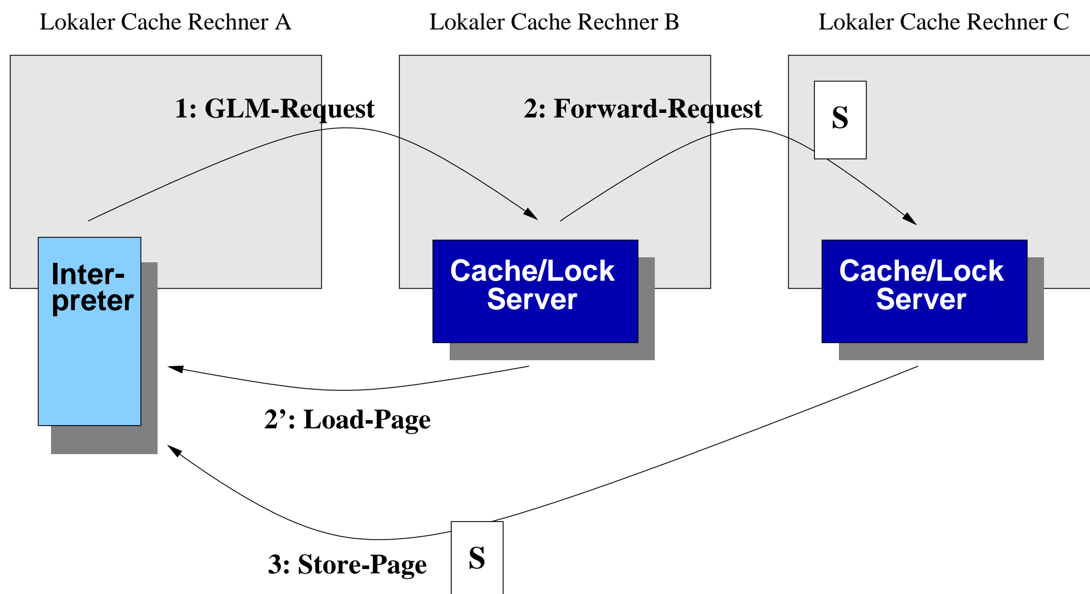

Abbildung 3.10: Nachrichten der Kohärenzkontrolle und Sperrverwaltung

Der allgemeine Ablauf des Algorithmus ist in Abbildung 3.10 schematisch dargestellt. Die Abbildung enthält keine vollständige Darstellung aller Fälle, die auftreten können und dient nur der Begriffsklärung für die in späteren Abschnitten verwendeten Begriffe. Eine ausführliche Beschreibung des Algorithmus findet sich in [Bozas 98].

In einem Interpreter auf Rechner A entsteht eine Seitenanforderung auf Seite S, die sich nicht im lokalen Cache befindet. Der Interpreter sendet daraufhin eine Nachricht (**GLM-Request**) an den Besitzer der Seite, den Rechner B, um die aktuelle Version der Seite und eine Sperre auf die Seite zu erhalten.

Ist die aktuelle Version der Seite in einem der lokalen Caches vorhanden (hier auf Rechner C), so wird die Anforderung an den Rechner C weitergeleitet (**Forward**), auf dem sich die Seite aktuell befindet. Dieser sendet die Seite dann zusammen mit der gewährten Sperre an Rechner A (**Store-Page**).

Ist die Seite in keinem der lokalen Caches vorhanden, bekommt Rechner A vom Besitzer der Seite die Nachricht, sich die Seite selbst von Festplatte zu laden (**Load-Page**).

3.7 Cachebereiche – Rahmenkontingente

In MIDAS wurde eine dynamische Verteilung von Cache-Rahmenkontingenten implementiert [Brandmayer 97]. Die im lokalen Cache vorhandenen Rahmen werden bis zur vollständigen Füllung des Caches mit MIDAS-Seiten an beliebige Segmenttypen vergeben. Als Segmenttypen werden permanente Segmente, Kommunikationssegmente und temporäre Segmente, die beispielsweise Zwischenergebnisse von Sortierungen beinhalten, unterschieden.

Bei der Vergabe der Rahmen spielt es zunächst keine Rolle, für welchen Segmenttyp der Rahmen angefordert wurde. Erst bei vollständiger Füllung des Caches, wenn die ersten Seiten wieder verdrängt werden müssen, wird an Hand einer prozentualen Einstellung entschieden, welcher Segmenttyp mehr als den ihm zustehenden Platz belegt. Seiten dieses Typs werden zuerst verdrängt, um freie Rahmen zu erhalten. Erst wenn keiner der Segmenttypen sein Kontingent überschritten hat und keine freien Rahmen mehr existieren, werden auch Seiten aus dem eigenen Kontingent an Rahmen verdrängt.

Diese Strategie sichert eine optimale Ausnutzung der gesamten zur Verfügung stehenden Anzahl an Cacherahmen, unabhängig von den speziellen Anforderungen der bearbeiteten Anfragen. Im Gegensatz dazu hat TransBase getrennte Bereiche fester Größe für permanente und temporäre Segmente, wodurch viel Speicherplatz ungenutzt bleibt, wenn der entsprechende Segmenttyp nicht oder nur wenig benutzt wird.

3.8 Zusammenfassung

In diesem Kapitel wurden wesentliche Aspekte, die sich bei der Implementierung des Ausführungssystems durch neue Konzepte oder Umstellung alter Systemkomponenten von TransBase ergaben, vorgestellt und ihre Auswirkungen auf die Leistungsfähigkeit des Systems analysiert.

Zunächst wurde gezeigt, wie das Konzept der Seiten und Subseiten einfach in

MIDAS zu integrieren war und dabei eine leichte Portierbarkeit alter TransBase-Anwendungen beibehalten werden konnte.

Des weiteren stellten sich Latches als am besten geeigneter Mechanismus zur Synchronisation auf den Datenstrukturen von MIDAS heraus. Das dabei verwendete Granulat der Synchronisation wurde so gewählt, daß ein hoher Parallelitätsgrad im Ausführungssystem ermöglicht wird, ohne zu viele Verwaltungskosten zu erzeugen.

Anschließend wurden die Kommunikationscharakteristika von MIDAS vorgestellt. Das Nachrichtenaufkommen ist kontinuierlich sehr hoch, so daß eine Umstellung der Architektur des Ausführungssystems sowie lokale Varianten der Kommunikationssegmente notwendig wurden, um teure Nachrichten einzusparen. PVM sowie einige PVM-spezifische Einstellungen wurden bewertet.

Weiterhin stellte es sich als notwendig heraus, Daten im parallelen Datenbanksystem MIDAS auf mehrere Festplatten zu verteilen, um die Festplatten nicht zu schnell zum Leistungsengpaß werden zu lassen und die Architektur einem Shared-Disk-System anzunähern. Die aktuelle Hardwarekonfiguration mit dem verwendeten Fast-Ethernet, das außer durch Systemnachrichten auch noch durch NFS-Zugriffe belastet wird, und Betriebssystemcaches, die das Verhalten des Datenbankcaches negativ beeinflussen, stellen zusätzliche Probleme bei der Ein- und Ausgabe dar.

Bei der Umsetzung von Konzepten in einem realen System sind viele zusätzliche Anforderungen an eine effiziente Implementierung zu beachten, ohne die kein gut funktionierendes System entstehen kann. Bei der Implementierung von MIDAS konnten diese Anforderungen erfüllt werden. MIDAS stellte sich als leistungsfähiges System für die Evaluierung verschiedener Datenbankkonzepte heraus.

Kapitel 4

Leistungsanalysen

Dieses Kapitel befaßt sich mit Leistungsanalysen, die die erfolgreiche Implementierung von MIDAS belegen. Dazu werden in Abschnitt 4.1 vergleichende Analysen mit TransBase vorgestellt, aus dem MIDAS entwickelt wurde. Danach folgen Aussagen über die Skalierbarkeit des Systems.

Es werden Aussagen über die Effizienz des Ausführungsystems getroffen. Dazu wurden hauptsächlich Messungen mit Inter-Transaktionsparallelität durchgeführt.

Zuerst wird die Leistung der Systeme MIDAS und TransBase verglichen, um so den Entwicklungsprozeß bewerten zu können (Abschnitt 4.1). Anschließend wird die Skalierbarkeit des Systems analysiert, dabei wird auch auf Intra-Transaktionsparallelität eingegangen (Abschnitt 4.2). In Abschnitt 4.3 erfolgt die Analyse des E/A-Verhaltens des Systems. Dabei werden sowohl Cacheeffekte als auch E/A-Granulate betrachtet. Abschließend zeigt Abschnitt 4.4 Leistungsgrenzen des Systems auf.

Sofern nicht anders vermerkt, wurden bei den durchgeführten Messungen die Rechnerkonfigurationen aus Anhang A sowie die in Anhang B bis Anhang D ausführlich beschriebenen Benchmarks, Datenbanken und Transaktionen verwendet. Die meisten Messungen wurden mit zwei unterschiedlichen, repräsentativen Lasten, einer OLTP-Last und einer komplexeren, E/A-intensiveren TPC–C-Last, durchgeführt.

4.1 Vergleich von MIDAS mit TransBase

In Abschnitt 2.5 wurde beschrieben, wie MIDAS aus TransBase entwickelt wurde und welche Teile des Source-Codes dabei wiederverwendet werden konnten. Bei der Entwicklung war zu erwarten, daß in MIDAS aufgrund der eingeführ-

ten Parallelität Leistungseinbußen gegenüber TransBase auftreten werden. Als Grund für die zu erwartenden Leistungseinbußen in MIDAS kamen die größere Anzahl an Prozessen, die Prozeßgrößen und die durch die verteilte Cacheverwaltung benötigten neuen Algorithmen zur Sperrverwaltung und Kohärenzkontrolle in Frage. Die ersten Fragestellungen, die sich für den lauffähigen Prototyp MIDAS ergaben, waren deshalb:

- Wie leistungsfähig ist MIDAS im Vergleich zu TransBase?
- Können einzelne Anfragen in etwa der gleichen Zeit beantwortet werden?
- Bringt die eingeführte Parallelität Vorteile bei mehreren parallel laufenden Anfragen?

4.1.1 Meßbedingungen

Die originale TransBase-Version, aus der MIDAS entwickelt wurde, ist nur unter SunOS 4.1 lauffähig, MIDAS dagegen wird unter Solaris ab Version 2.5 entwickelt. Um vergleichbare Meßergebnisse von beiden Systemen erhalten zu können, wurde TransBase ebenfalls auf Solaris portiert. Dadurch konnten bei den Vergleichsmessungen identische Bedingungen bei Hardware, Betriebssystem und Übersetzern geboten werden.

Auf der Seite der Datenbanksysteme wurde versucht, alle Systemparameter möglichst identisch einzustellen. Diese sind in Tabelle 4.1 gegenübergestellt.

Einstellung	**TransBase**	**MIDAS**
Cache für permanente Segmente	16 MByte	16 MByte
Seitengröße	8 kB	8 kB
Anzahl Subseiten	—	1
Datenbank TPC–A	20 tps (210 MByte)	
Datenbank TPC–C	3 Warehouses (320 MByte)	
Lage der Datenbanken	lokal	
Betriebssystem	Solaris	
Rechner[1]	4-Prozessorsystem	
Hauptspeicher	128 MByte	

Tabelle 4.1: Gegenüberstellung der Meßbedingungen in TransBase und MIDAS

Insbesondere wurden bei beiden Systemen gleich große Datenbankpuffer angelegt und identische Seitengrößen verwendet. Da in der Implementierung von MIDAS die Subseiten den TransBase-Seiten entsprechen, bestanden die MIDAS-Seiten daher auch jeweils in genau einer Subseite. Damit waren die pufferinternen Verarbeitungseinheiten (MIDAS-Subseiten, TransBase-Seite) sowie die Austauscheinheiten mit dem Sekundärspeicher (MIDAS-Seiten, TransBase-Seite) identisch. Durch die Einschränkung auf eine Subseite besitzt MIDAS allerdings einen kleinen Nachteil, da es Seiten und Subseiten verwalten muß, aber nicht von einer hohen Subseitenanzahl profitieren kann.

Die Datenbankgröße wurde so gewählt, daß die DB weder im Betriebssystem- noch im Datenbankcache gehalten werden kann. Alle gestarteten Prozesse befanden sich auf dem 4-Prozessorsystem.

4.1.2 OLTP-Last

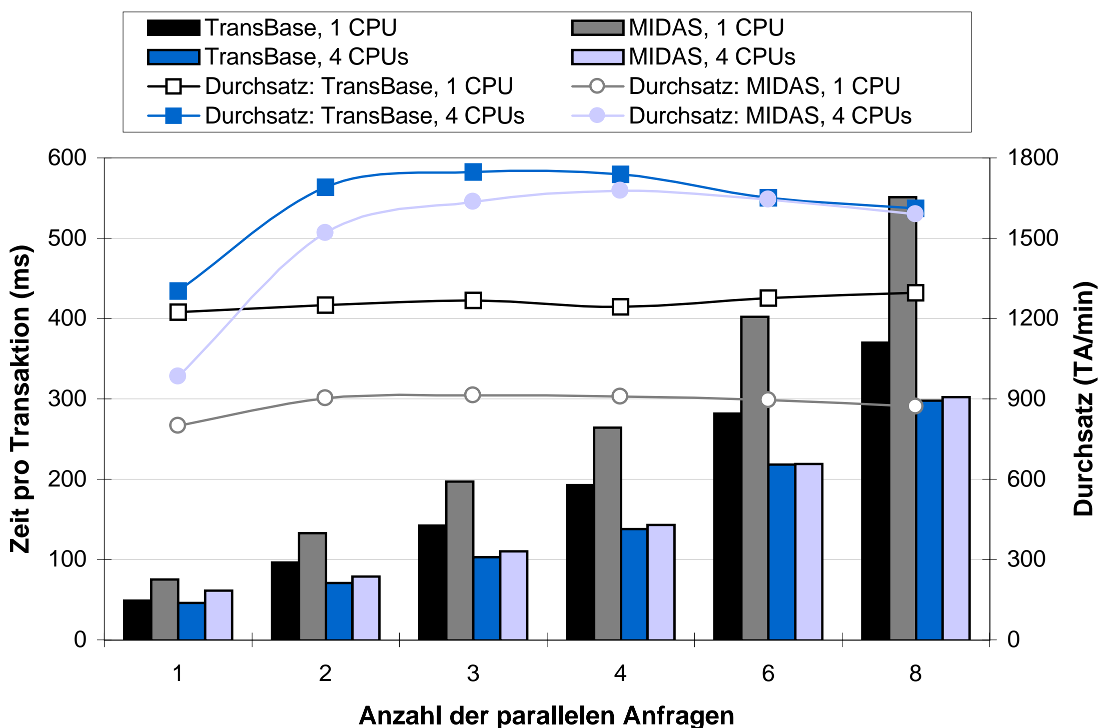

Abbildung 4.1: TransBase, MIDAS – Laufzeiten und Durchsatz unter TPC–A^{RO}

In Abbildung 4.1 sind die Meßwerte für eine unterschiedlich große Anzahl paralleler TPC–A^{RO}-Anfragen als Balken dargestellt. Bei jedem Parallelitätsgrad

werden die Werte von TransBase und MIDAS einander gegenübergestellt. Dabei existieren für jedes System zwei Balken, wobei einer die Ausführungszeit auf dem 4-Prozessorsystem mit nur einer aktiven CPU darstellt, was damit einem 1-Prozessorsystem entspricht. Der andere Balken symbolisiert die Ausführungszeit mit vier aktivierten CPUs. Die vier Kurven in der Abbildung stellen den Durchsatz in Transaktionen pro Minute dar, abgetragen auf der rechten Y-Achse.

Deutlich zu erkennen ist, daß auf dem 1-Prozessorsystem TransBase nur ca. $\frac{2}{3}$ der Zeit benötigt, die MIDAS zur Ausführung der kleinen TPC–A^{RO}-Anfragen braucht. Bei steigendem Parallelitätsgrad bleibt bei beiden der Durchsatz annähernd konstant, ebenso wie der Geschwindigkeitsvorteil von TransBase.

Dagegen erzielen beide Systeme auf dem 4-Prozessorsystem eine Durchsatzsteigerung bis zum Parallelitätsgrad vier. Bei höherem Parallelitätsgrad bleibt der Durchsatz konstant. Der Geschwindigkeitsvorteil von TransBase beim Parallelitätsgrad 1 von 25 % wird mit steigendem Parallelitätsgrad von MIDAS reduziert, und ab Parallelitätsgrad 6 kommen beide Systeme auf einen identischen Durchsatz (vergleiche auch die Skalierbarkeitsmessungen in Abschnitt 4.2.2).

Der höhere Durchsatz von TransBase kann mit dem Mehraufwand erklärt werden, der in MIDAS bei der Ausführung einer Transaktion betrieben wird. Der Mehraufwand ist zum einen bedingt durch die Prozeßgrenze zwischen Anfrage- und Ausführungssystem sowie die aufwendigere Cacheverwaltung, die für ein verteiltes Rechnersystem ausgelegt ist.

Da sich bei der Ausführung der nur durch die CPU beschränkten TPC–A^{RO}-Anfragen auf einem einzigen Prozessor die Prozesse gegenseitig behindern, sind auf dem 1-Prozessorsystem keine Verbesserungen bei mehreren parallelen Anfragen zu erwarten. Es überrascht daher nicht, daß die Durchsatzrate konstant ist, eine sequentielle Ausführung der Transaktionen wäre in der gleichen Zeit möglich. Allerdings tritt auch keine Verschlechterung des Durchsatzes ein.

Auf dem 4-Prozessorsystem können beide Systeme bei steigendem Parallelitätsgrad von den vier Prozessoren profitieren. Beide können den Durchsatz steigern, MIDAS sogar um 50 %. Eine größere Steigerung auf einem Rechner können beide Systeme nicht erzielen, da in beiden Systemen viel Synchronisation und damit Zwangssequentialisierung innerhalb der Cacheverwaltung betrieben werden muß, um ein korrektes Funktionieren der Pufferverwaltung zu garantieren. Diese Synchronisation geschieht bei beiden Systemen über Semaphore.

Bei TransBase wird bei jedem Aufruf von Funktionen der Cacheverwaltung die komplette Cacheebene gesperrt. Dies führt zu einfachen Algorithmen in der Cacheverwaltung, was neben der bei TransBase nicht zu berücksichtigenden Verteilung der Datenbank den deutlichen Geschwindigkeitsvorteil beim Parallelitätsgrad 1 bewirkt. Bei MIDAS werden dagegen nur einzelne, kleine Datenstrukturen durch die Semaphore gesperrt (siehe Abschnitt 3.2.3). Dadurch können die Pro-

zesse des Ausführungssystems in MIDAS verzahnter zueinander laufen und die zur Verfügung stehenden Prozessoren besser nutzen. So kann MIDAS bei hohem Parallelitätsgrad den gleichen Durchsatz wie TransBase erreichen. Der Nutzen solcher kleinen Synchronisationseinheiten ist bereits in Abschnitt 3.2.3.2 dargestellt.

4.1.3 TPC–C-Last

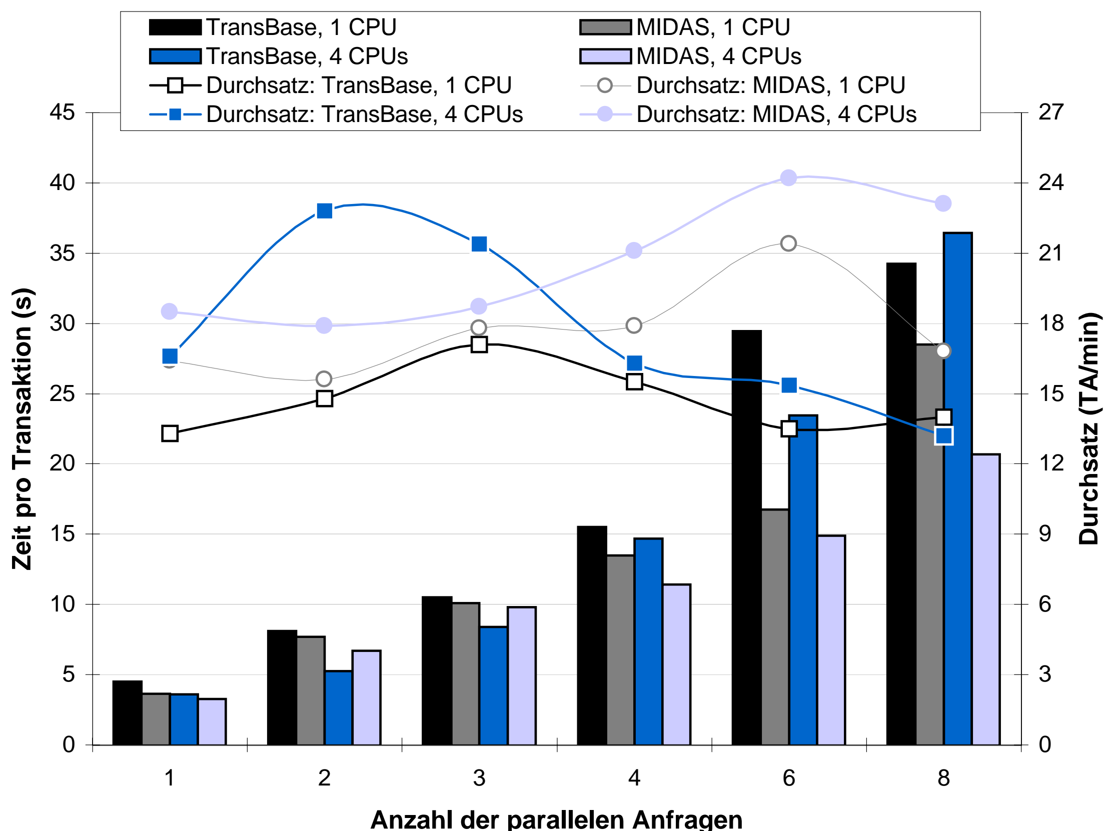

Abbildung 4.2: TransBase, MIDAS – Laufzeiten und Durchsatz unter TPC–C^{S+O}

Verglichen mit den Messungen der TPC–A^{RO}-Transaktionen überraschen die Meßwerte der TPC–C-Transaktionen auf den ersten Blick. Die ausgeführten TPC–C^{S+O}-Transaktionen sind eine zufällige Permutation aus 20 % Stock-Level-Transaktionen und 80 % Order-Status-Transaktionen (siehe Anhang D). Wie in Abbildung 4.2 zu erkennen, werden diese Transaktionen von MIDAS auf dem 1-Prozessorsystem schneller ausgeführt. Die Durchsatzrate von TransBase ist schon

ab Parallelitätsgrad 3 fallend, wohingegen bei MIDAS der maximale Durchsatz erst bei Parallelitätsgrad 6 erreicht wird. Auf dem 4-Prozessorsystem erreicht TransBase bei Parallelitätsgrad 2 den maximalen Durchsatz und liegt bei diesem deutlich vor MIDAS. Bei einer höheren Zahl paralleler Anfragen fällt der Durchsatz dann stark ab. MIDAS erreicht auch hier seinen höchsten Durchsatz bei Parallelitätsgrad 6.

Die Ursache dieser Ergebnisse ist in der speziellen Struktur der verwendeten Anfragen zu suchen. Etwa 80 % der Ausführungszeit entfallen auf die Stock-Level-Transaktion. Während dieser Transaktion wird im wesentlich die 120 MByte große Relation stock mit einem Relationen-Scan sequentiell gelesen. Durch zusätzliche Beobachtung von Systemmonitoren ergab sich, daß die Ausführung der Transaktionen durch die Leistung der Festplatten beschränkt ist.

Bei TransBase läßt sich schon bei geringem Parallelitätsgrad beobachten, daß durch das restriktive Sperren der gesamten Cacheverwaltung über Semaphore die parallelen Prozesse sich gegenseitig so verzögern, daß sie nicht die volle Leistungsfähigkeit der Festplatten ausnutzen können (vergleiche Abschnitt 3.2.3.2). Die Datenübertragungsraten sinken von Parallelitätsgrad 3 bis 8 von 3 MByte pro Sekunde auf 900 KByte pro Sekunde. Bei MIDAS tritt dieser Effekt nicht auf. Der Durchsatz wird hier durch die Anzahl paralleler Prozesse beschränkt, die vom System optimal bedient werden können. Dieser Punkt liegt bei Parallelitätsgrad 6, also höher als bei den TPC–A^{RO}-Transaktionen und damit über der Anzahl der Prozessoren. Dies liegt daran, daß die TPC–C-Last nicht CPU-beschränkt ist.

4.1.4 Schreibende Anfragen

Für den Vergleich zwischen TransBase und MIDAS mit schreibenden Transaktionen liegt nur eine alte Messung aus dem Jahr 1996 vor. Zu diesem Zeitpunkt liefen beide Systeme noch unter dem Betriebssystem SunOS 4.1. Die bei den Messungen in den vorherigen Abschnitten verwendete TransBase-Version für Solaris kann schreibende Transaktionen nur mir starken Zeitverzögerungen durchführen. Es werden deshalb die Messungen unter SunOS 4.1 präsentiert, die auf einer SunCLASSIC stattfanden.

Abbildung 4.3 zeigt die im Durchschnitt für eine TPC–A-Transaktion benötigten Zeiten. MIDAS weist dabei eine leicht bessere Ausführungszeit auf.

Diesem Diagramm kann man allerdings nur entnehmen, daß MIDAS *zum damaligen Zeitpunkt* bei kleinen schreibenden Transaktionen etwas schneller als TransBase war. Bei der Interpretation der Meßergebnisse muß aber berücksichtigt werden, daß MIDAS eine NoForce-Strategie anwendet, d. h. Änderungen nicht sofort, wie TransBase, auf die Festplatte schreibt. Dafür müssen bei dieser Strategie Recovery-Informationen zum Commit-Zeitpunkt geschrieben werden. Da in

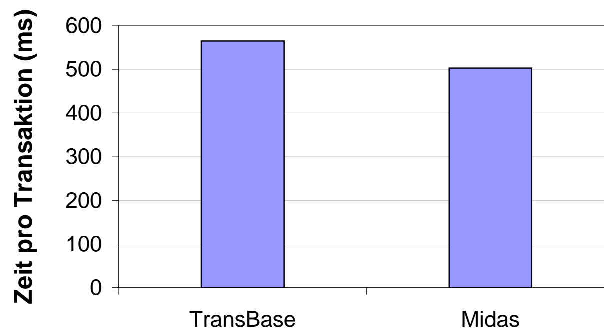

Abbildung 4.3: Durchschnittliche Zeiten für eine Transaktion TPC–A

MIDAS keine Recovery implementiert ist, fällt beim Commit einer schreibenden Transaktion fast keine Arbeit an, wogegen TransBase die Änderungen sofort permanent macht.

Betrachtet man die Werte vor diesem Hintergrund, so ist nach einer Implementierung von Logging und Recovery in MIDAS ein Vorteil für TransBase zu erwarten.

Tests mit mehreren parallelen Anfragen konnten wegen damals noch fehlender Deadlock-Behebung im Lockmanager nicht durchgeführt werden. TransBase dürfte bei einem solchen Test mit mehreren Anfragen aber sehr schlecht abschneiden, da in der verwendeten TransBase-Version Sperren nur auf ganze Relationen vergeben werden und daher keinerlei parallele Ausführung der Transaktionen möglich wäre.

4.1.5 Zusammenfassung

Insgesamt zeigt der Vergleich von TransBase und MIDAS Vorteile zugunsten von TransBase auf dem 1-Prozessorsystem bei kleinen Anfragen. Die Vorteile resultieren aus dem Mehraufwand, der für die Parallelisierung in MIDAS betrieben wurde. MIDAS besitzt eine aufwendigere, für ein verteiltes Rechnersystem ausgelegte Cacheverwaltung sowie mehr Prozesse. Für den betriebenen Mehraufwand ist der Leistungsunterschied von ca. $\frac{1}{3}$ als gut zu bewerten.

Bei den anderen Konfigurationen mit höherem Parallelitätsgrad oder mehr Prozessoren fällt TransBase gegenüber MIDAS ab, da die Nachteile von TransBase zutage treten. Hauptnachteil ist dabei das restriktive Sperren der gesamten Cacheverwaltung über Semaphore. MIDAS als ein auf die Parallelverarbeitung ausgelegtes System zeigt dabei schon auf einem Rechner seine guten Eigenschaften.

4.2 Skalierbarkeit

In dieser Meßreihe wurde die Skalierbarkeit von MIDAS auf unterschiedlichen Systemkonfigurationen und unter unterschiedlichen Lasten analysiert. Bei allen Messungen wurden 32 KByte MIDAS-Seiten mit jeweils vier Subseiten verwendet. Der lokale Datenbankcache war auf 800 Rahmen eingestellt.

4.2.1 Verteiltes Mehrrechnersystem

Verwendet wurde das in Anhang A beschriebene verteilte Mehrrechnersystem. Die Verteilung der Datenbank wurde zwischen entfernter und lokaler Verteilung sowie lokaler Replikation variiert (siehe Abschnitt 3.5.1.1). Die Skalierung fand von zwei auf vier Rechner statt. Pro Rechner war dabei eine Transaktion aktiv.

4.2.1.1 OLTP-Last

Als Transaktionslast wurde die OLTP-Version TPC–C^{N+P} des TPC–C Benchmarks verwendet. Abbildung 4.4 zeigt die bei den Messungen bestimmten Durchsätze sowie die Skalierbarkeitswerte bei einer Skalierung von zwei auf vier Rechner.

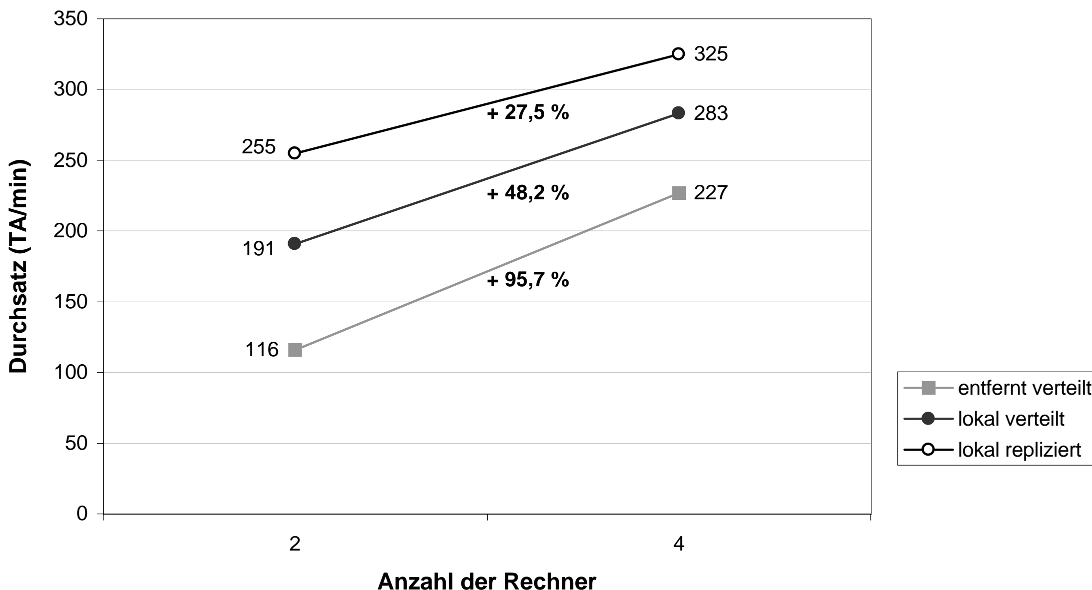

Abbildung 4.4: Mehrrechnersystem – Skalierbarkeit unter OLTP-Last

Die beste Skalierbarkeit von 95,7 % wird erreicht, wenn die Datenbank komplett auf entfernten Rechnern verteilt ist und somit alle Zugriffe über NFS erfolgen. Bei der üblicherweise verwendeten Konfiguration, der lokalen Verteilung, bei der die Datenbank auf die lokalen Platten der beteiligten Rechner verteilt ist, ist der erreichte Durchsatz höher. Es wird in dieser Konfiguration aber nur eine Skalierbarkeit von 48,2 % erreicht. Der höchste Durchsatz wird bei lokaler Replikation der Daten erreicht. Die Skalierbarkeit sinkt bei lokaler Replikation allerdings auf 27,5 %.

Die Skalierbarkeit der entfernten Verteilung gibt die Skalierbarkeit des Systems unter OLTP-Last am besten wieder. Keine der wichtigen Ressourcen, CPU, Festplatte und Netzwerk, wird hier zum Engpaß. Die Skalierbarkeit liegt mit 95,7 % nahezu an dem erwarteten Optimum.

Die schlechtere Skalierbarkeit bei lokaler Verteilung resultiert nicht aus einem Systemengpaß, sondern aus den Eigenschaften der Verteilung. Die 2-Rechnerkonfiguration ist gegenüber der 4-Rechnerkonfiguration bevorteilt. Bei ihr erfolgt die Hälfte der Festplattenzugriffe lokal, wohingegen bei der 4-Rechnerkonfiguration nur auf ein Viertel der Daten lokal zugegriffen werden kann.

Auch bei lokaler Replikation ist die niedrige Skalierbarkeit in der Bevorteilung der 2-Rechnerkonfiguration begründet. Bei der 4-Rechnerkonfiguration ist insgesamt mehr Datenbankcache vorhanden, wodurch mehr Cachetreffer auf entfernten Rechnern entstehen. In dieser Konfiguration ist allerdings der Zugriff auf entfernte Caches teurer als der Zugriff auf lokale Festplatten. Dieser Effekt, der aus dem verwendeten Netzwerk sowie aus Betriebssystemcaches resultiert, ist in Abschnitt 3.5.2 genauer beschrieben.

Bei allen 4-Rechnerkonfigurationen ist bereits eine deutlich erhöhte Belastung des Netzwerks bemerkbar. Die Bearbeitungszeit von Store-Page-Aufträgen (siehe Abschnitt 3.6), in denen ganze Seiten zwischen den lokalen Datenbankcaches ausgetauscht werden, steigt hier von 12 ms auf 30 ms an. Dies ist ein Hinweis darauf, daß sich dort die Leistungsgrenze des verwendeten Netzwerks nähert.

4.2.1.2 TPC–C-Last

Die Ergebnisse der Skalierungsmessungen auf MIDAS mit dem TPC–C Benchmark sind in Abbildung 4.5 dargestellt.

Den höchsten Durchsatz zeigt die Konfiguration mit lokal replizierten Daten. Diese weist aber eine geringe Skalierung von nur 9 % beim Übergang von zwei auf vier Rechner auf. Die Konfigurationen mit lokal verteilten und entfernt verteilten Daten zeigen in etwa dieselben Werte bei Durchsatz und Skalierung. Bei beiden Konfigurationen liegt die Skalierung zwischen 64 % und 74 %.

Diese Werte sind als gut einzustufen. Beide Konfigurationen können von der

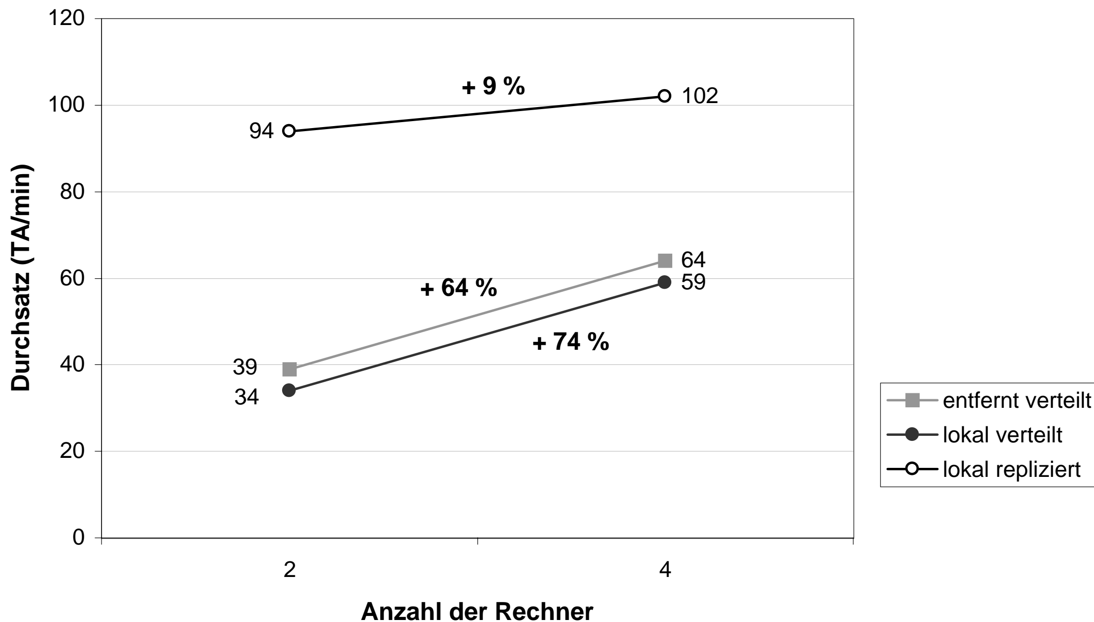

Abbildung 4.5: Mehrrechnersystem – Skalierbarkeit unter TPC–C-Last

Erhöhung der Anzahl von Prozessoren und Festplatten profitieren.

Wie erwartet zeigen sich unter dieser Last sehr große Vorteile beim Durchsatz, bei Verwendung von lokal replizierten Daten. Da diese Last viele Relationen-Scans aufweist und sehr viel E/A erzeugt, kann in hohem Maße von den lokalen Festplatten profitiert werden. Die geringe Skalierbarkeit bei dieser Datenverteilung hat dieselben Gründe wie unter OLTP-Last. Bei der 4-Rechnerkonfiguration entstehen mehr Treffer in den entfernten Datenbankcaches, da der gesamte Datenbankcache größer ist. Diese Zugriffe sind teurer als die lokalen Zugriffe auf Festplatte, die zusätzlich durch Betriebssystemcaches unterstützt werden (siehe Abschnitt 3.5.2).

4.2.2 Mehrprozessorsystem

Zusätzlich zu der Mehrrechnerkonfiguration wurden auch Skalierungsmessungen auf dem 4-Prozessorsystem durchgeführt. Im Gegensatz zum Mehrrechnersystem, bei dem die Leistungswerte in starkem Maße vom verwendeten Netzwerk abhängen, da viele NFS-Zugriffe und Nachrichten über das Netz stattfinden, wird hier die Parallelität auf einem Rechner bewertet. Bei dieser Konfiguration ist die gegenseitige Beeinflussung der Prozesse auf einem Rechner entscheidend.

Verwendet wurde das in Anhang A beschriebene Mehrprozessorsystem. Die Skalierung fand von einem auf vier aktive Prozessoren statt. Die Datenbank war auf

die lokalen Platten verteilt (siehe Abschnitt 3.5.1.1). Es wurden jeweils gleich viele Platten wie aktive Prozessoren verwendet. Pro Prozessor gab es eine aktive Transaktion.

4.2.2.1 OLTP-Last

Als Transaktionslast wurde die lesende Version TPC–A^{RO} des TPC–A Benchmarks verwendet. Abbildung 4.6 zeigt die ermittelten Werte.

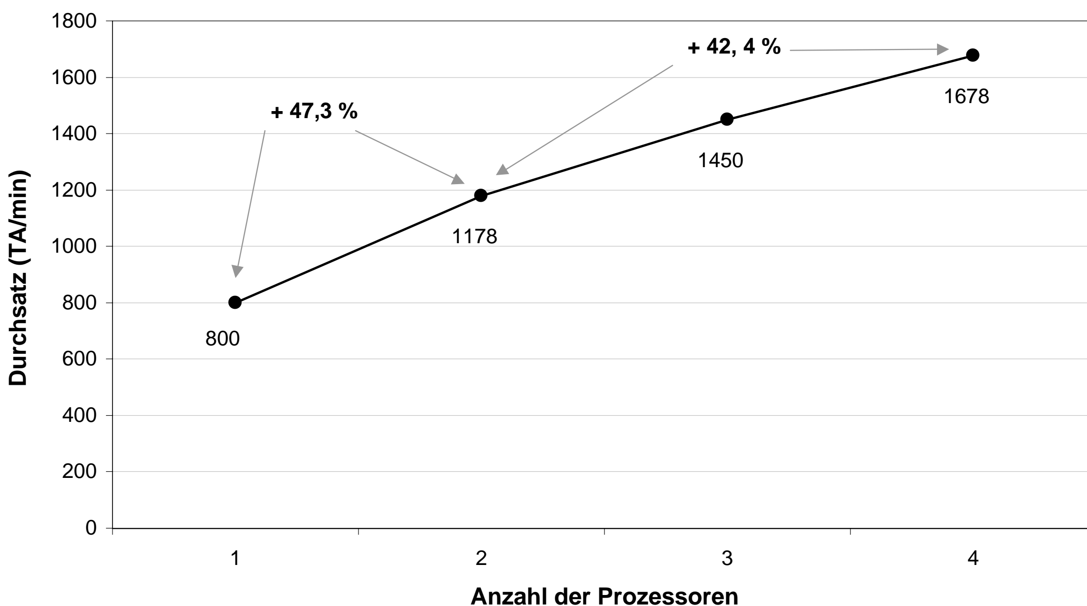

Abbildung 4.6: Mehrprozessorsystem – Skalierbarkeit unter OLTP-Last

Bei der Vergrößerung des Systems von einem auf zwei Prozessoren und Festplatten wurde eine Skalierung von 47,3 % erreicht. Die Skalierung des Systems von zwei auf vier Prozessoren und von zwei auf vier Festplatten brachte einen Leistungsgewinn von 42,2 %.

Die Datentransferrate von den Festplatten sank dabei von 600 KByte/s auf 180 KByte/s. Diese Werte sind charakteristisch für den TPC-A Benchmark, bei dem meist nur eine einzelne Seite pro Anfrage gelesen wird. Die maximalen Transferraten der Festplatten sind dabei bei weitem nicht erreicht.

Die Beschränkungen des Systems liegen in diesem Fall bei den Synchronisationsmechanismen des Ausführungssystems, in dem verschiedene, von den Prozessen gemeinsam verwendete Systemressourcen über Semaphore geschützt werden

müssen. Beim Zugriff auf diese Ressourcen findet eine Sequentialisierung der Prozesse statt, so daß keine hohe Skalierbarkeit von bis zu 100 % erreicht werden kann. Weitere Aussagen über die Effizienz dieser Synchronisation finden sich in Abschnitt 3.2.3.2.

Die erreichte Skalierbarkeit ist für ein 1-Rechnersystem als gut zu bewerten. Entsprechende Skalierbarkeitstests bei TransBase ergaben beispielsweise nur Skalierungswerte von 20 % von einem auf zwei Prozessoren und von 17 % von zwei auf 4 Prozessoren (vergleiche Abschnitt 4.1.2).

4.2.2.2 TPC–C-Last

Zum Vergleich mit dem Mehrrechnersystem wurde auch das Mehrprozessorsystem auf seine Skalierbarkeit unter TPC–C-Last untersucht. Die ermittelten Werte sind in Abbildung 4.7 dargestellt.

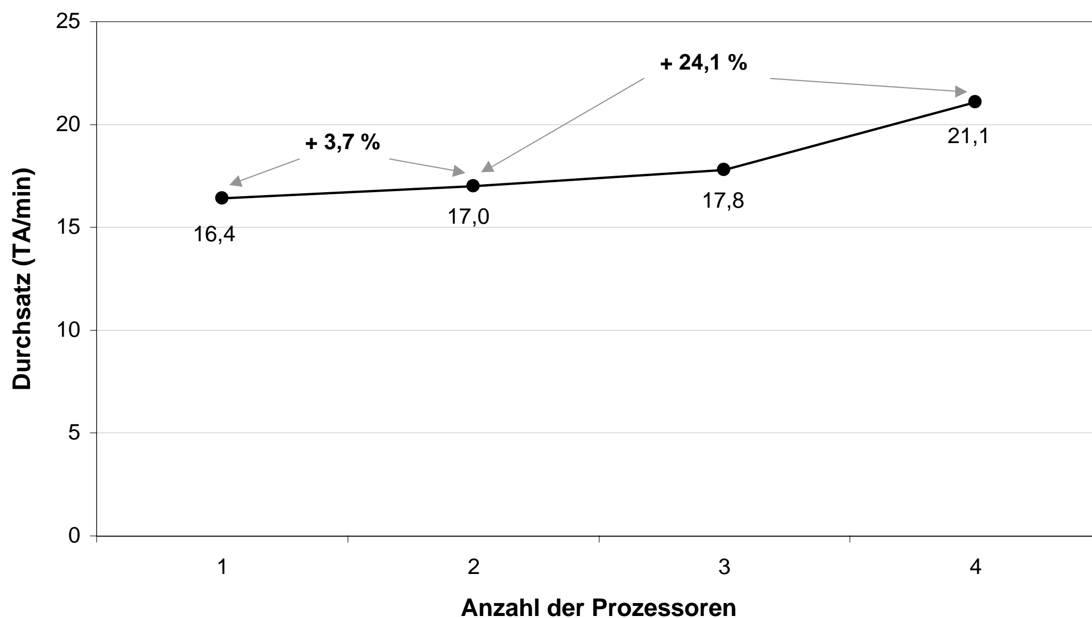

Abbildung 4.7: Mehrprozessorsystem – Skalierbarkeit unter TPC–C-Last

Bei der Skalierung von einem auf zwei Prozessoren wird ein Leistungsgewinn von 3,7 % erreicht, von zwei auf vier Prozessoren sind es 24,1 %.

Die erreichten Werte sind dabei deutlich schlechter als die unter TPC-A-Last gemessenen. Es kann dabei nur geringfügig von den zusätzlichen Prozessoren profitiert werden. Leistungseinschränkend ist die schon bei der OLTP-Last be-

schriebene, auf einem Rechner notwendige Synchronisation, die zu einer gewissen Sequentialisierung führt.

Als größter Engpaß stellen sich bei dieser Last allerdings die konkurrierenden Festplattenzugriffe heraus. Während bei einer aktiven Transaktion noch 3,2 MByte/s Datentransfer erzielt werden können, sinkt dieser Wert bei vier parallelen Transaktionen auf unter 1 MByte/s. Dies ist bedingt durch die konkurrierenden Zugriffe auf die Festplatten, wodurch die Relationen-Scans, die etwa 80 % der Ausführungszeit einer TPC–C-Transaktion ausmachen, nicht mehr optimal sequentiell lesend durchgeführt werden können.

4.2.3 Intra-Transaktionsparallelität

Die bisher gezeigten Skalierbarkeitsmessungen fanden alle unter einer Last statt, die Inter-Transaktionsparallelität erzeugt. Dies war der Hauptfokus der Arbeit. Die Leistungsfähigkeit des Ausführungssystems ist damit gezeigt worden.

Die Skalierung des Systems bei der Parallelisierung von Anfragen, bei denen Intra-Transaktionsparallelität eingeführt wird, hängt im wesentlich von den Möglichkeiten des Parallelisierers ab, dessen Entwicklung und Bewertung nicht Teil dieser Arbeit ist.

Um die Leistungsfähigkeit des Systems unter einer solchen Last zu dokumentieren, werden an dieser Stelle kurz die erzielten Leistungsdaten aus [Nippl 00, Abschnitt 4.7] vorgestellt.

Die Messungen wurden auf den vier Rechnern des Mehrrechnersystems (siehe Anhang A) durchgeführt. Verwendet wurden 16 Anfragen des TPC-D Benchmarks. Verglichen wurden die Ausführungszeiten der sequentiellen Anfragen auf einem Rechner mit denen der parallelisierten Anfragen auf den vier Rechnern des Mehrrechnersystems. Zeiten zur Parallelisierung der Anfragen sind dabei nicht berücksichtigt. 6 der 16 Anfragen konnten einen superlinearen Speedup zwischen 4,5 und 13 erzielen. Jeweils 5 der 15 Anfragen erreichten einen linearen Speedup von 4 beziehungsweise einen sublinearen Speedup zwischen 1,5 und 3,5.

