Erhard Rahm

Mehrrechner-Datenbank-systeme

Grundlagen der verteilten und parallelen Datenbankverarbeitung

Rahm, Erhard:

Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung / Erhard Rahm. -

HINWEISE: Das Buch wurde 1994 vom Addison-Wesley-Verlag unter der

ISBN 3-89319-702-8

publiziert. Seit 1997 erfolgte der Vetrieb des unveränderten Buchs durch den Oldenbourg-Verlag unter der

ISBN 978-3486243635.

Die gedruckte Auflage ist vergriffen. Seit 2004 liegen wieder sämtliche Urheberrechte beim Autor.

Die vorliegende elektronische Version entspricht der 1994 gedruckten Buchversion, jedoch sind die meisten Abbildungen jetzt farbig dargestellt. Der Seitenumbruch kann u.a. aufgrund geänderter Schriftarten von der Originalversion abweichen. Am Inhalt wurden keine Änderungen vorgenommen. An wenigen Stellen wurden jedoch kleinere Fehler behoben. Die entsprechenden Textstellen sind in blauer Schrift gehalten.

1. Auflage 1994

Satz: Erhard Rahm, Leipzig.

Umschlaggestaltung: Hommer DesignProduction, München

Vorwort

Traditionellerweise erfolgt die Datenbankverarbeitung zentralisiert, wobei die Datenbank an einem Rechner gehalten und verwaltet wird. Zudem arbeiten üblicherweise Anwendungen und Transaktionen nur mit einer Datenbank. Dieser einfache Ansatz verliert jedoch zunehmend an Bedeutung, da viele Anwendungsbereiche eine verteilte und parallele Datenbankverarbeitung verlangen. Treibende Kräfte hierfür sind Forderungen nach hoher Leistungsfähigkeit und Verfügbarkeit, Verbesserung der Kosteneffektivität sowie Unterstützung dezentraler Organisationsstrukturen. Weiterhin muß Anwendungen häufig der integrierte Zugriff auf mehrere unabhängige und heterogene Datenbanken ermöglicht werden. Zur Erfüllung dieser Anforderungen eignen sich verschiedene Klassen sogenannter *Mehrrechner-Datenbanksysteme*, bei denen Datenbanksysteme (DBS) auf mehreren Rechnern kooperieren. Diese Systeme lassen sich grob unterteilen in Verteilte Datenbanksysteme, heterogene bzw. föderative Datenbanksysteme, Parallele Datenbanksysteme und Workstation/Server-DBS, die jeweils bestimmten Anforderungen Rechnung tragen.

Das vorliegende Buch behandelt erstmalig diese unterschiedlichen Ansätze im Rahmen eines Lehrbuchs. Es basiert auf mehreren an der Universität Kaiserslautern gehaltenen Vorlesungen sowie auf langjähriger Erfahrung und Forschungsarbeit auf dem Gebiet der Mehrrechner-Datenbanksysteme. Es ist gleichermaßen als Begleitbuch zu Informatik-Vorlesungen im Hauptstudium als auch zum Selbststudium geeignet. Zum besseren Verständnis werden zahlreiche Abbildungen und Beispiele verwendet. Zudem bieten Übungsaufgaben die Gelegenheit, den Stoff zu vertiefen, wobei die vollständige Angabe der Aufgabenlösungen eine Selbstkontrolle ermöglicht.

Ausgangspunkt der Darstellung ist eine umfassende Klassifikation von Mehrrechner-Datenbanksystemen, die eine Definition und Abgrenzung der einzelnen Realisierungsalternativen gestattet. Die Beschreibung der einzelnen Systemklassen legt danach den Schwerpunkt darauf, grundlegende Konzepte zu vermitteln und die wichtigsten Implementierungsansätze aufzuzeigen. Dabei wird auch versucht, eine Brücke zwischen Praxis und Forschung zu schlagen, indem die wichtigsten

Ansätze aus beiden Bereichen Berücksichtigung finden. So werden etwa neuere Forschungsergebnisse aus den Gebieten der föderativen und Parallelen Datenbanksysteme präsentiert. Der hohe Praxisbezug zeigt sich u.a. an der Behandlung von Verteilten Transaktionssystemen*, wichtiger Standardisierungsansätze zur Transaktions- und Datenbankverarbeitung in heterogenen Systemen (X/Open DTP, RDA, ODBC, DRDA etc.) sowie von in existierenden Mehrrechner-DBS eingesetzten Implementierungstechniken. Daneben wird ein Überblick zu zahlreichen kommerziell verfügbaren Mehrrechner-Datenbanksystemen gegeben, darunter aktuelle Entwicklungen wie IBM Parallel Sysplex, DB2 Parallel Query Server und Sybase Navigational Server.

Das Buch gliedert sich in sieben Teile und 21 Kapitel. Der einleitende Teil I faßt nach einer Einführung Grundlagen von zentralisierten Datenbanksystemen (DBS) sowie Rechnernetzen zusammen, die für das weitere Verständnis des Buches hilfreich sind. In Kapitel 3 folgt dann eine Klassifikation von Mehrrechner-Datenbanksystemen sowie ein qualitativer Vergleich zwischen den wichtigsten Architekturtypen. Verteilte Datenbanksysteme stellen demnach geographisch verteilte, homogene Mehrrechner-DBS dar. Parallele Datenbanksysteme sind ebenfalls homogen, basieren jedoch auf einer lokalen Verteilung der Prozessoren (z.B. innerhalb eines Parallelrechners). Zur Realisierung Paralleler DBS bestehen neben Multiprozessor-DBS ("Shared Everything") vor allem die Alternativen *"Shared Disk"* und *"Shared Nothing"*. Föderative DBS eignen sich zur Unterstützung unabhängiger, heterogener Datenbanken, während Workstation/Server-DBS v.a. für nicht-konventionelle Datenbankanwendungen im Einsatz sind.

Teil II behandelt Verteilte Datenbanksysteme. Dazu werden zunächst (Kapitel 4) die Schemaarchitektur sowie Alternativen zur Katalog- und Namensverwaltung diskutiert. Kapitel 5 befaßt sich mit dem fundamentalen Problem der Datenverteilung, welches die Teilaufgaben der Fragmentierung und Allokation umfaßt. Danach werden die wichtigsten Ansätze zur verteilten Anfragebearbeitung (Kap. 6), Commit-Behandlung (Kap. 7), Synchronisation (Kap. 8) sowie zur Wartung replizierter Datenbanken (Kap. 9) beschrieben.

Teil III widmet sich heterogenen Datenbanken, die von unabhängigen DBS verwaltet werden. Hierzu werden die beiden wesentlichen Realisierungsalternativen, Verteilte Transaktionssysteme (Kap. 11) sowie föderative DBS (Kap. 12), behandelt. Ferner werden wichtige Standards zur Unterstützung von Interoperabilität und Portabilität erörtert.

---

* Verteilte Transaktionssysteme verkörpern eine weit verbreitete Alternative zu Mehrrechner-Datenbanksystemen zur verteilten Datenbankverarbeitung.

Parallele Datenbanksysteme sind Gegenstand von Teil IV und Teil V. In Teil IV wird der in der Praxis bedeutsame Shared-Disk-Ansatz vorgestellt, bei dem alle Rechner Zugriff auf alle Platten haben, so daß eine Partitionierung der Datenbank entfällt. Dafür stellen sich v.a. die Probleme der Synchronisation und Kohärenzkontrolle, auf die in Kap. 14 und Kap. 15 ausführlich eingegangen wird. Im Mittelpunkt von Teil V steht die Realisierung von Intra-Transaktionsparallelität, mit der für komplexe und datenintensive Transaktionen bzw. Anfragen kurze Antwortzeiten erreicht werden sollen. Hierzu werden eine geeignete Datenverteilung (Kap. 17), v.a. in Shared-Nothing-Systemen, sowie parallele Algorithmen zur Anfrageverarbeitung (Kap. 18) benötigt.

In Teil VI werden kommerziell verfügbare Mehrrechner-DBS von 12 Herstellern überblicksartig vorgestellt. Teil VII (Anhang) enthält eine vollständige Zusammenstellung der Aufgabenlösungen sowie Literaturverzeichnis und Index.

Mehrere Personen haben wesentliche Teile des Buchmanuskripts gelesen und zahlreiche Korrektur- und Verbesserungsvorschläge gemacht. Zu Dank verpflichtet bin ich hierfür u.a. Herrn Prof. Dr. T. Härder, Herrn M. Jaedicke sowie meinen Mitarbeitern Dr. D. Sosna und T. Stöhr. Zahlreiche Herstellerfirmen von Datenbanksystemen stellten mir bereitwillig Unterlagen für ihre Produkte zur Verfügung. Besonders hilfreich waren hierbei Mitarbeiter der Firmen IBM (insbesondere Frau I. Stelkens sowie die Herren Dr. K. Küspert, H. Baisch und E. Hechler) und Tandem. Die Fa. IBM informierte mich bereits vor der offiziellen Ankündigung über ihre neuen Produkte, im Rahmen eines eigens veranstalteten Seminars. Dem Vieweg-Verlag danke ich für die Einwilligung, einige Abbildungen aus meinem Buch "Hochleistungs-Transaktionssysteme" [Ra93a] verwenden zu können.

Für Hinweise auf Fehler, Unklarheiten und Verbesserungsmöglichkeiten bin ich sehr dankbar. Sie können an die unten angegebene Adresse bzw. an den elektronischen Briefkasten *mrdbs@informatik.uni-leipzig.de* gerichtet werden.

Leipzig, im August 1994 Erhard Rahm

Institut für Informatik
Universität Leipzig
Augustusplatz 10-11
04109 Leipzig

Inhaltsverzeichnis

Teil II: : Verteilte Datenbanksysteme

Teil III: Heterogene Datenbanken

Teil IV: Shared-Disk-Datenbanksysteme

Teil V: Parallele DB-Verarbeitung

Teil VI: Systemüberblicke

Teil VII: Anhang

Teil I Grundlagen

Der einleitende Teil des Buches umfaßt drei Kapitel. Zunächst wird diskutiert, warum die Datenverarbeitung zunehmend in verteilten Computer-Systemen stattfindet und welche Anforderungen von Mehrrechner-Datenbanksystemen zu erfüllen sind. Kapitel 2 faßt Grundlagen von zentralisierten Datenbanksystemen sowie bezüglich Rechnernetzen zusammen, die für das weitere Verständnis erforderlich sind. In Kapitel 3 werden eine Klassifikation von Mehrrechner-Datenbanksystemen vorgenommen sowie ein qualitativer Vergleich zwischen den wichtigsten Architekturtypen vorgestellt.

1 Einführung

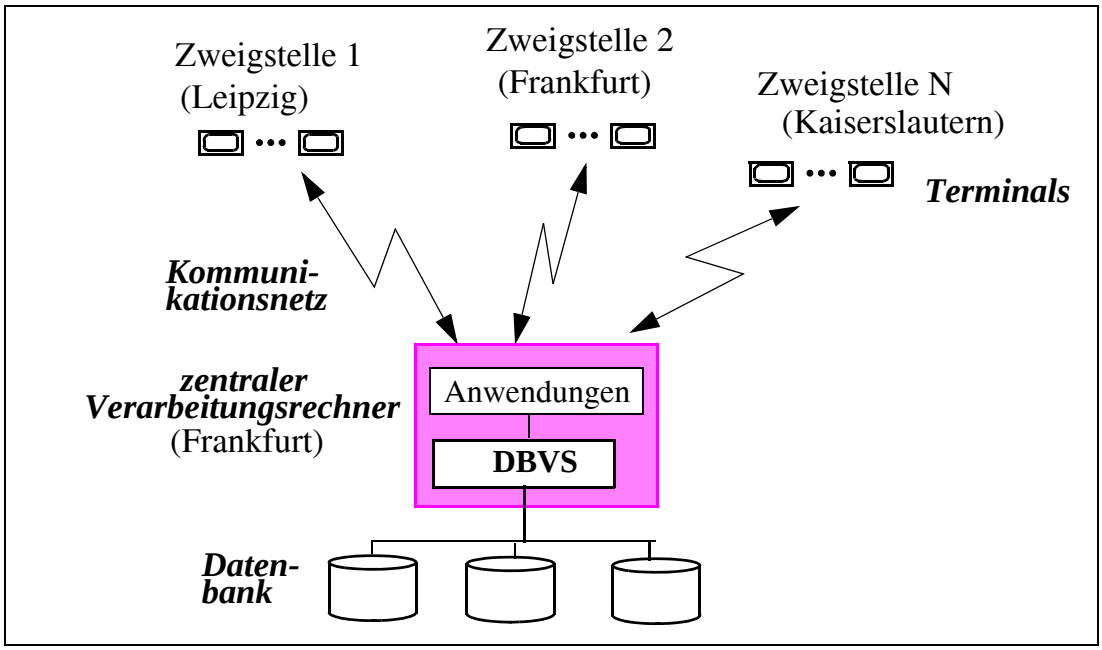
Die Daten eines Unternehmens, einer Institution o.ä. werden üblicherweise innerhalb von Datenbanksystemen (DBS) verwaltet. Das Datenbanksystem besteht dabei aus der eigentlichen Datenbank (DB) sowie dem Datenbankverwaltungssystem (DBVS) bzw. Datenbank-Management-System (DBMS). Die Datenbank enthält die relevanten Daten der jeweiligen Anwendung und ist auf Externspeichern (i.a. Magnetplatten) permanent gespeichert. Das DBVS verwaltet diese Daten und führt sämtliche Zugriffe darauf aus. Das Datenbanksystem bietet gegenüber Anwendungen einen hohen Grad an Datenunabhängigkeit, indem es interne Aspekte der Datenorganisation und -speicherung vollkommen verbirgt, so daß Änderungen hierbei (z.B. Definition einer neuen Indexstruktur) ohne Auswirkungen auf Anwendungsprogramme bleiben. Dies wird durch ein mächtiges Datenmodell (z.B. relationales Datenmodell) erreicht, mit dem der Datenbankaufbau auf logischer Ebene innerhalb eines sogenannten Datenbankschemas beschrieben wird. Für den Zugriff auf die Datenbank werden in der Regel deskriptive und ausdrucksstarke Anfragesprachen wie SQL benutzt.
Der Datenbankzugriff erfolgt innerhalb von Transaktionen, welche aus einer oder mehreren DB-Operationen der jeweiligen Anfragesprache bestehen. Das DBS garantiert unter anderem, daß Änderungen erfolgreich beendeter Transaktionen die Konsistenz der Datenbank bewahren und permanent sind, also auch durch Fehler wie Rechner- oder Plattenausfälle nicht verlorengehen. Umgekehrt bleiben Fehlersituationen während einer Transaktionsausführung ohne Auswirkungen auf die Datenbank, da entsprechende Änderungen vollständig zurückgesetzt werden (Alles-oder-Nichts-Eigenschaft von Transaktionen).

1.1 Von der zentralisierten zur verteilten Datenbankverarbeitung

Die traditionelle Datenbankverarbeitung ist zentralisiert, das heißt DBVS sowie Anwendungsprogramme, die auf die Datenbank zugreifen, laufen in einem Rechner ab. Die eigentlichen Benutzer (z.B. Sachbearbeiter einzelner Abteilungen) sind mit dem zentralen Verarbeitungsrechner nur über Terminals oder sonstige Endgeräte verbunden (Abb. 1-1). Diese Organisationsform bringt vor allem Admini-

strationsvorteile mit sich, da die Datenbank sowie die gesamte Software zentral verwaltet werden können.

Abb. 1-1: Zentralisierte Datenbankverwaltung in einer Bankanwendung



Andererseits wird die zentralisierte Systemstruktur in vielen Fällen der Organisationsform eines Unternehmens nicht gerecht. So würde bei einer Großbank mit mehreren Filialen damit verlangt, sämtliche Konten, Personaldaten usw. aller Zweigstellen zentral zu führen. Dies führt zum einen zu einem hohen Kommunikationsaufwand zwischen Filialen und Zentrale, da in den Zweigstellen kein lokaler Datenzugriff möglich ist. Zudem können die starken Abhängigkeiten von der Zentrale zu Akzeptanzproblemen führen und ggf. eine unkontrollierte Verwaltung lokaler Daten hervorrufen. Werden z. B. in den Zweigstellen Kopien der sie betreffenden Personaldaten geführt, dann sind dort eigene Anwendungen zu entwickeln, die u.a. sicherstellen, daß die Kopien der einzelnen Datenbestände auf dem gleichen Stand sind. Eine Vervielfachung derartiger Datenverwaltungs- und Konsistenzprobleme ergibt sich mit der Möglichkeit, private Datenbanken auf PCs und Workstations zu halten.

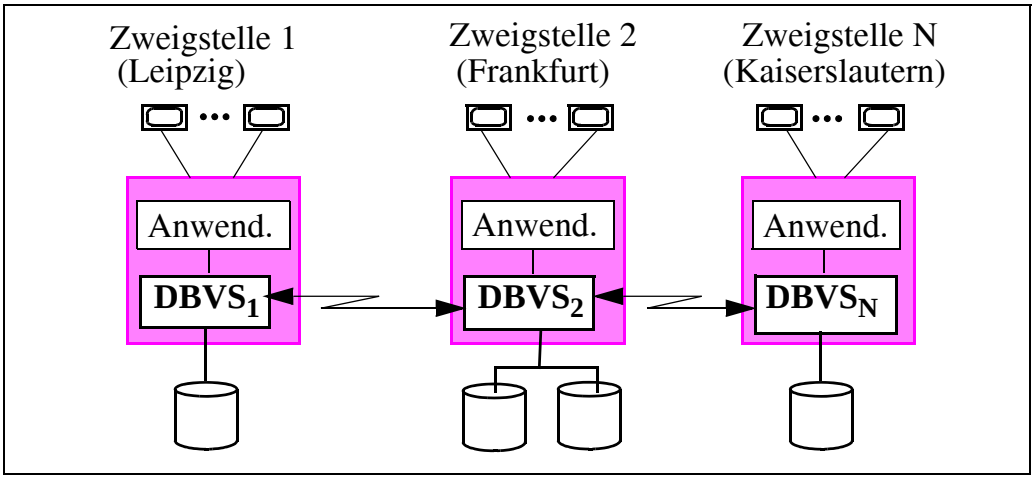
Weiterhin führt ein einziger Verarbeitungsrechner oft zu inakzeptablen Leistungs- und Verfügbarkeitsproblemen. Durchsatz- und Antwortzeitanforderungen können i.a. nur mit einem Großrechner (Mainframe) erfüllt werden, was zu hohen Kosten führt. Für große Anwendungen ist selbst die Kapazität eines Großrechners oft nicht ausreichend, zumal die Leistungsforderungen zum Teil stärker steigen als die Prozessorgeschwindigkeit. Weiterhin führt die zentralisierte Lösung zu einer geringen Verfügbarkeit, da der Ausfall des zentralen DB-Rechners den gesamten Betrieb lahmlegt, wenn nicht ein Reserverechner die Verarbeitung fortsetzen kann. Längere Ausfallzeiten beim Datenbankbetrieb sind in vielen Anwendungs-

bereichen wie etwa bei Telefongesellschaften, Platzreservierungssystemen oder Banken völlig inakzeptabel.

Aufgrund der Beschränkungen zentralisierter Datenbanksysteme wurden unterschiedliche Typen sogenannter *Mehrrechner-Datenbanksysteme* entwickelt, bei denen die Datenbankverwaltungsfunktionen auf mehreren Prozessoren bzw. Rechnern ausgeführt werden. Im folgenden wollen wir einführend auf einige Vertreter von Mehrrechner-Datenbanksystemen eingehen. Genauere Definitionen sowie eine Klassifikation und Abgrenzung der wesentlichen Alternativen erfolgen in Kap. 3.

Einen relativ geringen Entwicklungsschritt stellt der Übergang von einem zentralisierten DBS zu *Multiprozessor-Datenbanksystemen** dar, mit denen die Kapazität von Multiprozessoren zur DB-Verarbeitung genutzt werden kann. Hierzu wird das DBVS i.a. innerhalb mehrerer Prozesse ausgeführt, die vom Betriebssystem dynamisch den Prozessoren zugewiesen werden. Da mit einem solchen Ansatz die meisten Nachteile zentralisierter Datenbanksysteme weiterhin Bestand haben, wird bei den wichtigsten Mehrrechner-DBS die Datenbankverarbeitung auf mehreren unabhängigen Rechnern ausgeführt, die selbst wiederum als Multiprozessoren ausgelegt sein können. Dabei läuft auf jedem der Rechner ein eigenes DBVS; die DBVS kooperieren untereinander, um den Anwendungen gegenüber Aspekte der Verteilung möglichst zu verbergen.

Abb. 1-2: Einsatz eines Verteilten Datenbanksystems

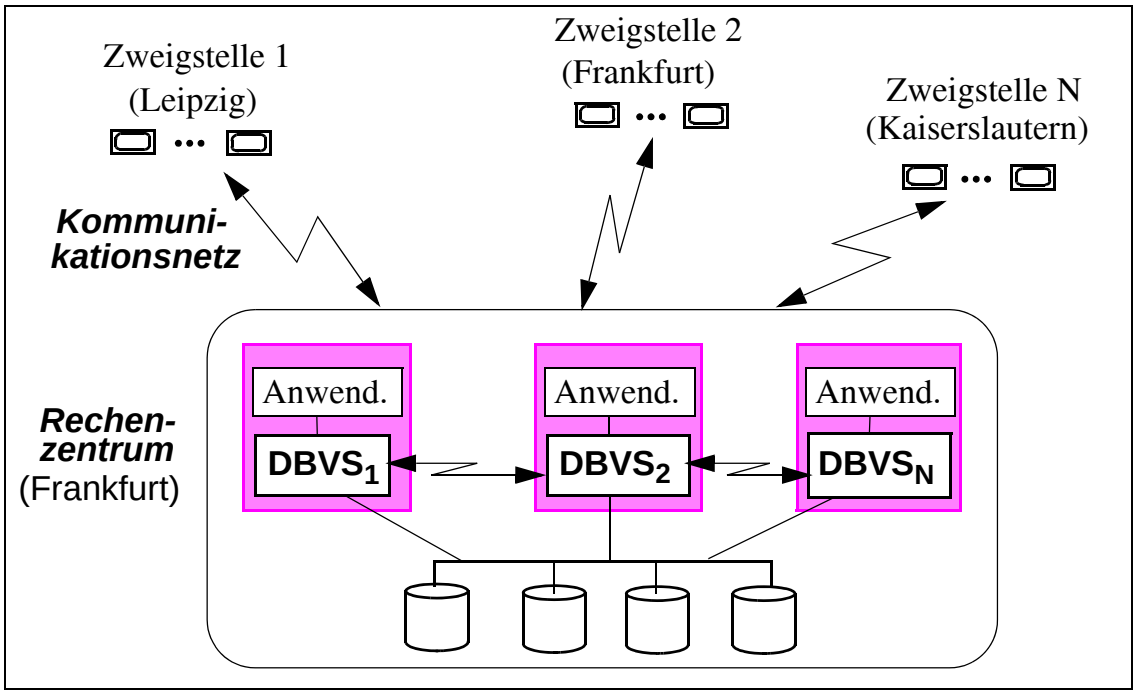


Verteilte Datenbanksysteme [BEKK84, CP84, Da86, ÖV91, BG92] repräsentieren einen bekannten Vertreter solch allgemeinerer Mehrrechner-Datenbanksysteme, sind jedoch nicht mit ihnen gleichzusetzen. Eine wesentliche Eigenschaft Verteilter Datenbanksysteme ist, daß sie eine geographisch verteilte Datenbankverwaltung und damit eine Anpassung an dezentrale Organisationsstrukturen gestatten.

* Solche Systeme werden wir später auch als "Shared-Everything" bezeichnen (Kap. 3).

Wie im Beispiel von Abb. 1-2 gezeigt, kann somit jeder Zweigstelle einer Bank ein lokales DBVS zugeordnet werden, das die Konto- und Personaldaten der jeweiligen Filiale in einer lokalen Datenbank(partition) hält. Da i.a. die meisten Datenbankzugriffe auf lokale Daten erfolgen, läßt sich ein Großteil der Kommunikationsverzögerungen einsparen. Dennoch kann weiterhin auf die gesamte Datenbank zugegriffen werden, da die DBVS der einzelnen Rechner hierzu miteinander kooperieren. Betrifft ein Datenbankzugriff Daten eines anderen Rechners, wird der entsprechende Auftrag unsichtbar für den Benutzer bzw. das Anwendungsprogramm an diesen weitergeleitet. Der Einsatz mehrerer Verarbeitungsrechner erlaubt eine gesteigerte Leistungsfähigkeit und Verfügbarkeit gegenüber zentralisierten DBS. Der Ausfall eines Rechners betrifft insbesondere lediglich dessen lokale Daten; auf die restliche Datenbank kann weiterhin zugegriffen werden. Nachteilig bei Verteilten Datenbanksystemen ist u.a. die Notwendigkeit einer Systemadministration an jedem Knoten. Weiterhin können Zugriffe auf nichtlokale Daten zu erheblichen Leistungseinbußen führen, da die Kommunikation in Weitverkehrsnetzen i.a. relativ langsam ist und einen hohen Instruktionsbedarf erfordert (Kap. 2.2).

Abb. 1-3: Einsatz eines Parallelen Datenbanksystems vom Typ "Shared Disk"



Einen Kompromiß zwischen zentralisierten bzw. Mehrprozessor-DBS und Verteilten Datenbanksystemen stellen bestimmte *Parallele Datenbanksysteme* bzw. *lokal verteilte Mehrrechner-DBS* dar (Abb. 1-3). Wie bei Verteilten Datenbanksystemen kooperieren dabei mehrere DBVS untereinander, die jeweils auf einem eigenen Rechner ablaufen. Allerdings sind die Rechner in räumlicher Nachbarschaft angeordnet (innerhalb eines Rechenzentrums), um die Nutzung eines schnellen

Kommunikationsmediums zwischen den Verarbeitungsrechnern sowie eine zentrale Systemadministration zu erlauben. Für die Zuordnung der physischen Datenbank auf den Externspeichern zu den DBVS bestehen zwei wesentliche Alternativen. Die Datenbank kann entweder partitioniert werden (ähnlich wie bei Verteilten Datenbanksystemen), so daß jedes DBVS eine Partition zugeordnet bekommt, oder jedes DBVS kann direkt auf die gesamte Datenbank (sämtliche Platten) zugreifen. Im letzteren Fall spricht man von einem DB-Sharing- oder *Shared-Disk*-Ansatz (Abb. 1-3), im ersteren Fall von *Shared-Nothing**.

Solche Parallelen DBS erlauben weit bessere Leistungs- und Verfügbarkeitsmerkmale als zentralisierte Datenbanksysteme unter Beibehaltung einer zentralen Administrierbarkeit. Gegenüber Verteilten Datenbanksystemen schlägt neben der einfacheren Systemverwaltung vor allem die Nutzung eines effizienten Kommunikationssystems positiv zu Buche. Dafür ist jedoch eine Anpassung an geographisch verteilte Organisationsstrukturen ("DB-Verarbeitung vor Ort") nicht mehr möglich. Die Bezeichnung "Parallele DBS" rührt daher, daß diese Ansätze die Nutzung von Parallelrechnern mit zahlreichen Prozessoren für die DB-Verarbeitung ermöglichen. Dabei soll auch eine Parallelverarbeitung innerhalb von Transaktionen bzw. DB-Anfragen unterstützt werden.

Mehrrechner-Datenbanksysteme stellen nicht die einzige Möglichkeit einer verteilten Transaktionsbearbeitung dar. Diese kann nämlich auch außerhalb des DBVS realisiert werden, z.B. indem Anwendungen und DBVS auf getrennten Rechnern ablaufen. Eine solche Systemstruktur findet sich häufig in *Client/Server-Umgebungen*, wenn Anwendungsprogramme von Workstations und PCs auf entfernte Datenbanken von Server-Rechnern zugreifen. Die Realisierungsalternativen zur verteilten Transaktionsverarbeitung werden in Kap. 11 behandelt.

1.2 Anforderungen an Mehrrechner-Datenbanksysteme

Die Anforderungen an Mehrrechner-Datenbanksysteme ergeben sich weitgehend aus den zum Teil schon erwähnten Beschränkungen zentralisierter Datenbanksysteme. Im einzelnen sollte ein "ideales" Mehrrechner-DBS folgende Anforderungen erfüllen:

Hohe Leistungsfähigkeit

In typischen Einsatzbereichen von Datenbanksystemen dominiert die Ausführung relativ einfacher Anwendungsfunktionen, z.B. zur Durchführung einer Reservierung oder einer Kontenbuchung. Diese Funktionen werden jedoch mit großer Häufigkeit ausgeführt, so daß ein hoher Durchsatz mit für die Dialogbearbeitung ausreichend kurzen Antwortzeiten zu gewährleisten ist. Da selbst einfache Transak-

* In unserer Klassifikation in Kap. 3 werden wir Verteilte DBS als geographisch verteilte Shared-Nothing-Systeme definieren.

tionen oft einen CPU-Bedarf von über 100.000 Instruktionen aufweisen, können hohe Transaktionsraten (z.B. > 1000 Transaktionen pro Sekunde) nur durch Einsatz mehrerer Verarbeitungsrechner erreicht werden. Um eine effektive Nutzung von deren Kapazität zu ermöglichen, muß die verteilte Transaktionsbearbeitung innerhalb des Mehrrechner-DBS mit einem Minimum an Kommunikationsvorgängen erfolgen. Ferner muß die Transaktionslast so unter die Rechner aufgeteilt werden, daß möglichst eine gleichmäßige Auslastung der Rechner erreicht und somit die Überlastung einzelner Rechner vermieden werden (Lastbalancierung).

Neben der Bearbeitung vorgeplanter Anwendungen sind durch das Datenbanksystem auch zunehmend komplexe Anfragen (Queries) zu bearbeiten, die den Zugriff auf große Datenmengen und/oder aufwendige Berechnungen erfordern. Um für solche Anfragen ausreichend kurze Antwortzeiten erreichen zu können, ist eine Parallelisierung der Anfrageverarbeitung notwendig, wobei sowohl Parallelität bezüglich der Daten-E/A als auch bei der Verarbeitung selbst zu unterstützen ist. Ansonsten würde bereits das (sequentielle) Einlesen von Platte bei Datenmengen von z.B. 100 Gigabyte mehrere Stunden erfordern.

Modulare Wachstumsfähigkeit des Systems (Skalierbarkeit)

Die Leistungsfähigkeit des Systems sollte durch Hinzunahme weiterer Verarbeitungsrechner inkrementell erweiterbar sein. Idealerweise steigt dabei der Durchsatz linear mit der Rechneranzahl bzw. die parallele Verarbeitung komplexer Anfragen kann proportional mit der Rechneranzahl beschleunigt werden.

Hohe Verfügbarkeit

Die starke Verbreitung von Datenbanksystemen führt in den jeweiligen Umgebungen auch zu entsprechenden Abhängigkeiten von der Verfügbarkeit des Systems. In Bereichen, wo Umsatz und Gewinn direkt von der Möglichkeit des Datenbankzugriffs abhängen, wird daher vielfach eine "permanente" Verfügbarkeit verlangt (z.B. Ausfallzeiten von weniger als 1 Stunde pro Jahr [GS91]). Dies kann nur durch ausreichende Redundanz in allen wichtigen Hardware- und Software-Komponenten erreicht werden. Insbesondere muß ein Mehrrechner-Datenbanksystem den Ausfall eines DBVS (Rechners) tolerieren können, indem die Verarbeitung in den überlebenden DBVS möglichst unterbrechungsfrei fortgeführt wird. Die Daten selbst müssen mehrfach gespeichert werden, um einen Datenverlust bei Ausfall eines Externspeichers zu vermeiden. Um auch gegenüber "Katastrophen" gewappnet zu sein (z.B. Ausfall eines gesamten Rechenzentrums durch Erdbeben, Überschwemmung oder Terroranschlag) ist ggf. eine Replikation der Daten an geographisch verteilten Rechnern zu unterstützen (Kap. 9.5).

Verteilungstransparenz

Die Tatsache, daß die Datenbankverarbeitung auf mehreren Rechnern erfolgt, sollte gegenüber Anwendungen und Benutzern vollkommen unsichtbar bleiben.

Diese Forderung der Verteilungstransparenz impliziert u.a. *Ortstransparenz,* das heißt, die physische Lokation eines Datenbankobjektes muß den Anwendungen gegenüber verborgen bleiben. Werden Daten vom Mehrrechner-DBS repliziert gespeichert, ist auch dies gegenüber Anwendungen vollständig zu verbergen (*Replikationstransparenz*); die notwendige Wartung der Replikation nach Änderungen ist Aufgabe des Mehrrechner-DBS*. Die Forderung der Verteilungstransparenz ist wesentlich für die Einfachheit der Anwendungserstellung und -wartung, da so z.B. Änderungen in der Verteilung von Daten ohne Rückwirkungen auf die Transaktionsprogramme bleiben. Die Nutzung des Datenbanksystems sollte wie im zentralisierten Fall möglich sein.

Verteilungstransparenz bedeutet ferner, daß die Zusicherungen des Transaktionskonzepts (Kap. 2.1.3) auch für Mehrrechner-DBS gewahrt bleiben.

Unterstützung dezentraler Organisationsstrukturen

Große Unternehmen und Institutionen sind häufig geographisch verteilt organisiert. Um die Abhängigkeiten von einem zentralen Rechenzentrum zu reduzieren, soll eine Datenverwaltung und Transaktionsbearbeitung vor Ort unterstützt werden [Gr86].

Integrierter Zugriff auf heterogene Datenbanken

In vielen Anwendungsfällen sind benötigte Informationen über mehrere unabhängige Datenbanken verstreut, die typischerweise von unterschiedlichen DBS auf mehreren Rechnern verwaltet werden [Th90, SW91]. Es sollte möglich sein, innerhalb einer Transaktion auf diese unabhängigen und i.a. heterogenen Datenbanken zuzugreifen. Der Zugriff sollte dabei über eine einheitliche Anfragesprache erfolgen können.

Einfache Systemadministration

Im verteilten Fall ist generell mit einer Verkomplizierung der Systemverwaltung gegenüber zentralisierten Systemen zu rechnen, da insbesondere eine Zuordnung von Programmen, Daten und Transaktionen unter mehrere Rechner festzulegen ist. Um die Komplexität dafür zu begrenzen, sollten diese Aufgaben entweder vollkommen automatisiert (z.B. Lastverteilung) oder zumindest durch entsprechende Tools (z.B. zur Allokation von Daten und Programmen) unterstützt werden. Entsprechend sollten Änderungen bei der Last-, Programm-, oder Datenallokation möglichst automatisch durchgeführt werden (z.B. nach einem Rechnerausfall oder bei dauerhaften Leistungsproblemen).

Im geographisch verteilten Fall ergibt sich eine weitergehende Erschwerung der Systemverwaltung, da i.a. in jedem Knoten eine eigene Administration notwendig ist. Diese sollte einerseits vor allem die Verarbeitung lokaler Anwendungen und

* Weitere Transparenzforderungen werden in Kap. 4.1 erhoben.

Benutzer unterstützen, jedoch ist auch eine globale Koordinierung zwischen den Knoten erforderlich (z. B. für Änderungen im logischen Datenbankaufbau oder zur Definition globaler Zugriffsberechtigungen). Die Abhängigkeit zu anderen Rechnern sollte jedoch möglichst gering bleiben, das heißt, es ist ein hohes Maß an *Knotenautonomie* anzustreben [GK88, SL90].

Hohe Kosteneffektivität

Die enormen Vorteile von Mikroprozessoren gegenüber Großrechnern bezüglich Kosteneffektivität (Kosten pro MIPS) sollten auch für die Datenbankverarbeitung genutzt werden. Dies gilt umso mehr, da die Leistungsfähigkeit von Mikroprozessoren (z.B. auf RISC-Basis) derzeit wesentlich schneller zunimmt als in anderen Rechnerklassen [Ra93a]. Mehrrechner-DBS sollten daher Mikroprozessoren als Rechnerknoten verwenden können.

Diskussion

Leider gibt es keinen idealen Typ eines Mehrrechner-Datenbanksystems, mit dem alle genannten Anforderungen erfüllt werden können. Dies liegt schon daran, daß einige Anforderungen gegensätzlicher Natur sind. So verlangt die Forderung nach Verteilungstransparenz eine enge Zusammenarbeit der einzelnen DBVS, was jedoch der Forderung einer hohen Knotenautonomie sowie Unterstützung heterogener Datenbanken entgegensteht. Weiterhin sind einfache Systemverwaltung sowie Unterstützung geographisch verteilter Organisationsstrukturen weitgehend gegensätzliche Anforderungen. Aus diesen Gründen wurden unterschiedliche Typen von Mehrrechner-DBS entwickelt, die bestimmte Teilmengen der Forderungen abdecken. Hierauf wird in Kapitel 3 genauer eingegangen.

2 Datenbanksysteme und Rechnernetze

Dieses Kapitel behandelt kurz Grundlagen zu Datenbanksystemen und Rechnernetzen, die für das weitere Verständnis des Buchs benötigt werden.

2.1 Datenbanksysteme

Das in diesem Buch unterstellte Datenmodell ist das relationale Datenmodell, wenngleich diese Annahme sich nur für bestimmte Funktionen auswirkt (z.B. bei der Datenverteilung oder Query-Bearbeitung). Das relationale Modell hat sich gegenüber seinen älteren Konkurrenzmodellen (hierarchisches Modell, Netzwerk-Modell nach CODASYL) eindeutig durchgesetzt; neuere (objekt-orientierte, semantische) Datenmodelle können auf absehbare Zeit die Marktposition relationaler Systeme nicht gefährden. Ein wesentlicher Vorteil des relationalen Modells zur verteilten und parallelen Datenbankverarbeitung liegt darin, daß die Anfragen mengenorientiert sind und somit ein weit größeres Optimierungspotential besteht als für satzorientierte, navigierende Anfragesprachen [[St80], [DG92]]. Die Partitionierung einer nicht-relationalen Datenbank unter mehreren Rechnern ist darüber hinaus auch nur sehr eingeschränkt möglich [[Fi85]]. Die Mehrzahl derzeitiger Mehrrechner-Datenbanksysteme basiert aus diesen Gründen auch auf dem relationalen Datenmodell.

Wir diskutieren zunächst die Grundzüge des relationalen Datenmodells einschließlich seiner Operationen. Danach beschreiben wir die Schemaarchitektur von Datenbanksystemen und führen ein dreistufiges Schichtenmodell ein, das den internen Aufbau von DBS verdeutlicht. Die zur Transaktionsverwaltung benötigten Funktionen werden separat mit dem Transaktionskonzept erläutert (Kap. 2.1.3). Abschließend skizzieren wir den Aufbau von Transaktionssystemen, die den wichtigsten Einsatzbereich von Datenbanksystemen bilden.

2.1.1 Relationale Datenbanken

In relationalen Datenbanken werden die Daten in Tabellen oder *Relationen* gespeichert. Jede Relation besteht aus einer bestimmten Anzahl von Spalten oder *Attributen* sowie einer Menge von Zeilen oder *Tupeln* (Sätzen). Die Anzahl der Attribute bestimmt den *Grad*, die Anzahl der Tupel die *Kardinalität* einer Relation. Abb. 2-1 zeigt ein Beispiel zweier Relationen vom Grad 3 und Kardinalität 6 bzw. 5.

Abb. 2-1: Beispiel von Relationen

Relation KONTO

KTONR	***KNR***	*KTOSTAND*
1234	K2	122,34
2345	K4	-12,43
3231	K1	1222,22
7654	K5	63,79
9876	K2	55,77
5498	K4	-4506,77

Relation KUNDE

KNR	*NAME*	*GEBDAT*
K2	Schulz	2.11.1976
K4	Meier	23.8.1972
K3	Müller	4.7.1987
K1	Scholz	24.4.1959
K5	Weber	17.3.1942

Jedem Attribut ist ein sogenannter *Domain* (Definitionsbereich) zugeordnet, der die zulässigen Werte festlegt. Die Beschreibung einer Relation umfaßt den Namen der Relation und deren Attribute sowie die zugehörigen Domains. Die Ausprägung der Relation ist durch die Menge der Tupel gegeben und entspricht einer Teilmenge des kartesischen Produktes über den Attribut-Domains. Die Mengeneigenschaft von Relationen bedeutet, daß kein Tupel mehrfach vorkommen darf und daß keine vorgegebene Reihenfolge unter den Tupeln besteht.

Das Relationenmodell schreibt zwei modellinhärente Integritätsbedingungen vor, die sogenannten *Relationalen Invarianten*. Die Primärschlüsselbedingung verlangt, daß zu jeder Relation ein Attribut bzw. eine Kombination von Attributen als *Primärschlüssel* fungiert, mit dem Tupel eindeutig identifiziert werden können. In Abb. 2-1 bilden z.B. die Attribute KTONR (Relation KONTO) und KNR (Relation KUNDE) geeignete Primärschlüssel. Die zweite modellinhärente Integritätsbedingung betrifft sogenannte *Fremdschlüssel*, mit denen Beziehungen zwischen Relationen realisiert werden können. In Abb. 2-1 ist das Attribut KNR (Kundennummer) in Relation KONTO ein solcher Fremdschlüssel, mit dem der Inhaber eines Kontos durch einen Verweis auf den Primärschlüssel der Relation KUNDE repräsentiert wird. Die Fremdschlüsselbedingung verlangt, daß das durch einen Fremdschlüsselwert referenzierte Tupel in der Datenbank existiert, d.h., daß in der referenzierten Relation ein entsprechender Primärschlüsselwert definiert sein muß (referentielle Integrität).

Zur Beschreibung relationaler Anfragen greifen wir (neben SQL) vor allem auf die *Relationenalgebra* zurück. Die Relationenalgebra erlaubt eine kompakte Darstellung von Operationen und spezifiziert explizit die Basisoperatoren, die bei der Anfragebearbeitung auszuführen sind. Die Operatoren der Relationenalgebra sind mengenorientiert, da sie aus einer oder zwei Eingaberelation(en) wiederum eine Relation erzeugen. Neben allgemeinen mengentheoretischen Operatoren wie Vereinigung ($\cup$), Durchschnittsbildung ($\cap$), kartesisches Produkt ($\times$) oder Differenz interessieren vor allem die relationalen Operatoren Selektion (Restriktion), Projektion und Verbund (Join). Mit der *Selektion*

$$\sigma_P\,(R)$$

wird eine zeilenweise (horizontale) Teilmenge einer Relation R gebildet, die alle Tupel enthält, die das Selektionsprädikat P erfüllen. Ein Beispiel dazu findet sich in Abb. 2-2a. Die *Projektion*

$$\pi_{Attributliste}\,(R)$$

bildet eine spaltenweise (vertikale) Teilmenge der Eingaberelation, wobei alle nicht spezifizierten Attribute "weggefiltert" werden (Beispiel: Abb. 2-2b). Dabei werden zudem Duplikate eliminiert, um die Mengeneigenschaft der Ergebnisrelation zu wahren. Der *Verbund* ($\bowtie$) schließlich erlaubt die Verknüpfung zweier Relationen, die Attribute mit übereinstimmenden Domains besitzen. Der mit Abstand wichtigste Verbundtyp ist der *Gleichverbund* (Equi-Join), bei dem die Verknüpfung durch eine Gleichheitsbedingung auf den Verbundattributen definiert ist. Der Gleichverbund zwischen Relation R mit Verbundattribut r und Relation S mit Verbundattribut s kann als Selektion auf dem kartesischen Produkt zwischen R und S definiert werden:

$$R \bowtie_{r=s} S \;=\; \sigma_{r=s}(R \times S)$$

Der *natürliche Verbund* zwischen zwei Relationen, bei denen die Verbundattribute den gleichen Namen besitzen, ist definiert als Gleichverbund zwischen den Relationen, wobei die Verbundattribute nur einmal im Ergebnis vorkommen. Aufgrund der Namenskonvention kann beim natürlichen Verbund die Join-Bedingung weggelassen werden ($R \bowtie S$). Abb. 2-2c zeigt ein Beispiel für den natürlichen Verbund.

Der *Semi-Join* zwischen zwei Relationen R und S ($R \ltimes S$) ist definiert als ein Verbund, in dessen Ergebnis nur Attribute der Relation R enthalten sind:

$$R \ltimes_{r=s} S \;=\; \pi_{R\text{-}Attribute}\,(R \bowtie_{r=s} S)$$

Abb. 2-2: Beispiele relationaler Operatoren (bezogen auf die Relationen aus Abb. 2-1)

a) Selektion: $\sigma_{KTOSTAND < 0}$ (KONTO)

(Konten mit negativem Kontostand)

KTONR	*KNR*	*KTOSTAND*
2345	K4	-12,43
5498	K4	-4506,77

b) Projektion

π_{KNR} (KONTO)

KNR
K2
K4
K1
K5

c) Natürlicher Verbund: **KONTO** $\bowtie$ **KUNDE**

KTONR	*KNR*	*KTOSTAND*	*NAME*	*GEBDAT*
1234	K2	122,34	Schulz	2.11.1976
2345	K4	12,43	Meier	23.8.1972
3231	K1	1222,22	Scholz	24.4.1959
7654	K5	63,79	Weber	17.3.1942
9876	K2	55,77	Schulz	2.11.1976
5498	K4	-4506,77	Meier	23.8.1972

d) Semi-Join: **KUNDE** $\ltimes$ **KONTO**

KNR	*NAME*	*GEBDAT*
K2	Schulz	2.11.1976
K4	Meier	23.8.1972
K1	Scholz	24.4.1959
K5	Weber	17.3.1942

e) Anfrage mit mehreren Operatoren:

π_{NAME} ($\sigma_{KTOSTAND < 0}$ (**KONTO** $\bowtie$ **KUNDE**))

(Namen der Kunden, die ein Konto mit negativem Kontostand besitzen)

NAME
Meier

Abb. 2-2d zeigt ein Beispiel für einen (natürlichen) Semi-Join. Semi-Joins haben vor allem im Kontext von Verteilten Datenbanksystemen Bedeutung erlangt. Sie

erlauben eine Verkleinerung der Ergebnismenge gegenüber dem normalen Join, was zu einer Reduzierung der zu übertragenden Datenmengen genutzt werden kann (s. Kap. 6.5.2).

Da die Operatoren Relationen als Eingabe verwenden und wiederum Relationen erzeugen, können sie miteinander kombiniert werden, um komplexere Anfragen zu formulieren. Ein Beispiel dazu ist in Abb. 2-2e gezeigt.