Zum einen wird durch diese Ergebnisse die gute Qualität des Parallelisierers dargestellt, zum anderen zeigen die Werte aber auch, daß das zugrunde liegende Ausführungssystem die erzielten Skalierbarkeitswerte ermöglicht.

4.2.4 Zusammenfassung

Insgesamt läßt sich feststellen, daß MIDAS ein gute Skalierbarkeit aufweist. Sie fällt auf dem Mehrrechnersystem besser aus als auf dem Mehrprozessorsystem, da auf einem Mehrprozessorsystem mehr gemeinsame Ressourcen existieren, die geteilt werden müssen.

Auf dem Mehrrechnersystem wird die gute Skalierbarkeit nur durch die in Abschnitt 3.5.2 beschriebenen Wechselwirkungen mit den Betriebssystemcaches und das Nichtvorhandensein eines “echten” Shared-Disk-Datenbanksystems negativ beeinflußt. Dort können lokale Festplattenzugriffe oder Festplattenzugriffe über das Netzwerk billiger als Zugriffe auf entfernte Datenbankcaches sein.

4.3 Ein- und Ausgabe

4.3.1 Zugriff auf Daten - Probleme durch NFS

MIDAS ist als Shared-Disk-Datenbanksystem konzipiert. Die zur Verfügung stehende Hardwareausstattung erlaubt in der Mehrrechnerkonfiguration, wie in Abschnitt 3.5.2 erläutert, kein “echtes” Shared-Disk-System.

Gewünscht wäre ein System, in dem die Zugriffszeiten auf Festplatte, entfernten Cache, lokalen Cache in dieser Reihenfolge jeweils deutlich sinken. Zudem sollten Festplattenzugriffe weder das Netzwerk belasten, noch sollten von verschiedenen Rechnern unterschiedliche Zugriffszeiten auf die Festplatten existieren.

In MIDAS bieten sich zwei Alternativen, sich dem gewünschten Shared-Disk-System anzunähern. Beide realisieren aber nicht vollständig die gewünschte “echte” Shared-Disk-Konfiguration.

Zum einen besteht die Möglichkeit der entfernten Verteilung von Daten. Festplattenzugriffe sind dann zwar teuer und belasten das Netzwerk, aber alle Rechner haben einen gleichförmigen Zugriff auf alle Daten. Datenbankcachezugriffe sind im allgemeinen billiger als Festplattenzugriffe.

Die andere Alternative ist die lokale Replikation der Daten auf den Festplatten. In diesem Fall sind die Festplattenzugriffe billig und gleichförmig. Allerdings kann dabei keine Datenkonsistenz gesichert werden, so daß nur lesende Anfragen möglich sind, und die Festplattenzugriffe werden billiger als Zugriffe auf entfernte Datenbankcaches.

Dieser Abschnitt beschäftigt sich mit dem Einfluß von NFS auf die Systemperformanz. Es wird versucht abzuschätzen, wie hoch der Gewinn einer “echten” Shared-Disk-Architektur ist. Dazu sollte kein NFS mehr im System verwendet werden, um die schnellen und gleichförmigen Festplattenzugriffe einer Shared-Disk-Architektur zu simulieren. Zudem sollte das Netzwerk frei von Festplattendatentransfer gehalten werden. Am geeignetsten hierfür erscheint die lokale Replikation der Daten (siehe Abschnitt 3.5.1.3).

Die durchgeführten Messungen fanden auf dem Mehrrechnersystem mit zwei und vier beteiligten Rechnern statt. Es wurde ein relativ kleiner Datenbankcache von 400 Rahmen mit jeweils 32 KByte gewählt, um die schon in Abschnitt 4.2.1 be-

schriebenen Auswirkungen von billigen Festplattenzugriffen und teuren Zugriffen auf entfernte Datenbankcaches gering zu halten.

4.3.1.1 OLTP-Last

Abbildung 4.8 zeigt die erzielten Ergebnisse unter der OLTP-Version TPC–C^{N+P} des TPC–C Benchmarks.

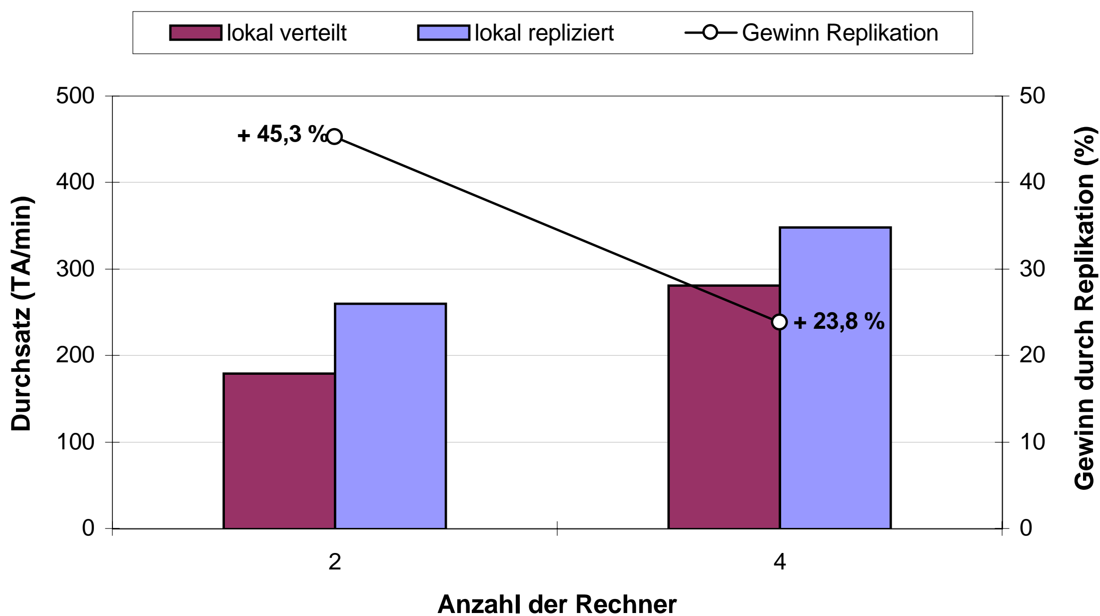

Abbildung 4.8: Gewinn durch Replikation – OLTP-Last

Allgemein läßt sich erkennen, daß das Ausschalten von NFS zu einer Verbesserung der Performanz führt. Das war zu erwarten, da dadurch eine Verbesserung der E/A-Zeiten und eine Netzentlastung möglich ist. Die Performanzgewinne bei der 2-Rechnerkonfiguration sind größer als bei der 4-Rechnerkonfiguration.

In der 2-Rechnerkonfiguration wird eine Leistungssteigerung von 45,3 % erreicht. In der 4-Rechnerkonfiguration fällt der Gewinn mit 23,8 % geringer aus.

Die Verbesserung beim Übergang von lokaler Verteilung auf lokale Replikation ist auf die Reduzierung der Kosten für das Einlesen von Seiten in den Systempuffer zurückzuführen. Beispielsweise sinkt die durchschnittliche Dauer eines Auftrags an die E/A-Komponente bei vier Rechnern von 16,5 ms auf 8,3 ms. Ähnliche Zahlen gelten auch für die 2-Rechnerkonfiguration.

Der geringere Vorteil der lokalen Replikation in der 4-Rechnerkonfiguration wurde

schon in Abschnitt 4.2 besprochen und resultiert aus dem größeren Datenbankcache und den durch Betriebssystemcaches zu billig gewordenen lokalen Festplattenzugriffen.

4.3.1.2 TPC–C-Last

Die Leistungsunterschiede unter TPC–C-Last werden in Abbildung 4.9 gezeigt.

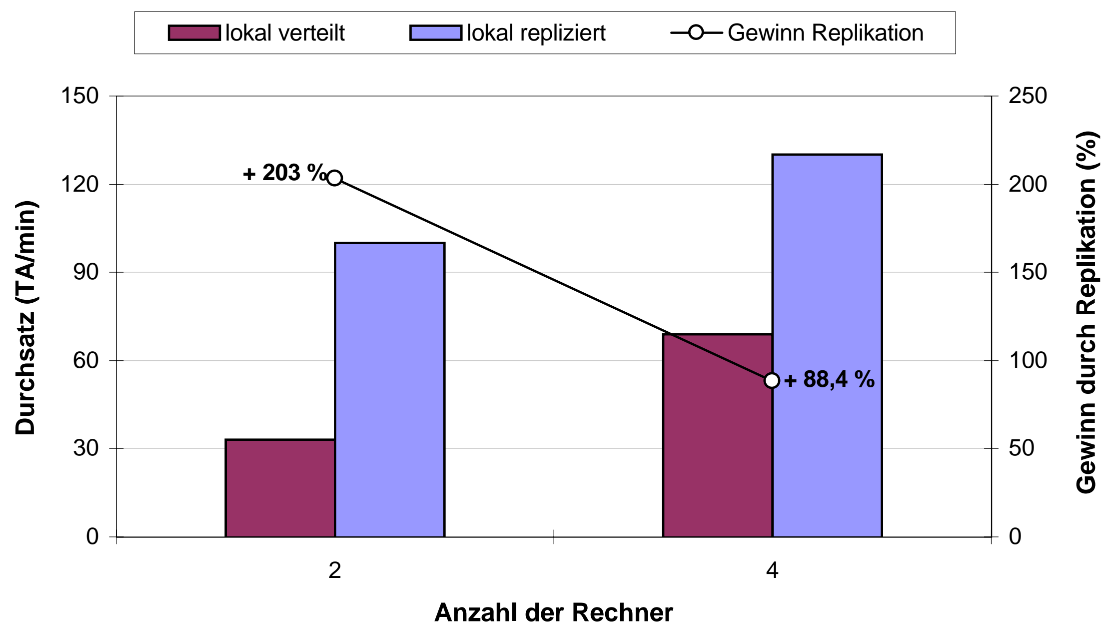

Abbildung 4.9: Gewinn durch Replikation – TPC–C-Last

Das Ausschalten von NFS führt hier zu deutlich höheren Performanzgewinnen als unter OLTP-Last. Das liegt daran, daß diese Last viel E/A-intensiver ist als die OLTP-Last.

Auch die erreichten Ersparnisse in den Zugriffszeiten sind viel höher. So sinkt die durchschnittliche Dauer eines Auftrages an die E/A-Komponente bei vier Rechnern von 12,5 ms auf 2,21 ms. Die extrem niedrigen Zeiten zum Laden einer Seite lassen sich damit erklären, daß die meisten Plattenzugriffe durch Relationen-Scans produziert werden, die ein sequentielles Lesen von der Festplatte ermöglichen.

Ähnlich wie in der Messung mit der OLTP-Last sind die Gewinne für die 4-Rechnerkonfiguration aus oben genannten Gründen deutlich geringer.

4.3.2 Zusammenfassung

In dieser Meßreihe war es möglich, die Auswirkungen von NFS auf die Performanz von MIDAS zu quantifizieren. Dazu wurde versucht, einem “echten” Shared-Disk-System nahezukommen, indem ein NFS-freies System simuliert wurde.

Dabei ergab sich, daß der Einfluß von NFS von der Art der Last abhängig ist und mit über 200 % Leistungsunterschied sehr deutlich ausfallen kann. Die Performanzgewinne beim Ausschalten von NFS resultieren hauptsächlich aus der Reduzierung der mittleren E/A-Zugriffszeiten und entsteht weniger durch die gleichzeitige Entlastung des Netzes.

4.3.3 Seiten und Subseiten

In diesem Abschnitt wird der Einfluß der Seitengröße auf die Systemperformanz analysiert. Um diesen Einfluß zu bestimmen, wurde die Anzahl der Subseiten[2] pro Seite und somit die Größe der Seite variiert. Dabei wurden vier Konfigurationen verwendet, die eine, zwei, vier beziehungsweise acht Subseiten pro Seite besaßen, so daß sich Seitengrößen von 8 KByte, 16 KByte, 32 KByte und 64 KByte ergaben. Die Anzahl der Rahmen im Datenbankcache wurde in allen Messungen so angepaßt, daß alle Meßkonfigurationen den gleichen Gesamtcache von 12 MByte zur Verfügung hatten.

Es wurde erwartet, daß größere Seiten Kosteneinsparungen in der E/A-Komponente des Systems bringen, da durch das Lesen größerer Seiten und damit von mehr Subseiten ein Prefetching-Effekt erzielt werden kann.

Setzen sich die Seiten aus weniger Subseiten zusammen, so sollten Einsparungen bei der internen Systemkommunikation erreicht werden.

Die folgenden Messungen wurden mit lokal replizierten Daten durchgeführt, so daß keine Zugriffe über NFS auf die Festplatten erfolgten. Dadurch wurden Nebenwirkungen von NFS-Caches sowie die Belastung des Netzes durch NFS-Zugriffe vermieden.

Bei allen vorgestellten Meßreihen wurde das Mehrrechnersystem mit zwei und vier Rechnern verwendet. Es wurden jeweils eine beziehungsweise zwei parallele Transaktionen pro Rechner ausgeführt.

4.3.3.1 OLTP-Last

Abbildung 4.10 zeigt die ermittelten Werte unter OLTP-Last mit TPC–C^{N+P}-Transaktionen.

[2]Die Subseiten haben dabei eine feste Größe von 8 KByte.

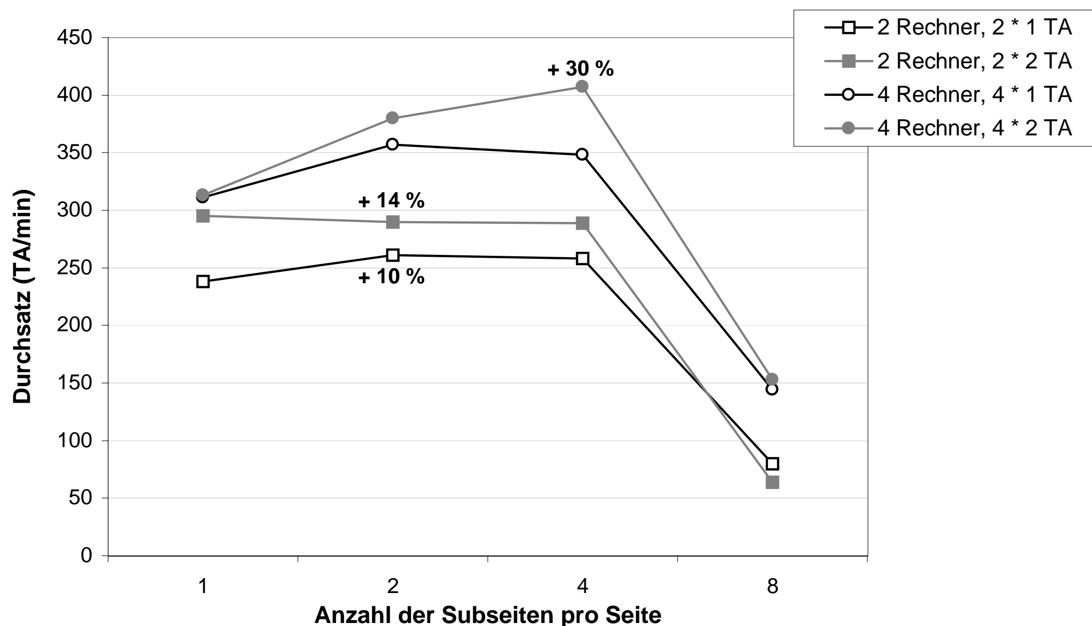

Abbildung 4.10: Verschiedene Seitengrößen unter OLTP-Last

Bei zwei verwendeten Rechnern ist der Durchsatz bei bis zu vier verwendeten Subseiten jeweils annähernd konstant.

In der 4-Rechnerkonfiguration dagegen steigt der Durchsatz an, wobei der höchste Durchsatzgewinn gegenüber einer Subseite mit 30 % bei vier Subseiten und zwei parallelen Transaktionen erreicht wird.

Werden acht Subseiten pro Seite verwendet, so fällt der Durchsatz in allen Konfigurationen sehr stark ab. Bei zwei parallelen Anfragen pro Rechner fällt dieser Leistungsverlust noch stärker aus.

Die niedrigen Leistungssteigerungen unter dieser Last ergeben sich aus der Tatsache, daß die meisten Zugriffe auf einzelne Daten auf zufällig ausgewählten Seiten erfolgen. Damit wird auf schon im Datenbankcache geladene Seiten selten öfter als einmal zugegriffen. Somit können größere Seiten mit mehreren Subseiten ihre Vorteile nicht zur Geltung bringen.

Der starke Abfall des Durchsatzes bei acht Subseiten pro Seite resultiert aus einer Netzüberlastung, die mit dieser Konfiguration erreicht wird. Diese Situation wird im folgenden Abschnitt 4.3.3.2 genauer beschrieben.

4.3.3.2 TPC–C-Last

Die Meßergebnisse unter TPC–C Benchmark Last werden in Abbildung 4.11 dargestellt.

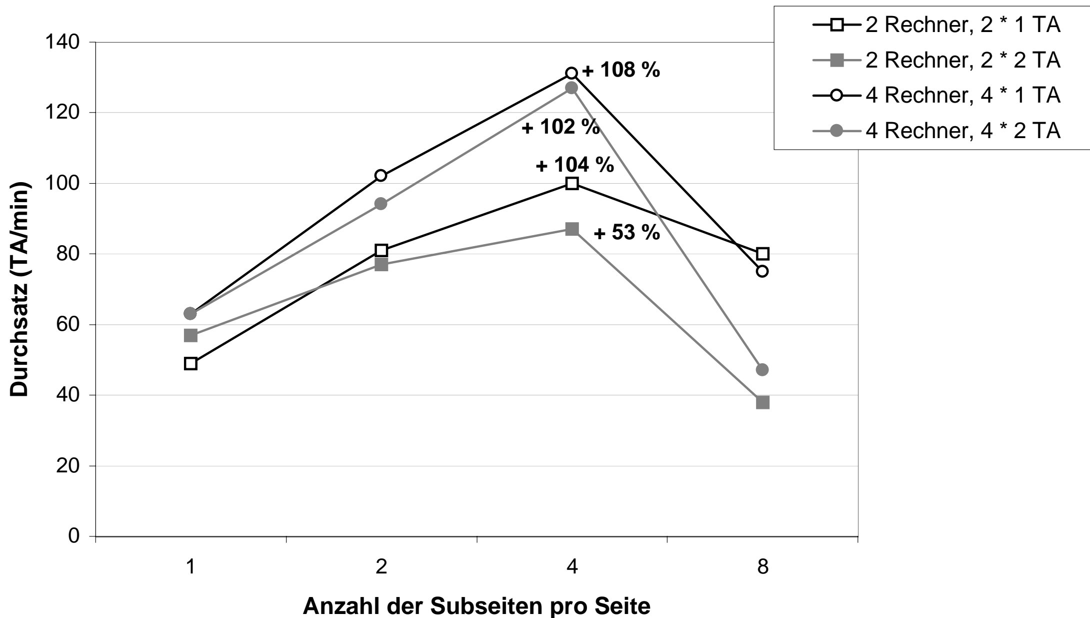

Abbildung 4.11: Verschiedene Seitengrößen unter TPC–C-Last

Alle Konfigurationen zeigen einen deutlichen Leistungsgewinn von bis zu 108 % beim Übergang von einer auf vier Subseiten pro Seite. Bei acht Subseiten pro Seite weisen alle Konfigurationen deutliche Leistungsverluste auf, die sogar unter die Durchsatzwerte bei der Verwendung von einer Subseite pro Seite fallen. Dabei treten diese Verluste deutlich stärker auf, wenn zwei parallele Anfragen pro Rechner ausgeführt werden.

Der bei der Verwendung von vielen Subseiten erhoffte Prefetching-Effekt kommt unter dieser Last stark zur Geltung. Dies liegt hauptsächlich daran, daß viele Datenanforderungen, wegen der häufigen Relationen-Scans unter dieser Last, sequentiell erfolgen und somit die geladenen Daten vollständig genutzt werden. Damit werden interne E/A-Systemkosten und externe E/A-Zugriffe verringert, die einen großen Anteil an den Ausführungskosten ausmachen. Diese E/A-Systemkosten werden mit der Benutzung größerer Seiten deutlich reduziert.

Die Unterteilung der Seiten in Subseiten kann die Kosten der internen Kommunikation weiter reduzieren, da bei schreibender Last zwischen den Rechnern öfter

die kleineren Subseiten als die ganzen Seiten verschickt werden. Bei dieser nur lesenden Last spielen Ersparnisse dieser Kategorie allerdings keine Rolle.

Wie der Abbildung 4.11 zu entnehmen ist, ist die Leistungsverbesserung bei zwei parallelen Transaktionen pro Rechner etwas geringer, da die erhöhte Anzahl paralleler Anfragen eine häufigere Repositionierung des Lesekopfes der lokalen Festplatte verursacht, wodurch das sequentielle Lesen gestört wird.

Der starke Abfall des Durchsatzes bei acht Subseiten pro Seite in allen Konfigurationen wird durch eine Überlastung des Netzes in dieser Situation verursacht. Alle Nachrichten der Cacheverwaltung, die ganze Seiten versenden, werden in diesem Fall 64 KByte groß, wodurch in dieser Systemkommunikation die Leistungsgrenze des Netzwerkes erreicht wird. Bei zwei parallelen Anfragen pro Rechner verstärkt sich dieser Effekt naturgemäß. Die Laufzeit kleiner Aufträge der Cacheverwaltung, die zwei kleine Kontrollnachrichten erzeugen (wie beispielsweise die Anfrage an und die Rückantwort des globalen Sperrverwalters, (GLM-Request)), steigen von 0,7 ms auf 37 ms an. Bei Aufträgen, die als Antwort eine komplette Seite senden (STORE-Page), verlängert sich die Laufzeit von 30 ms auf 130 ms.

4.3.3.3 Zusammenfassung

In dieser Meßreihe wurde die Anzahl der Subseiten pro Seite und somit die Seitengröße variiert. Unter geeigneter Last, die genügend Prefetching ermöglicht, können größere Seiten zu signifikanten Leistungsgewinnen von über 100 % führen. Die Performanzgewinne sind hauptsächlich auf geringere interne E/A-Systemkosten und weniger Plattenzugriffe, die eine Neupositionierung des Lesekopfes verursachen, zurückzuführen.

Seiten mit vier Subseiten und einer Gesamtgröße von 32 KByte haben bei den vorgestellten Messungen in diesem Abschnitt die besten Ergebnisse erzielt. Aus diesem Grund ist dies die Standardeinstellung der Seitengröße in MIDAS.

4.3.4 Datenbankcache und Festplattenzugriffe

Diese Messung befaßt sich mit der Bestimmung der Effizienz des Caches in MIDAS. Verwendet werden dazu die aus den vorherigen Messungen bekannten lesenden Versionen des TPC–A und TPC–C Benchmarks. Zusätzlich werden die durchschnittlichen Zeiten für das Lesen einer MIDAS-Seite von der Festplatte gemessen.

Zur Messung wurde das 4-Prozessorsystem mit vier parallelen Transaktionen verwendet. Die Datenbanken waren auf die vier lokalen Festplatten verteilt.

4.3.4.1 OLTP-Last

Verändert wurde die Größe des zur Verfügung stehenden Caches von 1 MByte bis zu 50 MByte, größere Werte wurden vom Betriebssystem nicht zur Verfügung gestellt.

In Abbildung 4.12 ist die Anzahl der wirklich von der Festplatte gelesenen MIDAS-Seiten bei der Transaktion TPC–A^{RO} dargestellt.

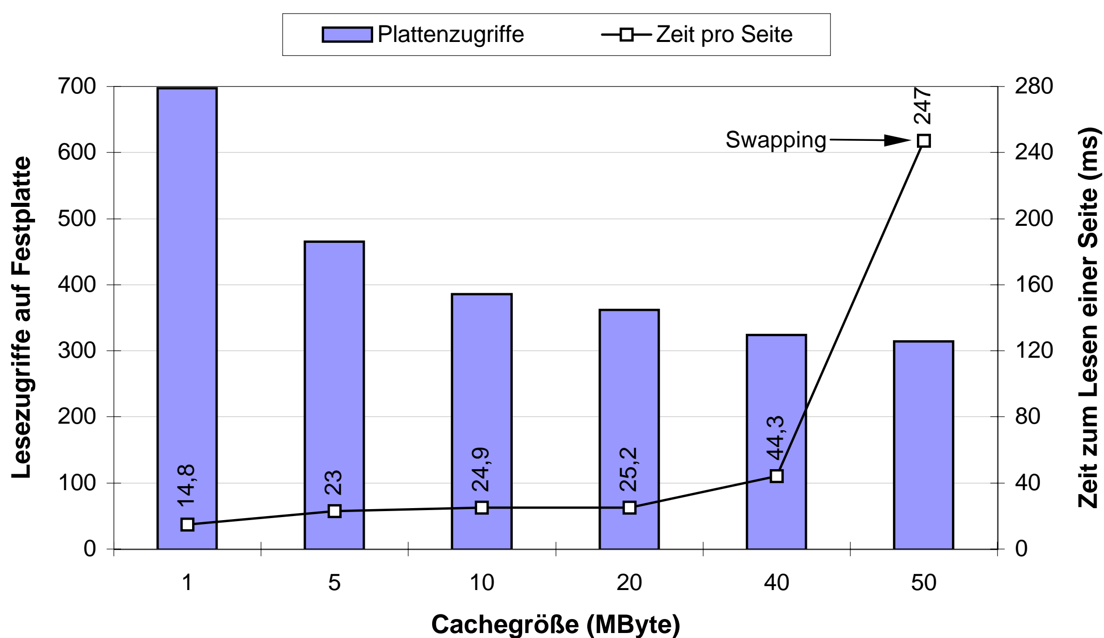

Abbildung 4.12: Anzahl Festplattenzugriffe und mittlere Lesezeit einer Seite bei TPC–A^{RO} und unterschiedlichen Cachegrößen

Diese Transaktion führt, wie in Anhang D erwähnt, drei SELECT-Statements auf zwei kleine und eine große Relation (`account`) aus. Der Zugriff erfolgt jeweils über den Primärschlüssel. Es wird genau der Wert eines Attributs aus dem Tupel gelesen[3]. Dabei befinden sich die beiden kleinen Relationen nach dem ersten Zugriff komplett im Cache und werden im folgenden bei allen Cachegrößen auch von dort benutzt. Wirklich relevant bleibt damit nur der Zugriff auf die große Relation, die eine Größe von 210 MByte besitzt. Hier wird zuerst auf eine B-Baum-Seite zugegriffen, dann erfolgt das Lesen der eigentlichen Information. Somit ergibt sich

[3] Jeweils die Kontostände der drei Einheiten Bank, Zweigstelle und Konto (vergleiche Anhang B).

rein rechnerisch für die vier verwendeten Applikationen, die jeweils 100 Transaktionen ausführen, eine Zahl von etwa 800 notwendigen Zugriffen auf die große Relation account.

TPC–A^{RO}	**1 MB**	**5 MB**	**10MB**	**20 MB**	**40 MB**	**50 MB**
# Plattenzugriffe	697	465	386	362	324	314
Lesezeit/Seite (ms)	14,8	23,0	24,9	25,2	44,3	247

Tabelle 4.2: Seitenzugriffsstatistik verschiedener Cachegrößen, TPC–A^{RO}

Wie aus Tabelle 4.2 zu entnehmen und nicht anders zu erwarten ist, sinkt die Anzahl der Festplattenzugriffe mit der Vergrößerung des Caches. Schon bei einer Größe von 10 MByte, wie sie in den meisten Messungen verwendet wurde, befinden sich fast alle B-Baum-Seiten im Cache[4]. Eine weitere Vergrößerung bringt bei dieser Transaktion nur noch geringe Verbesserungen. So erzielt die Verfünffachung von 10 MByte auf 50 MByte nur 70 Treffer mehr. Berechnet man die Treffer pro MByte bei 800 möglichen Treffern, so ergeben sich 69 Treffer bei 10 MByte. Bei 50 MByte fällt dieser Wert auf 9 Treffer/MByte.

Gleichzeitig wurde der den Prozessen zur Verfügung stehende gesamte Hauptspeicher von 128 MByte durch die Erhöhung des Datenbankcaches auf 40 MByte beziehungsweise 50 MByte so knapp, daß sich die Zeit für das Lesen einer Seite verzehnfachte. Dies ist auf Seitenauslagerungen des Betriebssystems (Swaping) zurückzuführen. Die kürzeren Lesezeiten bei 1 MByte können durch die Nummern geladener Seiten erklärt werden, die auf eine sequentielle Lage innerhalb der zugehörigen Datei schließen läßt. Bei nur 1 MByte Cache müssen fast alle B-Baum-Indexseiten jedesmal gelesen werden. Anschließend wird noch die Blattseite gelesen, welche physisch daneben plaziert ist. Deshalb können hier fast immer zwei Leseoperationen durchgeführt werden, bei denen nur einmal positioniert wird. Dadurch fällt der Durchschnitt auf den gezeigten Wert.

[4]Dies wurde bei näherer Betrachtung der Meßwertdateien bestätigt, ist jedoch nicht aus der Tabelle ersichtlich. Außerdem wird diese Tatsache sofort einsichtig, wenn man bedenkt, daß TPC–A^{RO} auf eine Verfehlung des Caches ausgerichtet ist und zufällige Werte aus einem großen Intervall abgefragt werden.

4.3.4.2 TPC–C-Last

Abbildung 4.13 und Tabelle 4.3 zeigen die Werte für die Transaktion TPC–C. Diese Transaktion liest mehr Seiten als im Cache gehalten werden können. Eine Vergrößerung des Caches bringt hier eine Verbesserung, da dann einige Seiten nicht mehr verdrängt werden und so nicht jedesmal von Festplatte geladen werden müssen. Entsprechend steigt hier die Effizienz auch jenseits von 10 MByte weiter an. Eine Sättigung tritt in diesem Fall zwischen 20 MByte und 40 MByte ein, danach verläuft die Effizienzkurve flacher.

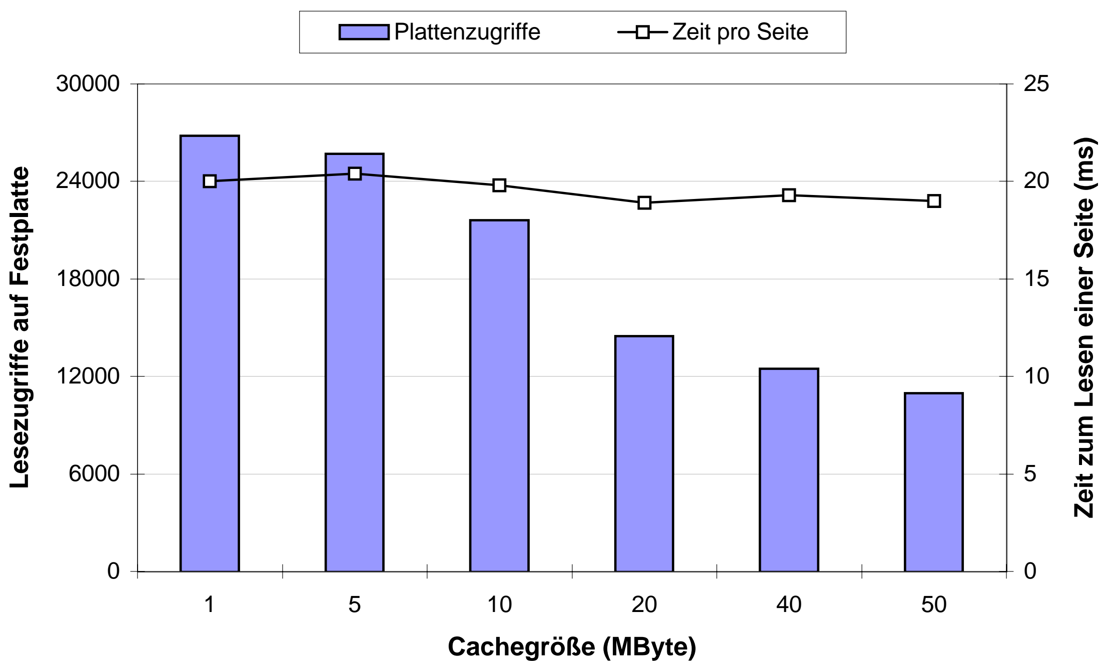

Abbildung 4.13: Anzahl der Festplattenzugriffe und mittlere Lesezeit einer Seite bei TPC–C und unterschiedlichen Cachegrößen

TPC–C	**1 MB**	**5 MB**	**10MB**	**20 MB**	**40 MB**	**50 MB**
# Plattenzugriffe	26793	25638	21608	14490	12487	10978
Lesezeit/Seite (ms)	20,0	20,4	19,8	18,9	19,3	19,0

Tabelle 4.3: Seitenzugriffsstatistik verschiedener Cachegrößen, TPC–C

4.3.4.3 Zusammenfassung

Wie erwartet führt eine Erhöhung der Cachegröße zu einer Verringerung der Anzahl an Festplattenzugriffen und somit zu einer Verkürzung der mittleren Transaktionslaufzeit und zu einer Steigerung des Durchsatzes. Die Stärke dieses Verhaltens ist lastabhängig, bei TPC–A^{RO} tritt eine Sättigung bereits bei ca. 10 MByte ein, bei TPC–C erst bei ca. 30 MByte. Größere Caches bringen nur noch leichte Vorteile. Bei der Vergrößerung muß auf die Systemgrenzen Rücksicht genommen werden, da sonst starke Leistungseinbußen zu verzeichnen sind (z. B. durch Swaping).

Abschließend sei bemerkt, daß auf einem Mehrrechnersystem zusätzlich zu den Auswirkungen des Datenbankcaches unerwünschte Cacheeffekte durch den NFS-Cache des Betriebssystems auftreten (siehe Abschnitt 3.5.2).

4.4 Leistungsgrenzen

Dieser Abschnitt beschäftigt sich mit den Leistungsgrenzen von MIDAS. Alle getroffenen Aussagen berücksichtigen die Implementierung von MIDAS, aber naturgemäß auch die zur Verfügung stehende Hardware.

Aussagen über die Leistungsabhängigkeit von der Datenverteilung wurden bereits in Abschnitt 4.3.1 getroffen.

Die maximale Seitengröße, die im System verwendet werden sollte, liegt bei 32 KByte. Größere Seiten führen zu einer starken Überlastung des Netzwerkes durch den cacheinternen Seitenaustausch.

Die maximale Systemgröße wurde nicht über vier Rechner hinaus skaliert, da für die Analysen keine gleichwertigen Rechner zur Verfügung standen und auch lange Zeit keine leistungsfähigen Netzanschlüsse.

Bisher wurden noch keine Analysen über die Leistungsgrenzen bei der Erhöhung der Last durch höhere Parallelitätsgrade getroffen. Diese folgen im nächsten Abschnitt.

Im Abschnitt danach werden Überlegungen angestellt, wie die gegebenen Leistungsgrenzen überschritten werden können.

4.4.1 Maximaler Parallelitätsgrad

Dieser Abschnitt zeigt die Leistungsgrenzen des Systems auf, die bei der Erhöhung der Anzahl paralleler Anfragen entstehen. Für die Analysen wurden Anfragen mit OLTP-Charakteristika gewählt, da diese nicht durch die Festplattenleistung

beschränkt sind, und so durch einen höheren Parallelitätsgrad besserer Durchsatz zu erwarten ist.

4.4.2 Mehrprozessorsystem

Auf dem Mehrprozessorsystem wurden alle vier Prozessoren aktiviert und die lesende Version des TPC–A Benchmarks ausgeführt, TPC–A^{RO}. Dabei wurde die Anzahl paralleler Transaktionen variiert. Abbildung 4.14 zeigt die ermittelten Ergebnisse.

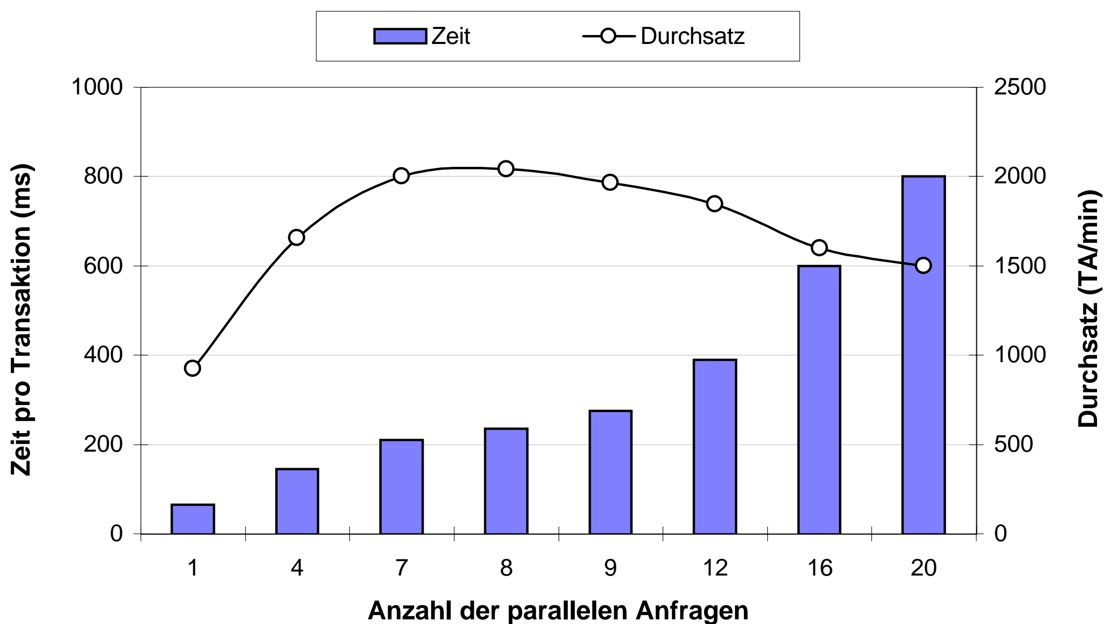

Abbildung 4.14: Maximaler Durchsatz auf Mehrprozessorsystem

Der höchste Durchsatz wird bei acht parallelen Anfragen erreicht. Bei höherem Parallelitätsgrad fällt der Durchsatz wieder leicht ab.

Die optimale Anzahl an parallelen Anfragen ist damit doppelt so hoch wie die Anzahl der zur Verfügung stehenden Prozessoren. Die Tatsache, daß bei dieser CPU-intensiven Last das Optimum höher als die Prozessorenanzahl ist, liegt daran, daß sich immer Prozesse in Wartesituationen auf Ein- oder Ausgabe oder andere, interner Ressourcen befinden. Während dieser Wartezeiten werden die anderen Anfragen durch die Prozessoren bedient, wodurch eine bessere Ressourcenauslastung erzielt wird.

4.4.3 Mehrrechnersystem

Auf dem Mehrrechnersystem wurden die Anfragen der OLTP-Version TPC–C^{N+P} des TPC–C Benchmarks als Last verwendet. Die Daten waren lokal verteilt. Die ermittelten Ergebnisse sind in Abbildung 4.15 dargestellt.

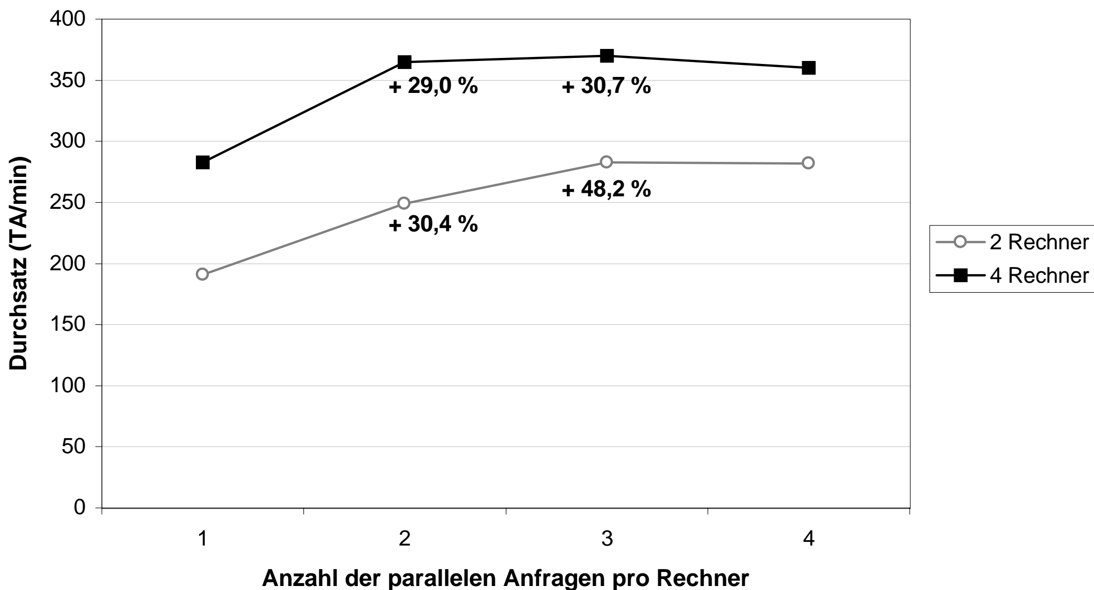

Abbildung 4.15: Maximaler Durchsatz auf Mehrrechnersystem

Sowohl bei zwei als auch bei vier verwendeten Rechnern wird der optimale Durchsatz des Gesamtsystems bei drei parallelen Anfragen pro Rechner erreicht.

Auch bei dieser Konfiguration wird der optimale Durchsatz bei mehreren parallelen Anfragen pro Prozessor erreicht. Die Gründe hierfür sind dieselben wie beim Mehrprozessorsystem. Durch Wartesituationen, die in den Prozessen des Ausführungsystems entstehen, ergeben sich freie CPU-Kapazitäten, die durch die zusätzlichen Anfragen genutzt werden können, womit eine bessere Auslastung der vorhandenen Ressourcen erreicht werden kann.

Insgesamt sind die Wartezeiten auf dem Mehrrechnersystem etwas länger als auf dem Mehrprozessorsystem, da zusätzlich längere Wartezeiten auf Nachrichten über das Netzwerk und NFS-Festplattenzugriffe hinzukommen. Daraus ergibt sich, daß die optimale Anzahl paralleler Anfragen pro Prozessor mit drei noch höher als beim Mehrprozessorsystem liegt.

4.4.4 Lokalitätsbasiertes Transaktionsrouting

In diesem Kapitel wurden bisher Untersuchungen präsentiert, die sich mit drei verschiedenen Aspekten des MIDAS-Systems befaßten, der Skalierbarkeit, der Beeinflussung der Performanz durch NFS und durch E/A-relevante Fragen. Nachdem die Leistungsgrenzen des Systems aufgezeigt wurden, stellt sich die Frage, wie noch weitere Leistungssteigerungen zu erzielen sind.

Der in diesem Abschnitt verfolgte Ansatz schlägt ein lokalitätsbasiertes Transaktionsrouting vor, um eine Reduzierung der Inter-Rechnerkommunikation zu erreichen. Dabei wird versucht, in einem Mehrrechnersystem auf Softwarebasis Zugriffscharakteristika analog zu einem Shared-Nothing-System entstehen zu lassen. Dieses Ziel wird in zwei Schritten erreicht.

Zuerst werden bei gegebener Datenbank und gegebenen Transaktionen geeignete disjunkte Datenbereiche gesucht, so daß die Transaktionen bei ihrer Ausführung möglichst viele Daten aus nur einem der Bereiche berühren. Alle Transaktionen, die bevorzugt auf denselben Datenbereich zugreifen, werden auf demselben Rechner zur Ausführung gebracht. Durch dieses Transaktionsrouting wird auf jedem Rechner häufiger auf gleiche Daten zugegriffen, die sich dann auch häufiger schon im lokalen Datenbankcache befinden. Durch diese erhöhte lokale Cachetrefferrate sind eine niedrigere Netzbelastung und schnellere Ausführungszeiten zu erwarten.

In einem zweiten Schritt wird die Pufferverwaltung angepaßt. Die Zuständigkeiten der Pufferverwaltung bei Sperrverwaltung und Kohärenzkontrolle werden gemäß den im ersten Schritt ausgewählten Bereichen verteilt. Dies bedeutet, daß die Besitzrechte der Kohärenzkontrolle an den Datenseiten eines Bereichs sowie die Zuständigkeit für die Sperrvergabe für alle Seiten des Bereichs auf den Rechner übertragen werden, auf den später die entsprechenden Transaktionen gesendet werden. Dadurch soll erreicht werden, daß möglichst viele Anfragen der Synchronisations- und Kohärenzkontrolle lokal bearbeitet werden können. Dadurch kann deutlich an Inter-Rechnerkommunikation gespart werden.

Anders als bei einem Shared-Nothing-System muß in diesem Fall keine komplette Trennung der Daten stattfinden, da nach wie vor die Möglichkeit besteht, Daten über den gemeinsamen Datenbankcache auszutauschen. In einem Shared-Nothing-System müßte gegebenenfalls eine geeignete, teure Repartitionierung stattfinden.

Bei den hier vorgestellten Ergebnissen wurde versucht, den maximalen Gewinn, der aus solch einem lokalitätsbasierten Transaktionsrouting resultiert, auf dem 4-Rechnersystem zu quantifizieren (siehe auch [Bozas 98]).

Dazu wurde eine Last mit OLTP-Charakteristika entworfen, für die die Vergabe der Zuständigkeiten in MIDAS leicht zu implementieren war. Die verwendete Datenbank entsprach einer TPC–A-Datenbank (siehe Anhang C) mit vier Hauptre-

lationen (account). Jede Transaktion bestand aus acht Anfragen, die jeweils auf zwei Datenseiten zugreifen. Sechs der Anfragen, also 75 %, greifen auf dieselbe der vier Hauptrelationen zu. Die beiden anderen Anfragen betreffen eine andere Hauptrelation.

Transaktionen mit Zugriffshäufung auf dieselbe Hauptrelation wurden auf denselben Rechner gesendet.

Der Datenbankcache wurde so groß gewählt, daß er die kompletten Relationen aufnehmen kann, damit keine negativen Effekte durch die Datenverteilung auftreten und der maximale Gewinn ohne das Warten auf E/A-Vorgänge ermittelt werden kann. Auf jedem Rechner war jeweils eine Transaktion aktiv. Insgesamt wurden 500 Transaktionen pro Rechner ausgeführt. Abbildung 4.16 zeigt die erzielten Ergebnisse.

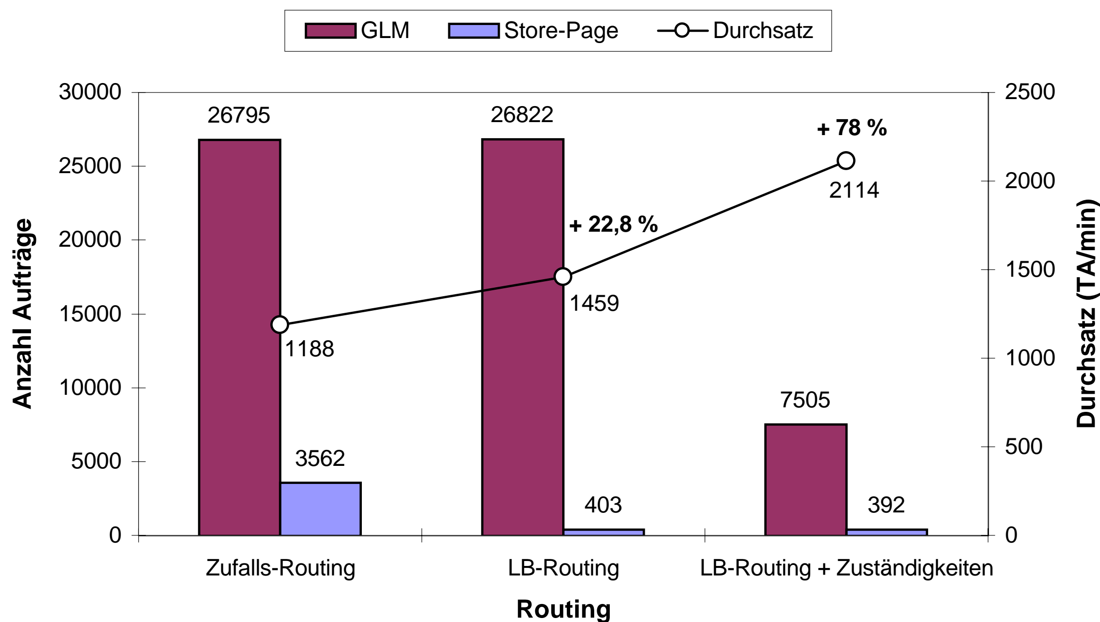

Abbildung 4.16: Lokalitätsbasiertes Transaktionsrouting

Es wurden die beiden oben beschriebenen Schritte einzeln nachvollzogen. Im ersten Schritt wurden nur die Transaktionen gemäß ihren Zugriffscharakteristika auf die vier vorhandenen Rechner verteilt. Dabei konnte eine Leistungssteigerung von 22,8 % gegenüber der normalen Ausführung mit Zufalls-Routing erzielt werden. Dieser Gewinn stammt wie erwartet nur aus der höheren Zahl lokaler Cachetreffer, wodurch die Zahl entfernter Cachetreffer (Store-Page, siehe Abschnitt 3.6) stark absinkt. Aus der Abbildung 4.16 ist zu entnehmen, daß die

Anzahl von Store-Page-Nachrichten, die diese Seiten übertragen, von 3562 auf 403 sinkt. Gleichzeitig bleibt die Anzahl der Nachrichten von Synchronisations- und Kohärenzkontrolle (GLM-Request) bei fast 27000 konstant.

In einer weiteren Messung wurden diese Transaktionen mit angepaßten Zuständigkeiten von Synchronisations- und Kohärenzkontrolle durchgeführt. Erwartungsgemäß sank die Zahl der GLM-Requests von 26795 auf 7505 deutlich ab. Der Leistungsgewinn fiel mit 78 % entsprechend stark aus. Die Zahl der entfernten Cachetreffer wurde durch diesen zweiten Schritt nicht nochmals gesenkt, was auch nicht erwartet worden war.

Durch lokalitätsbasiertes Transaktionsrouting ist also nochmals eine deutliche Leistungssteigerung erreichbar, wenn die Transaktionen entsprechende Lokalitätsbereiche aufweisen.

4.5 Zusammenfassung

In diesem Kapitel wurden Leistungsanalysen vorgestellt, die Aussagen über die Implementierung von MIDAS und insbesondere über die Effizienz des Ausführungsystem treffen.

Zuerst wurde MIDAS mit seinem Vorgänger TransBase verglichen. TransBase zeigt in diesem Vergleich leichte Effizienzvorteile auf. Dies ist zurückzuführen auf den Mehraufwand für die Parallelisierung des Systems durch mehrere Prozesse sowie komplexere Algorithmen in MIDAS. Bei einem höheren Parallelitätsgrad von vier bis sechs und mehr aktiven Prozessoren kann MIDAS diesen Nachteil aber wieder ausgleichen, da dort seine Eigenschaften als ein auf die Parallelverarbeitung ausgelegtes System zum Tragen kommen.

Die Skalierungseigenschaften sind gut. Sie können in einer Mehrrechnerkonfiguration bei einer Verdoppelung der Systemgröße zwischen 75 % und 95 % erreichen, falls eine Konfiguration gewählt wird, in der keine unerwünschten Nebeneffekte durch Betriebssystemcaches auftreten und in der entfernte Cachezugriffe nicht teurer als Festplattenzugriffe sind.

Auch unter einer Last, die Intra-Transaktionsparallelität beinhaltet, sind die Skalierungseigenschaften gut und zeigen, daß das Ausführungsystem diese Parallelitätsform gut unterstützt.

Auf einem Mehrprozessorsystem sind die Skalierungswerte bei geeigneter Last mit 40 % bis 50 % zwar geringer, aber ebenfalls gut. Auf diesem System wird die Skalierbarkeit durch den vermehrten Zugriff auf gemeinsam genutzte Ressourcen stärker begrenzt.

Durch Simulation einer NFS-freien Systemkonfiguration, die viele billige Festplattenzugriffe ermöglicht und somit nicht das Netzwerk belastet, wurde gezeigt, daß

diese einem Shared-Disk-System ähnlichere Architektur, weitere Leistungssteigerungen erzielen kann. In MIDAS kann eine solche Konfiguration allerdings nur durch lokale Replikation erreicht werden, wodurch nur lesende Transaktionen bearbeitet werden können. Außerdem erzeugt sie die ungewünschte Situation, daß Festplattenzugriffe billiger als entfernte Cachezugriffe sind.

Die Verwendung größerer Seiten und damit größerer Einheiten der E/A-Komponente erzielt beim Übergang von 8 KByte-Einheiten auf 32 KByte-Einheiten, Leistungsvorteile von bis zu 100 %. Bei noch größeren Einheiten wird allerdings schnell eine starke Netzüberlastung erreicht, was deutliche Leistungseinbußen mit sich bringt.

Optimaler Durchsatz läßt sich mit zwei bis drei parallelen Anfragen pro Prozessor erreichen. In diesem Fall werden die vorhandenen CPU-Kapazitäten optimal genutzt.

Schließlich wurde eine Möglichkeit aufgezeigt, die Systemleistung weiter zu erhöhen, indem durch lokalitätsbasiertes Transaktionsrouting die Anzahl von Nachrichten der Pufferverwaltung stark reduzieren wird. Mit dieser Technik sind im günstigsten Fall Leistungssteigerungen von bis zu 78 % erreichbar.

Um einem "echten" Shared-Disk-System näherzukommen, müßte ein besseres Netzwerk und eine direkte Anbindung der Festplatten an die einzelnen Rechner zur Verfügung stehen (siehe Abschnitt 3.5.2).

Insgesamt bestätigen aber die vorgestellten Leistungsanalysen die erfolgreiche Implementierung von MIDAS.

Kapitel 5

Aktivitätskontrolle und Modell zur Fehlerbehandlung

5.1 Motivation

Wird in einem parallelen Datenbanksystem auch Intra-Transaktionsparallelität unterstützt und nehmen damit an der Bearbeitung einer Transaktion mehrere Systemkomponenten als Ausführungseinheiten gleichzeitig teil, so muß ein stark an die Parallelverarbeitung angepaßtes Ausführungsmodell verwendet werden. Dabei werden beispielsweise, wie in Abschnitt 2.3 beschrieben, Parallelitätsformen wie Daten- und Pipeline-Parallelität zur Verfügung gestellt. Im Verlauf der Parallelisierung wird zum einen viel Aufwand in die Erzeugung der parallelen Verarbeitungspläne investiert, zum anderen werden viele Systemressourcen für die eigentliche, parallele Bearbeitung einer Anfrage belegt.

Tritt im Laufe der Bearbeitung ein Fehler auf, so sind, falls keine entsprechenden Vorkehrungen getroffen wurden, die vom System für diese Anfrage bis zu diesem Zeitpunkt erzeugten Ergebnisse verloren. Somit stellt sich die Frage, wie in einem solchen Fehlerfall möglichst große Teile der bereits verrichteten Arbeit gerettet und wiederverwertet werden können. Insbesondere ist es interessant, ob aus der durch die Parallelisierung eingeführten Strukturierung und Aufteilung auf mehrere Ausführungseinheiten Gewinn für eine derartige Fehlerbehandlung gezogen werden kann [Zim 97].

In diesem Kapitel wird ein Modell zur Fehlerbehandlung in parallelen Transaktionen, das FPT-Modell, als Erweiterung existierender Transaktionskonzepte vorgestellt, das die speziellen Eigenschaften eines stark auf die Parallelverarbeitung ausgerichteten Ausführungsmodells berücksichtigt, wie die verfügbaren Parallelitätsarten und die Strukturierung der Ausführungseinheiten.

Zuerst werden in Abschnitt 5.2 mögliche Fehlerfälle in Datenbanksystemen be-

schrieben. Danach folgt die Vorstellung einiger bereits existierender Transaktionskonzepte, die anfänglich vielversprechend erschienen, die gewünschten Anforderungen erfüllen zu können.

Anschließend wird in Abschnitt 5.3 und 5.4 eine genauere Analyse der von einem derartigen Transaktionsmodell geforderten Eigenschaften ermittelt und beschrieben, welche Fehlerfälle es abdecken können soll.

Bei einem ab Abschnitt 5.5 durchgeführten Vergleich mit den existierenden Konzepten zeigt sich, daß keines der Modelle den gestellten Anforderungen gerecht werden kann und aus verschiedenen Gründen auch nicht leicht an diese Anforderungen anpaßbar ist. Somit ist es notwendig, für das gewünschte Verhalten der effizienten und flexiblen Behandlung von Fehlerfällen ein eigenes Modell zu definieren, welches eine Erweiterung existierender Transaktionskonzepte darstellt. Dieses Modell, das FPT-Modell, wird im darauffolgenden Abschnitt 5.7 vorgestellt wird.

Kapitel 6 enthält eine detaillierte Darstellung, wie das vorgestellte FPT-Modell auf ein bestehendes Datenbanksystem abgebildet werden kann und beschreibt, wie eine Implementierung des Modells im Datenbanksystem MIDAS aussehen würde. Abschließend werden Resultate vorgestellt, die die Effizienz des entwickelten Konzeptes belegen.

5.2 Fehlerfälle und existierende Konzepte für Transaktionsmodelle

Zu den Grundeigenschaften, die ein Datenbanksystem bieten soll, gehören das effiziente Ausführen von Benutzeraufträgen und Fehlertoleranz. Die Forderung nach Effizienz entspricht dabei dem Wunsch nach möglichst hohem Durchsatz und kurzer Antwortzeiten auf Benutzeranfragen. Parallelisierung ist hier einer der möglichen Wege, diese Effizienz zu erzielen. Zudem wird von vielen Datenbanksystemen eine hohe Verfügbarkeit verlangt, was bedeutet, daß die Wiederanlaufzeit nach Systemausfällen möglichst kurz ist. Zur Bewältigung dieser Anforderungen kennen Datenbanksysteme das Konzept der Transaktion, das insbesondere einen Schutz vor Verletzung der Datenbankkonsistenz durch auftretende Fehler bietet.

Der folgende Abschnitt 5.2.1 zeigt eine Klassifizierung möglicher Fehler in einem Datenbanksystem. Die darauf folgenden Abschnitte stellen verschiedene, bekannte Transaktionskonzepte vor, die in heutigen Systemen zur Fehlerbehandlung eingesetzt werden.

5.2.1 Fehlerklassen

Fehler in einem Datenbanksystem werden nach ihren Ursachen kategorisiert. Jede der dabei entstehenden Fehlerklassen zieht andere Maßnahmen der Fehlerbehandlung nach sich.

1. Mediafehler:

   Mediafehler liegen vor, wenn Sekundärspeicherinhalte, also Teile des stabilen Speichers, verlorengehen. Sie können beispielsweise durch "Plattencrashs" oder fehlerhafte Betriebssystem-Routinen für das Schreiben auf Platte verursacht werden.

2. Systemfehler:

   Dies sind Fehler im Datenbanksystemprogrammcode, im Betriebssystem oder in der Hardware, auf der das Datenbanksystem arbeitet, incl. Stromausfällen. Sie haben den Effekt, daß Hauptspeicherinhalte, insbesondere Inhalte des Datenbankpuffers, verlorengehen.

3. Transaktionsfehler:

   Transaktionsfehler sind alle innerhalb der Ausführung einer Transaktion auftretenden Fehler, die diese an ihrem erfolgreichem Beenden hindern. Darunter fallen Verklemmungen zwischen Transaktionen, Ausfälle sowie Funktionsstörungen von Transaktionskomponenten, wie Prozessen und Kommunikationskanälen. Bei parallelen Datenbanksystemen können beispielsweise auch Verklemmungen innerhalb der Parallelausführung einer Transaktion auftreten (siehe auch Abschnitt 2.4.9).
   Semantik- bzw. Logikfehler, die durch die Applikation verursacht werden, fallen nicht darunter.

5.2.2 Flache Transaktionen

Eine Transaktion ist der Zusammenschluß einer Menge von Operationen auf einer Datenbank zu einer Einheit. Für sie gelten die sogenannten ACID-Eigenschaften [HärReu 83], durch die eine korrekte Synchronisation zwischen Transaktionen erreicht und Fehlertoleranz gewährleistet wird:

Atomarität (A, atomicity):

Die Zustandsänderungen durch die Transaktion sind atomar. Je nachdem, ob die Transaktion erfolgreich endet oder abbricht, werden nach außen, d. h. für die Benutzer oder andere Programme, entweder alle oder keine der Aktionen der Transaktion sichtbar. Der Fall, daß nur ein Teil der Aktionen

sich im Datenbankzustand nach Durchführung der Transaktion widerspiegelt, kann nicht eintreten. Insbesondere muß das Datenbanksystem im Falle des Abbruchs einer Transaktion dafür sorgen, daß alle eventuell intern schon vollzogenen Änderungen am Systemzustand wieder zurückgenommen werden. Dabei ist es gleichgültig, aus welchem Grund die Transaktion abgebrochen wurde.

Konsistenz (C, consistency):

Eine Transaktion produziert nur korrekte Ergebnisse und führt eine Datenbank, die sich in einem konsistenten Zustand befindet, wieder in eine solche über. Die Konsistenz ist dabei sowohl durch die beim Schemaentwurf festgelegten Integritätsbedingungen für die nach außen sichtbaren Daten der Datenbank als auch über den korrekten Zustand interner Speicherstrukturen und Zugriffspfade definiert. Wird bei der Bearbeitung einer Transaktion die Konsistenz verletzt, z. B. durch Verletzung von Primärschlüsseleigenschaften, so wird die Transaktion abgebrochen und die Datenbank wieder in den korrekten Zustand vor Beginn der abgebrochenen Transaktion zurückgesetzt.

Isolation (I, isolation):

Die Eigenschaft der Isolation bedeutet, daß die Transaktion logisch im Einbenutzerbetrieb abläuft. Sie läuft somit isoliert, bzw. unabhängig von anderen parallel ablaufenden Transaktionen ab. Die Transaktion sieht nur korrekte Datenbankzustände, also insbesondere keine Zwischenergebnisse anderer Transaktionen. Über diese Eigenschaft ergibt sich auch die Serialisierbarkeit von Transaktionen, die zu jeder korrekten nebenläufigen Ausführung von Transaktionen die Existenz einer sequentiellen Ausführung derselben Transaktionen mit gleichem Resultat verlangt.

Dauerhaftigkeit (D, durability):

Wird eine Transaktion erfolgreich beendet, so garantiert das Datenbanksystem die Persistenz aller von ihr erzeugten Effekte auf der Datenbank. Dies bedeutet insbesondere, daß das Datenbanksystem nach dem Bestätigen des erfolgreichen Abschlusses einer Transaktion durch entsprechende Mechanismen dafür Sorge zu tragen hat, daß es in der Lage ist, nach dem Auftreten eines beliebig gearteten Fehlers den Datenbankzustand, der nach Vollendung der Transaktion herrschte, wiederherzustellen. Dabei ist es unwichtig, ob der Fehler systemintern, vom Benutzer verursacht oder durch Hardware-Komponenten hervorgerufen wurde. Diese Wiederherstellung muß auch möglich sein, falls sich die von der Transaktion veränderten Daten beim Auftreten des Fehlers noch im flüchtigen Arbeitsspeicher befunden haben.

Im Gegensatz zu den im folgenden beschriebenen erweiterten Transaktionskonzepten werden diese klassischen Transaktionen als "flache" Transaktionen bezeichnet. Damit wird angedeutet, daß sie, aus der Sicht der Transaktion, keine weitere innere Strukturierung besitzen.

Flache Transaktionen sind der einfachste Typ der Transaktionen und in den meisten Systemen sind sie auch der einzige an der Programmierschnittstelle dem Benutzer angebotene Typ von Transaktionen.

5.2.3 Geschachtelte Transaktionen

Das Modell der flachen Transaktionen hat sich für die meisten Datenbanksysteme als adäquat erwiesen und bei diesen durchgesetzt. Trotzdem scheint es für komplexe Anwendungen, die hohe Anforderungen an Performanz und Flexibilität stellen, nicht immer die beste Lösung darzustellen. 1981 wurde das Konzept der geschachtelten Transaktionen (nested transactions) von Moss [Moss 81, Moss 85] vorgestellt. Sie stellen eine Erweiterung der flachen Transaktionen dar mit dem Ziel, deren Nachteile aufzuheben.

Im Gegensatz zu flachen Transaktionen, die als eine Folge primitiver Aktionen (Lese- und Schreiboperationen auf Objekten) angesehen werden können, besitzen geschachtelte Transaktionen eine hierarchische Struktur. Eine geschachtelte Transaktion ist nicht mehr eine Folge primitiver Aktionen, sondern zerfällt in mehrere Subtransaktionen. Diese können wiederum in mehrere Subtransaktionen zerlegt sein. Damit läßt sich eine geschachtelte Transaktion durch eine Baumstruktur beschreiben, wobei davon ausgegangen wird, daß die Subtransaktionen, die die Blätter dieses Baumes darstellen, wie flache Transaktionen die primitiven Operationen auf den physischen Daten enthalten (siehe Abbildung 5.1). Gegenüber anderen geschachtelten Transaktionen ist nur die Wurzel dieses Baumes, die Wurzeltransaktion, sichtbar. Sie verhält sich nach außen wie eine gewöhnliche Transaktion und besitzt die in Abschnitt 5.2.2 erwähnten ACID-Eigenschaften.

Formal lassen sich geschachtelte Transaktionen folgendermaßen definieren [Moss 81, GraReu 93, Elmagarmid 92]:

1. Eine geschachtelte Transaktion ist ein Baum von Transaktionen, deren Unterbäume entweder flache oder geschachtelte Transaktionen sind.

2. Transaktionen an den Blättern des Baumes sind flache Transaktionen. Blätter eines Baumes können eine unterschiedliche Tiefe besitzen.

3. Die Transaktion, die durch den Wurzelknoten des Baumes beschrieben ist, wird als Wurzeltransaktion, alle anderen werden als Subtransaktionen bezeichnet.

4. Eine Subtransaktion wird entweder abgebrochen oder ihre Ergebnisse werden freigegeben. Trotz der Freigabe einer Subtransaktion werden deren Ergebnisse nach außen gegenüber anderen Transaktionen erst mit der erfolgreichen Freigabe der Wurzeltransaktion sichtbar.

5. Der Abbruch einer beliebigen Subtransaktion im Baum verursacht den Abbruch aller ihrer Subtransaktionen.

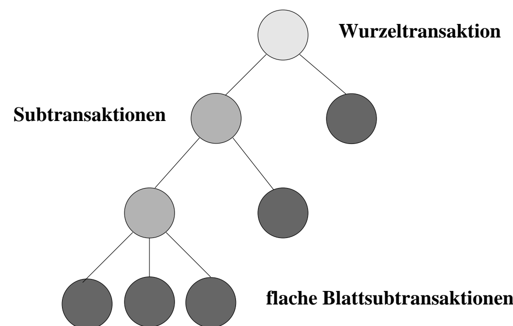

Abbildung 5.1: Baumstruktur einer geschachtelten Transaktion

Aus den letzten beiden Punkten ergibt sich, daß Subtransaktionen aus Sicht ihrer Wurzeltransaktion nur die Eigenschaften A, C und I, nicht aber D des ACID-Konzeptes besitzen. Die Dauerhaftigkeit besitzen die Subtransaktionen deshalb nicht, weil die Ergebnisse der Subtransaktion auch nach ihrer Beendigung beim Abbruch einer Vorgänger-Subtransaktion zurückgesetzt werden.

In dem von Moss vorgeschlagenen Modell können nur Blattsubtransaktionen Operationen auf den physischen Daten vornehmen. Innere Subtransaktionen der Transaktion übernehmen nur die Steuerung des Kontrollflusses sowie den Aufruf ihrer Subtransaktionen. Das dynamische Erzeugen von Subtransaktionen ist dabei explizit im Modell vorgesehen.

Nach außen wird nur von der Wurzeltransaktion die Erfüllung der ACID-Eigenschaften verlangt[1]. Von den Subtransaktionen werden aus dieser Sicht nur die Atomarität und die Isolation gefordert. Die Atomarität wird durch die Möglichkeit des Rücksetzens einzelner Subtransaktionen erzwungen.
Isolation zwischen Subtransaktionen einer Transaktionen ist notwendig, sobald Subtransaktionen nebenläufig ausgeführt werden sollen, damit es zu keinen Konflikten beim Zugriff auf gemeinsame Daten kommen kann.
Konsistenz nach außen wird von einzelnen Subtransaktionen nicht verlangt, da es

[1] Nach außen ist auch nur die Wurzeltransaktion, nicht aber deren Subtransaktionen sichtbar.

ausreichend ist, wenn am Ende der gesamten Transaktion wieder ein konsistenter Datenbankzustand erreicht wird.
Dauerhaftigkeit von Subtransaktionen ergibt sich aus den oben genannten Gründen nicht, da beim Abbruch einer Subtransaktion eventuell auch Ergebnisse schon beendeter Subtransaktionen wieder zurückgenommen werden müssen.

Zusammenfassend kann man das Verhalten geschachtelter Transaktionen durch folgende Regeln beschreiben:

Commit-Regel:

Der Commit einer Subtransaktion macht deren Ergebnisse nur für ihre Vatertransaktion sichtbar. Ihre Ergebnisse werden erst mit der Freigabe der Wurzeltransaktion nach außen sichtbar. Die endgültige Freigabe findet erst statt, wenn die Subtransaktionen selbst sowie all ihre Vorfahren bis hin zur Wurzel erfolgreich enden. Daraus ergibt sich, daß jede Subtransaktion nur dann ihre Ergebnisse freigeben kann, wenn auch ihre Wurzeltransaktion dies tut.

Rücksetzregel:

Wird eine Subtransaktion abgebrochen, so werden all ihre Kinder, unabhängig von deren aktuellem Status, ebenfalls abgebrochen. Dieses Rücksetzen setzt sich rekursiv auf den gesamten Unterbaum unter einer Subtransaktion fort. Daraus ergibt sich unmittelbar, daß mit dem Abbruch der Wurzeltransaktion auch alle Subtransaktionen der Transaktion abgebrochen werden, selbst wenn sie lokal ihre Ergebnisse schon freigegeben haben.

Sichtbarkeitsregel:

Mit dem Commit einer Subtransaktion werden all ihre Ergebnisse für ihre Vatertransaktion sichtbar. Alle Objekte einer Subtransaktion können dann durch den Vater auch für Geschwistersubtransaktionen zum Zugriff freigegeben werden. Im Fall parallel arbeitender Geschwistertransaktionen sind Änderungen durch eine Subtransaktion für deren Geschwistertransaktionen nicht sichtbar.

Geschachtelte Transaktionen erlauben die Parallelausführung ihrer Subtransaktionen, wobei zwischen Systemen unterschieden werden kann, die nur Parallelität zwischen Geschwistern erlauben, solchen, die nur Parallelität zwischen Vätern und Kindern ermöglichen und Systemen, die beide Formen unterstützen [HärProSch 90, HärRot 93]. Geschachtelte Transaktionen sind somit ein Mechanismus, der es ermöglicht, das in einer Transaktion verborgene Potential zur Parallelausführung auszuschöpfen. Dies geschieht über das Zurverfügungstellen einer Kontrollstruktur zur kontrollierten und damit sicheren Erzeugung und

Durchführung von Intra-Transaktionsparallelität. Dadurch kann die Effizienz und somit die Antwortzeit von Transaktionen verbessert werden.

Geschachtelte Transaktionen bieten einen Synchronisationsmechanismus mittels Sperren zwischen Subtransaktionen der gleichen Transaktion, wodurch sichergestellt wird, daß mehr Arbeit parallel innerhalb einer Transaktion ausgeführt werden kann (Intra-Transaktionsparallelität). Bei der Parallelausführung der Subtransaktionen einer Transaktion werden Sperren, die von einer Subtransaktion gehalten werden, bei Ende der Subtransaktion an ihre Vatertransaktion vererbt und erst mit dem Ende der Wurzeltransaktion freigegeben. Subtransaktionen innerhalb einer Transaktion sowie gegenüber anderen Transaktionen laufen somit isoliert. Sie werden als geschlossene geschachtelte Transaktionen bezeichnet[2].

Erweiterungen dieses Modells erlauben auch die Nach-Unten-Vererbung von Sperren von Subtransaktionen an ihre Kinder [HärRot 93], wodurch ein noch höherer Grad an *Intra-Transaktionsparallelität* erzielt werden soll. Parallelität innerhalb von Subtransaktionen wird vom Modell nicht unterstützt.

Die Hauptvorteile geschachtelter Transaktionen liegen in der Unterstützung von Modularisierung, Fehlerbehandlung und Intra-Transaktionsparallelität.
Die *Modularisierung* wird durch die Möglichkeit des hierarchischen Aufbaus von Transaktionen geboten, der damit eine Zerlegung in kleinere Teilaufgaben ermöglicht. Mit dieser Zerlegung in Subtransaktionen ist auch direkt die Kontrollstruktur der Transaktionen gegeben. Zudem können verschiedene Transaktionen als Subtransaktionen leicht zu einer einzelnen Transaktion zusammengeschlossen werden. Durch die korrekte Synchronisation, die geschachtelte Transaktionen zwischen ihren Subtransaktionen zur Verfügung stellen, entsteht dabei nicht die Gefahr, Inkonsistenzen bei der Parallelverarbeitung zu erzeugen.
Zur *Fehlerbehandlung* sind Subtransaktionen ein feineres Granulat der Recovery, im Gegensatz zu flachen Transaktionen, bei denen ein Fehler zwangsläufig das Rücksetzen der gesamten Transaktion zur Folge hat, wodurch eventuell große, bereits korrekt produzierte Ergebnisse verlorengehen.
Zudem unterstützen geschachtelte Transaktionen, wie oben erwähnt, *Intra-Transaktionsparallelität* durch die Möglichkeit, Subtransaktionen parallel auszuführen.

Geschlossene geschachtelte Transaktionen erlauben gegenüber flachen Transaktionen keinen höheren Grad an Inter-Transaktionsparallelität, was für einige Anwendungen ein Nachteil sein kann. Mit dem Ziel, auch diese Parallelitätsform besser nutzen zu können, sind die offenen geschachtelten Transaktionen entworfen worden. Bei ihnen kann die Freigabe von Ergebnissen von Subtransaktionen vor dem Ende der Wurzeltransaktion stattfinden. Ein Beispiel für diese

[2] Im Gegensatz dazu erlauben offene geschachtelte Transaktionen das Sichtbarwerden von Änderungen einer Transaktion vor deren Ende, mit dem Ziel, die Inter-Transaktionsparallelität zu erhöhen.

Art von Transaktionen sind die Mehrebenentransaktionen, die im folgenden Abschnitt 5.2.4 beschrieben sind.

5.2.4 Mehrebenentransaktionen

Das Konzept der Mehrebenentransaktionen (Multi-Level Transactions) ist ein weiteres Modell, welches auf der Unterteilung von Transaktionen in Subtransaktionen basiert [SchSchWei 95].

5.2.4.1 Einführung

Das Modell besitzt drei Hauptcharakteristika. Zum einen ist dies die Ausnutzung semantischer Eigenschaften der durchzuführenden Operationen, um die Isolation zwischen parallel ablaufenden Subtransaktionen teilweise aufzuheben. Dadurch soll die Nebenläufigkeit zwischen Subtransaktionen erhöht werden. Weiter wird anstelle von zustandsbasierten Undo-Operationen Kompensation verwendet, um die Atomarität einer Transaktion zuzusichern. Schließlich haben Subtransaktionen die Möglichkeit, ihre Ergebnisse persistent, d. h. global sichtbar zu machen, unabhängig vom Verarbeitungszustand der gesamten Transaktion, insbesondere also schon vor deren Commit.

Neben der Erhöhung der Inter-Transaktionsparallelität werden Mehrebenentransaktionen auch dazu verwendet, Intra-Transaktionsparallelität zu modellieren und zu beschreiben. Zusätzlich gestattet dieses Modell auch durch den Benutzer oder durch das System definierte Sicherungspunkte, wodurch ein teilweises Rücksetzen von Transaktionen möglich wird.

5.2.4.2 Das Modell der Mehrebenentransaktionen

Mehrebenentransaktionen sind eine Variante der geschachtelten Transaktionen, bei denen alle Blätter des Transaktionsbaumes die gleiche Tiefe besitzen. Zudem dürfen Zwischenergebnisse der Bearbeitung einer Transaktion bereits vor deren Ende sichtbar werden. Dies wird zum Beispiel dadurch erreicht, daß Subtransaktionen derselben Transaktion, die zwar isoliert voneinander ablaufen, erhaltene Sperren sofort nach ihrem Ende endgültig frei geben. Durch dieses vorzeitige Freigeben von Sperren entsteht das Problem, daß bereits erfolgreich beendete Subtransaktionen zusammen mit einem Abbruch der Vatertransaktion nachträglich ebenfalls abgebrochen werden müssen und sich deren vorzeitig sichtbar gewordene Resultate in der Datenbank befinden. An die Stelle der hier sonst üblichen Recovery-Protokolle treten nun kompensierende Subtransaktionen, welche die Datenbank zwar nicht genau in denselben Zustand vor Beginn der Transaktion überführen, aber in einen semantisch äquivalenten.

Die Entstehung des Transaktionsbaumes, also die Erzeugung von Subtransaktionen bei Mehrebenentransaktionen, richtet sich nach der Schichtenarchitektur des betreffenden Systems. In einem Datenbanksystem nimmt man für eine solche Unterteilung folgende typische Schichten: die deskriptive Daten-Ebene, die Tupel-Ebene, die Record- und die Seiten-Ebene. Dabei wird zu jeder Operation in einer Ebene E_{i+1} des Systems eine Subtransaktion in der nächst tieferliegenden Ebene E_i des Systems erzeugt. Diese beinhaltet wiederum mehrere Operationen der entsprechenden Ebene. Die Unterteilung wird bis zur untersten Ebene E_0 des Systems iteriert. Verschiedene Transaktionsbäume haben somit immer dieselbe Tiefe und Subtransaktionsknoten gleicher Tiefe korrespondieren immer mit Operationen der gleichen Ebene im System.

Die Hauptidee der Synchronisation bei Mehrebenentransaktionen besteht darin, ebenenspezifische Konfliktrelationen über den Operationen dieser Ebene zu bilden. Diese Konfliktrelationen geben semantische Informationen über die Operationen der Ebene wieder, wie deren Kommutativität bzw. Kompatibilität. So können beispielsweise zwei Operationen der Ebene E_i konfliktfrei sein, da sie kommutativ sind, obwohl ihre Implementierungen auf Ebene E_{i-1} miteinander im Konflikt stehen. In einem solchen Fall wird der Konflikt auf Ebene E_{i-1} zu einem Pseudokonflikt auf Ebene E_i. Dadurch kann eine Folge von Operationen, die auf Ebene E_{i-1} nicht serialisierbar ist, dies aber auf Ebene E_i sein.
Abbildung 5.2 zeigt ein Beispiel für eine derartige Konfliktsituation. Zwei Transaktionen, T_1 und T_2 heben Geld von einem Konto a ab und überweisen dies anschließend auf Konto b. Die Operationen sind auf der untersten Ebene des Systems durch Lese- und Schreiboperationen auf den entsprechenden Datensätzen implementiert. Mit der in der Abbildung vorgegebenen Reihenfolge dieser Operationen auf Ebene E_0 wären die Transaktionen T_1 und T_2 nicht serialisierbar. Da aber auf Ebene E_1 die beiden Überweisungsoperationen kommutativ sind, können sie vertauscht werden, wodurch auf dieser höheren Ebene eine serialisierbare Ausführungsreihenfolge entsteht. Dadurch gerät der Konflikt auf Ebene E_0 zum Pseudokonflikt und somit kann die Ausführungsreihenfolge als serialisierbar angesehen werden [SchWeiSch 91].

Durch Ausnutzen dieser ebenenspezifischen Konfliktinformationen erlauben Mehrebenentransaktionen einen höheren Grad an Parallelität zwischen Transaktionen, also Inter-Transaktionsparallelität, als andere Transaktionsarten wie flache oder geschachtelte Transaktionen. Andererseits kann der Abbruch von Transaktionen nicht mehr einfach durch Restaurieren der in Ebene E_0 modifizierten Objekte auf den Stand vor der Transaktion implementiert werden, da die Änderungen auf unterster Ebene sofort nach Beendigung der jeweiligen Subtransaktion global sichtbar wurden. Stattdessen werden bei Mehrebenentransaktionen dazu kompensierende Transaktionen verwendet, um die Effekte der abzubrechenden Transaktion rückgängig zu machen.

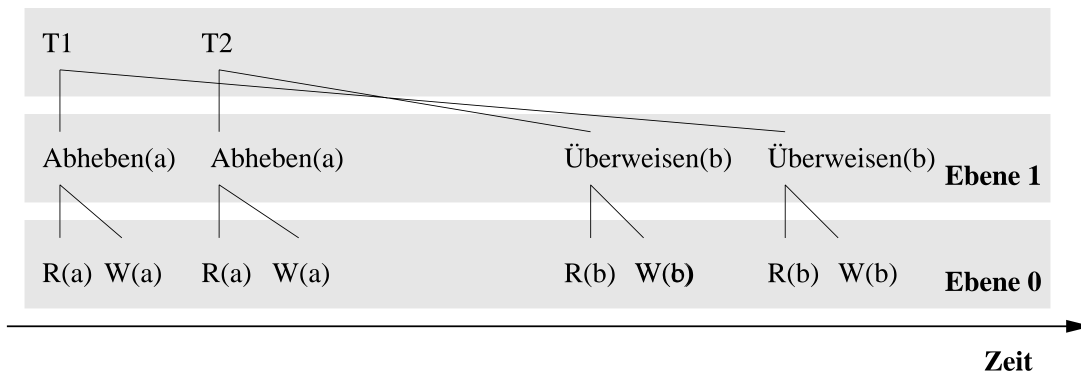

Abbildung 5.2: Mehrebenentransaktionen

Wichtig ist, daß sowohl die Subtransaktionen als auch ihre kompensierenden Subtransaktionen atomar sind. Ansonsten könnte sich die Datenbank nach einem System-Crash in einem inkonsistenten Zustand befinden. Mehrebenentransaktionen unterstützen Inter-Subtransaktionsparallelität, da Intra-Transaktionsparallelität auf höheren Ebenen zu Inter-Subtransaktionsparallelität auf niederen Ebenen wird [HasWei 93]. Da das Subtransaktionsmanagement Subtransaktionen auf den niedrigeren Ebenen unabhängig von ihrem Vaterknoten behandelt, können in einer Transaktion mehrere Subtransaktionen parallel durchgeführt werden [WeiHas 91].

In einem System mit Mehrebenentransaktionen können Inter- und Intra-Transaktionsparallelität kombiniert werden, was besonders günstig bei Systemen erscheint, die sowohl OLTP- als auch Decision Support-Transaktionen nebenläufig ausführen [HasWei 93].

5.2.5 Sicherungspunkte

Es gibt prinzipiell zwei verschiedene Techniken in Datenbanksystemen, die als Sicherungspunkte bekannt sind.

Zum einen sind dies Sicherungspunkte (checkpoint), die im Rahmen von Langzeit- bzw. Media-Recovery eingesetzt werden. Ein solcher Sicherungspunkt definiert einen logischen Zeitpunkt, zu dem sich das Datenbanksystem in einem konsistenten Zustand befunden hat und in den es auf Anforderung wieder zurückgebracht werden kann. Durch ihn wird sichergestellt, daß das Datenbanksystem im Katastrophenfall wieder in einen konsistenten, unter Umständen aber auch veralteten Zustand gelangen kann.
Dieser Typ von Sicherungspunkten ist im weiteren nicht von Interesse.

Die zweite Sicherungspunkttechnik findet im Rahmen der Kurzzeit-Recovery An-

wendung. Die hier verwendeten Sicherungspunkte dienen im Gegensatz zu der oben erwähnten Variante nicht zur Wiederherstellung eines globalen Systemzustandes. Die Sicherungspunkte werden innerhalb einer einzelnen Transaktion verwendet und dienen dazu, deren Zustand bei Bedarf wieder auf diesen, zuvor gesicherten Zustand, zurücksetzen zu können.

Wird ein solcher Sicherungspunkt innerhalb einer Transaktion gesetzt, so muß vom Datenbanksystem sichergestellt werden, daß die Transaktion später wieder in diesen Zustand, inklusive der zu diesem Zeitpunkt von der Transaktion verwendeten Ressourcen, gebracht werden kann. Solche transaktionsinternen Sicherungspunkte werden von Datenbanksystemen üblicherweise vor jeder Anfrage einer Transaktion gesetzt. Damit garantieren sie die vom SQL-Standard verlangte Atomarität (statement atomicity) einer Anfrage innerhalb einer Transaktion in Bezug auf den Fehlerfall [GraReu 93]. Dadurch wird sichergestellt, daß die Transaktion die Möglichkeit besitzt, falls eine Anfrage fehlschlägt, in den Zustand vor Beginn dieser Anfrage zurückzukehren, damit nicht die gesamte Transaktion abgebrochen werden muß.

Als Beispiel für den Nutzen dieser Sicherungspunkte kann eine Einfügeoperation eines Datensatzes mit einem in der Tabelle bereits existierenden Schlüssel dienen. Hier kann dem Benutzer ein Fehler gemeldet werden und die Transaktion kehrt in den Zustand vor dieser Einfügeoperation zurück, anstelle komplett abgebrochen zu werden. Somit besteht für den Benutzer die Möglichkeit, selbst zu entscheiden, wie die Transaktion fortgesetzt werden soll, beispielsweise die Einfügeoperation mit geänderter Eingabe zu wiederholen oder die Transaktion selbst abzubrechen.

Mit geringfügigen Änderungen im System ist es auch möglich, dem Benutzer das Setzen dieser internen Sicherungspunkte über die SQL-Schnittstelle des Systems zu ermöglichen. Damit hat er selbst die Möglichkeit, die Transaktion teilweise nach Bedarf zurückzusetzen. Verschiedene existierende Systeme wie Ingres, Oracle oder SQL Server bieten diese Sicherungspunkte für den Benutzer an [BonSar 95, BerNew 97].

Der zusätzliche Aufwand, der betrieben werden muß, um solche Sicherungspunkte in einem System zu unterstützen, ist dann vielversprechend, wenn im System lang laufende Transaktionen existieren. Im Fehlerfall oder im Falle des Abbruchs durch den Benutzer verlieren diese dann nicht all die zuvor erzeugten Ergebnisse.

Das im folgenden entwickelte Konzept wird sich auf diese zweite Art der Sicherungspunkte beziehen. Wichtig ist, dabei zu beachten, daß es sich bei den hier vorgestellten Sicherungspunkten um Sicherungspunkte zwischen den einzelnen Anfragen einer Transaktion handelt. Innerhalb einer einzelnen Anfrage werden dort keine Maßnahmen zur Behandlung von Fehlerfällen getroffen.

5.3 Ausführungsmodell in MIDAS – Aktivitätskontrolle und Fehlerbehandlung

In diesem Abschnitt werden nochmals die wichtigsten Eigenschaften von MIDAS vorgestellt, die für die Entwicklung eines Modells zur Fehlerbehandlung notwendig sind. Anschließend werden die Anforderungen an das gesuchte Modell zur Fehlerbehandlung dargestellt und die beim Entwurf wichtigen Fragenstellungen erörtert. Diese Anforderungen werden mit den in Abschnitt 5.2 vorgestellten existierenden Modellen verglichen. Dabei wird gezeigt, daß keines der existierenden Modelle den gestellten Anforderungen genügt. Unter Verwendung der in diesem Abschnitt gewonnenen Ergebnisse und ermittelten Anforderungen wird im folgenden Abschnitt ein speziell an diese Anforderungen angepaßtes Modell, das FPT-Modell, vorgestellt.

5.3.1 Charakteristika des MIDAS-Ausführungsmodells

Beim Entwurf eines Modells zur Fehlerbehandlung sind die im folgenden beschrieben Charakteristika des MIDAS-Ausführungsmodells von Bedeutung (siehe auch Abschnitt 2.3).

MIDAS besitzt eine konventionelle SQL-Schnittstelle. Insbesondere werden dem Benutzer keine parallelen Konstrukte an dieser Schnittstelle zur Verfügung gestellt. Jegliche Parallelisierung findet demnach im System selbst statt und ist daher auch komplett unter der Kontrolle des Systems.

MIDAS unterstützt echte Inter-Transaktionsparallelität (siehe Abschnitt 2.3). Prozesse, die unterschiedliche Transaktionen implementieren, können echt parallel auf verschiedenen Prozessoren des Systems laufen, speziell auch auf verschiedenen Rechnern des Systems. Dies geschieht unter voller Kontrolle durch das Datenbanksystem. Die Nebenläufigkeit ist damit nicht auf eine Pseudoparallelausführung auf einer Einprozessormaschine oder auf eine nur durch das Betriebssystem gesteuerte Parallelausführung auf einer Multiprozessormaschine beschränkt.

Zusätzlich zur Inter-Transaktionsparallelität wird in MIDAS Intra-Transaktionsparallelität für umfangreiche Anfragen eingesetzt (siehe Abschnitt 2.3). Da die Semantik von SQL von einer sequentiellen Ausführung der Anfragen ausgeht und Anfragen dabei auf die Ergebnisse vorheriger Anfragen zugreifen können, ist eine Parallelausführung mehrerer Anfragen einer Transaktion im allgemeinen nicht möglich. Intra-Transaktionsparallelität beschränkt sich in MIDAS somit auf die Parallelisierung einzelner Anfragen, also Intra-Query-Parallelität. In MIDAS wurde nicht versucht, Anfragen verschränkt, unter Beibehaltung der Semantik durchzuführen. Intra-Query-Parallelität wird dabei, wie in Abbildung 5.3 ge-

zeigt, durch den Parallelisierer des Systems ermöglicht. Der aus der sequentiellen Optimierung kommende Operatorbaum (a) wird durch den Parallelisierer (b) an speziellen Stellen aufgespalten. Die so entstehenden Teilbäume werden auf einzelne Ausführungseinheiten, die Interpreter, im Ausführungssystem verteilt. Diese Ausführungseinheiten sind in der Implementierung einzelne Prozesse, die asynchron zueinander arbeiten. Durch diesen Schritt wird Pipeline-Parallelität zwischen den Ausführungseinheiten ermöglicht (c).

In einem weiteren Schritt (d) werden einzelne Teilpläne mehrfach zur Ausführung gebracht, wodurch Intra-Operator-Parallelität, beziehungsweise Datenparallelität zwischen den Verarbeitungseinheiten entsteht.

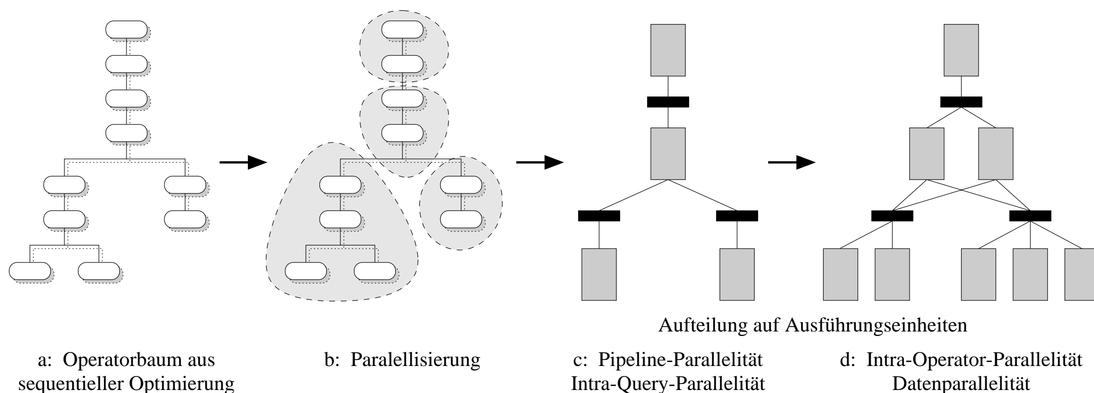

Abbildung 5.3: Intra-Transaktionsparallelität

Diese Ausführungseinheiten sind unabhängige, parallel arbeitende Komponenten. Wichtig für die spätere Fehlerbehandlung ist, daß die so entstandene Struktur zwischen den Ausführungseinheiten den Datenfluß und nicht den Kontrollfluß widerspiegelt. Die Ausführungseinheiten werden durch eine gesonderte Komponente gesteuert. Diese Steuerung beschränkt sich auf das Starten und Beenden der Komponenten sowie das Einrichten der Kommunikationskanäle zwischen den Komponenten. Ansonsten arbeiten die Komponenten autonom und asynchron zueinander.

Der Austausch von Zwischenergebnissen zwischen den Komponenten findet über spezielle Kommunikationskanäle, die Kommunikationssegmente (siehe Abschnitt 2.4.9), in der Cacheebene des Systems statt.

5.3.2 Aktivitätskontrolle

Das zu entwerfende Modell zur Fehlerbehandlung ist Teil der Aktivitätskontrolle im System. Innerhalb des Ausführungsmodells des Systems stellt die Aktivitätskontrolle die Kontrolleinheit zur Koordination der verschiedenen Aktivitäten dar. Dabei geht es um die Frage, wie die parallel laufenden Aktionen im System koordiniert werden und wie auf Fehlersituationen zu reagieren ist. Damit wird zum einen das Scheduling und zum anderen ein an das Verarbeitungsmodell angepaßtes Transaktionsmodell gesteuert.

Werden im System nur "flache" Transaktionen verwendet, so ergibt sich die in Abbildung 5.4 dargestellte Situation. Die gesamte Anfrage, welche auf viele einzelne Ausführungseinheiten aufgeteilt ist und daher eine starke Strukturierung aufweist, ist durch eine einzelne, geschlossene Transaktionshülle umschlossen. Diese gewährleistet die ACID-Eigenschaften der Transaktion gegenüber anderen Transaktionen. Allerdings ist durch die starke Parallelisierung, durch die die große Anzahl an Ausführungseinheiten entstanden ist, ein Ungleichgewicht zwischen der internen Strukturierung der Transaktion und den Fähigkeiten der Transaktion entstanden. Auf der einen Seite wurden durch die Strukturierung viele, relativ unabhängige Einheiten innerhalb einer Transaktion gewonnen, die nebenläufig und verteilt im System zur Ausführung gebracht werden können. Auf der anderen Seite ist das Transaktionsmodell starr und kann nur die Transaktion als Ganzes behandeln. So muß beispielsweise im Falle eines Fehlers, der nur auf eine einzelne Ausführungseinheit beschränkt ist, zwangsläufig die gesamte Transaktion abgebrochen werden. Wünschenswert ist in einem solchen Fall, speziell auf den Fehler reagieren zu können und nur die wirklich betroffene Ausführungseinheit einer Fehlerbehandlung zu unterziehen. Dadurch können vom Fehler nicht betroffene Ausführungseinheiten weiter existieren und deren bereits produzierte Ergebnisse wiederverwendet werden.

Eine derartige Möglichkeit, auf Fehler zu reagieren, bietet das Modell der flachen Transaktionen nicht, da es keinerlei Kenntnis über die interne Strukturierung der Transaktion besitzt.