In der Praxis werden DB-Anfragen natürlich in einer benutzerfreundlicheren und deskriptiveren Anfragesprache als der Relationenalgebra formuliert. Hier hat sich SQL eindeutig durchgesetzt. Diese Anfragesprache wird von nahezu allen Datenbanksystemen unterstützt und ist im Rahmen der ISO standardisiert. Der aktuelle Standard (SQL2, SQL92) ist gegenüber seinen Vorgängerversionen sehr mächtig und umfassend und wird von existierenden Implementierungen derzeit meist nur teilweise abgedeckt. Für dieses Buch werden lediglich geringe SQL-Kenntnisse vorausgesetzt, da nur vereinzelt Beispiele in SQL formuliert werden, die zudem weitgehend selbsterklärend sein dürften. Die Anfrage aus Abb. 2-2e würde in SQL z.B. folgendermaßen lauten:

```
SELECT  NAME                        { Projektion auf Attribut NAME }
FROM    KONTO,KUNDE                 { Namen der beteiligten Relationen }
WHERE   KTOSTAND < 0                    { Selektionsbedingung }
        AND KONTO.KNR = KUNDE.KNR   { Join-Bedingung }
```

Besonderheiten von SQL, die natürlich auch durch Mehrrechner-DBS zu unterstützen sind, betreffen u.a. die Berechnung sogenannter Aggregatfunktionen (MIN, MAX, SUM, COUNT, AVG) sowie die optionale Sortierung von Ergebnismengen. Für eine Einführung in SQL sei auf die reichlich vorhandene Literatur verwiesen; der aktuelle Sprachumfang von ISO-SQL wird z.B. in [[DD92], [MS92]] beschrieben.

2.1.2 Aufbau von Datenbanksystemen

Wir diskutieren zunächst die sogenannte Schemaarchitektur zentralisierter DBS, mit der unterschiedliche Sichtweisen auf eine Datenbank unterstützt werden. Anschließend beschreiben wir den internen Aufbau von DBVS anhand eines dreistufigen Schichtenmodells.

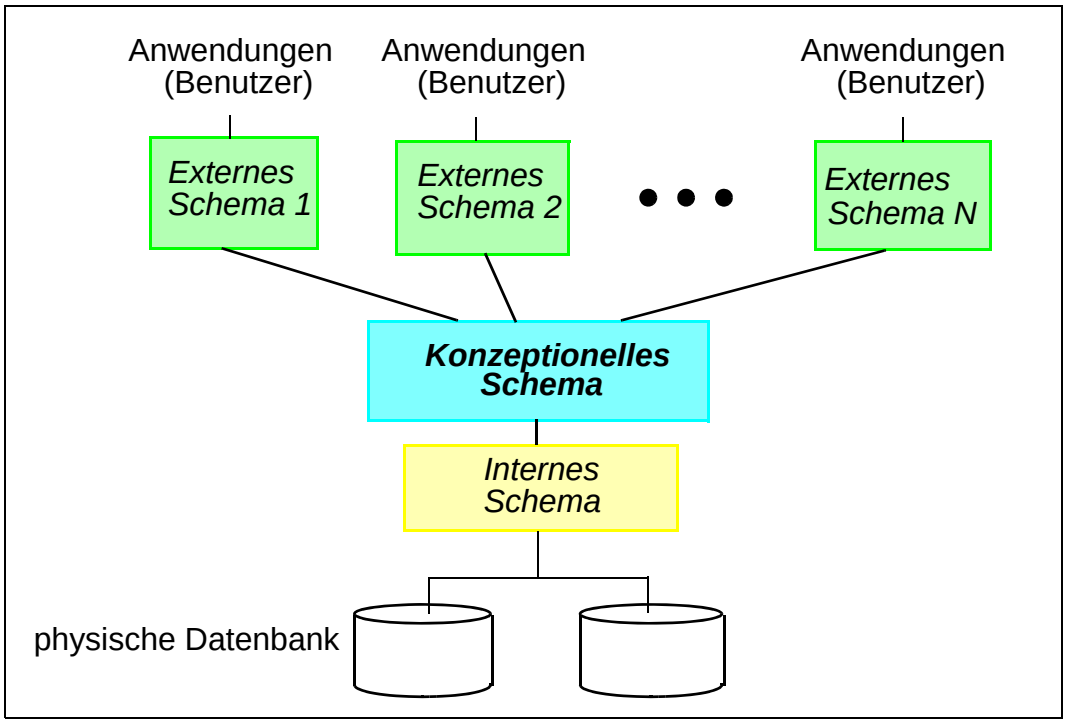
Schemaarchitektur

Ein wesentlicher Vorteil moderner Datenbanksysteme liegt in der Bereitstellung einer hohen Datenunabhängigkeit sowie der Möglichkeit, verschiedenen Benutzern unterschiedliche Sichtweisen auf eine Datenbank zu ermöglichen. Dies wird durch eine dreistufige Schemaarchitektur unterstützt (Abb. 2-3), welche von dem ANSI/SPARC-Kommittee vorgeschlagen wurde und in vielen DBS realisiert ist.

Im einzelnen sind dabei drei Arten von Schemata zur Beschreibung des Datenbankaufbaus zu unterscheiden:

- Im Mittelpunkt steht dabei das *konzeptionelle Schema,* das den logischen Aufbau der Datenbank vollständig beschreibt. Im Falle relationaler Datenbanken enthält das konzeptionelle Schema die Definition sämtlicher Relationen, Attribute, Domains, Integritätsbedingungen etc.
- Benutzer bzw. Anwendungsprogramme greifen i.a. nicht direkt über das konzeptionelle Schema auf die Datenbank zu, sondern über auf ihre Anforderungen und Zugriffsrechte ausgerichtete *externe Schemata*. Ein externes Schema enthält i.a. nur eine Teilmenge der Objekte des konzeptionellen Schemas, wodurch sich für die betreffende Benutzergruppe eine einfachere DB-Benutzung ergibt sowie eine Zugriffskontrolle unterstützt wird. Es wird damit auch ein höherer Grad an Datenunabhängigkeit als mit dem konzeptionellen Schema erreicht, da Änderungen im logischen DB-Aufbau nur dann Rückwirkungen auf Anwendungen haben, falls sie Objekte des jeweiligen externen Schemas betreffen (logische Datenunabhängigkeit).
- Das *interne Schema* beschreibt, wie logische Objekte des konzeptionellen Schemas physisch gespeichert werden sollen (Clusterbildung, Komprimierung etc.) und welche Indexstrukturen vom DBVS zu warten sind. Diese Angaben werden vom Datenbank-Administrator festgelegt und sind für den DB-Benutzer vollkommen transparent (physische Datenunabhängigkeit).

Abb. 2-3: DB-Schemaarchitektur nach ANSI/SPARC



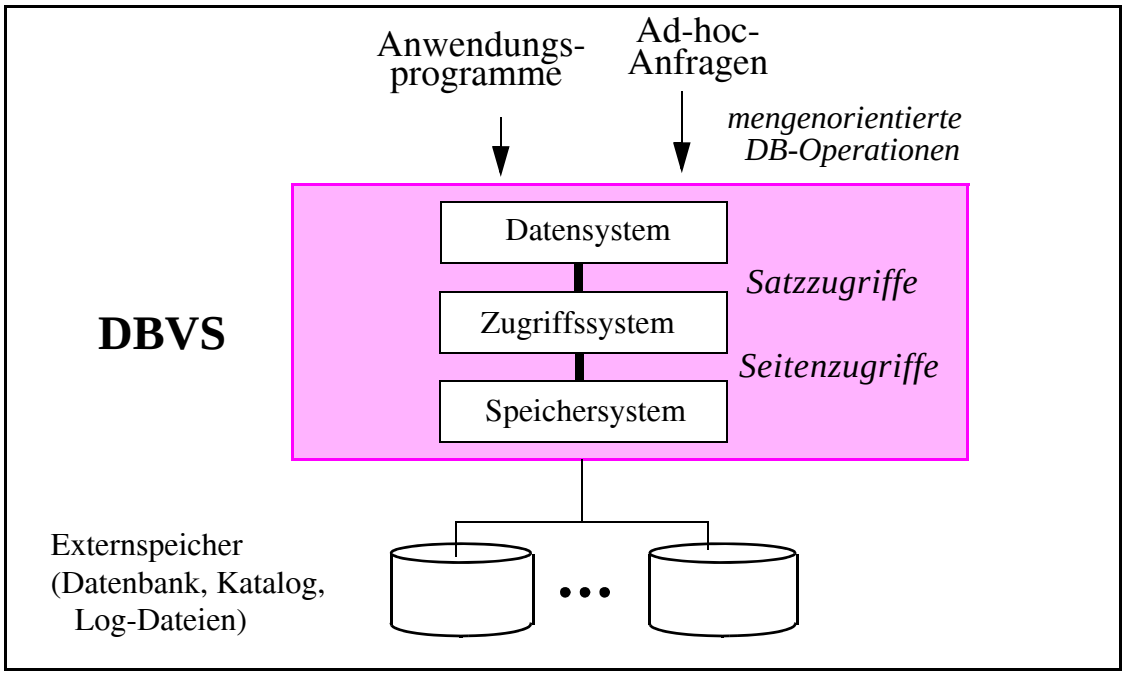
Die Schemaangaben repräsentieren Beschreibungsinformationen oder Metadaten über den Datenbankaufbau, die vom DBVS innerhalb eines *Katalogs* (Data Dictionary) geführt werden, ebenso wie benutzerspezifische Angaben (Zugriffsrechte etc.). Die Daten selbst werden physisch gemäß der Spezifikation des internen Schemas gespeichert. Beim Datenzugriff sind durch das DBVS gemäß der Schema-

architektur entsprechende Abbildungen zwischen externer und interner Repräsentation vorzunehmen.

Schichtenmodell

Der interne Aufbau eines Datenbankverwaltungssystems kann durch mehrstufige Schichtenmodelle beschrieben werden [[Hä87]]. Dabei werden die Funktionen des DBVS aufeinander aufbauenden Schichten zugeordnet, welche Operationen und Objekte der DBVS-Schnittstelle schrittweise auf interne Strukturen abbilden bis hinunter auf die Bitebene der Externspeicher. Abb. 2-4 verdeutlicht diesen Abbildungsprozeß anhand eines dreistufigen DBVS-Schichtenmodells bestehend aus einem Daten-, Zugriffs- und Speichersystem.

Abb. 2-4: Schichtenmodell eines zentralisierten Datenbanksystems



Das *Datensystem* bildet in diesem Modell die oberste Schicht, welche die mengenorientierten DB-Operationen einer deskriptiven Anfragesprache auf Objekten eines externen Schemas zu verarbeiten hat. Diese Operationen werden entweder im Rahmen vordefinierter Anwendungsprogramme gestellt oder aber interaktiv als Ad-hoc-Anfragen abgesetzt (direkt durch den Endbenutzer bzw. von einem Benutzer-Tool erzeugt). Aufgabe des Datensystems ist es, zu den DB-Operationen entsprechende Ausführungspläne zu erstellen (Query-Übersetzung und -Optimierung) und zu bearbeiten. Dabei werden u.a. externe Objektbezeichnungen auf interne Namen abgebildet und die Verwendung von Basisoperatoren wie Selektion (Scan), Projektion oder Join festgelegt. Für jeden der Basisoperatoren können mehrere Implementierungsalternativen bestehen, unter denen in Abhängigkeit der DB-Operation, vorhandener Indexstrukturen sowie anderer Faktoren eine

Auswahl erfolgt. Die Ausführung der Operatoren erfolgt mit den satzorientierten Operationen des Zugriffssystems.

Das *Zugriffssystem* verwaltet die DB-Sätze innerhalb der Seiten und unterstützt entsprechende Satzoperationen. Daneben werden dort Zugriffspfade und Indexstrukturen geführt. In existierenden DBS ist der B*-Baum Standard als Zugriffspfadtyp, da er sowohl den direkten Satzzugriff über Schlüsselwerte als auch die sortiert sequentielle Verarbeitung (z.B. zur Auswertung von Bereichsanfragen) effizient unterstützt. Hash-basierte Speicherungsstrukturen erlauben einen noch schnelleren Direktzugriff über Schlüsselwerte, sind jedoch für sequentielle Zugriffsmuster ungeeignet. Clusterbildung (Clusterung) von häufig zusammen referenzierten Sätzen ist ebenfalls eine wirkungsvolle Technik, um die Anzahl von Seiten- und damit Plattenzugriffen zu reduzieren.

Das *Speichersystem* schließlich ist für die Verwaltung von DB-Seiten zuständig, wobei insbesondere ein DB-Puffer (Systempuffer) im Hauptspeicher verwaltet wird, um Lokalität im Referenzverhalten zur Einsparung physischer E/A-Vorgänge zu nutzen. Die Zugriffe auf Externspeicher erfolgen üblicherweise über die Dateiverwaltung des Betriebssystems, welche eine Abstraktion von physischen Geräteeigenschaften bietet. Die zur Transaktionsverwaltung benötigten Systemfunktionen (Synchronisation, Logging, Recovery, Integritätssicherung; s. Kap. 2.1.3) sind in diesem Schichtenmodell nicht explizit berücksichtigt, da sie i.a. mehrere Schichten betreffen und nicht direkt am Abbildungsprozeß beteiligt sind.

2.1.3 Das Transaktionskonzept

Bezüglich der Ausführung von Transaktionen garantiert das Datenbanksystem die Einhaltung des sogenannten ***Transaktionskonzeptes*** [[Gr81], [HR83], [Hä88a], [We88]]. Dies betrifft die automatische Gewährleistung der folgenden vier kennzeichnenden ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability) von Transaktionen:

1. *Atomarität ("Alles oder nichts")*
   Änderungen einer Transaktion werden entweder vollkommen oder gar nicht in die Datenbank eingebracht. Diese Eigenschaft ermöglicht eine erhebliche Vereinfachung der Anwendungsprogrammierung, da Fehlersituationen während der Programmausführung (z.B. Rechnerausfall) nicht im Programm abgefangen werden müssen. Das Transaktionssystem sorgt dafür, daß die Transaktion in einem solchen Fall vollständig zurückgesetzt wird, so daß keine unerwünschten "Spuren" der Transaktion in der Datenbank verbleiben. Der Programmierer kann somit bei der Realisierung von Anwendungsfunktionen von einer fehlerfreien Umgebung ausgehen.
2. *Konsistenz*
   Die Transaktion ist die Einheit der Datenbank-Konsistenz. Dies bedeutet, daß bei Beginn und nach Ende einer Transaktion sämtliche physischen und logischen Integritätsbedingungen [[SW85], [Re87]] erfüllt sind.

3. *Isolation*
   Datenbanksysteme unterstützen typischerweise eine große Anzahl von Benutzern, die gleichzeitig auf die Datenbank zugreifen können. Trotz dieses *Mehrbenutzerbetriebes* wird garantiert, daß dadurch keine unerwünschten Nebenwirkungen eintreten (z.B. gegenseitiges Überschreiben derselben Datenbankobjekte). Vielmehr bietet das DBS jedem Benutzer bzw. Programm einen "logischen Einbenutzerbetrieb", so daß parallele Datenbankzugriffe anderer Benutzer unsichtbar bleiben. Auch hierdurch ergibt sich eine erhebliche Vereinfachung der Programmierung.
4. *Dauerhaftigkeit*
   Die Dauerhaftigkeit von erfolgreich beendeten Transaktionen wird garantiert. Dies bedeutet, daß Änderungen dieser Transaktionen alle erwarteten Fehler (insbesondere Rechnerausfälle, Externspeicherfehler und Nachrichtenverlust) überleben.

Eigenschaften 1 und 4 werden vom Datenbanksystem durch geeignete Logging- und Recovery-Maßnahmen [[Re81], [HR83]] eingehalten. Nach einem Rechnerausfall wird insbesondere der jüngste transaktionskonsistente Datenbankzustand hergestellt. Dazu erfolgt ein Zurücksetzen aller Transaktionen, die aufgrund des Rechnerausfalles nicht zu Ende gekommen sind (Undo-Recovery); für erfolgreiche Transaktionen wird eine Redo-Recovery vorgenommen, um deren Änderungen in die Datenbank einzubringen (falls erforderlich).

Zur Gewährung der Isolation im Mehrbenutzerbetrieb (Eigenschaft 3) sind geeignete Verfahren zur Synchronisation (Concurrency Control) bereitzustellen. Das allgemein akzeptierte Korrektheitskriterium, zumindest in kommerziellen Anwendungen, ist dabei die Serialisierbarkeit [[BHG87]]. Obwohl ein großes Spektrum von Synchronisationsverfahren zur Wahrung der Serialisierbarkeit vorgeschlagen wurde, verwenden existierende Datenbanksysteme nahezu ausschließlich Sperrverfahren [[Gr78], [Pe87], [BHG87], [Cl92], [GR93]] zur Synchronisation. Lese- und Schreibsperren werden dabei üblicherweise bis zum Transaktionsende gehalten, um Serialisierbarkeit zu gewährleisten und Änderungen einer Transaktion nicht vor deren Commit (erfolgreichem Ende) anderen Transaktionen zugänglich zu machen (striktes Zwei-Phasen-Sperrprotokoll). Für eine hohe Leistungsfähigkeit ist es wesentlich, eine Synchronisation mit möglichst wenig Sperrkonflikten und Rücksetzungen zu erreichen. Dies verlangt die Unterstützung feiner Sperrgranulate (z.B. einzelne DB-Sätze) innerhalb eines hierarchischen Verfahrens sowie ggf. Spezialprotokolle für spezielle Datenobjekte wie Verwaltungsdaten [[Ra88b], [GR93]].

Die Konsistenzüberwachung (Eigenschaft 2) wird in derzeitigen Datenbanksystemen meist noch nicht im wünschenswerten Umfang unterstützt. Insbesondere die Einhaltung anwendungsbezogener (logischer) Integritätsbedingungen muß meist durch die Transaktionsprogramme gewährleistet werden. Da in der neuen Version des SQL-Standards (SQL92) umfassende Sprachmittel zur Definition von Integritätsbedingungen vorgesehen sind, ist in Zukunft jedoch mit einer Verbesserung der Situation zu rechnen.

2.1.4 Transaktionssysteme

In der kommerziellen Datenverabeitung werden Datenbanksysteme vorwiegend im Rahmen sogenannter Transaktionssysteme eingesetzt [[Me88], [Be90], [GR93]]. Solche Transaktionssysteme gestatten die Ausführung vorgeplanter Anwendungsfunktionen (Platzreservierung, Kontostandsabfrage, u.ä.), die durch entsprechende Anwendungs- oder Transaktionsprogramme realisiert werden. Die Zugriffe auf die Datenbank erfolgen durch entsprechende DB-Operationen in den Anwendungsprogrammen und sind daher für den Endbenutzer vollkommen unsichtbar. Die Verarbeitung findet meist im Dialog statt, so daß man auch häufig von ***Online Transaction Processing (OLTP)*** spricht.

Abb. 2-5: Grobaufbau eines zentralisierten Transaktionssystems

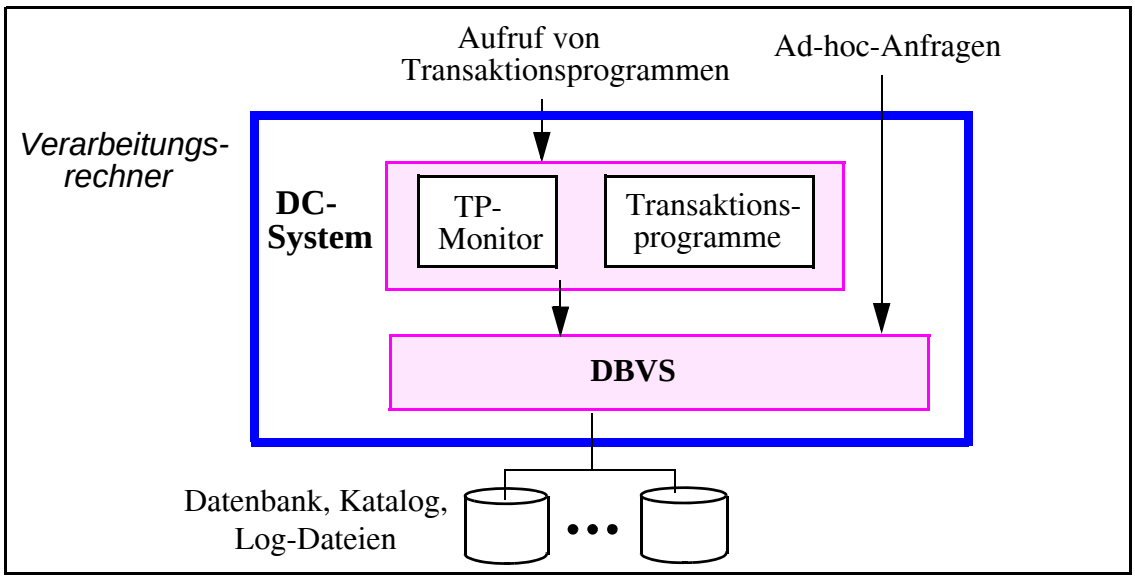


Abb. 2-5 zeigt den Grobaufbau eines zentralisierten Transaktionssystems, welches auf einem einzelnen Verarbeitungsrechner abläuft. Demnach besteht ein Transaktionssystem neben dem DBS vor allem aus einem sogenannten ***DC-System*** (Data Communication System). Das DC-System wiederum besteht aus einem TP-Monitor (Transaction Processing Monitor) sowie einer Menge anwendungsbezogener Transaktionsprogramme.

Der ***TP-Monitor*** kontrolliert die Ausführung der Transaktionsprogramme und realisiert die Kommunikation von Programmen mit Terminals sowie mit dem DBS. Insbesondere können so physische Eigenschaften der Terminals und des Verbindungsnetzwerks sowie Aspekte der Prozeß-Zuordnung von DBS und Programmen für den Anwendungsprogrammierer transparent gehalten werden ("Kommunikationsunabhängigkeit"). Zur Realisierung der Kommunikation benutzt der TP-Monitor typischerweise die entsprechenden Basisdienste des Betriebssystems.

Programm- und Prozeßverwaltung werden dagegen aus Leistungsgründen meist durch den TP-Monitor selbst realisiert. So können durch eine prozeßinterne Auftragsverwaltung ("Multi-Tasking") große Transaktionssysteme mit tausenden Terminals effizient verwaltet werden. Die Bereitstellung eines eigenen Betriebssystem-Prozesses pro Benutzer/Terminal würde in diesem Fall zu inakzeptablem Leistungsverhalten führen.

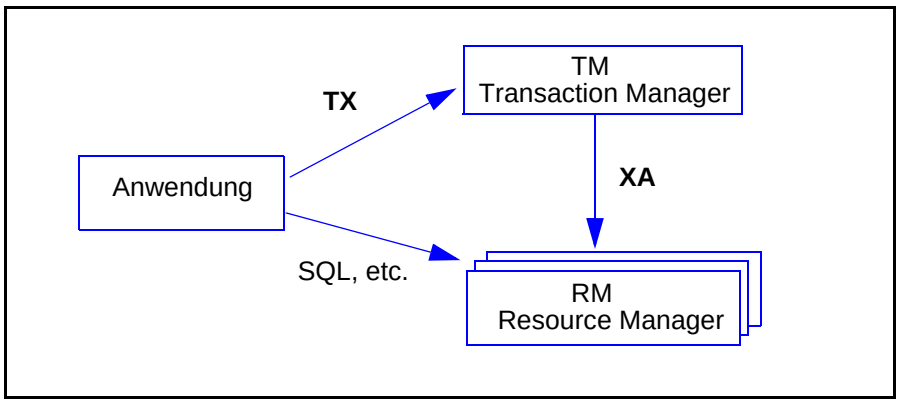
Weitere Aufgaben des TP-Monitors betreffen Nachrichtenverwaltung, Transaktionsverwaltung (in Kooperation mit dem DBVS), Authentifikation sowie Autorisierung. Zur Realisierung von Kommunikationsunabhängigkeit werden Nachrichten zwischen Terminal und Programmen vom TP-Monitor von einem gerätespezifischen in ein neutrales Format transformiert (Konzept des "virtuellen Terminals"). Die Nachrichten werden ferner i.a. protokolliert, um Fehler wie Nachrichtenverlust automatisch behandeln zu können. Desweiteren unterstützen TP-Monitore oft Synchronisations- und Recovery-Funktionen auf Dateien, die nicht unter Kontrolle des DBVS liegen. Zur Gewährleistung der ACID-Eigenschaften wird daher auch ein Commit-Protokoll zwischen TP-Monitor und DBVS erforderlich. Eine ausführliche Beschreibung von TP-Monitoren bietet [[GR93]].

Ein relativ abstraktes Modell eines Transaktionssystems wurde von der X/Open-Organisation definiert, um die Interoperabilität zwischen Teilsystemen (TP-Monitore, DBS etc.) unterschiedlicher Hersteller zu ermöglichen. Zur Verbesserung der Portabilität von Transaktionsanwendungen soll ferner eine einheitliche Anwendungsschnittstelle festgelegt werden. Das Modell für den zentralen Fall ist in Abb. 2-6 gezeigt (die Übertragung auf verteilte Transaktionssysteme wird in Kap. 11.4.1 diskutiert). Dabei wird die Verallgemeinerung getroffen, daß im Rahmen einer Transaktion mehrere sogenannter *Resource-Manager* aufgerufen werden können. Ein Datenbanksystem ist dabei nur ein möglicher Typ eines Resource-Managers; weitere Resource-Manager können z.B. Dateisysteme, Window-Systeme, Mail-Server oder TP-Monitor-Komponenten wie die Nachrichtenverwaltung sein. Ein *Transaction-Manager* (TM) führt die Transaktionsverwaltung durch, insbesondere ein gemeinsames Commit-Protokoll mit allen Resource-Managern, die an einer Transaktionsausführung beteiligt sind, um die Alles-oder-Nichts-Eigenschaft von Transaktionen zu gewährleisten. Der TM ist dabei in einer realen Implementierung üblicherweise Teil des TP-Monitors oder des Betriebssystems. Die Mehrzahl der TP-Monitor-Funktionen sind in dem Modell jedoch nicht explizit repräsentiert, sondern werden als Teil der Ablaufumgebung angesehen.

Die Schnittstellen zwischen Anwendungen und TM (TX-Schnittstelle) sowie zwischen TM und Resource-Managern (XA-Schnittstelle) wurden standardisiert, um eine Interoperabilität zwischen ansonsten unabhängigen Resource-Managern zu erreichen. Die Transaktionsausführung sieht vor, daß eine Anwendung über die TX-Schnittstelle dem TM Beginn, Ende und Abbruch einer Transaktion mitteilt. Der TM vergibt bei Transaktionsbeginn eine eindeutige Transaktionsnummer,

welche danach von der Anwendung bei allen RM-Aufrufen mitgegeben wird. Am Transaktionsende führt der TM dann über die XA-Schnittstelle ein Zweiphasen-Commit-Protokoll mit den Resource-Managern durch. Das X/Open-Modell unterstellt dabei, daß jeder Resource-Manager eine lokale Transaktionsverwaltung unterstützt, insbesondere Synchronisation, Logging und Recovery.

Abb. 2-6: X/Open-Modell eines zentralisierten Transaktionssystems [[GR93]]



Die Schnittstelle zwischen Anwendung und RM, welche vom jeweiligen RM abhängt (z.B. SQL für DBS), sowie die TX-Schnittstelle bilden das sogenannte *Application Programming Interface (API)*.

2.2 Rechnernetze

Ein Rechnernetz verbindet mehrere Rechner miteinander und erlaubt den Austausch von Informationen (Nachrichten) zwischen ihnen. Wir diskutieren zunächst unterschiedliche Typen von Rechnernetzen und gehen danach auf das ISO-Referenzmodell zur Kommunikation ein. Für eine detaillierte Behandlung von Rechnernetzen und Kommunikationsprotokollen muß auf die reichlich vorhandene Literatur verwiesen werden, z.B. [[Ta88], [Cy91]].

2.2.1 Typen von Rechnernetzen

Rechnernetze lassen sich nach vielerlei Gesichtspunkten klassifizieren, z.B. nach dem verwendeten Übertragungsmedium (Kupferkabel, Glasfaserkabel oder drahtlose Übertragung über Satelliten), der Netzwerktopologie (Ring, Bus, Sternstruktur, vollvermaschtes Netz etc.), Punkt-zu-Punkt- vs. Mehrpunktverbindung*, Leitungs- vs. Paketvermittlung oder nach räumlicher Entfernung [[Ta88], [Ha92]].

* Mehrpunktverbindungen erlauben eine Nachricht gleichzeitig an mehrere Rechner zu übertragen, um z.B. eine Multicast- oder Broadcast-Übertragung zu realisieren.

Für unsere Überlegungen ist vor allem die Einteilung nach räumlicher Entfernung von Interesse, wobei man grob vier Kategorien unterscheiden kann [[GR93]]:

- Cluster
- Lokales Netzwerk (Local Area Network, LAN)
- Stadtnetz (Metropolitan Area Network, MAN)
- Weitverkehrsnetz (Wide Area Network, WAN).

Ein *Cluster* besteht aus mehreren räumlich benachbarten Rechnern, die typischerweise nur wenige Meter voneinander getrennt sind. Die Kommunikation erfolgt über ein Hochgeschwindigkeitsnetz mit hoher Übertragungsbandbreite und kurzen Nachrichtenlaufzeiten. Lokale Netze erlauben größere Entfernungen zwischen den Rechnern bis zu wenigen Kilometern, Stadtnetze von bis zu ca. 100 km. Darüber hinaus spricht man von Weitverkehrsnetzen, die auch mehrere Kontinente umfassen können. Cluster und LANs befinden sich i.a. im Eigentum eines einzelnen Betriebes, während MAN und WAN üblicherweise öffentliche Netze nutzen.

Die genannten Netzwerktypen weisen erhebliche Unterschiede bezüglich Leistungs- und Verfügbarkeitsmerkmalen auf. Grob gesagt nehmen sowohl Leistungsfähigkeit als auch Fehlertoleranz (gegenüber Leitungsausfall, Nachrichtenverlust, Nachrichtenduplizierung etc.) mit zunehmender geographischer Verteilung stark ab. Während innerhalb eines Clusters derzeit Übertragungsbandbreiten von ca. 1 Gbit/s erreicht werden, operieren typische LANs wie Ethernet oder Token-Ring zwischen 10 und 16 Mbit/s. Im LAN-Bereich erreicht der neuere FDDI-Standard auf Basis von Glasfaserverbindungen jedoch bereits 100 Mbit/s. Die Übertragungsgeschwindigkeit von Weitverkehrsnetzen liegt dagegen derzeit meist zwischen 10 Kbit/s und 2 Mbit/s [[Ha92]]. Die Übertragungsdauer im Weitverkehrsbereich ist jedoch nicht nur durch die (derzeit) relativ geringe Bandbreite beeinflußt, sondern auch durch die aufgrund der großen geographischen Entfernungen beträchtlichen Signallaufzeiten, die letztlich durch die Lichtgeschwindigkeit begrenzt sind. Die Übertragungsdauer einer Nachricht ist zudem noch stark vom CPU-Aufwand zur Abwicklung des Kommunikationsprotokolls abhängig.

Zur Verdeutlichung der Leistungssunterschiede sind in Abb. 2-7 Schätzwerte für die Latenzzeit (Signallaufzeit), Bandbreite sowie Übertragungsdauer einer Nachricht von 1 KB für die Jahre 1990 und 2000 zusammengestellt. Die Übertragungsdauer enthält dabei die durch die geographische Entfernung bestimmte Signallaufzeit sowie die durch die Bandbreite begrenzte Übertragungszeit als auch CPU-Belegungszeiten zur Abwickung des Kommunikationsprotokolls. Man erkennt, daß die Übertragungsbandbreite durch die Verwendung von Glasfaserverbindungen selbst im Weitverkehrsbereich in naher Zukunft stark erhöht wird und damit keinen leistungsbegrenzenden Faktor mehr darstellt. Dennoch werden Weitverkehrsnetze auch künftig signifikant langsamer als lokale Netze bleiben, da die

Übertragungszeiten zunehmend durch die Signallaufzeiten (Lichtgeschwindigkeit) dominiert werden [[GR93]]. In lokalen Netzen (Cluster, LAN) sind die Übertragungszeiten bereits heute nicht von der Bandbreite, sondern durch den CPU-Aufwand zur Kommunikationsabwicklung begrenzt. Hier sind jedoch durch die steigenden CPU-Geschwindigkeiten auch erhebliche Verbesserungen zu erwarten. Die CPU-Kosten für die Kommunikation im Weitverkehrsbereich liegen aufgrund der aufwendigeren Protokolle absolut deutlich höher als in lokalen Netzen (z.B. 50,000 Instruktionen pro Sendevorgang für WAN vs. 1000-10000 Instruktionen für Cluster oder LAN).

Abb. 2-7: Kennzeichnende Leistungsmerkmale von Rechnernetzen [[GR93]]

	Durchmesser	Latenzzeit	Bandbreite (Mbit/s)		Übertragung 1 KB	
			1990	2000	1990	2000
Cluster	100 m	0,0005 ms	1000	1000	10 ms	0,005 ms
LAN	1 km	0,005 ms	10	1000	1 ms	0,01 ms
MAN	100 km	5 ms	1	100	10 ms	0,6 ms
WAN	10.000 km	50 ms	0,05	100	210 ms	50 ms

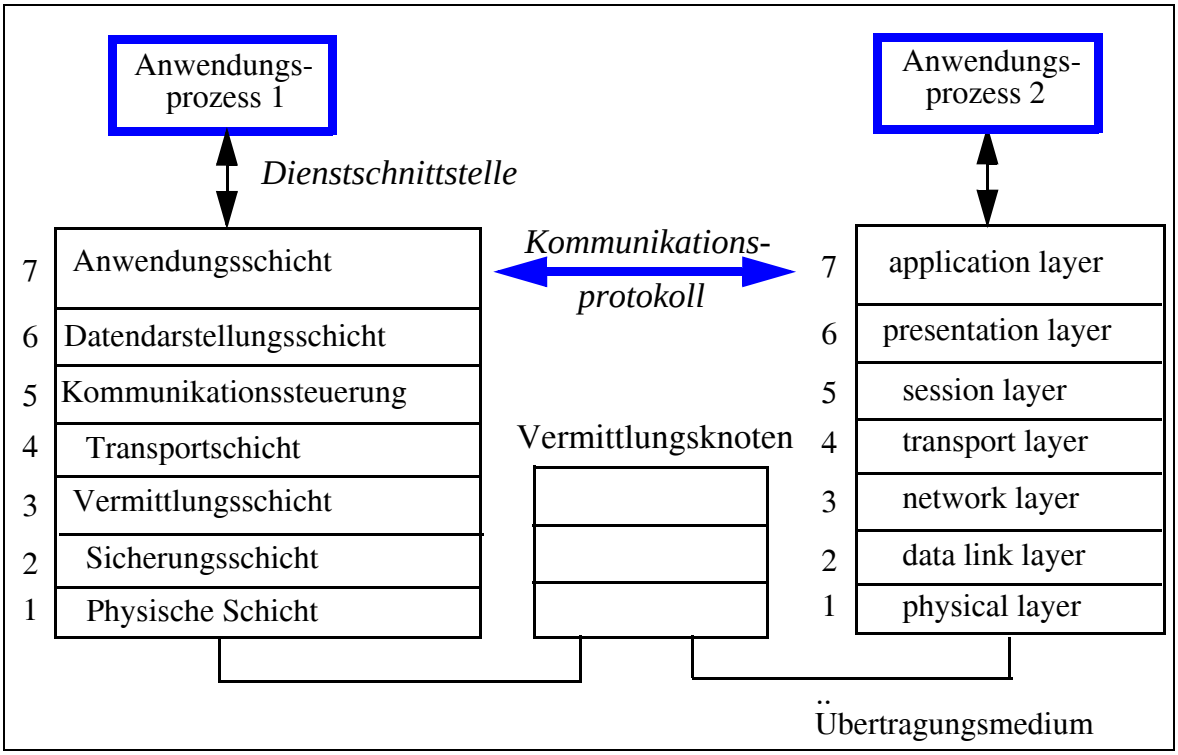
2.2.2 ISO-Referenzmodell

Der Nachrichtenaustausch zwischen zwei Kommunikationspartnern erfolgt nach festgelegten Kommunikationsprotokollen, welche regeln, wie die einzelnen Nachrichten aufgebaut sind und in welcher Reihenfolge sie verschickt werden können. Üblicherweise kommt eine Hierarchie von Protokollen zum Einsatz, die innerhalb einer *Kommunikationsarchitektur* angeordnet sind. Solche Architekturen wurden sowohl herstellerneutral (ISO OSI, TCP/IP etc.) als auch von einzelnen Herstellern für ihre Plattformen definiert (z.B. SNA von IBM, DNA von DEC, TRANSDATA von SNI etc.). Den Kommunikationsarchitekturen ist gemein, daß sie als Schichtenmodelle realisiert sind. Die einzelnen Kommunikationsprotokolle regeln dabei jeweils die Kommunikation zwischen Instanzen derselben Schicht. Zur Realisierung eines Protokolls der Schicht i+1 werden die Dienste der Schicht i genutzt, ohne die Realisierungseinzelheiten der darunterliegenden Schichten kennen zu müssen.

Die bekannteste herstellerunabhängige Kommunikationsarchitektur ist das OSI (Open Systems Interconnection) 7-Schichten-Modell der ISO (International Organization for Standardization). Diese Architektur bildet ein Referenzmodell, mit dem nur die wichtigsten funktionalen Eigenschaften der einzelnen Schichten festgelegt werden. Die Dienste der einzelnen Schichten sowie die Schichtenprotokolle sind damit noch nicht definiert, sondern Gegenstand einer eigenen Standardisie-

rung. Die Kommunikation verläuft i.a. "verbindungsorientiert", d.h. es sind zunächst logische Verbindungen zwischen den Kommunikationspartnern aufzubauen, über die dann die Nachrichten ausgetauscht werden*. Nach Beendigung der Datenübertragung werden die Verbindungen wieder abgebaut.

Abb. 2-8: Die 7 Schichten des ISO-Referenzmodells



Wie Abb. 2-8 zeigt, kommen Anwendungsprozesse, die miteinander kommunizieren möchten, nicht direkt mit den Kommunikationsprotokollen in Berührung. Vielmehr veranlassen sie die Kommunikation durch den Aufruf von Funktionen der Dienstschnittstelle einer bestimmten Schicht (i.a. der Anwendungsschicht, Schicht 7)**. Die Dienste der obersten Schicht werden danach sukzessive in Aufrufe der darunterliegenden Schichten umgesetzt bis zur physischen Übertragung der Nachrichten über das jeweilige Medium. Im Empfangsrechner wird der Abbildungsprozeß in umgekehrter Richtung vorgenommen. Die sieben Schichten des

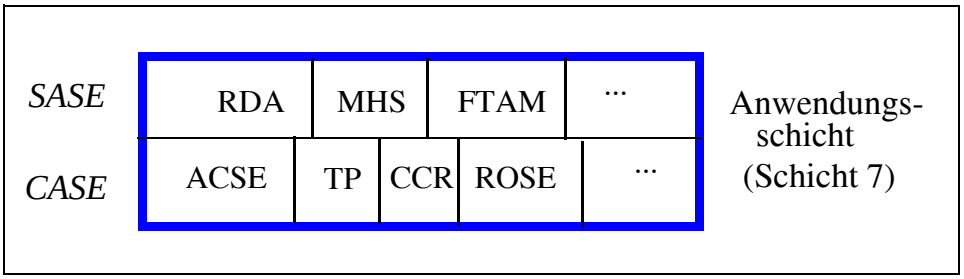
* Der Aufbau der Verbindungen führt zu einem relativ hohen Aufwand im Gegensatz zu einer "verbindungslosen" Kommunikation über sogenannte Datagramme. Dafür kann die Kommunikation über eine bereits erstellte Verbindung schneller ablaufen, was insbesondere in Weitverkehrsnetzen von Bedeutung ist. Weiterhin bieten verbindungsorientierte Protokolle eine höhere Fehlertoleranz hinsichtlich Nachrichtenverlust oder Überholvorgängen.

** Anwendungsprogramme rufen dabei - im Gegensatz zu Systemprogrammen wie einem TP-Monitor - i.a. auch nicht direkt die einzelnen Dienste auf, sondern Funktionen eines allgemeineren API (Application Programming Interface). Da die API-Funktionen auf unterschiedliche Kommunikationsarchitekturen abgebildet werden können, ergibt sich eine höhere Unabhängigkeit und Portierbarkeit von Anwendungen.

Modells sind dabei nur in den Endknoten vollständig repräsentiert, während in Zwischen- bzw. Vermittlungsknoten lediglich die drei untersten Schichten vorliegen. Dies ist ausreichend, da diese Knoten lediglich zur Vermittlung (Aufbau, Aufrechterhaltung und Abbau von Verbindungen) sowie zur reinen Datenübertragung dienen.

Hier soll auf eine genauere Funktionsbeschreibung der einzelnen Schichten verzichtet werden (s. [[Ta88], [LKK93]]), da sie für das weitere Verständnis nicht benötigt wird. Für unsere Zwecke sind jedoch einige Standardisierungsanstrengungen der Anwendungsebene (Schicht 7) von Interesse, welche die Transaktions- und Datenbankverarbeitung in verteilten Systemen betreffen. Wie in Abb. 2-9 gezeigt, kann die Anwendungsebene selbst wiederum in zwei Schichten unterteilt werden. In der oberen Teilschicht befinden sich dabei spezifische Anwendungsdienste (SASE, Specific Application Service Elements), welche allgemeinere Dienste der unteren Teilschicht verwenden. Spezifische Anwendungsdienste sind u.a. RDA (Remote Database Access), FTAM (File Transfer, Access, and Management), MHS (Message Handling Systems), VTP (Virtual Terminal Protocol) und DS (Directory Service). Die untere Teilschicht enthält allgemeinere Anwendungsdienste (CASE, Common Application Service Elements), welche von mehreren der übergeordneten Dienste benötigt werden. Solche Standards sind u.a. ACSE (Association Control Service Element), TP (Transaction Processing), CCR (Commitment, Concurrency, and Recovery) und ROSE (Remote Operations Service Element).

Abb. 2-9: ISO-Standards der Anwendungsebene



RDA ermöglicht dabei die Verteilung von DB-Operationen in heterogenen Umgebungen und wird in Kap. 11.4.3 näher vorgestellt. TP unterstützt die Initiierung und Abwicklung verteilter Transaktionen. Das hierbei notwendige verteilte Commit-Protokoll zur Gewährleistung der Atomarität verteilter Transaktionen wird mit Hilfe der CCR-Komponente realisiert. ACSE bietet Funktionen zum Auf- und Abbau logischer Verbindungen (Assoziationen).

3 Klassifikation von Mehrrechner-Datenbanksystemen

Unter der Bezeichnung "Mehrrechner-Datenbanksysteme" sollen sämtliche Architekturen zusammengefaßt werden, bei denen mehrere Prozessoren oder DBVS-Instanzen an der Verarbeitung von DB-Operationen beteiligt sind. Dabei ist natürlich eine Kooperation der Prozessoren bzw. DBVS bezüglich der DB-Verarbeitung bestimmter Anwendungen vorzusehen, um den Fall voneinander isoliert arbeitender DBVS oder Prozessoren auszuschließen. Die eingeführte Definition ist sehr allgemein und läßt eine Vielzahl von generellen Realisierungsalternativen zu*, welche in diesem Kapitel klassifiziert werden. Unser Klassifikationsschema umfaßt im wesentlichen folgende Kriterien:

- *Funktionale Spezialisierung vs. Gleichstellung*
  Diese Unterscheidung legt fest, ob unter den zur DB-Verarbeitung vorgesehenen Prozessoren eine funktionale Spezialisierung erfolgt oder nicht. Bei funktionaler Gleichstellung besitzt jeder Prozessor die gleiche Funktionalität hinsichtlich der Ausführung von DBVS-Funktionen, ansonsten erfolgt eine Aufteilung der DBVS-Funktionalität unter verschiedene Prozessoren. In [Ra93a] bezeichneten wir funktional spezialisierte Ansätze als *vertikal verteilte Mehrrechner-DBS;* funktionale Gleichstellung dagegen als *horizontal verteilte Mehrrechner-DBS.*
- *Externspeicheranbindung (gemeinsam vs. partitioniert)*
- *Räumliche Verteilung (lokal vs. ortsverteilt)*
- *Rechnerkopplung (eng, nahe, lose)*
- *Integrierte vs. föderative Mehrrechner-DBS*
- *Homogene vs. heterogene DBVS.*

---

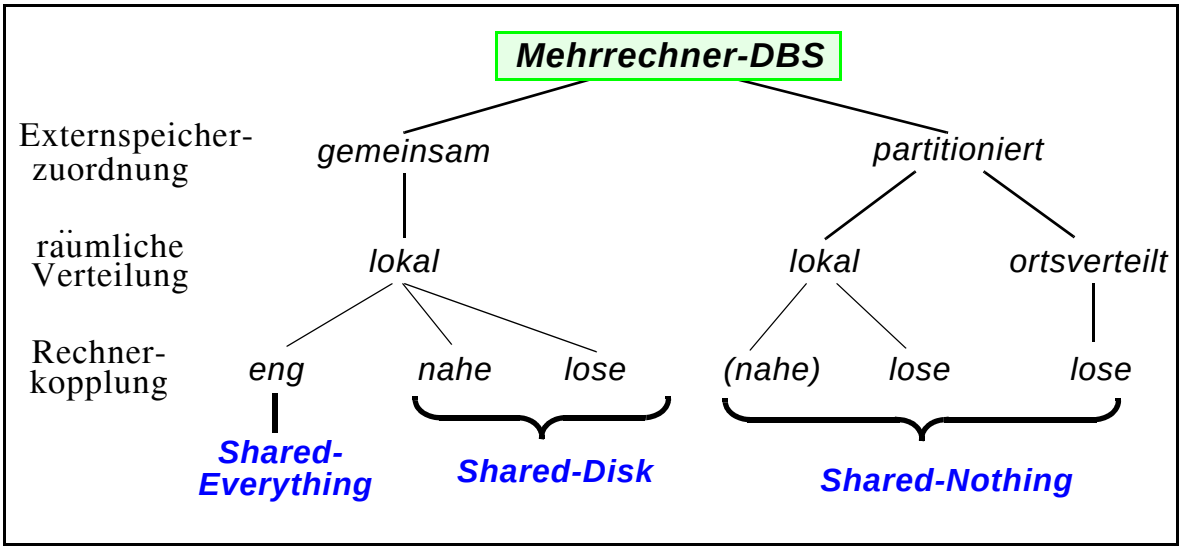
* Insbesondere ist der Spezialfall mit nur einem DBVS und mehreren Prozessoren enthalten, welcher die Klasse der Shared-Everything-Systeme kennzeichnet. Prinzipiell ist es auch denkbar, daß nur ein Prozessor und mehrere (kooperierende) DBVS vorliegen. Dies ist zwar i.a. wenig sinnvoll, jedoch wären auch hierbei (innerhalb eines Rechners) die Implementierungsprobleme von Mehrrechner-DBS zu lösen.

Da die gleichrangige Behandlung dieser sechs Merkmale zu einer recht unübersichtlichen Strukturierung führt, erfolgt unsere Klassifikation im folgenden in drei Stufen. Dabei konzentrieren wir uns zunächst auf horizontal verteilte Mehrrechner-DBS (funktionale Gleichstellung). Anhand der Kriterien bezüglich der Rechnerorganisation (Externspeicheranbindung, räumliche Verteilung, Rechnerkopplung) werden hierzu im ersten Schritt (Kap. 3.1) drei wesentliche Klassen von Mehrrechner-DBS definiert: Shared-Everything, Shared-Disk sowie Shared-Nothing. Mit den beiden letzten (DBS-spezifischen) Klassifikationsmerkmalen wird darauf aufbauend in Kap. 3.2 die Unterscheidung zwischen integrierten und föderativen sowie homogenen und heterogenen Mehrrechner-DBS eingeführt. Funktional spezialisierte (vertikal verteilte) Mehrrechner-DBS werden danach in Kap. 3.3 diskutiert, insbesondere Workstation/Server-DBS. In Kap. 3.4 führen wir dann einen Vergleich zwischen den eingeführten Typen von Mehrrechner-DBS hinsichtlich der zu erfüllenden Anforderungen (Kap. 1.2) durch.
Es sei hier bereits bemerkt, daß eine verteilte DB- und Transaktionsverarbeitung auch ohne direkte Kooperation mehrerer DBS realisiert werden kann, z.B. durch TP-Monitore. In diesem Fall spricht man von verteilten Transaktionssystemen, die wir in diesem Buch im Hinblick auf die Unterstützung heterogener Datenbanken behandeln werden (Kap. 11).

3.1 Shared-Everything, Shared-Disk, Shared-Nothing

Eine erste Grobklassifikation von (horizontal verteilten) Mehrrechner-DBS ergibt sich durch die Verwendung der drei Kriterien Externspeicherzuordnung, räumliche Verteilung sowie Rechnerkopplung, die im folgenden näher diskutiert werden. Wie Abb. 3-1 zeigt, lassen sich hiermit drei Klassen von Mehrrechner-DBS unterscheiden, nämlich Shared-Everything, Shared-Disk (DB-Sharing) sowie Shared-Nothing (DB-Distribution). Nach der Diskussion der Klassifikationsmerkmale erfolgt in Kap. 3.1.2 eine Gegenüberstellung der bei diesen drei Ansätzen zu behandelnden Implementierungsprobleme.

Abb. 3-1: Grobklassifikation von Mehrrechner-DBS



3.1.1 Klassifikationsmerkmale

Externspeicheranbindung

Bezüglich der Externspeicheranbindung unterscheiden wir zwischen partitioniertem und gemeinsamem Zugriff. Beim *partitioniertem Zugriff*, der die Klasse der Shared-Nothing-Systeme kennzeichnet, erfolgt eine Partitionierung der Externspeicher unter den Prozessoren bzw. Rechnern. Im Falle von Magnetplatten als Externspeicher bedeutet dies, daß jedes Plattenlaufwerk und die darauf gespeicherten Daten genau einem Rechner zugeordnet ist. Jeder Rechner hat nur auf seine Externspeicherpartition direkten Zugriff. Der Zugriff auf Daten anderer Partitionen erfordert Kommunikation mit dem Besitzer der jeweiligen Partition, was zu einer verteilten Ausführung der jeweiligen DB-Operation bzw. Transaktion führt. Die Partitionierung der Plattenperipherie impliziert jedoch nicht notwendigerweise die Partitionierung der Datenbank. Vielmehr kann diese auch teilweise oder vollkommen repliziert an mehreren bzw. allen Rechnern gespeichert werden. Eine solche Replikation kann zur Einsparung von Kommunikationsvorgängen für Lesezugriffe sowie zur Erhöhung der Fehlertoleranz genutzt werden, allerdings auf Kosten eines erhöhten Speicherplatzbedarfs und Änderungsaufwandes.

Beim *gemeinsamen Zugriff* hat jeder Prozessor direkten Zugriff auf alle Platten und damit auf die gesamte Datenbank. Damit entfällt die Notwendigkeit einer verteilten Transaktionsverarbeitung wie bei der partitionierten Plattenanbindung, jedoch sind ggf. Synchronisationsmaßnahmen für den Externspeicherzugriff vorzusehen. Der gemeinsame Externspeicherzugriff liegt bei Shared-Disk- sowie Shared-Everything-Architekturen vor.

Räumliche Anordnung

Wir unterscheiden hier nur grob zwischen lokaler und ortsverteilter Rechneranordnung. Von den in Kap. 2.2 eingeführten Rechnernetzen fallen Cluster unter lokale, WANs unter ortsverteilte Anordnung. Für LANs und MANs ist eine eindeutige Zuordnung schwierig, da sie bereits ein breites Spektrum an geographischer Entfernung abdecken und auch bezüglich der Leistungsfähigkeit starke Unterschiede bestehen (z.B. Ethernet/Token-Ring vs. FDDI).

Generell gilt, daß lokal verteilte Systeme eine wesentlich leistungsfähigere Interprozessor-Kommunikation ermöglichen sowie robuster gegenüber Fehlern im Kommunikationssystem sind. Die effiziente Kommunikation ist ein entscheidender Vorteil lokal verteilter Mehrrechner-DBS gegenüber ortsverteilten Ansätzen. Dies ist vor allem auch für die parallele Datenbankverarbeitung von Bedeutung, bei der Transaktionen bzw. DB-Operationen zur Verkürzung der Bearbeitungszeiten intern zerlegt und mehreren Prozessoren zugeordnet werden, da diese Parallelisierung zwangsweise einen Mehrbedarf an Kommunikation mit sich bringt. Auch kann in lokal verteilten Systemen eine dynamische Verteilung und Balancierung der Transaktionslast unter Berücksichtigung des aktuellen Systemzustands eher erfolgen, da die Wartung entsprechender Statusinformationen mit geringerem Aufwand und größerer Aktualität als bei Ortsverteilung möglich ist. Lokal verteilte Mehrrechner-DBS weisen daneben Vorteile hinsichtlich der Administration auf gegenüber geographisch verteilten Systemen, bei denen i.a. in jedem Knoten eine eigene Systemverwaltung erforderlich wird. Wir werden lokal verteilte Systeme vom Typ Shared-Nothing, Shared-Disk und Shared-Everything auch als *Parallele Datenbanksysteme* bezeichnen (Kap. 16).

Auf der anderen Seite kann nur mit ortsverteilten Mehrrechner-DBS eine Anpassung an verteilte Organisationsstrukturen erfolgen, wie es in großen Unternehmen vielfach wünschenswert ist. Auch bieten ortsverteilte Konfigurationen eine größere Fehlertoleranz gegenüber "Katastrophen" (z.B. Bombenanschlag, Überschwemmung, Erdbeben), welche den Ausfall eines kompletten Rechenzentrums bzw. Clusters mit entsprechendem Datenverlust verursachen können (s. Kap. 9.5).

Für Shared-Everything bzw. Shared-Disk schreibt die gemeinsame Externspeicheranbindung in der Regel eine lokale Rechneranordnung vor. Größere Entfernungen zwischen Rechnern und Platten sind bei Glasfaserkopplung bedingt möglich (derzeit bis zu ca. 30 km), jedoch zu deutlich erhöhten Latenzzeiten. Eine solche Form der Verteilung kann zur Steigerung der Verfügbarkeit genutzt werden, da die Externspeicher selbst den Ausfall sämtlicher Verarbeitungsrechner überleben können. Auch können sich ökonomische Vorteile ergeben, z.B. indem die oft umfangreichen Plattenfarmen an Orten mit günstigen Mietpreisen gehalten werden. Allgemeine dezentrale Organisationsstrukturen werden damit jedoch nicht unterstützt, da hierzu neben einer ortsverteilten Rechneranordnung (LAN, MAN oder WAN) eine partitionierte Externspeicherzuordnung zur Wahrung einer mög-

lichst hohen Unabhängigkeit (Knotenautonomie) einzelner Abteilungen vorzusehen ist.
Shared-Nothing-Systeme gestatten dagegen aufgrund der partitionierten Externspeicheranbindung sowohl eine lokale als auch eine ortsverteilte Rechneranordnung. *Verteilte Datenbanksysteme* repräsentieren einen bekannten Vertreter geographisch verteilter Shared-Nothing-Systeme.

Rechnerkopplung

Die bekanntesten Ansätze, mehrere Prozessoren miteinander zu verbinden, sind die enge und lose Rechnerkopplung. Bei der ***engen Kopplung*** (Multiprozessor-Ansatz) teilen sich die Prozessoren einen gemeinsamen Hauptspeicher (Abb. 3-2a); Software wie Betriebssystem, DBVS oder Anwendungsprogramme liegen nur in einer Kopie vor. Die Nutzung solcher Systeme zur DB-Verarbeitung bezeichnet man als *Mehrprozessor-DBS* bzw. als *"Shared-Memory"*- bzw. "*Shared-Everything"*-Ansatz* [St86, HR86, Bh88]. Dieser Ansatz gestattet eine effiziente Kooperation zwischen Prozessoren über gemeinsame Hauptspeicher-Datenstrukturen. Eine effektive Lastbalancierung wird i.a. bereits durch das Betriebssystem unterstützt, indem gemeinsame Auftragswarteschlangen im Hauptspeicher gehalten werden, auf die alle Prozessoren zugreifen können.

Allerdings bestehen Verfügbarkeitsprobleme, da der gemeinsame Hauptspeicher nur eine geringe Fehlerisolation bietet und Software wie das Betriebssystem oder das DBVS nur in einer Kopie vorliegt [Ki84]. Auch die Skalierbarkeit ist i.a. stark begrenzt, da mit wachsender Prozessoranzahl der Hauptspeicher leicht zum Engpaß wird. Die Verwendung großer Prozessor-Caches erlaubt zwar eine Reduzierung von Hauptspeicherzugriffen und damit prinzipiell eine bessere Erweiterbarkeit, jedoch führen diese Caches zu einem Kohärenz- oder Invalidierungsproblem [St90]. Denn da nun Speicherinhalte repliziert in den privaten Cache-Speichern der Prozessoren gehalten werden, führen Änderungen in einem der Caches zu veralteten Daten in den anderen Caches.
Die erforderliche Kohärenzkontrolle führt jedoch in der Regel zu vermehrten Hauptspeicherzugriffen und Kommunikationsvorgängen, wodurch die Erweiterbarkeit oft wiederum stark beeinträchtigt wird. Schließlich wird die Erweiterbarkeit oft durch die für den Zugriff auf gemeinsame Hauptspeicher-Datenstrukturen notwendige Synchronisation zwischen Prozessen (z.B. über Semaphore) eingeschränkt, da die Konfliktwahrscheinlichkeit mit der Anzahl zugreifender Prozesse/Prozessoren zunimmt. Aus diesen Gründen sind derzeitige Multiprozessoren meist auf 2 bis 30 Prozessoren beschränkt, wobei bereits ab 10 Prozessoren i.d.R. nur noch geringe Leistungssteigerungen erzielt werden. Die mit Multiprozessoren

* Die Bezeichnung "Shared Everything" geht darauf zurück, daß neben dem Hauptspeicher auch Terminals sowie Externspeicher von allen Prozessoren erreichbar sind.

effektiv nutzbare CPU-Kapazität ist somit relativ beschränkt und für Hochleistungssysteme vielfach nicht ausreichend.

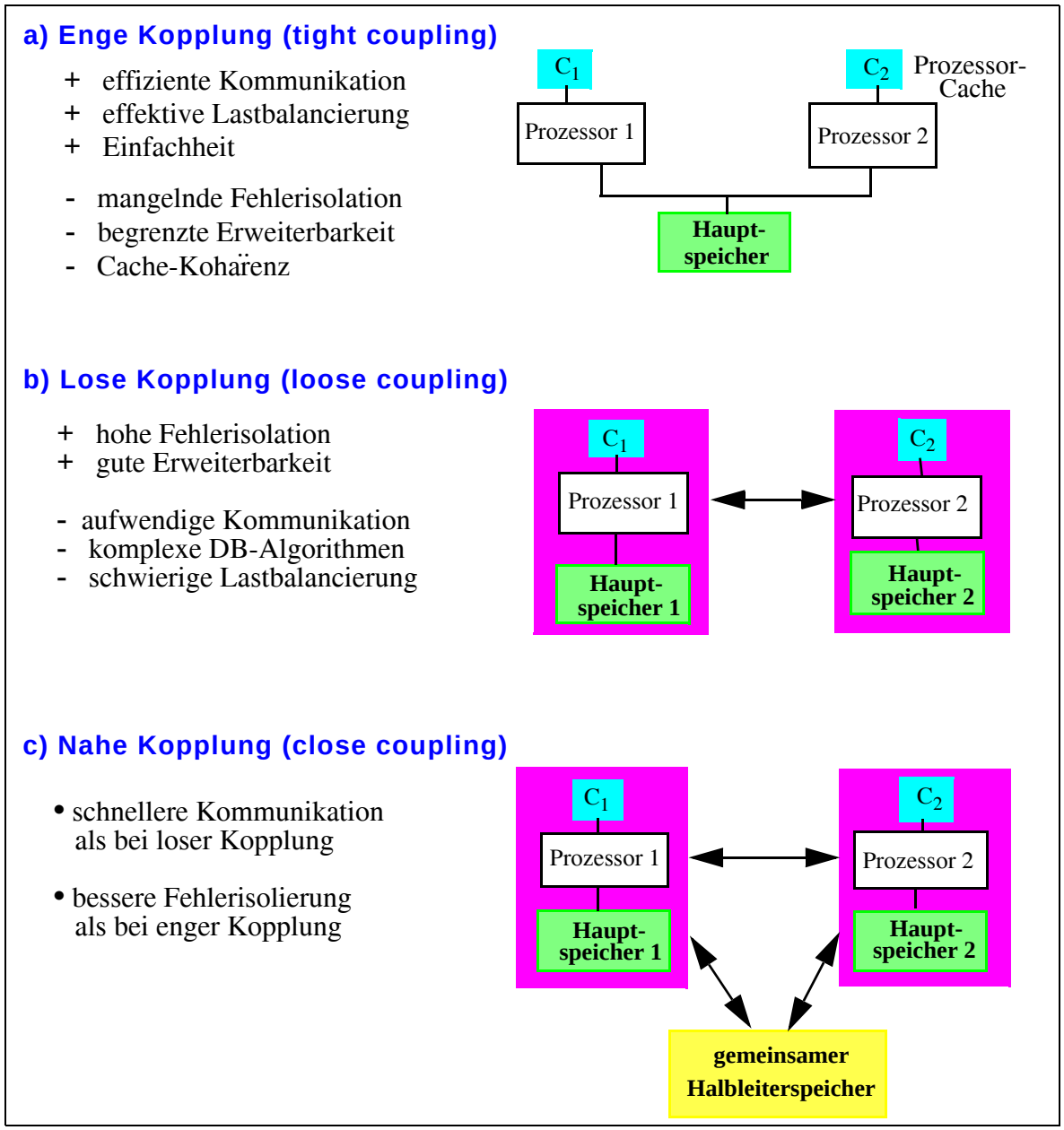
Bei der ***losen Rechnerkopplung*** (Abb. 3-2b) kooperieren mehrere unabhängige Rechner, die jeweils einen eigenen Hauptspeicher besitzen sowie private Software-Kopien (DBVS, Betriebssystem, Anwendungen). Dies ermöglicht eine weit bessere Fehlerisolation als bei enger Kopplung. Das Fehlen eines gemeinsamen Hauptspeichers führt auch zu einer besseren Erweiterbarkeit. Es lassen sich somit von der Gesamt-CPU-Kapazität her gesehen wesentlich leistungsfähigere Konfigurationen realisieren als bei enger Kopplung, insbesondere da bei der losen Kopplung jeder Rechnerknoten selbst wieder ein Multiprozessor sein kann. Hauptnachteil ist die aufwendige Kommunikation, die bei der losen Kopplung ausschließlich durch Nachrichtenaustausch über ein Verbindungsnetzwerk erfolgt (die Diskussion von Rechnernetzen in Kap. 2.2 bezog sich ausschließlich auf lose gekoppelte Architekturen). Die Kommunikationsprimitive zum Übertragen und Empfangen von Nachrichten sind selbst bei lokaler Verteilung meist sehr teuer; zusätzlich entstehen Prozeßwechselkosten, da Kommunikationsverzögerungen von typischerweise mehreren Millisekunden kein synchrones Warten auf eine Antwortnachricht zulassen.

Für lose gekoppelte Mehrrechner-DBS liegt in jedem Rechner eine Kopie des DBVS vor, die zur koordinierten DB-Verarbeitung miteinander kommunizieren. Damit ist im Gegensatz zu Shared-Everything, wo das Betriebssystem bereits die Existenz mehrerer Prozessoren weitgehend verbergen kann, die Verteilung im DBVS sichtbar, was zu einer aufwendigen Realisierung der DBVS führt. Welche Aufgaben Kommunikation erfordern, ist durch den gewählten Architekturtyp, Shared-Nothing oder Shared-Disk, bestimmt (s.u.). Auch die Realisierung einer effektiven Lastbalancierung ist weit schwieriger als bei enger Kopplung, da bei der Lastzuordnung neben einer Vermeidung von Überlastsituationen auch eine Minimierung von Kommunikationsvorgängen bei der DB-Verarbeitung unterstützt werden sollte.

Ziel einer ***nahen Rechnerkopplung*** [HR86, Ra86a] ist es, die Vorteile von loser und enger Kopplung miteinander zu verbinden. Wie bei der losen Kopplung besitzen die am Systemverbund beteiligten Rechner einen eigenen Hauptspeicher sowie separate Kopien der Software. Um die Kommunikation der Rechner effizienter realisieren zu können, ist jedoch die Nutzung gemeinsamer Systemkomponenten vorgesehen. Wie in Abb. 3-2c angedeutet, eignen sich hierzu besonders gemeinsame Halbleiterspeicher, über die ein schneller Nachrichten- bzw. Datenaustausch möglich werden soll. Desweiteren können globale Kontrollaufgaben ggf. erheblich vereinfacht werden, wenn statt eines verteilten Protokolls globale Datenstrukturen in einem solchen Speicher genutzt werden können. Aus Leistungsgründen ist es wünschenswert, wenn auf solch gemeinsame Speicher über spezielle Maschinenbefehle synchron zugegriffen werden kann, was Zugriffszeiten von wenigen Mikrosekun-

den voraussetzt*. Durch den Wegfall der Prozeßwechselkosten können dann enorme Einsparungen gegenüber einer losen Kopplung erwartet werden.

Abb. 3-2: Alternativen zur Rechnerkopplung [Ra93a]



Natürlich birgt der gemeinsame Halbleiterspeicher ähnliche Verfügbarkeits- und Skalierbarkeitsprobleme wie ein gemeinsamer Hauptspeicher bei enger Kopplung. Eine größere Fehlerisolation als bei enger Kopplung wird jedoch bereits dadurch unterstützt, daß die einzelnen Rechner weitgehend unabhängig voneinan-

* Eine Speicherzugriffszeit von z.B. 100 µs entspricht bei einer CPU-Leistung von 50 MIPS bereits einem Overhead von 5000 Instruktionen. Je schneller die Prozessoren werden, desto schneller muß der Speicherzugriff sein, um einen synchronen Zugriff zu gestatten.

der sind (private Hauptspeicher) und der gemeinsame Speicherbereich nur für globale Kontrollaufgaben genutzt wird. Weiterhin verlangen wir, daß der gemeinsame Speicher nicht instruktionsadressierbar sein darf [Ra86a], sondern die Auswertung und Manipulation von Speicherinhalten innerhalb der privaten Hauptspeicher durchzuführen ist. Desweiteren ist durch entsprechende Recovery-Protokolle zu gewährleisten, daß sich Software- und Hardware-Fehler einzelner Verarbeitungsrechner nicht über den gemeinsamen Speicherbereich ausbreiten und daß ein Ausfall des gemeinsamen Speichers schnell behandelt wird. Inwieweit Erweiterbarkeitsprobleme drohen, hängt vor allem von der Nutzung des gemeinsamen Speichers ab. Generell ist vorzusehen, daß eine Kommunikation wie bei loser Kopplung weiterhin möglich ist, und der gemeinsame Speicherbereich nur für spezielle, besonders leistungskritische Aufgaben verwendet wird, um so die Engpaßgefahr gering zu halten. Die Nutzung eines gemeinsamen Speichers verursacht natürlich Zusatzkosten, wobei zur Wahrung der Kosteneffektivität entsprechende Leistungsverbesserungen erreicht werden müssen.

Die nahe Kopplung schreibt eine lokale Rechneranordnung vor. Sie ist prinzipiell bei Shared-Disk und Shared-Nothing anwendbar. Jedoch ergeben sich für Shared-Nothing geringere Einsatzmöglichkeiten gemeinsamer Halbleiterspeicher, da dieser Architekturtyp auf eine Partitionierung von Haupt- und Externspeicher ausgerichtet ist. Allerdings könnte bei lokaler Rechneranordnung der Nachrichtenaustausch auch bei Shared-Nothing über gemeinsame Halbleiterspeicher abgewickelt werden. Eine solche allgemeine Einsatzform der nahen Kopplung wird daher auch sinnvollerweise bereits im Betriebssystem realisiert und bleibt daher für die beteiligten DBVS transparent. Wie in Kap. 13.4 gezeigt wird, ergeben sich für Shared-Disk weitergehende Einsatzformen einer nahen Kopplung.

3.1.2 Technische Probleme

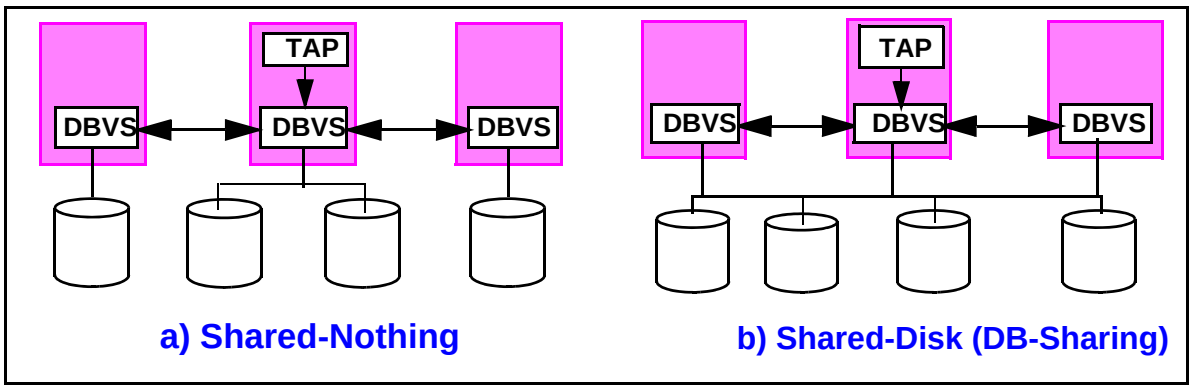
Die vorgestellten Klassifikationsmerkmale ergeben die Unterscheidung zwischen drei verbreiteten Typen von Mehrrechner-DBS (Abb. 3-1), die zusammenfassend wie folgt charakterisiert werden können:

- **Shared-Everything** (Multiprozessor-DBS):
  Die DB-Verarbeitung erfolgt durch ein DBVS auf einem Multiprozessor.
- **Shared-Nothing** (Database Distribution, DB-Distribution, Abb. 3-3a):
  Die DB-Verarbeitung erfolgt durch mehrere, i.a. lose gekoppelte Rechner, auf denen jeweils ein DBVS abläuft. Die Externspeicher sind unter den Rechnern partitioniert, so daß jedes DBVS nur auf Daten der lokalen Partition direkt zugreifen kann. Die Rechner können lokal oder geographisch verteilt angeordnet sein. Multiprozessoren sind als Rechnerknoten möglich, das heißt, die lokalen DBVS können vom Typ "Shared-Everything" sein.
- **Shared-Disk** (Database Sharing, DB-Sharing, Abb. 3-3b):
  Wie bei Shared-Nothing liegt eine Menge von Verarbeitungsrechnern mit je einem DBVS vor, wobei jeder Rechner ein Multiprozessor sein kann. Es liegt eine gemein-

same Externspeicherzuordnung vor, so daß jedes DBVS direkten Zugriff auf die gesamte Datenbank hat. Die Rechner sind i.a. lokal innerhalb eines Clusters verteilt, wobei entweder eine lose oder nahe Rechnerkopplung möglich ist.

Von diesen Ansätzen ist die Realisierung von Mehr- oder ***Multiprozessor-DBS*** (Shared-Everything) mit Abstand am einfachsten, da das Betriebssystem die Verteilung bereits weitgehend verbirgt ("single system image"). Folglich ergeben sich kaum Änderungen bezüglich den in Kap. 2.1.2 und Kap. 2.1.3 vorgestellten Funktionen zentralisierter DBVS. Allerdings verlangt die volle Nutzung eines Multiprozessors eine DB-Verarbeitung in mehreren Prozessen. Der DBVS-Code sowie die Datenstrukturen des DBVS (insbesondere DB-Puffer und Sperrtabelle) werden im gemeinsamen Hauptspeicher für alle DBVS-Prozesse zugänglich gehalten. Insbesondere für den Zugriff auf die DBVS-Datenstrukturen ist eine Synchronisation zwischen den Prozessen durchzuführen, z.B. über Semaphore [Hä79]. Weiterhin ist eine Zuordnung abzuarbeitender DB-Operationen zu den DBVS-Prozessen vorzunehmen (Lastbalancierung). Dies läßt sich z.B. durch Verwendung einer gemeinsamen Auftragswarteschlange für DB-Operationen im Hauptspeicher geeignet realisieren, auf die alle DBVS-Prozesse Zugriff haben. Daneben sind Erweiterungen bei der Query-Optimierung erforderlich, wenn eine Parallelarbeit innerhalb von Transaktionen auf mehreren Prozessoren unterstützt werden soll.
Die meisten kommerziellen DBS unterstützen den Shared-Everything-Ansatz, das heißt, sie können Multiprozessoren zur DB-Verarbeitung nutzen.

Abb. 3-3: Shared-Nothing vs. Shared-Disk



Der ***Shared-Nothing-Ansatz*** erfordert, die Daten (Datenbank) unter mehrere Rechner bzw. DBVS aufzuteilen. Hierzu sind zunächst die Einheiten der Datenverteilung oder Fragmente festzulegen. Danach wird im Rahmen der Allokation bestimmt, welchen Rechnern die Fragmente zugeordnet werden, wobei im Falle replizierter Datenbanken einzelne Fragmente mehreren Rechnern zugeteilt werden können. Die Bearbeitung von DB-Operationen muß auf die vorgenommene Datenverteilung abgestimmt werden, da jedes DBVS nur auf Daten der lokalen Partition zugreifen kann. Dies erfordert die Erstellung verteilter bzw. paralleler Ausführungspläne durch das DBVS (Query-Übersetzung und -Optimierung). Dar-

über hinaus ist durch die DBVS ein rechnerübergreifendes Commit-Protokoll zu unterstützen, um die Atomarität verteilter Transaktionen zu gewährleisten. Weitere technische Probleme betreffen die Verwaltung von Katalogdaten, die Behandlung globaler Deadlocks sowie die Wartung replizierter Datenbanken. Die Lösung dieser Probleme wurde vor allem für Verteilte Datenbanksysteme intensiv untersucht, die im Teil II dieses Buchs ausführlich behandelt werden.

Die großen Forschungsanstrengungen bezüglich Verteilter Datenbanksysteme führten zur Entwicklung zahlreicher Prototypen sowie einiger kommerzieller Produkte (Kap. 19). Implementierungen lokal verteilter Shared-Nothing-Systeme (Bsp.: Teradata) wurden vor allem im Hinblick auf die Realisierung von Hochleistungssystemen entwickelt, insbesondere zur parallelen Bearbeitung komplexer DB-Anfragen.

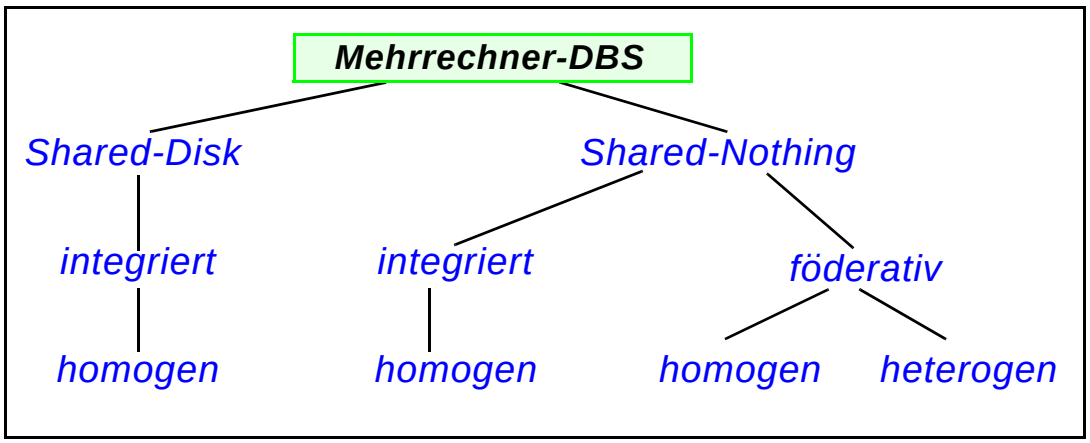
In ***Shared-Disk-DBS*** ist aufgrund der gemeinsamen Externspeicherzuordnung keine physische Datenaufteilung unter den Rechnern vorzunehmen. Insbesondere können sämtliche DB-Operationen einer Transaktion innerhalb eines Rechners abgearbeitet werden, so daß keine verteilten Ausführungspläne und kein verteiltes Commit-Protokoll zu unterstützen sind. Kommunikation zwischen den DBVS wird v.a. zur globalen Synchronisation der DB-Zugriffe notwendig. Ein weiteres Problem stellt die Behandlung sogenannter Pufferinvalidierungen dar, die sich dadurch ergeben, daß jedes DBVS Seiten der gemeinsamen Datenbank in seinem lokalen Hauptspeicherpuffer hält. Weitere technische Probleme betreffen Logging und Recovery sowie Lastverteilung und -balancierung. Die Realisierung dieser Funktionen wird im Teil IV dieses Buches ausführlich behandelt.

Der Shared-Disk-Ansatz hat in der Forschung im Vergleich mit Shared-Nothing (und Verteilten DBS) ein eher geringes Interesse gefunden. Dies steht jedoch im Gegensatz zur Bedeutung dieses Ansatz in der Praxis, wo er von einer zunehmenden Anzahl von DBS unterstützt wird (s. Kap. 19).

3.2 Integrierte vs. föderative Mehrrechner-DBS

Eine weitergehende Klassifizierung ergibt sich durch die Unterscheidung zwischen integrierten und föderativen sowie zwischen homogenen und heterogenen Mehrrechner-DBS (Abb. 3-4). Diese Merkmale sind nur für Architekturen mit mehreren DBVS relevant, also nicht für Shared-Everything.

Abb. 3-4: Weitergehende Klassifikation von Mehrrechner-Datenbanksystemen



Integrierte Mehrrechner-DBS sind dadurch gekennzeichnet, daß sich alle Rechner (DBVS) eine gemeinsame Datenbank teilen, deren Aufbau durch ein einziges konzeptionelles (logisches) Schema beschrieben ist. Da dieses gemeinsame Schema von allen DBVS unterstützt wird, kann den Transaktionsprogrammen (TAP) volle Verteilungstransparenz geboten werden; sie können über das lokale DBVS wie im zentralisierten Fall auf die Datenbank zugreifen (Abb. 3-3). Auf der anderen Seite ist die Autonomie der einzelnen Rechner/DBVS stark eingeschränkt, da Schemaänderungen, Vergabe von Zugriffsrechten etc. global koordiniert werden müssen. Daneben wird vorausgesetzt, daß jede DB-Operation auf allen Rechnern gleichermaßen gestartet werden kann und daß potentiell alle DBVS bei der Abarbeitung einer DB-Operation kooperieren. Diese Randbedingungen erfordern i.d.R., daß die DBVS in allen Rechnern identisch (homogen) sind.

Verteilte Datenbanksysteme sind ein Beispiel solch integrierter Mehrrechner-Datenbanksysteme, wenngleich die Begriffsbildung in der Literatur meist nicht eindeutig ist. Nach unserer in diesem Buch verwendeten Terminologie stellen ***Verteilte Datenbanksysteme*** *geographisch verteilte, integrierte Shared-Nothing-Mehrrechner-Datenbanksysteme* dar. Parallele DBS vom Typ Shared-Disk und Shared-Nothing repräsentieren ebenfalls integrierte Mehrrechner-DBS. Auch bei ihnen liegt in jedem Rechner eine identische DBVS-Version vor; durch Kooperation zwischen den DBVS wird Transaktionsprogrammen sowie Endbenutzern gegenüber eine vollkommene Verteilungstransparenz geboten.

Föderative Mehrrechner-DBS (federated database systems) streben nach größerer Knotenautonomie im Vergleich zu den integrierten Systemen, wobei die beteiligten DBVS entweder homogen oder heterogen sein können. Aufgrund der geforderten Unabhängigkeit der einzelnen Datenbanksysteme und Rechner kommt zur Realisierung dieser Systeme nur eine Partitionierung der Externspeicher in Betracht (Shared-Nothing). Kennzeichnende Eigenschaft der föderativen DBS ist, daß jeder Rechner eine eigene Datenbank verwaltet, die durch ein lokales (priva-

tes) konzeptionelles Schema beschrieben ist. Durch eine begrenzte Kooperation der DBVS soll es möglich werden, auf bestimmte Daten anderer Rechner zuzugreifen, falls dies von dem "Besitzer" der Daten zugelassen wird. Die Menge kooperierender DBVS wird auch als *Föderation* bezeichnet; prinzipiell kann ein DBS an mehreren Föderationen beteiligt sein, falls diese anwendungsbezogen definiert werden. Idealerweise unterstützen föderative Mehrrechner-DBS trotz Unterschieden bei den lokalen DBVS ein einheitliches Datenmodell bzw. bieten eine gemeinsame Anfragesprache “nach außen” an. Insbesondere soll es damit auch möglich werden, innerhalb einer DB-Operation auf Daten verschiedener Datenbanken zuzugreifen sowie einen möglichst hohen Grad an Verteilungstransparenz zu erreichen.

Obwohl bei der Realisierung von föderativen Mehrrechner-DBS auf Techniken von Verteilten DBS zurückgegriffen werden kann, verursachen die Forderungen nach Knotenautonomie sowie Unterstützung heterogener DBVS zusätzliche schwierige Probleme. Insbesondere kann Verteilungstransparenz i.a. nur zum Teil erreicht werden, da ansonsten die lokalen Schemata innerhalb eines globalen Schemas zu integrieren sind. Dies führt jedoch zur Beeinträchtigung von Knotenautonomie und verlangt eine sogenannte Schemaintegration, die vor allem aufgrund semantischer Abbildungsprobleme nur bedingt möglich ist. Weitere Probleme betreffen v.a. die Unterstützung heterogener Datenmodelle und Anfragesprachen sowie die Koordinierung unterschiedlicher Methoden zur Transaktionsverwaltung in den beteiligten Rechnern. Wir behandeln föderative Mehrrechner-Datenbanksysteme in Kap. 12.

3.3 Mehrrechner-DBS mit funktionaler Spezialisierung

Bei den bisher diskutierten Mehrrechner-Datenbanksystemen wurde unterstellt, daß die beteiligten Prozessoren bzw. Rechner hinsichtlich der Verarbeitung von DB-Operationen funktional gleichgestellt sind. Insbesondere wurden als Zuordnungsgranulate jeweils ganze DBVS betrachtet, so daß jeder Prozessor sämtliche Funktionen eines DBVS ausführen konnte. Bei Mehrrechner-DBS mit funktionaler Spezialisierung (vertikal verteilten Mehrrechner-DBS) dagegen werden Verteilungsgranulate innerhalb der DBVS betrachtet, wobei einzelne Funktionen unterschiedlichen Prozessoren/Rechnern zugewiesen werden. Dies führt zu einer funktionalen Spezialisierung dieser Prozessoren/Rechner hinsichtlich der DB-Verarbeitung. Bestimmte Funktionen können dabei durchaus mehreren Prozessoren zugewiesen werden, so daß also für Teilfunktionen eine Replizierung entsteht ähnlich wie bei den bisher diskutierten Mehrrechner-DBS bezüglich vollständiger DBVS.

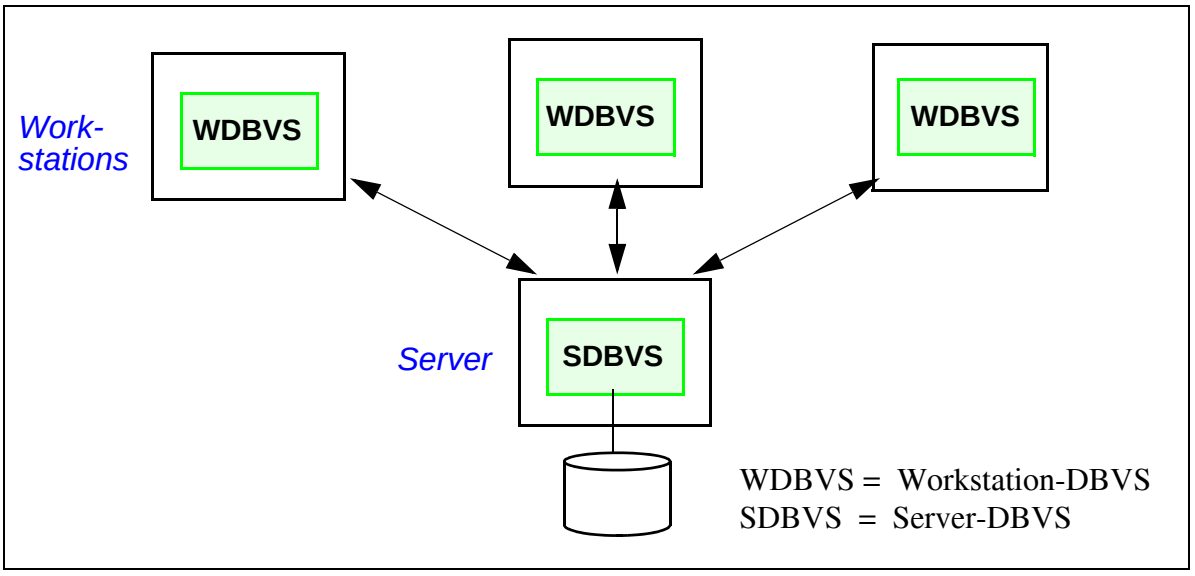
Workstation/Server-DBS (bzw. Client/Server-DBS) sowie bestimmte Datenbankmaschinen stellen bekannte Vertreter von Mehrrechner-DBS mit funktionaler

Spezialisierung dar, die im folgenden diskutiert werden. Einige Anmerkungen zur Klassifikation beschließen das Kapitel.

3.3.1 Workstation/Server-Datenbanksysteme

In Workstation/Server-DBS erfolgt eine Zweiteilung der DBVS-Funktionalität zwischen Workstation-DBVS und Server-DBVS (Abb. 3-5). Die Nutzung von Workstations für die DB-Verarbeitung kann eine Verbesserung der Kosteneffektivität ergeben, da diese Rechnertypen i.a. auf Mikroprozessoren basieren und ein günstiges Preis-/Leistungsverhältnis gegenüber allgemeinen Verarbeitungsrechnern (vor allem Mainframes) aufweisen. Außerdem lassen sich möglicherweise die Bearbeitungszeiten von Transaktionen verkürzen, wenn eine weitgehend lokale DB-Verarbeitung auf den Workstations möglich ist, da deren Verarbeitungskapazität i.a. einem Benutzer exklusiv zur Verfügung steht (Einbenutzerbetrieb auf Workstation-Seite). Dies gilt vor allem für komplexere Verarbeitungsvorgänge bei sogenannten "Nicht-Standard"-DB-Anwendungen wie CAD, Software Engineering, u.ä., wie sie vor allem von objekt-orientierten Datenbanksystemen unterstützt werden sollen. Solche Systeme (GemStone, Iris, O2 etc.) sind daher auch nahezu ausschließlich als Workstation/Server-DBS realisiert.

Abb. 3-5: Grobstruktur von Workstation/Server-Datenbanksystemen



Für die Funktionsaufteilung sowie Kooperation zwischen Workstation- und Server-DBS bestehen unterschiedliche Möglichkeiten. Generell realisiert das Server-DBVS globale Dienste wie Externspeicherverwaltung, Synchronisation und Logging, während Anfragen durch das Workstation-DBVS verarbeitet werden. Weiterhin erfolgt eine Pufferung von DB-Objekten durch die Workstation-DBVS, um durch Nutzung von Lokalität Kommunikationsvorgänge mit dem Server-DBVS einzusparen. Funktionen wie Pufferverwaltung oder Anfrageverarbeitung können

jedoch auch auf Server-Seite vorliegen, so daß sich eine teilweise Replizierung von DBVS-Funktionen ergibt. Als Aufrufeinheiten zwischen Workstation- und Server-DBVS kommen vor allem mengenorientierte Teilanfragen, einzelne Objekt-/Satzanforderungen oder Seitenanforderungen in Betracht, so daß man entsprechend auch von Query-, Objekt- und Page-Server-Ansätzen spricht. Eine Gegenüberstellung dieser Ansätze findet sich u.a. in [DFMV90, HMNR95].

Technische Probleme bei der Realisierung von Workstation/Server-DBS betreffen die Festlegung geeigneter Kooperationsformen zwischen Workstation- und Server-DBVS mit Hinblick auf eine hohe Leistungsfähigkeit. Die replizierte Pufferung von Objekten in den Workstations führt zu einem ähnlichen Kohärenzproblem wie bei Shared-Disk-DBS. Die unterschiedlichen Ausstattungs-, Zuverlässigkeits- und Leistungsmerkmale von Workstations und Server-Rechnern haben Rückwirkungen auf Logging und Recovery, insbesondere die Commit-Behandlung. Darüber hinaus erfordert eine geeignete Unterstützung von Nicht-Standard-Anwendungen erweiterte Transaktions- bzw. Verarbeitungskonzepte (z.B. für lang-lebige Entwurfsvorgänge). Diese Aspekte sind Gegenstand zahlreicher Forschungsarbeiten, können jedoch im Rahmen dieses einführenden Buches nicht mehr behandelt werden.

Die diskutierten Workstation/Server-DBS für Nicht-Standard-Anwendungen stellen integrierte Mehrrechner-DBS dar, da dabei eine logische Datenbank vorliegt und vollkommene Verteilungstransparenz geboten wird. Föderative Mehrrechner-DBS können jedoch auch Client/Server-Architekturen nutzen, indem z.B. von einem Workstation-DBS aus auf unabhängige Server-DBS zugegriffen wird (Kap. 11.1).

3.3.2 Datenbankmaschinen

Mehrrechner-DBS mit funktionaler Spezialisierung ergeben sich auch, wenn bestimmte DBVS-Aufgaben durch Ausführung auf *Spezialprozessoren* optimiert werden sollen. Solche Ansätze wurden vor allem im Kontext von ***Datenbankmaschinen*** vorgeschlagen, welche die Funktionalität eines DBVS innerhalb eines verteilten Back-End-Systems realisieren und diese Anwendungen auf Front-End-Rechnern (z.B. Mainframe-Hosts, Workstations) anbieten. Dabei wurde vielfach für aufwendige DBVS-Funktionen wie Join-Berechnung oder Sortierung eine hardware-basierte Lösung auf dedizierten Rechnern vorgesehen [Su88, KP92]. Solche Ansätze sind jedoch weitgehend gescheitert, vor allem da die Realisierung der Spezial-Hardware lange Entwicklungszeiten und Kosten verursacht [DG92]. Erfolgreicher dagegen ist die software-mäßige Optimierung solcher Funktionen durch Parallelverarbeitung innerhalb allgemeiner (horizontal verteilter) Mehrrechnerarchitekturen (Shared-Nothing oder Shared-Disk). Diese Ansätze sind nicht nur wesentlich einfacher und flexibler realisierbar; sie können vor allem auch die enormen Leistungssteigerungen allgemeiner (Mikro-)Prozessoren nut-

zen. Generell führt die Zerlegung eines DBVS auf mehrere Prozessoren oder Rechner zu einem vermehrten Kommunikationsaufwand innerhalb von DB-Operationen. Zudem steigt bei spezialisierten Prozessoren die Gefahr von Engpässen, und die Möglichkeiten zur Lastbalancierung werden eingeschränkt. Außerdem ergeben sich Verfügbarkeitsprobleme, falls bestimmte Funktionen nur von einem Rechner ausführbar sind.

3.3.3 Abschließende Bemerkungen

Die vorgenommene Unterscheidung zwischen funktionaler Gleichstellung und Spezialisierung erlaubte die Gegenüberstellung wesentlicher Realisierungsalternativen bei Mehrrechner-Datenbanksystemen. Allerdings weisen bei genauerer Betrachtung viele Realisierungen sowohl Merkmale horizontal als auch vertikal verteilter Ansätze auf. So kann auch in Shared-Nothing- oder Shared-Disk-Systemen die Realisierung globaler Kontrollaufgaben zu einer begrenzten Spezialisierung einzelner Rechner führen (z.B. globale Deadlock-Erkennung oder globale Synchronisation auf einem zentralen Rechner). Selbst in Shared-Everything-DBS (bzw. zentralisierten DBVS) findet man teilweise eine funktionale Spezialisierung, wenn die Verarbeitung von DB-Operationen durch mehrere Prozesse unterschiedlicher Funktionalität erfolgt. Auf der anderen Seite können, wie erwähnt, in Architekturen mit funktionaler Spezialisierung bestimmte Teilfunktionen von mehreren Prozessoren ausgeführt werden, so daß es hierfür zu einer horizontalen Verteilung kommt. So kann in Workstation/Server-DBS das Server-System verteilt realisiert sein, um dessen Leistungsfähigkeit und Verfügbarkeit zu verbessern; hierzu ist ein Shared-Everything-, Shared-Disk- oder Shared-Nothing-Ansatz anwendbar. Da die Workstations als Multiprozessoren ausgelegt sein können, ist auch für die Workstation-DBVS eine Shared-Everything-Realisierung möglich. Ferner liegt natürlich in jeder Workstation eine Kopie des Workstation-DBVS vor, wenngleich diese üblicherweise nur mit dem Server-DBVS kommunizieren und nicht untereinander (Abb. 3-5). Daraus folgt, daß bezüglich dieses Klassifikationspunktes die Grenzen zwischen verschiedenen Ansätzen fließend sind. Seine Berechtigung ergibt sich jedoch schon aus der notwendigen Abgrenzung von Workstation/Server-DBS zu anderen Mehrrechner-DBS.

3.4 Vergleich und qualitative Bewertung

Die vorgestellte Klassifikation ergab im wesentlichen folgende Typen von Mehrrechner-DBS: Parallele DBS (Shared-Everything, Shared-Disk, integrierte und lokal verteilte Shared-Nothing-Systeme), Verteilte DBS, föderative DBS und Workstation/Server-DBS. Dabei verkörpert der Shared-Everything-Ansatz nur eine relativ geringe Erweiterung gegenüber zentralisierten DBS, so daß deren Beschränkungen (Leistungsfähigkeit, Skalierbarkeit, Verfügbarkeit etc.) auch zum

Großteil weiter Bestand haben. Im Rest dieses Buchs werden daher - mit Ausnahme der parallelen DB-Verarbeitung (Teil V) - nur noch die anderen, allgemeineren Ansätze behandelt. Bei ihnen kooperieren mehrere DBVS, die jeweils vom Typ "Shared-Everything" sein können. Im folgenden sollen die einzelnen Architekturen hinsichtlich der in Kap. 1.2 aufgestellten Forderungen an Mehrrechner-DBS grob verglichen werden; eine zusammenfassende Bewertung zeigt Abb. 3-6.