Das Ziel des in diesem Kapitel zu entwickelnden Modells zur Fehlerbehandlung ist es, ein Konzept zur Fehlerbehandlung bereitzustellen, welches den Transaktionsschutz, beziehungsweise das Granulat der Transaktionsverarbeitung an die Struktur des Verarbeitungsmodells anpaßt.

Nach außen, gegenüber anderen Transaktionen, sollen wie bisher alle ACID-Eigenschaften gelten. Ziel ist es, von der internen Strukturierung der Verarbeitung, also der Aufteilung des Gesamtplans auf mehrere Ausführungseinheiten, zu profitieren und diese auf die Transaktionsverarbeitung zu übertragen. Dabei sollte der durch die Parallelisierung erreichte Grad an Parallelität möglichst nicht durch Maßnahmen der Transaktionskontrolle eingeschränkt werden, wie dies z. B.

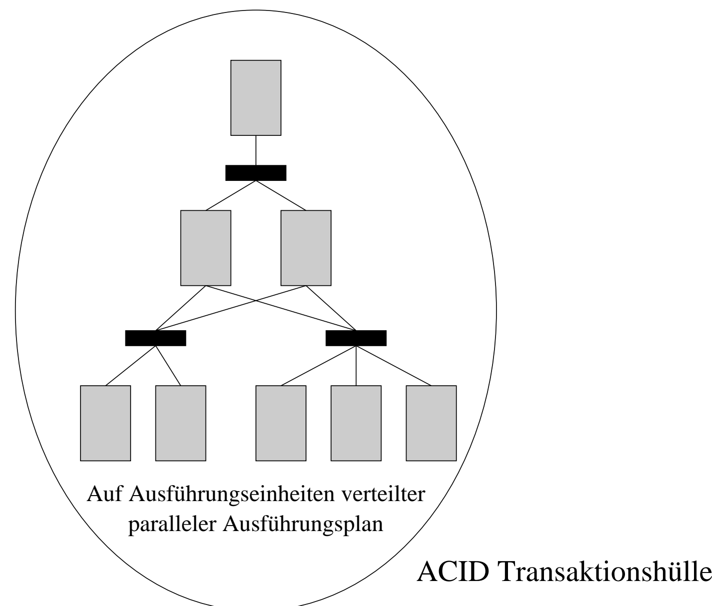

Abbildung 5.4: Parallelisierung und Transaktionen

durch Isolation zwischen einzelnen Komponenten entstehen könnte, und gleichzeitig ein Gewinn durch Transaktionsschutz auf feinerem Granulat erzielt werden. Der Normalfall, in dem kein Fehler auftritt, soll nicht oder nur möglichst wenig belastet sein, und im Fehlerfall sollen die Verluste durch Rücksetzen und Neustart der Transaktion gegenüber einer unstrukturierten Transaktionsverarbeitung deutlich verringert werden. Wichtig dabei ist, insbesondere ein sinnvolles Granulat der Strukturierung der Transaktion zu finden, aus dem im Fehlerfall Gewinn gezogen werden kann, das aber im Normalfall nicht zu viel Aufwand erzeugt.

5.3.3 Fehlerbehandlung

Die zentrale Frage bei einem solchen Modell zur Fehlerbehandlung als Teil der Aktivitätskontrolle ist, wie auf das Auftreten eines Fehlers reagiert werden soll. Bei dem im folgenden entwickelten Modell soll insbesondere die Möglichkeit bestehen, daß das System die Fähigkeit besitzt, Fehler zu erkennen und zu kompensieren, d. h. es soll ohne Eingriff durch den Benutzer auf gewisse Fehlersituationen reagieren und somit diese Fehlersituationen vor dem Benutzer verbergen können. Als Fehlerfälle, für die eine solche Fehlertoleranz angeboten werden kann, kommen Verarbeitungsfehler wie zum Beispiel Verklemmungen zwischen Transaktionen, aber auch Fehler, die erst durch die Parallelverarbeitung entstehen, wie Verklemmungen zwischen einzelnen Ausführungseinheiten, in Betracht. Weiterhin kann diese Möglichkeit auch für Komponentenausfälle angeboten werden. Knoten-

beziehungsweise Rechner-Ausfälle können zwar von dem zu entwickelnden Modell theoretisch behandelt werden, dürften aber in der Praxis schwer umzusetzen sein. Die Klasse der Fehler, die das Modell behandeln können soll, sind somit die Transaktionsfehler, die in Abschnitt 5.2.1 beschriebenen werden.

Die zentrale Idee bei dem zu entwickelnden Modell ist es, nach dem Auftreten eines Fehlers beim Wiederaufsetzen die bereits produzierten Zwischenergebnisse, d. h. den Inhalt der Datenströme, die zwischen den Ausführungseinheiten ausgetauscht werden, wiederzuverwenden. Diese Wiederverwendung soll das erfolgreiche Beenden einer Transaktion trotz eines aufgetretenen Fehlers ermöglichen.

In dem Modell muß sicherlich ein Rücksetzen der Transaktion auf den Zustand vor der letzten Anfrage möglich sein. Diese Fähigkeit wird als Anfrageatomarität (statement atomicity) bezeichnet und ist allgemein für die Bereitstellung der Semantik von SQL notwendig [GraReu 93]. Sie erlaubt dem System, nach dem Fehlschlagen einer einzelnen Anfrage wieder einen konsistenten Datenbankzustand herzustellen und schafft damit die Möglichkeit, dem Benutzer das Fehlschlagen einzelner SQL-Anfragen zu melden und diesen darauf reagieren zu lassen. Dieses Verfahren entspricht in etwa einem einzelnen Sicherungspunkt (siehe Abschnitt 5.2.5), der direkt vor der aktuell laufenden Anfrage gesetzt wurde.

Weiterhin soll das Modell das Abbrechen und den Neustart einzelner Ausführungseinheiten innerhalb einer Anfrage unterstützen, um auf Fehler, die auf einzelne Teile einer Anfrage beschränkt sind, reagieren zu können und nicht die komplette Transaktion abbrechen zu müssen. Der Abbruch der kompletten Transaktion wäre die notwendige Konsequenz bei flachen Transaktionen.

5.4 Anforderungen an das Modell zur Fehlerbehandlung

Aus den beiden letzten Abschnitten 5.3.2 und 5.3.3 ergibt sich folgende Kurzfassung der Anforderungen an das zu entwerfende Modell zur Fehlerbehandlung:

- Das Modell soll ein an die Struktur der Parallelverarbeitung angepaßtes Granulat der Fehlerbehandlung besitzen.
- Transaktionsfehler in einzelnen Ausführungseinheiten bei der Parallelverarbeitung sollen nicht zwangsläufig den Abbruch der gesamten Transaktion zur Folge haben.
- Einzelne, von Fehlern betroffene Ausführungseinheiten sollen vom System neu gestartet werden, so daß die Transaktion erfolgreich beendet werden kann. Dem Benutzer soll dieses systeminterne Vorgehen verborgen bleiben.

Dieses Vorgehen soll effizienter als der Abbruch und Neustart der kompletten Transaktion sein.

- Die Leistungsfähigkeit des Systems soll nicht durch vorsorgliche Maßnahmen für eine spätere Fehlerbehandlung eingeschränkt werden. Insbesondere soll der Grad an Daten- und Pipeline-Parallelität erhalten bleiben.

5.5 Evaluierung der Modelle

In den vorherigen Abschnitten wurden kurz die Fehler vorgestellt, die das gesuchte Modell behandeln soll. Des weiteren wurde das Ausführungsmodell vorgestellt, in dessen Rahmen die Fehlerbehandlung stattfinden soll. Es stellt sich nun die Frage, inwieweit die bereits existierenden, in Abschnitten 5.2 dargestellten Konzepte, die gestellten Anforderungen an ein solches Modell erfüllen können.

5.5.1 Sicherungspunkte

Sicherungspunkte bieten zwar die Möglichkeit, Transaktionen in kleinere Einheiten zu zerteilen, einen Eingriff innerhalb einzelner Anfragen ermöglichen sie allerdings nicht. Sie können damit nicht der Hauptanforderung an das gesuchte Modell gerecht werden, die *Kontrolle über die einzelnen Ausführungseinheiten*, auf denen eine Anfrage berechnet wird, zu übernehmen.

Die Möglichkeit, Transaktionen auf den Zustand vor Beginn der letzten Anfrage zurückzusetzen, müssen, wie bereits oben erwähnt, ohnehin alle SQL-Systeme bereitstellen. Die Fähigkeit, eine Transaktion um mehrere Anfragen zurückzusetzen ist eine Eigenschaft, die das System nicht eigenmächtig vollziehen kann, sondern die durch den Benutzer gesteuert werden muß, da dieser interaktiv zwischen der Ausführung der einzelnen Anfragen einer Transaktion eingreifen kann und damit, bei einem eigenmächtigen Rücksetzen durch das System, die Benutzersemantik nicht mehr gewährleistet werden kann. In dem gesuchten Modell sollen aber alle *Eingriffe zur Fehlertoleranz vollständig durch das System* durchgeführt und vor dem Benutzer verborgen werden. Diese Fähigkeit der Sicherungspunkttechniken ist daher für das gesuchte Modell nicht relevant.

Insgesamt läßt sich feststellen, daß die Sicherungspunkte zwar eine interessante Technik darstellen und gewisse Ähnlichkeiten zu dem gesuchten Modell aufweisen. Die Sicherungspunkttechniken bieten aber keine Lösungen für die gewünschten Anforderungen an das Modell.

5.5.2 Geschachtelte Transaktionen

Das Modell der geschachtelten Transaktionen scheint auf den ersten Blick am ehesten den Anforderungen an das gesuchte Modell zu entsprechen [Voinou 96]. Es wird eine baumartige Strukturierung von Transaktionen unterstützt. Parallelität innerhalb einer Transaktion wird auf einfache Weise ermöglicht. Die Subtransaktionen im Modell der geschachtelten Transaktionen können nebenläufig arbeiten. Weiterhin bietet es sich an, Subtransaktionen auf die einzelnen Komponenten des Systems, die Ausführungseinheiten, abzubilden. Dadurch ergibt sich auf einfache Weise die Strukturierung der Transaktion, indem man die Aktionen einer Ausführungseinheit als Subtransaktion definiert. Der Graph der Ausführungseinheiten wäre damit identisch mit dem Subtransaktionsbaum und eine explizit implementierte Kontrollstruktur wäre gegeben.

Bei einer genaueren Betrachtung finden sich aber schnell Eigenschaften der geschachtelten Transaktionen, die mit den gewünschten Anforderungen des gesuchten Modells nicht in Übereinstimmung zu bringen sind.

Als erstes zeigt sich, daß eine direkte Abbildung der Subtransaktionen der geschachtelten Transaktionen auf die Ausführungspläne nicht möglich ist, da die parallelen Ausführungspläne gerichtete, *azyklische Graphen* und damit keine *Bäume* sind. Eine unmittelbare Anpassung der geschachtelten Transaktionen an derartige Graphen ist nicht offensichtlich.

Weiterhin widerspricht die von den geschachtelten Transaktionen geforderte *Isolation zwischen Subtransaktionen* dem in den Ausführungsplänen gewünschten Pipelining. Durch die Isolation ist keine *Parallelität* zwischen Vater- und Sohnknoten möglich. Der Grad an Parallelität wird damit eingeschränkt. Diese Parallelitätsform ist aber, zusammen mit der Datenparallelität, eine der beiden wichtigen Möglichkeiten, Parallelität innerhalb einer Anfrage zu verwirklichen. Auf sie kann nicht verzichtet werden.

Zudem ist die grundlegende Strukturierung bei geschachtelten Transaktionen sehr verschieden gegenüber der Strukturierung der Ausführungspläne. Bei den geschachtelten Transaktionen werden die Aufgaben, die in einem Knoten vollzogen werden sollen, komplett auf dessen Söhne verteilt und der Knoten übernimmt selbst nur noch Kontrollaufgaben. Die komplette, eigentliche Arbeit findet somit in den Blättern des Baumes der geschachtelten Transaktionen statt. Bei dem gegebenen Ausführungsmodell wird zwar auch nur in den Blättern auf Datenobjekte zugegriffen, allerdings finden Berechnungen in allen Knoten des azyklischen Graphen statt.

Wesentlich ist weiter die Tatsache, daß die parallelen Ausführungspläne den *Datenfluß* wiedergeben, wohingegen ein Baum von Subtransaktionen im Modell der geschachtelten Transaktionen den *Kontrollfluß* widerspiegelt. Eine direkte Abbildung der Ausführungspläne auf das Modell der geschachtelten Transaktionen

würde, wie in Abbildung 5.5 gezeigt, einen Baum ergeben, in dem alle Knoten des parallelen Ausführungsplans als Blätter in einem Baum mit Höhe 1 hingen und ein spezieller Kontrollknoten die Wurzel wäre. In diesem Baum verliefe der Datenfluß dann horizontal durch die Blätter, was wiederum nicht dem Modell der geschachtelten Transaktionen entspricht.

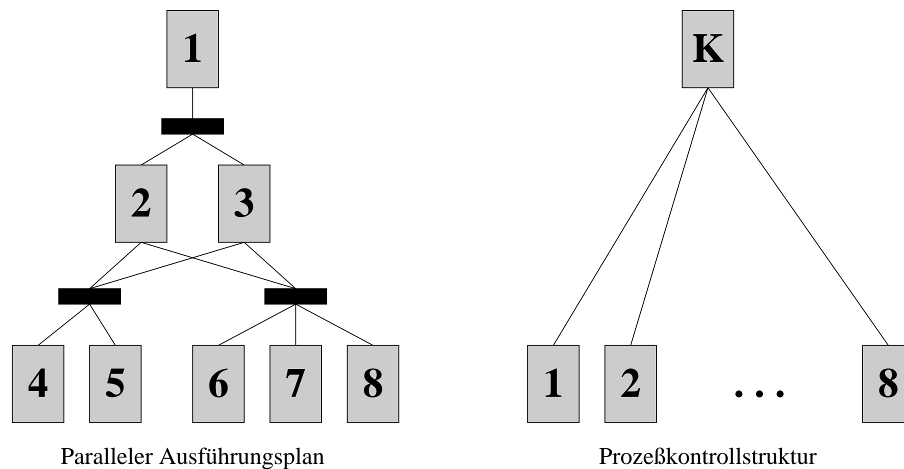

Abbildung 5.5: Paralleler Ausführungsplan und Prozeßkontrollstruktur

Maßnahmen zur eingeschränkten Sichtbarkeit von Daten, beispielsweise mit Hilfe von Isolation, sind im gesuchten Modell nicht notwendig, da die parallelisierten Ausführungspläne nicht gemeinsam auf permanente Daten schreiben und somit nicht voneinander abgeschirmt werden müssen.

Nebenbei sei noch erwähnt, daß die Möglichkeit der geschachtelten Transaktionen zum *dynamischen Erzeugen von Subtransaktionen* nicht benötigt wird. Spezielle Maßnahmen, um dies zur Verfügung zu stellen, sind im gesuchten Modell nicht nötig, da die gewünschte Struktur des parallelen Ausführungsplans bereits zu Beginn der Laufzeit bekannt ist und nicht dynamisch angepaßt werden muß.

Geschachtelte Transaktionen in ihrer vollen Form würden beim Einsatz auf einer so tief im System liegenden Ebene, wie sie hier angedacht ist, sicher viel *zu hohe Kosten* verursachen. Die Fähigkeit zu voller Recovery auf einer so niedrigen, feingranularen Ebene würde zu viel System-Overhead durch zu viel Logging und andere interne Verwaltung erfordern. Das System könnte im Normalfall nicht mehr effizient arbeiten. Geschachtelte Transaktionen eignen sich von ihrer Konzeption her eher dazu, auf Benutzerebene angeboten zu werden, beispielsweise um dem Benutzer die Kontrolle über einzelne Teile seiner Transaktion zu geben. Ein typisches Beispiel hierfür wäre die Reisebuchungstransaktion, bei der der

Benutzer getrennt über Flug, Hotel und Mietwagen innerhalb einer Transaktion entscheiden und gegebenenfalls Einzelbuchungen ohne Verlassen der Transaktion rückgängig machen kann.

Beim gesuchten Modell ist dagegen *keine Recovery auf permanenten Datenbankdaten* gewünscht, sondern nur die eventuell, also nicht zwingend notwendige Wiederverwendbarkeit von Zwischenergebnissen, d. h. abgeleiteten Daten. Es müssen insbesondere keine speziellen Vorkehrungen getroffen werden, um die Wiederherstellung eines konsistenten Datenbankzustandes zu ermöglichen, da im Zweifelsfall immer das vollzogen werden kann, was bei normalen, flachen Transaktionen auch notwendig wäre, nämlich die gesamte Transaktion abzubrechen.

Zusammenfassend läßt sich feststellen, daß die geschachtelten Transaktionen die gestellten Anforderungen nicht erfüllen können, da eine Abbildung der Baumstruktur der geschachtelten Transaktionen auf den Graph der Ausführungseinheiten nicht möglich erscheint, und die interne Struktur sowie der Kontroll- und Datenfluß zwischen den Ausführungseinheiten und innerhalb der geschachtelten Transaktionen nicht übereinstimmen. Außerdem würden geschachtelte Transaktionen mit ihren vollen Recovery-Fähigkeiten bei dieser Verwendung zu hohe Kosten verursachen.

5.5.3 Mehrebenentransaktionen

Die Mehrebenentransaktionen haben als Variante der geschachtelten Transaktionen ähnliche Nachteile bei der Erfüllung der gestellten Anforderungen wie die geschachtelten Transaktionen selbst. Insbesondere stellen die Mehrebenentransaktionen eine Transaktionsvariante dar, die auf *verstärkte Inter-Transaktionsparallelität* ausgerichtet ist. Dagegen soll das gesuchte Modell vor allem die Intra-Transaktionsparallelität unterstützen.

Zudem ist die von den Mehrebenentransaktionen vorgeschlagene *Baumstruktur*, die einen Durchlauf durch die verschiedenen Schichten des Systems widerspiegeln soll, für das gesuchte Modell ungeeignet. Die parallelen Ausführungspläne geben den Datenfluß innerhalb einer Schicht des Systems wieder und lassen sich nicht auf die vorgeschlagenen Bäume abbilden, wie dies bereits bei der Argumentation in Bezug auf geschachtelte Transaktionen gezeigt wurde.

Auch die *Recovery-Maßnahmen* und Sichtbarkeitsmechanismen der Mehrebenentransaktionen werden nicht gebraucht, da nur die Verwaltung von Zwischenergebnissen betrachtet wird und permanente Datenbankdaten, und damit die Datenbankkonsistenz, von dem gesuchten Modell nicht berührt werden.

Die bei den Mehrebenentransaktionen verwendeten *Kompensationsmechanismen* finden im gesuchten Modell ebenfalls keine Anwendungsmöglichkeit.

Insgesamt läßt sich feststellen, daß die Mehrebenentransaktionen, trotz ihrer Ähn-

lichkeit zu den geschachtelten Transaktionen, kaum dem gesuchten Modell zur Fehlerbehandlung gerecht werden können. Sie erfüllen nicht die an das Modell gestellten Anforderungen und bieten zudem eine große Anzahl an Eigenschaften, die für das Modell nicht benötigt werden.

5.6 Änderungen am Modell der geschachtelten Transaktionen

Aus dem Vergleich mit den existierenden Konzepten ergab sich eine gewisse Ähnlichkeit des gesuchten Modells zu dem Modell der geschachtelten Transaktionen. Diese legt es nahe, das gesuchte Modell aus dem Modell der geschachtelten Transaktionen zu entwickeln. Wie in Abschnitt 5.5.2 gezeigt, können nur einige der Eigenschaften des Modells der geschachtelten Transaktionen für das neue, gesuchte Modell verwendet werden. Die anderen müssen angepaßt oder aufgehoben werden.

Dazu zählt die *Aufhebung der Isolation* zwischen den einzelnen Komponenten. Durch sie kann es im Fehlerfall zur Weiterleitung fehlerhafter Daten kommen, was zu einem kaskadierenden Rücksetzen von Komponenten führen muß, falls Väter zusammen mit einem Sohn zurückgesetzt werden müssen, da diese inkonsistente oder unvollständige Daten gesehen haben. Das zu entwerfende Modell muß dies berücksichtigen.

Die Hauptcharakteristik bei dem neuen Modell wird sein, möglichst viele, von einem auftretenden Fehler nicht betroffene Komponenten einer Transaktion normal weiterlaufen zu lassen, und nur den wirklich von einem Fehler betroffenen Teil der Transaktion abzubrechen und nochmals korrekt ablaufen zu lassen. Dazu dient die *Speicherung von Zwischenergebnissen* in den, in Abschnitt 2.4.9 vorgestellten, Kommunikationskanälen und die Wiederverwendung dieser Zwischenergebnisse nach dem Auftreten eines Fehlers. Im Falle von Knotenausfällen müssen diese Zwischenergebnisse auf Externspeicher ausgelagert sein. Für Komponentenausfälle oder Verarbeitungsfehler reicht es aus, sie noch im Speicher vorliegen zu haben. Ein allgemeines externes Speichern aller Kommunikationskanäle ist sicher zu teuer, deshalb findet dies nur an ausgewählten Stellen statt. An solchen Punkten kann man auch gegebenenfalls Isolation einführen, um ein kaskadierendes Rücksetzen einzuschränken. Dies wird auch automatisch erreicht, sobald der Inhalt eines solchen Segments komplett gespeichert vorliegt.

Bei dem neuen Modell muß auch die *Commit- und Rücksetzregel* der geschachtelten Transaktionen (siehe 5.2.3) geändert werden. Die Commit-Eigenschaft, Daten nur nach dem Commit einer Subtransaktion an den Vaterknoten weiterzugeben, wird durch das geforderte Pipelining aufgehoben. Das Rücksetzverhalten bei den geschachtelten Transaktionen sieht vor, mit dem Vaterknoten auch alle

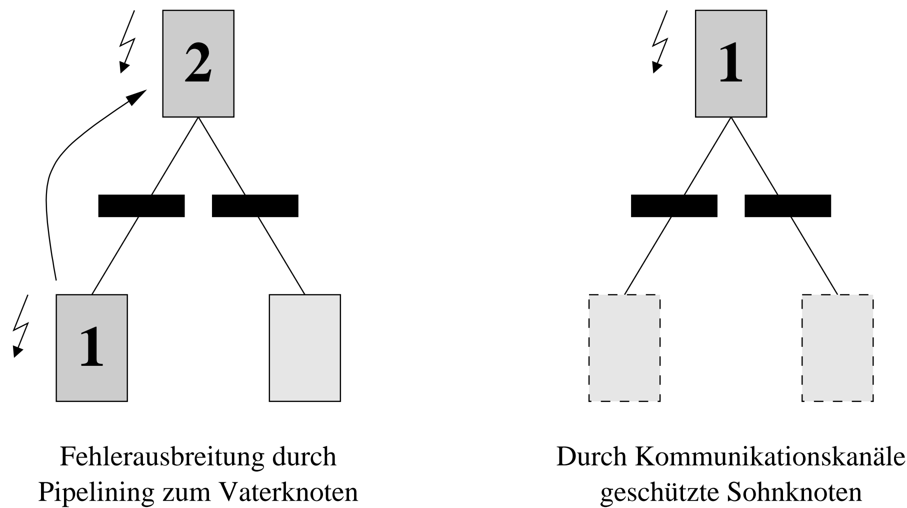

Abbildung 5.6: Das neue Rücksetzverhalten

seine Söhne abzubrechen. Wie in Abbildung 5.6 zu sehen ist, muß beim Auftreten eines Fehlers mit einem Sohn (Knoten 1) nun eventuell auch der Vater (Knoten 2) zurückgesetzt werden, da dieser durch das eingeführte Pipelining fehlerhafte Daten gelesen haben kann. Ein Rücksetzen in Richtung der Wurzel gibt es in dieser Form bei den geschachtelten Transaktionen nicht. Umgekehrt muß nun mit dem Rücksetzen des Vaters nicht notwendigerweise der Sohn rückgesetzt und neu gestartet werden. Falls die produzierten Ergebnisse des Sohnes noch in einem gespeicherten Kommunikationskanal vorliegen, können diese beim Neustart des Vaters wiederverwendet werden, und ein Neustarten des Sohnes ist nicht notwendig.

Zur erneuten Bereitstellung der Zwischenergebnisse müssen keine umfangreichen, viel Overhead erzeugenden Maßnahmen ergriffen werden, wie sie für Recovery-Maßnahmen notwendig sind, da die Speicherung der Zwischenergebnisse auf permanenten Speichermedien teilweise schon durch die Parallelisierung veranlaßt wird oder diese Zwischenergebnisse noch im Cache des Systems vorhanden sein können. Außerdem ist das Modell nicht darauf angewiesen, daß die Zwischenergebnisse unter allen Umständen vorhanden sein müssen. Sind sie nicht vorhanden, so kann beim Neustart einer Transaktion nur kein Profit aus den noch existierenden Zwischenergebnissen gezogen werden. Eine Gefährdung der Konsistenz der Datenbank kann also nicht entstehen. Die Maßnahmen sind demnach nicht zwingend durchzuführen und das System kann frei entscheiden, wann Zwischenergebnisse aufzubewahren sind.

Mit diesen Vorüberlegungen läßt sich ein Modell mit den gewünschten Eigenschaften beschreiben. Dieses wird im folgenden Abschnitt vorgestellt.

5.7 Das Modell zur Fehlerbehandlung – FPT-Modell

In Abschnitt 5.5 wurde gezeigt, daß bisher kein geeignetes Konzept zur Behandlung beziehungsweise Tolerierung von Fehlern innerhalb einer Anfrage einer Transaktion existiert.

Das Modell der geschachtelten Transaktionen schien anfangs eine geeignete Strukturierung für Transaktionen zu besitzen, um die geforderten Ansprüche der hier gewünschten Aktivitätskontrolle zu erfüllen. Es ließ sich bei genauerer Betrachtung aber zeigen, daß die geschachtelten Transaktionen mit ihrer Strukturierung einer Transaktion nicht ohne große Änderungen des Modells auf die gegebene Problemstellung abbildbar sind. Weiter stellte sich heraus, daß geschachtelte Transaktionen durch ihre starken Restriktionen bei der Datenverarbeitung und den zu erwartenden sehr hohen Verwaltungsaufwand nicht die gewünschten Leistungsanforderungen erfüllen würden.

Die vorgestellten Sicherungspunkttechniken erwiesen sich als geeignet, Transaktionen im Fehlerfall auf den Zustand vor Beginn einer Anfrage zurückzusetzen. Das Konzept selbst bietet aber keine Vorgehensweise zur Fehlerbehebung beziehungsweise Fehlertolerierung an. Es bleibt dem Anwender überlassen, wie auf einen derartigen Fehler reagiert wird. Außerdem werden von den Sicherungspunkttechniken keine Aufsetzpunkte innerhalb einer einzelnen Anfrage unterstützt. Tritt ein Fehler innerhalb einer Anfrage auf, so muß die Transaktion mindestens bis auf den Zeitpunk vor Beginn dieser Anfrage zurückgesetzt werden. Dabei gehen alle bis zu diesem Zeitpunkt erzeugten Ergebnisse und verrichteten Arbeiten verloren.

In diesem Abschnitt wird ein Modell zur **F**ehlerbehandlung in **p**arallelen **T**ransaktionen, das FPT-Modell, vorgestellt, dessen Ziel es ist, die in Abschnitt 5.4 aufgestellten Forderungen zu erfüllen. Zu diesen zählt unter anderem, die oben beschriebenen Verluste im Fehlerfall zu vermeiden oder zu minimieren und eine automatische, d. h. nicht durch den Anwender gesteuerte Fehlerbehandlung zu ermöglichen, um so die in Abschnitt 5.4 beschrieben Klassen von Fehlern tolerieren zu können.

Das FPT-Modell soll den fehlerbedingten Leistungsverlust reduzieren – die vor einem Fehler bereits geleistete Arbeit innerhalb einer Anfrage einer Transaktion soll erhalten bleiben und nicht prinzipiell verworfen werden müssen. Bevor das Datenbanksystem die gesamte Transaktion abbricht, wenn ein unkorrigierbarer Fehler auftritt, soll es möglich sein, nur die fehlerbehafteten Teile abzubrechen und zu wiederholen. Erst wenn diese Maßnahme fehlschlägt, soll die einzelne Anfrage als fehlerhaft abgebrochen, und gegebenenfalls die ganze Transaktion zurückgesetzt werden [Zim 97, Drügh 99].

5.7.1 Subtransaktionskonzept

Um der Aktivitätskontrolle das Rücksetzen und das Wiederholen von Teilen einer Transaktion zu ermöglichen, ist eine stärkere interne Strukturierung der Transaktion notwendig, als dies bei flachen Transaktionen der Fall ist. Es muß möglich sein, die internen Vorgänge einer Transaktion auch im Modell der Transaktion zu beschreiben, um das Vorgehen im Fehlerfall in das Modell zu integrieren.

Dazu werden sowohl Vorgänge, d. h. Operationen als auch der Informationsfluß im FPT-Modell beschrieben. Zu diesem Zweck wird eine Transaktion aus zwei Arten von Komponenten gebildet. Teiloperationen einer Transaktion werden durch **Subtransaktionen** beschrieben. Die Informationsflüsse zwischen diesen Teiloperationen werden durch sogenannte **Datenflüsse** repräsentiert.

Mit diesen im folgenden beschriebenen Komponenten einer Transaktion wird es möglich sein, Fehler, die innerhalb einer Transaktion auftreten, einer oder einigen wenigen Komponenten zuzuordnen. Bei einer späteren Fehlerbehandlung muß dann nicht die gesamte Transaktion wiederholt werden, sondern es genügt, nur die vom Fehler betroffenen Komponenten neu zu starten.

5.7.1.1 Subtransaktionen

Um die Kontrolle über einzelne Teile der Transaktion zu ermöglichen, werden Operationen der Transaktion, die sich auf die Datenbank beziehen, auf Subtransaktionen abgebildet. Da diese Subtransaktionen ihre übergeordnete Transaktion beschreiben müssen, herrscht auf ihnen eine entsprechende Ordnung. Eine Transaktion besteht aus Subtransaktionen, die in der Form eines azyklischen, gerichteten Graphen geordnet sind (siehe Abbildung 5.7). Dabei werden alle von der Transaktion zu verrichtenden Aufgaben auf die einzelnen Subtransaktionen verteilt.

Jede Subtransaktion wird durch einen Knoten bezeichnet. Jede Kante symbolisiert die Weitergabe von Daten der erzeugenden Subtransaktion, dem Vorgänger, an ihre nachfolgenden Subtransaktionen. Ein solches Paar von Subtransaktionen wird als **benachbart** bezeichnet. Auf diese Weise wird die Datenverarbeitung der Transaktion beschrieben.

Benachbart:

Zwei Subtransaktionen einer Transaktion werden als benachbart bezeichnet, wenn sie im Graphen der Subtransaktionen durch eine Kante verbunden sind.

Subtransaktionen können in beliebiger Weise nebenläufig ausgeführt werden. Die einzige, indirekte Regulierung des parallelen Ablaufs der Subtransaktionen wird

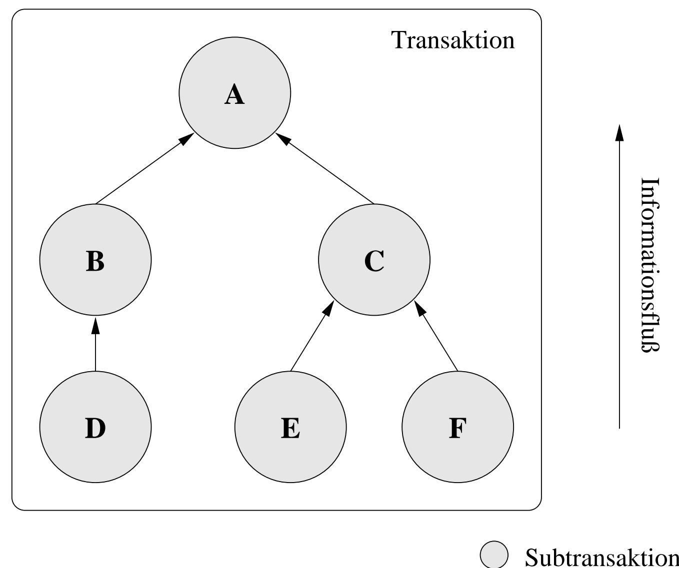

Abbildung 5.7: Transaktion als Baum von Subtransaktionen

dadurch erzeugt, daß der Nachfolger in einem Paar benachbarter Subtransaktionen auf die von seinem Vorgänger erzeugten Daten angewiesen ist und daher gegebenenfalls auf deren Produktion warten muß. Dies ist die einzige, auch zwangsläufig notwendige Beschränkung der Intra-Transaktionsparallelität, die durch das FPT-Modell vorgegeben wird. Insbesondere bestehen also keinerlei Beschränkungen für die Parallelausführung von Geschwister-Subtransaktionen.

5.7.1.2 Datenflüsse

Zwischen den einzelnen Subtransaktionen einer Transaktion werden Informationen ausgetauscht. Von Interesse sind hier nur die Nutzdaten – also die Daten, die eine Subtransaktion für ihren Nachbarn erzeugt und weiterleitet, da sie für die Datenverarbeitung essentiell sind. Diese Informationsflüsse sollen im folgenden als Datenfluß bezeichnet werden. Der Datenfluß zwischen zwei Subtransaktionen beschreibt den gesamten Umfang der Informationen, die während der Durchführung der übergeordneten Transaktion zwischen diesen Subtransaktionen ausgetauscht werden. Ein Datenfluß existiert, solange er als Träger von Informationen dient.

Der Datenfluß fügt sich in die Ordnung der Subtransaktionen ein – er ist ebenfalls gerichtet und darf keine Abhängigkeiten erzeugen, die zyklisch sind. In diesem Sinne gibt es bezüglich eines Datenflusses zwei Arten beteiligter Subtransaktionen, die datenproduzierenden und die datenverbrauchenden. In Abbildung 5.8 sind die Produzenten unterhalb und die Konsumenten oberhalb des zugehörigen Datenflusses angeordnet, so daß der gesamte Datenfluß in der schematischen

Abbildung von unten nach oben verläuft.

Der von einer Subtransaktion verarbeitete Datenfluß wird als Eingabe bezeichnet, der produzierte Datenfluß als Ausgabe. Die Ordnung der Subtransaktionen wird dadurch stärker definiert, daß die Eingabe einer Subtransaktion nur die Eingabe der Transaktion oder die Ausgabe einer benachbarten[3] Subtransaktion sein darf. Aus dieser Forderung ergeben sich die Vorschriften für die Zuordnung zwischen Datenflüssen und Subtransaktionen:

- Im Rahmen der auf den Subtransaktionen definierten Ordnung dürfen Datenflüsse nur zwischen benachbarten Subtransaktionen bestehen.
- Ein Datenfluß ist die Menge aller Ausgabedaten genau einer Subtransaktion.
- Ein Datenfluß dient mindestens einer Subtransaktion als Eingabe.

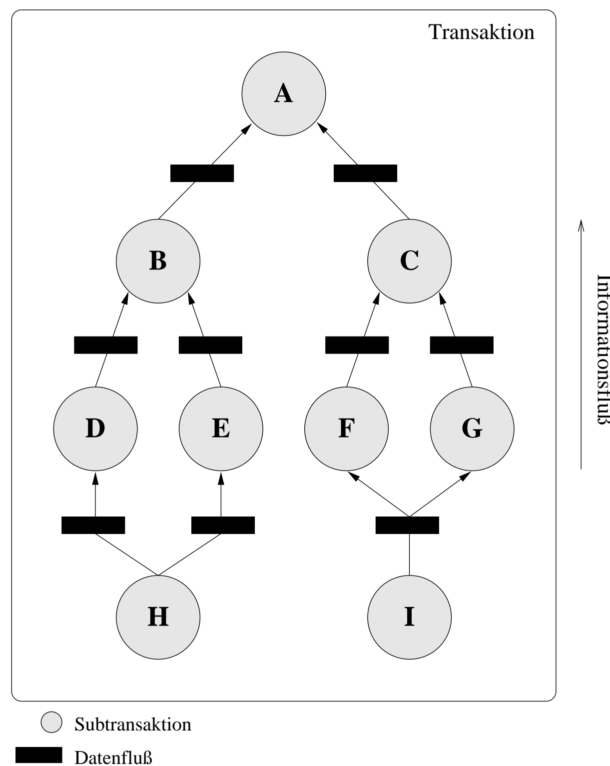

Abbildung 5.8: Datenflüsse in Transaktionsstruktur

[3] "benachbart" gemäß 5.7.1.1.

Abbildung 5.8 veranschaulicht die Struktur einer Transaktion unter Einbeziehung von Datenflüssen. Jeder Datenfluß ist durch einen schwarzen Balken symbolisiert. Insbesondere erlaubt das FPT-Modell einer Subtransaktion, Ausgaben in mehrere Datenflüsse zu schicken oder auch aus mehreren Datenflüssen ihre Eingaben zu erhalten.

Datenflüsse sind als Transportkanäle und Speicherungspuffer für die Ausgabedaten von Subtransaktionen anzusehen.

Weiter können, bezüglich eines Zeitpunktes, folgende drei Zustände eines Datenflusses definiert werden:

Vollständigkeit:
Ein Datenfluß wird als vollständig bezeichnet, wenn alle Daten, die die produzierende Subtransaktion im Laufe der gesamten Transaktion liefern wird, bereits im Datenfluß aufgenommen wurden und für die konsumierende Subtransaktion die Möglichkeit besteht, alle Daten, die den Datenfluß bereits erreicht haben, nochmals abzurufen.

Potentielle Vollständigkeit:
Ein Datenfluß gilt als potentiell vollständig, wenn er im weiteren Verlauf der Transaktion die Vollständigkeit erreichen wird. Dabei wird davon ausgegangen, daß die ihm zugeordneten produzierenden Subtransaktionen fehlerfrei beendet werden.

Unvollständigkeit:
Ein Datenfluß gilt als unvollständig, wenn er weder vollständig noch potentiell vollständig ist.

Bei diesen Definitionen der Vollständigkeit und potentiellen Vollständigkeit wird insbesondere die Situation ausgeschlossen, daß Daten, die den Datenfluß bereits durchlaufen haben, verworfen werden und diese Daten nur durch einen Neustart der produzierenden Subtransaktion wiedergewonnen werden können. Genau die Datenflüsse, bei denen diese Situation eingetreten ist, sind die unvollständigen Datenflüsse.

Die Definition der potentiellen Vollständigkeit deckt damit auch den Fall ab, daß noch überhaupt keine Daten zwischen den Subtransaktionen ausgetauscht wurden.

Daraus ergibt sich, daß jeder leere Datenfluß potentiell vollständig ist. Im zeitlichen Verlauf der Transaktion kann er vollständig oder unvollständig werden. Auch ein bereits vollständiger Datenfluß kann durch Verwerfen von Daten unvollständig werden. Den Zustand der Unvollständigkeit kann ein Datenfluß nicht mehr verlassen. Die möglichen Zustände und Zustandsübergänge von Datenflüssen sind in

Abbildung 5.9 dargestellt. Zusätzlich befinden sich unter den Zuständen Symbole für Datenflüsse. Diese Symbole geben die Zustände von Datenflüssen wieder und werden in den folgenden Abbildungen verwendet.

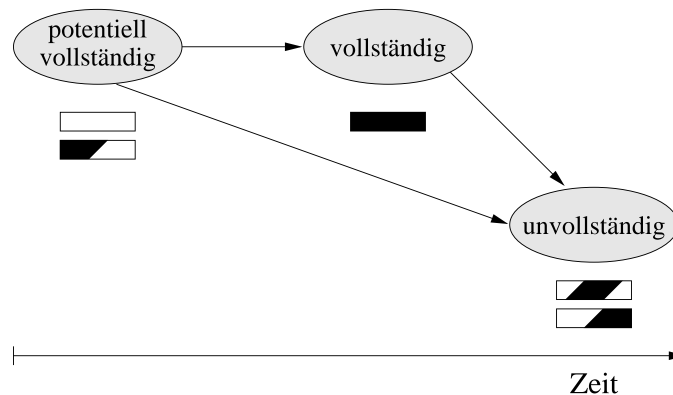

Abbildung 5.9: Zustandsübergänge von Datenflüssen

Die Vollständigkeit von Datenflüssen ist für das FPT-Modell von besonderer Bedeutung. Ein vollständiger Datenfluß beinhaltet die Ergebnisse aller Subtransaktionen, die an der Produktion seiner Daten beteiligt waren. Seine Funktion entspricht damit dem Aufbewahren des Zwischenergebnisses all seiner Vorgänger-Subtransaktionen. Bei potentiell vollständigen Datenflüssen ist im weiteren Verlauf der Transaktion damit zu rechnen, daß dieses Zwischenergebnis in Form eines vollständigen Datenflusses vorliegen wird.

Im Laufe der später vorgestellten Fehlerbehandlung (Abschnitt 5.7.3) des FPT-Modells werden diese Datenflüsse eine besondere Rolle spielen. Dort bieten vollständige und potentiell vollständige Datenflüsse die Möglichkeit, fehlerbehafteten Subtransaktionen, die neu gestartet werden müssen, die Eingabedaten erneut zur Verfügung zu stellen. Werden dagegen beim Neustart Zwischenergebnisse benötigt, die durch inzwischen unvollständige Datenflüsse gelaufen sind, müssen die an der Produktion dieser Zwischenergebnisse beteiligten Kind-Subtransaktionen neu gestartet werden, wodurch erhöhter Aufwand entsteht.

5.7.2 Fehler in Transaktionskomponenten

Im weiteren werden Transaktionsfehler dargestellt, die vom FPT-Modell wahrgenommen werden. Die folgende Aufzählung stellt dabei eine genauere Unterteilung der in Abschnitt 5.2.1 vorgestellten Transaktionsfehler dar. Sie beziehen sich auf

die Komponenten des FPT-Modells, also auf Subtransaktionen sowie Datenflüsse.

1. **Datenfehler in Komponenten:**

   Ein Datenfehler in einer Komponente liegt vor, wenn die Komponente entweder selbst fehlerhafte Daten produziert und diese an andere Komponenten weitergibt, oder wenn sie fehlerhafte oder ungültige Daten als Eingabe erhält. Die korrekte Datenverarbeitung innerhalb der Transaktion ist dann nicht möglich, die Komponente gilt als fehlerbehaftet.

   Der Fall, daß eine Komponente fehlerhafte Daten als Eingabe erhält, kann dadurch zustande kommen, daß die Komponente Daten einer anderen Komponente liest, die diese fehlerhaften Daten weitergeleitet, selbst produziert oder aus fehlerhaften Eingabedaten abgeleitet hat. Eine weitere Möglichkeit, fehlerhafte beziehungsweise ungültige Daten als Eingabe zu haben, besteht darin, daß die Komponente eine Version permanenter Datenobjekte gelesen hat, die sie aufgrund der ACID-Eigenschaften der Transaktion nicht hätte lesen dürfen. Dies kann in einem Datenbanksystem beispielsweise dann eintreten, wenn im Zuge der Konfliktauflösung zwischen Transaktionen einer Transaktion die Sperre auf Datenobjekte entzogen wird[4].

2. **Ausfall von Komponenten:**

   Der Ausfall einer Komponente verhindert den notwendigen Informationsaustausch zwischen den an der Transaktion beteiligten Subtransaktionen. Dieser Ausfall bewirkt zwangsläufig, daß mindestens einer Subtransaktion keine vollständige beziehungsweise korrekte Eingabe vorliegt. Die Subtransaktion kann damit ihre Funktion innerhalb der Transaktion nicht mehr wahrnehmen, da ihre Aufgabe über ihre Eingabe definiert wird. Aus Sicht der Transaktion ist dieser Funktionsverlust der Subtransaktion gleichbedeutend mit einem Ausfall der Subtransaktion. Als Ausfall wird sowohl ein technisches Versagen als auch eine Unerreichbarkeit der Komponente gewertet.

3. **Verklemmungen von Komponenten:**

   Eine weitere Klasse von Fehlern stellt die Verklemmung von Komponenten dar. Dabei arbeiten die Komponenten jede für sich korrekt, aber verschiedene beteiligte Subtransaktionen blockieren sich gegenseitig während der Erfüllung ihrer Aufgabe. Diese Situation kann beispielsweise bei Sperrkonflikten oder bei Verklemmungen in der Parallelverarbeitung (siehe Abschnitt 2.4.9) auftreten. Wiederum können die Subtransaktionen ihre Aufgaben nicht erfüllen und gelten aus Sicht der Transaktion als ausgefallen.

---

[4] Das Vorgehen in diesem Fall wird beispielhaft in Abschnitt 6.2.1.1 diskutiert.

5.7.3 Die Fehlerbehandlung

Für die in Abschnitt 5.7.2 beschriebenen Fehler existieren Behandlungsmaßnahmen, die eine Korrektur des aufgetretenen Fehlers erlauben und damit Fehlertoleranz bereitstellen.

Die grundlegende Idee dieser Art der Fehlerbehandlung ist, die den Fehler verursachende Komponente zu bestimmen und alle weiteren von diesem Fehler betroffenen Komponenten zu ermitteln. Diese Komponenten werden abgebrochen und neu gestartet. Alle betroffenen Komponenten werden aus der Transaktion und dem System entfernt, neue Instanzen dieser Komponenten werden erzeugt und gestartet, die dann die Aufgaben der abgebrochen Komponenten übernehmen. Zusammen mit den verbliebenen Transaktionskomponenten können die neu gestarteten Komponenten die Transaktion erfolgreich beenden.

Der Vorteil dieser Methode gegenüber einem Abbruch aller Transaktionskomponenten und dem Neustart der gesamten Transaktion besteht im wesentlichen darin, daß die Ergebnisse bereits verrichteter Arbeiten noch im System als Zwischenergebnisse vorhanden sind und beim Neustart wiederverwendet werden können. Die Zwischenergebnisse liegen in Form von vollständigen oder potentiell vollständigen Datenflüssen vor und repräsentieren einen kompletten Teilbaum von Subtransaktionen. Alle Subtransaktionen dieses Teilbaumes müssen beim Wiederaufsetzen nach einem Fehler nicht neu gestartet und deren Arbeit daher nicht erneut durchgeführt werden.

5.7.3.1 Reproduktion von Komponenten

Im folgenden wird beim Neustart einer abgebrochenen Komponente auch von deren Reproduktion gesprochen. Bei der Reproduktion von Komponenten kann nach der Art der Komponente unterschieden werden:

Reproduktion von Subtransaktionen:
Die Subtransaktion wird im Kontext der Transaktion als Ganzes wiederholt. Zu diesem Zweck müssen ihre korrekten Eingabedaten vorhanden sein, da diese ihre Aufgabe in der Transaktion definieren. Sind die Eingabedaten, d. h. die zuliefernden Datenflüsse nicht existent, so müssen sie reproduziert werden. Hier setzt eine der wesentlichen Strategien dieses Konzeptes ein, die der vollständigen oder potentiell vollständigen Datenflüsse. Sind solche Datenflüsse vorhanden, so kann ein rekursives Rücksetzen aller produzierenden Subtransaktionen bis hin zu den Blättern des Transaktionsbaumes verhindert werden.

Reproduktion von Datenflüssen:
Um einen Datenfluß zu reproduzieren, müssen die Ausgabedaten der Sub-

transaktionen, die die Eingabedaten des Datenflusses produzieren, existieren.

Die Reproduktion einer Komponente setzt also die Verfügbarkeit ihrer Eingabedaten voraus. Sind diese Eingabedaten nicht verfügbar, müssen die Komponenten, die die Eingabe erzeugen, reproduziert werden, um diese Daten zu erhalten.

Die folgenden Abschnitte definieren detailliert das Verfahren für die Auswahl der zu reproduzierenden Komponenten und beschreiben das Vorgehen beim Neustart einer Transaktion.