Abb. 3-6: Grobe Bewertung von Mehrrechner-DBS

	Parallele DBS (SD, SN)	Verteilte DBS	Föderative DBS	Workstation/ Server-DBS
Hohe Transaktionsraten	++	o/+	o	o
Intra-Transaktionsparallelität	++	o/+	-/o	o/+
Erweiterbarkeit	+	o/+	o	o
Verfügbarkeit	+	+	-	o
Verteilungstransparenz	++	+	o	++
geographische Verteilung	-	+	+	o
Knotenautonomie	-	o	+	-
DBS-Heterogenität	-	-	+	-/o
Administration	o	-	-/--	o

Eine ***hohe Leistungsfähigkeit*** läßt sich demnach am ehesten von Parallelen DBS vom Typ Shared-Disk (SD) und Shared-Nothing (SN) erreichen. Denn diese Ansätze können die Verarbeitungskapazität mehrerer Rechner sowie ein schnelles Kommunikationsnetz zur Erlangung hoher Transaktionsraten auf einer logischen Datenbank nutzen. Sie bieten aufgrund ihrer lokalen Rechneranordnung auch ein höheres Potential für die Parallelisierung komplexer Operationen und Transaktionen (Intra-Transaktionsparallelität) als Verteilte Datenbanksysteme. Diese können zwar ausschließlich lokale Transaktionen effizient bearbeiten, bewirken jedoch im Falle externer Datenzugriffe signifikante Leistungseinbußen. Außerdem kann i.a. keine effektive, dynamische Lastbalancierung erreicht werden; vielmehr bearbeitet jeder Rechner sämtliche Transaktionen der lokal angeschlossenen Benutzer, was nahezu zwangsläufig zu starken Unterschieden in der Auslastung einzelner Rechner führt. Für föderative DBS steht die Leistungsfähigkeit nicht im Vordergrund; sie ist relativ begrenzt, falls pro Datenbank nur ein zentralisiertes DBS eingesetzt wird.

Workstation/Server-DBS versprechen eine gute Leistungsfähigkeit primär für Nicht-Standard-Anwendungen, wenn sich nach Laden benötigter DB-Objekte in den Workstation-Hauptspeicher eine weitgehend lokale Verarbeitung erreichen läßt. Selbst dies garantiert jedoch nicht unbedingt kurze Antwortzeiten, da die Verarbeitung durch das Workstation-DBVS aufgrund hoher Anwendungskomplexität noch sehr aufwendig sein kann, so daß hier ggf. zusätzlich eine Parallelisierung vorzusehen ist. Eine hohe Leistungsfähigkeit auf Server-Seite erfordert dort eine verteilte Realisierung (Shared-Disk, Shared-Nothing) sowie möglicherweise eine Parallelisierung von Server-Operationen. Für einfache Transaktionen kommerzieller DB-Anwendungen erscheint die Verarbeitung innerhalb von Workstation-DBVS dagegen weniger aussichtsreich, da sich hierbei i.a. ein ungünstiges Verhältnis zwischen Kommunikationsaufwand mit dem Server-DBVS sowie lokaler Bearbeitungsdauer ergibt. Allerdings lassen sich Workstation/Server-Architekturen auch für solche Anwendungen im Rahmen verteilter Transaktionssysteme nutzen, wobei die Datenbankverarbeitung dann jedoch i.a. vollkommen auf Server-Seite verbleibt (s. Kap. 11).

Die ***Verfügbarkeit*** ist prinzipiell für integrierte Mehrrechner-Datenbanksysteme (Parallele und Verteilte DBS) am besten, da hierbei nach Ausfall eines Rechners die DB-Verarbeitung nach Durchführung bestimmter Recovery-Aktionen von den überlebenden Rechnern fortgeführt werden kann*. Geographisch verteilte Systeme bieten aufgrund der starken Entkopplung der Verarbeitungsrechner Vorteile vor allem hinsichtlich "Katastrophen", dafür ist bei ihnen vermehrt mit Fehlern im Kommunikationssystem zu rechnen. Insbesondere kann es zu sogenannten "Netzwerk-Partitionierungen" kommen, bei denen Teile des Verteilten DBS nicht mehr miteinander kommunizieren können. In Workstation/Server-Systemen kommt es aufgrund der funktionalen Spezialisierung zu komplexeren Fehlermöglichkeiten; daneben ist die Zuverlässigkeit von Workstations als gering einzustufen. Hohe Fehlertoleranz für das Server-System ist zudem nur bei dessen verteilter Realisierung möglich. Föderative DBS weisen i.a. eine geringe Verfügbarkeit auf, da nach Ausfall eines DBVS die ihm zugeordnete Datenbank nicht mehr erreichbar ist.

Die Vorteile föderativer Mehrrechner-DBS liegen dagegen bei der Unterstützung von heterogenen DBS, einer hohen Knotenautonomie sowie geographischer Verteilung. Dies wird erreicht unter Inkaufnahme einer eingeschränkten Verteilungstransparenz sowie einer komplexen Administration. Bezüglich Kosteneffektivität sind auf der gewählten Betrachtungsebene keine generelle Aussagen mög-

---

* Bei Shared-Nothing muß gewährleistet sein, daß die Daten der von einem Rechnerausfall betroffenen Partition weiterhin zugänglich sind, z.B. aufgrund von replizierter Speicherung oder - bei lokaler Verteilung - durch Übernahme der betroffenen Externspeicher durch überlebende Rechner.

lich. Mikroprozessoren können prinzipiell bei allen betrachteten Arten von Mehrrechner-DBS genutzt werden, nicht nur bei Workstation/Server-DBS.

Die Untersuchung zeigt, daß für jede Verteilform Vor- und Nachteile hinsichtlich der angeführten Bewertungskriterien bestehen, so daß es keinen "idealen" Typ eines Mehrrechner-DBS gibt. Die Eignung verschiedener Architekturansätze ist daher von dem hauptsächlichen Einsatzbereich bestimmt sowie der konkreten Lösung der jeweils zu behandelnden Probleme. Interessanterweise spricht relativ wenig für Verteilte DBS, die für bestimmte Aufgaben Parallelen DBS bzw. föderativen DBS unterlegen sind. Dennoch werden wir Verteilte DBS ausführlich behandeln, da die für sie entwickelten Verfahren auch in lokal verteilten Shared-Nothing-Systemen sowie in föderativen DBS zu einem großen Teil eingesetzt werden können. Teile II und III behandeln die Grundlagen zu verteilten und heterogenen Datenbanken, die Teile IV und V widmen sich Parallelen DBS. In Teil II geht es zunächst um die Realisierung geographisch verteilter, homogener DBS. Darauf aufbauend wird in Teil III die Unterstützung heterogener Datenbanken untersucht. Neben föderativen DBS werden hierbei vor allem Verteilte Transaktionssysteme sowie wichtige Standardisierungsanstrengungen berücksichtigt. Teil IV diskutiert den Shared-Disk-Ansatz zur Realisierung von Mehrrechner-DBS hoher Leistungsfähigkeit. Teil V behandelt die Realisierung von Intra-Transaktionsparallelität im Rahmen Paralleler DBS. Im abschließenden Teil VI werden einige kommerziell verfügbare Mehrrechner-DBS überblicksartig vorgestellt.

Übungsaufgaben

Aufgabe 3-1: Shared-Everything

Die Synchronisation über Semaphore beim Zugriff auf gemeinsame Datenstrukturen kann zu Leistungsengpässen führen. Insbesondere wenn ein DBVS-Prozeß während eines kritischen Abschnitts deaktiviert wird (z.B. wegen E/A-Vorgang oder Zeitscheibenentzug), können schnell sämtliche DBS-Prozesse blockiert werden (Konvoi-Phänomen). Wie kann die Gefahr solcher Synchronisationsengpässe reduziert werden ?

Aufgabe 3-2: Shared-Nothing vs. Shared-Disk

In Kap. 3.4 wurde für Parallele DBS vom Typ Shared-Nothing und Shared-Disk vereinfachend dieselbe Bewertung vorgenommen. Führen Sie einen genaueren Vergleich durch. Welche architekturspezifischen Merkmale haben Rückwirkungen auf Leistungsfähigkeit, Erweiterbarkeit, Verfügbarkeit und Administration ?

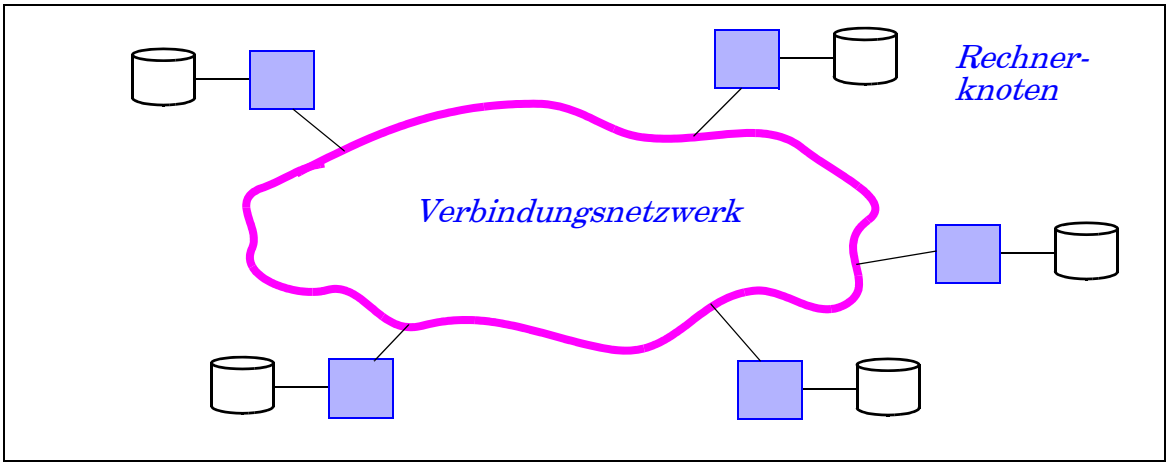
Teil II: Verteilte Datenbanksysteme

Teil II besteht aus sechs Kapiteln. Zunächst (Kap. 4) diskutieren wir die Architektur von Verteilten DBS, insbesondere die Schemaarchitektur sowie Alternativen zur Katalog- und Namensverwaltung. Kap. 5 widmet sich dem zentralen Problem der Datenverteilung, welches in die Teilaufgaben der Fragmentierung und Allokation zerlegt werden kann. Darauf aufbauend diskutieren wir die verteilte Anfragebearbeitung in Kap. 6. Die drei weiteren Kapitel behandeln die Transaktionsverwaltung (Commit-Protokolle und Synchronisation) sowie die Wartung replizierter Datenbanken. In Teil VI werden einige Implementierungen Verteilter DBS vorgestellt.

4 Architektur von Verteilten Datenbanksystemen

Nach unserer Klassifikation stellen Verteilte Datenbanksysteme integrierte, geographisch verteilte Mehrrechner-Datenbanksysteme vom Typ "Shared Nothing" dar. Abb. 4-1 verdeutlicht noch einmal die physikalische Architektur mit der partitionierten Zuordnung von Externspeichern zu Rechnerknoten. An jedem Knoten liegt eine vollständige Instanz des DBVS vor; die einzelnen DBVS-Instanzen kooperieren durch Nachrichtenaustausch über ein Weitverkehrsnetz. Wir werden statt des Begriffs "DBVS-Instanz" hier meist die ungenaueren Bezeichnungen "Knoten" oder "Rechner" verwenden

Abb. 4-1: Grobstruktur eines Verteilten Datenbanksystems.



Im folgenden diskutieren wir zunächst noch einmal die an Verteilte DBS gestellten Anforderungen, insbesondere verschiedene Transparenzeigenschaften, die dem Benutzer gegenüber zu gewährleisten sind. Danach stellen wir als Referenzmodell eine einfache Erweiterung der Schemaarchitektur nach ANSI/SPARC für

den verteilten Fall vor. Abschließend werden die wichtigsten Alternativen zur Katalog- und Namensverwaltung in Verteilten DBS behandelt.

4.1 Transparenzeigenschaften

Die in Kap. 1.2 genannten Anforderungen an Mehrrechner-DBS sollen natürlich auch von Verteilten DBS erfüllt werden, insbesondere hohe Leistungsfähigkeit, hohe Verfügbarkeit, Verteilungstransparenz sowie Unterstützung dezentraler (geographisch verteilter) Organisationsstrukturen. Ähnliche Anforderungen postulierte Date [Da90] innerhalb von zwölf "Regeln" (Abb. 4-2), wobei als Grundregel ("Regel 0") *Verteilungstransparenz* (distribution transparency) verlangt wird. Die meisten der 12 Anforderungen betreffen daher auch Einzelaspekte hinsichtlich der Gewährleistung von Verteilungstransparenz, insbesondere die Unterstützung von *Orts-, Fragmentierungs-* und *Replikationstransparenz* (Regeln 4-6) sowie Unabhängigkeit gegenüber der eingesetzten Hardware, Betriebssysteme, Netzwerke und Datenbanksysteme (Regeln 9-12). Ferner muß das Transaktionskonzept auch im verteilten Fall gewahrt bleiben (Regel 8), insbesondere für Änderungstransaktionen. Dies impliziert die Gewährleistung von *Fehlertransparenz* (Atomarität von Transaktionen) sowie *Transparenz der Nebenläufigkeit* (concurrency transparency [TGGL82], logischer Einbenutzerbetrieb). Da eine DB-Anfrage (Query) den Zugriff auf Daten mehrerer Rechner erfordern kann, ist daneben eine verteilte Anfragebearbeitung zu unterstützen (Regel 7).

Die Diskussion in Kap. 3 zeigte bereits, daß heterogene DBS i.a. den Einsatz föderativer Mehrrechner-DBS bzw. von verteilten DC-Systemen verlangen, so daß Forderung 12 von Verteilten DBS nicht erfüllt werden kann*. Auch die Forderung nach Knotenautonomie bzw. *lokaler Autonomie* (Regel 1) kann i.a. nur teilweise erfüllt werden, da sie der Unterstützung vollständiger Verteilungstransparenz entgegensteht. Denn die Unterstützung eines globalen konzeptionellen Schemas durch alle Knoten erfordert, daß jede DB-Operation darauf auf jedem Rechner ausführbar sein muß. Dies verlangt eine enge Zusammenarbeit der einzelnen DBVS bei der Anfragebearbeitung, eine Koordinierung bei Änderungen im logischen DB-Aufbau sowie die globale Definition von Zugriffsberechtigungen. Allerdings ist generell anzustreben, daß die Zugriffe auf lokale Daten eines Rechners unabhängig von der Verfügbarkeit externer Knoten möglich sein sollten. Dies impliziert u.a., daß Abhängigkeiten zu zentralisierten Systemfunktionen möglichst zu verhindern sind (Regel 2). Jeder Knoten sollte stattdessen auch die Metadaten, Zugriffsrechte, Sperren etc. für die bei ihm gespeicherten Daten verwalten.

---

* Diese Aussage bezieht sich auf unsere Definition von Verteilten Datenbanksystemen. Leider wird dieser Begriff in vielfältiger Bedeutung angewendet, so zum Teil als Oberbegriff von homogenen und heterogenen Mehrrechner-DBS.

Abb. 4-2: Zwölf "Regeln" für Verteilte DBS nach Date [Da90]

1. *Lokale Autonomie*
   Jeder Rechner sollte ein Maximum an Kontrolle über die bei ihm gespeicherten Daten haben. Insbesondere sollte der Zugriff auf diese Daten nicht von anderen Rechnern abhängen.
2. *Keine Abhängigkeit von zentralen Systemfunktionen*
   Zur Unterstützung einer hohen Knotenautonomie und Verfügbarkeit sollte die Datenbankverarbeitung nicht von zentralen Systemfunktionen abhängen. Solche Komponenten stellen zudem einen potentiellen Leistungsengpaß dar.
3. *Hohe Verfügbarkeit*
   Die Datenbankverarbeitung sollte trotz Fehlern im System (z.B. Rechnerausfall) oder Konfigurationsänderungen (Installation neuer Software oder Hardware) idealerweise nicht unterbrochen werden.
4. *Ortstransparenz*
   Die physische Lokation von Datenbankobjekten sollte für den Benutzer verborgen bleiben. Der Datenzugriff sollte wie auf lokale Objekte möglich sein.
5. *Fragmentierungstransparenz*
   Eine Relation der Datenbank sollte verteilt an mehreren Knoten gespeichert werden können. Die dabei zugrundeliegende Zerlegung (Fragmentierung) der Relation sollte für den Datenbankbenutzer transparent bleiben.
6. *Replikationstransparenz*
   Die replizierte Speicherung von Teilen der Datenbank sollte für den Benutzer unsichtbar bleiben; die Wartung der Redundanz obliegt ausschließlich der DB-Software.
7. *Verteilte Anfrageverarbeitung*
   Innerhalb einer DB-Operation sollte auf Daten mehrerer Rechner zugegriffen werden können. Zur effizienten Bearbeitung sind durch das Verteilte DBS geeignete Techniken bereitzustellen (Query-Optimierung).
8. *Verteilte Transaktionsverwaltung*
   Die Transaktionseigenschaften sind auch bei verteilter Bearbeitung einzuhalten, wozu entsprechende Recovery- und Synchronisationstechniken bereitzustellen sind (verteiltes Commit-Protokoll, Behandlung globaler Deadlocks, u.a.).
9. *Hardware-Unabhängigkeit*
   Die DB-Verarbeitung sollte auf verschiedenen Hardware-Plattformen möglich sein. Sämtliche Hardware-Eigenschaften sind für den DB-Benutzer zu verbergen.
10. *Betriebssystemunabhängigkeit*
    Die DB-Benutzung sollte unabhängig von den eingesetzten Betriebssystemen sein.
11. *Netzwerkunabhängigkeit*
    Die verwendeten Kommunikationsprotokolle und -netzwerke sollten ohne Einfluß auf die DB-Verarbeitung bleiben.
12. *Datenbanksystemunabhängigkeit*
    Es sollte möglich sein, unterschiedliche (heterogene) Datenbanksysteme an den verschiedenen Rechnern einzusetzen, solange sie eine einheitliche Benutzerschnittstelle (z.B. eine gemeinsame SQL-Version) anbieten.

Neben den erwähnten Transparenzeigenschaften wird gelegentlich noch *Leistungstransparenz* (performance transparency) gefordert. Dies bedeutet, daß die verteilte Bearbeitung von Anfragen und Transaktionen trotz Kommunikationsverzögerungen möglichst keine merkliche Verschlechterung in den Bearbeitungszeiten verursachen sollte*. Weiterhin soll die Bearbeitungszeit einer Anfrage weitgehend unabhängig davon sein, an welchem Rechner sie gestartet wurde [St89].

4.2 Schemaarchitektur

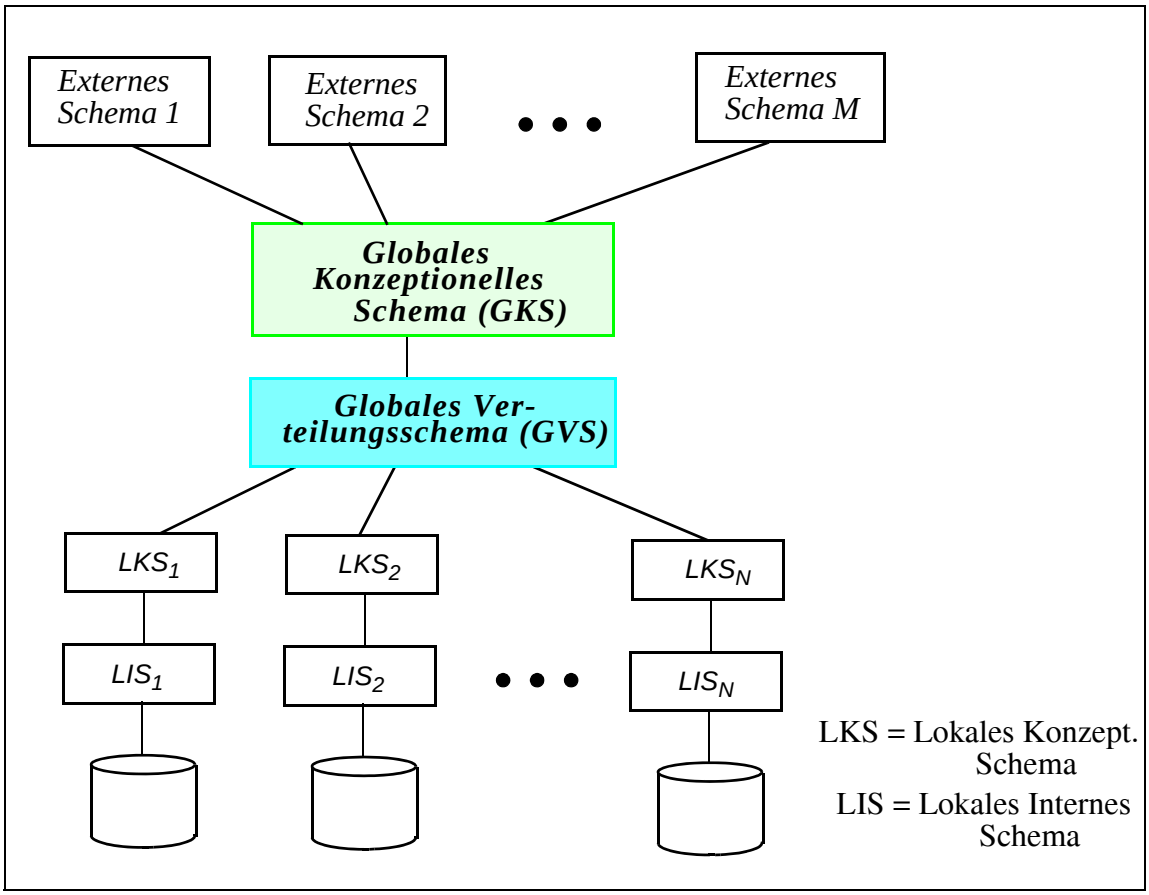
Die Schemaarchitektur eines zentralisierten DBS nach ANSI/SPARC unterstützte physische Datenunabhängigkeit durch Trennung von physischem und konzeptionellen DB-Schema sowie eine begrenzte Form der logischen Datenunabhängigkeit über externe Schemata (Kap. 2.1.2). Bei der Übertragung auf Verteilte DBS ist zusätzlich Verteilungstransparenz zu gewährleisten.

Ein einfacher Ansatz dazu sieht vor, wie im zentralen Fall lediglich ein konzeptionelles und ein internes Schema festzulegen, die von alle Rechnern übernommen werden. In diesem Fall sind sämtliche Verteilungsaspekte Teil des globalen internen Schemas, also insbesondere die Definition der Verteilungseinheiten (Fragmente) sowie ihrer Zuordnung (Allokation) zu Rechnern. Der Nachteil eines solchen Ansatzes liegt darin, daß nahezu keine Knotenautonomie unterstützt wird, da sämtliche Änderungen am konzeptionellen und internen Schema global abzustimmen sind und für alle Rechner Gültigkeit haben. Dies ist besonders in geographisch verteilten Systemen problematisch, da hier lokal unterschiedliche Zugriffsoptimierungen sinnvoll sein können und einzelne Knoten häufiger nicht verfügbar sind.

---

* Bei paralleler Anfragebearbeitung wird sogar eine Verbesserung der Bearbeitungszeiten angestrebt.

Abb. 4-3: Schemaarchitektur von Verteilten DBS



Eine Alternative stellt der in Abb. 4-3 gezeigte Ansatz dar [Da86, ÖV91], bei dem durch die Einführung lokaler Schemata ein höherer Grad an Knotenautonomie erreicht wird. Der logische DB-Aufbau ist wiederum durch ein globales konzeptionelles Schema (GKS) festgelegt, das von allen Rechnern unterstützt wird und auf dem die externen Schemata definiert sind. Daneben besitzt jedoch jeder Rechner ein lokales konzeptionelles Schema (LKS) sowie ein lokales internes Schema (LIS). Die LKS ergeben sich durch die Definition der Datenverteilung, die in einem *globalen Verteilungsschema* (GVS) festgelegt ist. Darin werden vor allem die Angaben zur Fragmentierung von Relationen sowie die Allokation und Replikation von Relationenfragmenten (Kap. 5) geführt. Bei fehlender Replikation stellen die LKS jeweils eine echte Teilmenge des GKS dar; umgekehrt entspricht das GKS der Vereinigung der LKS. Inwieweit Änderungen des LKS, z.B. nach Erzeugung eines neuen Objektes, explizit im System propagiert werden müssen, hängt von der gewählten Katalogarchitektur ab (s.u.). Die LIS erlauben Zugriffspfade und Speicherungsstrukturen auf lokale Erfordernisse abzustimmen und tragen somit ebenfalls zu einer Erhöhung der Knotenautonomie bei.

Dieser Ansatz gewährleistet prinzipiell volle Verteilungstransparenz, da sich sämtliche DB-Operationen auf die externen Schemata bzw. das GKS beziehen. Die Abbildung der Anfragen auf die lokalen Schemata erfolgt automatisch durch die

beteiligten DBS unter Zuhilfenahme der im globalen Verteilungsschema gespeicherten Angaben zur Datenverteilung.

4.3 Katalogverwaltung

Wie im zentralen Fall werden auch in Verteilten DBS die Schemaangaben zum DB-Aufbau sowie weitere Metadaten wie Zugriffsberechtigungen, Paßwörter oder Statistiken (Relationengrößen, Werteverteilungen, Zugriffshäufigkeiten etc.) in einem Katalog verwaltet. Aufgrund der eingeführten Schemaarchitektur kann im verteilten Fall zwischen lokalem und globalem Katalog unterschieden werden. Ein *lokaler Katalog* entspricht im wesentlichen dem Katalog eines zentralisierten DBS und enthält insbesondere das LKS und LIS eines Knotens. Der *globale Katalog* dagegen umfaßt u.a. das GKS und GVS und ist damit maßgebend zur Gewährleistung von Verteilungstransparenz. Insbesondere lassen sich mit ihm logische Objektnamen des globalen konzeptionellen Schemas transparent für den Benutzer auf physische Adressen abbilden. Der globale Katalog dient ferner zur systemweiten Verwaltung von Benutzern und Zugriffsrechten. Die Katalogdaten werden vor allem bei der Übersetzung, Optimierung und Ausführung von DB-Anfragen benötigt (Kap. 6).

In relationalen Datenbanksystemen hat es sich durchgesetzt, den Katalog selbst als Datenbank zu führen, auf die mit der DB-Anfragesprache zugegriffen werden kann. Zudem lassen sich dann die Mechanismen zur Transaktionsverwaltung (Synchronisation, Logging, Recovery) auch auf den Katalog anwenden. Der SQL-92-Standard schreibt ein sogenanntes INFORMATION-SCHEMA vor, das aus verschiedenen Sichten (Views) auf den eigentlichen Katalog ("Definition Schema") besteht [[DD92], [MS92]]. Zur Unterstützung von Datenunabhängigkeit bzw. Verteilungstransparenz sind im INFORMATION_SCHEMA keine Angaben bezüglich des internen Schemas (z.B. existierende Indexstrukturen) und Verteilungsschemas vorgesehen. Diese Angaben befinden sich lediglich im Katalog selbst, dessen Aufbau jedoch nicht standardisiert ist, sondern herstellerspezifisch festgelegt werden kann.

Bei der Verwaltung des Katalogs ist neben dem Aufbau des Katalogs vor allem zu klären, wo dieser gespeichert werden soll. Dabei ist generell vorzusehen, daß jeder Knoten einen lokalen Katalog bezüglich der bei ihm vorliegenden Objekte führt. Damit können für lokale Daten auch für den Katalogzugriff Kommunikationsvorgänge vermieden werden. Dies ist bei der oft hohen Lokalität des Datenzugriffs für das Leistungsverhalten sehr wesentlich. Weiterhin wird ein hohes Maß an Knotenautonomie unterstützt. Für die Speicherung des globalen Katalogs bestehen im wesentlichen folgende Alternativen:

- *Zentralisierter Katalog*
  Sämtliche globalen Katalogangaben werden an einem zentralen Knoten gespeichert.

Dieser Ansatz ist einfach, führt jedoch einen hohen Kommunikationsaufwand ein. Weiterhin stellt der zentrale Knoten einen potentiellen Leistungs- und Verfügbarkeitsengpaß dar. Zudem ergibt sich eine für geographisch verteilte Systeme inakzeptable Beeinträchtigung der Knotenautonomie.

- *Replizierter Katalog*
  Jeder Knoten führt eine vollständige Kopie der globalen Katalogdaten. Der Vorteil liegt darin, daß lesende Katalogzugriffe stets lokal, ohne Kommunikationsverzögerungen, durchführbar sind. Änderungen sind dagegen sehr aufwendig, insbesondere in geographisch verteilten Systemen, da sie an allen Knoten durchzuführen sind. Zur Wartung der Replikation kommen dabei dieselben Techniken wie für replizierte Datenbanken (Kap. 9) in Betracht. In geographisch verteilten Systemen ist es auch unter Schutzaspekten problematisch, daß in jedem Rechner Metadaten von allen Knoten vorliegen, auch wenn deren Daten nicht referenziert werden.
- *Mehrfachkataloge*
  Hierbei partitioniert man die Rechner in mehrere Cluster, wobei pro Cluster jeweils ein Knoten den vollständigen globalen Katalog hält. Dies entspricht einem Kompromiß aus den beiden vorhergehenden Ansätzen. Insbesondere wird der Änderungsaufwand reduziert; die Einschränkungen des zentralisierten Ansatzes hinsichtlich Engpaßgefahr, Verfügbarkeit und unzureichender Knotenautonomie entfallen weitgehend. Allerdings ist weiterhin für jeden globalen Katalogzugriff Kommunikation erforderlich.
- *Partitionierter Katalog*
  Der globale Katalog wird unter allen Rechnern partitioniert gespeichert. Dabei erfolgt keine explizite Speicherung des GKS, sondern dieses ergibt sich als Vereinigung der LKS. Eine Partitionierung des GVS ist möglich, da durch erweiterte Objektbezeichnungen ermittelt werden kann, wo die relevanten Verteilungsinformationen vorliegen, um nicht-lokale Objekte zu lokalisieren (s. Namensverwaltung, Kap. 4.4).

Von diesen Alternativen erscheint der letzte Ansatz am vielversprechendsten, da er den höchsten Grad an Knotenautonomie bewahrt. Allerdings werden dabei für nicht-lokale Daten Kommunikationsvorgänge bereits für Katalogzugriffe erforderlich. Dieser Nachteil kann durch eine *Pufferung (Caching) nicht-lokaler Katalogdaten* abgeschwächt werden. Damit läßt sich für häufiger benötigte Katalogdaten anderer Rechner ebenfalls Kommunikation vermeiden.

Bei der Pufferung nicht-lokaler Katalogdaten besteht ein ähnliches Änderungsproblem wie bei replizierten Katalogen, da gepufferte Kopien nach einer Änderung an einem anderen Knoten ungültig sind. Im Zusammenhang von (replizierter) Pufferung von Objekten spricht manedoch üblicherweise von der Notwendigkeit einer *Kohärenzkontrolle* [Ra93b]; Anzahl und Ort der Kopien sind im Gegensatz zu replizierten Katalogen (Datenbanken) nicht vorbestimmt sondern ergeben sich dynamisch durch die Referenzverteilung im System. Zur Kohärenzkontrolle für gepufferte Katalogdaten werden im wesentlichen zwei Alternativen eingesetzt:

- Der Besitzerknoten der Katalogdaten vermerkt sich, an welchen Rechnern eine Pufferung seiner Daten erfolgte. Bei einer Änderung werden die betroffenen Kopien explizit invalidiert. Damit kann der Zugriff auf invalidierte Daten vermieden werden,

jedoch zu Lasten eines hohen Wartungs- und Kommunikationsaufwandes. Ein solcher Ansatz wurde in der Prototyp-Implementierung SDD-1 verfolgt.

- Bei der im Prototyp R* realisierten Alternative wird eine Invalidierung von Katalogdaten zugelassen [Li81]. Die Verwendung veralteter Katalogdaten bei der Erstellung von Ausführungsplänen für DB-Operationen wird erst zur Ausführungszeit an den datenhaltenden Knoten erkannt. Dies geschieht durch Führen von Versionsnummern für Katalogeinträge, wobei in den Ausführungsplänen die Versionsnummern der verwendeten Kopien aufgenommen werden. Wird zur Ausführungszeit festgestellt, daß nach Erstellung des Ausführungsplanes eine Änderung der Katalogdaten erfolgte (z.B. Löschen einer Indexstruktur), muß ein neuer Plan bestimmt werden.

Ein Sonderfall bei der lokalen Katalogverwaltung ergibt sich, wenn Objekte (Relationen) über mehrere Knoten partitioniert oder repliziert sind. In diesem Fall empfiehlt sich, die Kataloginformation der Objekte repliziert an allen Knoten zu führen, an denen sie (teilweise) gespeichert werden [Li81]. Denn dann können die Katalogzugriffe für lokal vorliegende Objekte weiterhin lokal abgewickelt werden.

4.4 Namensverwaltung

Sämtliche Datenbankobjekte in einem Verteilten DBS müssen systemweit eindeutige Namen besitzen, die zudem möglichst stabil sein sollten. Daneben sollte die Namensvergabe Ortstransparenz unterstützen, so daß sich insbesondere nach Migration eines Objektes (Änderung des speichernden Knotens) keine Auswirkungen für bestehende Programme ergeben. Weiterhin sollte ein bestimmtes Programm ohne Modifikation an allen Knoten eines Verteilten DBS ablauffähig sein. Zur Unterstützung von Knotenautonomie gilt es zudem, eine lokale Namensvergabe bei Erzeugung neuer Objekte zu ermöglichen, also ohne durch Kommunikation mit anderen Knoten die globale Eindeutigkeit sicherzustellen. Schließlich muß es auch in Verteilten DBS möglich sein, daß verschiedene Benutzer (unterschiedliche) Objekte mit demselben (logischen) Namen erzeugen.

Zur Unterstützung von Verteilungstransparenz ist es wünschenswert, daß Benutzer die für zentralisierte DBS gültigen Namenskonventionen weiterhin benutzen können. Zentralisierte SQL-Implementierungen verwenden oft eine zweiteilige Namensstruktur für DB-Objekte* (Relationen, Sichten, ...), bei der ein *logischer Objektname* folgendermaßen aufgebaut ist:

[<Benutzername>.]<Objektname>.

Dabei sind sämtliche DB-Objekte dem erzeugenden Benutzer zugeordnet. Die Angabe des Benutzernamens ist nur für Zugriffe auf Objekte eines anderen Benutzers erforderlich. Dieser Ansatz ermöglicht, daß verschiedene Benutzer Objekte mit dem gleichen Objektnamen erzeugen können.

---

* In SQL92 wurde eine erweiterte Namensstruktur eingeführt (s. Übungsaufgaben).

Im (geographisch) verteilten Fall garantiert diese Namensstruktur jedoch keine globale Eindeutigkeit von Objektbezeichnungen, da Benutzernamen i.a. nicht systemweit eindeutig sind. Prinzipiell läßt sich dieses Problem dadurch beheben, indem auf das globale konzeptionelle Schema (GKS) zugegriffen wird, um die Eindeutigkeit eines Objektnamens sicherzustellen. Dies ist am ehesten möglich bei einem *Name-Server-Ansatz*, bei dem sämtliche Namen des GKS (sowie die physische Lokation der Objekte) an einer zentralen Stelle geführt werden. Dies entspricht einer zentralisierten Katalogverwaltung und verursacht wiederum Kommunikationsverzögerungen zur Namensvergabe und verletzt die Knotenautonomie. Ähnlich problematisch ist die vollständige Replikation des GKS, vor allem bei hoher Rechneranzahl. Denn hierbei sind die durch Erzeugen neuer Objekte bedingten Änderungen des GKS systemweit zu synchronisieren und zu propagieren.

Die Alternative besteht in einem hierarchischen Namenskonzept, bei dem die Objektbezeichnungen zur Erlangung globaler Eindeutigkeit um Knotennamen erweitert werden. Wir bezeichnen solchermaßen erweiterte, systemweit eindeutige Bezeichnungen als *globale Objektnamen.* Eine mögliche Struktur globaler Objektnamen sieht folgendermaßen aus*:

[[<Knotenname>.]<Benutzername>.]<Objektname>.

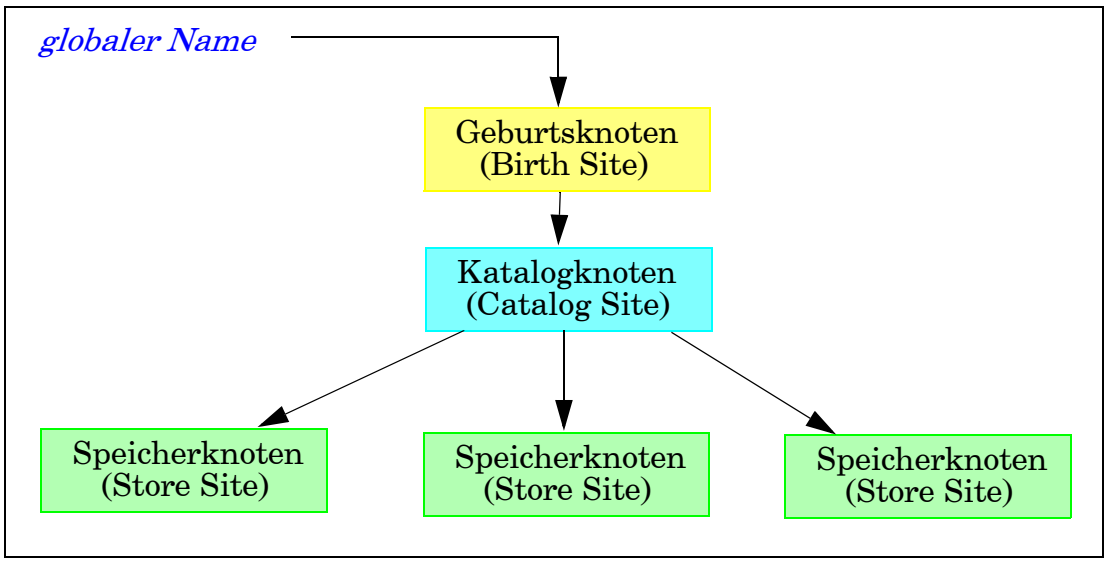
Dabei sei <Knotenname> die Bezeichnung des Rechners, an dem das Objekt erzeugt wurde ("birth site") [Li81]. Ist dieser Name nicht spezifiziert, wird defaultmäßig der Rechner verwendet, an dem die betreffende DB-Operation gestartet wird. Damit kann für lokal erzeugte Objekte die Angabe von Objektnamen wie im zentralen Fall erfolgen. Insbesondere braucht bei der Erzeugung eines neuen Objektes kein Knotenname angegeben werden; zudem läßt sich die globale Eindeutigkeit bereits mit dem lokalen Katalog überprüfen. Für den Zugriff auf nicht-lokale Objekte ist jedoch die Spezifizierung des Knotennamens erforderlich, was Zusatzmaßnahmen zur Gewährleistung von Verteilungstransparenz verlangt (s.u.).

Insgesamt sind drei Arten von Knoten bei der Verwaltung und Speicherung von Objekten beteiligt (Abb. 4-4): der Knoten, an dem ein Objekt erzeugt wurde (birth site), der (oder die) Knoten, an dem Katalogdaten zu dem Objekt verwaltet werden (catalog site), und schließlich die Knoten, an denen das Objekt gespeichert ist (store site). Es handelt sich dabei um eine logische Sichtweise, da ein einzelner Knoten durchaus die verschiedenen Funktionen auf sich vereinen kann (was zur Reduzierung von Kommunikationsvorgängen auch wünschenswert ist). Wesentlich dabei ist, daß der im globalen Objektnamen explizit bezeichnete Geburtsknoten unabhängig vom Speicherungsort ist. Dies ermöglicht Ortstransparenz, da die physische Lokation eines Objektes wechseln kann (Migration von Objekten), ohne daß der globale Namen zu ändern ist. Auch werden Objekte stets an einem Knoten

* Ein ähnliches Namensschema wird in [Da92] diskutiert.

erzeugt, können aber über mehrere Knoten partitioniert oder repliziert gespeichert werden.

Abb. 4-4: Geburts-, Katalog- und Speicherknoten von Objekten [CS91]



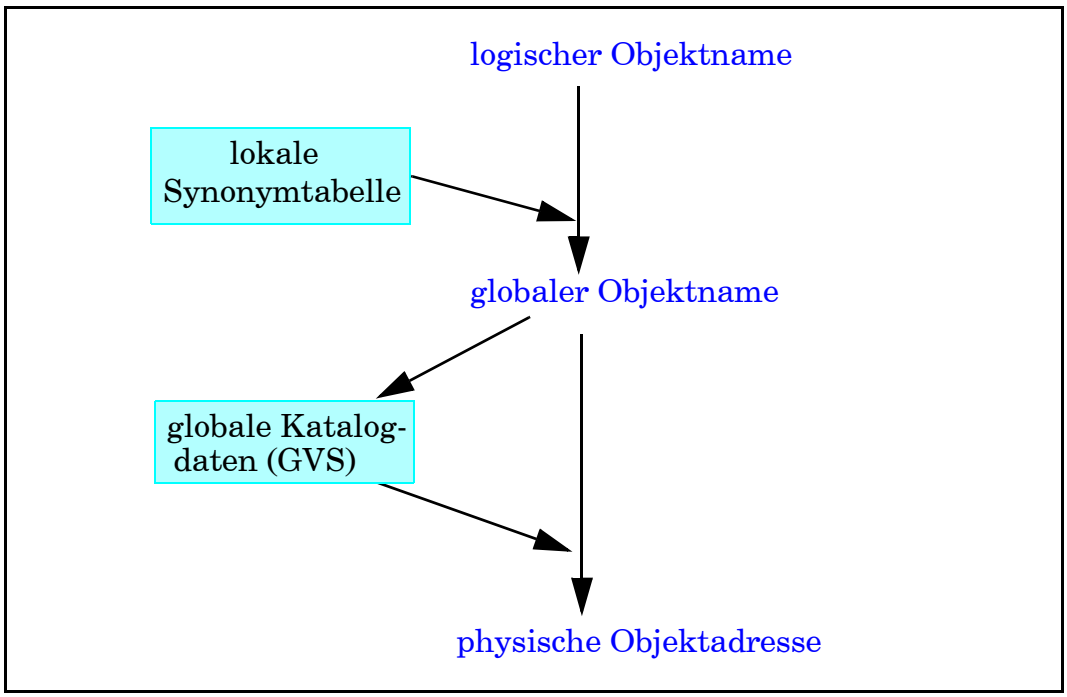
Die physische Adresse eines Objektes wird über die globalen Katalogdaten (Verteilungsschema) bestimmt und ist von der gewählten Katalogarchitektur abhängig. Anstatt einer zentralisierten oder replizierten Speicherung dieser Angaben bietet es sich bei der eingeführten Namensstruktur an, diese am "birth site" zu hinterlegen [Li81], so daß dann Geburts- und Katalogknoten zusammenfallen. Dies führt zu einer Partitionierung der globalen Katalogdaten, wobei die globalen Objektnamen direkt spezifizieren, wo die Verteilungsinformation für ein Objekt vorliegt. Dieser partitionierte Ansatz zur Katalog- und Namensverwaltung wurde in R* verwendet. Da zur Unterstützung von Knotenautonomie die globalen Katalogdaten auch an den Speicherknoten zu führen sind, kann es bei Speicherung eines Objektes an mehreren Knoten (Replikation, Fragmentierung) bzw. wenn aufgrund von Datenmigration Speicher- und Geburtsknoten differieren jedoch auch zu einer begrenzten Replikation von Katalogdaten kommen.

Durch die Hinzunahme des Geburtsknotens in globalen Objektnamen ergeben sich jedoch Probleme hinsichtlich der Gewährleistung von Verteilungstransparenz. Denn um auf extern erzeugte Objekte zugreifen zu können, ist die Spezifikation von Knotennamen erforderlich. Soll ein Anwendungsprogramm bzw. eine DB-Operation unverändert an mehreren Rechnern ausführbar sein, sind daneben sämtliche DB-Objekte vollständig zu spezifizieren, um eine korrekte Namensauflösung zu erreichen. Der üblicherweise verwendete Ansatz, um Programme und DB-Operationen ohne Spezifikation von Knotennamen erstellen zu können, liegt in der Definition sogenannter *Synonyme* (Alias-Namen). Diese werden vom DBS benutzerspezifisch verwaltet und gestatten die Abbildung von einfachen Objektnamen in vollqualifizierte globale Objektnamen. Die Knotenangaben befinden sich dabei le-

diglich in den Synonymtabellen des DBS, die typischerweise vom Datenbankadministrator für die lokalen Benutzer angelegt werden*. Synonyme werden u.a. in R* [Li81] und vielen kommerziellen Systemen (Tandem NonStop SQL [Ta89], DB2, Oracle etc.) verwendet.

Die sich damit ergebenden Einzelschritte bei der *Namensauflösung* (name resolution) sind in Abb. 4-5 noch einmal zusammengefaßt. Zunächst erfolgt die Umsetzung logischer Namen in globale Objektnamen mit der im lokalen Katalog vorliegenden Synonymtabelle. Liegt kein Synonym vor, erfolgt die defaultmäßige Vervollständigung (Benutzername, Knotenname) zu einem globalen Namen. Der globale Objektname spezifiziert den Geburtsknoten, an dem auch die globale Verteilungsinformation für das betreffende Objekt vorliegt. Damit kann dann schließlich die physische Adresse (Speicherknoten) der zu referenzierenden Daten ermittelt werden.

Abb. 4-5: Namensauflösung in Verteilten DBS



Übungsaufgaben

Aufgabe 4-1: Katalogverwaltung

Welche Alternativen eignen sich zur Katalogverwaltung in lokal verteilten Shared-Nothing-Systemen, bei denen keine Knotenautonomie zu unterstützen ist ?

* Alternativ läßt sich auch mit den externen Schemata (bzw. Sichten im Relationenmodell) Verteilungstransparenz erreichen. In diesem Fall sind die logischen Namen der externen Schemata in die globalen Namen des GKS zu transformieren. Synonyme stellen dagegen eine Indirektion auf Ebene der konzeptionellen Schemata dar.

Aufgabe 4-2: Namensauflösung in R*

In R* wurde folgende vierteilige Struktur globaler Objektnamen realisiert:

[<Benutzername> [@<Benutzerknoten>].]<Objektname>[@<Objektknoten>]

<Benutzername> und <Benutzerknoten> ergeben zusammen global eindeutige Bezeichnungen für Benutzer; <Objektknoten> entspricht wiederum dem Ort, an dem das Objekt erzeugt wurde. Bei der Namensauflösung werden nicht angegebene Knotennamen mit dem aktuellen Knoten ersetzt.

a) Welche Vorteile ergeben sich durch die Hinzunahme von <Benutzerknoten> ?

b) Die Relation BEISPIEL mit dem globalen Namen RAHM@L.BEISPIEL@F soll von verschiedenen Benutzern an verschiedenen Orten referenziert werden. Wie sieht der jeweils kürzeste Name aus, der (ohne Synonyme) korrekt zum vollständigen globalen Namen expandiert werden kann

- für Benutzer RAHM
- für sonstige Benutzer
- am Knoten L, am Knoten B und am Knoten F ?

Aufgabe 4-3: Namensverwaltung (SQL92)

Der SQL92-Standard verwendet dreiteilige Objektbezeichnungen

[[<Katalogname>.]<Schemaname>.]<Objektname>

Dabei ist jedes Schema eindeutig einem Benutzer zugeordnet, jedoch kann ein Benutzer mehrere Schemata besitzen. Mehrere Schemata können zu einem "Katalog" zusammengefaßt werden; die DB-Installation kann wiederum mehrere Kataloge umfassen. In einer SQL-Anweisung lassen sich Objekte aus unterschiedlichen Katalogen referenzieren*.

Reicht diese Namensstruktur für Verteilte Datenbanksysteme aus bzw. unter welchen Voraussetzungen ? Durch welche Erweiterungen könnte globale Eindeutigkeit der Namen unter Bewahrung von Knotenautonomie erreicht werden ?

Aufgabe 4-4: Herausnahme von Knoten aus dem System

Ein Knoten des Verteilten DBS soll dauerhaft aus dem System genommen werden. Kann dies mit der eingeführten Struktur globaler Namen unterstützt werden ?

Aufgabe 4-5: Synonyme

Welche Probleme ergeben sich bei Verwendung von Synonymen zur Unterstützung von Verteilungstransparenz ?

Aufgabe 4-6: Nutzung standardisierter Directory-Dienste

Die Katalog- und Namensverwaltung in verteilten Systemen ist ein generelles Problem. Inwieweit könnten standardisierte Directory-Dienste wie X.500 [Cy91] zur Lokalisierung von DB-Objekten genutzt werden ?

---

* Diese Verwendung des Begriffs "Katalog" als Container für mehrere Datenbanken einzelner Benutzer weicht von unserer ab, da wir mit Katalog die Menge der Metadaten (Dictionary) bezeichnen.

5 Datenbankverteilung

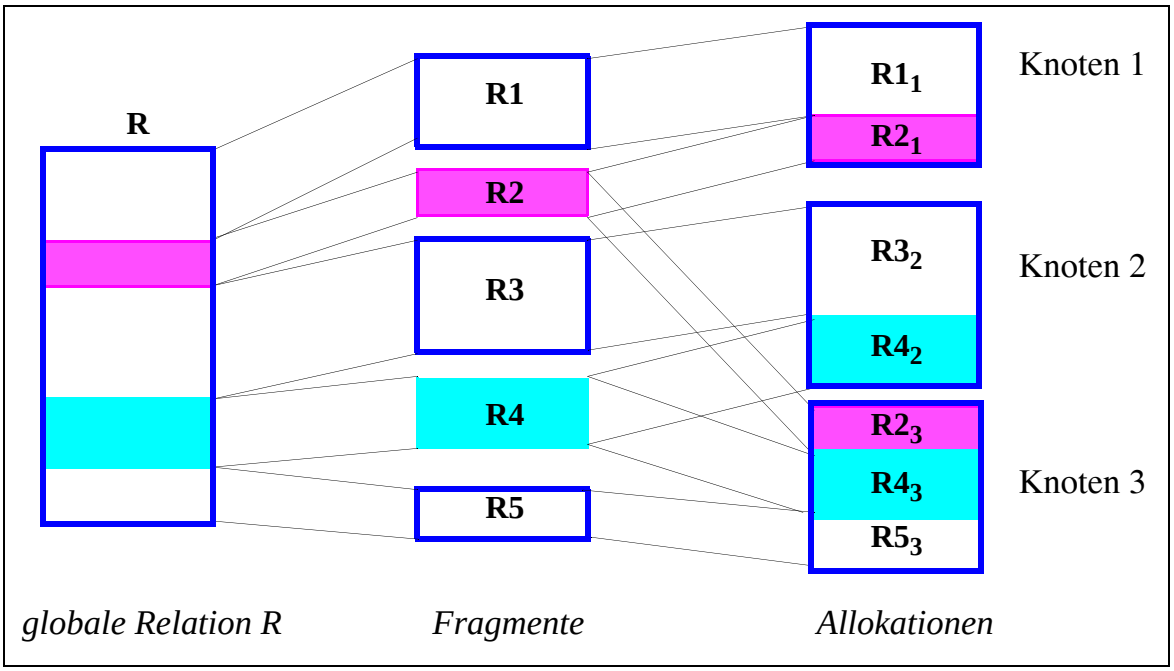
Der Einsatz eines Verteilten Datenbanksystems (bzw. von Shared-Nothing-Systemen generell) verlangt die Festlegung einer physischen Aufteilung des Datenbestandes. Die Bestimmung der Datenverteilung ist dabei für die Leistungsfähigkeit des Systems von fundamentaler Bedeutung, da sie zu einem großen Teil den Kommunikationsaufwand zur DB-Verarbeitung bestimmt. Weiterhin ergeben sich durch die Datenverteilung Rückwirkungen auf die Lastbalancierung, da die einem Rechner zugeordneten Daten die von ihm zu verarbeitenden Operationen bestimmen. Durch die Möglichkeit, Teile der Datenbank repliziert zu speichern, wirkt sich die Datenverteilung auch auf die Verfügbarkeit aus.

Die Bestimmung einer Datenbankverteilung kann nach einem Top-Down- oder einem Bottom-Up-Ansatz erfolgen. Der *Bottom-Up-Ansatz* eignet sich vor allem, wenn mehrere existierende Datenbanken integriert werden sollen. Dabei werden dann insbesondere die einzelnen lokalen konzeptionellen Schemata (LKS) zu einem gemeinsamen konzeptionellen Schema (GKS) zusammengefaßt. Dieser Ansatz kommt vor allem im Kontext von föderativen DBS in Frage und wird daher in Kap. 12 weiterbetrachtet. Beim *Top-Down-Ansatz* dagegen liegt bereits ein GKS einer logischen Datenbank vor. Hier gilt es zu entscheiden, wie die Objekte (Relationen) des GKS auf die einzelnen Rechner und deren LKS verteilt werden.

Diese Aufgabe läßt sich in zwei Teilprobleme untergliedern: Fragmentierung und Allokation. Im Rahmen der *Fragmentierung* werden dabei zunächst die Einheiten der Datenverteilung (Fragmente) festgelegt. In relationalen Datenbanken kommen hierzu vor allem eine zeilenweise (horizontale) oder spaltenweise (vertikale) Fragmentierung in Betracht. Die *Allokation* (Ortszuweisung) bestimmt danach, welchem Rechner jedes der Fragmente zugeordnet wird, wobei eine replizierte Alllokation von Fragmenten möglich ist. Dieser zweistufige Abbildungsprozeß wird durch Abb. 5-1 veranschaulicht. Dabei wird die im GKS definierte *globale Relation* R zunächst in mehrere Fragmente zerlegt. Danach erfolgt die möglicherweise replizierte Allokation dieser Fragmente zu Rechnern. Im Beispiel wird so eine Zerlegung in fünf Fragmente vorgenommen, von denen für zwei (R2, R4) eine replizierte Allokation stattfindet. Die Menge der einem Rechner zugeordneten Fragmente einer globalen Relation ergeben dessen *lokale Relation*; z.B. umfaßt die an

Knoten 2 vorliegende lokale Relation die Fragmente R3 und R4. Die DB-*Partition* eines Rechners besteht aus der Menge seiner lokalen Relationen.

Abb. 5-1: Fragmentierung und Allokationen von Relationen [CP84]



Nachfolgend diskutieren wir zunächst generelle Alternativen zur Fragmentierung (Kap. 5.1), Allokation und Replikation (Kap. 5.2). Danach werden die Alternativen zur Fragmentierung im Detail behandelt, insbesondere die horizontale Fragmentierung (Kap. 5.3), vertikale Fragmentierung (Kap. 5.4) und hybride Fragmentierung (Kap. 5.5). Anhand eines Beispiels wird dann die Bedeutung des Begriffs "Fragmentierungstransparenz" verdeutlicht (Kap. 5.6). Abschließend wird dann auf die Bestimmung der Datenverteilung eingegangen, insbesondere auf die Berechnung einer Datenallokation.

5.1 Fragmentierung

In relationalen Datenbanksystemen stellen im einfachsten Fall ganze Relationen die kleinsten Verteilungsgranulate dar. In diesem Fall entfällt die Notwendigkeit einer expliziten Fragmentierung; durch die geringere Anzahl von Verteilungseinheiten vereinfacht sich auch das Allokationsproblem. Ein weiterer Vorzug liegt darin, daß Operationen auf einer Relation stets an einem Rechner ausführbar sind, also mit geringen Kommunikationskosten. Dies ist auch vorteilhaft zur Überprüfung von Integritätsbedingungen (z.B. Eindeutigkeit von Attributwerten, referentielle Integrität).

Auf der anderen Seite sprechen wichtige Gründe gegen eine Datenverteilung auf Relationenebene und für eine feinere Fragmentierung:

- *Lastbalancierung*
  Die Allokation vollständiger Relationen ist meist zu inflexibel, um eine effektive Nutzung aller Rechner zu erlauben. Denn aufgrund der begrenzten Anzahl von Relationen sowie oft starker Schwankungen in den Relationengrößen und Zugriffshäufigkeiten wird es nur schwer möglich sein, diese so unter den Rechnern aufzuteilen, daß eine einigermaßen gleichmäßige Lastbalancierung erreicht wird.
- *Nutzung von Lokalität*
  Die meisten Relationen werden an mehreren Knoten referenziert, wobei oft ein hoher Grad an (rechnerspezifischer) Zugriffslokalität vorliegt, so daß verschiedene Rechner vorwiegend disjunkte Teile einer Relation referenzieren. Durch eine Fragmentierung und partitionierte Allokation von Relationen kann dies zur Einsparung von Kommunikationsvorgängen genutzt werden. So dominieren in einer Bankanwendung in einzelnen Zweigstellen meist Zugriffe auf lokal eingerichtete Konten; die Speicherung der gesamten Kontorelation an einem Knoten würde ein Vielfaches an Kommunikation (sowie eine stark ungleiche Lastbalancierung) verursachen.
- *Reduzierung des Verarbeitungsumfangs*
  Vielfach können Operationen auf ein Fragment (bzw. eine Teilmenge der Fragmente) beschränkt werden. Durch die Verringerung der zu verarbeitenden Datenmenge ergibt sich eine effizientere Bearbeitung gegenüber der Ausführung auf der gesamten Relation.
- *Unterstützung von Parallelverarbeitung*
  Durch die Fragmentierung wird es möglich, Operationen auf einer Relation in Teiloperationen auf Fragmenten unterschiedlicher Rechner zu zerlegen, die zur Verkürzung der Bearbeitungszeit parallel ausgeführt werden können.

Zur Zerlegung einer globalen Relation in Fragmente kommen mehrere Alternativen in Betracht, die im weiteren Verlauf dieses Kapitels beschrieben werden. Insbesondere ist eine horizontale (zeilenweise) oder vertikale (spaltenweise) Fragmentierung möglich, die jeweils durch einen relationalen Ausdruck spezifizierbar sind. Bei der Fragmentierung sind drei Regeln zu beachten, um die Korrektheit der Zerlegung sicherzustellen [CP84]:

- *Vollständigkeit*
  Jedes Datenelement (Tupel, Attributwert) der globalen Relation muß in wenigstens einem Fragment enthalten sein.
- *Rekonstruierbarkeit*
  Die Zerlegung muß verlustfrei sein, so daß die globale Relation aus den einzelnen Fragmenten wieder vollständig rekonstruiert werden kann.
- *Disjunktheit*
  Fragmente sollten möglichst disjunkt sein, da ihre Replikation im Rahmen der Allokation festgelegt wird. Bei der vertikalen Partitionierung ist jedoch die Disjunktheit von Fragmenten nicht vollständig einzuhalten, um Rekonstruierbarkeit zu gewährleisten (s. Kap. 5.4).

5.2 Allokation und Replikation

Nach Festlegung der Fragmentierung für globale Relationen besteht der zweite Schritt bei der Datenverteilung in der Allokation der Fragmente zu Knoten. Wird

dabei keines der Fragmente mehrfach allokiert, erhält man eine *partitionierte Datenbank*, anderenfalls eine replizierte Datenbank. Im letzteren Fall unterscheidet man noch zwischen *voller Replikation*, bei der jede globale Relation vollständig an allen Rechnern gespeichert wird, und *partieller Replikation.* Bei voller Replikation entfällt das Fragmentierungs- und Allokationsproblem. Weiterhin erübrigt sich eine verteilte Anfrageverarbeitung, da jede lesende DB-Operation lokal verarbeitet werden kann. Die Speicherplatz- und Änderungskosten einer vollen Replikation sind jedoch meist prohibitiv.

Ein wesentliches Ziel der replizierten Datenhaltung ist die Steigerung der Verfügbarkeit, da im Gegensatz zur partitionierten Datenallokation replizierte Objekte auch nach Ausfall eines der Speicherknoten zugreifbar bleiben. Replikation kann jedoch - für Leseoperationen - auch unter Leistungsgesichtspunkten sinnvoll sein. Denn wenn mehrere Knoten ein bestimmtes Fragment führen, bestehen größere Optimierungsmöglichkeiten zur Erstellung eines kostengünstigen Ausführungsplanes. Insbesondere kann ein repliziertes Objekt an mehreren Knoten lokal referenziert werden, so daß Kommunikationsvorgänge gegenüber einer partitionierten Datenbankallokation eingespart werden können. Weiterhin erhöhen sich die Möglichkeiten zur Lastbalancierung, da bestimmte Datenzugriffe von mehreren Rechnern ausführbar sind. Diese Vorteile werden jedoch zu Lasten von Änderungsvorgängen erkauft, da jede Objektänderung auf alle Kopien (Replikate) propagiert werden muß.

Aufgrund der geforderten Replikationstransparenz ist die Wartung der Replikation alleinige Aufgabe des Verteilten DBS. Für den Benutzer bleibt die Existenz von Replikation vollständig verborgen. Ebenso unsichtbar bleibt die Lokation einzelner Fragmente (Ortstransparenz).

5.3 Horizontale Fragmentierung

Die horizontale Fragmentierung ist die in existierenden Systemen bedeutendste Fragmentierungsform und wird daher am ausführlichsten behandelt. Bei ihr wird eine globale Relation zeilenweise in disjunkte Teilmengen zerlegt. Die Zuordnung von Sätzen zu Fragmenten wird dabei i.a. über Selektionsprädikate definiert. Hierbei sind zwei Fälle zu unterscheiden. Bei der *einfachen (primären) horizontalen Fragmentierung* beziehen sich die Selektionsprädikate ausschließlich auf Attribute der zu zerlegenden Relation. Bei der *abgeleiteten horizontalen Fragmentierung* dagegen wird die Zerlegung auf die horizontale Fragmentierung einer anderen Relation abgestimmt (s.u.).

5.3.1 Primäre horizontale Fragmentierung

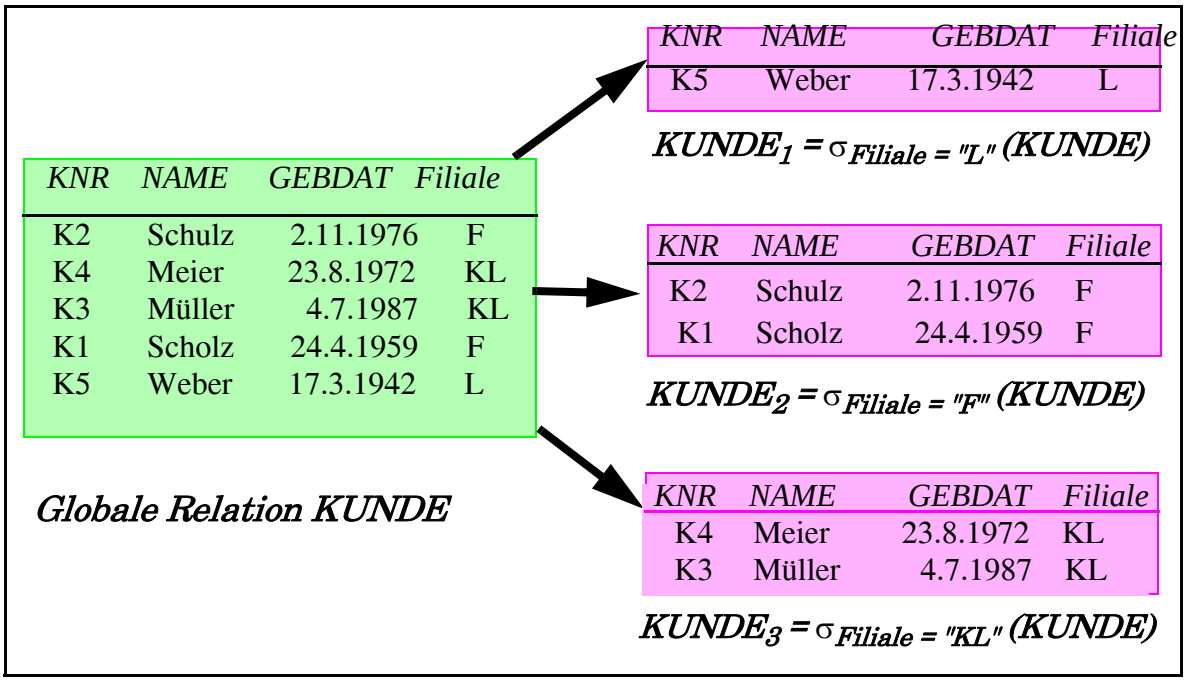
Die primäre horizontale Fragmentierung einer globalen Relation R in n Fragmente R_i erfordert also die Festlegung von n Selektionsprädikaten P_i auf Attributen von R:

$$R_i := \sigma_{Pi}\ (R) \qquad (1 \leq\ i \leq\ n) \qquad \leq$$

Die Prädikate P_i werden auch als *Fragmentierungsprädikate* bezeichnet. Die Forderungen nach Vollständigkeit und Disjunktheit verlangen, daß jeder Satz der globalen Relation genau einem horizontalem Fragment zugeordnet wird. Die Rekonstruierbarkeit ist dann einfach gewährleistet; die globale Relation entspricht der Vereinigung all ihrer Fragmente. Es muß also gelten

$$R_i \cap R_j = \{\} \quad (1 \leq i\ ,\ j \leq\ n;\quad i \neq j\ )\ ,$$
$$R = \cup\ R_i \qquad\qquad (1 \leq\ i\ \leq n)\ .$$

Abb. 5-2: (Primäre) Horizontale Fragmentierung der Kundenrelation nach Filialen



Beispiel 5-1

In der Bankanwendung aus Abb. 2-1 könnte z.B. die KUNDE-Relation, erweitert um das Attribut "Filiale", horizontal nach der einem Kunden zugeordneten Zweigstelle fragmentiert werden. Eine solche (einfache) Zerlegung für drei Filialen ist in Abb. 5-2 gezeigt. Wird diese Fragmentierung zu einer filialbezogenen geographischen Verteilung des Datenbestandes genutzt, kann eine hohe Lokalität im Zugriffsverhalten erreicht werden. Denn in jeder Filiale können dann sämtliche Datenzugriffe bezüglich der eigenen Kunden lokal abgewickelt werden.

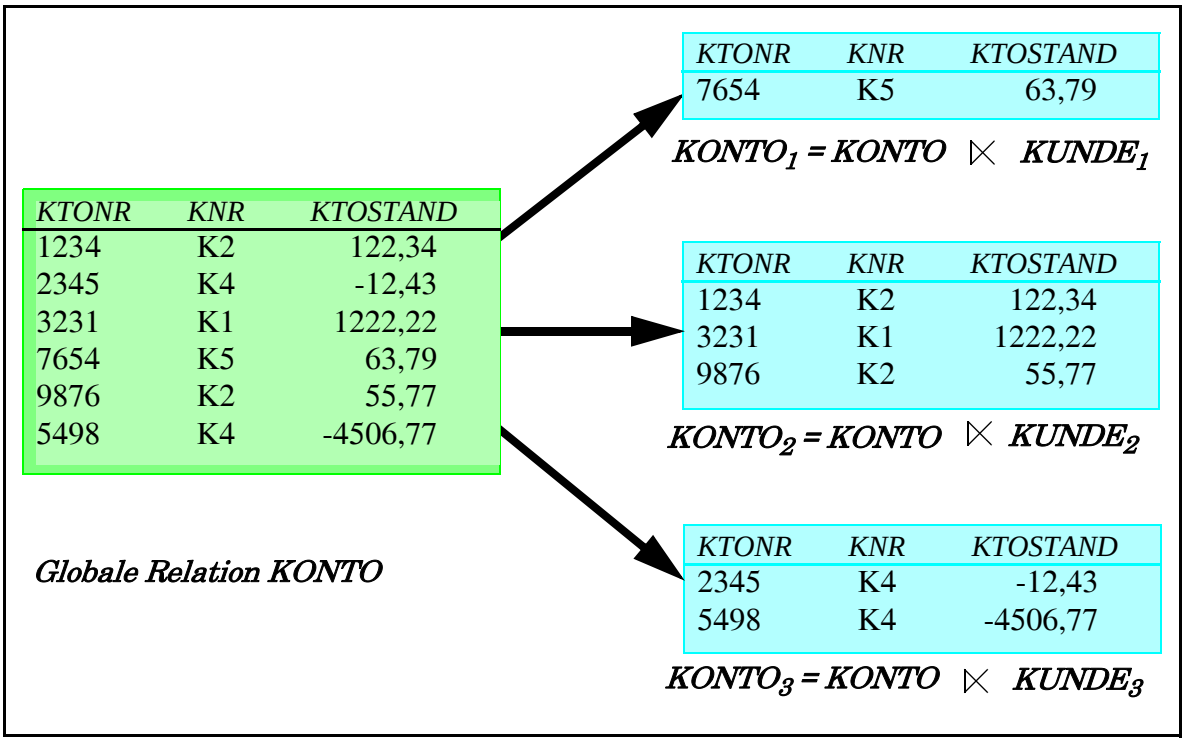
5.3.2 Abgeleitete horizontale Fragmentierung

Es kann sinnvoll sein, die Definition einer horizontalen Fragmentierung nicht auf Attributen der zu zerlegenden Relation vorzunehmen, sondern auf die horizontale Fragmentierung einer anderen Relation abzustimmen. In diesem Fall spricht man von einer abgeleiteten horizontalen Fragmentierung (derived horizontal fragmentation). Dies empfiehlt sich vor allem bei Relationen, die über eine Fremdschlüssel-Primärschlüssel-Beziehung von einer anderen Relation abhängen.

Beispiel 5-2

In unserem Bankbeispiel (Abb. 2-1) besteht eine Abhängigkeit zwischen der KONTO- und der KUNDE-Relation über das Kundennummer-Attribut KNR. Dabei ist KNR ein Fremdschlüssel in KONTO, der sich auf den Primärschlüssel von KUNDE bezieht. Eine abgeleitete horizontale Fragmentierung von KONTO orientiert sich daher an der horizontalen Fragmentierung von KUNDE, das heißt, die den Kunden eines Fragmentes zugeordneten Konten werden ebenfalls innerhalb eines Fragmentes zusammengefaßt. Wenn, wie in der Fall, KUNDE nach Filialen horizontal fragmentiert ist, werden somit alle Konten nach der Filialzugehörigkeit ihrer Kontoinhaber fragmentiert, wie in Abb. 5-3 verdeutlicht. Das Fragment $KONTO_2$ enthält z.B. alle Konten von Kunden der Filiale "F" (Frankfurt).

Abb. 5-3: Abgeleitete horizontale Fragmentierung der Kontorelation



Die einem Fragment R_i der übergeordneten Relation R zugeordneten Tupel der abhängigen Relation S erhält man durch Anwendung des Semi-Joins zwischen S und R_i. Bei einer Zerlegung von R in n Fragmente R_i ergibt sich also folgende abgeleitete horizontale Fragmentierung von S in n Fragmente S_i:

$$S_i = S \ltimes R_i = S \ltimes \sigma_{Pi}(R)$$
$$= \pi_{S\text{-}Attribute}(S \bowtie \sigma_{Pi}(R)) \qquad (1 \leq i \leq n)$$

Die zur Definition der Fragmentierung verwendeten Selektionsprädikate P_i beziehen sich hierbei ausschließlich auf Attribute von R, also nicht auf Attribute der zu zerlegenden (abhängigen) Relation S. Die Rekonstruktion der globalen Relation S ergibt sich analog wie für R durch Vereinigung der einzelnen Fragmente S_i. Die Vollständigkeit der Zerlegung ist gewährleistet, falls der Semi-Join zwischen S und R verlustfrei ist, da dann zu jedem S-Tupel ein Verbundpartner in R existiert. Die Verlustfreiheit des Semi-Joins ist gegeben, wenn für den Fremdschlüssel in S keine Nullwerte zulässig sind, da dann aufgrund der referentiellen Integrität ein Verbundpartner in R garantiert ist*. In unserem Bankbeispiel ist dies der Fall, da für den Fremdschlüssel KNR in KONTO (Kontoinhaber) sinnvollerweise keine Nullwerte zugelassen werden. Die Disjunktheit der S-Fragmente ergibt sich aus der Disjunktheit der R-Fragmente, da zwischen den beiden Relationen eine N:1-Beziehung besteht.

Ein wesentlicher Vorteil der abgeleiteten horizontalen Fragmentierung liegt darin, daß sie eine effiziente (Equi-)Join-Berechnung zwischen S und R unterstützt. Denn durch die Fragmentierung ist gewährleistet, daß alle zum Fragment R_i gehörenden Tupel der Relation S in Fragment S_i liegen (und umgekehrt). Werden die Fragmente R_i und S_i jeweils dem gleichen Rechner zugeordnet, kann daher eine lokale Join-Berechnung erfolgen. Ferner wird eine lokale Überprüfung der referentiellen Integrität möglich.

Beispiel 5-3

In dem eingeführten Bankbeispiel mit abgeleiteter horizontaler Fragmentierung der KONTO-Relation können Join-Berechnungen zwischen KUNDE und KONTO weitgehend lokal erfolgen, falls die entsprechenden Fragmente jeweils demselben Rechner zugeordnet werden. Dies trifft für die Bestimmung von Kontoangaben zu einem Kunden ebenso zu wie die Ermittlung von Kundenangaben zu einem bestimmten Konto.
Auch die referentielle Integrität kann lokal überwacht werden. Wird ein neues Konto eröffnet (bzw. der Kontoinhaber eines bestehenden Kontos geändert), kann die Gültigkeit der anzugebenden Kundennummer (KNR) lokal geprüft werden. Bei Löschungen in der Kundenrelation (bzw. Änderungen der Kundennummer) können die davon betroffenen Kontosätze lokal ermittelt werden. Somit läßt sich die Zulässigkeit der Operation lokal überprüfen bzw. Folgeänderungen in der Kontorelation können lokal durchgeführt werden.

* Bei Zulässigkeit von Nullwerten wird eine Erweiterung bei der Definition der abgeleiteten horizontalen Fragmentierung erforderlich (s. Übungsaufgaben).

5.3.3 Unterstützung von Parallelverarbeitung

Die horizontale Fragmentierung ist hervorragend geeignet, eine *parallele Anfragebearbeitung* zu unterstützen. Denn Operationen auf einer derart fragmentierten globalen Relation lassen sich leicht in eine Menge parallel ausführbarer Teiloperationen auf den einzelnen Fragmenten zerlegen. Das Gesamtergebnis erhält man durch anschließendes Mischen der Teilergebnisse. Bei einer abgeleiteten horizontalen Fragmentierung läßt sich auch die Join-Berechnung einfach parallelisieren. Hierzu genügt es, den Join auf jedem der n Fragmentpaare der beiden Relationen lokal und parallel durchzuführen und anschließend die lokalen Join-Ergebnisse zu mischen.

Beispiel 5-4

Auf der KONTO-Relation aus Beispiel 5-2 sei die Summe sämtlicher Kontostände zu berechnen. Da dies das Lesen jedes Kontosatzes erfordert (Relationen-Scan), ergibt sich bei sequentieller Bearbeitung eine sehr hohe Bearbeitungsdauer. Eine horizontale Fragmentierung wie in Abb. 5-3 erlaubt eine parallele Bearbeitung der Anfrage. Dabei wird auf jedem Fragment (für jede Filiale) parallel die lokale Summe der Kontostände ermittelt. Das Gesamtergebnis ergibt sich durch die abschließende Summenbildung der lokalen Zwischenergebnisse.

Zur Unterstützung einer parallelen Anfragebearbeitung werden vor allem zwei Arten einer horizontalen Fragmentierung bevorzugt, eine sogenannte *Bereichsfragmentierung* sowie eine *Hash-Fragmentierung*. In beiden Fällen erfolgt die Zerlegung i.a. auf einem ausgezeichneten Fragmentierungsattribut. Bei der Bereichsfragmentierung wird durch Festlegung von Wertebereichen für dieses Attribut die Zuordnung von Tupeln zu Fragmenten festgelegt, während bei der Hash-Fragmentierung die Zuordnung über eine Hash-Funktion definiert wird. Die in Beispiel 5-2 verwendete Fragmentierung entspricht einer speziellen Bereichsfragmentierung auf dem Fragmentierungsattribut "Filiale". Generell wird versucht, etwa gleich große Fragmente zu erzeugen, damit bei der Parallelverarbeitung eine in etwa gleich schnelle Bearbeitung auf den einzelnen Fragmenten ermöglicht wird. Wir werden auf die Datenverteilung zur Unterstützung von Parallelverarbeitung in Kap. 17 noch näher eingehen.

5.4 Vertikale Fragmentierung

Die *vertikale Fragmentierung* zerlegt eine Relation spaltenweise durch Definition von Projektionen auf den Attributen der Relation. Die Forderungen nach Vollständigkeit und Rekonstruierbarkeit verlangen, das jedes Attribut in wenigstens einem Fragment enthalten ist. Die Rekonstruktion der globalen Relation erfordert den natürlichen Verbund (Join) zwischen den einzelnen Fragmenten. Um diese Join-Berechnung verlustfrei vornehmen zu können, ist es jedoch erforderlich, den Primärschlüssel der globalen Relation (bzw. einen sonstigen Schlüsselkandidaten)

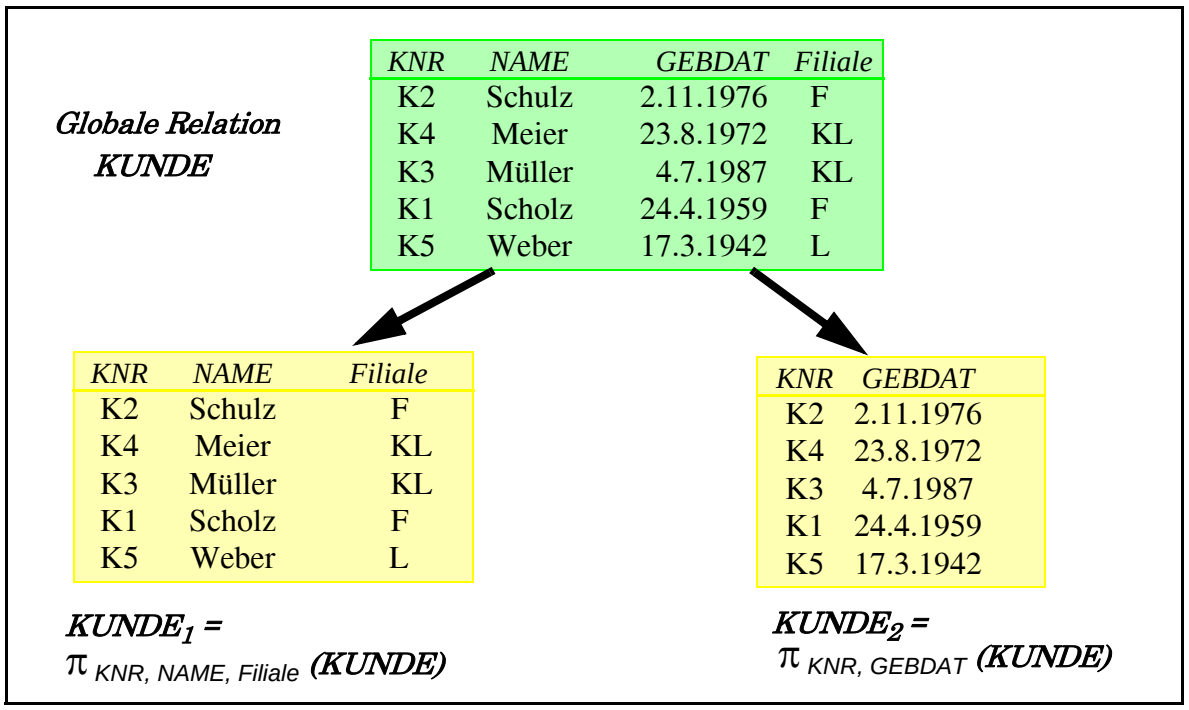
in jedem Fragment mitzuführen. Die Forderung nach Disjunktheit muß also bei der vertikalen Fragmentierung auf die Nicht-Primärattribute (Attribute, die nicht Teil des Primärschlüssels sind) eingeschränkt werden.

Beispiel 5-5

Die Kundenrelation aus unserer Bankanwendung wurde in Abb. 5-4 durch Anwendung von Projektionen in zwei vertikale Fragmente $KUNDE_1$ und $KUNDE_2$ zerlegt. Der Primärschlüssel KNR ist in beiden Fragmenten enthalten, um die Gesamt-Relation durch Join-Bildung wieder rekonstruieren zu können. Es gilt

$$KUNDE = KUNDE_1 \bowtie KUNDE_2.$$

Abb. 5-4: Vertikale Fragmentierung der Kundenrelation



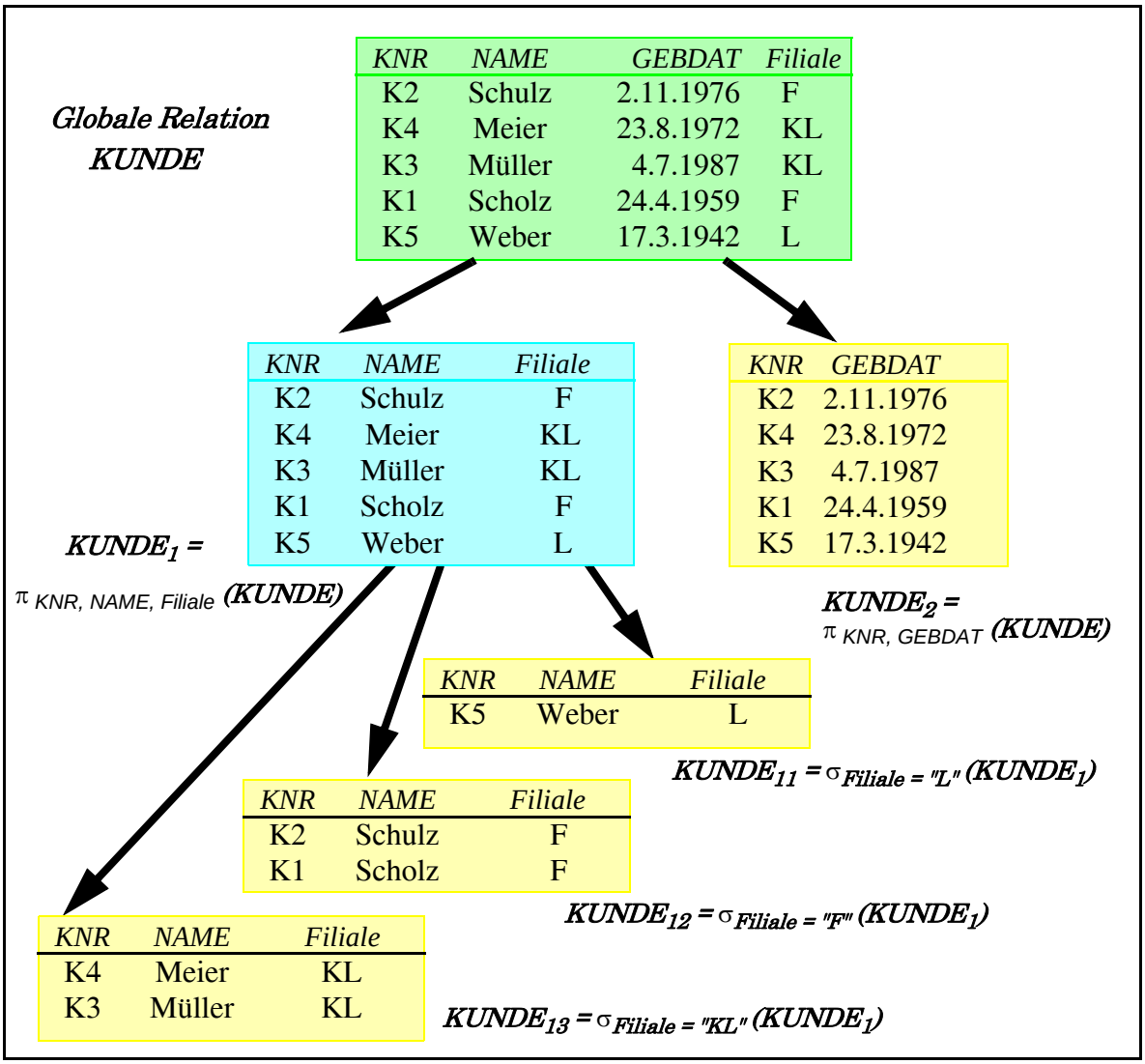
Die vertikale Fragmentierung kommt vor allem für Relationen mit einer großen Anzahl von Attributen zum Tragen, wenn an verschiedenen Knoten vorwiegend unterschiedliche Teilmengen der Attribute benötigt werden. In diesem Fall kann durch die vertikale Fragmentierung eine Reduzierung des Kommunikationsaufwandes erreicht werden sowie eine Einschränkung des zu verarbeitenden Datenumfanges. Auf der anderen Seite ist die Rekonstruktion der vollständigen Tupel über Verbunde teuer. Sehr aufwendig sind auch Änderungsvorgänge wie Einfügen oder Löschen von Tupeln, da sie sämtliche vertikalen Fragmente einer Relation betreffen.

5.5 Hybride Fragmentierung

Da die Fragmente einer Relation selbst wiederum Relationen darstellen, kann die Fragmentierung rekursiv fortgesetzt werden, sofern die Korrektheit in jedem

Fragmentierungsschritt gewahrt bleibt. Insbesondere können dabei die unterschiedlichen Fragmentierungsarten kombiniert werden, wobei man dann von einer *hybriden Fragmentierung* der globalen Relation spricht. Die Wiederherstellung einer derart zerlegten Relation erfordert die Ausführung der Rekonstruktionsschritte in der inversen Reihenfolge, in der die Fragmentierungen vorgenommen wurden.

Abb. 5-5: Hybride Fragmentierung der Kundenrelation



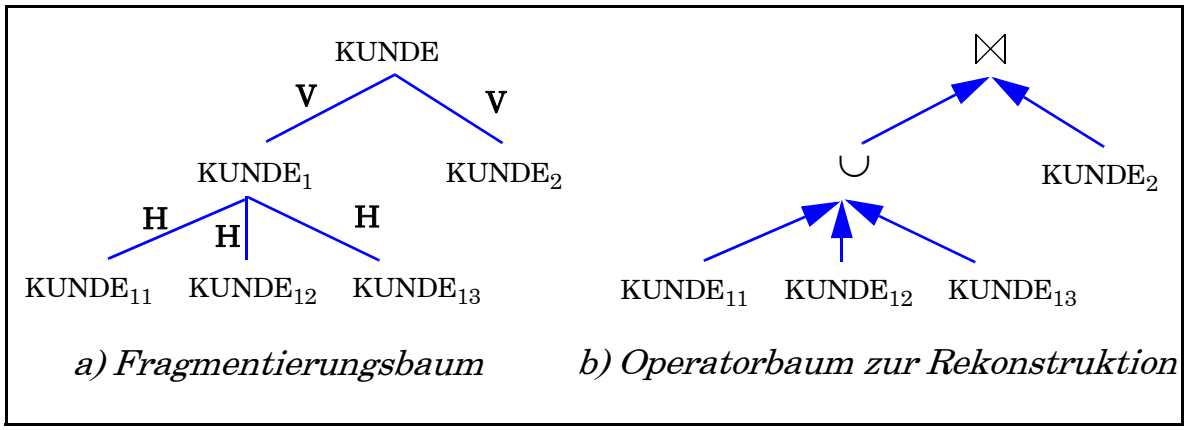
Beispiel 5-6

Die Kundenrelation wird zunächst wie in Abb. 5-4 vertikal in die Fragmente $KUNDE_1$ und $KUNDE_2$ zerlegt werden. Anschließend erfolgt auf Fragment $KUNDE_1$ eine horizontale Zerlegung nach Filialen, wodurch die Fragmente $KUNDE_{11}$, $KUNDE_{12}$ und $KUNDE_{13}$ entstehen (Abb. 5-5). Zur Rekonstruktion der Gesamt-Relation ist zunächst eine Vereinigung der durch die horizontale Fragmentierung erzeugten Fragmente erforderlich, danach wird die vertikale Fragmentierung über Join-Bildung aufgehoben. Es gilt also

$$KUNDE = (KUNDE_{11} \cup KUNDE_{12} \cup KUNDE_{13}) \bowtie KUNDE_2.$$

Die hybride Fragmentierung läßt sich in kompakter Weise durch einen *Fragmentierungsbaum* beschreiben. Dabei bildet die globale Relation den Wurzelknoten; die Blätter entsprechen den erzeugten Fragmenten. Die Nachfolger eines bestimmten Knotens im Baum entsprechen den aus einem Fragmentierungsschritt entstandenen Zwischenergebnissen. Abb. 5-6a zeigt den Fragmentierungsbaum für die hybride Fragmentierung aus Beispiel 5-6. Wie in Abb. 5-6b veranschaulicht, läßt sich aus dem Fragmentierungsbaum unmittelbar ein Operatorbaum zur Rekonstruktion der globalen Relation ableiten. Dabei erfolgt die Verarbeitung von den Blättern (Fragmenten) aus unter Anwendung der in den Knoten des Baumes spezifizierten Operatoren.

Abb. 5-6: Baumdarstellung von Fragmentierung und Rekonstruktion



5.6 Fragmentierungstransparenz

Eine Forderung an Verteilte Datenbanksysteme ist die Unterstützung von Fragmentierungstransparenz (Kap. 4.1), unabhängig davon, welche der vorgestellten Alternativen zur Fragmentierung eingesetzt wird. Dies bedeutet, daß die vorgenommene Zerlegung für den Endbenutzer und Anwendungsprogrammierer vollkommen unsichtbar bleibt; es ist alleinige Aufgabe des Verteilten DBS Datenbankoperationen auf die einzelnen Fragmente abzubilden.

Beispiel 5-7

Die Kundenrelation sei wie in horizontal in die drei Fragmente $KUNDE_1$, $KUNDE_2$ und $KUNDE_3$ unterteilt. Um den Namen des Kunden mit KNR=K4 zu ermitteln, kann bei Gewährleistung von Fragmentierungstransparenz die Anfrage wie im zentralen Fall formuliert werden:

SELECT NAME FROM KUNDE WHERE KNR=K4.

Ohne Fragmentierungstransparenz dagegen müßte u.U. auf jedes der drei Fragmente explizit zugegriffen werden:

SELECT NAME FROM $KUNDE_1$ WHERE KNR=K4;

IF NOT-FOUND THEN
SELECT NAME FROM $KUNDE_2$ WHERE KNR=K4;
IF NOT-FOUND THEN
SELECT NAME FROM $KUNDE_3$ WHERE KNR=K4;

Noch signifikanter sind die Vorteile der Fragmentierungstransparenz bezüglich Änderungen, die dazu führen können, daß Tupel von einem Fragment in ein anderes verlegt werden (migrieren) müssen. Wenn etwa der Kunde K3 aufgrund eines Wohnungswechsels die Filialzugehörigkeit ändert, so ist dies bei Fragmentierungstransparenz eine einfache Operation, z.B.:

UPDATE KUNDE SET Filiale = "L" WHERE KNR=K3;

Ohne Fragmentierungstransparenz dagegen müßte ein Löschen von K3 in $KUNDE_3$ und Einfügen in $KUNDE_1$ vorgenommen werden:

SELECT NAME, GEBDAT INTO :Name, :Gebdat
FROM $KUNDE_3$ WHERE KNR =K3;

INSERT INTO $KUNDE_1$ (KNR, NAME, GEBDAT, Filiale)
VALUES (K3, :Name, :Gebdat, "L");

DELETE $KUNDE_3$ WHERE KNR=K3;

Noch aufwendiger wird die Programmierung, wenn auch abgeleitete horizontale Fragmentierungen von Änderungen betroffen sind, z.B. die Umverteilung aller Konten des Kunden, der die Filialzugehörigkeit gewechselt hat. Bei Gewährleistung von Fragmentierungstransparenz werden solche Wartungsoperationen automatisch durch das DBS durchgeführt.

Das Beispiel verdeutlicht, daß Fragmentierungstransparenz eine erhebliche Erleichterung der DB-Benutzung bedeutet. Weiterhin kann nun die Fragmentierung jederzeit geändert werden, ohne daß sich dadurch Rückwirkungen auf bestehende Anwendungen ergeben.

5.7 Bestimmung der Datenverteilung

Bei der Bestimmung von Fragmentierung und Allokation gilt es, sowohl eine hohe Leistungsfähigkeit als auch eine hohe Verfügbarkeit (durch Replikation) zu unterstützen. Diese Aufgabe ist sehr komplex, da eine Vielzahl von Abhängigkeiten eingehen, die bestenfalls in grober Annäherung berücksichtigt werden können. Ein grundsätzliches Problem besteht darin, daß die Kosten der Ausführung nicht nur durch die Datenverteilung bestimmt sind, sondern auch von den eingesetzten Verfahren zur Anfrageoptimierung und -Bearbeitung, zur Transaktionsverwaltung und zur Wartung von Replikation. Weiterhin ist die Last, für die eine günstige Datenverteilung zu finden ist, i.a. nur ungenau bekannt; insbesondere kann die Datenverteilung nicht auf die Ausführung von Ad-hoc-Anfragen hin optimiert werden. Aussagen hinsichtlich der Ausfallwahrscheinlichkeit einzelner Knoten oder Kommunikationsverbindungen sind generell nur schwer möglich.

Die Bestimmung der Datenverteilung muß zwangsweise gegensätzliche Anforderungen ausgleichen. Kompromisse müssen v.a. hinsichtlich folgender Kriterien eingegangen werden:

- *Lese- vs. Änderungsoperationen*
  Leseoperationen können durch Einsatz von Replikation stark optimiert werden, jedoch auf Kosten von Änderungstransaktionen. Je höher also der Anteil von Lesezugriffen liegt, desto stärker kann Replikation zur Verbesserung des Leistungsverhaltens sowie der Verfügbarkeit eingesetzt werden.
- *Lokalität vs. Parallelität*
  Zur Reduzierung von Kommunikationsvorgängen empfiehlt sich eine Datenverteilung, die Lokalität im Referenzverhalten dadurch ausnutzt, daß häufig gemeinsam benötigte Daten demselben Rechner zugewiesen werden (Cluster-Bildung). Zur Parallelisierung von Operationen ist es dagegen erforderlich, daß gemeinsam benutzte Daten mehreren Rechnern zugewiesen werden (Declustering).
  Dieser Zielkonflikt (Trade-off) ist zudem noch vom Lasttyp beeinflußt. Die Unterstützung von Parallelisierung empfiehlt sich vor allem für Operationen auf großen Datenmengen, während für einfachere Operationen die Unterstützung von Lokalität anzustreben ist.
- *Lokalität vs. Lastbalancierung*
  Die Datenverteilung zur Maximierung von Lokalität führt leicht zu ungleicher Rechnerauslastung. Zur Reduzierung von lokalen Überlastsituationen gilt es, die Daten so aufzuteilen, daß die Rechner in etwa gleichmäßig belastet werden.