5.7.3.2 Die Fehlerbehandlung

Im weiteren stellt sich die Frage, wie beim Auftreten eines Fehlers in den Transaktionskomponenten vorzugehen ist. Die allgemeine Vorgehensweise wird sein, die von Fehlern betroffenen Komponenten abzubrechen und zusammen mit den verbliebenen Komponenten neu zu starten, also eine Reproduktion der betroffenen Komponenten. Dabei stellt die Identifizierung der betroffenen Komponenten den schwierigsten Teil der Aufgabe dar. Zum einen soll ein möglichst kleiner Teil der Komponenten abgebrochen werden, um beim Neustart viel Nutzen aus bereits verrichteter Arbeit ziehen zu können. Zum anderen ist es natürlich notwendig, mindestens so viele Komponenten neu zu starten, daß ein korrektes Ergebnis der Transaktion gesichert ist.

Das entscheidende Kriterium dabei ist, die Korrektheit der gesamten Transaktion zu gewährleisten. Diese Korrektheit der Transaktion kann durch die im folgenden formulierten Anforderungen an die verbleibenden Komponenten garantiert werden.

Anforderungen

Eine Menge M von Komponenten einer Transaktion kann beim Neustart der Transaktion wiederverwendet werden, wenn alle Komponenten folgende Anforderungen erfüllen:

1. Für jede Subtransaktion gilt:
   (a) Sie ist nicht ausgefallen.
   (b) In ihr sind keine Datenfehler aufgetreten.
   (c) All ihre Eingabedatenflüsse sind ebenfalls in der Menge M vorhanden.
2. Für jeden Datenfluß gilt:
   (a) Er ist nicht ausgefallen.

(b) In ihm sind keine Datenfehler aufgetreten.

(c) Er wird nach der Reproduktion wieder dieselbe Eingabe bekommen wie zuvor. Zusätzlich bekommt er eventuell noch weitere Eingabedaten, falls seine Eingabe vor der Reproduktion noch nicht komplett war. Nur falls die Eingabe schon komplett gewesen ist, ist es auch möglich, daß er keine Eingabe bekommt.

(d) Falls durch den Datenfluß schon Daten transportiert wurden, so ist sichergestellt, daß er seine bisher schon weitergeleitete Ausgabe korrekt vervollständigt.

(e) Ist eine seiner nachfolgend von ihm lesenden Subtransaktionen nicht in M, so muß dieser seine komplette Ausgabe zur Verfügung gestellt werden können.

Begründung der Korrektheit

Bei den Anforderungen 1a und 2a ist unmittelbar ersichtlich, daß ausgefallene Komponenten nicht in M vorhanden sein können, da sie nicht mehr existent sind.

Ebenso ersichtlich ist, daß fehlerbehaftete Komponenten nicht in der Menge M erscheinen dürfen. Dies wird durch die Anforderungen 1b und 2b sichergestellt.

Die Anforderung 2d garantiert, daß ein Datenfluß eine Reproduktion seiner Eingabe-Subtransaktion nach oben vor anderen Subtransaktionen verbergen kann. Haben also Subtransaktionen aus dem Datenfluß schon Daten gelesen, so ist nach der Reproduktion sichergestellt, daß die Folge der Daten, die sie erhalten, identisch mit der Folge der Daten ist, die sie erhalten hätten, wenn kein Fehler aufgetreten wäre.

Für ein korrektes Arbeiten einer Subtransaktion nach einer Reproduktion scheint es notwendig zu sein, auch die folgende Forderung aufzustellen: "Falls die Subtransaktion schon Daten als Eingabe gelesen hat, so ist sichergestellt, daß nach einem Fehler im weiteren Verlauf der Transaktion diese Eingabe korrekt vervollständigt wird.". Die Erfüllung dieser Anforderung für alle Subtransaktionen aus M ergibt sich allerdings automatisch aus den Anforderungen 2d und 1c und muß somit nicht gesondert aufgeführt werden. Die Anforderung 2d stellt daher die korrekte Beendigung von Subtransaktionen, die ebenfalls in M sind, sicher.

Die Anforderung 2e ist das Gegenstück zu 2d. Sie garantiert die korrekten und damit auch vollständigen Eingaben für zu reproduzierende Subtransaktionen, also solche, die sich nicht in M befinden. Die Anforderung garantiert, daß eine Subtransaktion nach ihrem Neustart auf noch vollständige oder potentiell vollständige Datenflüsse zugreifen kann, wenn der Datenfluß über Anforderung 2e in M enthalten ist. Sind die Datenflüsse, die die Eingaben der Subtransaktionen erzeugen, nicht in M, so müssen sie, wie die Subtransaktionen, neu gestartet wer-

den. Dadurch werden den Subtransaktionen ebenfalls korrekte Eingabedaten zur Verfügung gestellt.

Die Anforderung 2c ist zusätzlich zu 2d für den Fall notwendig, daß ein Datenfluß schon alle Daten transportiert hat, seine Eingabe-Subtransaktion reproduziert wird und diese dabei eine andere Ausgabe als zuvor erzeugt. In diesem Fall sind die Daten, die der Datenfluß schon weitergeleitet hat, als inkorrekt anzusehen. Ist der Datenfluß noch nicht vollständig, so wird er gegen Anforderung 2d verstoßen. Hat er aber zuvor schon alle Daten transportiert, so greift 2d nicht und Anforderung 2c wird benötigt. Sie ermöglicht auch nach dem kompletten Durchlauf aller Daten durch einen Datenfluß, diesen nachträglich für ungültig zu erklären.

Aus der Kombination von Anforderung 1c mit 2c folgt wiederum die Korrektheit der Subtransaktion, analog zur Kombination der Anforderungen 2d und 1c.

Diese Anforderungen sind nicht als solche Anforderungen konzipiert, die einzeln für jede Komponente geprüft werden. Sie müssen immer für die komplette Menge M aller Komponenten gelten, die nach dem Neustart wiederverwendet werden sollen. Daß dies nicht für jede Komponente einzeln, unabhängig von den anderen in M vorhandenen Komponenten möglich ist, ist unmittelbar aus den wechselseitigen Beziehungen zwischen den Komponenten bei den Anforderungen (beispielsweise 1c oder 2c) ersichtlich.

Zu beachten ist, daß bei den Anforderungen keine Unterscheidung getroffen wird, ob Komponenten noch untätig sind, bereits zu arbeiten begonnen haben oder bereits beendet sind. Sie gelten alle bis zum Ende der Transaktion als existent und können somit in die Fehlerbehandlung mit einbezogen werden.

Mit diesen Anforderungen läßt sich die im folgenden beschriebene Vorgehensweise für die Fehlerbehandlung einer Transaktion bestimmen.

Algorithmus zur Fehlerbehandlung:

1. **Erkennungsphase:**
   Bestimmung einer Menge M von Komponenten, die beim Neustart wiederverwendet werden können. M sollte dabei möglichst maximal gewählt werden.
2. **Beseitigungsphase:**
   Alle Komponenten der Transaktion, die nicht in M sind, werden abgebrochen.
3. **Reproduktionsphase:**
   Alle abgebrochenen Komponenten werden durch neue Komponenten mit identischem Auftrag ersetzt.

Durch diese Vorgehensweise wird erreicht, daß die Transaktion, wie ursprünglich geplant, erfolgreich beendet werden kann und dabei möglichst viele, vom aufgetretenen Fehler unberührte Zwischenergebnisse wiederverwendet werden können.

5.7.3.3 Fehlererkennung

Der für den Algorithmus zur Fehlerbehandlung wichtigste Schritt ist die Erkennungsphase. Hierbei müssen die zu reproduzierenden Komponenten, beziehungsweise die Komponenten, die wiederverwendet werden können, identifiziert werden. Die Anforderungen, denen die wiederzuverwendenden Komponenten genügen müssen, wurden in Abschnitt 5.7.3.2 vorgestellt.

Dieser Abschnitt beschreibt detailliert, wie die zu reproduzierenden Komponenten in der Erkennungsphase bestimmt werden können.

1. **Ermittlung der direkt von Fehlern betroffenen Komponenten:**
   Im ersten Schritt werden die direkt von Fehlern betroffenen Komponenten zu der Liste der zu reproduzierenden Komponenten hinzugenommen. Dies sind ausgefallene beziehungsweise nicht erreichbare Komponenten oder Komponenten, in denen Datenfehler entstanden sind.

2. **Vermeidung der Ausbreitung von Datenfehlern:**
   Eine Bestimmung aller von einem Datenfehler betroffenen Komponenten findet über Datenabhängigkeiten statt.

   Datenabhängigkeit:

   > Daten, die durch Verarbeitung aus anderen Daten entstehen, sind von diesen abhängig.
   >
   > Insbesondere müssen Daten, die von fehlerhaften Daten abhängig sind, ebenfalls als fehlerhaft angesehen werden.

   Diese Abhängigkeit schließt explizit mehrere Verarbeitungschritte mit ein, d. h. Datenabhängigkeit kann transitiv über einen ganzen Subtransaktionsgraph entstehen.

   Um alle Komponenten zu bestimmen, in denen Datenfehler aufgetreten sind, wird von allen Komponenten ausgegangen, bei denen in Schritt 1 erkannt wurde, daß sie fehlerhafte Daten produziert haben. Alle weiteren Komponenten, die direkt diese mit Datenfehlern belasteten Ausgabedaten oder von den Ausgabedaten abhängige Daten gelesen haben, sind ebenfalls mit Datenfehlern belastet und erweitern die Liste der zu reproduzierenden Komponenten.

   Beispiel 5.10 zeigt einen Subtransaktionsbaum, in dem ein Datenfehler in Knoten F aufgetreten ist. Seine Ausgabedaten sind damit als fehlerhaft

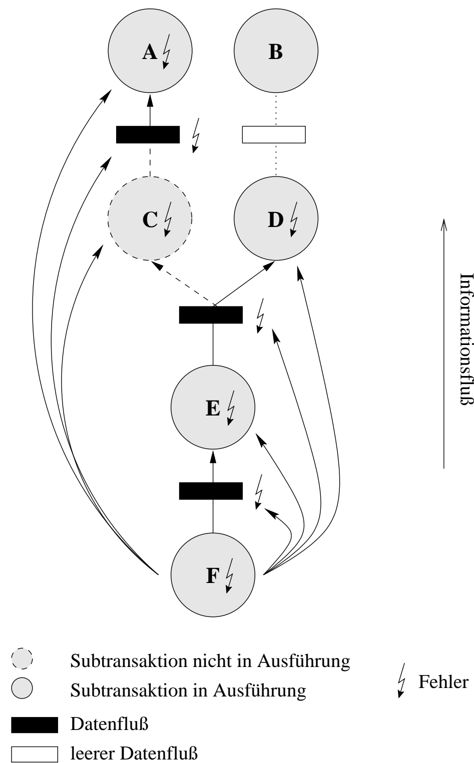

Abbildung 5.10: Ausbreitung von Datenfehlern

anzusehen. Alle weiteren, oberhalb von F in Richtung des Informationsflusses liegenden Knoten, haben die Ausgabedaten von F entweder direkt oder indirekt über abhängige Daten gelesen und sind daher ebenfalls mit Datenfehlern belastet. Einzige Ausnahme bildet der Knoten B, der noch keine Daten über seinen Eingabedatenfluß gelesen hat.

Bei der Bestimmung der mit Datenfehlern belasteten Komponenten ist es unbedeutend, ob sich die identifizierten Komponenten noch in Ausführung befinden oder bereits beendet sind. In beiden Fällen sind sie als fehlerhaft anzusehen. In Beispiel 5.10 wird auch Konten C, obwohl schon beendet, als fehlerhaft identifiziert.

Durch diese zwei vorangegangenen Schritte sind die direkt von Fehlern betroffenen Komponenten bestimmt. Dadurch werden jeweils die Punkte a und b der Anforderungen an die wiederzuverwendenden Komponenten aus Abschnitt 5.7.3.2 erfüllt.

Zunächst erscheint es naheliegend, nur diese direkt von Fehlern betroffenen Komponenten zu reproduzieren und die restlichen Komponenten weiterlaufen zu lassen. Allerdings kann dabei kein erfolgreiches Beenden der Transaktion garantiert

werden, da es zu sogenannten reproduktionsbedingten Fehlern kommen kann.

Reproduktionsbedingte Fehler:

Wenn im Laufe einer Transaktion ein Fehler auftritt und nur die direkt vom Fehler betroffenen Komponenten reproduziert werden, so können nach der Reproduktion Fehler in Komponenten entstehen, die nicht direkt vom ursprünglichen Fehler betroffen waren. Diese Fehler, die infolge der Reproduktion von Komponenten entstehen, werden als reproduktionsbedingte Fehler bezeichnet.

Diese Art von Fehlern und die Maßnahmen zu ihrer Vermeidung sind in den folgenden Abschnitten 3 bis 5 beschrieben. Ihr mögliches Auftreten gilt es im voraus zu erkennen und durch Aufnahme der verursachenden Komponenten in die Liste der zu reproduzierenden Komponenten zu vermeiden. Die im folgenden beschriebenen Vorgehensschritte stellen die Erfüllung der restlichen Anforderungen an die wiederzuverwendenden Komponenten aus Absatz 5.7.3.2 sicher.

3. **Bereitstellen von Eingabedaten:**

   Wird eine Komponente reproduziert, so müssen ihr, damit sie korrekt arbeiten kann, korrekte Eingabedaten vorliegen. Bei der Reproduktion von Komponenten kann aber der Fall eintreten, daß der reproduzierten Komponente keine oder keine vollständigen Eingaben mehr zur Verfügung stehen (*Eingabeverlust*).

   Ist die zu reproduzierende Komponente ein Datenfluß und ist der Datenfluß leer, so muß nur der Datenfluß selbst reproduziert werden. Sind durch ihn schon Daten geflossen, so muß auch die Subtransaktion, die seine Eingabedaten erzeugt, reproduziert werden, da das FPT-Modell keine Möglichkeit vorsieht, Eingabedaten eines Datenflusses von einer Subtransaktion erneut bereitstellen zu lassen. Pufferung von Daten wird ausschließlich in den Datenflüssen vorgenommen.

   Ist die zu reproduzierende Komponente eine Subtransaktion, so muß sichergestellt sein, daß ihr all ihre Eingabedatenflüsse korrekt vorliegen werden. Im Fall, daß diese Datenflüsse vollständig oder potentiell vollständig sind, ist dies gewährleistet und die Datenflüsse müssen nicht reproduziert werden. Sind Datenflüsse, die Eingabe der Subtransaktion sind, unvollständig, so müssen diese reproduziert werden.

   Ohne diese Bereitstellung von Eingabedaten kann beim Neustart der Transaktion ein sogenannter reproduktionsbedingter Fehler entstehen, da einer Komponente keine korrekten Eingabedaten vorliegen würden. Die Transaktion könnte somit nicht erfolgreich beendet werden.

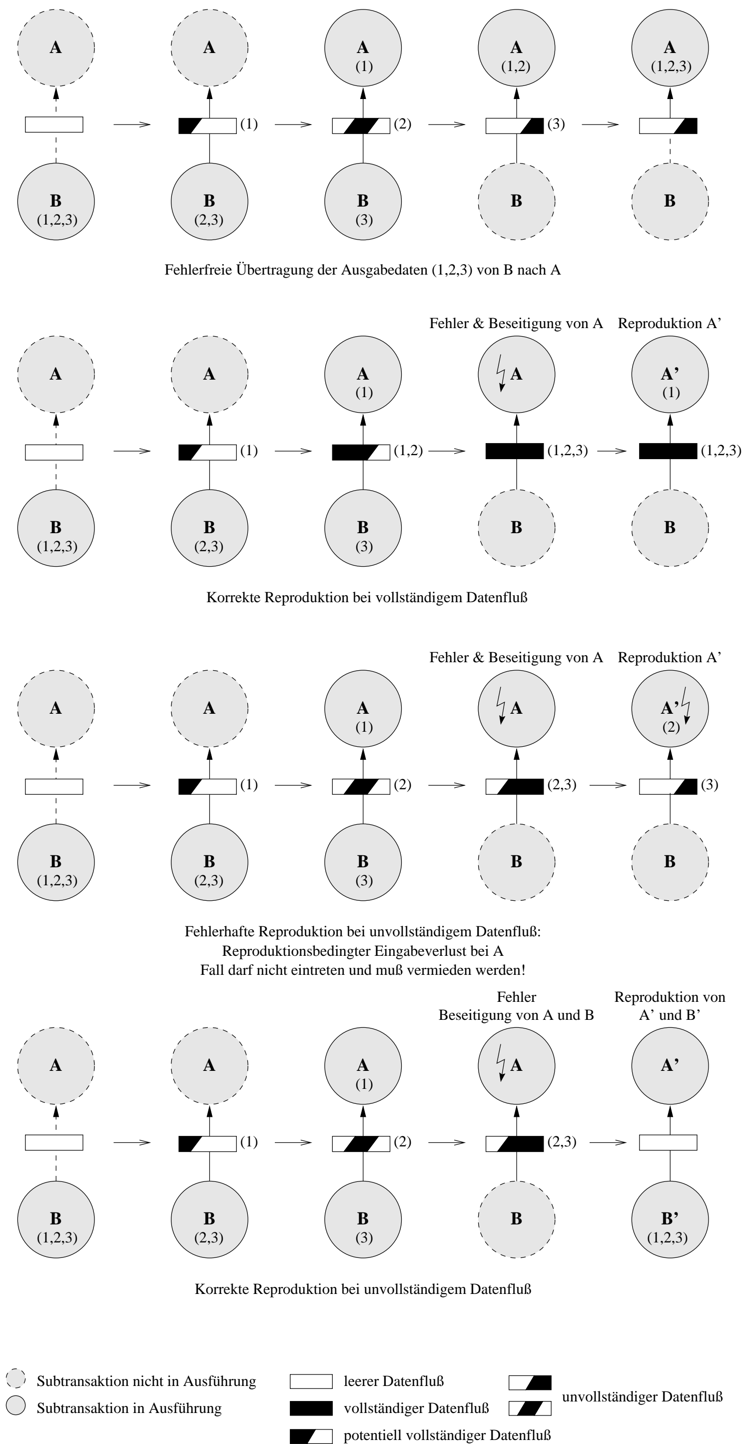

Abbildung 5.11: Reproduktion mit Verlust von Eingabedaten

Die so zusätzlich identifizierten Komponenten werden ebenfalls in die Liste der zu reproduzierenden Komponenten aufgenommen.

Die Beispiele aus Abbildung 5.11 veranschaulichen diesen Sachverhalt.
In der ersten Zeile ist der normale Ablauf der Übertragung von Daten von Subtransaktion B zu Subtransaktion A dargestellt.
Die zweite Zeile zeigt den Fall, in dem in der konsumierenden Subtransaktion A ein Fehler auftritt und A reproduziert wird. Nach ihrem Neustart liegt ihr eine korrekte Eingabe in Form eines vollständigen oder potentiell vollständigen Datenflusses vor. Somit kann A ihre Arbeit erfolgreich beenden.

Im Fall, der in der dritten Zeile dargestellt ist, wird die Subtransaktion A ebenfalls nach dem Auftreten eines Fehlers reproduziert. Allerdings liegt ihr nach dem Neustart ein unvollständiger Datenfluß als Eingabe vor. Somit kommt es zu einem reproduktionsbedingten Fehler in A.
Um diesen Fehler zu vermeiden und somit eine korrekte Reproduktion zu gewährleisten, muß auch dieser Datenfluß reproduziert werden. Im vorliegenden Beispiel muß demnach, um den Datenfluß zu reproduzieren, auch die Subtransaktion B reproduziert werden. Dieser korrekte Ablauf der Fehlerbehandlung ist in der vierten Zeile dargestellt.

Durch den hier beschriebenen Schritt bei der Identifizierung der zu reproduzierenden Komponenten wird der Fall aus Zeile drei ausgeschlossen.

4. **Korrekter Informationsfluß:**

   Das FPT-Modell erlaubt, daß konsumierende Komponenten bereits Daten erhalten und verarbeiten, bevor die Eingabeverarbeitung der produzierenden Komponenten abgeschlossen ist (Pipelining). Die konsumierende Komponente erhält ihre Eingabe dann nicht als Ganzes, sondern dynamisch, je nach Fortschritt der Verarbeitung des Produzenten. Hat nun die datenverbrauchende Komponente zwar einen Teil, nicht aber die vollständige Eingabe erhalten und erkennt die konsumierende Komponente die Reproduktion des Produzenten nicht, so fügt sie dessen reproduzierte Ausgabedaten an den bereits erhaltenen Teil an. Damit kann nicht mehr sichergestellt werden, daß die Eingabedaten des Konsumenten korrekt sind. Mit einer Verarbeitung dieser Eingabedaten würde der Konsument fehlerhaft werden und es läge ein reproduktionsbedingter Fehler vor.

   Abbildung 5.12 zeigt einen solchen Fall. Subtransaktion B überträgt Daten an Subtransaktion A. Nun wird B abgebrochen und reproduziert. B produziert dieselben Daten wie zuvor und sendet diese über den Datenfluß zu A. Subtransaktion A, die durch den Datenfluß von B abgeschirmt ist und von deren Reproduktion nichts mitbekommen hat, erhält nun einen inkorrekten Eingabedatenstrom und arbeitet deshalb fehlerhaft.

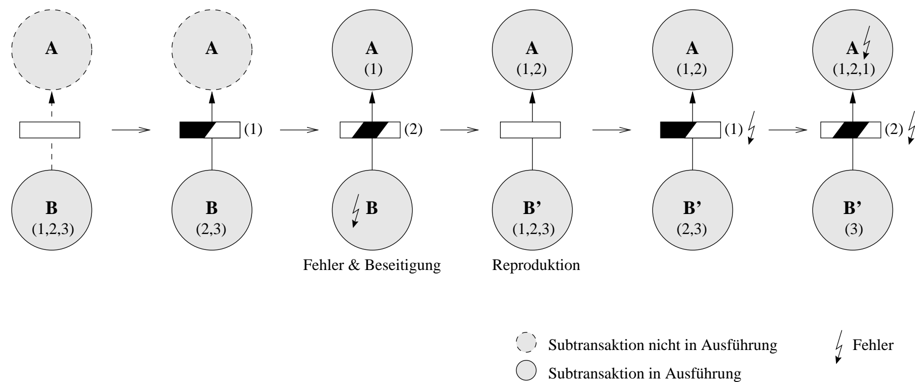

Abbildung 5.12: Störung des Informationsflusses

Eine derartige *Störung des Informationsflusses* tritt nur bei nicht leeren und noch nicht komplett übertragenen Datenflüssen auf.

Im Fall, daß eine Subtransaktion reproduziert wird, müssen also auch alle von ihr produzierten Datenflüsse, die weder leer noch komplett übertragen wurden, reproduziert werden. Daraus ergibt sich logischerweise auch, daß die Subtransaktionen, deren Eingaben diese Datenflüsse sind, reproduziert werden müssen. Durch diese Maßnahme wird das Auftreten dieser Art reproduktionsbedingter Fehler vermieden. In den Anforderungen ist dies durch den Punkt 1c repräsentiert.

Eine Optimierung dieses Falles kann durch eine Erweiterung der Datenflüsse erzielt werden. Besitzen Datenflüsse die Eigenschaft, Störungen, die durch erneutes Senden identischer Daten entstehen, zu kompensieren, so können diese Störungen vor dem Konsumenten verborgen werden. Dazu müssen die Datenflüsse wiederholt gesendete Daten erkennen und diese nicht weiterleiten. Für den Konsumenten bleibt damit die Reproduktion des Produzierten verborgen, er erhält die identische Sequenz an Daten, die er ohne das Auftreten eines Fehlers bekommen hätte. Somit kann der Konsument korrekt arbeiten und muß nicht reproduziert werden.

Mit dieser zusätzlichen Eigenschaft der Datenflüsse kann diese Art von reproduktionsbedingtem Fehler nie eintreten. Daher kann dieser Fall aus der Fehlerbehandlung ausgenommen werden. Datenflüsse, die die beschriebene Eigenschaft besitzen, werden im folgenden als **kompensierende Datenflüsse** bezeichnet. Die Vorteile ihrer Verwendung werden nochmals in den Abschnitten 5.7.3.4, 6.3.2 sowie 6.3.5.1 veranschaulicht.

5. **Behandlung nichtdeterministischer Komponenten:**
Werden Komponenten reproduziert, die nichtdeterministisch arbeiten, so

müssen all ihre bisherigen Ausgaben als fehlerhaft angesehen werden. Der Grund hierfür ist, daß bei der Reproduktion solcher Komponenten potentiell andere Ausgaben entstehen als vor der Reproduktion. Dadurch ist nicht mehr gewährleistet, daß alle auf diesen Ausgabendaten und den davon abhängigen Daten arbeitenden Komponenten einen aktuellen und damit korrekten Stand der Transaktion widerspiegeln.

Ähnlich wie bei der Bestimmung aller von Datenfehlern betroffenen Komponenten, müssen hier auch alle auf abhängigen Daten der reproduzierten, nichtdeterministischen Komponenten arbeitenden Komponenten ebenfalls reproduziert werden.

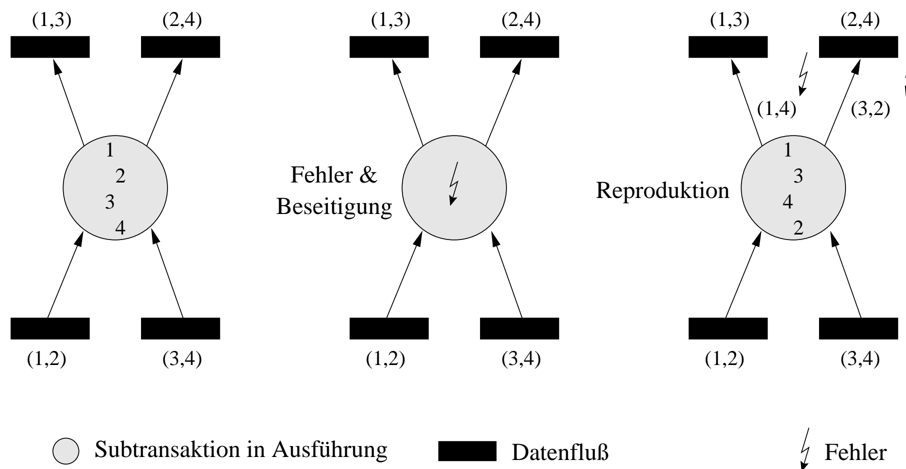

Abbildung 5.13: Reproduktionsfehler bei nichtdeterministischen Komponenten

Als Beispiele für derartige nichtdeterministische Komponenten kommen unter anderem Subtransaktionen in Frage, die Ergebnisse aus Zufallszahlen erzeugen, oder deren Ergebnisse von der nichtvorhersagbaren, zeitlichen Abfolge von Ereignissen abhängen. Abbildung 5.13 zeigt eine Subtransaktion, die von zwei Datenflüssen gleichzeitig liest und die gelesenen Daten abwechselnd auf zwei Datenflüsse ausgibt. Die Ausgabe auf jedem Ausgabedatenfluß hängt stark vom zeitlichen Eintreffen der Daten bei der Subtransaktion ab und ist nicht vorhersagbar. Im Fall der Reproduktion der Subtransaktion würde sie zwar wieder dieselben Eingaben lesen, aber, wie hier gezeigt, auf dem einen Ausgabedatenfluß eine andere Reihenfolge der Daten erzeugen. Im gezeigten Beispiel kamen vor der Reproduktion die Daten in der Reihenfolge $(1, 2, 3, 4)$ an, nach der Reproduktion in der Reihenfolge $(1, 3, 4, 2)$.

Die Schritte 3 bis 5 müssen so lange iteriert werden, bis die Liste der zu reproduzierenden Komponenten nicht mehr wächst. Ein einmaliges Durchlaufen der Schritte reicht nicht aus, da bei jedem Schritt wieder neue zu reproduzierende Komponenten bestimmt werden. Für diese müssen durch die Schritte 3 bis 5 wiederum die reproduktionsbedingten Fehler vermieden werden.

5.7.3.4 Algorithmus zur Fehlerbehandlung

Mit den oben erläuterten Schritten läßt sich der Algorithmus zur Fehlerbehandlung wie im folgenden beschrieben verfeinern.

Die Fehlerbehandlung unterteilt sich in die Erkennungs-, Beseitigungs- und Reproduktionsphase. In der Erkennungsphase werden zunächst alle von einem Fehler betroffenen und bedrohten Komponenten festgestellt (siehe reproduktionsbedingte Fehler, Abschnitt 5.7.3.3, Punkte 3 bis 5). Zusammen sind dies genau die Komponenten, die nicht in der Menge M (siehe Abschnitt 5.7.3.2) enthalten sind. In der anschließenden Beseitigungsphase werden diese Komponenten beseitigt und schließlich in der Reproduktionsphase neu gestartet. Zusammen mit den restlichen Komponenten, denjenigen aus der Menge M, können sie die Transaktion dann zu einem erfolgreichen Abschluß bringen.

Algorithmus zur Fehlerbehandlung:

1. **Erkennungsphase:**
   Bestimmung aller zu reproduzierenden Komponenten durch Hinzufügen von:
   (a) direkt von Fehlern betroffenen Komponenten,
   (b) allen von Datenfehlern betroffenen Komponenten,
   (c) allen Komponenten, die benötigt werden, um den zu reproduzierenden Komponenten korrekte Eingabedaten zur Verfügung zu stellen,
   (d) allen Datenflüssen, bei denen eine Störung des Informationsflusses vorliegt,
   (e) allen Komponenten, die abhängige Daten von zu reproduzierenden, nichtdeterministischen Komponenten gelesen haben.

   Die Schritte 1c bis 1e werden so lange wiederholt, bis keine weiteren zu reproduzierenden Komponenten mehr gefunden werden.
   Schritt 1d entfällt, wenn, wie in Abschnitt 5.7.3.3 Punkt 4 beschrieben, kompensierende Datenflüsse unterstützt werden.
2. **Beseitigungsphase:**
   Alle Komponenten der Transaktion, die in Schritt 1 identifiziert wurden, werden abgebrochen.

3. **Reproduktionsphase:**
   Alle in Schritt 2 abgebrochenen Komponenten werden durch neue Komponenten mit identischem Auftrag ersetzt.

Bei einer Verklemmung von Komponenten muß zusätzlicher Aufwand betrieben werden. Da kein Komponentenfehler auf den verklemmten Komponenten vorliegt, würde diese Situation nicht entsprechend behandelt. Um den Konflikt dennoch aufzulösen, wird ein möglichst kleiner Teil der verklemmten Komponenten als fehlerhaft klassifiziert und anschließend die oben beschriebene Maßnahme ergriffen. Dies führt zur Beseitigung und anschließenden Reproduktion der als fehlerhaft klassifizierten Komponenten. Sollte dieses Vorgehen wiederholt nicht zur Lösung der Verklemmung führen, werden alle verklemmten Komponenten als fehlerhaft betrachtet. Dabei hofft man, daß die Verklemmung nach dem Wiederaufsetzen nicht nochmals auftritt.

Führen diese Eingriffe nicht zum erfolgreichen Beenden der Transaktion, wird die gesamte Transaktion abgebrochen.

5.7.4 Bewertung des Modells

Eine Bewertung des neuen FPT-Modells im Rahmen der Aktivitätskontrolle findet an Hand des Kriteriums einer möglichst hohen Reduzierung des Leistungsverlustes im Fehlerfall statt. Die Anwendung des FPT-Modells gilt als erfolgreich, wenn Abbruch und Wiederholung der Transaktion einen höheren, mindestens aber den gleichen Aufwand verursachen wie der Einsatz des vorgeschlagenen Modells.

Diese Situation ist im Idealfall immer gegeben, da kein fehlerbehafteter Teil größer sein kann als die Transaktion selbst. Das Wiederaufsetzen dieses Teils sollte also auch im ungünstigsten Fall nicht mehr Aufwand verursachen als das Wiederholen der Transaktion. In einem realen System kann allerdings Verwaltungsarbeit notwendig werden, die durch das separate Aufsetzen von Transaktionskomponenten entsteht. Dies können beispielsweise Kosten für den Algorithmus zur Bestimmung der Komponenten oder zusätzlich zu sammelnde Daten sein. Ist diese Art von Mehraufwand so hoch, daß die Anwendung der Aktivitätskontrolle statistisch größer ist als das Wiederholen der Transaktion, sollte sie nicht zum Einsatz kommen. Eine solche Bewertung muß jeweils am realen System durchgeführt werden.

Man kann aber zumindest feststellen, inwieweit welche Faktoren einen positiven Einfluß auf die Effizienz des Konzepts haben. Für eine Bewertung der Effizienz wird die Abweichung zwischen dem von der Aktivitätskontrolle wiederaufgesetzten und dem ursprünglich fehlerbehafteten Teil betrachtet. Diese Abweichung wird bestimmt, durch den Grad der Fehlerausbreitung und die Verfügbarkeit von Zwischenergebnissen der Transaktion.

Hinsichtlich der Verfügbarkeit von Zwischenergebnissen sind die im Transaktionsmodell angesprochenen vollständigen und potentiell vollständigen Datenflüsse von Bedeutung. Je nachdem, wie kostenaufwendig einzelne Subtransaktionen sind, sollten ihre Ergebnisse in diesen Datenflüssen aufbewahrt werden. Die Effizienz der Aktivitätskontrolle hängt daher auch davon ab, wie strategisch günstig und in welchem Ausmaß persistente Datenflüsse im Ausführungsplan der Transaktion verwendet werden. Je höher die Verfügbarkeit, desto geringer ist tendenziell die oben genannte Abweichung.

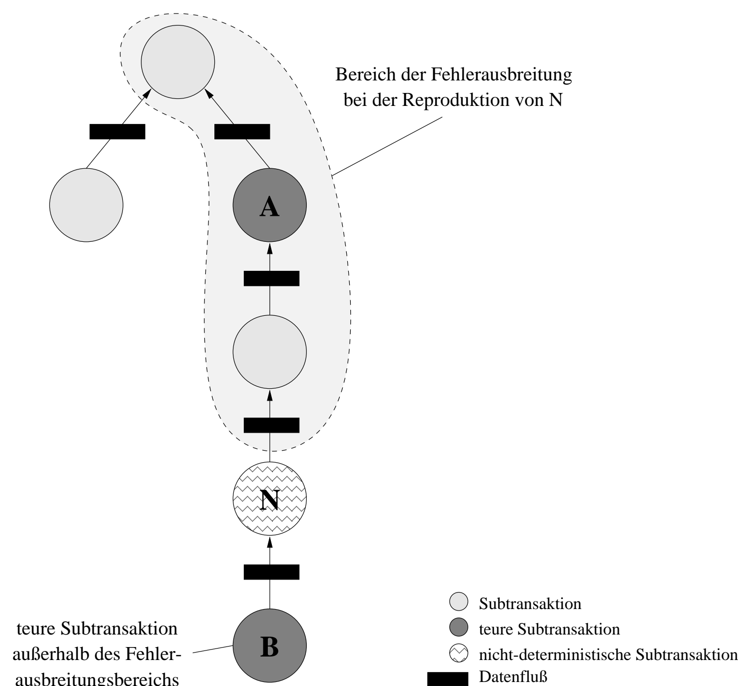

Abbildung 5.14: Bedeutung der Position nichtdeterministischer Komponenten

Ein weiterer Bewertungsfaktor ist die Fehlerausbreitung. Die Fehlerausbreitung wird um so mehr beschränkt, je mehr die Transaktionskomponenten eines Datenbanksystems voneinander isoliert sind. Der Isolationsgrad von Komponenten ist an Hand des Subtransaktionsgraphen abschätzbar. In einem stark verzweigten Graph ist der Isolationsgrad im allgemeinen höher, in linearen Graphen tendenziell geringer. Dies ist darin begründet, daß Subtransaktionen auf verschiedenen Datenwegen keine voneinander abhängigen Daten verarbeiten können. Datenbanksysteme, die Transaktionen bevorzugt in einem linearen Subtransaktionsgraph realisieren, sind dann auch anfälliger für die Fehlerausbreitung.

Schließlich spielt die Bedrohung durch reproduktionsbedingte Fehler für die Ef-

fizienz der Fehlerbehandlung eine entscheidende Rolle. Komponenten, die zwar fehlerfrei sind, aber von reproduktionsbedingten Fehlern bedroht werden, müssen ebenfalls reproduziert werden.

Die Bedeutung vollständiger und potentiell vollständiger Datenflüsse wurde oben schon beschrieben. Sie verhindern den Eingabeverlust und damit eine Ausbreitung reproduktionsbedingter Fehler im Subtransaktionsgraph nach unten (*Bereitstellung von Eingabedaten*).

Kompensierende Datenflüsse nehmen ebenfalls eine wichtige Rolle ein. Sie verhindern Störungen des Informationsflusses und damit die Ausbreitung reproduktionsbedingter Fehler in Richtung der Wurzel des Subtransaktionsgraphen (*Korrekter Informationsfluß*).

Schließlich muß noch betrachtet werden, welche Bedeutung die Reproduktion nichtdeterministischer Komponenten hat. Das Ausmaß reproduktionsbedingter Fehler ist geringer, wenn nichtdeterministische Komponenten auf ihrem Informationsweg nach beziehungsweise oberhalb kostenaufwendiger Subtransaktionen liegen. Operieren diese aufwendigen Komponenten nicht auf Daten, die von der Ausgabe der nichtdeterministischen Komponenten abhängig sind, so sind sie nicht betroffen, wenn die nichtdeterministischen Komponenten reproduziert werden, da sie sich dann nicht in dem Bereich befinden, in dem der reproduktionsbedingte Fehler aufgetreten ist. Abbildung 5.14 veranschaulicht diese Überlegung. Die kostenintensive Subtransaktion B liegt außerhalb des Bereiches, der bei der Reproduktion von N von reproduktionsbedingten Fehlern bedroht wird. Sie kann komplett wiederverwendet werden. Subtransaktion A dagegen ist von reproduktionsbedingten Fehlern betroffen und wird reproduziert.

5.7.5 Fallbeispiel

In diesem Abschnitt wird an einem Beispiel das vorgestellte FPT-Modell zur Fehlerbehandlung erläutert, wobei von der Konstellation in Abbildung 5.15 ausgegangen wird. Die linke Seite zeigt die Ausgangssituation, die von der Aktivitätskontrolle behoben werden soll, die rechte Seite stellt die Entscheidungen des Algorithmus zur Fehlerbehandlung aus Abschnitt 5.7.3.4 nach der Fehlererkennungsphase dar.

Der Transaktionszustand stellt sich folgendermaßen dar: Die Subtransaktionen H, I, J, K und L haben ihre Arbeit bereits erledigt und die Ergebnisse ihrer Berechnungen sind in dem vollständigen Datenfluß gespeichert, der die Eingabe von F beinhaltet. In F ist der Fehler aufgetreten. Die Komponenten B, C, D und E sowie F und G haben sich vor dem Fehler in nebenläufiger Ausführung befunden und über zwei Datenflüsse Informationen ausgetauscht. Dabei ist zu bemerken, daß der Datenfluß mit der Eingabe von F vollständig, der Datenfluß mit der

Ausgabe von F potentiell vollständig und der Datenfluß mit der Ausgabe von G unvollständig ist. Letzterer ist unvollständig, da er bereits Daten verworfen hat. Subtransaktion A schließlich nimmt noch nicht an der Datenverarbeitung der Transaktion teil, weil alle Datenflüsse, die ihre Eingabe bilden, noch leer sind.

Nun sind die Komponenten zu bestimmen, die reproduziert werden müssen, um ein erfolgreiches Beenden der Transaktion zu ermöglichen.

Zunächst erhält die Fehlerkontrolle die Nachricht, daß Subtransaktion F von einem Fehler betroffen ist. Dabei kann es sich um einen Komponentenfehler oder einen Datenfehler handeln. Weiterhin ist von Interesse, ob das System die in Abschnitt 5.7.3.3 Punkt 4 beschriebenen kompensierenden Datenflüsse besitzt und ob F eine nichtdeterministische Komponente ist.

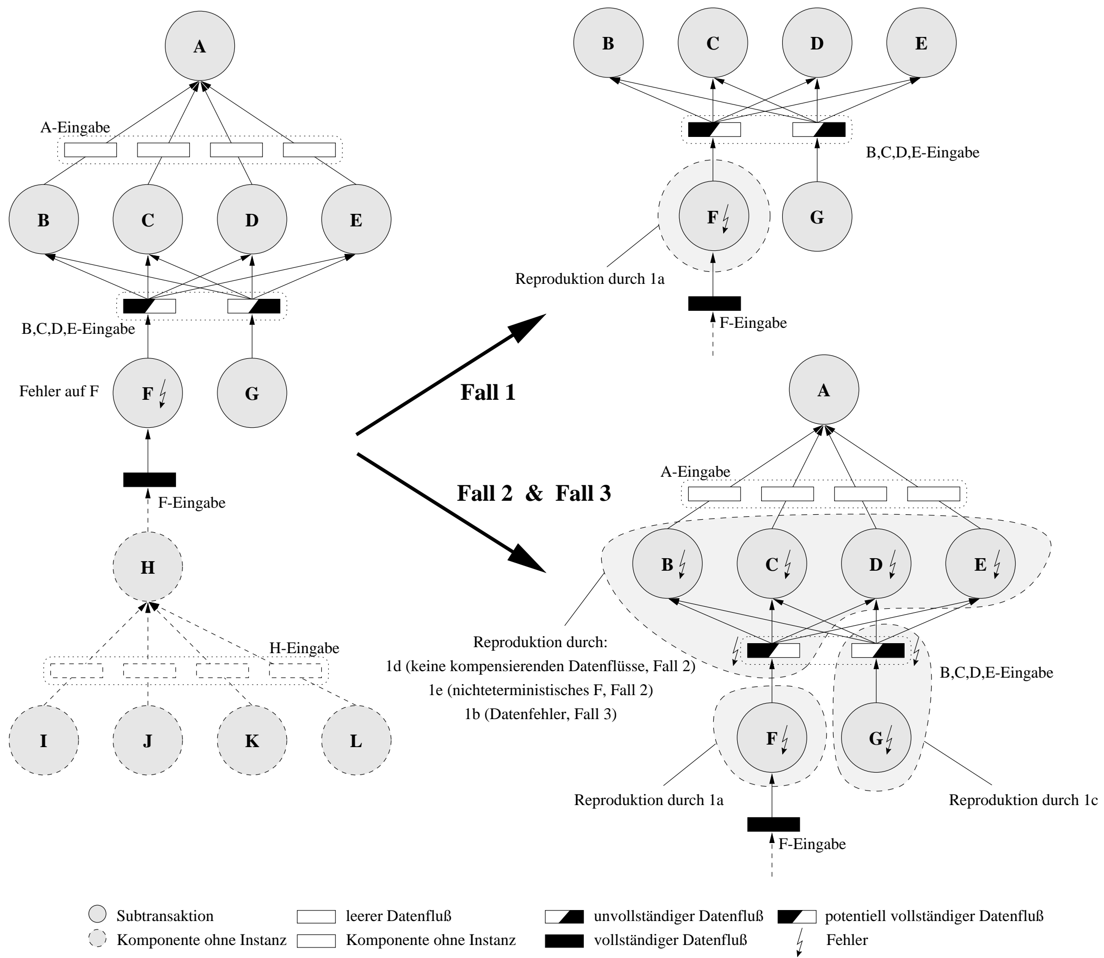

Abbildung 5.15: Beispiel der Fehlerbehandlung

Die sich aus diesen Vorgaben ergebenden möglichen Fälle sind im folgenden vorgestellt:

1. **In F ist ein Komponentenfehler aufgetreten, beispielsweise sei die Komponente komplett ausgefallen. F arbeitet deterministisch, das heißt F ist keine nichtdeterministische Komponente und das System besitzt kompensierende Datenflüsse.**

   Dieser Fall stellt sich als besonders günstig heraus. Der Algorithmus zur Fehlerbehandlung aus Abschnitt 5.7.3.4 bestimmt in der Erkennungsphase nur die Subtransaktion F selbst als zu reproduzierende Komponente. Dies geschieht im Schritt 1a des Algorithmus.
   Da ein Komponentenfehler vorliegt und kein Datenfehler, entfällt Schritt 1b.

   In den Schritten 1c bis 1e des Algorithmus muß dann das mögliche Auftreten von reproduktionsbedingten Fehlern untersucht werden. Die Eingabe von F ist in einem vollständigen Datenfluß gespeichert. Die Bereitstellung der Eingabedaten für F nach der Reproduktion ist somit gesichert und ein reproduktionsbedingter Eingabeverlust bei F kann nicht eintreten. Schritt 1c liefert daher keine weiteren zu reproduzierenden Komponenten.
   Der Datenfluß, der die Ausgabe von F aufnimmt, hat bereits Daten empfangen. Nach Reproduktion von F werden an diesen Datenfluß Daten gesendet, die dieser schon erhalten hat. Da es sich bei dem Datenfluß um einen kompensierenden Datenfluß handelt, kann es dabei zu keiner Störung des Informationsflusses kommen. Er wird trotz der Reproduktion von F eine korrekte Datensequenz an die Subtransaktionen B bis E senden. Schritt 1d des Algorithmus entfällt somit.
   Da F eine deterministische Komponente ist, werden alle Komponenten, die bereits von F abhängige Daten gelesen haben, nach der Reproduktion dieselben Datensequenzen erhalten können. Schritt 1e des Algorithmus wird, da keine nichtdeterministischen Komponenten von Fehlern betroffen sind, ebenfalls keine weiteren zu reproduzierenden Komponenten bestimmen.

   In dem geschilderten Fall werden also keine reproduktionsbedingten Fehler auftreten und allein die Reproduktion der vom Fehler direkt betroffenen Subtransaktion F reicht aus, um der gesamten Transaktion ein erfolgreiches Beenden zu ermöglichen.

2. **In F ist ein Komponentenfehler aufgetreten. Das System besitzt keine kompensierenden Datenflüsse oder F arbeitet nichtdeterministisch oder beides.**

   Der Algorithmus zur Fehlerbehandlung aus Abschnitt 5.7.3.4 wird, gemäß Schritt 1a, Subtransaktion F als die vom Fehler direkt betroffene Komponente zur Reproduktion bestimmen.

Sind keine kompensierenden Datenflüsse vorhanden, so werden zusätzlich zu F weitere Komponenten reproduziert werden. Der der Ausgabe von F zugeordnete Datenfluß hat bereits Daten empfangen. Bei der Reproduktion von F wird es zu Störungen des Informationsflusses auf dem Datenfluß kommen, da der Datenfluß das doppelte Senden gleicher Daten nach der Reproduktion von F nicht vor seinen Konsumenten, den Subtransaktionen B bis E, verbergen kann. Er ist kein kompensierender Datenfluß. Somit greift Schritt 1d des Algorithmus und erweitert die Liste der zu reproduzierenden Komponenten um den Datenfluß sowie die Subtransaktionen B bis E.

Dieselben Komponenten müssen reproduziert werden, wenn Subtransaktion F nichtdeterministisch arbeitet. Denn wird F reproduziert, so liefert sie über den Datenfluß andere Datensequenzen an die Subtransaktionen B bis E. Selbst wenn der Datenfluß das doppelte Senden kompensieren kann, so gelingt dies nur, wenn identische Sequenzen gesendet werden. Ein korrekter Informationsfluß zu den konsumierenden Subtransaktionen ist somit nicht gewährleistet. Gemäß Schritt 1e des Algorithmus müssen hier ebenfalls der Datenfluß sowie die Subtransaktionen B bis E reproduziert werden.