5.7.1 Festlegung der Fragmentierung

Zur Festlegung einer (primären) horizontalen Fragmentierung gilt es, die definierenden Selektionsprädikate zu spezifizieren. In vielen Fällen genügt hierzu die Definition von Wertebereichen auf einem der Attribute (Bereichsfragmentierung). Die Festlegung einer solchen primären Fragmentierung ist durch eine Anwendungsanalyse oft relativ einfach. In unserem Bankbeispiel bot sich so die Fragmentierung nach Filialen an, um eine Unterstützung geographischer Zugriffslokalität zu ermöglichen. Ähnliches gilt für andere geographisch zu verteilende Datenbestände, z.B. die Fragmentierung von Abteilungsdaten nach Abteilungsorten. Durch Analyse von Fremdschlüssel-Primärschlüssel-Beziehungen sowie Join-Häufigkeiten kann darauf aufbauend eine Definition von abgeleiteten horizontalen Fragmentierungen erfolgen.

Ein solcher "semantischer" Ansatz wurde auch in [BG92] empfohlen. Alternativ dazu kann versucht werden, ein mathematisches Modell zur Festlegung der Selektionsprädikate heranzuziehen, wie z.B. in [CP84, ÖV91] vorgestellt. Diese Ansätze erfordern die Festlegung, welche "Minterme" (dies sind Kombinationen einfacher Prädikate der Form <Attribut> <Vergleichsoperator> <Konstante> in konjunktiver Normalform) in welcher Häufigkeit und Selektivität auf einer Relation auszuführen sind. Mit diesen Angaben wird dann eine Zerlegung bestimmt, so daß ein Fragment nicht in unterschiedlicher Weise von zwei (oder mehr) der unterstellten

Anfragetypen referenziert wird und sich für die Tupel eines Fragmentes eine in etwa gleiche Zugriffshäufigkeit ergibt. Algorithmen zur Festlegung einer vertikalen Fragmentierung versuchen, eine Partitionierung der Attributmengen zu finden, so daß Anfragetypen verschiedener Knoten möglichst disjunkte Teilmengen der Attribute benötigen [CP84, ÖV91].

5.7.2 Festlegung der Allokation

Die Bestimmung einer Datenallokation in verteilten Systemen ist ein intensiv untersuchtes Optimierungsproblem [DF82, Wa84, Ap88, HR88]. Das Allokationsproblem wurde meist für die Zuordnung ganzer Dateien betrachtet (file allocation problem, FAP); die entwickelten Ansätze lassen sich oft jedoch auch zur Allokation von Fragmenten heranziehen. Das Optimierungsziel ist i.a. die Minimierung einer globalen Kostenfunktion unter Berücksichtigung bestimmter Randbedingungen. Bei der Ausgestaltung von Kostenfunktion sowie den betrachteten Randbedingungen bestehen erhebliche Unterschiede zwischen den einzelnen Ansätzen. Im einfachsten Fall wird lediglich eine Minimierung von Kommunikationsvorgängen durch Unterstützung von Lokalität angestrebt; komplexere Modelle berücksichtigen daneben auch die Speicherkosten sowie lokale Verarbeitungskosten. Mögliche Randbedingungen sind die Beachtung knotenspezifischer Obergrenzen hinsichtlich der Speicher- oder Verarbeitungskapazität, um z.B. eine Lastbalancierung zu unterstützen.

Um die prinzipielle Vorgehensweise zu veranschaulichen, soll im folgenden ein einfacher Ansatz zur Datenallokation skizziert werden. Wir beschränken uns dabei auf die partitionierte Allokation von Fragmenten (keine Replikation)*. Als Eingabe werden folgende Größen benötigt:

- Anzahl der Transaktionstypen L, Anzahl der Knoten N, Anzahl der Fragmente M
- Lastverteilung W ($N*L$ Einträge). Eintrag $W(n, l)$ gibt an, mit welcher Häufigkeit Transaktionstyp l an Knoten n gestartet wird
- Referenzmatrix R ($L*M$ Einträge): Zugriffsverteilung zwischen Transaktionstypen und Fragmenten. Eintrag $R(l, m)$ gibt an, wieviele Zugriffe pro Ausführung von Transaktionstyp l im Mittel auf Fragment m entfallen
- Mittlere Anzahl von Instruktionen für eine lokale DB-Referenz: I_{ref}
- Mittlere Anzahl von Instruktionen zur Kommunikation, die pro externem Zugriff im beauftragenden sowie im ausführenden Rechner anfallen: I_{komm}
- CPU-Kapazität von Knoten n: $C(n)$ $(1 \le n \le N)$
- maximale CPU-Auslastung pro Knoten: u_{max} $(0 < u_{max} < 1)$.

* Nach Festlegung der partitionierten Allokation könnte in einem zusätzlichen Schritt überprüft werden, für welche Fragmente eine Replikation sinnvoll ist. Unter Leistungsgesichtspunkten wäre dabei eine zusätzliche Allokation an einen Rechner sinnvoll, wenn die damit verbundenen Speicher- und Änderungskosten durch die erwarteten Vorteile für Leser aufgewogen würden.

Als Ausgabe soll eine Allokation von Fragmenten zu Rechnern bestimmt werden:

- Allokationsmatrix *A* (*N*M* Einträge). Eintrag *A (n, m)* = 1 (0), falls Fragment *m* an Knoten *n* (nicht) gespeichert werden soll.

Als Optimierungsziel streben wir dabei die Minimierung des Kommunikationsaufwandes an, d.h. die Maximierung lokaler Fragmentzugriffe. Auf Fragment m besteht insgesamt folgende Zugriffsfrequenz:

$$ZF(\boldsymbol{m}) = \sum_{n=1}^{N} \sum_{l=1}^{L} W(n, l) \times R(l, \boldsymbol{m})$$

Davon sind jedoch nur die Zugriffe in dem Knoten *n* lokal, dem Fragment *m* zugewiesen wurde, das heißt, für den *A (n,m)* = 1 gilt. Damit ergibt sich folgende Häufigkeit lokaler Zugriffe auf Fragment m:

$$ZFL(\boldsymbol{m}) = \sum_{n=1}^{N} \sum_{l=1}^{L} W(n, l) \times R(l, \boldsymbol{m}) \times A(n, \boldsymbol{m})$$

Das Optimierungsziel, die Summe lokaler Fragmentzugriffe zu maximieren, lautet demnach:

$$\sum_{m=1}^{M} ZFL(m) = Max!$$

Dabei sind zwei Nebenbedingungen einzuhalten. Zum einen muß jedes Fragment genau einem Knoten zugeordnet werden (partitionierte Allokation):

$$\sum_{n=1}^{N} A(n, m) = 1 \qquad (\forall \text{ m } | \ \ldots 1 \leq \text{m} \leq$$

Daneben muß eine Lastbalancierung erreicht werden, das heißt, kein Rechner darf überlastet werden. Die Auslastung eines Rechners setzt sich in unserem Modell aus drei Komponenten, *L1, L2* und *L3,* zusammen. Der erste Lastanteil *L1* entsteht durch die eigentlichen Fragmentzugriffe, die aufgrund der Datenallokation an Rechner *n* auszuführen sind:

$$L1(\boldsymbol{n}) = I_{ref} \times \sum_{m=1}^{M} ZF(m) \times A(\boldsymbol{n}, m)$$

Lastanteil *L2* entsteht durch den Kommunikationsaufwand für Zugriffe auf lokale Datenfragmente durch externe Transaktionen:

$$L2(\boldsymbol{n}) = I_{komm} \times \sum_{m=1}^{M} (ZF(m) - ZFL(m)) \times A(\boldsymbol{n}, m)$$

Der dritte Anteil *L3* schließlich entsteht für Kommunikationsvorgänge lokaler Transaktionen, um auf extern allokierte Fragmente zuzugreifen:

$$L3(\boldsymbol{n}) = I_{komm} \times \sum_{m=1}^{M} \sum_{l=1}^{L} W(\boldsymbol{n}, l) \times R(l, m) \times (1 - A(\boldsymbol{n}, m))$$

Die zweite Nebenbedingung lautet somit insgesamt

$$L1(n) + L2(n) + L3(n) < u_{max} \times C(n) \qquad (\forall \text{ n } | \ \ldots 1 \leq \text{n} \leq$$

Die exakte Lösung dieses ganzzahligen Optimierungsproblems ist sehr aufwendig, so daß sich die Anwendung einfacherer Heuristiken empfiehlt, welche zu geringen Berechnungskosten eine gute Lösung liefern. Wir skizzieren dazu im folgenden eine einfache Strategie. Wir benötigen dazu Hilfsvariablen CU_n, die für jeden Rechner die CPU-Belastung festhalten, die sich aufgrund der bereits vorgenommenen Allokationen ergibt. Der Algorithmus läuft in folgenden Schritten ab:

1. Berechne aus der Lastverteilung *L* und der Referenzverteilung *R* zunächst für jedes Fragment m die Zugriffshäufigkeit *ZF (m)*.
2. Bestimme das nächste, noch nicht allokierte Fragment m mit der höchsten Zugriffsfrequenz *ZF (m)*.
3. Berechne für jeden Rechner die lokale Zugriffsfrequenz *ZFL (m)*, die sich durch Allokation von Fragment m an diesen Rechner ergibt, ebenso die sich daraus ergebende Erhöhung der CPU-Auslastung.
4. Ordne das Fragment demjenigen Rechner zu, für den sich der größte Anteil lokaler Zugriffe ergibt, ohne daß sich ein Überschreiten der maximalen CPU-Auslastung ergibt. Ergänze die Allokationsmatrix *A* und die *CU*-Werte für die getroffene Allokation.
5. Falls noch weitere Fragmente zuzuordnen sind, gehe zu Schritt 2. Ansonsten ist die Berechnung der Allokation beendet.

Beispiel 5-8

Der vorgestellte Allokationsansatz soll auf die in Abb. 5-7 gezeigte Last- und Referenzverteilung angewendet werden (*L*=4, *N*=3, *M*=5). Wir nehmen ferner an, daß I_{ref} = 10000, I_{komm} = 2500, *C(n)*=30 MIPS und u_{max} = 0.8 gilt (jeder Rechner soll also höchstens zu 80% ausgelastet werden). In Schritt 1 berechnen wir zunächst die Zugriffshäufigkeiten auf die Fragmente, wobei sich das in Abb. 5-7c gezeigte Resultat ergibt. Da die Fragmente in der Reihenfolge ihrer Zugriffshäufigkeit allokiert werden, ist zunächst Fragment F4 zuzuord-

nen. Von den 1400 Zugriffen pro Sekunde auf diesem Fragment können 210 in Rechner R1, 400 in R2 und 790 in R3 lokal bearbeitet werden, wenn F4 dem jeweiligen Knoten zugeordnet wird. Somit wird F4 Rechner R3 zugewiesen, wodurch sich dort bereits eine Last von CU_3=15,525 MIPS ergibt (14 MIPS für die Fragmentzugriffe, der Rest für Kommunikation aufgrund von externen F4-Zugriffen). Kommunikationsbelastungen für Zugriffe auf F4 fallen auch in R1 und R3 an: CU_1=0,525 MIPS, CU_2=1,0 MIPS.

Bei der Zuweisung von Fragment F1 ergibt sich für R2 der höchste Anteil lokaler Zugriffe. Nach dieser Allokation besteht folgende Lastsituation: CU_1=1,5875 MIPS; CU_2=14,0 MIPS; CU_3=16,0625 MIPS. Fragment F5 geht danach an R1 (550 von 1010 Zugriffen lokal), woraus folgende Auslastungswerte resultieren: CU_1=12,8375 MIPS; CU_2=14,75 MIPS; CU_3=16,4625 MIPS.

Für die Allokation von F3 ergäbe sich anschließend für Rechner R2 der höchste Anteil lokaler Verarbeitung, jedoch würde sich durch diese Allokation die Auslastung um 10,7 MIPS in R2 erhöhen, so daß der Auslastungsgrenzwert von 24 MIPS überschritten wäre. Die Zuordnung zum nächstbesseren Kandidaten R1 kann dagegen durchgeführt werden. Danach gilt CU_1=23,7125 MIPS; CU_2=15,8 MIPS; CU_3=16,8875 MIPS. Abschließend erfolgt noch die Allokation von F2 zu Rechner R2, wodurch sich insgesamt folgende projezierte Rechnerauslastung ergibt: CU_1=23,7125 MIPS (79 %); CU_2=20,45 MIPS (68 %); CU_3=17,3375 MIPS (58%).

Die Allokation ordnet also R1 die Fragmente F3 und F5, R2 die Fragmente F1 und F2 sowie R3 Fragment F4 zu. Der Anteil lokaler Zugriffe ist für die Beispiellast relativ gering (49,45%), so daß ein hoher Kommunikations-Overhead entsteht (12,4 MIPS der insgesamt 61,5 MIPS).

Abb. 5-7: Beispiel einer Last- und Referenzverteilung

W	R1	R2	R3
T1	5	10	
T2		8	6
T3	3		7
T4	5		2

a) Lastverteilung W
(Aufrufhäufigkeiten pro Sekunde)

R	F1	F2	F3	F4	F5
T1	50		30		30
T2		30	15	50	
T3	25			70	
T4	20		40		80

b) Referenzverteilung R

m	F1	F2	F3	F4	F5
ZF(m)	1140	420	940	1400	1010

c) Zugriffshäufigkeiten (pro Sekunde)

Im Beispiel wurden die Kommunikationskosten pro Zugriff (I_{komm}) relativ gering angesetzt, da i.a. nicht jede externe Referenz einen eigenen Kommunikationsvorgang verursacht, sondern i.a. mehrere Fragmentzugriffe pro (externer) Teiloperation durchgeführt werden. Die zur Lastbalancierung eingeführte Nebenbedingung sieht nicht vor, daß alle Rechner möglichst gleichmäßig ausgelastet werden, sondern daß die Auslastung pro Rechner einen bestimmten Grenzwert nicht über-

schreitet. Dieser weniger restriktive Ansatz gestattet generell einen höheren Anteil an lokalen Zugriffen.

Es ist möglich, daß der Algorithmus nicht erfolgreich zu Ende kommt, wenn nämlich in Schritt 4 für keinen der Rechner eine Zuordnung möglich ist, ohne daß seine Auslastungsgrenze überschritten wird. Dies deutet daraufhin, daß die vorgesehene CPU-Kapazität der Rechner für die unterstellte Last nicht ausreicht. Möglicherweise kann auch durch eine Verfeinerung der Fragmentierung und damit der zuzuordnenden Lastanteilen eine Lösung gefunden werden. Ein solcher Schritt kann auch sinnvoll sein, um den Anteil lokaler Zugriffe zu erhöhen. Voraussetzung für die erneute Anwendung des Allokationsalgorithmus' ist dann natürlich, daß die Referenzverteilung bezüglich der verfeinerten Fragmentierung bekannt ist bzw. bestimmt wird (z.B. durch Monitoring der tatsächlichen Referenzverteilung).

Der vorgestellte Allokationsansatz strebt vor allem die Maximierung lokaler Datenzugriffe an, nicht jedoch die Unterstützung von Parallelverarbeitung. Dies wird erst für lokal verteilte Shared-Nothing-Systeme untersucht (Kap. 17), da diese aufgrund der wesentlich effizienteren Kommunikation für die Parallelverarbeitung besser geeignet sind

Übungsaufgaben

Aufgabe 5-1: Korrektheit der abgeleiteten horizontalen Fragmentierung

Die in Kap. 5.3.2 vorgestellte Definition der abgeleiteten horizontalen Fragmentierung ist nur korrekt, falls in der abhängigen Relation S keine Nullwerte als Fremdschlüssel vorkommen dürfen. Wie ist die Fragmentierung zu erweitern, falls diese Voraussetzung nicht mehr zutrifft? Welche Auswirkungen ergeben sich dadurch hinsichtlich der Berechnung von Joins zwischen R und S ?

Aufgabe 5-2: Abgeleitete horizontale Fragmentierung

Gegeben seien folgende Relationen:

ABT (ANR, ANAME, AORT, ABUDGET)

PERSONAL (PNR, PNAME, ANR, BERUF, GEHALT)

PROJEKT (PRONR, PRONAME, PROBUDGET)

PMITARBEIT (PNR, PRONR, DAUER).

Es handelt sich um das klassische Beispiel von Abteilungen, Projekten und ihren Mitarbeitern. Primärschlüssel sind unterstrichen; übereinstimmende Attributnamen kennzeichnen Fremdschlüssel-Primärschlüsselbeziehungen.

Abteilungen seien an drei Standorten vertreten: Kaiserslautern, Frankfurt und Leipzig. Definieren Sie eine entsprechende horizontale Partitionierung von ABT sowie abgeleitete horizontale Partitionierungen für PERSONAL und PMITARBEIT. Welche Alternative besteht für PMITARBEIT zur Definition einer abgeleiteten horizontalen Fragmentierung ? Wann wäre diese sinnvoll ?

Aufgabe 5-3: Hybride Fragmentierung

Auf der horizontalen Fragmentierung von PERSONAL aus der vorherigen Aufgabe soll zusätzlich eine vertikale Fragmentierung durchgeführt werden, so daß das GEHALT-Attribut in einem eigenen Fragment geführt wird. Geben Sie die Spezifikation einer solchen vertikalen Fragmentierung an. Zeichnen Sie den Fragmentierungsbaum sowie den Operatorbaum zur Rekonstruktion der PERSONAL-Relation.

Aufgabe 5-4: Bestimmung der Datenallokation

Bestimmen Sie die Datenallokation für die gezeigte Last- und Referenzverteilung. Verwenden Sie ansonsten die Parameter aus Beispiel 5-8. Welcher Anteil der Referenzen kann lokal bearbeitet werden? Welcher Kommunikations-Overhead (in MIPS) ergibt sich insgesamt?

W	R1	R2	R3
T1		16	8
T2	15	6	
T3	7		10

R	F1	F2	F3	F4
T1	70	30		10
T2	5		50	
T3	5			80

6 Verteilte Anfragebearbeitung

Aufgabe der Anfragebearbeitung (Query Processing) ist es, für abzuarbeitende DB-Operationen möglichst effiziente Ausführungspläne zu erstellen und diese auszuführen. Für den DB-Benutzer hat die Verwendung eines Verteilten DBS keine Auswirkungen; er formuliert seine Anfragen (Queries) bzw. Änderungsoperationen wie im zentralen Fall mit einer deskriptiven Anfragesprache wie SQL unter Bezugnahme auf die logischen DB-Objekte des globalen konzeptionellen Schemas (GKS). Dagegen ist die physische Verteilung der Daten natürlich bei der Anfragebearbeitung eines Verteilten DBS zu berücksichtigen, um die benötigten Objekte referenzieren zu können. Die Ausführung von DB-Operationen muß damit auf die vorliegende Fragmentierung und Allokation der Daten abgestimmt werden. Zur Erstellung eines kostengünstigen Ausführungsplanes (Query-Optimierung) sind vor allem die durch Kommunikationsvorgänge eingeführten Kosten zusätzlich zu berücksichtigen. Weiterhin ergeben sich neue Optimierungsmöglichkeiten aufgrund replizierter Speicherung von Objekten sowie der Möglichkeit zur Parallelisierung. Diese Faktoren führen zu einer erheblichen Komplexitätssteigerung der Anfragebearbeitung im verteilten Fall. Die Qualität des Query-Optimierers ist von entscheidender Qualität für die Leistungsfähigkeit eines Verteilten Datenbanksystems. Es ist keine Seltenheit, daß die Kosten verschiedener (funktional gleichwertiger) Ausführungspläne um mehrere Größenordnungen auseinanderliegen.

Im folgenden geben wir zunächst einen Überblick über die wichtigsten Schritte bei der verteilten Anfragebearbeitung. In den drei anschließenden Teilkapiteln werden die wichtigsten Phasen genauer vorgestellt: Anfragetransformation (Kap. 6.2), die Erzeugung sogenannter Fragment-Ausdrücke (Kap. 6.3) sowie die globale Query-Optimierung (Kap. 6.4). Abschließend stellen wir dann einzelne Algorithmen zur verteilten Join-Berechnung vor (Kap. 6.5).

6.1 Überblick

Die Erstellung verteilter Ausführungspläne erfolgt üblicherweise im Rahmen mehrerer Phasen, wie sie in Abb. 6-1 dargestellt sind. Von Interesse sind hier lediglich verteilte oder *globale Anfragen*, welche Daten mehrerer Knoten betreffen, da anderenfalls die Anfragebearbeitung wie im lokalen Fall erfolgen kann*. Wie die Abbildung zeigt, läßt sich die Bearbeitung einer verteilten Anfrage in einen globalen sowie in einen lokalen Teil untergliedern. Dabei erfolgt die globale Anfragebearbeitung an einem dedizierten Koordinatorknoten unter Nutzung globaler Katalogdaten.** Die lokale Anfragebearbeitung findet an jedem Knoten statt, dessen Daten bei der Ausführung einer verteilten Anfrage benötigt werden. Die vier einzelnen Phasen der Anfragebearbeitung werden im folgenden überblicksartig vorgestellt.

Der erste Schritt der *Anfragetransformation* überführt DB-Operationen einer deskriptiven Anfragesprache wie SQL in eine interne Darstellung. Wir nehmen an, daß hierzu eine Transformation in einen äquivalenten Ausdruck der Relationenalgebra erfolgt. Diese Aufgabe kann weitgehend wie in zentralisierten DBS durchgeführt werden, da sich die Anfragen auf Objekte des globalen konzeptionellen Schemas (GKS) beziehen und noch keine Verteilungsinformation verwendet wird (für Katalogzugriffe kann jedoch Kommunikation erforderlich werden). Die Anfragezerlegung selbst unterteilt sich wiederum in mehrere Teilschritte wie Syntaxanalyse (Parsing), Namensauflösung, semantische Analyse und Normalisierung. Weiterhin können bereits algebraische Optimierungen zur Vereinfachung und Restrukturierung des relationalen Ausdrucks vorgenommen werden. Damit lassen sich schon vorab einige Ausführungspläne mit hohen Kosten ausschließen.

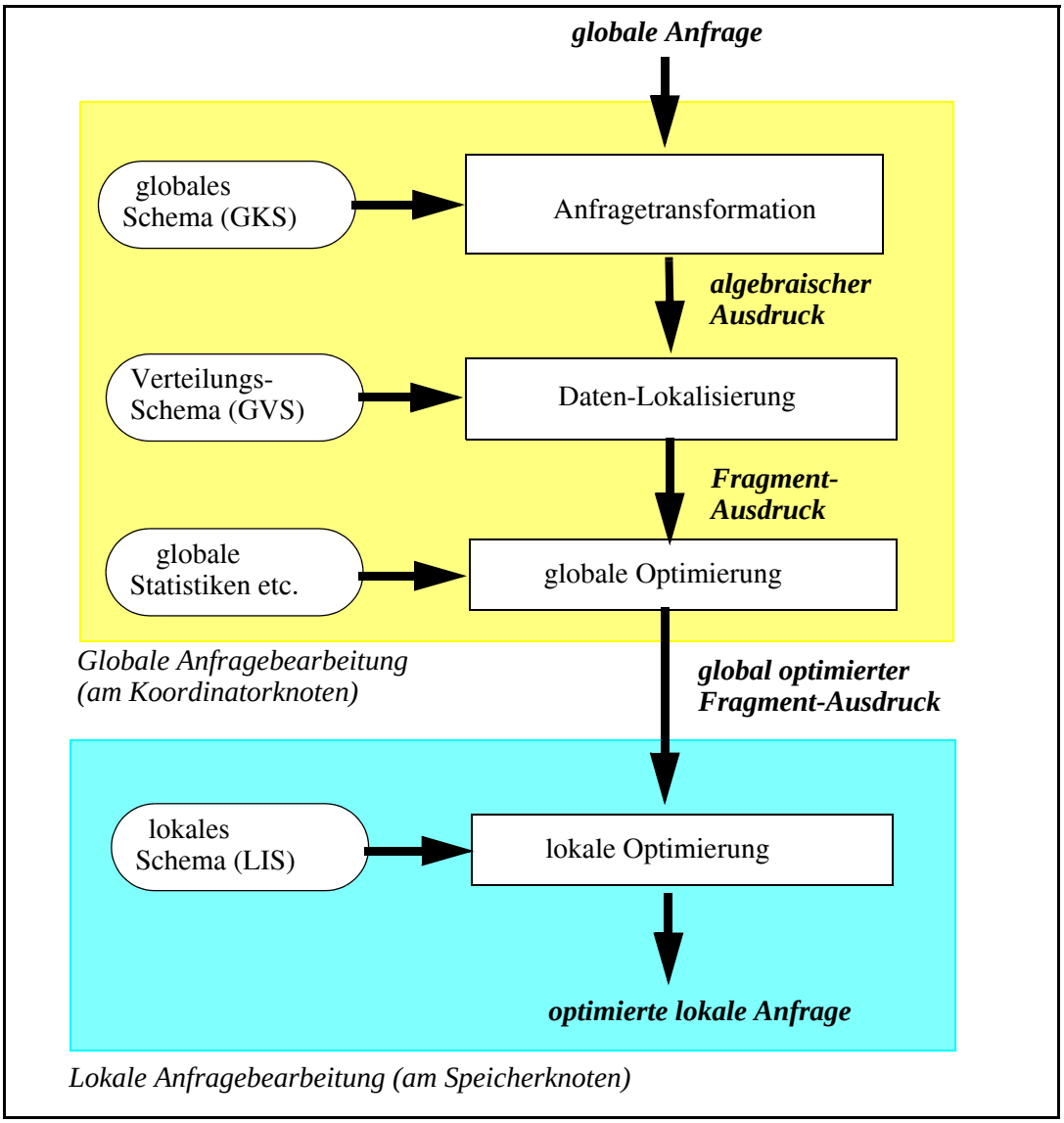
Der zweite Schritt der *Daten-Lokalisierung* verwendet die Angaben des globalen Verteilungsschemas, um die Knoten zu bestimmen, an denen sich Fragmente der zu referenzierenden Relationen befinden. Der im ersten Schritt erzeugte algebraische Ausdruck wird dazu in einen sogenannten *Fragment-Ausdruck* (Fragment-Anfrage) transformiert. Dabei wird für globale Relationen der jeweilige relationale Ausdruck zur Rekonstruktion der Relation aus ihren Fragmenten eingesetzt. Auf dem derart erzeugten Ausdruck werden wiederum algebraische Optimierungen zur Vereinfachung und zur effizienten Rekonstruktion angewendet.

---

* Üblicherweise sollten die Daten jedoch derart verteilt sein, daß lokale Anfragen häufiger als verteilte Anfragen vorkommen, um ein günstiges Leistungsverhalten zu erreichen. Hier steht jedoch gerade die Fähigkeit von Verteilen DBS im Vordergrund, DB-Operationen transparent auf verteilt gespeicherten Daten ausführen zu können.

** Im Falle von eingebetteten DB-Operationen, welche innerhalb von Anwendungsprogrammen aufgerufen werden, würde z.B. der Rechner, an dem die Übersetzung des Programms gestartet wird, die globale Anfragebearbeitung vornehmen. Für Ad-hoc-Anfragen bietet sich dafür der Rechner an, dem die Anfrage zur Ausführung zugeordnet wird.

Abb. 6-1: Phasen der verteilten Anfragebearbeitung [ÖV91]



Im Rahmen der *globalen Query-Optimierung* erfolgt danach die Festlegung eines aus globaler Sicht möglichst kostengünstigen Ausführungsplanes. Hierzu werden im Gegensatz zu den algebraischen Optimierungen Eigenschaften der einzelnen Fragmente (z.B. Kardinalitäten) sowie Kommunikationsoperationen berücksichtigt. Weiterhin ist die Ausführungsreihenfolge sowie der Ausführungsort der einzelnen Operationen so festzulegen, daß eine bestimmte Kostenfunktion minimiert wird. In dieser Kostenfunktion nehmen neben lokalen Anteilen für CPU, Speicherbedarf und E/A vor allem die Kommunikationskosten (Anzahl und Umfang der Nachrichten) eine wesentliche Stellung ein. Das Ergebnis dieses Schritts ist ein optimierter Fragment-Ausdruck, in dem die einzelnen Kommunikationsoperationen zusätzlich spezifiziert sind.

Schließlich kann jeder Knoten, der an der Anfrage beteiligt ist, die ihm zugeordnete Teilanfrage lokal optimieren (*lokale Optimierung*), wobei z.B. der Ausfüh-

rungsalgorithmus für bestimmte Operatoren festgelegt wird. Dies kann wie in zentralisierten DBS unter Einsatz lokaler Katalogdaten erfolgen. Die Aufgabenverteilung zwischen globaler und lokaler Optimierung hängt wesentlich von der Verteilung der Katalogdaten ab, z.B. ob Angaben des internen Schemas wie Indexstrukturen global oder nur lokal bekannt sind. Nach Abschluß der Optimierung erfolgt noch die *Code-Generierung* für den ermittelten Ausführungsplan, d.h. die Erstellung eines ausführbaren Programmes. Dabei wird üblicherweise für jede lokale Teilanfrage ein Ausführungsmodul erstellt und an dem jeweiligen Knoten gespeichert.

Zur Realisierung der angesprochenen Schritte kommen prinzipiell ein Übersetzungs- oder ein Interpretationsansatz in Betracht. Beim *Übersetzungsansatz* werden Erstellung eines Ausführungsplanes sowie Ausführung desselben in zeitlich getrennten Schritten vorgenommen. Dies empfiehlt sich vor allem für in Anwendungsprogramme eingebettete DB-Operationen, für die bereits zur Übersetzungszeit des Programms die Ausführungspläne erstellt werden können. Damit geht der meist hohe Aufwand zur Query-Optimierung nicht in die Ausführungszeiten ein, und der Aufwand fällt auch bei wiederholter Ausführung des Programmes bzw. von DB-Operationen (z.B. innerhalb von Programmschleifen) nur einmal an. Allerdings muß gegebenenfalls eine Neubestimmung von Ausführungsplänen veranlaßt werden, wenn zwischenzeitlich relevante Änderungen in den Schemaangaben erfolgten (z.B. Änderungen im konzeptionellen Schema, bezüglich Datenverteilung oder Indexstrukturen).

Beim *Interpretationsansatz* erfolgt dagegen die Festlegung der Auswertungsstrategie zum Ausführungszeitpunkt, wie es z.B. für Ad-hoc-Anfragen erforderlich ist. Der Hauptvorteil liegt darin, daß dynamische Kontrollentscheidungen möglich sind, basierend auf dem aktuellen Verarbeitungszustand (z.B. Größe von Zwischenergebnissen, Systemauslastung). Allerdings sind die Bearbeitungszeiten für komplexere Anfragen insgesamt wesentlich höher als beim Übersetzungsansatz, wenn erst während der Ausführung die Verarbeitungsstrategie für einzelne Teilschritte festgelegt wird. Daher wird selbst für Ad-hoc-Anfragen empfohlen, zunächst einen Ausführungsplan gemäß dem Übersetzungsansatz zu bestimmen und diesen direkt anschließend auszuführen [Hä87]. Eine Zwischenform bildet ein *hybrider Optimierungsansatz*. Dabei erfolgt eine statische Festlegung eines Ausführungsplanes nach dem Übersetzungsansatz, jedoch wird für bestimmte Kontrollentscheidungen eine dynamische Einflußnahme zur Laufzeit vorgesehen.

Im folgenden werden die drei Phasen der globalen Anfragebearbeitung genauer vorgestellt.

6.2 Anfragetransformation

Aufgabe der Anfragetransformation ist die Überführung einer z.B. in SQL deskriptiv formulierten DB-Operation in eine funktional äquivalente Interndarstellung, auf der die weiteren Schritte der Anfragebearbeitung aufbauen. Zur Interndarstellung verwenden wir hier die Relationenalgebra, da sie eine prozedurale Spezifikation der Verarbeitung zuläßt. Insbesondere können die zur Realisierung benötigten relationalen Basisoperatoren (Projektion, Selektion, Join etc.) sowie die Reihenfolge ihrer Bearbeitung spezifiziert werden. Die auf die Anfragetransformation folgende Phase der Daten-Lokalisierung führt im Prinzip ebenfalls eine Anfragetransformation durch, jedoch unter Berücksichtigung der Verteilungsinformation

Abb. 6-2: Teilschritte bei der Anfragetransformation .

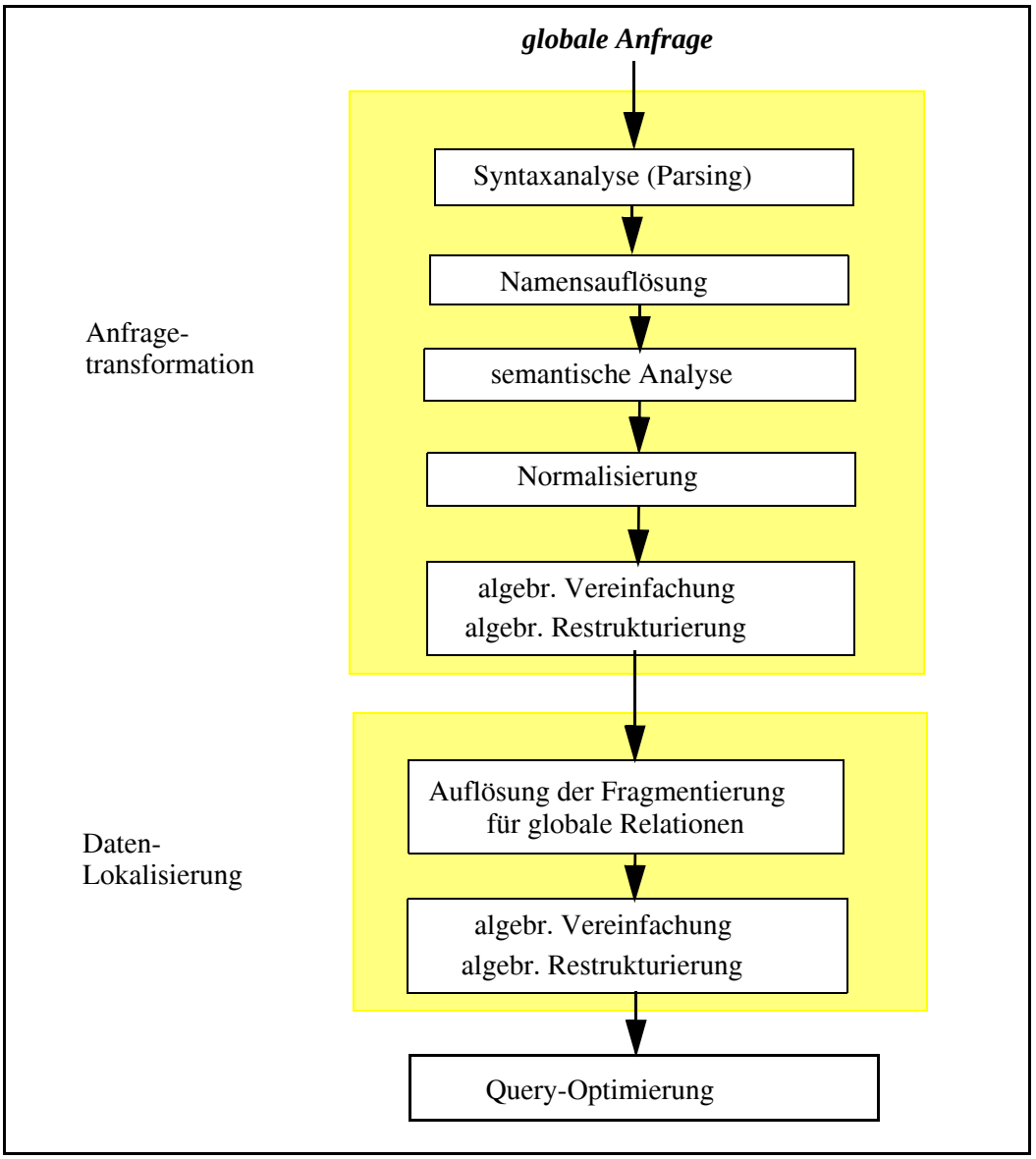


Abb. 6-2 veranschaulicht, daß die Anfragetransformation in mehrere aufeinanderfolgende Teilschritte unterteilt werden kann, wie sie bereits für zentralisierte DBS anfallen [JK84, Hä87]. Dabei werden zunächst die bei der Übersetzung üblichen lexikalischen und syntaktischen Analysen vorgenommen (*Parsing*), um die syntaktische Korrektheit der Anfrage sicherzustellen. Daran schließt sich die *Namensauflösung* an, d.h. die Abbildung logischer Objektbezeichnungen in interne Namen. Im verteilten Fall kann dies die Nutzung von Synonymtabellen erfordern, um die global eindeutigen Namen zu generieren (Abb. 4-5). Weiterhin sind - in Abhängigkeit der gewählten Katalogverwaltung - für den Zugriff auf globale Katalogdaten ggf. Kommunikationsvorgänge in Kauf zu nehmen.

Im Rahmen der *semantischen Analyse* wird geprüft, ob die verwendeten Relationen und Attribute im globalen Schema definiert sind. Falls die Operation auf Sichten (Views) spezifiziert wurde, erfolgt eine Abbildung auf die zugrundeliegenden Basisrelationen. Daneben können bereits einfache Integritätsbedingungen überprüft werden, z.B. die Typverträglichkeit in Vergleichsprädikaten oder die Einhaltung von Wertebereichen. Weiterhin lassen sich zu diesem Zeitpunkt einfache (wertunabhängige) Zugriffskontrollbedingungen überprüfen, soweit sie im globalen Schema festgelegt sind.

Aufgabe der *Normalisierung* ist es, die Anfrage in ein vereinheitlichtes (normalisiertes, kanonisches, standardisiertes) Format zu überführen, das die Durchführung der nachfolgenden Optimierungsschritte erleichtert. Dies betrifft vor allem die Repräsentation von Selektions- bzw. Join-Bedingungen (WHERE-Klausel in SQL). Hierzu kommen im wesentlichen zwei Alternativen in Betracht: die konjunktive bzw. die disjunktive Normalform. Die *konjunktive Normalform* ist eine Konjunktion ($\wedge$- bzw. AND-Verknüpfung) von Disjunktionen ($\vee$– bzw. OR-Prädikaten):

$$(p_{11} \vee p_{12} \vee \ldots \vee p_{1n}) \ \wedge \ \ldots \ \wedge \ (p_{m1} \vee p_{m2} \vee \ldots \vee p_{mn}) \,,$$

wobei p_{ij} einfache Prädikate der Form <Attribut> <Vergleichsoperator> <Attribut bzw. Konstante> darstellen. Die disjunktive Normalform dagegen stellt eine Disjunktion von Konjunktionen dar:

$$(p_{11} \wedge \ p_{12} \wedge \ldots \wedge p_{1n}) \ \vee \ \ldots \ \vee \ (p_{m1} \wedge p_{m2} \wedge \ldots \wedge p_{mn}).$$

Die beiden Normalformen geben also für Konjunktionen bzw. Disjunktionen unterschiedliche Auswertungsreihenfolgen vor (die jedoch im Zuge der weiteren Optimierung noch geändert werden können). Die Überführung von Bedingungen in eine dieser Normalformen ist einfach möglich durch Anwendung der bekannten Äquivalenzbeziehungen für logische Operationen ($\wedge$, $\vee$, $\neg$). Diese sind der Vollständigkeit halber in Abb. 6-3 zusammengestellt.

Abb. 6-3: Äquivalenzbeziehungen für logische Operationen

1. $p_1 \wedge p_2 \Leftrightarrow p_2 \wedge p_1$
2. $p_1 \vee p_2 \Leftrightarrow p_2 \vee p_1$
3. $p_1 \wedge (p_2 \wedge p_3) \Leftrightarrow (p_1 \wedge p_2) \wedge p_3$
4. $p_1 \vee (p_2 \vee p_3) \Leftrightarrow (p_1 \vee p_2) \vee p_3$
5. $p_1 \wedge (p_2 \vee p_3) \Leftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$
6. $p_1 \vee (p_2 \wedge p_3) \Leftrightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$
7. $\neg (p_1 \wedge p_2) \Leftrightarrow \neg p_1 \vee \neg p_2$
8. $\neg (p_1 \vee p_2) \Leftrightarrow \neg p_1 \wedge \neg p_2$
9. $\neg (\neg p_1) \Leftrightarrow p_1$

Beispiel 6-1

In unserer Bankanwendung mit den Relationen KONTO und KUNDE (Kap. 5.3) sollen die Namen der Kontoinhaber aus den Filialen Kaiserslautern und Leipzig ermittelt werden, die ihr Konto überzogen haben. In SQL könnte diese Anfrage folgendermaßen formuliert werden:

```
SELECT NAME
FROM KONTO, KUNDE
WHERE KONTO.KNR = KUNDE.KNR AND
        KTOSTAND < 0 AND
        FILIALE = "KL" OR FILIALE = "L" .
```

Die Selektionsbedingung sieht in konjunktiver Normalform folgendermaßen aus:

KONTO.KNR = KUNDE.KNR $\wedge$ KTOSTAND < 0 $\wedge$ (FILIALE = "KL" $\vee$ FILIALE = "L").

In disjunktiver Normalform dagegen lautet die Bedingung:

(KONTO.KNR = KUNDE.KNR $\wedge$ KTOSTAND < 0 $\wedge$ FILIALE = "KL") $\vee$
(KONTO.KNR = KUNDE.KNR $\wedge$ KTOSTAND < 0 $\wedge$ FILIALE = "L").

Das Beispiel zeigt, daß die Normalisierung zur Wiederholung von Selektions- und Verbundprädikaten führen kann. Für die disjunktive Normalform gilt dies bezüglich mit AND verknüpften Prädikaten (aufgrund von Regel 5 in Abb. 6-3), während die konjunktive Normalform bei OR-Verknüpfungen zur Wiederholung von Teilausdrücken führen kann (Regel 6 in Abb. 6-3). Da in der Praxis AND-Verknüpfungen meist häufiger als OR-Prädikate auftreten, empfiehlt sich die konjunktive Normalform.

Auf die Normalisierung folgen dann schließlich algebraische Optimierungsschritte, um damit bereits zu effizienter ausführbaren Anfrageformulierungen zu kommen. Als erster Transformationsschritt empfiehlt sich dazu die Durchführung *algebraischer Vereinfachungen*, um redundante Teilausdrücke zu eliminieren und damit Arbeit bei der Auswertung einzusparen. Eine solche Redundanz kann

z.B. durch eine ungeschickte Anfrageformulierung entstanden sein oder durch Anfragen auf Sichten, bei denen die Abbildung auf Basisrelationen zu redundanten Teilausdrücken geführt hat. Die Elimination redundanter Ausdrücke kann über die bekannten und in Abb. 6-4 zusammengestellten Idempotenzregeln erfolgen. Daneben ist es in einfachen Fällen möglich, durch Berücksichtigung von Integritätsbedingungen redundante Ausdrücke zu eliminieren.

Abb. 6-4: Idempotenzregeln für logische Operationen

1. $p \wedge p \Leftrightarrow p$	6. $p \vee TRUE \Leftrightarrow TRUE$
2. $p \vee p \Leftrightarrow p$	7. $p \wedge \neg p \Leftrightarrow FALSE$
3. $p \wedge TRUE \Leftrightarrow p$	8. $p \vee \neg p \Leftrightarrow TRUE$
4. $p \vee FALSE \Leftrightarrow p$	9. $p_1 \wedge (p_1 \vee p_2) \Leftrightarrow p_1$
5. $p \wedge FALSE \Leftrightarrow FALSE$	10. $p_1 \vee (p_1 \wedge p_2) \Leftrightarrow p_1$

Beispiel 6-2

In der Bankanwendung sei die Integritätsbedingung definiert, daß der Kontostand nicht negativ werden darf (was z.B. für Sparbücher sinnvoll ist). Das diese Bedingung definierende Prädikat

$$\neg (KTOSTAND < 0)$$

kann bei Anfragen, die das Attribut KTOSTAND betreffen, zur Verschärfung der Selektionsbedingung verwendet werden. Dies ist z.B. für die Anfrage aus Beispiel 6-1 möglich, für die sich daraufhin folgende Qualifikationsbedingung in konjunktiver Normalform ergibt:

$$KONTO.KNR = KUNDE.KNR \wedge$$
$$KTOSTAND < 0 \wedge$$
$$(FILIALE = "KL" \vee FILIALE = "L") \wedge$$
$$\neg (KTOSTAND < 0).$$

Durch die Anwendung der Regeln 7 und 5 aus Abb. 6-4 kann diese Bedingung sofort zu FALSE ausgewertet werden. Damit kann eine weitere Anfrageverarbeitung eingespart werden, da sich als Ergebnis der Anfrage die leere Menge ergibt.

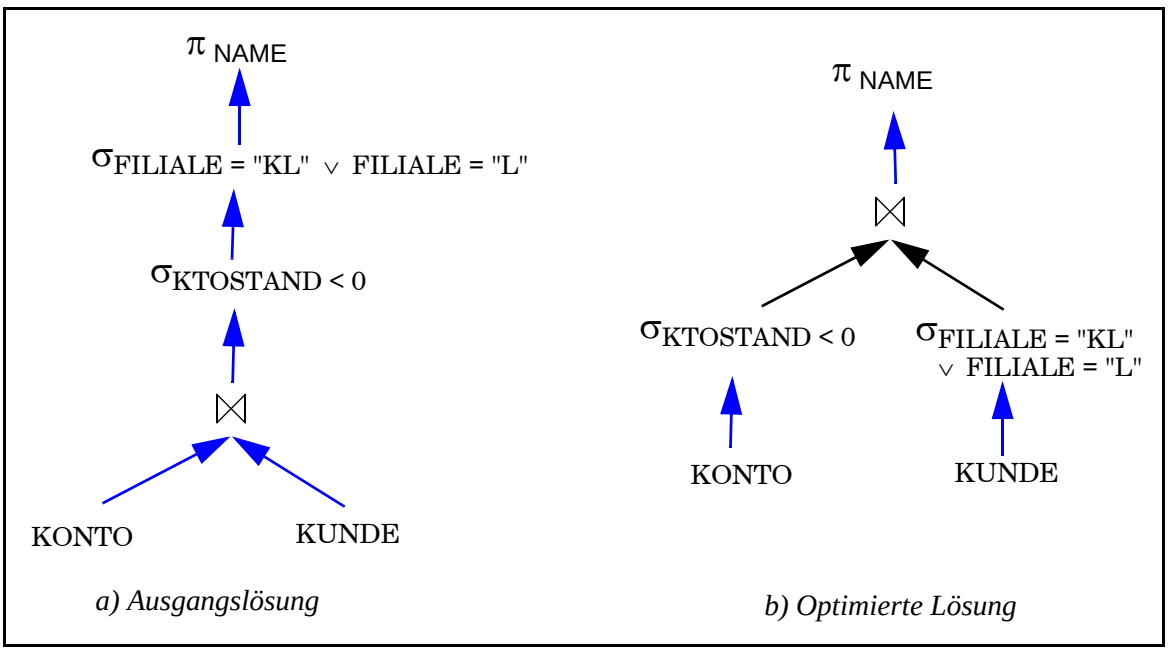
Ein zweiter Schritt bei der algebraischen Optimierung besteht darin, Restrukturierungen in der Anfragedarstellung vorzunehmen, welche zu einer Verbesserung der Auswertung führen. Diese Restrukturierungen basieren auf heuristischen Optimierungsregeln, die sich als allgemein sinnvoll herausgestellt haben. Zur Anwendung dieser Restrukturierungsregeln sehen wir vor, daß der bis dahin vorliegende Anfrageausdruck in Form eines *Operatorbaumes* dargestellt wird. In einem solchen Baum, der einen azyklischen Graphen darstellt, bilden die Blattknoten Basisobjekte (Relationen), auf denen die Operatoren angewendet werden. Die Operatoren korrespondieren zu den restlichen (Nicht-Blatt-) Knoten des Baumes. Die gerichteten Kanten spezifizieren ausgehend von den Blättern die Richtung des Datenflusses. Der Wurzelknoten schließlich bestimmt das Gesamtergebnis der

Anfrage. Im Rahmen der Fragmentierung hatten wir solche Operatorgraphen bereits kurz kennengelernt (Abb. 5-6).

In Abb. 6-5a ist ein möglicher Operatorbaum zur Anfrage aus Beispiel 6-1 gezeigt. Das Beispiel veranschaulicht auch, wie aus einer Anfrage ein Operatorbaum gewonnen werden kann. Zunächst bildet jede der beteiligten Relationen ein Blatt im Operatorbaum (FROM-Klausel in SQL). Der Wurzelknoten entspricht der Projektion auf die auszugebenden Attribute (SELECT-Klausel in SQL). Die Zwischenknoten schließlich korrespondieren zu den Selektions- und Join-Operationen, wie sie in dem normalisierten Qualifikationsausdruck spezifiziert sind (WHERE-Klausel von SQL).

Zu einer bestimmten Anfrage existiert in der Regel eine große Anzahl funktional äquivalenter Operatorbäume. Dies ergibt sich aufgrund von Äquivalenzbedingungen für relationale Operatoren, wie sie in einer Auswahl in Abb. 6-6 zusammengestellt sind. Die Optimierung einer Anfrage erfolgt immer unter Berücksichtigung dieser Regeln, um die funktionale Äquivalenz bei der Durchführung von Anfragetransformationen zu gewährleisten. Ein naiver Ansatz zur Query-Optimierung wäre die Bestimmung und Bewertung sämtlicher äquivalenter Opertorbäume, um darunter den günstigsten auszuwählen. Dieser Ansatz scheidet jedoch aufgrund des unverhältnismäßig hohen Aufwandes aus. Durch Anwendung von Heuristiken kann man jedoch bereits vielfach eine Vorauswahl treffen, so daß sich das Optimierungsproblem vereinfacht.

Abb. 6-5: Beispiel von Operatorbäumen



Solche Heuristiken zur *Restrukturierung* von Operatorbäumen können bereits im Rahmen der algebraischen Optimierung vorgenommen werden, bei der physi-

sche Aspekte wie Datenverteilung oder Indexstrukturen unberücksichtigt bleiben. Hierzu haben sich u.a. folgende Ansätze als effektiv erwiesen:

Abb. 6-6: Äquivalenzbeziehungen für relationale Operatoren (Auswahl)

1. $\sigma_{P1}(\sigma_{P2}(R)) \Leftrightarrow \sigma_{P2}(\sigma_{P1}(R))$
2. $\sigma_{P1}(\sigma_{P2}(R)) \Leftrightarrow \sigma_{P1 \wedge P2}(R)$
3. $\pi_A(\pi_B(R)) \Leftrightarrow \pi_B(\pi_A(R))$
4. $\pi_A(\pi_{A,B}(R)) \Leftrightarrow \pi_A(R)$
5. $\sigma_P(\pi_A(R)) \Leftrightarrow \pi_A(\sigma_P(R))$ *falls Prädikat P nur Attribute aus A umfaßt*
   $\sigma_P(\pi_A(R)) \Leftrightarrow \pi_A(\sigma_P(\pi_{A,B}(R)))$ *falls P auch Attribute aus B umfaßt*
6. $R1 \bowtie R2 \Leftrightarrow R2 \bowtie R1$
7. $R1 \bowtie (R2 \bowtie R3) \Leftrightarrow (R1 \bowtie R2) \bowtie R3$
8. $\sigma_{P(R1)}(R1 \bowtie R2) \Leftrightarrow \sigma_{P(R1)}(R1) \bowtie R2$ *(P(R1) sei Prädikat auf R1)*
9. $\pi_{A,B}(R1 \bowtie R2) \Leftrightarrow \pi_A(R1) \bowtie \pi_B(R2)$ *(A/B seien Attributmengen aus R1/R2 inklusive Join-Attribut)*
10. $\sigma_P(R1 \cup R2) \Leftrightarrow \sigma_P(R1) \cup \sigma_P(R2)$
11. $\pi_A(R1 \cup R2) \Leftrightarrow \pi_A(R1) \cup \pi_A(R2)$
12. $R1 \bowtie (R2 \cup R3) \Leftrightarrow (R1 \bowtie R2) \cup (R1 \bowtie R3)$

- Möglichst frühzeitige Ausführung von Selektionsoperationen, um eine Reduzierung des weiterzuverarbeitenden Datenvolumens zu erreichen (Anwendung der Regeln 5, 8 und 10 in Abb. 6-6).
- Frühzeitige Durchführung von Projektionen zur Reduzierung des Datenvolumens, sofern die (teuere) Eliminierung von Duplikaten nicht erforderlich ist (Regeln 9 und 11 in Abb. 6-6). Dabei sollten Projektionen jedoch nicht vor Selektionen vorgenommen werden.
- Zusammenfassung mehrerer Selektionen und Projektionen auf demselben Objekt (Regeln 2 und 4 in Abb. 6-6).
- Bestimmung gemeinsamer Teilausdrücke, wie sie etwa durch die Normalisierung eingeführt wurden, um diese nur einmal zu berechnen.

Die frühzeitige Reduzierung der Datenmengen durch Vorziehen einstelliger Operatoren wie Selektion und Projektion ist im verteilten Fall i.a. noch effektiver als schon in zentralisierten DBS, weil damit auch der Kommunikationsumfang vermindert werden kann. Der in Abb. 6-5a gezeigte Operatorbaum läßt sich durch Anwendung der genannten Heuristiken und unter Beachtung der Äquivalenzbedingungen leicht verbessern. Durch Vorziehen der Selektionsbedingungen ergibt sich so die in Abb. 6-5b dargestellte Lösung. Eine weitere Optimierung wäre durch teilweises Vorziehen von Projektionen möglich (Regel 11).

6.3 Erzeugung von Fragment-Anfragen

In diesem Schritt wird eine weitergehende Transformation des relationalen Anfrageausdrucks (Operatorbaumes) unter Berücksichtigung der Datenverteilung vorgenommen. Wie in Abb. 6-2 gezeigt, sind hierzu zwei Teilschritte vorgesehen. Zunächst werden Zugriffe auf fragmentierte Relationen durch Zugriffe auf die einzelnen Fragmente ersetzt. Dies läßt sich einfach erreichen, indem die betroffenen Relationen durch ihren Rekonstruktionsausdruck ersetzt werden, wodurch ein sogenannter (initialer) Fragment-Ausdruck entsteht. Wie wir in Kap. 5.5 gesehen haben, kann jeder Rekonstruktionsausdruck selbst durch einen Operatorbaum beschrieben werden. Diese Operatorbäume werden jetzt quasi als Teilbäume in den Operatorbaum für die Anfrage eingefügt. Daran anschließend können auf dem entstandenen Baum (Ausdruck) wiederum algebraische Vereinfachungen und Restrukturierungen vorgenommen werden. Ziel dieser Optimierungen ist es, die Verarbeitung auf möglichst wenige der einzelnen Fragmente zu reduzieren. Damit kann das zu verarbeitende Datenvolumen und somit der Rekonstruktions- und Kommunikationsaufwand begrenzt werden.
Wir werden diese Vorgehensweise nachfolgend für jede der in Kap. 5 eingeführten Fragmentierungsarten diskutieren. Wir gehen dabei vereinfachend davon aus, daß keine Replikation von Fragmenten vorliegt.

6.3.1 Daten-Lokalisierung bei primärer horizontaler Fragmentierung

Die primäre horizontale Fragmentierung (Kap. 5.3) zerlegt eine Relation aufgrund von Selektionsprädikaten auf Attributen der Relation. Die Rekonstruktion der Relation ergibt sich durch Vereinigung der einzelnen Fragmente. Selektionsanfragen auf derart fragmentierten Relationen lassen sich ggf. vereinfachen, falls sie sich auf *Fragmentierungsattribute* beziehen, welche zur Definition der Fragmentierung verwendet wurden. Damit kann die Anfrage ggf. auf eine Teilmenge der Fragmente begrenzt werden, so daß sich der Verarbeitungsaufwand sowie der Kommunikationsbedarf reduzieren.

Beispiel 6-3

Die Relation KONTO sei folgendermaßen horizontal auf dem Attribut KNR (Kundennummer) fragmentiert:

$$KONTO_1 = \sigma_{KNR \leq "K3"}(KONTO)$$
$$KONTO_2 = \sigma_{"K3" < KNR \leq "K6"}(KONTO)$$
$$KONTO_3 = \sigma_{KNR > "K6"}(KONTO)$$

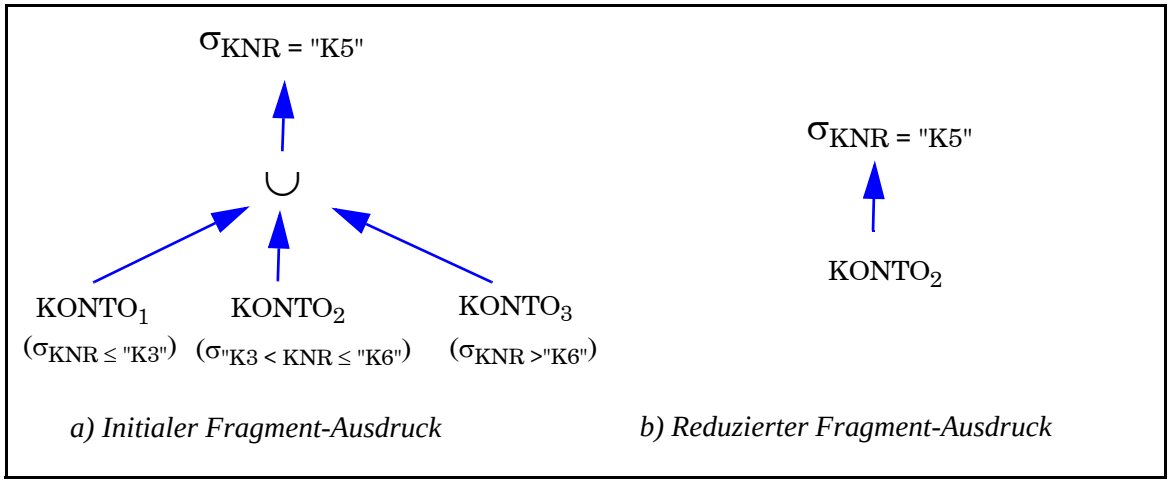
Folgende Anfrage sei zu bearbeiten:

```
SELECT * FROM KONTO WHERE  KNR="K5"
```

Im ersten Schritt wird im Operatorbaum für diese Anfrage die Relation KONTO durch den sie rekonstruierenden Operatorbaum ersetzt; das Ergebnis zeigt Abb. 6-7a. Da sich die Selektion auf das Fragmentierungsattribut KNR bezieht, kann eine algebraische Vereinfachung vorgenommen werden. Nach Vertauschung von Vereinigung und Selektion (Vorziehen der Selektionsoperation auf die Fragmente) erkennt man leicht, daß sich auf

den Fragmenten $KONTO_1$ und $KONTO_3$ die leere Ergebnismenge ergibt. Die Verarbeitung ist daher auf Fragment $KUNDE_2$ beschränkt, so daß sich der Vereinigungsoperator ebenfalls erübrigt. Damit ergibt sich der in Abb. 6-7b gezeigte Operatorbaum, der lokal an einem Rechner ausführbar ist.

Abb. 6-7: Daten-Lokalisierung bei horizontaler Fragmentierung (Selektion)



Solche Vereinfachungen lassen sich ggf. auch für *Join-Anfragen* nutzen, falls die beteiligten Relationen horizontal fragmentiert sind. Ein erster Join-Ansatz besteht darin, zunächst für die beteiligten Relationen die Vereinigung der einzelnen Fragmente zu bilden und danach die Join-Berechnung vorzunehmen. Dabei können ggf. wieder einige Fragmente von der Bearbeitung ausgeschlossen werden, falls zusätzliche Selektionsbedingungen auf dem Fragmentierungsattribut vorliegen. Alternativ dazu kann jedoch auch zunächst eine Join-Berechnung zwischen den einzelnen Fragmenten und anschließend eine Vereinigung der Teilergebnisse vorgenommen werden (aufgrund von Regel 12 in Abb. 6-6). Ein Vorteil dabei liegt darin, daß die einzelnen Teil-Joins parallel zueinander ausgeführt werden können. Im schlimmsten Fall ist jedoch für jedes Fragment der ersten Relation eine Join-Berechnung mit jedem Fragment der zweiten Relation erforderlich. Allerdings kann der Aufwand oft erheblich reduziert werden, falls beide Relationen auf dem Join-Attribut fragmentiert sind. Denn dann ist für ein Fragment der einen Relation eine Join-Berechnung i.a. nur mit einer Teilmenge der Fragmente der anderen Relation erforderlich.

Beispiel 6-4

Die Relationen KUNDE und KONTO seien beide horizontal auf dem Attribut KNR (Kundennummer) fragmentiert. Für KONTO unterstellen wir die Fragmentierung aus dem vorherigen Beispiel 6-3; KUNDE sei folgendermaßen zerlegt:

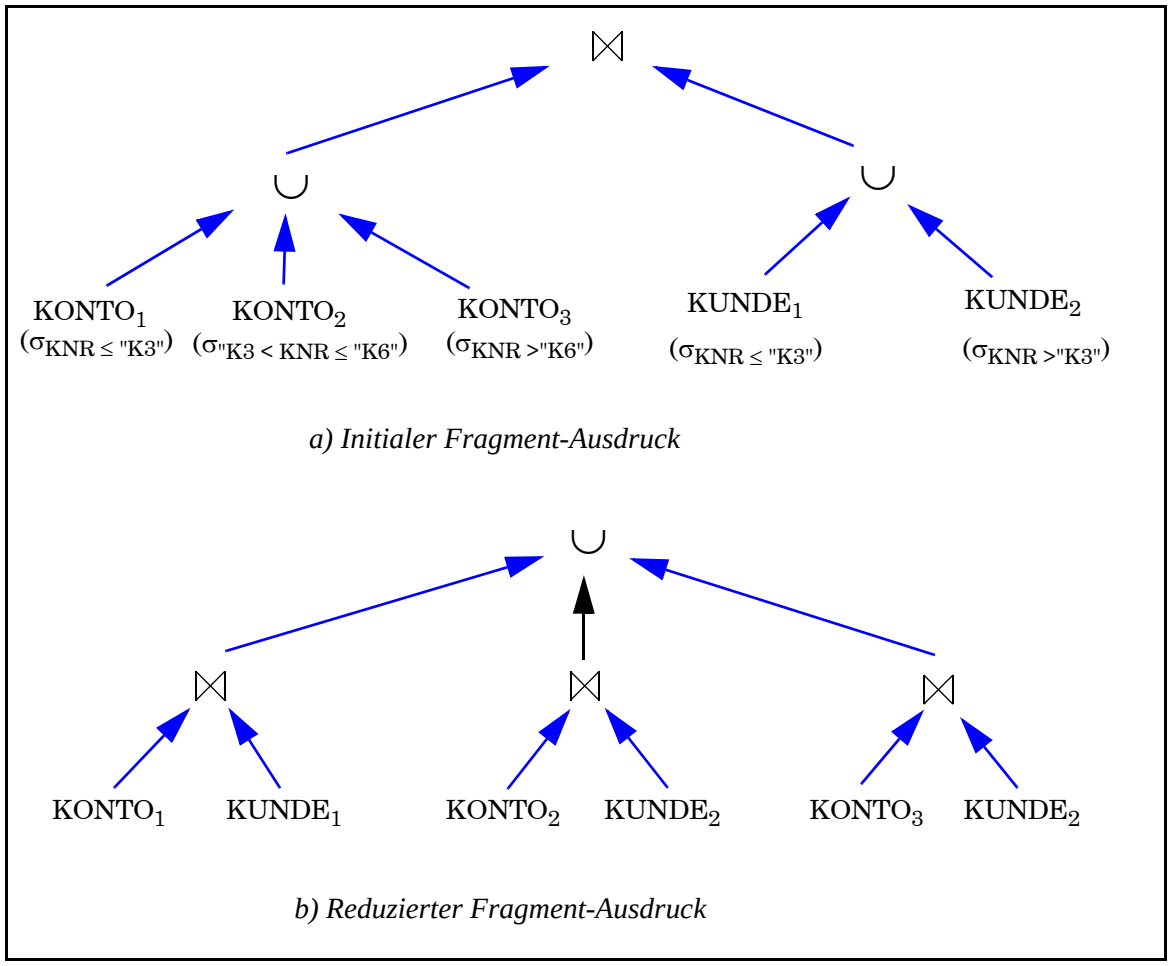
$KUNDE_1 = \sigma_{KNR \leq "K3"}(KUNDE)$
$KUNDE_2 = \sigma_{KNR > "K3"}(KUNDE)$.

Für die Join-Anfrage

```
SELECT * FROM KUNDE, KONTO
WHERE KUNDE.KNR=KONTO.KNR
```

ergibt sich damit der in Abb. 6-8a gezeigte Fragment-Ausdruck. Nach Vertauschen der Ausführungsreihenfolge für Vereinigung und Join-Berechnung wäre für jedes der drei KONTO-Fragmente die Join-Bildung mit den beiden KUNDE-Fragmenten erforderlich. Aufgrund der Fragmentierung über das Join-Attribut sowie der Tatsache, daß die Fragmentierungsprädikate für $KONTO_1$ und $KUNDE_1$ übereinstimmen und das Fragmentierungsprädikat für $KUNDE_2$ der Vereinigung der Fragmentierungsprädikate von $KONTO_2$ und $KONTO_3$ entspricht, können jedoch einige Join-Berechnungen eliminiert werden, da sie die leere Ergebnismenge liefern. Dies trifft für die Teil-Joins $KONTO_1 \bowtie KUNDE_2$, $KONTO_2 \bowtie KUNDE_1$ sowie $KONTO_3 \bowtie KUNDE_1$ zu. Der somit reduzierte Operatorbaum ist in Abb. 6-8b gezeigt. Die dabei anfallenden Join-Operationen können parallel berechnet werden. Zudem konnte die bei der Join-Berechnung zu berücksichtigende Datenmenge reduziert werden, da jeder Kontosatz nur mit einer Teilmenge der Kundensätze abgeglichen werden muß und nicht mehr mit der gesamten Kundenrelation wie in der Ausgangslösung.

Abb. 6-8: Daten-Lokalisierung bei horizontaler Fragmentierung (Join-Berechnung)



6.3.2 Daten-Lokalisierung bei abgeleiteter horizontaler Fragmentierung

Eine ähnliche, jedoch noch weitergehende Optimierung von Join-Anfragen wird bei der abgeleiteten horizontalen Fragmentierung möglich, wie bereits in Kap. 5.3.2 erwähnt. In diesem Fall wird die Fragmentierung einer Relation auf die Fragmentierung einer "übergeordneten" Relation abgestimmt, zu der eine funktio-

nale (hierarchische) Beziehung über die Werte eines gemeinsamen Attributs besteht (i.a. Fremd-/Primärschlüssel-Beziehung). Es ergibt sich daher in der Regel für beide Relationen die gleiche Anzahl von Fragmenten. Werden die zusammengehörigen Fragmentpaare der beiden Relationen jeweils demselben Rechner zugeordnet, so können die einzelnen Teil-Joins jeweils lokal (und parallel) berechnet werden. Kommunikation fällt nur zum Starten der Teil-Joins sowie zum Mischen der Teilergebnisse an.

Beispiel 6-5

Wir unterstellen die in Beispiel 5-2 verwendeten Fragmentierungen nach Filialzugehörigkeit. Dabei wird KUNDE horizontal in drei Fragmente unterteilt; für KONTO wird darauf aufbauend eine abhängige Fragmentierung definiert:

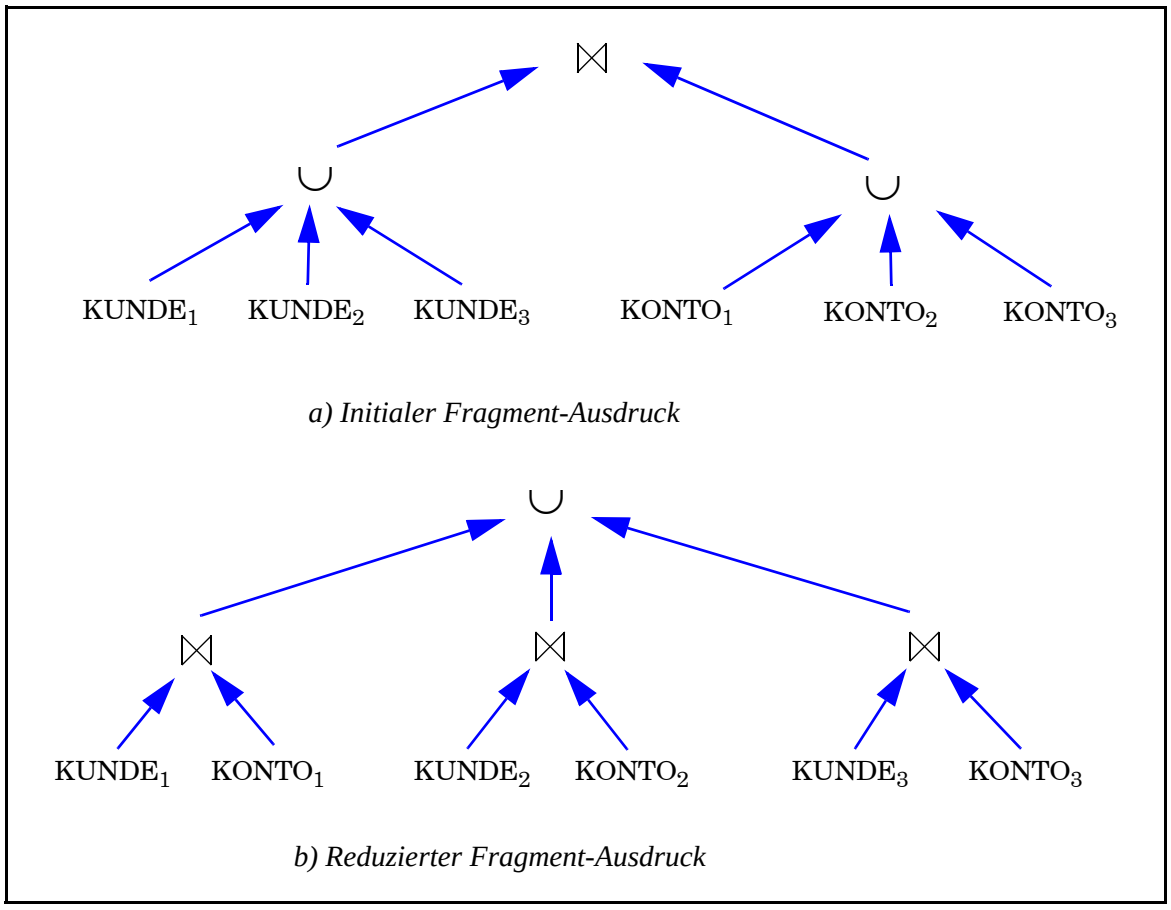
$KUNDE_1 = \sigma_{FILIALE="L"}(KUNDE)$
$KUNDE_2 = \sigma_{FILIALE="F"}(KUNDE)$
$KUNDE_3 = \sigma_{FILIALE="KL"}(KUNDE)$.

$KONTO_1 = KONTO \ltimes KUNDE_1$
$KONTO_2 = KONTO \ltimes KUNDE_2$
$KONTO_3 = KONTO \ltimes KUNDE_3$.

Folgende Join-Anfrage sei wiederum zu beantworten:

```
SELECT * FROM KUNDE, KONTO
WHERE        KUNDE.KNR=KONTO.KNR.
```

Abb. 6-9: Join-Berechnung bei abgeleiteter horizontaler Fragmentierung



Der Operatorbaum des initialen Fragment-Ausdrucks für diese Anfrage ist in Abb. 6-9a dargestellt. Nach Vorziehen der Join-Berechnungen können von den insgesamt 9 Teil-Joins 6 eliminiert werden, da sie aufgrund der abhängigen horizontalen Verteilung leere Ergebnismengen liefern. Lediglich die in Abb. 6-9b gezeigten Teil-Joins, die jeweils zusammengehörige Fragmentpaare betreffen, sind noch auszuführen. Die Teil-Joins sind jeweils an einem Rechner lokal ausführbar und können parallel berechnet werden.

Das Beispiel zeigt, daß für die abgeleitete horizontale Fragmentierung eine ähnliche Zerlegung der Join-Bearbeitung wie für übereinstimmende primäre horizontale Fragmentierungen (Beispiel 6-4) möglich ist. Allerdings ist jetzt gewährleistet, daß jedes Fragment der einen Relation nur mit genau einem Fragment der zweiten Relation zu verknüpfen ist, während dies ansonsten nur in Spezialfällen erreichbar ist. So war in Beispiel 6-4 für Fragment $KUNDE_2$ eine Join-Berechnung mit zwei KONTO-Fragmenten erforderlich.

Selektionsbedingungen auf Fragmentierungsattributen können auch bei abgeleiteter horizontaler Fragmentierung zur weitergehenden Reduzierung der Verarbeitung genutzt werden. Wird in Beispiel 6-5 etwa die WHERE-Klausel um die Bedingung FILIALE="F" erweitert, kann die Anfrage auf die lokale Join-Berechnung zwischen $KUNDE_2$ und $KONTO_2$ reduziert werden.

6.3.3 Daten-Lokalisierung bei vertikaler Fragmentierung

Bei einer vertikalen Fragmentierung sind die Fragmente durch Projektionsoperationen definiert; die Rekonstruktion der Relation erfordert die Join-Bildung auf den Fragmenten (Kap. 5.4). Fragment-Ausdrücke für vertikal fragmentierte Relationen lassen sich einfach reduzieren, da eine Bearbeitung nur der Fragmente erforderlich ist, deren Attribute auszugeben sind bzw. in Zwischenschritten benötigt werden. Die "nutzlosen" Fragmente lassen sich durch Vorziehen von Projektionen im Operatorbaum leicht bestimmen. Durch ihre Wegnahme können die entsprechenden Verbundoperationen eingespart werden.

Beispiel 6-6

Wir unterstellen folgende vertikale Fragmentierung der KUNDE-Relation:

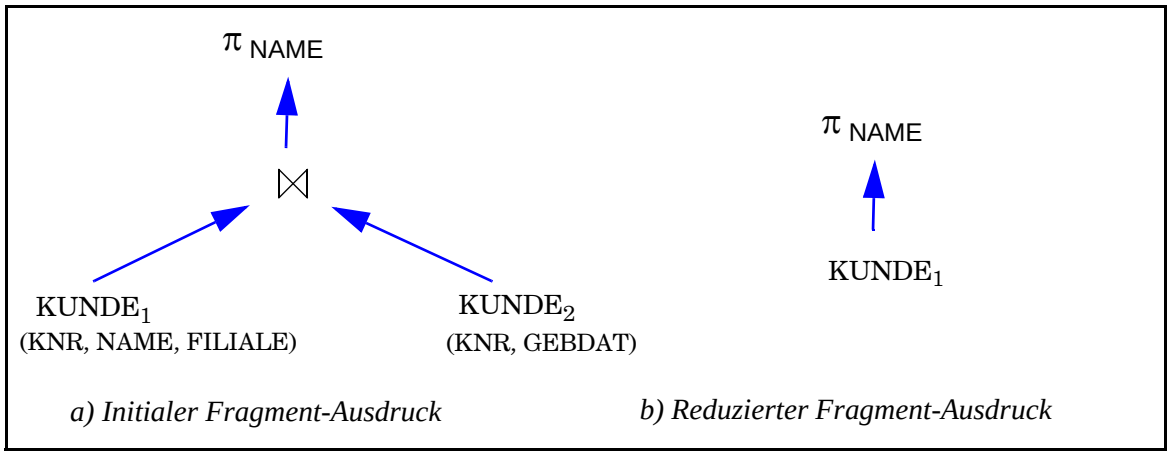
$$KUNDE_1 = \pi_{KNR,\ NAME,\ FILIALE}(KUNDE)$$
$$KUNDE_2 = \pi_{KNR,\ GEBDAT}(KUNDE).$$

Zu bestimmen sei folgende Anfrage

```
SELECT NAME FROM KUNDE.
```

Der initiale Fragment-Ausdruck für diese Anfrage ist in Abb. 6-10a gezeigt. Nach Vertauschen der Ausführungsreihenfolge von Projektion und Join erkennt man, daß die Projektion auf $KUNDE_2$ die leere Menge ergibt, da das Attribut NAME in diesem Fragment nicht enthalten ist. Die Projektion ist daher nur (lokal) auf $KUNDE_1$ auszuführen, wie in Abb. 6-10b verdeutlicht

Abb. 6-10: .Daten-Lokalisierung bei vertikaler Fragmentierung



6.3.4 Daten-Lokalisierung bei hybrider Fragmentierung

Die vorgestellten Ansätze zur Bildung und Reduzierung von Fragment-Ausdrükken können natürlich miteinander kombiniert werden, wie es im Falle von hybrider Fragmentierung (Kap. 5.5) erforderlich wird. Horizontale Fragmente können also durch Vorziehen von Selektionsbedingungen auf einem der Fragmentierungsattribute ggf. von der Verarbeitung ausgeschlossen werden; vertikale Fragmente durch Analyse der Projektionsoperationen.

Beispiel 6-7

Gegeben sei folgende hybride Fragmentierung der Relation KUNDE:

$KUNDE_1 = \sigma_{KNR \leq "K3"} (\pi_{KNR, NAME, FILIALE} (KUNDE))$
$KUNDE_2 = \sigma_{KNR > "K3"} (\pi_{KNR, NAME, FILIALE} (KUNDE))$
$KUNDE_3 = \pi_{KNR, GEBDAT} (KUNDE)$.

Für die Rekonstruktion gilt also

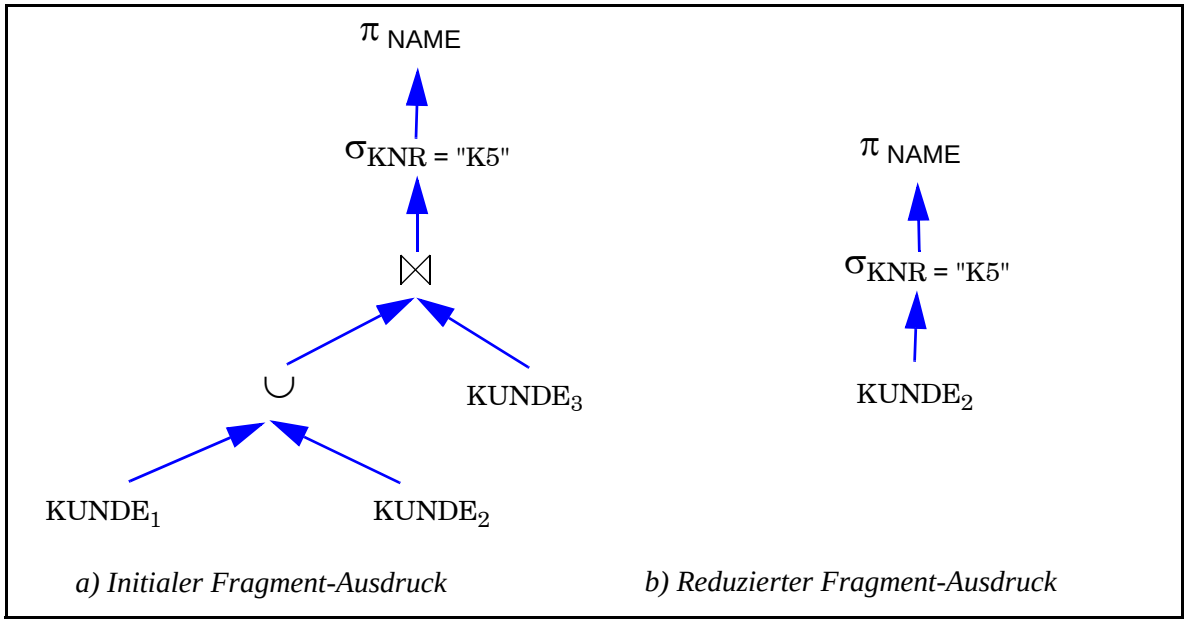
$KUNDE = (KUNDE_1 \cup KUNDE_2) \bowtie KUNDE_3$.

Zu berechnen sei folgende Anfrage:

```
SELECT NAME FROM KUNDE
WHERE KNR="K5".
```

Auf dem initialen Fragment-Ausdruck von Abb. 6-11a wird zunächst die Selektionsbedingung auf die Fragmente vorgezogen, wobei Fragment $KUNDE_1$ eliminiert wird. Danach erfolgt das Vorziehen der Projektion, wodurch die Verarbeitung auf Fragment $KUNDE_3$ entfällt. Den reduzierten Fragment-Ausdruck zeigt Abb. 6-11b

Abb. 6-11: Daten-Lokalisierung bei hybrider Fragmentierung.



6.4 Globale Query-Optimierung

Aufgabe der Query-Optimierung ist es, unter der großen Anzahl äquivalenter Ausführungspläne einen möglichst kostengünstigen auszuwählen. Diese Aufgabe wurde bereits durch die algebraischen Optimierungen im Rahmen der Anfragetransformation und Daten-Lokalisierung unterstützt. Dabei konnten u.a. "nutzlose" Teilausdrücke eliminiert werden und einfache Restrukturierungen wie Zusammenfassung und Vorziehen einstelliger Operationen vorgenommen werden. Von einfachen Fällen abgesehen reichen diese Schritte jedoch nicht aus, um einen günstigen Plan zu bestimmen. Hierzu ist eine möglichst präzise quantitative Abschätzung der Bearbeitungskosten vorzunehmen, wobei entsprechende Statistiken benötigt werden. Weiterhin sind physische Aspekte der Hardware-Umgebung (CPU-Geschwindigkeit, Kosten der Nachrichtenübertragung etc.) zu berücksichtigen, ebenso der Einfluß der vorliegenden Speicherungsstrukturen und Zugriffspfade sowie unterschiedlicher Implementierungsformen für einzelne Operatoren.

Ein Problem bei der Query-Optimierung im verteilten Fall ergibt sich durch die Trennung zwischen globaler und lokaler Optimierung, wie sie für eine möglichst hohe Knotenautonomie erforderlich ist. Denn damit kann der globale Optimierer die Kosten eines Ausführungsplanes nur partiell abschätzen, im wesentlichen nämlich die Kommunikationskosten. Dies kann dazu führen, daß ein bezüglich der globalen Kosten optimaler Plan hohe lokale Kosten verursacht, während alternative Strategien trotz höherer Kommunikationskosten insgesamt billiger abschneiden. Da der globale Optimierer i.a. keine Kenntnisse über lokale Schemaangaben der einzelnen Rechner hat, kann über die Nutzung von Indexstrukturen oder die Auswahl einer lokalen Join-Strategie (z.B. Nested-Loop-, Sort-Merge- oder Hash-

Join) nur der lokale Optimierer entscheiden. Der globale Optimierer bestimmt dagegen vor allem die Ausführungsreihenfolge sowie die Ausführungsorte der einzelnen Operatoren sowie die Methode des Datenaustauschs. Bei einfachen Operatoren wie Selektion und Projektion ist dabei der Ausführungsort durch die Datenallokation gegeben. Hier bestehen nur im Falle von Replikation Auswahlmöglichkeiten. Für komplexere Berechnungen wie Join-Bildung besteht dagegen ein größeres Optimierungspotential, wie in Kap. 6.5 gezeigt wird.

Zur Bewertung alternativer Ausführungspläne ist ein geeignetes *Kostenmodell* erforderlich. Im Prototyp R* wurde hierzu die gewichtete Summe aus CPU-, E/A- und Kommunikationskosten verwendet [ML86]:

$$\text{Gesamtkosten} = W_{CPU} * \#\text{Instruktionen} + W_{I/O} * \#\text{E/A} + W_{Msg} * \#\text{Nachrichten} + W_{Byt} * \#\text{Bytes}.$$

Dabei sind die Kommunikationskosten über zwei Komponenten berücksichtigt, die Anzahl der Nachrichten sowie den Nachrichtenumfang. Die einzelnen Gewichte W_x, die z.B. Zeiteinheiten darstellen können, sind auf die jeweilige Umgebung abzustimmen (Leistungsfähigkeit der CPUs, des Kommunikationsnetzwerkes etc). Bei Trennung von globaler und lokaler Optimierung können, wie erwähnt, bei der globalen Optimierung lediglich die Kommunikationskosten abgeschätzt werden. In geographisch verteilten Systemen ist dies i.a. keine signifikante Einschränkung, da hier die teuere Kommunikation oft einen Großteil der Verarbeitungsdauer verursacht. In lokal verteilten Systemen gilt dies sicherlich nicht mehr, so daß CPU- und E/A-Kosten auch für globale Optimierungsentscheidungen zu berücksichtigen sind. In diesem Fall spielt die Knotenautonomie jedoch auch eine untergeordnete Rolle, so daß ggf. auf die Trennung zwischen globaler und lokaler Optimierung verzichtet werden kann*.

Wenn die Gewichte W_x in Zeiteinheiten angegeben werden, dann stellt obige Kostenformel eine grobe Abschätzung hinsichtlich der sequentiellen Bearbeitungszeit dar (ohne Wartesituationen aufgrund konkurrierender Operationen). Allerdings kann bei Parallelisierung einzelner Teilschritte diese Bearbeitungszeit möglicherweise signifikant reduziert werden. Um dies bei der Kostenbewertung zu berücksichtigen, müßten für parallelisierte Teilschritte die Kosten gegenüber einer sequentiellen Bearbeitung entsprechend geringer angesetzt werden.

Die Kosten relationaler Operatoren sind wesentlich vom Umfang der zu verarbeitenden Datenmenge bestimmt. Um eine Kostenbewertung vornehmen zu können, müssen daher entsprechende *Statistiken* vorliegen, insbesondere hinsichtlich der Kardinalität (Tupelanzahl) der einzelnen Relationen und Fragmente, der Größe

* In R* kann der globale Optimierer "Vorschläge" hinsichtlich der lokalen Ausführungsstrategie abgeben, wobei die lokalen Kosten der entsprechenden Vorschläge dann bei der Auswahl eines Plans berücksichtigt werden. Die lokalen Optimierer können jedoch von den Vorschlägen abweichende Ausführungsstrategien bestimmen [ML86].

einzelner Attribute und Tupel sowie bezüglich der Häufigkeitsverteilung von Attributwerten. Aus diesen Angaben werden bei der Optimierung dann für die einzelnen Ausführungspläne die Größe von Zwischenergebnissen ermittelt, um wiederum die Kosten für nachfolgend auszuführende Operationen abzuschätzen.

Der Bestimmung von Zwischenergebnis-Größen kommt somit eine wesentliche Bedeutung zu. Bei Selektionen ist dabei ein sogenannter *Selektivitätsfaktor SF* ($0 \leq SF \leq 1$) zu ermitteln, der angibt, welcher Anteil der Relation sich für eine bestimmte Bedingung qualifiziert*. Sei *Card (R)* die Kardinalität von Relation R und *SF (p)* der Selektivitätsfaktor für Prädikat p (auf R) dann gilt:

$$Card(\sigma_p(R)) = SF(p) * Card(R).$$

Da es nicht möglich ist, für jedes beliebige Prädikat Selektivitätsstatistiken zu führen, ist man auf Abschätzungen angewiesen. Hierzu benötigt man für jedes Attribut A wenigstens die Anzahl unterschiedlicher Attributwerte (= $Card(\pi_A(R))$ sowie den minimalen und maximalen Attributwert. Unter der Annahme, daß die einzelnen Attributwerte gleichverteilt vorkommen, kann man dann z.B. die Selektivität von exakten Anfragen (exact match queries) bzw. Bereichsanfragen bezüglich einer Eingabekonstanten k folgendermaßen abschätzen (k liege im Intervall zwischen *Min* (A) und *Max* (A)):

$$SF(A = k) = 1 / Card(\pi_A(R))$$

$$SF(A > k) = (Max(A) - k) / (Max(A) - Min(A))$$

$$SF(A < k) = (k - Min(A)) / (Max(A) - Min(A)).$$

Unter der Annahme, daß die Werte verschiedener Attribute unabhängig voneinander sind, kann man weiterhin die Selektivität von Konjunktionen und Disjunktionen einfacherer Prädikate abschätzen, z.B.:

$$SF(p(A) \wedge p(B)) = SF(p(A)) * SF(p(B)).$$

Ähnliche Abschätzungen können auch für die anderen relationalen Operatoren vorgenommen werden. Für den Join zwischen zwei Relationen ist ein sogenannter *Join-Selektivitätsfaktor JSF* zu ermitteln, der angibt, welcher Anteil aus dem kartesischen Produkt der Relationen im Join-Ergebnis enthalten ist:

$$Card(R \bowtie S) = JSF * Card(R) * Card(S).$$

Einen wichtigen Spezialfall hierbei bilden N:1-Joins (hierarchische Joins) aufgrund einer Fremdschlüssel/Primärschlüsselbeziehung, bei denen zu jedem Wert des Join-Attributs der größeren Relation genau ein Verbundpartner in der kleineren Relation existiert. In diesem Fall gilt:

$$Card(R \bowtie S) = \text{Max}(Card(R), Card(S)).$$

* Wir sprechen von einer "hohen Selektivität", falls der Selektivitätsfaktor einen kleinen Wert annimmt (z.B. SF < 0.01), also nur ein kleiner Teil der Relation sich qualifiziert.

Problematisch bei der skizzierten Vorgehensweise ist, daß die erwähnten Annahmen (Gleichverteilung von Attributwerten, Unabhängigkeit verschiedener Attribute) oft nicht gegeben sind. Folglich können die darauf basierenden Schätzungen starken Fehlern unterworfen sein, so daß es ggf. zur Auswahl ungünstiger Pläne kommt. Diese Problematik stellt sich bereits in zentralisierten DBS; eine Verbesserung der Situation kann z.B. unter Verwendung genauerer Statistiken über die Werteverteilungen erfolgen [LNS90]. Weiterhin könnten Statistiken zur Join-Selektivität mit vertretbarem Aufwand geführt werden, da die Anzahl unterschiedlicher (nicht-hierarchischer) Join-Verknüpfungen oft begrenzt ist.

6.5 Bearbeitung von Join-Anfragen

Wir verdeutlichen die Ansätze zur globalen Optimierung beispielhaft für die (natürliche) Join-Berechnung, da diese Aufgabe häufig anfällt und oft hohe Ausführungskosten verursacht, so daß sich ein entsprechend großes Optimierungspotential ergibt. Dies gilt bereits für einfache Joins zwischen zwei Relationen. Für Join-Berechnungen zwischen mehr als zwei Relationen ist zusätzlich eine günstige Ausführungsreihenfolge zu bestimmen. In unseren Diskussionen nehmen wir vereinfachend an, daß die beteiligten Relationen jeweils vollständig an einem Knoten gespeichert sind. Die Algorithmen lassen sich jedoch auch für die Join-Berechnung zwischen einzelnen Fragmenten verwenden. Alternativ dazu können die einzelnen Relationen vor der Join-Bildung aus den Fragmenten rekonstruiert werden (Kap. 6.3.1).

Wir stellen zunächst einfache Strategien zur Join-Berechnung vor, bei denen ganze Relationen zwischen den Rechnern verschickt werden bzw. bei denen die zur Join-Berechnung benötigten Sätze der anderen Relation einzeln angefordert werden. Danach wird untersucht, wie durch Einsatz von Semi-Joins der Kommunikationsumfang reduziert werden kann. Anschließend (Kap. 6.5.3) wird noch auf eine Variante dieser Methode eingegangen, bei der Bitvektoren (Hash-Filter) zur Bestimmung von Verbundpartnern verwendet werden. Abschließend diskutieren wir die Berechnung von Mehr-Wege-Joins.

6.5.1 Einfache Strategien

Zu berechnen sei der natürliche Verbund zwischen Relation R an Knoten K_R und Relation S an Knoten K_S. Die einfachen Join-Strategien sind dadurch gekennzeichnet, daß der Join vollständig an einem Knoten ausgeführt wird. Damit sind im wesentlichen noch zwei Punkte zu klären: a) an welchem Knoten erfolgt die Join-Berechnung und b) wie wird der Datenaustausch zwischen den Knoten durchgeführt. Für den Ausführungsort des Joins kommen drei Möglichkeiten in Betracht, nämlich Knoten K_R, Knoten K_S oder ein sonstiger Knoten, an dem z.B. eine

Weiterverarbeitung des Ergebnisses vorzunehmen ist. Bezüglich des Datenaustauschs bestehen zwei Alternativen, die z.B. im Prototyp R* unterstützt werden [ML86]:

1. *Transfer ganzer Relationen ("Ship Whole")*
   Hierbei werden Relationen vollständig an den Join-Knoten übertragen, wo sie in temporären Relationen gespeichert werden, bevor die Join-Ausführung erfolgt. Soll die Join-Berechnung an einem der beiden datenhaltenden Knoten erfolgen, wird der Transfer natürlich für die kleinere Relation durchgeführt.
2. *Satzweises Anfordern von Verbundpartnern ("Fetch As Needed")*
   In diesem Fall erfolgt die Join-Berechnung an einem der datenhaltenden Knoten K_R bzw. K_S. Für jedes Tupel der lokal vorliegenden Relation wird der Wert des Join-Attributs an den anderen Rechner geschickt. Dieser bestimmt daraufhin die Tupel mit übereinstimmenden Werten im Verbundattribut und überträgt sie an den anfordernden Rechner.