Allen vier Subtransaktionen B, C, D und E wird nach ihrer Reproduktion keine vollständige Eingabe vorliegen. Der Datenfluß, der die Ausgabe von G beinhaltet, hat bereits Daten verworfen und ist damit unvollständig. Daher müssen der Datenfluß und die Subtransaktion G in Schritt 1c ebenfalls reproduziert werden, um den Subtransaktionen B bis E nach deren Reproduktion korrekte Eingaben zur Verfügung zu stellen.

Die vier Datenflüsse, die Ausgabedaten von B, C, D und E weiterleiten, sind noch leer. Dort kann daher kein Fehler weitergegeben werden. Die Fehlererkennung ist damit abgeschlossen.

Nach der Erkennungsphase wird die Fehlerkontrolle in die Beseitigungsphase eintreten. Die Subtransaktionen B bis G und die beiden an ihrer Kommunikation beteiligten Datenflüsse werden beseitigt werden. Schließlich werden die acht Komponenten in der Reproduktionsphase wiederaufgesetzt und die Transaktion kann ihre Arbeit fortsetzen.

3. **In Subtransaktion F ist ein Datenfehler aufgetreten.**

   Der Algorithmus zur Fehlerbehandlung wird wiederum Subtransaktion F als die vom Fehler direkt betroffene Komponente zur Reproduktion bestimmen.

   Die Fehlerkontrolle ermittelt im Schritt 1b alle Komponenten, die von der Ausgabe von F abhängige Daten verarbeitet haben. Auch in diesem Fall sind dies die Subtransaktionen B, C, D und E.

   Wie im Fall 2 werden auch hier Subtransaktion G und ihr Datenfluß im Schritt 1c zur Reproduktion bestimmt, um den Subtransaktionen B bis E

nach deren Reproduktion korrekte Eingaben zur Verfügung zu stellen.

Auch der Rest der Fehlerbehandlung wird sich wie im Fall 2 beschrieben vollziehen. Beide Fälle bestimmen in der Fehlererkennung die Subtransaktionen B bis G und ihre Datenflüsse, beseitigen und reproduzieren diese.

Das Beispiel zeigt eine für das Konzept der Aktivitätskontrolle vorteilhafte Situation. Wegen des vollständigen Datenflusses, der Eingabe von F ist, ist die Arbeit der Subtransaktionen H, I, J, K und L gesichert. Dies hat den Vorteil, daß trotz des in F aufgetretenen Fehlers ein großer Teil der Transaktion nicht wiederholt werden muß.

Im Fall 1, in dem kompensierende Datenflüsse existieren, beschränkt sich die Menge der zu reproduzierenden Komponenten sogar einzig und allein auf die Subtransaktion F, in der der Fehler aufgetreten ist.

Nur für den Fall, daß die Eingabe von F unvollständig ist und keine kompensierenden Datenflüsse existieren, müssen in dem gezeigten Beispiel alle Komponenten reproduziert werden, was einem Wiederaufsetzen der gesamten Transaktion entspricht. In allen anderen Fällen kann beim Wiederaufsetzen der Transaktion mehr oder weniger von der bereits verrichteten Arbeit von Komponenten profitiert werden.

Eine Variante des vorherigen Beispiels ist in Abbildung 5.16 dargestellt. Dort wird der Fall betrachtet, daß der Ausgabedatenfluß von G nicht unvollständig, sondern potentiell vollständig ist. In den geschilderten Fällen 2 und 3 müssen dieser Datenfluß und die Subtransaktion G gemäß Schritt 1c reproduziert werden. Hier ist dies nicht der Fall, da der potentiell vollständige Datenfluß nach der Reproduktion der Subtransaktionen B bis E von diesen wieder komplett von Anfang an gelesen werden kann. Eine Reproduktion des Datenflusses und der Subtransaktion G ist daher nicht notwendig. Diese Variante ist in Abbildung 5.16 dargestellt.

Dieses Fallbeispiel stellte noch einmal die Bedeutung des Vorhandenseins vollständiger, potentiell vollständiger und kompensierender Datenflüsse heraus. Durch sie wird die Menge der zu reproduzierenden Komponenten stark begrenzt. Vollständige und potentiell vollständige Datenflüsse verhindern die Ausbreitung reproduktionsbedingter Fehler nach unten (von F auf die Komponenten H bis L und in Beispiel 2, Fall 2 und 3 von B bis E auf G). Kompensierende Datenflüsse können die Ausbreitung reproduktionsbedingter Fehler nach oben im Graphen in Richtung Wurzel verhindern (von F auf die Komponenten B bis E in Fall 1). Im Idealfall, in dem jede Fehlerausbreitung verhindert wird, ist damit nur die direkt vom Fehler betroffene Komponente F zu reproduzieren.

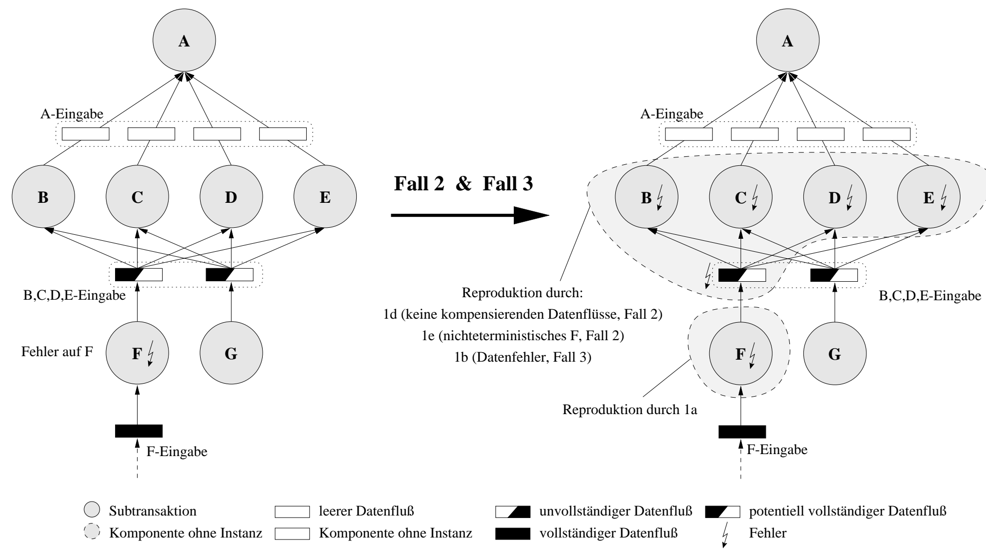

Abbildung 5.16: Beispiel 2 – potentiell vollständige Ausgabe von G

5.7.6 Zusammenfassung

In diesem Kapitel wurde ein Modell zur Fehlerbehandlung gesucht, das speziell auf die Eigenschaften des Ausführungsmodells eines parallelen Datenbanksystems abgestimmt ist. Dazu wurden zuerst mögliche Fehler beschrieben und bereits existierende Transaktionskonzepte vorgestellt und ihre Tauglichkeit untersucht, eine solche Fehlerbehandlung durchzuführen. Nach einer genauen Spezifikation der Eigenschaften, die ein solches Modell zur Fehlerbehandlung haben soll, konnte festgestellt werden, daß die existierenden Transaktionskonzepte die gestellten Anforderungen nicht erfüllen können.

Im folgenden wurde das FPT-Modell, basierend auf der Unterteilung von Transaktionen in Subtransaktion entwickelt und beschrieben, das die gestellten Anforderungen erfüllt. Es paßt das Granulat der Transaktionsverarbeitung an die Struktur des parallelen Ausführungsmodells an. Transaktionsfehler in einzelnen Komponenten verursachen nicht mehr zwangsläufig den Abbruch der kompletten Transaktion. Vielmehr bietet das FPT-Modell die Möglichkeit, große, schon berechnete Teile der Transaktion nach einem Neustart wiederzuverwenden und somit deutlich an Effizienz zu gewinnen. Zusätzlich zu dem FPT-Modell wurde ein Algorithmus entworfen, der die vom Modell vorgegebenen Spezifikationen erfüllt und eine Anleitung dazu gibt, wie die korrekte Fehlerbehandlung konkret durchzuführen ist.

Die Effizienz der von dem Algorithmus durchgeführten Korrektur ist davon abhängig, auf welchem System, unter Berücksichtigung der Struktur und der Eigenschaften der Transaktion und der Transaktionskomponenten, die Fehlerbe-

handlung operieren soll. Sie wird vom Umfang der Bedrohung durch reproduktionsbedingte Fehler negativ beeinflußt. Je mehr die Transaktionskomponenten von einander isoliert sind, beziehungsweise je weniger Abhängigkeiten zwischen den operierenden Komponenten bestehen, desto kleiner ist die Ausbreitung dieser Fehler.

Die wichtigste Rolle nehmen potentiell vollständige, vollständige und kompensierende Datenflüsse ein. Sie sind die Hauptmechanismen um die Fehlerausbreitung einzuschränken. Im Idealfall ist nur die den Fehler verursachende Komponente selbst zu reproduzieren, alle übrigen Komponenten können wiederverwendet werden.

Bei dem vorgestellten FPT-Modell wird keine Recovery auf permanenten Daten vollzogen, sondern nur die eventuell, d. h. nicht zwingend notwendige Wiederverwendbarkeit von Zwischenergebnissen, also abgeleiteten Daten ausgenutzt. Es müssen insbesondere keine speziellen Vorkehrungen getroffen werden, um die Wiederherstellung eines konsistenten Datenbankzustandes zu ermöglichen, da im Zweifelsfall immer das vollzogen werden kann, was bei flachen Transaktion auch notwendig wäre, nämlich die gesamte Transaktion abzubrechen. Somit entfallen aufwendige Mechanismen, die den normalen, d. h. fehlerfreien Fall belasten würden.

An einem Fallbeispiel, bei dem große Teile der Transaktion wiederverwendet werden konnten, wurde die Durchführung und Effizienz der vorgestellten Methode gezeigt.

Kapitel 6

Abbildung des Modells zur Fehlerbehandlung auf MIDAS

Die Aktivitätskontrolle in MIDAS besaß bisher keine Komponente zur Fehlerbehandlung. Traten in MIDAS Fehler in einer Transaktion auf, so mußte die gesamte Transaktion abgebrochen werden.

Dieses Kapitel stellt nun vor, wie eine solche Komponente in MIDAS realisiert werden kann. Dazu wird gezeigt, wie das im vorherigen Kapitel 5 entwickelte FPT-Modell, dem Modell zur Fehlerbehandlung in parallelen Transaktionen, auf MIDAS abgebildet werden kann und welche Ergänzungen gemacht werden müssen, um es zu integrieren.

Abschnitt 6.1 stellt die Abbildung des FPT-Modells auf MIDAS vor. In Abschnitt 6.2 werden die Funktionsweisen des neuen FPT-Modells in MIDAS beschrieben. Schließlich folgen ab Abschnitt 6.3 Analysen zum Einsatz des Modells in MIDAS.

6.1 Abbildung der Transaktionsstruktur auf MIDAS

Der Entwurf des in Abschnitt 5.7 vorgestellten FPT-Modells erfolgte gemäß den in Abschnitt 5.4 aufgeführten Anforderungen des Ausführungsmodells von MIDAS. Die Abbildung auf das Ausführungsmodell gestaltet sich damit einfach. Die Subtransaktionen des FPT-Modells werden auf Teilpläne einer Anfrage und somit fast immer auf komplette Ausführungseinheiten abgebildet. Datenflüsse werden auf Kommunikationskanäle abgebildet.

Abbildung 6.1 zeigt, wie ein paralleler Ausführungsplan auf einen Graph von Subtransaktionen abgebildet wird und diese dann in eine Struktur von Ausführungs-

einheiten eingebettet werden.

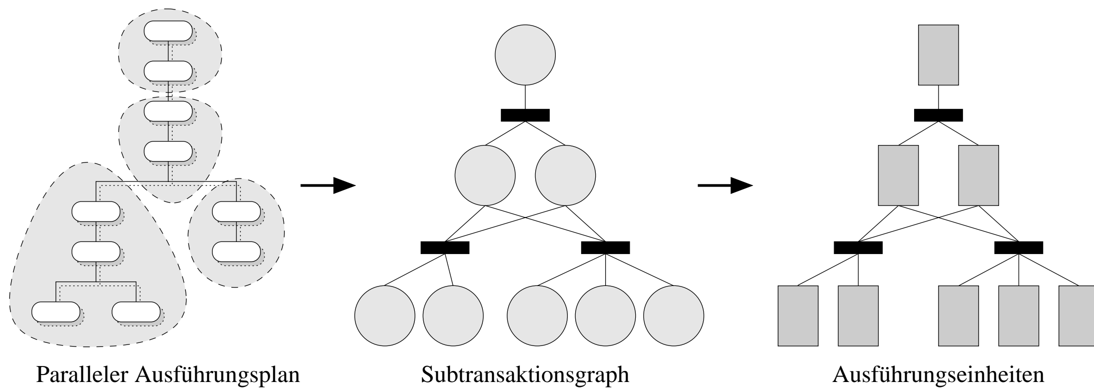

Abbildung 6.1: Ausführungseinheiten und Subtransaktionen

Wie diese Abbildung des FPT-Modells auf MIDAS im Detail stattfindet, wird in den folgenden Abschnitten gezeigt.

6.1.1 Subtransaktionen

In MIDAS findet die Verteilung einer Anfrage statt, indem Operator-Teilpläne auf verteilten Ausführungseinheiten zur Ausführung gebracht werden. Die Kommunikationsoperatoren (siehe Abschnitt 2.4.9) Send und Recv in diesen Teilplänen leiten Tupel zwischen den Ausführungseinheiten bis zu der Ausführungseinheit weiter, die den Teilplan mit der Wurzel des Operatorbaumes auswertet. Diese gibt dann die Ergebnisse an den Benutzer der Transaktion weiter.

Diese Art der Verarbeitung ist der im vorgestellten FPT-Modell sehr ähnlich. Zwischen den Operatoren und somit auch zwischen den Ausführungseinheiten findet ein Datenaustausch immer in einer Richtung statt. Eine nebenläufige Durchführung der Teilpläne auf den Ausführungseinheiten wird ebenfalls unterstützt.

Eine Subtransaktion wird daher auf die Durchführung eines von Send- und Recv-Operatoren eingerahmten Teilplans abgebildet.

In MIDAS ist es weiterhin möglich, mehrere benachbarte Teilbäume desselben Astes auf einer einzigen Ausführungseinheit ausführen zu lassen [Brandmayer 97]. Diese Option wurde aus Leistungsgründen realisiert. Dabei können mehrere Teilpläne, getrennt durch ein Paar von Kommunikationsoperatoren, gleichzeitig auf derselben Ausführungseinheit aktiv sein. Die Teilpläne auf derselben

Ausführungseinheit behalten dabei ihre Send- und Recv-Operatoren und tauschen auch weiterhin Daten mit anderen Ausführungseinheiten aus. Es ist also nicht immer der komplette Teilbaum, der auf einer Ausführungseinheit ausgewertet wird, in eine einzige Subtransaktion eingeschlossen. Es kann sogar jeder beliebige Übergang zwischen Operatoren als Teilplangrenze und somit Subtransaktionsgrenze verwendet werden. Abbildung 6.2 zeigt zwei Möglichkeiten, wie Subtransaktionen in Ausführungseinheiten eingebettet werden können.

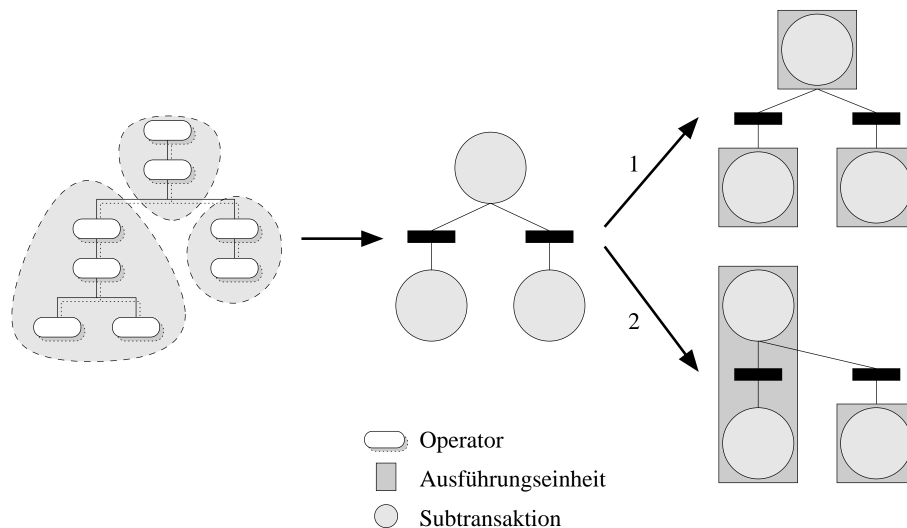

Abbildung 6.2: Abbildung von Subtransaktionen auf Ausführungseinheiten

Es lassen sich auch Eigenschaften von Subtransaktionen von deren Operatorbäumen ableiten. Operatoren können blockierend sein, d. h. sie besitzen die Eigenschaft, die Verarbeitung ihre gesamten Eingaberelation abgeschlossen zu haben, bevor er Ergebnistupel weitergibt. Entsprechend kann ein nichtblockierender Operator schon Tupel ausgeben, wenn er nur einen bestimmten Teil der Eingaberelation bearbeitet hat. Blockierende Operatoren haben zur Folge, daß Operatoren, die ober- und unterhalb eines solchen Operators liegen, nicht nebenläufig operieren können.

Diese Eigenschaft überträgt sich, je nach Beschaffenheit des Teilplans, der innerhalb einer Ausführungseinheit ausgeführt wird, auch auf die dort befindlichen Subtransaktionen. Liegt ein blockierender Operator auf einem Ast des Ausführungsplans, der von allen Datenwegen geschnitten wird, so ist die Subtransaktion ebenfalls blockierend und gibt ihre Ergebnisse erst dann weiter, wenn sie ihre komplette Eingabe verarbeitet hat. Ansonsten schränken blockierende Operatoren zumindest teilweise die sofortige Weitergabe von Zwischenergebnissen

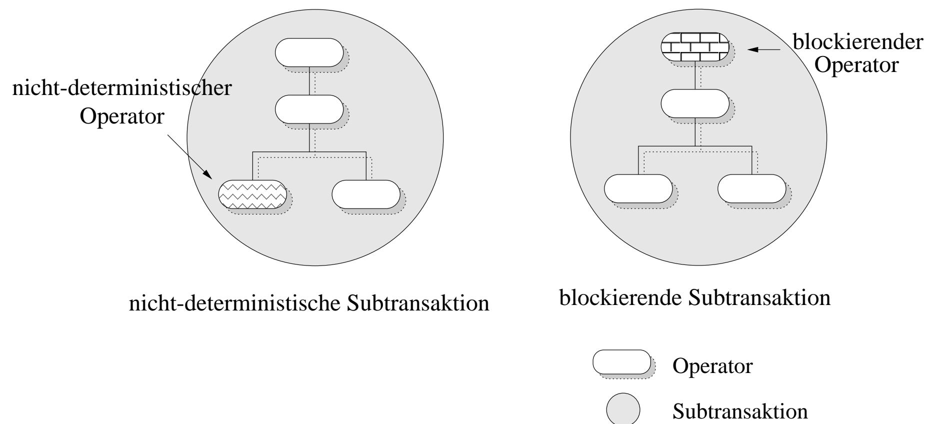

Abbildung 6.3: Nichtdeterministische und blockierende Subtransaktionen

ein. Blockierende Operatoren begünstigen somit die Bildung von Zwischenergebnissen in Form vollständiger Datenflüsse, worauf in Abschnitt 6.3 noch genauer eingegangen wird.

Weiter kann sich eine Subtransaktion nichtdeterministisch verhalten, wenn sich in ihrem Teilplan mindestens ein nichtdeterministischer Operator befindet. Diese Eigenschaft ist bei der Reproduktion von Transaktionskomponenten von Interesse und wird ebenfalls in Abschnitt 6.3 diskutiert. Abbildung 6.3 veranschaulicht beide zuletzt beschriebenen Eigenschaften.

6.1.2 Datenflüsse

Die Kommunikation zwischen den Ausführungseinheiten bzw. Interpretern findet über Kommunikationskanäle statt. Ein Kanal, der eine m:n-Kommunikation realisiert, ist dabei durch $m * n$ Kommunikationssegmente implementiert (siehe Abschnitt 2.4.9). Die Segmente werden zur Datenübertragung benachbarter Teilpläne verwendet. Sie fügen sich in die Struktur der Teilpläne ein und entsprechen den Forderungen des FPT-Modells an die Datenflüsse. Datenflüsse werden daher direkt auf einzelne Kommunikationssegmente abgebildet.

Werden mehrere Teilpläne auf einem Interpreter zusammengefaßt, erzeugt MIDAS keine Kommunikationssegmente. In diesem Fall werden die Tupel von einem Send-Operator an den Recv-Operator direkt durch die sogenannte lokale Partition übergeben (siehe Abschnitt 3.4). Dieser Spezialfall ist aber nur bei der Implementierung zur Steigerung der Effizienz des Modells von Bedeutung. Die Eigenschaften der Datenflüsse werden dadurch nicht verletzt, da auch die lokale Partition in MIDAS intern als nicht instantiiertes Kommunikationssegment angesehen werden kann. Die Segmente zwischen Kommunikationsoperatorpaaren

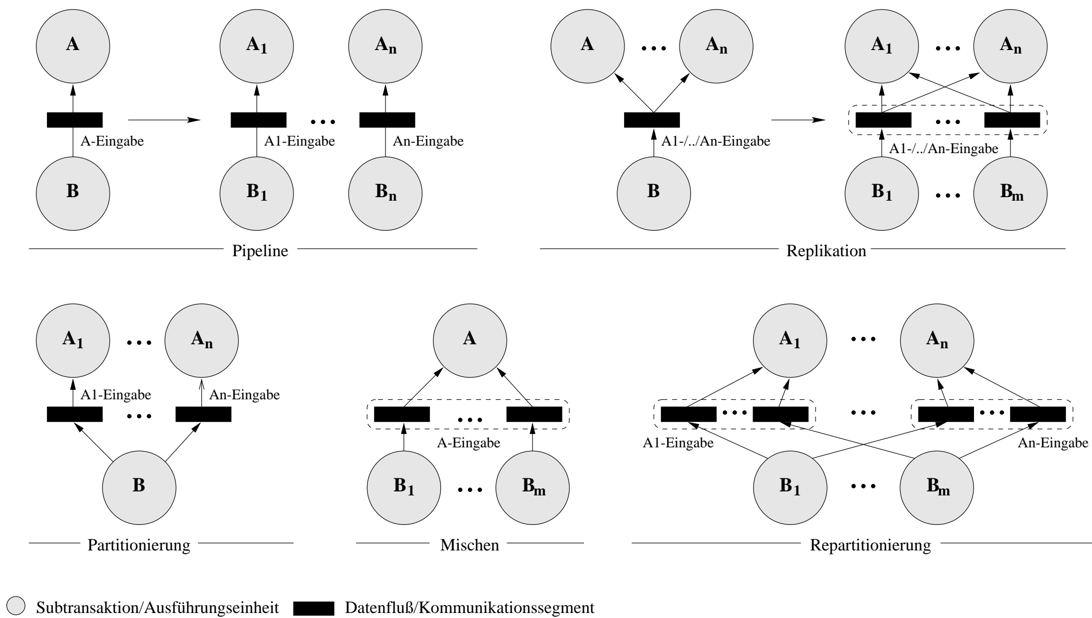

Abbildung 6.4: Kommunikationsformen

verhalten sich genauso wie die Datenflüsse zwischen den Subtransaktionen im FPT-Modell.

Ein Kommunikationssegment wird mit den Optionen *temporär* oder *persistent* angelegt. Bei temporären Kommunikationssegmenten werden Daten sofort nachdem sie vom Konsumenten gelesen werden verworfen (Option ReadOnce, siehe Abschnitt 2.4.9). Sie sind also genau dann im Rahmen der Fehlerbehandlung wiederverwendbar, d. h. vollständig oder potentiell vollständig, wenn die erste Seite noch nicht nach dem Lesen gelöscht wurde. Persistente Kommunikationssegmente sind sicherlich vollständig oder potentiell vollständig, solange sie existieren.

Die schon vorgestellten Kommunikationsformen zwischen Ausführungseinheiten sind Pipelining, Mischen, Partitionierung, Repartitionierung und Replikation. Es lassen sich alle möglichen Konstellationen zwischen Ausführungseinheiten und Kommunikationssegmenten auf eines dieser fünf Muster zurückführen. Alle diese Kommunikationsformen können von dem vorgeschlagenen FPT-Modell nachgebildet werden. Wie die Datenflüsse erlauben auch die Kommunikationssegmente nur einen Produzenten. Einschränkungen in den Kommunikationsformen gibt es deshalb nicht, weil Kommunikationskanäle mit m:n-Kommunikation, wie oben beschrieben, durch mehrere Segmente implementiert werden können. Abbildung 6.4 zeigt noch einmal die fünf Kommunikationsformen.

6.1.3 Transaktionsstruktur einer Beispielanfrage

Die Abbildung der Transaktionsstruktur auf eine konkrete Anfrage in MIDAS wird am Beispiel der Struktur der TPC–D Anfrage 3 veranschaulicht. Abbildung 6.5 zeigt die parallelisierte Anfrage. Die Subtransaktionen umfassen die einzelnen Teilpläne, die jeweils auf einer Ausführungseinheit interpretiert werden. Jedes Kommunikationssegment entspricht einem Datenfluß.

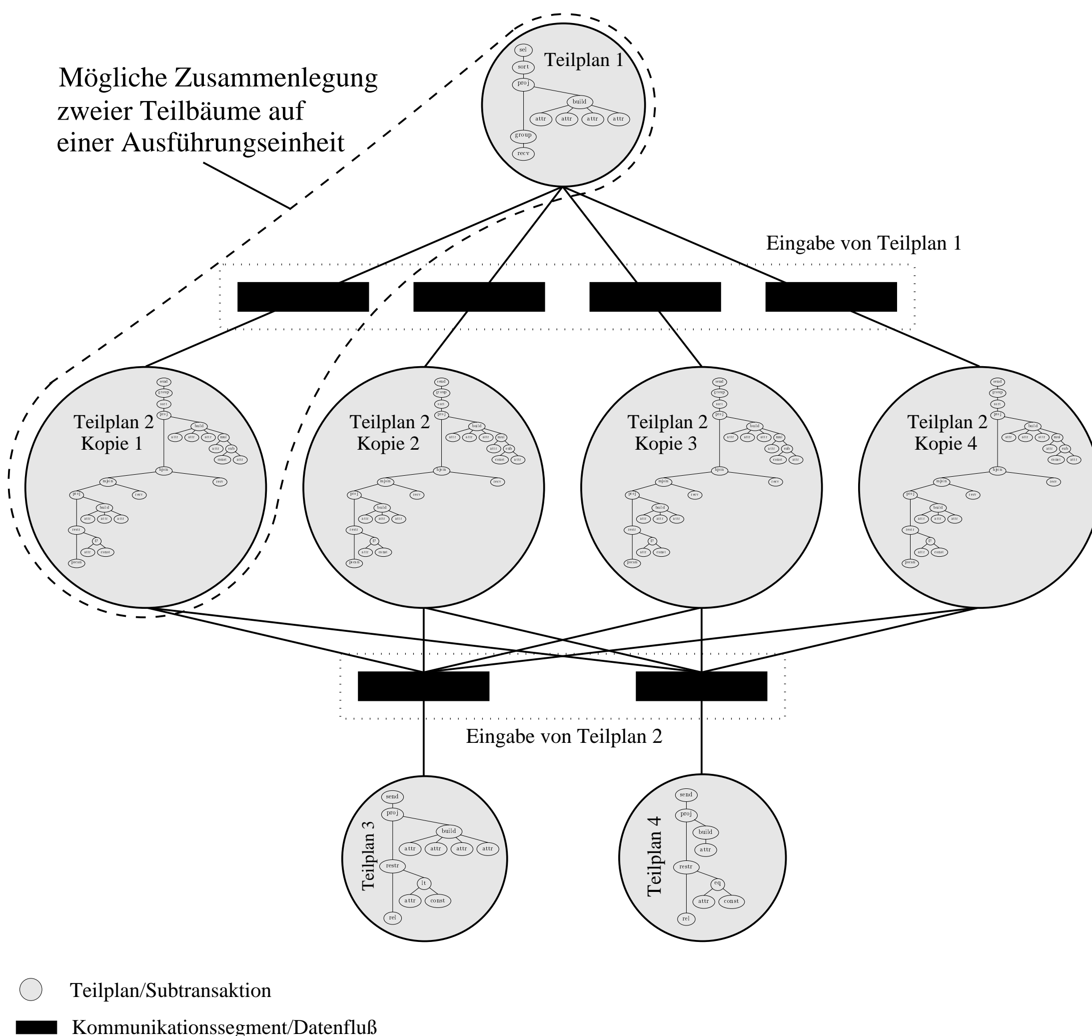

Abbildung 6.5: Transaktionsstruktur von TPC–D Anfrage 3

Wie schon erwähnt, können mehrere Subtransaktionen auf einer Ausführungseinheit ausgewertet werden. Der Parallelisierer kann vorschlagen, Teilplan 1 und

eine Kopie des zweiten Teilplans auf dieselbe Ausführungseinheit zu plazieren. Diese wird durch das gestrichelte Zusammenfassen beider Teilpläne ausgedrückt. In diesem Fall wird das vorgesehene Kommunikationssegment nicht realisiert und an seine Stelle tritt eine lokale Partition.

6.2 Fehlerbehandlung in MIDAS

Dieser Abschnitt beschäftigt sich mit den speziellen Eigenschaften von MIDAS, die bei der Fehlerbehandlung berücksichtigt werden müssen. Dabei wird insbesondere die Fehlererkennung und das Vorgehen bei der Vermeidung reproduktionsbedingter Fehler betrachtet. Diese Eigenschaften müssen bei einer Umsetzung der Erkennungsphase des in Abschnitt 5.7.3.4 vorgestellten Algorithmus zur Fehlerbehandlung berücksichtigt werden.

6.2.1 Fehlererkennung

6.2.1.1 Transaktionsfehler

Bei der Erkennung der Transaktionsfehler muß vor allem eine Klassifikation nach der Art des Fehlers stattfinden. Es gilt zu bestimmen, wann Datenfehler und wann Komponentenfehler in MIDAS auftreten.

- Datenfehler verhalten sich in MIDAS wie im FPT-Modell. Der Fehler kann auf Daten in Operatoren, Kommunikationssegmenten oder permanenten Segmenten während der Transaktion erzeugt oder durch den Entzug einer Sperre auf ein Datenobjekt verursacht werden. Die betroffenen Daten sind dadurch fehlerhaft und somit ungültig.

  Verklemmungen von Transaktionen, die wegen eines Sperrkonflikts existieren, werden üblicherweise dadurch gelöst, daß einer Transaktion die Sperre, die den Konflikt verursacht, entzogen wird. Normalerweise muß die betroffene Transaktion daraufhin komplett abgebrochen werden. Das FPT-Modell sieht vor, diese Transaktion auf den Stand vor Erhalt der Sperre zurückzusetzen. Dies geschieht dadurch, daß alle Subtransaktionen, die Datenobjekte gelesen haben, und die durch die Sperre geschützt sind, mit einem Datenfehler belegt werden. Daraufhin greifen die Mechanismen des FPT-Modells und alle Daten, die aus den Datenobjekten, für die die Transaktion keine Sperre mehr hält, errechnet wurden, werden für ungültig erklärt. Die zugehörigen Subtransaktionen werden abgebrochen. Der Sperrkonflikt ist damit aufgelöst, ohne die Transaktion komplett abbrechen zu müssen.

- Komponentenfehler in MIDAS sind entweder Subtransaktionsfehler oder Fehler in Kommunikationssegmenten.

  Subtransaktionsfehler sind alle Fehler, die eine Ausführungseinheit, also die Interpreterprozesse betreffen. Dies umfaßt auch Prozeßfehler, die während einer Ausführung der Methoden der Segmentschicht auftreten, also unterhalb der eigentlichen Interpreterebene. Auch ein Komplettausfall der Interpreter zählt dazu.

  Fehler in Kommunikationssegmenten sind Verluste beim Versenden der Daten des Segments über das Netzwerk, Verluste im Speicher, oder Fehler in der Segmentverwaltung.

  Zyklische Verklemmungen zwischen Ausführungseinheiten und Kommunikationssegmenten, wie sie in Abschnitt 2.4.9 beschrieben sind, fallen ebenfalls unter die Komponentenfehler. Bei solchen Verklemmungen innerhalb einer Anfrage wird ein möglichst kleiner Teil der an der Verklemmung beteiligten Komponenten als fehlerhaft klassifiziert, der ausreicht, um die Verklemmung aufzulösen. Die so bestimmten Komponenten werden mit zur Reproduktion vorgesehen.

Wird ein Datenfehler erkannt, können gemäß den Datenabhängigkeiten (siehe Abschnitt 5.7.3.3) nur die Komponenten sicher als nicht betroffen gelten, die keine Ausgabetupel der fehlerhaften Komponente gelesen haben. Die Erkennung, mit welchem beziehungsweise mit dem wievielten Tupel der Fehler weitergegeben wurde, ist dabei im allgemeinen nicht möglich und wird deshalb nicht versucht.

6.2.1.2 Reproduktionsbedingte Fehler

Es ist zu betrachten, welche der in Abschnitt 5.7.3.3 beschriebenen, reproduktionsbedingten Fehler in MIDAS auftreten können. Situationen, in denen sie auftreten können, müssen bei der Fehlererkennung identifiziert werden. Durch die Reproduktion von ausreichend vielen Komponenten wird das Auftreten dieser Fehler im Anschluß vermieden.

- **Bereitstellen von Eingabedaten – Eingabeverlust:**

  In MIDAS kann der Fall auftreten, daß Eingabedaten im Fehlerfall nicht bereitgestellt werden können (Eingabeverlust). Ist bei Kommunikationssegmenten die Option ReadOnce eingestellt, löscht der Schreiber die Seite oder Subseite, sobald sie vom Leser empfangen wurde. Wird der Leser reproduziert und hatte er zuvor bereits eine Seite oder Subseite erhalten, hat der Schreiber diese schon verworfen. Damit steht dem reproduzierten Leser nicht mehr die komplette Eingabe zur Verfügung. Ist die Option ReadOnce

dagegen nicht gesetzt, sind die Daten weiterhin in den Kommunikationssegmenten verfügbar. Die Seiten des Segments können allerdings im Rahmen der Cacheverdrängungsstrategien inzwischen auf externe Speichermedien ausgelagert worden sein. Wird der Leser also reproduziert, müssen die entsprechenden Seiten gegebenenfalls neu in den Cache eingelagert werden.

- **Korrekter Informationsfluß – Störung des Informationsflusses:**

  Eine Störung des Informationsflusses konnte in MIDAS ebenfalls auftreten. Nochmaliges Senden von Daten durch einen reproduzierten Produzenten kann von den Kommunikationssegmenten nicht erkannt werden. Sie würden die Daten einfach weiterleiten, wodurch den Konsumenten eine inkorrekte Datensequenz erreicht. Eine kleine Änderung in der Implementierung der Kommunikationssegmente kann diese Art von Fehler komplett ausschließen. Durch die Änderung werden kompensierende Datenflüsse implementiert. Wie dies realisiert wird, ist in Abschnitt 6.3.5.1 beschrieben.

- **Reproduktion nichtdeterministischer Komponenten:**

  Das Auftreten dieses Fehlers droht, wenn zu reproduzierende Subtransaktionen nichtdeterministische Operatoren beinhalten. In MIDAS gibt es drei derartige Operatoren, den DtCur- (DaTe CURrent), den asynchronen Recv- und den Func-Operator. Die Funktion des DtCur-Operators besteht darin, das aktuelle Datum auszugeben. Schreitet das Datum voran, bevor die Reproduktion erfolgt ist, liefert der Operator ein anderes Datum. In der Folge werden die vorher weitergeleiteten Daten inkonsistent.

  Empfängt ein Recv-Operator asynchron von mehreren Produzenten, verhält er sich ebenfalls nichtdeterministisch. Die Empfangsreihenfolge der Tupel ist dann nicht vorhersehbar. Verzögerungen bei den Produzenten, die beispielsweise auf anderen Maschinen liegen können, oder Netzverzögerungen verursachen ein "zufälliges" Eintreffen der Daten bei diesem Operator. Dadurch erzeugt er als Ausgabe eine nicht reproduzierbare Sequenz der Daten. Ob sich damit auch die zugehörigen Subtransaktionen nichtdeterministisch verhalten, hängt davon ab, ob die Daten nach dem Recv-Operator im ungeordneten Zustand weitergegeben werden. Findet vor der Weitergabe an andere Subtransaktionen eine totale Sortierung statt, also eine Sortierung auf allen Attributen, verhält sich die Subtransaktionen trotzdem deterministisch. Entsprechend entsteht auf den nachfolgenden Subtransaktionen kein Fehler.

  Der Func-Operator schließlich ist benutzerdefiniert. Je nach Implementierung müßte sein Verhalten speziell geprüft werden. Da das System im allgemeinen nicht feststellen kann, ob sich der Operator deterministisch verhält, muß jeder Func-Operator als nichtdeterministisch betrachtet werden. Eine spezielle in MIDAS implementierte Variante des Func-Operators stellt der

UDTO-Operator dar. Er implementiert benutzerdefinierte Tabellenoperatoren in MIDAS [Jaedicke 99] und ist, wie der FUNC-Operator, als nichtdeterministisch anzusehen.

Das Auftreten solcher Fehler kann teilweise verhindert oder zumindest eingeschränkt werden. Möglichkeiten, wie dies geschehen kann, sind in Abschnitt 6.3.5 aufgezeigt.

6.2.2 Fehlerbeseitigung

Die Fehlerbeseitigung entspricht der Beseitigungs- und Reproduktionsphase des Algorithmus, in der die fehlerhaften und von reproduktionsbedingten Fehlern bedrohten Komponenten abgebrochen und neu gestartet werden.

Bei der Reproduktion der betroffenen Komponenten müssen diese von den restlichen isoliert und abgebrochen werden. Hierfür sorgt die Komponente der Fehlerkontrolle und startet die abgebrochenen Subtransaktionen anschließend neu. Abgebrochene Kommunikationssegmente werden von der Cacheverwaltung entfernt und bei der Reproduktion durch neue Instanzen ersetzt.

6.3 Bewertung

Die Bewertung aus Abschnitt 5.7.4 der Aktivitätskontrolle kann nun in MIDAS präzisiert werden. Dazu wird betrachtet, wie sich Fehler in MIDAS ausbreiten, wie für Eingabeverfügbarkeit nach der Reproduktion gesorgt werden kann und welche weiteren Maßnahmen zur Leistungssteigerung getroffen werden können.

6.3.1 Ausbreitung von Datenfehlern

Für die Ausbreitung von Datenfehlern gelten dieselben Überlegungen wie im FPT-Modell. Tritt in einer Subtransaktion ein Datenfehler auf, so wird dieser an alle von ihm lesenden Komponenten weitergegeben, also an alle im Subtransaktionsgraph über ihr liegenden Komponenten, die schon Daten verarbeitet haben. Die Ausbreitung unterscheidet sich dabei leicht bei den verschiedenen Kommunikationsformen:

- Pipelining und Mischen:

  Der Fehler breitet sich, wie in Abbildung 6.6 gezeigt, auf genau einem Ast nach oben aus, da bei beiden Formen nur ein Ast existiert.

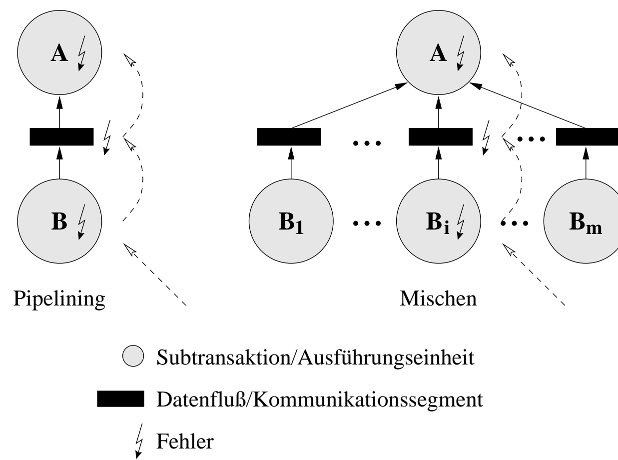

Abbildung 6.6: Fehlerausbreitung bei Pipelining und Mischen

- Replikation:
  Der Fehler breitet sich, wie in Abbildung 6.7 gezeigt, auf alle Äste nach oben aus. Da dort nur ein Kommunikationssegment beteiligt ist, wird der Fehler gleichzeitig an alle oberhalb liegenden Subtransaktionen weitergegeben.

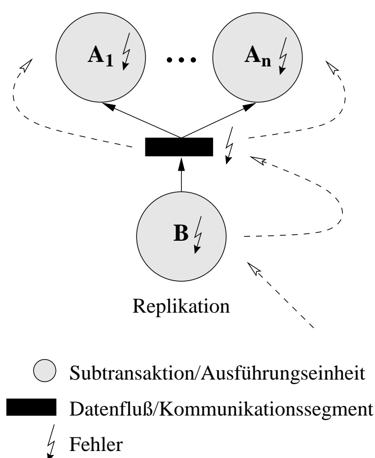

Abbildung 6.7: Fehlerausbreitung bei Replikation

- Partitionierung und Repartitionierung:
  Auch hier kann sich der Fehler auf alle Äste nach oben ausbreiten. Allerdings müssen davon nicht sofort alle Partitionen betroffen sein. Die Verteilung findet entweder nach dem Round-Robin-Verfahren oder, bei Bereichs-

und Hash-Partitionierung, in Abhängigkeit vom Tupelwert in verschiedene Kommunikationssegmente statt. Dabei kann der Fall eintreten, daß mehrere Partitionen zu Anfang längere Zeit noch nicht gefüllt werden. Ist beim Auftreten des Fehlers eine Partition, und damit ein Kommunikationssegment, noch leer, so breitet sich der Fehler, wie in Abbildung 6.8 gezeigt, dorthin nicht aus. Damit sind nicht alle oberhalb liegenden Komponenten betroffen.

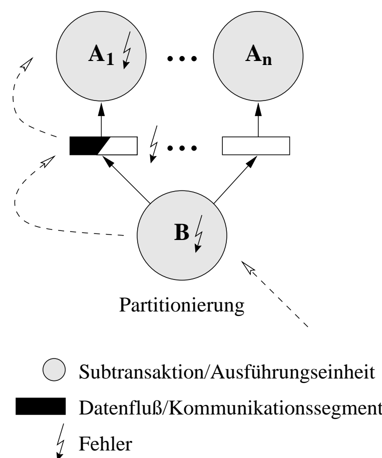

Abbildung 6.8: Fehlerausbreitung bei Partitionierung

6.3.2 Ausbreitung reproduktionsbedingter Fehler

Wie bei der Ausbreitung von Datenfehlern lassen sich aus der Struktur der Kommunikationsformen Aussagen über die Gefährdung durch reproduktionsbedingte Fehler ableiten. Zu den Kommunikationsformen lassen sich bestimmte Fehlerszenarien feststellen, die exemplarisch für eine Bewertung oder für Verbesserungen herangezogen werden können.

Bei einer Form der reproduktionsbedingten Fehler, der Reproduktion nichtdeterministischer Komponenten, gelten dieselben Überlegungen wie bei der im vorherigen Abschnitt 6.3.1 besprochenen Ausbreitung von Datenfehlern. Im folgenden werden daher nur die beiden anderen Arten reproduktionsbedingter Fehler, Eingabeverlust und Störung des Informationsflusses, betrachtet.

Beide Fehler treten in unterschiedlichen Richtungen im Subtransaktionsgraph auf. Störungen des Informationsflusses können nur auf nachfolgenden, beziehungsweise oberhalb angeordneten Komponenten entstehen, es sei denn, kompensierende

Datenflüsse sind implementiert (siehe Abschnitt 5.7.3.3 Punkt 4.). In diesem Fall können sie nicht auftreten. Dagegen führen Eingabeverluste zunächst zu einem Fehler auf der Komponente selbst. Da die Fehlerkontrolle eine verlorene Eingabe vervollständigen muß, kann sich der Fehler zu einer der vorangegangenen beziehungsweise unterhalb liegenden Komponenten ausbreiten. Dies geschieht, wenn die unterhalb liegende Komponente kein vollständiger oder potentiell vollständiger Datenfluß ist, aus dem die komplette Eingabe erneut gelesen werden kann.

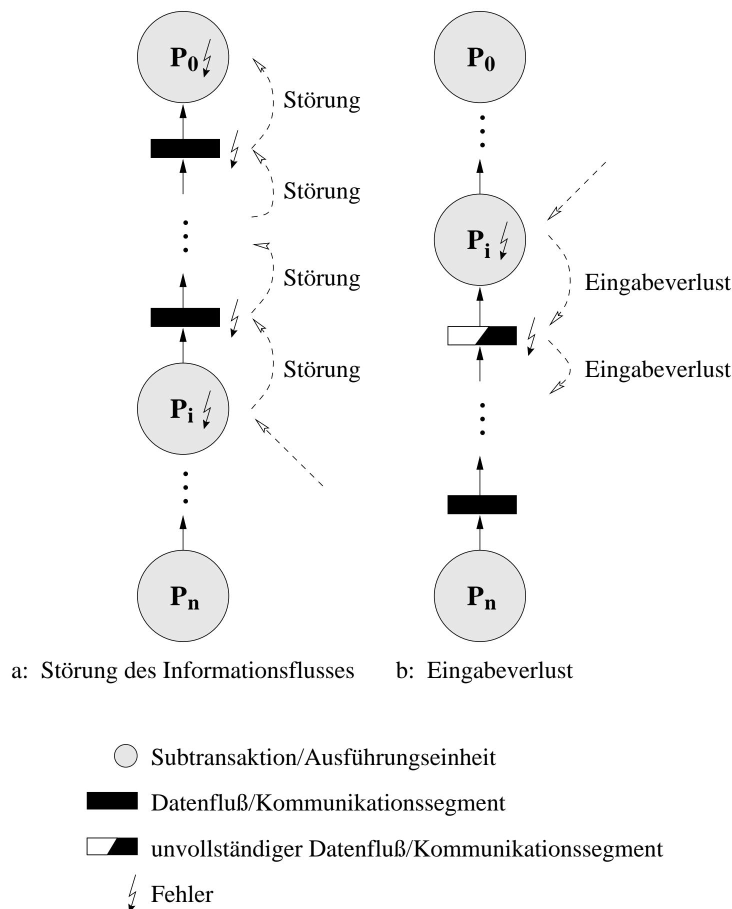

Abbildung 6.9: Reproduktionsbedingte Fehler bei Pipelining

- Pipelining:

  Zweck einer Pipeline ist die effiziente nebenläufige Datenverarbeitung entlang eines Astes im Ausführungsplan. Daher wird sehr schnell der Zustand erreicht, in dem alle an dieser Kommunikationsform beteiligten Subtransaktionen gleichzeitig auf Daten operieren. Tritt in einer Subtransaktion ein

Fehler auf, so sind alle auf dem Ast oberhalb liegenden Subtransaktionen durch Störungen des Informationsflusses bedroht, es sei denn, es existieren kompensierende Kommunikationssegmente. Wie in Abbildung 6.9 angedeutet, müssen nach dem Auftreten eines Fehlers in P_i alle durch ihn von reproduktionsbedingten Fehlern bedrohten Subtransaktionen P_{i-1} bis P_0 zur Reproduktion vorgesehen werden (a). Ähnlich stark gefährdet sind unterhalb liegende Komponenten. Alle Subtransaktionen, die nicht auf vollständige oder potentiell vollständige Datenflüsse zugreifen können, müssen reproduziert werden (b). In der Abbildung sind dies die Subtransaktionen P_{i+1} bis P_{n-1}.

- Partitionierung:

  Bei der Partitionierung verhält sich die Weitergabe von Störungen des Informationsflusses nach oben genau wie die im vorigen Abschnitt beschriebene Ausbreitung von Datenfehlern. Der Teil der Konsumenten, die eine bereits gefüllte Partition haben, sind von dem Fehler betroffen. Die Ausbreitung nach unten findet wie beim Pipelining statt. Entlang eines Astes sind alle unterhalb liegenden Komponenten betroffen, die nicht auf vollständige oder potentiell vollständige Datenflüsse zugreifen können. Abbildung 6.10 zeigt diese Ausbreitung.

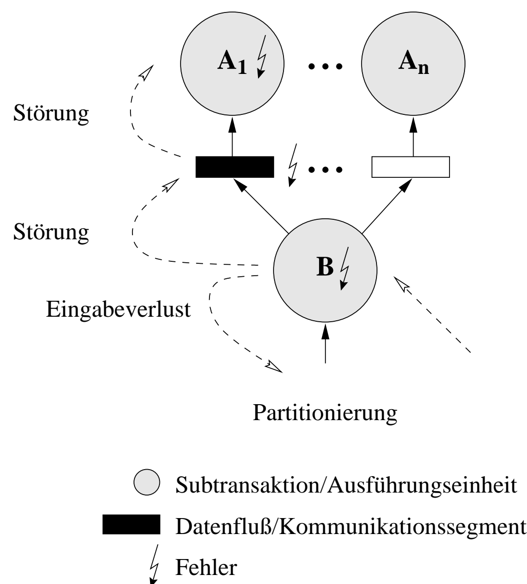

Abbildung 6.10: Reproduktionsbedingte Fehler bei einer Partitionierung

- Mischen, Repartitionierung und Replikation:

  Die Ausbreitung von reproduktionsbedingten Fehlern bei diesen Kommunikationsformen verhält sich wie bei der Partitionierung. Zusätzlich können auch Verzweigungen der Ausbreitung nach unten, also bei der Bestimmung des Eingabeverlustes entstehen, da bei diesen Formen mehrere Produzenten existieren. Des weiteren tritt bei diesen Kommunikationsformen der Effekt auf, daß sich Störungen des Informationsflusses und Eingabeverlust gegenseitig verursachen. Exemplarisch sei dies am Beispiel des Mischens in Abbildung 6.11 vorgeführt. Ein Fehler tritt in Subtransaktion B_i auf. Durch Störungen des Informationsflusses sind der darüberliegende Datenfluß und Subtransaktion A betroffen. Da A reproduziert werden muß, tritt nun ein Eingabeverlust in Richtung B_m auf, da dort nicht auf einen vollständigen oder potentiell vollständigen Datenfluß zugegriffen werden kann. Somit müssen auch dieser unvollständige Datenfluß sowie Subtransaktion B_m reproduziert werden. Auf diese Weise kann das Auftreten eines Fehlers nicht nur zur Reproduktion des Astes führen, auf dem er liegt, sondern der Fehler kann sich quer durch den gesamten Ausführungsplan ausbreiten.

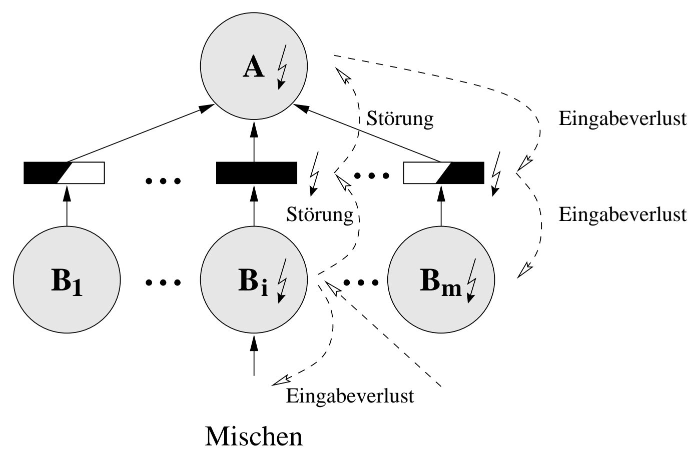

Abbildung 6.11: Reproduktionsbedingte Fehler beim Mischen

Fazit

Aus diesen Beispielen läßt sich ersehen, wie wichtig die Existenz von *vollständigen* oder *potentiell* vollständigen Datenflüssen beim Auftreten eines Fehlers ist. Ohne sie ist zum Fehlerzeitpunkt der komplette, unterhalb des Fehlers liegende Teil der Anfrage von reproduktionsbedingten Fehlern bedroht und kann beim Neustart nicht wiederverwendet werden. Gleiches gilt für *kompensierende Datenflüsse* (siehe Abschnitt 5.7.3.3 Punkt 4). Sind sie implementiert, können keine Störungen des Informationsflusses auftreten, alle oberhalb des Fehlers liegenden Komponenten sind nicht von reproduktionsbedingten Fehlern bedroht und können wiederverwendet werden.

6.3.3 Betrachtung midasspezifischer Situationen

Bei der Optimierung und Parallelisierung [Nippl 00] in MIDAS wird in der Regel ein möglichst hoher Grad an Unabhängigkeit zwischen den Transaktionskomponenten erzeugt. Die Subtransaktionen können dann unter geringen Einschränkungen nebenläufig und verteilt durchgeführt werden, d. h. es findet eine effiziente Nutzung der Systemressourcen statt.

Auf die Isolation zwischen einzelnen Ausführungseinheiten wurde bewußt verzichtet, damit im Rahmen der Parallelverarbeitung Pipelining möglich ist. Dadurch sind deutliche Performanzgewinne erreichbar. Das *Fehlen der Isolationseigenschaft* ist jedoch allein für die in den beiden vorangegangenen Abschnitten beschriebenen Ausbreitungsmöglichkeiten von Fehlern verantwortlich. Eine effiziente Parallelverarbeitung und die Fehlerbeseitigung sind somit *nicht orthogonal* zueinander und behindern sich damit in diesem Punkt.

Eine weitere Maßnahme ist das *Zusammenfassen mehrerer Subtransaktionen auf einer Ausführungseinheit*. Dies ist zwar nützlich, um den Kommunikationsaufwand zwischen verschiedenen Komponenten zu minimieren, hat aber gleichzeitig auch Auswirkungen auf die Fehlerbehandlung. Subtransaktionen auf einer Ausführungseinheit sind in MIDAS nicht voneinander isoliert. Tritt ein Komponentenfehler auf dieser Ausführungseinheit auf, so sind zwangsläufig alle auf ihr laufenden Subtransaktionen betroffen und müssen reproduziert werden. Auch die Datenflüsse zwischen den Subtransaktionen einer Ausführungseinheit beeinflussen die Fehlerbehandlung nicht positiv. Diese Datenflüsse sind als lokale Partitionen implementiert. Aus technischen Gründen sind solche Datenflüsse immer leer oder unvollständig, und sie können nicht als kompensierende Datenflüsse implementiert werden. Daher kann durch sie die Ausbreitung von Störungen des Informationsflusses oder des Eingabeverlustes nicht begrenzt werden. Sollen diese Eigenschaften auch innerhalb einer Ausführungseinheit unterstützt werden, so müßten statt lokalen Partitionen komplette Kommunikationssegmente eingesetzt werden, die dann allerdings die Vorteile der Zusammenlegung von Subtransak-

tionen auf einer Ausführungseinheit zunichte machen würden.

Der Optimierer in MIDAS bevorzugt *stark verzweigte* gegenüber *linearen Operatorbäumen*. Der Grund hierfür ist das Problem, daß zwischen Operatoren, die sich unter- und oberhalb blockierender Operatoren befinden, keine Parallelität stattfinden kann. Der Optimierer versucht daher, stark verzweigte Operatorbäume zu erzeugen und die blockierenden und zudem teuren Operatoren auf unterschiedlichen Ästen zu verteilen, da eine parallele Durchführung dieser Operatoren auf demselben Ast wegen ihrer blockierenden Eigenschaft nicht möglich ist. Zudem verteilt der Parallelisierer teure Operatoren möglichst auf verschiedene Ausführungseinheiten und damit Subtransaktionen. Das hat zur Folge, daß der Parallelisierer in seinem Bestreben, mehrere Teilpläne zusammenzufassen, eingeschränkt wird, da die Länge der Informationswege im stark verzweigten Baum tendentiell wesentlich kürzer ist als im linearen Baum. Abbildung 6.12 zeigt die Situation.

Durch die Verteilung blockierender Operatoren auf verschiedene Ausführungseinheiten und damit auch auf verschiedene Subtransaktionen, entsteht zusätzlich die günstige Situation, daß *Datenflüsse, die verschiedene Subtransaktionen verbinden, lange leer bleiben*. Treten Fehler in einzelnen Subtransaktionen auf, so können sie sich nicht über die leeren Datenflüsse ausbreiten, wodurch die übrigen Subtransaktionen wiederverwendet werden können.

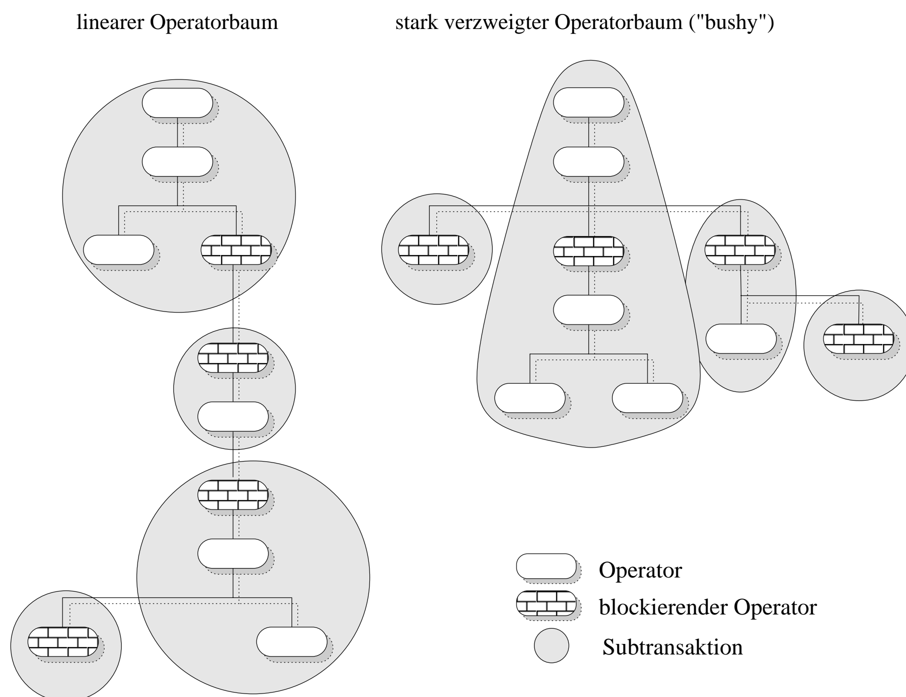

Abbildung 6.12: Verzweigung von Operatorbäumen und blockierende Operatoren

6.3.4 Eingabeverfügbarkeit

Eingabeverfügbarkeit wird in MIDAS durch vollständige oder potentiell vollständige Datenflüsse beziehungsweise Kommunikationssegmente erreicht. Solche Kommunikationssegmente sollten so plaziert werden, daß bei Auftreten eines Fehlers ein geringer Verlust an Informationen stattfindet, d. h. ein möglichst kleiner Teil der gesamten Anfrage betroffen ist.

Der Aufwand, der bei der Verarbeitung eines Zwischenergebnisses für die Erzeugung des jeweils nächsten Zwischenergebnisses entsteht, sollte für alle Subtransaktionen gleich sein. Das heißt, der Subtransaktionsplan sollte durch vollständige oder potentiell vollständige Datenflüsse in Gruppen von Subtransaktionen unterteilt sein, daß jede Gruppe annähernd den gleichen Rechenaufwand verursacht. Damit kann der maximale Reproduktionsaufwand beim Auftreten eines Fehlers beschränkt werden. Dieses Ziel wird durch den Parallelisierer unterstützt, da dieser versucht, kostenträchtige Teile der Anfragen möglichst gleichmäßig auf Ausführungseinheiten zu verteilen, um eine gute Ressourcenauslastung zu erzielen. Daher lassen sich vollständige Kommunikationssegmente ohne größere Behinderungen an den günstigen Positionen im Operatorbaum einfügen, die jeweils das *Zwischenergebnis eines teuren Teilplans* enthalten.

Ein weiterer Vorteil ist, daß der Parallelisierer die Subtransaktionen unter der Berücksichtigung von *Datenengpässen* bildet. Das sind Stellen, an denen wenige Daten fließen, die aber durch hohen Aufwand entstanden sind, beispielsweise oberhalb von Joins mit niedriger Joinrate oder Gruppierungen. Die Erzeugung derartiger Stellen wird dadurch gefördert, daß die dort entstehenden Kommunikationskosten vom Parallelisierer berücksichtigt werden und versucht wird, diese Kosten zu minimieren. Diese Stellen, an denen wenige Daten übergeben werden, sind besonders günstige Stellen, um Kommunikationssegmente potentiell vollständig oder vollständig zu halten, da diese Maßnahme bei kleinen Zwischenergebnissem geringe Kosten verursacht[1].

Des weitern wird vom Parallelisierer oft bestimmt, daß Kommunikationssegmente auf Festplatte gespeichert werden sollen. Dies geschieht bei großen Zwischenergebnissen, bei denen der Produzent schneller als der Konsument arbeitet oder wenn Verklemmungen der Subtransaktionen vermieden werden sollen [NipMit 98b, Nippl 00]. In diesen Fällen können die gespeicherten Zwischenergebnisse von der Fehlerbehandlung genutzt werden, ohne daß zusätzliche Kosten entstehen.

Der Fall, daß korrekt arbeitende Subtransaktionen abgebrochen werden müssen, um die Eingabeverfügbarkeit für fehlerhafte Subtransaktionen herzustellen, tritt

[1] Entweder durch wenig Schreiboperationen auf Platte oder bei sehr kleinen Zwischenergebnissen kann es sogar im Speicher gehalten werden.

nur dann ein, wenn Daten, die den Datenfluß schon durchlaufen haben, verworfen werden, d. h. bei unvollständigen Datenflüssen. Durch geeignete Auswahl von Datenflüssen, die immer potentiell vollständig oder vollständig gehalten werden, kann die Eingabeverfügbarkeit also sehr hoch gehalten werden.

6.3.5 Maßnahmen zur Leistungssteigerung

Die vorangegangenen Abschnitte haben gezeigt, welche die leistungskritischsten Eigenschaften der Fehlerbehandlung bei der Abbildung des FPT-Modells auf die MIDAS-Gegebenheiten sind und welche Einschränkungen sich daraus ergeben. Insbesondere die Ausbreitung reproduktionsbedingter Fehler hat Einfluß darauf, wie effizient die Fehlerbehandlung in MIDAS ist. In den folgenden Abschnitten werden Maßnahmen abgeleitet, die die Effizienz steigern.

6.3.5.1 Kompensierende Datenflüsse

Die Implementierung kompensierender Datenflüsse in MIDAS gewährleistet den korrekten Informationsfluß nach der Reproduktion von Subtransaktionen. Diese Art reproduktionsbedingter Fehler kann dann nicht mehr auftreten, wodurch die Fehlerausbreitung stark eingeschränkt wird.

Für die Implementierung dieser Datenflüsse muß gewährleistet sein, daß die Reproduktion des Produzenten vor den Konsumenten verborgen bleibt. Nach dem Neustart des Produzenten wird dieser seine Ausgabe erneut erzeugen. Der Datenfluß muß in der Lage sein, aus dieser Sequenz den schon vor dem Auftreten des Fehlers weitergeleiteten Teil herauszufiltern und nicht noch einmal an den Konsumenten weiterzugeben.

Für die Realisierung muß den Kommunikationssegmenten die Fähigkeit gegeben werden, zu erkennen, welche Tupel bereits vom Konsumenten gelesen wurden. Dazu muß die Anfangssequenz in den Daten der reproduzierten Subtransaktionen bestimmt werden können, die schon gelesen wurde. Kommunikationssegmente speichern keine Identifikatoren der Tupel, die sie transportieren. Eine direkte Identifizierung einzelner Tupel ist damit nicht möglich. Für Kommunikationssegmente, die noch die komplette Sequenz beinhalten, die sie vor Auftreten des Fehlers transportiert hatten, ist eine solche Erkennung einfach. Sie entsprechen den potentiell vollständigen Datenflüssen. Es müssen nur die im Kommunikationssegment befindlichen Daten mit den neu gesendeten verglichen werden. Aber auch für unvollständige Kommunikationssegmente ist dies einfach zu erreichen und bedarf keines aufwendigen Vergleichs. Da die Segmente nicht in beliebiger Reihenfolge, sondern nur sequentiell beschrieben werden können, genügt ein Zähler, der die Position des letzten geschriebenen Tupels enthält. Nach Reproduktion des Konsumenten muß das Kommunikationssegment nur die Anzahl der neu pro-

duzierten Tupel zählen, alle bis zu dem gespeicherten Zählerwert ankommenden Tupel verwerfen und danach alle weiteren Tupel normal weiterleiten.

Dies funktioniert nur, wenn der Konsument deterministisch arbeitet, d. h. nach der Reproduktion dieselbe Datensequenz liefert. Dies ist jedoch gewährleistet, da der Algorithmus zur Fehlerbehandlung nichtdeterministische Subtransaktionen speziell behandelt.

6.3.5.2 Vermeidung von nichtdeterministischen Operatoren

Nichtdeterministische Operatoren, wie sie in den Abschnitten 5.7.3.3 und 6.2.1.2 beschrieben wurden, können auf viele Weisen vermieden werden.

- DtCur-Operator:

  Die Implementierung dieses Operators berechnet das Datum mit dem Beginn der zugehörigen Subtransaktion. Dadurch liefert er bei einer Reproduktion der ihn enthaltenden Subtransaktion einen anderen Wert. Dies kann, wenn bereits Daten mit dem alten Wert weitergeleitet wurden, zu inkonsistenten Ergebnissen führen.
  Abhilfe schafft hier, den Wert des Operators zum Zeitpunkt des Beginns der Anfrage festzulegen. Diese Regelung erfüllt die Spezifikation des Operators und verursacht keine zusätzlichen Kosten.

- Func- und Udto-Operator:

  Bei jeder Implementierung dieser benutzerdefinierten Operatoren muß im allgemeinen davon ausgegangen werden, daß sie sich nichtdeterministisch verhalten. Zur Unterstützung der Fehlerbehandlung kann eine Erweiterung der Operatoren dienen, in der der Programmierer der benutzerdefinierten Funktion angibt, ob sich der Operator deterministisch verhält. Die Fehlerkontrolle kann dann zwischen den deterministischen und den wirklich nichtdeterministischen Operatoren unterscheiden. Sie muß dann nicht alle als nichtdeterministisch einstufen.

- Recv-Operator:

  Der Recv-Operator verhält sich nichtdeterministisch, wenn er asynchron mischt. Das Problem entsteht mit der Weitergabe der ungeordneten Daten. Vor der Weitergabe der Daten an andere Subtransaktionen müßte daher in ausreichender Weise sortiert werden. Wenn dies ohnehin geschieht, ist kein Problem vorhanden. Ansonsten ist eine sortierte Weitergabe dadurch zu erreichen, daß hinter dem Recv- ein Sortieroperator eingefügt wird, der die Tupel vollständig sortiert. Diese kostenaufwendige Methode würde aber den Performanzvorteil des asynchronen Mischens völlig zunichte machen. Alternativ könnte der Recv-Operator zwar asynchron empfangen, aber einen

Satz von Tupeln jeweils dann weitergeben, wenn er aus jedem Kommunikationssegment mindestens eine Subseite beziehungsweise Seite gelesen hat. Weitergegeben werden dann zuerst alle Tupel einer Seite des ersten Kommunikationssegments, dann die einer Seite des zweiten, bis hin zum letzten Segment. Dadurch würde zwar keine sortierte Ausgabe von Tupeln erzeugt werden, aber eine deterministische, was für die Reproduktion genügt. Dies wäre eine Variante zwischen dem asynchronen und dem normalen Mischen, da im Gegensatz zum normalen Mischen keine Sortierung stattfände. Entsprechend wäre mit einer geringeren Performanzeinbuße zu rechnen. Ob die Realisierung dieser Variante empfehlenswert ist, hängt von der allgemeinen Fehleranfälligkeit des Systems ab und von dem Grad der Bevorzugung des asynchronen Mischens durch den Parallelisierer.

6.4 Transaktionen mit mehreren Anfragen und ändernde Anfragen

6.4.1 Transaktionen mit mehreren Anfragen

Die bisherigen Abschnitte haben gezeigt, wie die Subtransaktionen des FPT-Modells auf die Teilpläne einer Anfrage abgebildet werden. Der gesamte Subtransaktionsbaum hat dabei eine komplette Anfrage dargestellt. Die gesamte Anwendung dieses Konzepts ist demnach bisher auf die Fehlerkorrektur innerhalb einzelner Anfragen beschränkt.

Daraus ergibt sich die Frage, ob und wie dieses Konzept anwendbar ist, falls mehrere SQL-Anfragen zu einer SQL-Transaktion zusammengeschlossen sind. Nach den bisherigen Überlegungen könnte die Fehlerkorrektur mit dem beschriebenen Vorgehen immer auf der letzten, gerade aktiven Anfrage einer SQL-Transaktion stattfinden. Das heißt, daß nach dem Auftreten eines Fehlers kontrolliert werden muß, welchen Teil der SQL-Transaktion der Fehler betrifft. Ist nur die in diesem Moment aktive Anfrage betroffen, so kann das vorgeschlagene Konzept zum Einsatz kommen und nach dem Algorithmus zur Fehlerbehandlung verfahren werden. Betrifft der Fehler auch frühere Anfragen der SQL-Transaktion, so müßte die gesamte Transaktion abgebrochen werden.

Nun muß festgestellt werden, wie relevant der zweite Fall ist, bei dem die komplette Transaktion abgebrochen werden muß. Zwei Argumente zeigen, daß dieser Fall nur selten eintreten kann.

Als erstes wird festgehalten, von welcher Bedeutung es ist, wenn ein Fehler frühere Anfragen betrifft. Soll das vorgeschlagene FPT-Modell verwendet werden, so heißt das, daß eine der früheren Anfragen teilweise rückgängig gemacht werden muß und reproduziert, also erneut gestartet wird. Ein solcher Neustart einer

schon beendeten Anfrage steht aber in Konflikt mit der Tatsache, daß zwischen den Anfragen einer SQL-Transaktion Benutzerinteraktion stattfinden kann. Bei einer automatischen Reproduktion eines Teils einer SQL-Transaktion, der mehr als die gerade laufende Anfrage umfaßt, kann diese Benutzerinteraktion nicht mit einkalkuliert werden. Die Reproduktion ist daher nicht möglich. Eine solche Behandlung von Fehlern, die mehr als die aktive Anfrage betreffen, kommt demnach nur für SQL-Transaktionen in Frage, die in einem Stapelbetrieb ohne Benutzerinteraktion stattfinden.

Als zweites muß betrachtet werden, welche Fehlerarten überhaupt schon beendete Anfragen einer SQL-Transaktion betreffen. Dabei stellt sich heraus, daß von den Transaktionsfehlern (siehe Abschnitt 5.7.2), die vom FPT-Modell unterstützt werden, alle Komponentenfehler nur aktive Komponenten betreffen. Sowohl der Ausfall als auch die Verklemmung zwischen Transaktionskomponenten betreffen immer nur die gerade aktive Anfrage. Auch Datenfehler werden vorwiegend in der aktuell aktiven Anfrage auftreten. Einzige Ausnahme ist der in Abschnitt 6.2.1.1 beschriebene Fall, in dem einer Transaktion zur Auflösung eines Sperrkonflikts eine Sperre entzogen wird und daraufhin alle Subtransaktionen einen Datenfehler erhalten, die Datenobjekte gelesen haben, die durch diese Sperre geschützt sind. Findet zwischen den einzelnen Anfragen einer Transaktion Benutzerinteraktion statt, so kann dieser Fall, wie oben beschrieben, vom FPT-Modell nicht behandelt werden. Findet keine Benutzerinteraktion statt, so hat die schon beendete Anfrage mindestens eine schon sichtbare Ausgabe produziert oder permanente Daten der Datenbank verändert. In beiden Fällen kann das FPT-Modell nicht eingesetzt werden, da es nur auf die Wiederverwendung von Zwischenergebnissen ausgelegt ist, nicht aber auf Recovery permanenter Datenbankdaten oder kontrolliertes Rücksetzen von Transaktionsteilen durch den Benutzer[2].

Daraus ergibt sich, daß alle Fälle, in denen schon beendete Anfragen einer SQL-Transaktion von Fehlern betroffen sind, von dem FPT-Modell nicht korrigiert werden können. Bei solchen Fehlern muß daher die komplette SQL-Transaktion abgebrochen werden, was dem Verhalten der flachen Transaktionen entspricht. Wie bereits beschrieben, ist allerdings in den meisten Fällen nur die aktuell aktive Anfrage von Fehlern betroffen. Somit kann das vorgeschlagene FPT-Modell eingesetzt werden.

6.4.2 Ändernde Anfragen

Bei einer Betrachtung der Sprache SQL und ihrer Übersetzung läßt sich feststellen, daß die Operatoren, die ändernde DML-Anfragen implementieren, wie

[2]Kontrolliertes Rücksetzen von Transaktionsteilen wird in einigen Implementierungen geschachtelter Transaktionen unterstützt. Beispiel: Reisebuchungstransaktion.

INSERT, UPDATE oder DELETE, immer direkt an der Wurzel des aus der Übersetzung entstandenen Operatorbaumes stehen müssen. Der komplette Rest des Operatorbaumes entspricht einer SELECT-Abfrage, die die von der Änderung betroffenen Tupel oder Werte bestimmt. Diese Operatoren werden in MIDAS nicht parallelisiert, das heißt, es gibt in ihnen keine Intra-Operator-Parallelität. Dies hat zur Folge, daß jede parallelisierte, ändernde Anfrage maximal einen solchen Operator enthält und dieser sich in der Subtransaktion befindet, die Wurzel des Subtransaktionsbaumes ist.

Alle Maßnahmen des FPT-Modells können nur so lange greifen, als keine Daten die Wurzelsubtransaktion erreicht haben, die den ändernden Operator enthält. Sobald dieser Operator Änderungen an Datenbankdaten vornimmt, können nur noch die normalen Maßnahmen der Recovery greifen, wie die Sicherung der Anfrageatomarität (siehe Abschnitt 5.3.3), wodurch die gesamte Anfrage zurückgesetzt wird. Das vorgeschlagene FPT-Modell sieht für Änderungen an permanenten Datenbankdaten keine Konzepte vor, es ist allein auf die Wiederverwendung von Zwischenergebnissen ausgelegt.

Bei allen ändernden Anfragen, bei denen der SELECT-Anteil der Anfrage groß ist, verspricht der Einsatz des Konzepts trotzdem lohnend zu sein. Insbesondere dann, wenn dieser Anteil blockierend ist, d. h. erst alle zur Änderung vorgesehenen Tupel bestimmt und danach die Änderungen vollzogen werden (deferred update). Enthält eine Anfrage keine blockierenden Operatoren und alle Änderungen werden sofort vollzogen (update on the fly), wird das Konzept im Fehlerfall keine Vorteile bringen können. Es entstehen dadurch allerdings auch keine Nachteile.

6.5 Leistungsanalysen

6.5.1 Implementierung

Die Implementierung des FPT-Modells in MIDAS konnte nicht vollständig durchgeführt werden. Es war einfach, den Algorithmus zur Bestimmung der neu zu startenden Subtransaktionen zu realisieren. Die Durchführung des Abbruchs und Neustarts von Subtransaktionen erwies sich aber als sehr schwer durchführbar [Drügh 99].

Der Grund hierfür ist die Art der Implementierung der Kommunikationssegmente [Brandmayer 97], die sehr komplexe Verwaltungsstrukturen aufweisen. Um für eine fehlerhafte Subtransaktion eine neue Subtransaktion in den Subtransaktionsbaum einzufügen, beziehungsweise eine bereits laufende Subtransaktion neu zu starten (Reproduktion), müßten alle Kommunikationsverbindungen der alten Subtransaktion aus dem Subtransaktionsbaum gelöst und neu gestartet, oder durch neue Kommunikationsverbindungen ersetzt werden. Dies war allerdings

bei den jetzigen Verwaltungsmechanismen der Kommunikationssegmente nicht durchführbar. Es ist weder das asynchrone Abbrechen der Kommunikationsverbindungen möglich, das bei der Reproduktion eines Kommunikationssegments notwendig wäre, noch kann an ein Kommunikationssegment ein neuer Produzent angehängt werden, falls die bisher in das Kommunikationssegment schreibende Subtransaktion reproduziert wird. In beiden Fällen bleiben viele Ressourcen des Systems belegt, wodurch Verklemmungen in der Pufferverwaltung entstehen.

Die Kommunikationssegmente unterstützen diese Mechanismen zum Abbruch in keiner Weise, da sie nicht für derartige Eingriffe ausgelegt wurden. Zum Zeitpunkt des Entwurfs der Kommunikationssegmente waren solche Anforderungen noch nicht absehbar. Eine Anpassung der Kommunikationssegmente würde eine Neuimplementierung notwendig machen, was zu aufwendig ist.

Allerdings sind auch ohne die vollständige Implementierung Aussagen über den Einsatz des FPT-Modells in einem realen System möglich. Dies wird an einem Fallbeispiel im folgenden Abschnitt vorgeführt.

6.5.2 Fallbeispiel

Dieser Abschnitt zeigt ein Fallbeispiel, an dem sich Aussagen über den Einsatz des FPT-Modells in einem realen System treffen lassen.

Dazu wurden parallelisierte Anfragen ausgeführt und für den so erzeugten fehlerfreien Ablauf die Laufzeiten sowie das entstandene Nachrichtenvolumen in den Systemkomponenten bestimmt. Daraus läßt sich ermitteln, wie die Ausführung der Anfragen im Fehlerfall ablaufen würde.

Der bei der Parallelisierung der Anfrage entstehende parallele Subtransaktionsbaum ist in Abbildung 6.13 dargestellt. In den einzelnen Subtransaktionen sind kurz die wichtigsten Operatoren vermerkt, die darin ausgeführt werden.

Die Anfrage wurde auf dem 4-Rechnersystem mit einer 168 MByte großen TPC–D-Datenbank ausgeführt (siehe Anhang C und Anhang D). Die Laufzeit betrug 98 Sekunden. Abbildung 6.14 stellt den zeitlichen Verlauf des Verhaltens der Subtransaktionen und Kommunikationssegmente während der Laufzeit dar. Die Skalierung der Achse erfolgt in Prozent der Gesamtlaufzeit der Anfrage bei fehlerloser Ausführung. Im folgenden Text werden alle Zeitpunkte des zeitlichen Verlaufs auf die Gesamtlaufzeit der Anfrage bei fehlerloser Ausführung bezogen. "Zeitpunkt 30" entspricht demnach dem Zeitpunkt nach Versteichen von 30 % der Gesamtlaufzeit.

Die Subtransaktionen A und A' werden in einer Ausführungseinheit ausgeführt und sind als Wurzelknoten während der gesamten Laufzeit aktiv (Zeitpunkte 0 – 100). Zwischen ihnen befindet sich das Kommunikationssegment KS 1, das in Form einer lokalen Partition vorliegt (siehe Abschnitt 3.4). Als lokale Partiti-

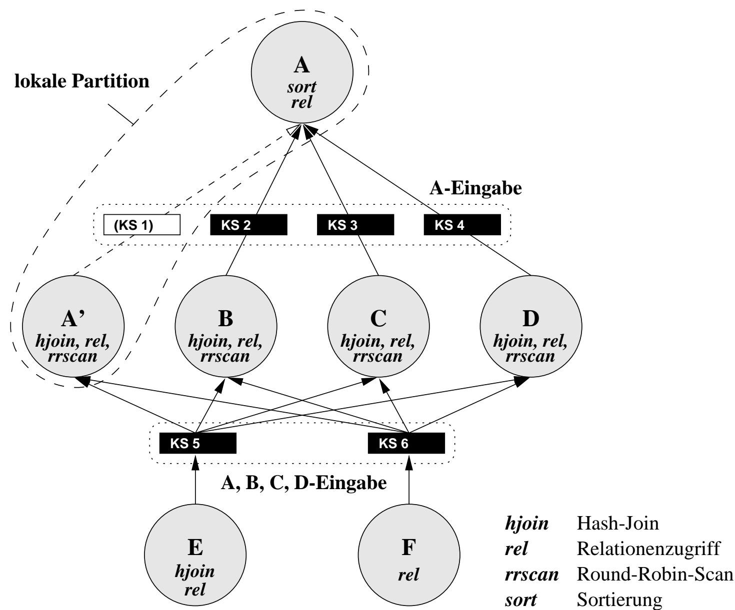

Abbildung 6.13: Subtransaktionsbaum der TPC–D-Anfrage 9

on ist es immer unvollständig. Die Subtransaktionen B, C und D führen die Hauptteile der Berechnung durch (Zeitpunkte 0 – 70). Die von ihnen erzeugten Zwischenergebnisse, mit einer Größe von jeweils ca. 2 MByte, werden über die Kommunikationssegmente KS 2 bis KS 4 zu A transferiert. A' verrichtet dieselbe Aufgabe wie B, C und D, braucht dafür aber länger (bis ca. 96), da es sich mit A im selben Prozeß befindet. Die Subtransaktionen E und F sind nur sehr kurz aktiv (bis zum Zeitpunkt 8). Ihre Ausgaben sind klein (25 und 49 KByte) und werden von mehreren Transaktionen (A' – D) gelesen. Deshalb bleiben sie bis zum Ende der Transaktion vollständig erhalten.

Die Arbeitsweise und Effizienz des FPT-Modells wird nun an folgenden Beispielen möglicher Fehlerszenarien verdeutlicht:

- **Keine Vorkehrungen** — Fehler bei Zeitpunkt 30 in A oder A', KS 2 – KS 4 sind unvollständig:

  Dies ist der ungünstigste Fall. Es müssen A und A' reproduziert werden, da sie direkt vom Fehler betroffen sind. Zusätzlich werden B, C und D reproduziert (Bereitstellung von Eingabedaten). Damit ist bis auf die kleinen Subtransaktionen E und F die gesamte Transaktion betroffen. Der größte Teil der bis zum Fehlerzeitpunkt verrichteten Arbeit ist verloren gegangen. Das FPT-Modell kann nicht gewinnbringend eingesetzt werden.

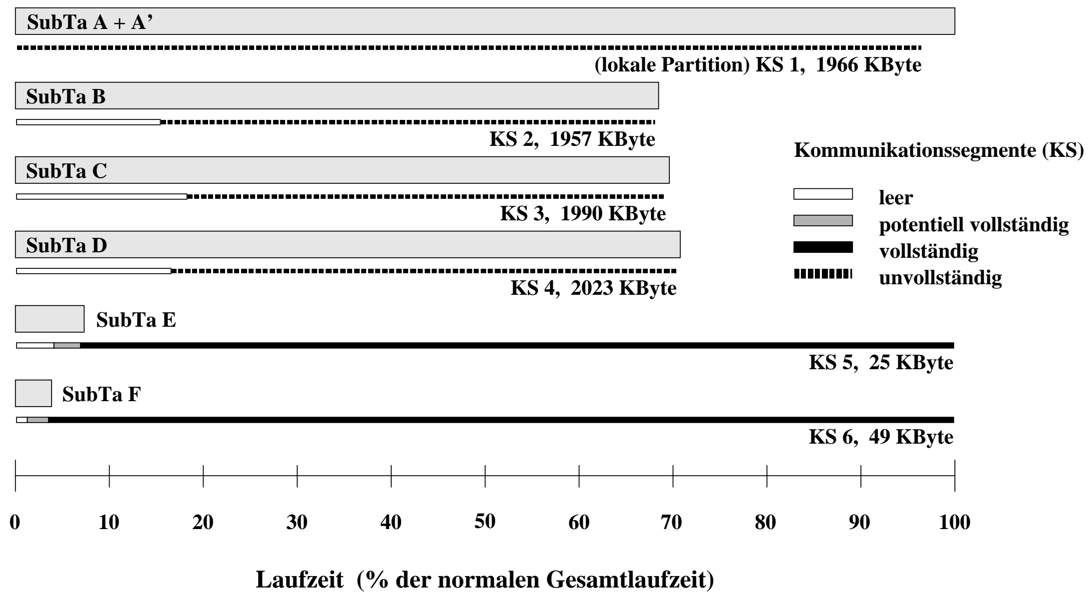

Abbildung 6.14: Laufzeitverhalten der TPC–D-Anfrage 9

- **Potentielle Vollständigkeit** — Fehler bei Zeitpunkt 30 in A oder A', KS 2 – KS 4 werden vollständig oder potentiell vollständig gehalten:

  Werden die Kommunikationssegmente KS 2 bis KS 4 so konfiguriert, daß sie während der Laufzeit nie unvollständig werden, kann sich der Fehler nicht nach unten ausbreiten. Der Fehler bleibt auf A beschränkt. Abbildung 6.15 veranschaulicht diese Situation.

  Nach dem Auftreten des Fehlers bei Zeitpunkt 30 werden A und A' reproduziert. Eine Reproduktion der Subtransaktionen B, C und D ist nicht notwendig, da die Kommunikationssegmente KS 2 bis KS 4 zu diesem Zeitpunkt potentiell vollständig sind und so die Eingabedaten für A bereitstellen können. Die Subtransaktionen B, C und D sind, wie im fehlerlosen Fall, zum Zeitpunkt 70 beendet. A benötigt nach dem Neustart bei Zeitpunkt 30 wieder seine ursprüngliche Laufzeit, so daß die gesamte Transaktion zum Zeitpunkt 130 endet.

  Bei Abbruch der kompletten Transaktion beim Auftreten des Fehlers und dem anschließenden Neustart wäre die Transaktion ebenfalls zu diesem Zeitpunkt beendet gewesen. Es kann demnach kein Laufzeitgewinn erzielt werden. Allerdings kann etwa 80 % der vor dem Auftreten des Fehlers verrichteten Arbeit wiederverwendet werden, da nur A und A' reproduziert werden müssen. Durch diese Ersparnis kann das System entsprechend entlastet werden.

  Dieser Gewinn im Fehlerfall wurde durch die entsprechende Konfiguration

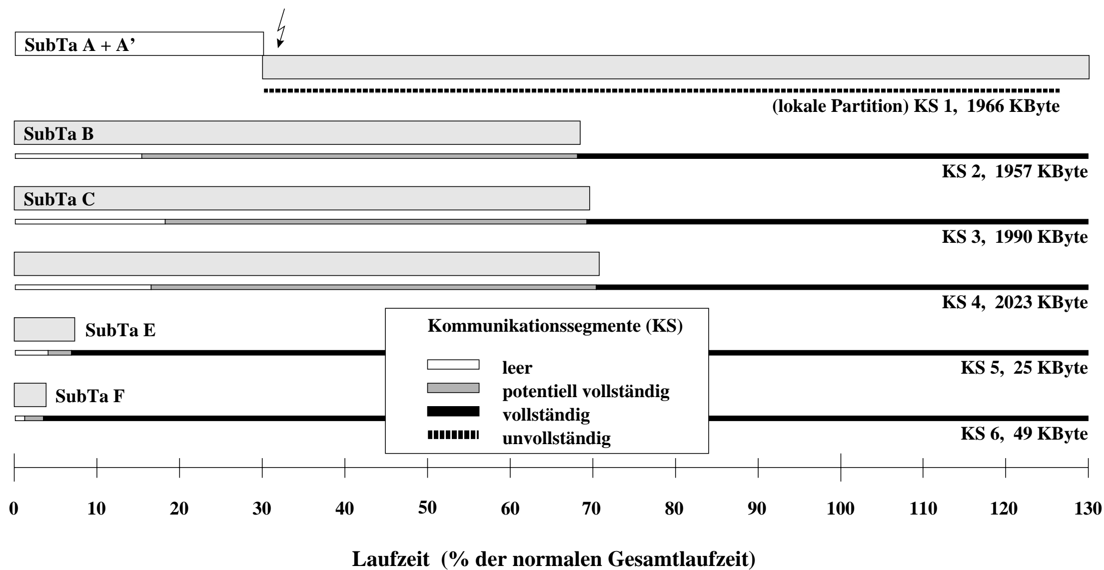

Abbildung 6.15: TPC–D-Anfrage 9 – Potentielle Vollständigkeit

der Kommunikationssegmente erzielt. Die Kosten dieser Maßnahme betragen bei dieser Anfrage 12 Sekunden. Wird also diese Sicherheitsvorkehrung bei Beginn der Anfrage gewählt, so verlängert sich die Laufzeit der Anfrage im fehlerlosen Fall um etwa 12 %.

- **Kompensierende Kommunikationssegmente** — Fehler bei Zeitpunkt 30 in B:

  Sind kompensierende Kommunikationssegmente implementiert, wird die Fehlerausbreitung in Richtung Wurzel verhindert. Der Fehler bleibt auf B beschränkt. Abbildung 6.16 veranschaulicht diese Situation.

  Nach dem Auftreten des Fehlers zum Zeitpunkt 30 wird B reproduziert. Eine Reproduktion der Subtransaktionen E und F ist nicht notwendig, da die Kommunikationssegmente KS 5 und KS 6 zu diesem Zeitpunkt vollständig sind und so die Eingabedaten für B bereitstellen können. A muß ebenfalls nicht reproduziert werden, da das kompensierende Kommunikationssegment KS 2 die Reproduktion von B vor A verbergen kann. Somit tritt keine Störung des Informationsflusses auf.

  Das Ende der Subtransaktion B, und somit das Eintreffen der letzten Daten von B nach A, wird durch die Reproduktion etwa auf den Zeitpunkt 98 verschoben. Da A seine Arbeit etwa 4 Einheiten nach Eintreffen der letzten Daten beenden kann, endet A und damit die gesamte Transaktion zum Zeitpunkt 102.

  Trotz des Auftretens eines Fehlers ist die Laufzeit der Anfrage nur unwe-

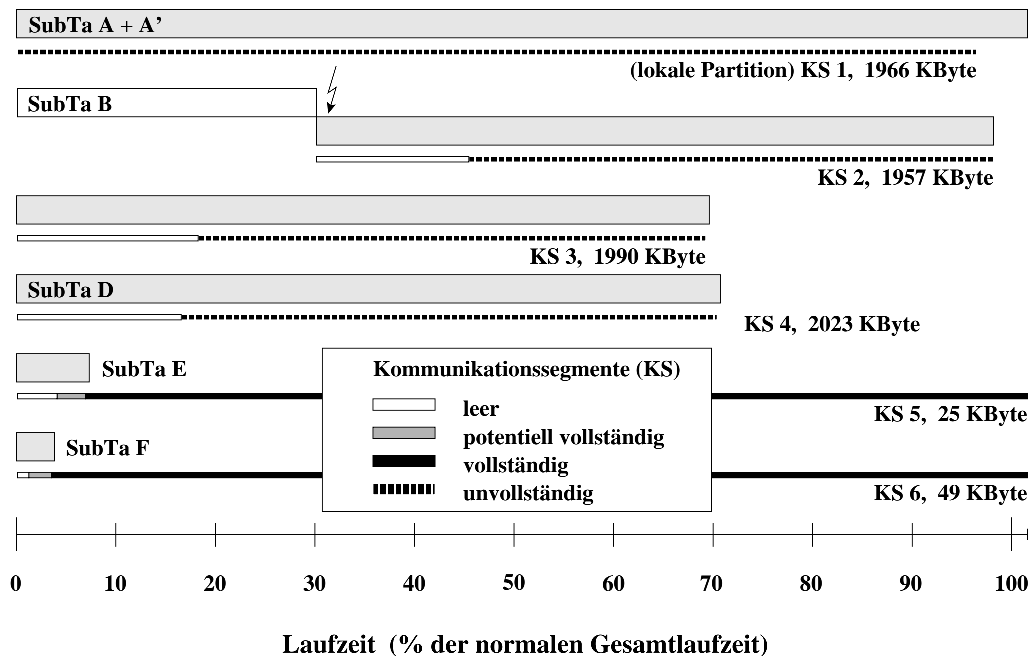

Abbildung 6.16: TPC–D-Anfrage 9 – Kompensierende Datenflüsse

sentlich verlängert worden. Ebenso wie im vorherigen Fall konnten 80 % der vor dem Fehler verrichteten Arbeit wiederverwendet werden. Die Arbeit von Subtransaktion B, die wiederholt werden muß, kann während der Ausführung von A von einem der freien Prozessoren durchgeführt werden.

Kompensierende Kommunikationssegmente verursachen keine zusätzlichen Kosten zur Laufzeit und sind einfach zu implementieren (siehe Abschnitt 6.3.5.1).

Diese Fallbeispiele zeigen, wie groß der Gewinn im Fehlerfall bei Einsatz des FPT-Modells sein kann. Die Gewinne resultieren im wesentlichen aus dem Vorhandensein vollständiger oder potentiell vollständiger sowie kompensierender Kommunikationssegmente. Kompensierende Kommunikationssegmente verursachen keinen Mehraufwand zur Laufzeit. Vollständige oder potentiell vollständige Kommunikationssegmente erzeugen ebenfalls keine Zusatzkosten, wenn sie ohnehin schon eingesetzt werden (KS 5 und KS 6, vom Parallelisierer bestimmt). Ansonsten entstehen Zusatzkosten, deren Höhe von der Größe des Datenstroms abhängt, der das Kommunikationssegment durchläuft. In diesem Fall ist abzuwägen, ab welcher Höhe der Kosten die Sicherheitsmaßnahme zu aufwendig wird.

6.6 Zusammenfassung

Die Abbildung des FPT-Modells zur Fehlerbehandlung auf das Ausführungsmodell von MIDAS läßt sich ohne Schwierigkeiten durchführen. Dies liegt nicht zuletzt daran, daß die Entwicklung des FPT-Modells eng an den Anforderungen eines stark auf Parallelausführung ausgerichteten Systems wie MIDAS orientiert ist.

Das FPT-Modell profitiert im wesentlichen von der Wiederverwendung von Zwischenergebnissen. Es kann ohne zusätzliche Maßnahmen während des normalen Ablaufs der Anfragen eingesetzt werden. Dadurch entsteht auch keine Beeinflussung der Laufzeit der Anfragen. Beim Auftreten eines Fehlers sind dann aber unter Umständen nur wenige Zwischenergebnisse vorhanden, und ein großer Teil der letzten Anfrage muß wiederholt werden. Dies ist aber immer noch günstiger, als die komplette Anfrage wiederholen zu müssen.

Andererseits können auch während des Ablaufs der Anfragen schon vorsorgliche Maßnahmen für den Fehlerfall getroffen werden, indem gezielt Zwischenergebnisse aufbewahrt werden. Im Fehlerfall müssen dann nur noch kleine Teile der Anfrage wiederholt werden. Dafür kann sich die normale Anfrageausführung wegen des zusätzlichen Speicherbedarfs im Arbeitspeicher oder wegen des Auslagerns der Zwischenergebnisse verlangsamen.

Als Stellen, an denen solche Zwischenergebnisse aufbewahrt werden sollen, bieten sich Kommunikationskanäle an, deren Datenmengen klein sind. Die Erzeugung kleiner Zwischenergebnisse ist aber gleichzeitig auch ein Ziel der Parallelisierung, da dort eine potentiell teure Überschreitung von Prozeßgrenzen stattfindet. Somit unterstützen sich beide Konzepte. Außerdem kann gezielt oberhalb teurer Teilbäume gesichert werden. Es existieren zudem Kommunikationskanäle, die aus Gründen der Parallelisierung sicherlich vollständig gehalten werden. Diese sind demnach kostenlos für die Fehlerbehandlung zu erhalten. Durch gute Auswahl der für die Wiederverwendung ausgewählten Stellen sollte der Normalfall nur sehr wenig belastet sein.

Wichtig ist, daß das FPT-Modell nicht auf die Sicherung von Zwischenergebnissen angewiesen ist und damit nur nach Kostenaspekten entscheiden kann, wann eine solche Sicherung vorzunehmen ist.

Wieviele vorsorgliche Sicherungen von Zwischenergebnissen eingesetzt werden, hängt sicherlich von den Anforderungen an das System bezüglich der Fehlertoleranz ab. Für genaue Strategien, wann vorsorgliche Sicherungen von Zwischenergebnissen stattzufinden haben, müßte ein Kostenmodell entworfen werden, durch das beim Anlegen der Kommunikationskanäle entsprechende Maßnahmen ergriffen werden können. Dazu sind aber umfangreichere Studien an einer konkreten Implementierung notwendig.

Weiterhin ist unbedingt zu empfehlen, kompensierende Datenflüsse zu implementieren, was leicht zu bewerkstelligen ist, keine Laufzeitnachteile bringt, dafür aber die Fehlerausbreitung sehr stark einschränkt.

Durch Analysen realer Anfragen wurde die Leistungsfähigkeit des FPT-Modells im Fehlerfall bestätigt.

Insgesamt läßt sich sagen, daß die Realisierung des FPT-Modells zur Fehlerbehandlung in MIDAS innerhalb der Aktivitätskontrolle gewinnbringend ist.

Als Einsatzgebiet des FPT-Modells können insbesondere Anwendungen mit großen, lesenden, lang laufenden Anfragen, wie Data Warehousing, Decision Support oder Systeme mit Anfragen ähnlich dem TPC–D-Benchmark gesehen werden.

Kapitel 7

Zusammenfassung der Arbeit

7.1 Ergebnisse

Diese Arbeit konzentrierte sich auf die zwei Schwerpunkte Entwicklung und Evaluierung des relationalen, parallelen Shared-Disk-Datenbanksystems MIDAS sowie den Entwurf des FPT-Modells, eines Modells zur Fehlerbehandlung in parallelen Transaktionen.

7.1.1 Der Datenbankprototyp MIDAS

Die Entwicklung des parallelen Datenbanksystems MIDAS aus dem sequentiellen Datenbanksystem TransBase konnte erfolgreich vollzogen werden. Es wurde eine Systemarchitektur geschaffen, die sowohl Inter-Transaktionsparallelität als auch Intra-Transaktionsparallelität ermöglicht. Intra-Transaktionsparallelität kann dabei sogar bis hin zur Intra-Operator-Parallelität ausgenutzt werden.

Bei der Implementierung des Prototyps konnten sehr große Teile von TransBase fast unverändert übernommen werden. Durch diesen evolutionären Entwicklungsansatz stand sehr schnell und mit verringertem Arbeitsaufwand ein funktionsfähiger Prototyp zur Verfügung, bei dem große Subsysteme von vornherein stabil und ausgereift waren.

Im Laufe der Entwicklung von MIDAS konnten unter anderem eine verteilte Pufferverwaltung mit in Subseiten unterteilten Seiten (vorgeschlagen in [Listl 96]), ein On-Request-Invalidierungsverfahren mit zugehöriger Sperrverwaltung (vorgeschlagen in [Bozas 98]), Operatoren zur Kommunikation paralleler Prozesse, die PVM-Nachrichtenbibliothek sowie ein neuer Optimierer, Parallelisierer und Scheduler [Nippl 00] und benutzerdefinierte Tabellenoperatoren [Jaedicke 99] erfolgreich in die Systemarchitektur integriert werden.

Durch die leichte Portierbarkeit der Anwendungen von TransBase auf MIDAS standen mit dem Bibliothekssystem OMNIS und einigen Benchmark-Programmen schnell geeignete Anwendungen zur Lastgenerierung zur Verfügung.

Die Implementierung von MIDAS zeigte, wie über geeignete interne Synchronisationsverfahren eine effiziente Ausführung der Aufträge mit hohem Parallelitätsgrad gewährleistet werden kann.

Systeminterne Kommunikationskosten stellten sich in vielen Fällen als der leistungsbegrenzende Systemfaktor heraus. MIDAS als paralleles Shared-Disk-Datenbanksystem verursacht ein hohes Nachrichtenaufkommen mit vielen kleinen Kontrollnachrichten und großen Seitennachrichten. Bei steigender Last wurde die teure Kommunikation schnell zum Engpaß im System.
Geeignete Verfahren, um dieses Problem zu umgehen, sind eine effizientere Implementierung der Kommunikation im System (wodurch aber nur ein konstanter Faktor gewonnen werden kann), die Vermeidung von Kommunikation durch eine geeignete Systemarchitektur (Architekturumstellung), die Wahl von Algorithmen der Pufferverwaltung mit niedriger Nachrichtenanzahl (beispielsweise On-Request-Invalidierung) sowie andere Verfahren, die in einer geringeren Zahl versendeter Nachrichten resultieren (beispielsweise lokalitätsbasiertes Transaktionsrouting).

Zudem stellte es sich als notwendig heraus, Daten im parallelen Datenbanksystem MIDAS auf mehrere Festplatten zu verteilen, um diese nicht zu schnell zum Leistungsengpaß werden zu lassen. Dadurch näherte sich die Architektur einem "echten" Shared-Disk-System an. Die aktuelle Hardwarekonfiguration mit dem verwendeten Fast-Ethernet, das außer durch Systemnachrichten auch noch durch NFS-Zugriffe belastet wird, und Betriebssystemcaches, die das Verhalten des Datenbankcaches negativ beeinflussen, stellen zusätzliche Probleme bei der Ein- und Ausgabe dar.

In dieser Arbeit wurden umfangreiche Leistungsanalysen des Systems vorgestellt, die den erfolgreichen und effizienten Übergang von TransBase zu MIDAS dokumentieren. TransBase besitzt bei nur einer aktiven Transaktion leichte Effizienzvorteile, die durch den in MIDAS zusätzlich betriebenen Mehraufwand für die Parallelisierung sowie die verwendeten komplexeren, verteilten Algorithmen entstehen. Bei steigendem Parallelitätsgrad kann MIDAS diesen Nachteil gegenüber TransBase aber schnell wieder ausgleichen, da es stark auf die Parallelverarbeitung ausgerichtet arbeitet.

Das System zeigt gute Skalierungseigenschaften. In einer Mehrrechnerkonfiguration kann bei einer Verdoppelung der Systemgröße zwischen 75 % und 95 % Skalierbarkeit unter Inter-Transaktionsparallelität erreicht werden. Auch Intra-Transaktionsparallelität wird gut unterstützt. Bei Messungen mit Intra-Transaktionsparallelität können sogar superlineare Speedups entstehen.

Weitere Analysen zu E/A-Granulaten und maximalen Parallelitätsgraden zeigten die Grenzen des Systems auf. Dabei wurde das beste Systemverhalten bei zwei bis drei parallelen Anfragen pro Rechner und bei einer Seitengröße von 32 KByte beobachtet.

Insgesamt bestätigen die vorgestellten Leistungsanalysen die erfolgreiche Implementierung von MIDAS.

7.1.2 Das FPT-Modell

Im zweiten Teil der Arbeit wurde das FPT-Modell vorgestellt, ein Modell zur Fehlerbehandlung in parallelen Transaktionen. Es basiert auf der Unterteilung von Transaktionen in Subtransaktionen und erlaubt nach dem Auftreten eines Fehlers die Kontrolle über einzelne, an einer Transaktion beteiligte Komponenten.

Transaktionsfehler in einer der Komponenten verursachen nicht mehr zwangsläufig den Abbruch der kompletten Transaktion. Vielmehr bietet das FPT-Modell die Möglichkeit, große, schon berechnete Teile der Transaktion nach einem Neustart wiederzuverwenden und somit deutlich an Effizienz zu gewinnen. Im Idealfall ist nur die den Fehler verursachende Komponente selbst zu reproduzieren, alle übrigen Komponenten können wiederverwendet werden.

Das FPT-Modell profitiert im wesentlichen von der Wiederverwendung von Zwischenergebnissen und der Implementierung kompensierender Kommunikationskanäle. Es kann ohne zusätzliche Maßnahmen während des normalen Ablaufs der Anfragen eingesetzt werden. Dadurch entsteht auch keine Beeinflussung der Laufzeit der Anfragen.

Um die Effizienz des FPT-Modells im Fehlerfall noch steigern zu können, werden ausgewählte Zwischenergebnisse zur Laufzeit vollständig gehalten. Dadurch entsteht zwar eine Beeinträchtigung der Laufzeit, aber es können beim Wiederaufsetzen einer Transaktion nach einem Fehler noch größere Teile der Transaktion wiederverwendet werden. Durch eine gute Auswahl der für die Wiederverwendung ausgewählten Stellen sollte der Normalfall, in dem kein Fehler auftritt, nur sehr wenig belastet sein.

Wichtig ist, daß das FPT-Modell nicht auf die Sicherung von Zwischenergebnissen angewiesen ist und damit nur nach Kostenaspekten entscheiden kann, in welchen Fällen eine solche Sicherung vorzunehmen ist.

Zusätzlich zu dem FPT-Modell wurde ein Algorithmus entworfen, der die vom Modell vorgegebenen Spezifikationen erfüllt und eine Anleitung gibt, wie die korrekte Fehlerbehandlung konkret durchzuführen ist.

Abschließend wurde gezeigt, wie das FPT-Modell auf ein System wie MIDAS abgebildet und dort integriert werden kann. Die Abbildung des FPT-Modells zur

Fehlerbehandlung auf das Ausführungsmodell von MIDAS läßt sich ohne Schwierigkeiten durchführen. Durch Analysen realer Anfragen wurde die Leistungsfähigkeit des FPT-Modells im Fehlerfall bestätigt.

Insgesamt zeigte sich, daß zur Fehlerbehandlung in einem parallelen Datenbanksystem wie MIDAS ein Modell wie das FPT-Modell gewinnbringend eingesetzt werden kann.

7.2 Ausblick

Eine zukünftige Weiterentwicklung des MIDAS-Systems ist wünschenswert. Aus den Ergebnissen dieser Arbeit zeigt sich, daß für die Gestaltung des Laufzeitsystems keine vollständig neuen Konzepte mehr entworfen werden müssen. Es stehen im wesentlichen Optimierungen an. So könnte zum einen das Laufzeitsystem über Threads realisiert werden, zum anderen muß die Kommunikation im System weiter reduziert und verbilligt werden. Ansätze dazu wurden mit der Architekturumstellung, dem On-Request-Invalidierungsverfahren und dem lokalitätsbasierten Transaktionsrouting schon vollzogen.

Indem NFS durch ein geeigneteres System ersetzt wird, sollten die Kohärenzprobleme beseitigt werden, die NFS im System erzeugt. Über eine geeignete Netzhardware und eine effizienter implementierte Kommunikation könnte zusätzlich dafür gesorgt werden, daß Cachetreffer in entfernten Caches wieder billiger als Festplattenzugriffe sind. Um einem "echten" Shared-Disk-System näherzukommen, müßten zusätzlich die Festplatten direkt an alle Rechner des Systems angeschlossen werden, um so ebenfalls das Netzwerk zu entlasten und einen gleichförmigen Zugriff aller Rechner auf die Festplatten zu ermöglichen.

Zur weiteren Analyse von MIDAS sollten Skalierungsmessungen mit größeren Systemkonfigurationen durchgeführt werden.

Für eine abschließende Bewertung des FPT-Modells wäre es wünschenswert, dieses in einer kompletten Implementierung zu realisieren. Beim anschließenden Einsatz des FPT-Modells wird die Frage zu klären sein, welche Zwischenergebnisse präventiv vollständig gehalten werden sollen. Hierzu muß ein Kostenmodell erstellt werden, an Hand dessen die Entscheidung getroffen werden kann, ob eine vorsorgliche Sicherung von Zwischenergebnissen vielversprechend ist, oder ob durch die Sicherung der normale Ablauf zu sehr beeinträchtigt wird.

Anhang A

Rechnerkonfigurationen

Folgende Hardwarekonfigurationen wurden bei den in dieser Arbeit präsentierten Leistungsmessungen verwendet:

Mehrprozessorsystem:
Sunbayer57

- SUN SPARCstation 20
- 4 Prozessoren SPARC 20, 100 MHz
- 128 MByte Arbeitsspeicher
- 4 lokale 4 GByte Fast-Wide-SCSI Festplatten für Datenbanken, eine lokale 2 GB Festplatte für Zwischenergebnisse und System

Verteiltes Mehrrechnersystem:
mit den vier identisch ausgestatteten Rechnern
Sunbayer51, Sunbayer60, Sunbayer61, Sunbayer62

- SUN Ultra 1
- 1 Prozessor Ultra1, 143 MHz
- 128 MByte Arbeitsspeicher
- eine lokale 4 GByte Festplatte für Datenbanken, eine lokale 2 GByte Festplatte für Zwischenergebnisse und System
- verbunden über 100 Mbit/s Fast-Ethernet-Switch

Anhang B

TPC Benchmarks

Die in dieser Arbeit durchgeführten Leistungsmessungen verwenden alle Anfragen und Datenbanken, die an den TPC-Benchmarks ausgerichtet sind [Gray 93].

B.1 TPC–A – TPC Benchmark A

Der TPC Benchmark A [Gray 93] ist ein reiner OLTP-Benchmark, mit einer sehr kurz laufenden Transaktion. Zur Erzeugung der Last wird bei TPC–A eine einfache, update-intensive Transaktion benutzt.

Der Benchmark simuliert eine hypothetische Bank[1] mit mehreren Zweigstellen. Jede Zweigstelle hat mehrere Bankschalter. Die Bank hat mehrere Kunden, jeder mit eigenem Konto. In der Datenbank werden die Kontostände jeder Einheit (Bank, Zweigstelle, Konto) und die abgewickelten Transaktionen gespeichert.

Die verwendete TPC–A-Transaktion führt die Aktionen durch, die bei Einzahlung oder Abhebung durch einen Kunden an einem Bankschalter notwendig sind.

Es handelt sich um eine einfache Transaktion, die den Kontostand des Kunden, der Zweistelle und der Bank verändert und diese Änderung über das Einfügen eines Tupels in einer weiteren Relation protokolliert. Die Auswahl des zu bearbeitenden Kontos ist zufällig. Folglich wird bei einer genügend großen Datenbank sehr häufig der Datenbankcache verfehlt. Dadurch werden E/A-Operationen notwendig.

Die TPC–A-Transaktion greift dabei jeweils auf ein zufälliges Tupel einer großen und zweier kleiner Relationen zu. Die beiden kleinen Relationen Bank und Zweig-

[1] Ursprünglich wurden die Parameter des *DebitCredit*-Benchmarks, aus dem TPC–A hervorging, so gewählt, daß sie in etwa den tatsächlichen Werten der kalifornischen Bank of America Anfang der 70er Jahre entsprachen.

stelle werden sich nach wenigen Transaktionen vollständig im Datenbankcache befinden. Die Konto-Relation account wurde bei den meisten Messungen mit 210 MByte (siehe Anhang C) so groß gewählt, daß sie weder in den Cache des Betriebssystems noch in den Datenbankcache paßt.

B.2 TPC–C – TPC Benchmark C

Der TPC Benchmark C unterscheidet sich vom TPC Benchmark A vor allem durch mehrere, teilweise komplexe Transaktionen und ein komplexeres Datenbankmodell.

In Bild B.1 wird die logische Struktur der Datenbank dargestellt. Es wird ein Versandhaus (company) simuliert, das mehrere Zweigstellen (warehouses) betreibt.

Jede Zweigstelle betreut zehn Geschäftsbezirke (districts) mit jeweils 3.000 Kunden. Das Versandhaus vertreibt 100.000 Artikel (items). Es wird davon ausgegangen, daß nicht alle Artikel immer auf Lager sind, deshalb werden 10 % der Bestellungen teilweise aus anderen Zweigstellen geliefert.

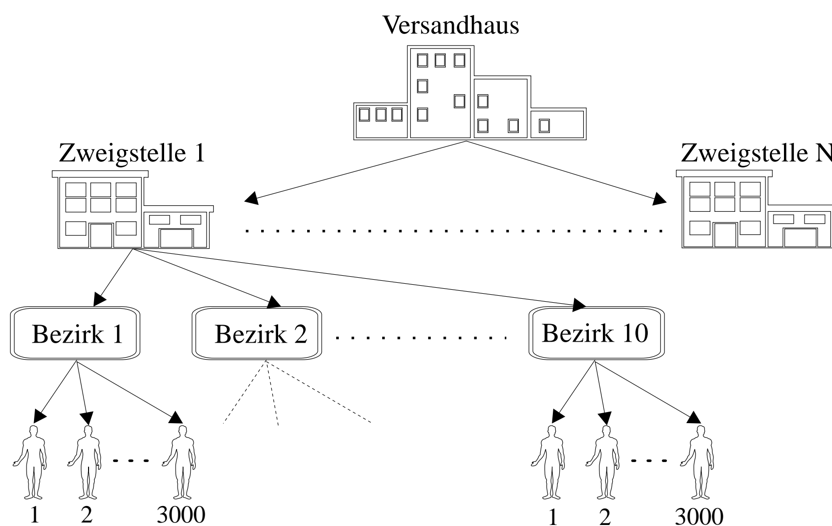

Abbildung B.1: Logische Struktur der TPC–C Datenbank

Der Angestellte kann im Bestellsystem des Versandhauses zwischen fünf vorgegebenen Transaktionen wählen, die Teile der Abwicklung einer Bestellung darstellen. Die Häufigkeit der Transaktionen ist prozentual vorgegeben.

Die fünf Transaktionen in der Häufigkeit ihres Auftretens sind:

1. **Die New-Order Transaktion**
   Hierbei wird eine Bestellung mit gegebenenfalls mehreren Positionen aufgenommen. Es handelt sich um eine Schreib- und Lese-Transaktion mittlerer Komplexität, die sehr häufig (ca. 44 %) ausgeführt wird.

2. **Die Payment Transaktion**
   Die Bezahlung einer Bestellung erfolgt durch Update des Kontostandes des Kunden, der Zweigstelle und des Geschäftsbezirks. Auch diese Schreib- und Lese-Transaktion wird sehr häufig ausgeführt (mindestens 44 %), sie ist jedoch von geringerer Komplexität.

3. **Die Order-Status Transaktion**
   Mit dieser Transaktion kann der Status der aktuellsten Bestellung eines Kunden abgefragt werden. Es werden nur lesende Anfragen verwendet. Diese Transaktion ist von mittlerer Komplexität und wird in mindestens 4 % der Fälle ausgewählt.

4. **Die Delivery Transaktion**
   Diese Transaktion vollzieht die Auslieferung von Bestellungen. Bearbeitet werden jeweils 10 Bestellungen innerhalb einer oder mehrerer Transaktionen. Es handelt sich um eine Schreib- und Lese-Transaktion mittlerer Komplexität, die selten (mindestens 4 %) ausgeführt wird.

5. **Die Stock-Level Transaktion**
   Damit wird die Anzahl von Artikeln, deren Bestand unter eine angegebene Schwelle gesunken ist, abgefragt. Die Stock-Level-Transaktion repräsentiert eine komplexe Anfrage, die selten (mindestens 4 %) ausgeführt wird. Im wesentlichen erfolgt hierbei ein sequentieller Relationen-Scan über eine sehr große Relation.

Mit diesen Transaktionen werden Transaktionsmixe gebildet, die in zufälliger Reihenfolge abgearbeitet werden. In Tabelle B.1 wird die Verteilung eines solchen Mixes mit 23 Transaktionen gezeigt.

Das Datenmodell beinhaltet 9 verschiedene Relationen: die Zweigstellen warehouses, die Verkaufsbezirke districts, die Kunden customers, die Produkte items, den Lagerbestand stock, die Kaufaufträge order (insgesamt drei verschiedene Relationen) und Information über früher getätigte Transaktionen history. Die Skalierung des Benchmarks erfolgt über die Veränderung der Anzahl der Zweigstellen. Pro Zweigstelle fallen etwa 120 MByte an Daten an.

Die warehouse- und district-Relation sind klein. Alle Transaktionstypen des Benchmarks greifen auf diese Relationen zu. Sie sind also die "hot spots" der Datenbank. Auf beiden Relationen wird sowohl lesend als auch schreibend zugegriffen.

Transaktion	Anzahl Transaktionen
New-Order	10
Payment	10
Order-Status	1
Delivery	1
Stock-Level	1

Tabelle B.1: Transaktionsmix TPC-Benchmark C

Im Gegensatz dazu sind die stock- und customer-Relationen groß (ca. 70 % der Datenbank). Daher ist die Wahrscheinlichkeit von Cachetreffern bei diesen Tabellen gering. Es gibt drei verschiedene order-Relationen und eine history-Relation. Ihre Größe wächst mit der Zeit, wenn neue Tupel sequentiell angehängt werden. Die item-Relation ist klein und von konstanter Größe (ca. 3 % der Datenbank). Auf sie wird nur lesend zugegriffen.

B.3 TPC–D – TPC Benchmark D

Der TPC–D Benchmark wurde eingerichtet, um die für Data Warehousing und Decision Support relevanten Fähigkeiten über eine Reihe von geschäftsorientierten Abfragen zu testen.

Es unterscheidet sich von früheren TPC Benchmarks, die vor allem die Performanz von Transaktionen kleinerer Kunden bewertete, durch umfangreiche und komplexe Anfragen auf sehr große Relationen. Das TPC–D-Modell besitzt 17 komplexe Anfragen.

Abbildung B.2 zeigt das Schema der TPC–D-Datenbank, mit dem Skalierungsfaktor SF.

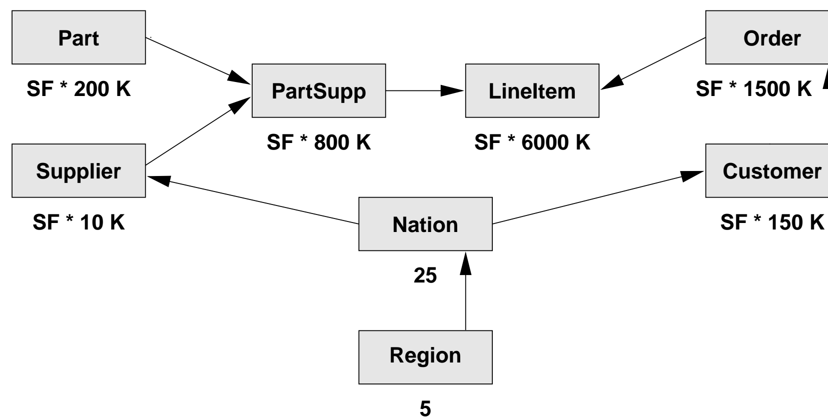

Abbildung B.2: Schema der TPC–D Datenbank

Anhang C

Datenbanken

Datenbank		Relation	Tupel	Größe
TPC–A	(1 tps)	branch	1	32 KByte
		teller	10	32 KByte
		account	100000	10 MByte
	gesamt			10 MByte

Datenbank		Relation	Tupel	Größe
TPC–A	(20 tps)	branch	20	32 KByte
		teller	200	64 KByte
		account	2000000	215 MByte
	gesamt			215 MByte

Tabelle C.1: TPC–A-Datenbanken

Datenbank		Relation	Tupel	Größe
TPC–C	(3 warehouses)	warehouse	3	32 KByte
		districts	30	32 KByte
		new-order	27000	1 MByte
		order	90000	8 MByte
		items	100000	14 MByte
		customers	90000	85 MByte
		order-line	900000	115 MByte
		stock	300000	140 MByte
	gesamt			365 MByte

Tabelle C.2: TPC–C-Datenbank

Datenbank	Relation	Tupel	Größe
TPC–D	region	5	65 KByte
	nation	25	65 KByte
	supplier	1000	65 KByte
	customer	15000	4 MByte
	part	20000	4 MByte
	partsupp	80000	13 MByte
	order	150000	23 MByte
	lineitem	600000	120 MByte
gesamt			168 MByte

Tabelle C.3: TPC–D-Datenbank

Anhang D

Transaktionen

Die in dieser Arbeit bei Leistungsmessungen zur Lastgenerierung verwendeten Transaktionen basieren auf den TPC-Spezifikationen [Gray 93].

Frühe Messungen wurden meist mit den TPC–A- und der TPC–C^{S+O}-Transaktionen durchgeführt, da die anderen Transaktionen erst später implementiert wurden.

Alle Versionen der TPC–C-Transaktionen sind nur lesend, da zum einen die Sperrverwaltung noch nicht lange einsatzbereit ist, und zum anderen die Stock-Level-Transaktion in einer schreibenden Version das gesamte System für Mehrbenutzerbetrieb blockieren würde. Im originalen Benchmark ist vorgesehen, diese Transaktionen auf einer niedrigen Konsistenzebene auszuführen, um solche Blockierungen zu vermeiden. Dies ist in MIDAS allerdings nicht möglich, da keine verschiedenen Konsistenzebenen implementiert sind.

Lesende Versionen von Transaktionen werden aus den Originaltransaktionen erzeugt, indem nur der SELECT-Anteil jeder schreibenden Anfrage ausgeführt wird.

Bei den durchgeführten Messungen wurden folgende Transaktionen verwendet:

- TPC–A:

  Die Transaktion des TPC–A-Benchmarks greift schreibend auf jeweils ein Tupel einer von zwei kleinen (ein bis zwei Seiten) und einer großen (siehe Anhang C) Relation zu. Sehr kurze Ausführungszeit.

- TPC–A^{RO}:

  Die nur lesende Version der TPC–A-Transaktion (read only).

- TPC–C:

  Ein Mix aus allen fünf Transaktionen des TPC–C-Benchmarks in lesender Version.

- TPC–C^{N+P}:

  Ein 50:50-Mix aus den Transaktionen New-Order und Payment des TPC–C-Benchmarks in lesender Version. Die OLTP-Version des TPC–C-Benchmarks mit kurzer Ausführungszeit.

- TPC–C^{S}:

  Nur die Stock-Level-Transaktion des TPC–C-Benchmarks. Eine sehr E/A-intensive Transaktion, die nur einen Relationen-Scan einer großen Relation durchführt. Ist nur für den Einbenutzerbetrieb bestimmt, da im Mehrbenutzerbetrieb eine Synchronisation zwischen den Transaktionen stattfinden kann, indem alle dieselbe Seite des Relationen-Scans lesen.

- TPC–C^{S+O}:

  Ein 20:80-Mix der nur lesenden Transaktionen Stock-Level und Order-Status des TPC–C-Benchmarks. Eine sehr E/A-intensiver Transaktionsmix. Dieser Mix besitzt die gleiche Datenintensität wie die TPC–C^{S}-Transaktion. 80 % der Ausführungszeit wird von der Stock-Level-Transaktion benötigt. Durch Einmischen von durchschnittlich vier Order-Status-Transaktionen zwischen zwei Stock-Level-Transaktionen wird der Synchronisationseffekt vermieden, der bei TPC–C^{S}-Transaktion auftritt.

- TPC–D Anfrage 9:

  Eine komplexe Anfrage mit 6-Wege-Join ohne Beschränkung der beteiligten Relationen (außer part), Gruppierung und Sortierung. Alle Tabellen außer region und customer sind einbezogen.

Anhang E

Abkürzungsverzeichnis

DB	Datenbank
DBMS	Datenbank Management System
DBS	Datenbanksystem
DDL	Data Definition Language
DML	Data Modification Language
E/A	Ein- und Ausgabe
FPT-Modell	Modell zur Fehlerbehandlung in parallelen Transaktionen
LRU	Least Recently Used
MIDAS	Munich Parallel Database System
MMK	Multiprozessor Multitasking Kernel
MPI	Message-Passing Interface
NFS	Network File System
OLTP	On Line Transaction Processing
PDBMS	Paralleles Datenbank Management System
PQEP	Parallel Query Execution Plan
PVM	Parallel Virtual Machine
QEP	Query Execution Plan
RAID	Redundant Array of Inexpensive Disks
SD	Shared-Disk
SN	Shared-Nothing
SQL	Structured Query Language
TA	Transaktion

TB	TransBase
TBX	TransBase-Exchange
TPC	Transaction Processing Council
TPC–A	TPC Benchmark™ A
TPC–C	TPC Benchmark™ C
TPC–D	TPC Benchmark™ D
VDBC	Virtueller Datenbank Cache

Anhang F

Begriffsfestlegungen und Definitionen

benachbart, 133

Datenflüsse, 133

GLM-Request, 76

kompensierende Datenflüsse, 148

Load-Page, 77

Partitionierung
  entfernte, 71
  lokale, 71

Replikation, lokale, 71

Store-Page, 77
Subtransaktionen, 133

Verteilung
  entfernte, 71
  lokale, 71

Literaturverzeichnis

[Bayer 95] BAYER, R.: *The Digital Library System OMNIS/Myriad.* 18th Australasian Computer Science Conference, Adelaide, Australien, Februar 1995.

[BemLud 90] BEMMERL, T. UND LUDWIG, T.: *MMK — A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing.* CONPAR 90 – VAPP Conference, Zürich, Schweiz, 1990.

[BerNew 97] BERNSTEIN, P. UND NEWCOMER, E.: *Principles of Transaction Processing.* Morgan Kaufmann Publishers, 1997.

[BJLMRZ 96a] BOZAS, G., JAEDICKE, M., LISTL, A., MITSCHANG, B., REISER, A. UND ZIMMERMANN, S.: *On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS-Project.* In Proceedings of 2nd International Euro-Par Conference, Parallel Processing, Seiten 881–886, Lyon, Frankreich, September 1996. Spinger-Verlag, LNCS 1123, Berlin.

[BJLMRZ 96b] BOZAS, G., JAEDICKE, M., LISTL, A., MITSCHANG, B., REISER, A. UND ZIMMERMANN, S.: *On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS-Project.* SFB-Bericht 342/14/96 A, TUM-I 9625, 19 Seiten, Institut für Informatik, Technische Universität München, 1996.

[BoLeLiPaRe 94] BOZAS, G., LEHN, R., LISTL, A., PAWLOWSKI, M. UND REISER, A.: *Using PVM to implement a Parallel Database System.* In Proceedings of the 1st European PVM User Group Meeting, Rom, Italien, Oktober 1994.

[BonSar 95] Bontempo, C. J. und Saracco, C. M.: *Database Management — Principles and Products.*
Prentice Hall PTR, 1995.

[Bozas 98] Bozas, G.: *Scalability in Parallel Database Systems.*
Dissertation, Fakultät für Informatik, Technische Universität München, 1998.

[BozFleZim 97] Bozas, G., Fleischhauer, M. und Zimmermann, S.: *PVM Experiences in Developing the MIDAS Parallel Database System.*
In Proceedings of the 4th European PVM User Group Meeting, Krakau, Polen, November 1997.

[BozLis 95] Bozas, G. und Listl, A.: *Performance Gains Using Subpages for Cache Coherency Control.*
SFB-Bericht 342/21/95 A, TUM-I 9538, Institut für Informatik, Technische Universität München, 1995.

[BozLis 97] Bozas, G. und Listl, A.: *Performance Gains Using Subpages for Cache Coherency Control.*
In Proceedings of the 8th International Conference and Workshop on Database and Expert Systems Applications, DEXA, Toulouse, Frankreich, September 1997.

[Brandmayer 97] Brandmayer, F.: *Parallele Anfrageausführung in MIDAS.*
Diplomarbeit, Institut für Informatik, Technische Universität München, Februar 1997.

[CheDay 97] Chen, Q. und Dayal, U.: *Failure Handling for Transaction Hierarchies.*
3th *International Conference on Data Engineering, ICDE 97*, Seiten 245–254, 1997.

[CJMNRZ 97] Clausnitzer, A., Jaedicke, M., Mitschang, B., Nippl, C., Reiser, A. und Zimmermann, S.: *On the Application of Parallel Database Technology for Large Scale Document Management Systems.*
In International Database Engineering and Applications Symposium, Montreal, Kanada, August 1997.

[ClaVogWie 95] A. Clausnitzer, P. Vogel und S. Wiesener: *WWW interface to the OMNIS/Myriad literature retrieval engine.* In Proceedings of the 3rd International World-Wide Web Conference, Technology, Tools and Applications, Darmstadt (s. http://omnis.informatik.tu-muenchen.de), Elsevier North-Holland, 1995.

[Dallmeyr 95] Dallmeyr, L.: *Implementierung des Segment-Layers für ein paralleles Datenbanksystem auf einem Netz von Workstations.* Diplomarbeit, Institut für Informatik, Technische Universität München, 1995.

[DewGra 92] DeWitt, D. und Gray, J.: *Parallel Database Systems: The Future of High Performance Database Systems.* In Communications of the ACM, Vol. 35, No. 6, Juni 1992.

[Drügh 99] Drügh, R.: *Verwendung von Zwischenergebnissen im parallelen Datenbanksystem MIDAS.* Diplomarbeit, Institut für Informatik, Technische Universität München, Juli 1999.

[ElKiLeSe 87] Elhardt, K., Killer, D., Lehnert, K. und Seibt, C.: *The Database System MERKUR.* Technischer Bericht TUM-I 8702, Institut für Informatik, Technische Universität München, 1987.

[Elmagarmid 92] Elmagarmid, A.: *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, San Mateo, California, USA, 1992.

[Fehr 97] Fehr, M.: *Implementierung einer graphischen Benutzeroberfläche für das Simulationssytem DBSIM.* Diplomarbeit, Institut für Informatik, Technische Universität München, August 1996.

[Fleischhauer 94] Fleischhauer, M.: *Implementierung der internen Operatorbaum-Schnittstelle für das parallele Datenbanksystem MIDAS.* Fortgeschrittenenpraktikum, Institut für Informatik, Technische Universität München, 1994.

[Fleischhauer 97] Fleischhauer, M.: *Parallelisierung relationaler Datenbankanfragen in MIDAS.* Diplomarbeit, Institut für Informatik, Technische Universität München, Mai 1997.

[GBDJMS 94a] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. und Sunderam, V.: *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Network Parallel Computing.*
The MIT Press, 1994.

[GBDJMS 94b] Geist, A., Beguelin, A, Dongarra, J., Jiang, W., Manchek, R. und Sunderam, V.: *PVM 3 User's Guide and Reference Manual.*
Technical Report ORNL/TM12187, Oak Ridge National Laboratory, September 1994.

[GeiKohPap 96] Geist, A., Kohl, J. und Papadopoulos, P.: *PVM and MPI: a Comparison of Features.*
Calculateurs Paralleles Vol. 8 No. 2, 1996.

[Gesmann 97] Gesmann, M.: *Parallele Anfrageverarbeitung in Komplexobjekt-Datenbanksystemen — Verarbeitungskonzepte, Realisierungsaspekte und Betriebssystemeinbettung.*
Dissertation, Universität Kaiserslautern, Fachbereich Informatik, AG Datenverwaltungssyteme, 1997.

[Gouvedaris 96] Gouvedaris, A.: *Messung der Leistungsfähigkeit von E/A-Systemen und Netzwerken in einer Ethernet, ATM und SP-2 Umgebung.*
Diplomarbeit, Institut für Informatik, Technische Universität München, November 1996.

[Graefe 90] Graefe, G.: *Encapsulation of Parallelism in the Volcano Query Processing System.*
In Proceedings of the ACM SIGMOD International Conference on Management of Data, Seiten 102–111, 1990.

[Graefe 95] Graefe, G.: *The Cascades Framework for Query Optimization.*
In Data Engineering Bulletin, 18(3), Seiten 19–29, 1995.

[GraReu 93] Gray, J. und Reuter, A.: *Transaction Processing: Concepts and Techniques.*
Morgan Kaufmann Publishers, 1993.

[Gray 93] Gray, J.: *The Benchmark Handbook for Database and Transaktion Processing Systems..*
2nd edition, Morgan Kaufmann Publishers, Inc., San Mateo, California, USA, 1993.

[Gray 95] GRAY, J.: *A Survey of Parallel Database Techniques and Systems.*
In Tutorial Handout at the 21st International Conference on Very Large Databases, VLDB, 1995.

[Haas 98] HAAS, S.: *Erweiterung des SQL-Parsers von MIDAS um benutzerdefinierte Funktionen.*
Fortgeschrittenenpraktikum, Institut für Informatik, Technische Universität München, 1998.

[HärRah 99] HÄRDER, T. UND RAHM, E.: *Datenbanksysteme: Konzepte und Techniken der Implementierung.*
Springer-Verlag, 1999.

[HärReu 83] HÄRDER, T. UND REUTER, A.: *Principles of Transaction-Oriented Database Recovery.*
ACM Computing Surveys, 15(4): Seiten 287–317, Dezember 1993.

[HärProSch 90] HÄRDER, T., PROFIT, M. UND SCHÖNING, H.: *Supporting Parallelism in Engineering Databases by Nested Transactions.*
Universität Kaiserslautern, 1990.

[HärRot 87] HÄRDER, T. UND ROTHERMEL, K.: *Concepts for Transaction Recovery in nested Transactions.*
In Proceedings of the ACM SIGMOD International Conference on Management of Data, Seiten 239–247, San Francisco, USA, 1987.

[HärRot 93] HÄRDER, T. UND ROTHERMEL, K.: *Concurrency Control Issues in Nested Transactions.*
VLDB Journal, 2(1): Seiten 39–74, 1993.

[HasWei 93] HASSE, C. UND WEIKUM, G.: *Inter- and Intra-Transaction Parallelism in Database Systems.*
In Proceedings of the 14th Workshop “Applications on Massively Parallel Systems”, Zürich, Schweiz, September 1993.

[Heupel 98] HEUPEL, S.: *Implementierung erweiterbarer Datenbankoperatoren in einem parallelen, objekt-relationalen Datenbanksystem.*
Diplomarbeit, Institut für Informatik, Technische Universität München, Oktober 1998.

[Hilbig 98] HILBIG, M.: *Entwicklung eines kostenbasierten Anfrageoptimiers für das parallele, relationale Datenbanksystem MIDAS.* Diplomarbeit, Institut für Informatik, Technische Universität München, Oktober 1998.

[Humm 93] HUMM, B. G.: *An Extended Scheduling Mechanism for Nested Transactions.* In Proceedings of the 1993 International Workshop on Object-Orientation in Operating Systems, IWOOOS'93, Asheville, North Carolina, USA, 1993.

[Intel 89] INTEL CORPORATION: *iPSC/2 User's Guide: Intel Scientific Super-Computer Division.* Intel Corporation, 1989.

[Jaedicke 99] JAEDICKE, M.: *Fortschrittliche Konzepte zur Anfrageverarbeitung.* Dissertation, Fakultät für Informatik, Universität Stuttgart, August 1999.

[Krü-Bar 98] KRÜGER-BARVELS, K.: *Entwicklung eines regelbasierten Anfrageoptimierers für das parallele objektrelationale Datenbanksystem MIDAS.* Diplomarbeit, Institut für Informatik, Technische Universität München, Oktober 1998.

[Lamberts 97] LAMBERTS, S.: *Parallele verteilte Dateisysteme in Rechnernetzen.* Dissertation, Technische Universität München, Januar 1997.

[LiPaReBoLe 95] LISTL, A., PAWLOWSKI, M., REISER, A., BOZAS, G. UND LEHN, R.: *Architektur des parallelen Datenbanksystems MIDAS.* In Datenbanksysteme in Büro, Technik und Wissenschaft (BTW), 1995.

[LisPaw 92] LISTL, A. UND PAWLOWSKI, M.: *Parallel Cache Management of a RDBMS.* SFB-Bericht 342/18/92 A, Institut für Informatik, Technische Universität München, August 1992.

[LisSchFri 94] LISTL, A., SCHNEKENBURGER, T. UND FRIEDRICH, M.: *Zum Entwurf eines Prototypen für MIDAS.* SFB-Bericht 342/01/94 B, Institut für Informatik, Technische Universität München, April 1994.

[Listl 94a] Listl, A.: *Using Subpages for Cache Coherency Control in Parallel Database Systems.*
SFB-Bericht 342/06/94 A, Institut für Informatik, Technische Universität München, 1994.

[Listl 94b] Listl, A.: *Using Subpages for Cache Coherency Control in Parallel Database Systems.*
In Proceedings of PARLE'94 Parallel Architectures and Languages Europe, Athen, Griechenland, Juli 1994.

[Listl 96] Listl, A.: *Effiziente Pufferverwaltung in parallelen relationalen Datenbanksystemen.*
DISDBIS Dissertation, Technische Universität München, Infix-Verlag, Sankt Augustin, 1996.

[LoiObePaw 91] Loibl, E., Obermaier, H. und Pawlowski, M.: *Towards Parallelism in a Relational Database System.*
SFB-Bericht 342/10/91 A, Institut für Informatik, Technische Universität München, April 1994.

[Mariucci 97] Mariucci, M.: *Implementierung eines verteilten, hierarchischen Lockmanagers in MIDAS.*
Systementwicklungsprojekt, Institut für Informatik, Technische Universität München, 1997.

[Menzel 91] Menzel, D.: *Paralleles externes Sortieren auf Multiprozessoranlagen.*
Diplomarbeit, Institut für Informatik, Technische Universität München, Dezember 1991.

[Mitschang 95] Mitschang, B.: *Anfrageverarbeitung in Datenbanksystemen.*
Vieweg, Braunschweig, 1995.

[Moss 81] Moss, J. E. B.: *Nested Transactions: An Approach to Reliable Distributed Computing.*
PhD Thesis, Department of Electrical Engineering and Computer Science, MIT, 1981.

[Moss 85] Moss, J. E. B.: *Nested Transactions — An Approach to Reliable Distributed Computing.*
The MIT Press, 1985.

[NipMit 98a] Nippl, C. und Mitschang, B.: *TOPAZ: a Cost-Based, Rule-Driven, Multi-Phase Parallelizer.*
In Proceedings of the 24th International Conference on Very Large Databases, VLDB, New York City, USA, 1998.

[NipMit 98b] NIPPL, C. UND MITSCHANG, B.: *Towards Deadlock-Preventing Query Optimization and Parallelization.*
In Proceedings of the International Conference on Parallel and Distributed Computing Systems, Chicago, Illinois, USA, 1998.

[Nippl 00] NIPPL, C.: *Providing efficient, extensible and adaptive intra-query parallelism for advanced applications.*
Dissertation, Fakultät für Informatik, Technische Universität München, 2000.

[NipZimMit 99] NIPPL, C., ZIMMERMANN, S. UND MITSCHANG, B.: *Design, Implementation and Evaluation of Data Rivers for Efficient Intra-Query Parallelism.*
SFB-Bericht 342/08/99 A, TUM-I 9918, Institut für Informatik, Technische Universität München, November 1999.

[ÖzsVal 91] ÖZSU, M. T. UND VALDURIEZ, P.: *Principles of Distributed Database Systems.*
Prentice Hall, 1991.

[PapPapTsa 96] PAPAKOSTAS, N., PAPAKONSTANTINOU, G. UND TSANAKAS, P.: *Using PVM to Implement PPARDB/PVM, a portable Parallel Database Management System.*
In Proceedings of the 3rd European PVM Conference, Spinger-Verlag, LNCS 1156, Seiten 108–115, Oktober 1996.

[Perathoner 98a] PERATHONER, S.: *Erweiterung des System-Katalogs von MIDAS um Statistiken und objekt-relationale Funktionalität.*
Fortgeschrittenenpraktikum, Institut für Informatik, Technische Universität München, 1998.

[Perathoner 98b] PERATHONER, S.: *Entwicklung einer Komponente zur Lastverteilung für das Parallele Datenbanksystem MIDAS.*
Diplomarbeit, Institut für Informatik, Technische Universität München, November 1998.

[Pries 97] PRIES, K.: *Leistungsmessungen in MIDAS.*
Diplomarbeit, Institut für Informatik, Technische Universität München, Februar 1997.

[Rahm 91] RAHM, E.: *Concurrency and Coherency Control in Database Sharing Systems.*
Bericht 3/91, Universität Kaiserslautern, Department of Computer Science, Dezember 1991.

[Rahm 94] Rahm, E.: *Mehrrechner-Datenbanksysteme — Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Addison-Wesley, 1994.

[RezHär 95] Rezende, F. F. und Härder, T.: *Concurrency Control in Nested Transactions with Enhanced Lock Modes for KBMSs.* In Proceedings of the International Conference and Workshop on Database and Expert Systems Applications, DEXA'95, London, GB, 1995.

[RysNorSch] Rys, M., Norrie, M. C. und Scheck, H.-J.: *Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System.* In Proceedings of the 22nd International Conference on Very Large Databases, VLDB, Mumbai (Bombai), Indien, 1996.

[Schöning 90] Schöning, H.: *Preserving Consistency in Nested Transactions.* In Proceedings of the 23rd International Conference on System Sciences, in HICSS-23, Vol. II, Seiten 472–480, Januar 1990.

[SchSchWei 95] Schaad, W., Schek, H.-J. und Weikum, G.: *Implementation and Performance of Multi-level Transaction Management in Multidatabase Environment.* In Proceedings of the 5th International Workshop on Research Issues on Data Engineering: Distributed Object Management, RIDE-DOM'95, Taipei, Taiwan, März 1995.

[Schult 97] Schult, M.: *Portierung von OMNIS auf das parallele Datenbanksystem MIDAS.* Diplomarbeit, Institut für Informatik, Technische Universität München, Mai 1997.

[SchWeiSch 91] Schek, H.-J., Weikum, G. und Schaad, W.: *A Multi-Level Transaction Approach to Federated DBMS Transaction Management.* In Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems, IMS'91, Kyoto, Japan, April 1991.

[SpeZimCla 97] Specht, G., Zimmermann, S. und Clausnitzer, A.: *Introducing Parallelism in Multimedia Database Systems.* In Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architecture Synthesis, Seiten 348–355, Aizu-Wakamatsu, Fukushima, Japan, März 1997 IEEE Computer Society Press, Los Alamitos, California, USA.

[Stonebraker 94] STONEBRAKER, M.: *Readings in Database Systems.* 2nd Edition, Morgan Kaufmann Publishers, 1994.

[TransBaseP 89] TRANSACTION SOFTWARE GMBH: *TransBase Relational Database System, Version 3.3, Programming Interface TBX.* TransAction Software GmbH, 1989.

[TransBaseR 88] TRANSACTION SOFTWARE GMBH: *TransBase Relational Database System, Version 3.3, TB/SQL Reference Manual.* TransAction Software GmbH, 1988.

[TransBaseS 88] TRANSACTION SOFTWARE GMBH: *TransBase Relational Database System, Version 3.3, System Guide.* TransAction Software GmbH, 1988.

[Tschaffler 98] TSCHAFFLER, M.: *Portierung und Erweiterung eines Visualisierungstools für das parallele Datenbanksystem MIDAS.* Diplomarbeit, Institut für Informatik, Technische Universität München, Februar 1998.

[Usner 94] USNER, M.: *Kommunikation durch Segmente in parallelen Datenbanksystemen.* Diplomarbeit, Institut für Informatik, Technische Universität München, 1994.

[Voinou 96] VOINOU, A.: *Untersuchungen für ein Transaktionskonzept in MIDAS.* Diplomarbeit, Institut für Informatik, Technische Universität München, November 1996.

[Vossen 93] VOSSEN, G., GROSS-HARDT, M.: *Grundlagen der Transaktionsverarbeitung.* Addison-Wesley, 1993.

[WeiHas 91] WEIKUM, G. UND HASSE, C.: *Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism.* Technical Report 162, ETH Zürich, Institute of Theoretical Computer Science, Juli 1991.

[Zim 97] ZIMMERMANN, S.: *Transaktionen in parallelen Datenbanksystemen.* Bericht, 8. Workshop "Transaktionskonzepte", GI Datenbankrundbrief, Ausgabe 19, 1997.