Der Vorteil der ersten Methode (Ship Whole) liegt darin, daß sie mit einer minimalen Nachrichtenanzahl auskommt. Dafür ist jedoch ein hohes Datenvolumen zu übertragen. Außerdem ist im Join-Rechner ggf. eine hohe Anzahl von E/A-Vorgängen für die zu erzeugenden temporären Relationen vorzunehmen. Der Ansatz scheint daher vor allem für kleinere Relationen von Interesse zu sein. Die Alternative (Fetch As Needed) verursacht dagegen eine sehr hohe Nachrichtenanzahl, da für jedes Tupel der ersten Relation eine Anforderung der Verbundpartner stattfindet. Dafür läßt sich ggf. der Übertragungsumfang kleiner halten, da nur die relevanten Tupel der zweiten Relation übertragen werden. Dies ist vor allem bei hoher Join-Selektivität vorteilhaft. Die Menge zu berücksichtigender Verbundpartner kann daneben aufgrund zusätzlicher Selektionsbedingungen reduziert werden. In diesem Fall läßt sich jedoch auch beim Ship-Whole-Ansatz der Kommunikationsumfang reduzieren, indem vor dem Transfer die anwendbaren Selektionen (und Projektionen) durchgeführt werden.

Zur Kostenabschätzung der einzelnen Join-Alternativen beschränken wir uns auf die Kommunikationsanteile. Dazu berücksichtigen wir zwei Komponenten, die Nachrichtenanzahl (*#Nachrichten*) sowie den Übertragungsumfang. Bezüglich des Übertragungsvolumens unterscheiden wir nicht zwischen unterschiedlichen Attributgrößen, sondern bestimmen lediglich die Anzahl zu übertragender Attributwerte (*#AW*).

Beispiel 6-8

Die Join-Strategien sollen zunächst für die in Abb. 6-12 gezeigten Beispielrelationen betrachtet werden. Das Join-Ergebnis (über Attribut B) enthält lediglich 2 Tupel; es liegt also eine hohe Join-Selektivität vor. Insbesondere ist nicht jedes R- bzw. S-Tupel im Join-Ergebnis repräsentiert. Zur Bestimmung der Kommunikationskosten berücksichtigen wir nur die Join-Berechnung selbst, nicht also die Nachrichten zum Starten des Verbundes

oder der Weiterleitung des Ergebnisses. Für die einzelnen Strategien ergeben sich folgende Kosten:

S1 (Ship Whole, Join-Berechnung am R-Knoten):
#Nachrichten = 2; #AW = 18.

S2 (Ship Whole, Join-Berechnung am S-Knoten):
#Nachrichten = 2; #AW = 14.

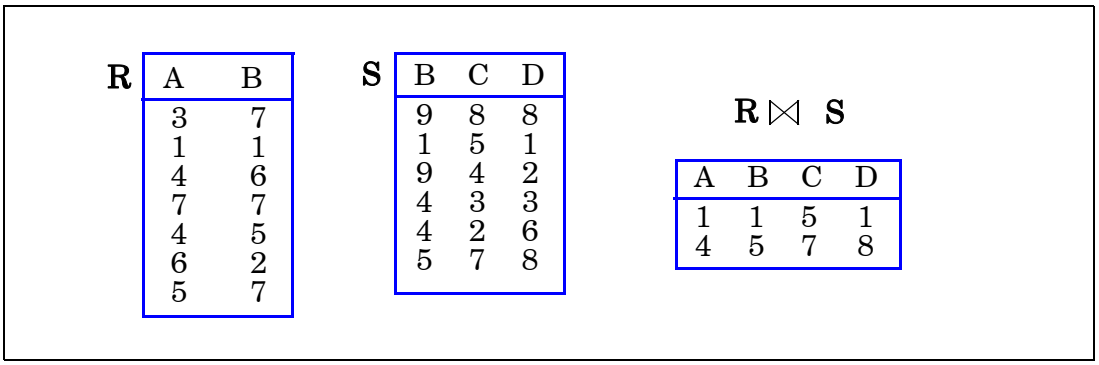
S3 (Ship Whole, Join-Berechnung an drittem Knoten):
#Nachrichten = 4; #AW = 32.

F1 (Fetch As Needed; Join-Berechnung am R-Knoten):
#Nachrichten = 7 * 2 = 14; #AW = 2 * 3 = 6.

F2 (Fetch As Needed; Join-Berechnung am S-Knoten):
#Nachrichten = 6 * 2 = 12; #AW = 2 * 2 = 4.

Man erkennt, daß die Fetch-As-Needed-Strategien aufgrund der hohen Join-Selektivität den geringsten Nachrichtenumfang verursachen, jedoch eine deutlich höhere Nachrichtenanzahl.

Abb. 6-12: Beispielrelationen zur Join-Berechnung



Beispiel 6-9

Zwischen den Relationen KONTO (10000 Tupel) und KUNDE (5000 Tupel) soll folgende (hierarchische) Join-Anfrage bestimmt werden:

```
SELECT * FROM KUNDE, KONTO
WHERE KUNDE.KNR=KONTO.KNR  AND KONTO.KTOSTAND > 20000
```

Beide Relationen sollen je 10 Attribute umfassen. Falls 5% der Konten mehr als 20000 DM aufweisen (SF = 0.05), ergeben sich folgende Ausführungskosten für die Join-Strategien:

S1 (Ship Whole, Join-Berechnung am KONTO-Knoten):
#Nachrichten = 2; #AW = 5 000 * 10 = 50 000.

S2 (Ship Whole, Join-Berechnung am KUNDE-Knoten):
#Nachrichten = 2; #AW = 500 * 10 = 5 000.
Es wird nicht die gesamte KONTO-Relation übertragen, sondern zunächst die Selektion auf KTOSTAND ausgewertet (ansonsten wäre #AW = 100 000).

S3 (Ship Whole, Join-Berechnung an drittem Knoten):
#Nachrichten = 4; #AW = 55 000.

F1 (Fetch As Needed; Join-Berechnung am KONTO-Knoten):
#Nachrichten = 500 * 2 = 1000; #AW = 500 + 500 * 10 = 5 500.

Für jeden KONTO-Satz sind 1 Attributwert (Join-Attribut KNR) an K_{KUNDE} sowie 10 Attributwerte von K_{KUNDE} an K_{KONTO} zu übertragen

F2 (Fetch As Needed; Join-Berechnung am KUNDE-Knoten):
#Nachrichten = 5 000 * 2 = 10000; #AW = 5000 + 500 * 10 = 10 000.

In diesem Beispiel ist Strategie S2 die beste, da sie auch die Selektionsbedingung zur Reduzierung des Kommunikationsumfangs nutzen kann. Von den Strategien S1 und F1, die beide eine Join-Berechnung am KONTO-Knoten vornehmen, schneidet F1 vom Nachrichtenumfang deutlich besser ab. Die Einzelanforderungen von Verbundpartnern (F1 und F2) verursachen wiederum eine sehr hohe Nachrichtenanzahl.

Die Leistungsanalyse in [ML86] zeigte, daß der Transfer ganzer Relationen in den meisten Fällen günstiger abschneidet als der Fetch-As-Needed-Ansatz, und zwar auch bei geographischer Verteilung, wo der Nachrichtenumfang stärker als bei lokaler Verteilung ins Gewicht fällt. Der Grund liegt in der exzessiv hohen Nachrichtenanzahl der Fetch-As-Needed-Strategie, die für jedes Tupel die Verbundpartner anfordert.

6.5.2 Semi-Join-Strategien

Der Overhead des Fetch-As-Needed-Ansatzes kann jedoch leicht reduziert werden, indem nicht für jeden Satz einer Relation die Verbundpartner einzeln angefordert werden, sondern alle auf einmal. Dies ist die Idee, die von Semi-Join-Strategien genutzt wird. Hierzu werden sämtliche Verbundattributwerte der ersten Relation in einer Nachricht an den Knoten der zweiten Relation übertragen. Daraufhin können alle Verbundpartner in einer Nachricht an den ersten Knoten zurückgesendet werden, wo dann die Join-Bildung erfolgt.

Diese Vorgehensweise entspricht der Anwendung eines Semi-Joins. Denn es gilt:

$$R \bowtie S \quad \Leftrightarrow \quad R \bowtie (S \ltimes R) \Leftrightarrow R \bowtie (S \bowtie \pi_V (R))$$

wobei V dem Verbundattribut entspricht (bzw. der Menge der Verbundattribute). Die Join-Berechnung $R \bowtie S$ läuft demnach in folgenden Schritten ab:

1. In K_R: Bestimmung der Verbundattributwerte $\pi_V (R)$ und Übertragung an K_S
2. In K_S: Bestimmung von $Z = (S \bowtie \pi_V (R)) = (S \ltimes R)$ und Übertragung an K_R
3. In K_R: Bestimmung von $R \bowtie Z = R \bowtie S$.

Aufgrund folgender Äquivalenzen

$$\begin{aligned} R \bowtie S \quad &\Leftrightarrow R \bowtie (S \ltimes R) \\ &\Leftrightarrow (R \ltimes S) \bowtie S \\ &\Leftrightarrow (R \ltimes S) \bowtie (S \ltimes R) \end{aligned}$$

gibt es bei zwei Relationen jedoch insgesamt (wenigstens) drei Semi-Join-Strategien. Die zweite Strategie entspricht der ersten, wobei die Rollen von R und S vertauscht sind (Join-Bildung am S-Knoten). Die dritte Strategie enthält zwei Semi-Joins und kann z.B. sinnvoll sein, wenn das Gesamtergebnis an einem dritten Knoten berechnet werden soll. In letzterem Fall werden insgesamt fünf Teilschritte notwendig:

1a. In K_R: Bestimmung der Verbundattributwerte $\pi_V(R)$ und Übertragung an K_S

1b. In K_S: Bestimmung der Verbundattributwerte $\pi_V(S)$ und Übertragung an K_R

2a. In K_S: Bestimmung von $Z1 = (S \bowtie \pi_V(R)) = (S \ltimes R)$ und Übertragung an K

2b. In K_R: Bestimmung von $Z2 = (R \bowtie \pi_V(S)) = (R \ltimes S)$ und Übertragung an K

3. In K: Bestimmung von $Z1 \bowtie Z2 = R \bowtie S$.

Dabei entspricht K dem Knoten, an dem das Gesamtergebnis bestimmt wird. Zu beachten ist, daß die Schritte 1a und 1b sowie 2a und 2b jeweils parallel ausgeführt werden können. Die Nachrichtenanzahl hat sich von zwei auf vier erhöht. Wird auf die Parallelbearbeitung der beiden Relationen verzichtet, läßt sich der Übertragungsumfang reduzieren, indem Knoten K_S nur die Join-Attributwerte an K_R (oder umgekehrt) zurückliefert, die tatsächlich im Verbundergebnis erscheinen (s. Aufgabe 6-8:).

Beispiel 6-10

Die drei Semi-Join-Strategien sollen auf die Beispielrelationen in Abb. 6-12 angewendet werden. Wir veranschaulichen die Einzelschritte für den komplizierteren Fall mit zwei parallelen Semi-Joins. Die fünf Schritte liefern folgende Zwischenergebnisse:

1a. In K_R: Bestimmung von $\pi_B(R)$

B
7
1
6
5
2

→ K_S

1b. In K_S: Bestimmung von $\pi_B(S)$

B
9
1
4
5

→ K_R

2a. In K_S: Bestimmung von $Z1 = (S \bowtie \pi_B(R))$

B	C	D
1	5	1
5	7	8

→ K

2b. In K_R: Bestimmung von $Z2 = (R \bowtie \pi_B(S))$

A	B
1	1
4	5

→ K

3. In K: Bestimmung von Z1 ⋈ Z2 = R ⋈ S.

A	B	C	D
1	1	5	1
4	5	7	8

Der Nachrichtenumfang dieser Strategie beträgt #AW = 19. Für die einfachen Semi-Join-Strategien betragen die Kosten #AW=11 bzw. #AW=8 für die Join-Berechnung an K_R bzw. K_S. Diese Kosten erhöhen sich jedoch um 8 Attributwerte, falls das Ergebnis an einen anderen Knoten transferiert werden muß.
Der Vergleich mit den einfachen Join-Strategien (Beispiel 6-8) zeigt, daß die Semi-Join-Methoden den Fetch-As-Needed-Ansätzen nicht nur in der Nachrichtenanzahl, sondern auch im Übertragungsumfang überlegen sind. Gegenüber Ship-Whole sind nur wenige zusätzliche Nachrichten erforderlich, jedoch kann das Übertragungsvolumen deutlich reduziert werden (8 vs. 14 AW).

Echte Zusatzkosten der Semi-Join-Strategien im Vergleich zum Transfer ganzer Relationen (Ship Whole) liegen in den Übertragungskosten für die Verbundattributwerte. Damit Semi-Join-Strategien insgesamt besser abschneiden, sind diese Kosten durch eine entsprechende Reduzierung bei der Rücklieferung der Verbundpartner zu amortisieren. Die Join-Selektivität muß also ausreichend hoch sein, so daß die Menge der angeforderten Verbundpartner (wesentlich) kleiner ist als die ganze Relation. Dies ist häufig der Fall, gilt jedoch z.B. nicht für viele hierarchische Joins, wenn nahezu jeder Primärschlüsselwert auch als Fremdschlüssel vorkommt. Die lokalen Join-Kosten können bei Semi-Join-Ansätzen auch höher liegen als beim Transfer ganzer Relationen, da mehr Teiloperationen auszuführen sind, wenngleich mit kleineren Operanden. Dennoch ist es z.B. erforderlich, einzelne Relationen mehrfach zu referenzieren (z.B. zunächst zur Bestimmung der Verbundattributwerte und später zur eigentlichen Join-Berechnung), was hohe E/A-Kosten verursachen kann.

6.5.3 Bitvektor-Join

Der Kommunikationsaufwand der Semi-Join-Strategien läßt sich reduzieren, indem die Verbundattributwerte auf einen kompakten Bitvektor abgebildet werden und nur dieser übertragen wird. Da die Abbildung i.a. über eine Hash-Funktion erfolgt, spricht man gelegentlich auch von einem *Hash-Filter-Join*.

Der Bitvektor B sei folgendermaßen definiert:

B: Array [1:n] of BIT.

Zur Abbildung der Verbundattributwerte wird eine Hash-Funktion h verwendet, die jeden Attributwert eindeutig auf eine ganze Zahl zwischen 1 und n abbildet. Für jeden Verbundattributwert wird das derart zugeordnete Bit im Bitvektor gesetzt. Als Verbundpartner werden alle Tupel ermittelt, für deren Verbundattributwert das Bit gesetzt ist. Die Join-Berechnung R ⋈ S läuft demnach in folgenden Schritten ab (im Bitvektor B seien anfangs alle Bits zurückgesetzt):

1. In K_R: Für jeden Attributwert v in π_V (R) setze zugehöriges Bit $B(h(v))$;
   Übertragung des Bitvektors B an K_S
2. In K_S: Bestimmung von Z = { s ∈ S | $B(h(s.v))$ gesetzt} und Übertragung an K_R
3. In K_R: Bestimmung von R ⋈ Z = R ⋈ S.

Die Einsparungen im Übertragungsvolumen durch Verwendung eines Bitvektors sind natürlich um so höher, je länger die Verbundattributwerte sind oder je mehr unterschiedliche Werte ansonsten zu übertragen wären. Ein weiterer Vorteil liegt darin, daß die Bestimmung der Verbundpartner in Schritt 2 sehr einfach wird. Hierzu ist kein Join durchzuführen, sondern es genügt das Lesen der Relation (Scan) in einer beliebigen Reihenfolge. Insbesondere dieser Aspekt ergab in Performance-Abschätzungen signifikant bessere Resultate für Bitvektor-Joins verglichen mit allgemeinen Semi-Join-Methoden [ML86].

Auf der anderen Seite ist die Hash-Abbildung i.a. nicht injektiv, so daß mit dem Bitvektor nur potentielle Verbundpartner bestimmt werden. Es qualifizieren sich dabei jedoch auch Tupel, die im Verbundergebnis nicht beteiligt sind. Die Übertragung solcher Tupel führt dazu, daß die durch den Bitvektor erreichten Einsparungen zumindest teilweise wieder zunichte gemacht werden. Das Ausmaß dieser Effekte hängt neben der Verteilung der Verbundattributwerte v.a. von der Länge des Bitvektors n und der Hash-Funktion h ab. Durch Wahl eines ausreichend großen n und geeigneter Hash-Funktionen kann die Filterwirkung jedoch meist sehr gut gehalten werden. Diese Problematik wurde bereits in anderen Kontexten untersucht, wo solche Hash-Filter (auch Bloom-Filter genannt) anwendbar sind [Bl70, SL76].

Beispiel 6-11

Der Bitvektor-Join soll auf die in Abb. 6-12 gezeigten Beispielrelationen angewendet werden. Wir betrachten die Join-Berechnung am Knoten K_S, da hiermit bei den Semi-Join-Methoden (Beispiel 6-10) der geringste Kommunikationsaufwand verbunden war ($\#AW$=8).Wir verwenden n=5 und h(v) = (v MOD n) + 1.

Die Berechnung des Bitvektors am Knoten K_S liefert den Wert B = 11001. In K_R wird damit neben (1,1) und (4,5) als dritter potentieller Verbundpartner das Tupel (4,6) ermittelt, da h(6)=2 und B(2) gesetzt ist. Dieses Tupel ist daher ebenfalls an K_S zu übertragen, wo es jedoch nicht ins Join-Ergebnis eingeht. Setzt man die Länge des Bitvektors einem Attributwert gleich, so ergibt sich für den Bitvektor-Join als Übertragungsvolumen $\#AW$=7.

6.5.4 Mehr-Wege-Joins

Mehr-Wege-Joins lassen sich generell als Folge von Zwei-Wege-Joins realisieren. Es gilt somit neben der Festlegung der Join-Strategie pro Verbund (Ship Whole, Semi-Join, Bitvektor-Join) vor allem die Reihenfolge der einzelnen Join-Berechnungen zu bestimmen. Die Bestimmung der günstigsten Reihenfolge ist jedoch sehr aufwendig, da die Anzahl möglicher Ausführungsstrategien exponentiell mit

der Relationenanzahl zunimmt. In R*, wo keine Semi-Join-Methoden unterstützt wurden, erfolgte dennoch eine solch vollständige Bewertung aller Join-Reihenfolgen (exhaustive search). Im SDD-1-Prototyp dagegen wurde die Ausführungsreihenfolge für Semi-Join-Operationen durch einen heuristischen Ansatz bestimmt [Be81]. Dabei wird zunächst eine mögliche Reihenfolge als Initiallösung festgelegt, die iterativ verbessert wird. Hierzu werden in jedem Schritt die Kommunikationskosten für die möglichen Semi-Joins bewertet. Die bis zu diesem Zeitpunkt festgelegte Ausführungsfolge wird dann um denjenigen Semi-Join erweitert, der bei dieser Bewertung die geringsten Kommunikationskosten verursacht.

Für zentralisierte DBS erfolgt die Bestimmung von Join-Reihenfolgen vor allem im Hinblick auf die Reduzierung von Zwischenergebnissen. Obwohl in Verteilten DBS die Größe der Zwischenrelationen die Kommunikationskosten beeinflußt, reicht es zu deren Minimierung i.a. nicht aus, diejenige Join-Reihenfolge zu finden, bei der die Vereinigung der Zwischenergebnisse den geringsten Umfang hat. Dies gilt selbst für die einfachen Ship-Whole-Strategien. Denn anstatt der Zwischenrelationen können auch die Basisrelationen übertragen werden, falls diese kleiner sind.

Beispiel 6-12

Zu berechnen sei der Join R⋈ S ⋈ T mit *Card (R)* = *Card (T)* = 10000, *Card (S)* = 1000, *JSF* (R⋈ S) = 0.01, *JSF* (S ⋈ T) =0.1. Jede der Relationen habe 10 Attribute. Bei den Kommunikationskosten beschränken wir uns auf den Umfang (*#AW*), da nur hierbei Unterschiede anfallen. Es bestehen u.a. folgende Ship-Whole Strategien (R -> K bedeutet Transfer von Relation R an Knoten K, Z bezeichne eine Zwischenrelation):

S1. R -> K_S. Z := R⋈ S. Z -> K_T. Berechnung des Ergebnisses Z⋈ T.
#AW = 10000 * 10 + 0.01 * 10000 * 1000 * 19 = 2 000 000
(bei einem einfachen Join-Attribut befinden sich in der Zwischenrelation 19 Attribute)

S2. S -> K_R. Z := R⋈ S. Z -> K_T. Berechnung des Ergebnisses Z⋈ T.
#AW = 1000 * 10 + 0.01 * 1000 * 10000 * 19 = 1 910 000

S3. S -> K_T. Z := S⋈ T. Z -> K_R. Berechnung des Ergebnisses Z⋈ R.
#AW = 1000 * 10 + 0.1 * 1000 * 10000 * 19 = 19 010 000

S4. T -> K_S. Z := S⋈ T. Z -> K_R. Berechnung des Ergebnisses Z⋈ R.
#AW = 10000 * 10 + 0.1 * 10000 * 1000 * 19 = 19 100 000

S5. R -> K_S; T -> K_S. Berechnung des Ergebnisses R⋈ S ⋈ T.
#AW = 10000 * 10 + 10000 * 10 = 200 000

S6. S -> K_R; T -> K_R. Berechnung des Ergebnisses R⋈ S ⋈ T.
#AW = 1000 * 10 + 10000 * 10 = 110 000.

Die Kommmunikationskosten liegen teilweise um mehr als zwei Größenordnungen auseinander (Faktor 175). Aufgrund der geringen Join-Selektivitäten schneiden diejenigen Strategien am besten ab, die nicht die Zwischenergebnisse, sondern die Basisrelationen zwischen den Knoten verschicken (S5 und S6).

Die Reduzierung des Kommunikationsaufwandes durch *Semi-Join-Ansätze* kommt für Mehr-Wege-Joins noch stärker zur Geltung. Denn ein Tupel wird nur

dann im Ergebnis eines Mehr-Wege-Joins berücksichtigt, wenn es Verbundpartner in allen anderen Relationen findet. Die Wahrscheinlichkeit dafür sinkt jedoch mit wachsender Relationenzahl. Das Optimierungsproblem besteht darin, eine Abfolge von Semi-Join-Operationen zu finden, die eine möglichst weitgehende Reduzierung beteiligter Relationen ermöglicht, so daß das Übertragungsvolumen entsprechend gering gehalten werden kann. Diejenige Lösung, welche jede der beteiligten Relationen auf diejenigen Tupel reduzieren kann, welche auch im Endergebnis repräsentiert sind, wird als *"vollständige Reduzierung"* (full reducer) bezeichnet. Leider bestehen nicht für alle Typen von Join-Anfragen solche vollständigen Reduzierungen, sondern nur für azyklische Join-Anfragen [BG81].

Beispiel 6-13

Zu berechnen sei für die in Abb. 6-13 gezeigten Beispielrelationen der Join R⋈ S ⋈ T. Eine mögliche Folge von Semi-Joins hierfür sieht folgendermaßen aus:

$$R' := R \ltimes S$$
$$S' := S \ltimes T$$
$$T' := T \ltimes S.$$

In jedem der drei Schritte wird eine der Relationen durch Semi-Join-Bildung reduziert. So wird Relation R im ersten Schritt um fünf Tupel reduziert, S und T im zweiten und dritten Schritt um jeweils zwei Tupel. Das Gesamtergebnis erhält man, indem man den Join der reduzierten Relationen berechnet (z.B. an einem weiteren Knoten). Es liegt jedoch keine vollständige Reduzierung vor, da von jeder der Relationen nur jeweils ein Tupel im Endergebnis repräsentiert ist; die angegebenen Semi-Joins bewirkten jedoch lediglich eine Reduzierung auf zwei bis vier Tupel.

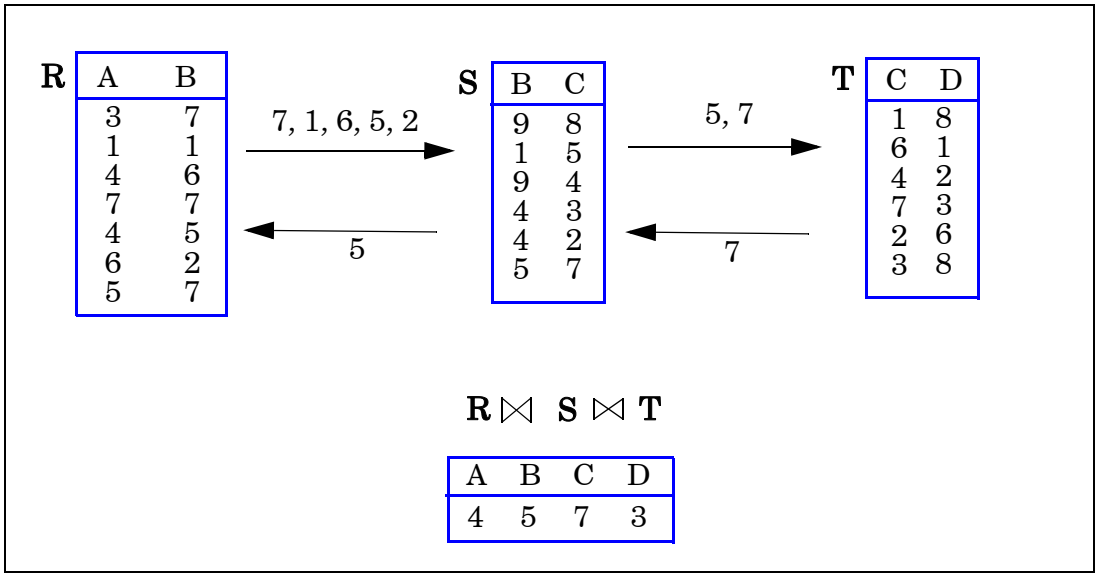
Die vollständige Reduzierung sieht folgendermaßen aus:

$$S' := S \ltimes R$$
$$T' := T \ltimes S'$$
$$S'' := S' \ltimes T'$$
$$R' := R \ltimes S''.$$

Im Gegensatz zur ersten Lösung erfolgt hier eine geschachtelte Semi-Join-Bildung, um den Reduzierungseffekt über mehrere Stufen fortzuführen. Die angegebene Abarbeitungsreihenfolge ist in Abb. 6-13 zusätzlich durch die Pfeile verdeutlicht, wobei die zur Semi-Join-Berechnung übertragenen Verbundattributwerte ebenfalls angegeben sind. Zunächst wird Relation S mit den Verbundattributwerten von R um vier Tupel auf die zwei Tupel (1,5) und (5,7) reduziert (Relation S'). Im zweiten Schritt werden die beiden verbleibenden Verbundattributwerte 5 und 7 zur Semi-Join-Berechnung mit T verwendet, wobei diese Relation auf ein einziges Tupel reduziert wird, nämlich (7,3). Damit erfolgt in Schritt 3 eine weitergehende Reduzierung von S auf ebenfalls ein Tupel (5,7). Im letzten Schritt erfolgt dann die Reduzierung von R auf Tupel (4,5). Es handelt sich um eine vollständige Reduzierung, da jede der beteiligten Relationen auf die Tupel reduziert werden konnte (hier: jeweils 1 Tupel), die im Endergebnis enthalten sind.

Die vollständige Reduzierung benötigte einen Schritt mehr als die erste Lösung, dafür reduziert sie das Übertragungsvolumen. Für den Transfer der Verbundattributwerte fallen bei der ersten Lösung 4+6+6=16 AW an, bei der vollständigen Reduzierung lediglich 5+2+1+1=9 AW. Die Haupteinsparung ergibt sich jedoch beim anschließenden Transfer der reduzierten Relationen zur Join-Berechnung. Wenn diese an einem separaten Knoten erfolgt, fallen bei der vollständigen Reduzierung lediglich 2+2+2=6 AW Übertragungskosten an, bei der ersten Lösung dagegen 4+8+8=20 AW.

Abb. 6-13: Beispielrelationen zur Mehr-Wege-Join-Berechnung



Die vollständige Reduzierung einer Anfrage ist unabhängig von den in den beteiligten Relationen vorliegenden Attributwerten [Ul88]; die im Beispiel genannte Lösung gilt somit für beliebige Ausprägungen der Relationen R, S und T. Leider ist die Bestimmung einer vollständigen Reduzierung auch für azyklische Join-Anfragen meist nicht mit polynomialem Aufwand möglich, so daß man häufig auf heuristische Lösungen zurückgreift. Eine Ausnahme besteht jedoch für die Teilklasse *geketteter Join-Anfragen* (chained queries), zu der auch die in Beispiel 6-13 verwendete Anfrage gehört. Eine gekettete Join-Anfrage hat allgemein die Form $R_1 (A, A_1) \bowtie R_2 (A_1, A_2) \bowtie \ldots \bowtie R_n (A_{n-1}, B)$,

wobei die A_i die Join-Attribute kennzeichnen. Es liegt also eine Ordnung unter den zu verknüpfenden Relationen vor, da jede Relation nur mit ihren jeweiligen "Nachbarn" in der Kette verknüpft werden kann. Zur Bestimmung der vollständigen Reduzierung genügen 2n-2 Semi-Joins, die innerhalb von zwei Phasen ausgeführt werden. Dabei erfolgt zunächst in einer Vorwärts-Reduzierung ausgehend von R_1 die Semi-Join-Bildung zwischen den jeweils benachbarten Relationen bis R_n:

$$R_2' := R_2 \ltimes R_1$$
$$R_3' := R_3 \ltimes R_2'$$
$$\ldots$$
$$R_n' := R_n \ltimes R_{n-1}'.$$

Danach erfolgt in der zweiten Phase eine Rückwärts-Reduzierung durch Semi-Join-Bildung in der umgekehrten Reihenfolge

$$R_{n-1}'' := R_{n-1} \ltimes R_n'$$
$$R_{n-2}'' := R_{n-2} \ltimes R_2''$$
$$\ldots$$
$$R_1' := R_1 \ltimes R_2''.$$

Diese Vorgehensweise wurde auch in Beispiel 6-13 angewendet.

Übungsaufgaben

Die Aufgaben beziehen sich auf folgende Relationen:

PERSONAL (PNR, PNAME, BERUF, GEHALT)

PROJEKT (PRONR, PRONAME, PROBUDGET)

PMITARBEIT (PNR, PRONR, DAUER).

Aufgabe 6-1: Normalisierung

Bestimmen Sie für die Qualifikationsbedingung der folgenden Anfrage die konjunktive und disjunktive Normalform:

```
SELECT * FROM PERSONAL
WHERE (PNAME LIKE "M%" AND BERUF = "Techniker") OR
((PNR > 550 OR BERUF="Programmierer") AND GEHALT < 80000)
```

Aufgabe 6-2: Vereinfachung

Vereinfachen Sie die Qualifikationsbedingung der folgenden Anfrage durch Anwendung der Idempotenzregeln:

```
SELECT * FROM PERSONAL
WHERE PNR > 456 AND
NOT (BERUF = "Techniker" OR GEHALT < 50000) AND
BERUF ≠ "Techniker" AND GEHALT < 50000
```

Aufgabe 6-3: Operatorbaum und Rekonstruktion

Führen Sie die Anfragetransformation für folgende Query durch:

```
SELECT PNAME, PRONAME
FROM PERSONAL P, PROJEKT PT, PMITARBEIT PM
WHERE DAUER > 10 AND P.PNR=PM.PNR AND
BERUF="Programmierer" AND PT.PRONR=PM.PRONR.
```

Bestimmen Sie den Operatorbaum und führen darauf Vereinfachungen und Restrukturierungen zur algebraischen Optimierung durch.

Aufgabe 6-4: Daten-Lokalisierung (abgeleitete Fragmentierung)

Gegeben sei folgende horizontale Fragmentierung von Relation PROJEKT:

$$PROJEKT_1 = \sigma_{PROBUDGET < 100000}(PROJEKT)$$
$$PROJEKT_2 = \sigma_{100000 \le PROBUDGET \le 800000}(PROJEKT)$$
$$PROJEKT_3 = \sigma_{PROBUDGET > 800000}(PROJEKT).$$

Für PMITARBEIT liege eine abhängige horizontale Fragmentierung vor:

$$PMITARBEIT_1 = PMITARBEIT \ltimes PROJEKT_1$$
$$PMITARBEIT_2 = PMITARBEIT \ltimes PROJEKT_2$$

$$PMITARBEIT_3 = PMITARBEIT \ltimes PROJEKT_3.$$

Bestimmen Sie für die Anfrage

```
SELECT PNR
FROM PROJEKT PT, PMITARBEIT PM
WHERE DAUER > 10 AND PROBUDGET > 1000000
AND PT.PRONR=PM.PRONR
```

zunächst den initialen Fragment-Ausdruck. Nehmen Sie algebraische Optimierungen zur weitestgehenden Reduzierung des Ausdrucks vor.

Aufgabe 6-5: Daten-Lokalisierung (hybride Fragmentierung)

Relation PERSONAL sei folgendermaßen fragmentiert:

$$PERSONAL_1 = \pi_{PNR, PNAME} (\sigma_{PNR < 20000} (PERSONAL))$$
$$PERSONAL_2 = \pi_{PNR, BERUF, GEHALT} (\sigma_{PNR < 20000} (PERSONAL))$$
$$PERSONAL_3 = \sigma_{PNR \geq 20000} (PERSONAL).$$

Bestimmen Sie für die Anfrage

```
SELECT PNAME
FROM PERSONAL
WHERE PNR=4711
```

zunächst den initialen Fragment-Ausdruck. Nehmen Sie algebraische Optimierungen zur weitestgehenden Reduzierung des Ausdrucks vor.

Aufgabe 6-6: Einfache Join-Strategien

Sei *Card* (R) = 10000, *Card* (S) = 1000, *JSF* (R⋈ S) = 0.001. Jede Relation soll 5 Attribute umfassen. Welche Kommunikationskosten ergeben sich für "Ship Whole" bzw. "Fetch as needed" bei Join-Ausführung an K_R bzw. an K_S ?

Aufgabe 6-7: Ship-Whole vs. Semi-Join vs. Bitvektor-Join

Auf den Relationen PERSONAL und PMITARBEIT sei folgende Join-Query zu bearbeiten:

```
SELECT P.PNR, PNAME, BERUF, PRONR, DAUER
FROM PERSONAL P, PMITARBEIT PM
WHERE P.PNR=PM.PNR AND P.GEHALT > 60000
```

Es gelte *Card* (PERSONAL) = 1000, *Card* (PMITARBEIT) = 1500; beide Relationen seien an verschiedenen Knoten gespeichert. Die Anfrage soll an einem dritten Knoten K initiiert werden; das Ergebnis ist dort auch auszugeben. Die Gehaltsbedingung soll von 20% der Angestellten erfüllt werden (*SF*=0.2); 25% der Angestellten sollen in keinem Projekt mitarbeiten.

Bestimmen Sie die Kommunikationskosten (#Nachrichten, #AW) für folgende Join-Strategien:

- Ship-Whole; Join-Berechnung an Knoten $K_{PMITARBEIT}$
- Ship-Whole; Join-Berechnung an Knoten K
- Semi-Join; Join-Bestimmung an Knoten $K_{PERSONAL}$
- Semi-Join; Join-Berechnung an Knoten K

- Bitvektor-Join; Join-Berechnung an Knoten K

Vor der Übertragung sollen alle anwendbaren Selektionen und Projektionen durchgeführt werden. Die Länge des Bitvektors soll 5 Attributwerten entsprechen; durch Anwendung des Bitvektors soll sich die zurückzuliefernde Tupelanzahl um 5% erhöhen.

Aufgabe 6-8: Semi-Join-Berechnung

Wie kann der Übertragungsumfang der in Kap. 6.5.2 vorgestellten Strategie mit zwei Semi-Joins und Join-Berechnung an Knoten K verbessert werden? Hinweis: Reduzieren Sie die Anzahl der Join-Attributwerte, die zwischen den Datenknoten K_R und K_S verschickt werden.

a) Geben Sie alle Einzelschritte an.

b) Wie reduziert sich der Übertragungsumfang für Beispiel 6-10.

Aufgabe 6-9: Mehr-Wege-Join

Bestimmen Sie für folgende Query

```
SELECT *
FROM PERSONAL P, PROJEKT PT, PMITARBEIT PM
WHERE P.PNR=PM.PNR AND PT.PRONR=PM.PRONR
AND BERUF="Programmierer".
```

die Kommunikationskosten für Ship-Whole und Semi-Join-Berechnung mit vollständiger Reduzierung. Jede der drei Relationen sei an einem separaten Knoten gespeichert. Ferner sei *Card* (PERSONAL) = 1000, *Card* (PMITARBEIT) = 1500*; Card* (PROJEKT) = 200. Die Anfrage soll an Knoten $K_{PERSONAL}$ initiiert werden; das Ergebnis ist dort auch auszugeben. Die Berufsbedingung soll von 10% der Angestellten erfüllt werden (*SF*=0.1); 25% der Angestellten sollen in keinem Projekt mitarbeiten.

7 Transaktionsverwaltung in Verteilten Datenbanksystemen

Die in Kap. 2.1.3 eingeführten ACID-Transaktionseigenschaften müssen natürlich auch von Verteilten Datenbanksystemen gewährleistet werden. Dies ist Aufgabe der Transaktionsverwaltung, bestehend aus Komponenten für Logging/Recovery, Synchronisation (Concurrency Control) und Integritätssicherung. Die verteilte Ausführung von Transaktionen führt zu einigen neuen Problemen, die bei der Realisierung dieser Funktionen zu lösen sind. Weiterhin erhöhen sich im verteilten Fall die Fehlermöglichkeiten (Verbindungsausfall, Nachrichtenverlust, Ausfall einer Teilmenge der Knoten etc.), auf die reagiert werden muß. Ein besonderes Problem bilden sogenannte *Netzwerk-Partitionierungen*. Bei ihnen entstehen aufgrund von Fehlern im Kommunikationssystem disjunkte Teilnetze oder Partitionen, so daß Rechner verschiedener Partitionen nicht mehr miteinander kommunizieren können.

Wir behandeln die Transaktionsverwaltung in drei Kapiteln. In Kap. 8 steht die Synchronisation in Verteilten DBS im Mittelpunkt; Kap. 9 widmet sich der Behandlung replizierter Datenbanken. Nachfolgend führen wir zunächst einige Begriffe ein und präzisieren die Struktur verteilter Transaktionen. Danach stellen wir verschiedene verteilte Commit-Protokolle vor (Kap. 7.2), deren Aufgabe die Einhaltung der Atomarität ist. Dies ist im verteilten Fall ein zentrales Problem, dessen Lösung die wichtigste Erweiterung hinsichtlich Logging und Recovery gegenüber zentralisierten DBS darstellt. In Kap. 7.3 betrachten wir dann noch notwendige Erweiterungen zur Integritätssicherung.

7.1 Struktur verteilter Transaktionen

Eine Transaktion besteht aus einer Folge von DB-Operationen, die durch eine BOT-Operation (Begin of Transaction) und eine Commit- oder EOT-Operation

(End of Transaction) geklammert sind. Daneben besteht die Möglichkeit, mit einer Rollback-Anweisung eine Transaktion abzubrechen. Diese Benutzersicht auf Transaktionen besteht für zentralisierte DBS und ist auch von Verteilten DBS bei Wahrung der ACID-Eigenschaften zu unterstützen (Verteilungstransparenz).

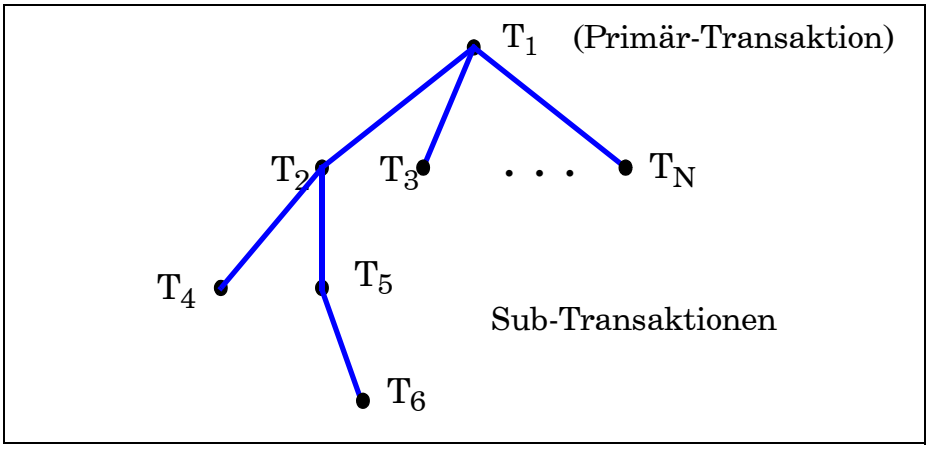
In Verteilten DBS können bei der Ausführung einer Transaktion ein oder mehrere Knoten beteiligt sein. Eine Sonderrolle kommt dabei dem Knoten zu, an dem die Transaktion gestartet wurde, wo also die BOT-Operation ausgeführt wurde. Dieser Rechner wird als *Heimat-* oder *Koordinator-Knoten* einer Transaktion bezeichnet. Die weiteren DB-Operationen sowie die Commit-Anweisung werden ebenfalls am Heimatknoten einer Transaktion gestartet, können jedoch weitere Knoten involvieren. Falls eine Transaktion vollständig an ihrem Heimat-Knoten ausgeführt wird, sprechen wir von einer *lokalen Transaktion*, anderenfalls von einer verteilten oder *globalen Transaktion*. An jedem bei der Ausführung einer globalen Transaktion T beteiligten Knoten wird eine *Teil-* oder *Sub-Transaktion* ausgeführt, die alle DB-Operationen bzw. Teiloperationen von T umfaßt, die an dem Knoten bearbeitet wurden. Die am Heimat-Knoten ausgeführte Teil-Transaktion wird auch als *Primär-Transaktion* bezeichnet.

Die Aufrufstruktur zwischen Primär- und Sub-Transaktionen bildet im allgemeinen Fall einen gerichteten Graphen, bei dem die beteiligten Rechner (Sub-Transaktionen) als Knoten fungieren. Zyklische Aufrufbeziehungen sind möglich, da z.B. während der Ausführung einer DB-Operation möglicherweise Daten eines Knotens benötigt werden, der bereits vorher an der Transaktion beteiligt war. Zur Transaktionsverwaltung reicht es jedoch aus, die Aufrufbeziehungen in reduzierter (hierarchischer) Form darzustellen, innerhalb eines sogenannten *Transaktionsbaumes* [GR93]. Darin bildet die Primär-Transaktion die Wurzel; jeder andere Knoten im Baum entspricht einer Sub-Transaktion eines anderen, an der Transaktionsausführung beteiligten Rechners. Dabei wird für jede Sub-Transaktion lediglich der erste Aufruf im Baum aufgenommen. Im Beispiel von Abb. 7-1 wurde so Teil-Transaktion T_5 zuerst von Sub-Transaktion T_2 aufgerufen. Weitere Anforderungen an den Rechner von T_5, z.B. während der Ausführung von T_3 oder T_4, haben keine Auswirkungen mehr auf den Transaktionsbaum. Ein Transaktionsbaum ist im allgemeinen Fall nicht balanciert und in der Höhe nur durch die Rechneranzahl beschränkt. Weiterhin können Teil-Transaktionen im allgemeinen parallel ausgeführt werden.

Die Struktur verteilter Transaktionen ist wesentlich durch die Datenverteilung und der von der Transaktion benötigten Daten bestimmt. Zwischen Primär- und Sub-Transaktionen besteht i.a. eine hierarchische Aufrufstruktur, die im Transaktionsbaum reflektiert ist. Allerdings bedeutet dies nicht, daß es sich bei verteilten Transaktionen um sogenannte *geschachtelte Transaktionen* (nested transactions) [Mo85] handelt. Geschachtelte Transaktion erlauben auch, eine Transaktion in eine Hierarchie von Sub-Transaktionen zu zerlegen, die parallel zueinander

ausgeführt werden können. Ein wesentliches Merkmal geschachtelter Transaktionen ist, daß Sub-Transaktionen isoliert zurückgesetzt werden können, ohne daß die Gesamt-Transaktion abgebrochen werden muß. Weiterhin erfolgt eine spezielle Synchronisation innerhalb einer Transaktionshierarchie. So können Sperren unter bestimmten Bedingungen an untergeordnete Sub-Transaktionen "vererbt" werden; am Ende einer Sub-Transaktion gehen deren Sperren in den Besitz der übergeordneten Transaktion über. Für die gesamte Transaktion gelten die herkömmlichen ACID-Eigenschaften, für Sub-Transaktionen dagegen lediglich die Atomarität und Isolation [HR93]. Geschachtelte Transaktionen könnten bei der Realisierung verteilter Transaktionen vorteilhaft eingesetzt werden, allerdings wird dieses Konzept in derzeitigen Implementierungen von Verteilten DBS nicht genutzt. Der Abbruch einer Sub-Transaktion resultiert daher in das Zurücksetzen der gesamten Transaktion.

Abb. 7-1: Beispiel eines Transaktionsbaumes



7.2 Commit-Behandlung

Bereits in zentralisierten DBS kommt der Commit-Behandlung durch das DBS eine entscheidende Rolle hinsichtlich der Atomarität und Dauerhaftigkeit von Transaktionen zu. Nachdem der Benutzer (bzw. das Anwendungsprogramm) mit der EOT-Anweisung das Transaktionsende signalisiert, werden DBVS-seitig zwei Phasen durchlaufen. Zunächst werden sämtliche Datenbankänderungen sowie ein sogenannter Commit-Satz auf die Log-Datei geschrieben (Phase 1)*. Wurde diese Phase erfolgreich abgeschlossen, d.h., der Commit-Satz wurde auf die Log-Datei geschrieben, ist das erfolgreiche Ende der Transaktion garantiert ("Alles"-Fall). Durch das Schreiben der Log-Daten ist die Wiederholbarkeit der Transaktion auch

* Wir gehen in diesem Kapitel stets von einer sogenannten Noforce-Strategie [HR83] zur Propagierung von DB-Änderungen aus. Dabei werden während der Commit-Behandlung (im Gegensatz zur Force-Alternative) geänderte DB-Seiten nicht in die physische Datenbank zurückgeschrieben, sondern die Änderungen lediglich in der Log-Datei protokolliert.

nach einem Rechnerausfall gewährleistet. In Phase 2 der Commit-Behandlung werden dann erst die Änderungen der erfolgreich beendeten Transaktion anderen Transaktionen sichtbar gemacht, z.B. durch Freigabe der Sperren. Transaktionen, die vor Abschluß der ersten Commit-Phase (z.B. durch einen Rechnerausfall) unterbrochen wurden, werden zurückgesetzt ("Nichts"-Fall).

Zur Wahrung der Transaktions-Atomarität in Verteilten DBS müssen sich alle an einer globalen Transaktion T beteiligten Knoten auf ein gemeinsames Commit-Ergebnis einigen. T muß also entweder an allen Knoten abgebrochen werden (Abort) oder an allen Knoten erfolgreich zu Ende kommen (Commit). Um dies zu erreichen, ist ein verteiltes Commit-Protokoll erforderlich. Eine Grundforderung an ein geeignetes Protokoll ist die Korrektheit, d.h., es muß die Alles-oder-Nichts-Eigenschaft sowie die Dauerhaftigkeit von Transaktionen auch im Fehlerfall gewährleisten. Unter Leistungsgesichtspunkten sollten dabei möglichst wenige Nachrichten und Schreibvorgänge auf die Log-Datei benötigt werden, da beide Faktoren die Antwortzeiten verlängern und Overhead einführen. Eine weitere Forderung ist eine hohe Robustheit des Protokolls gegenüber Fehlern, um z.B. die Wahrscheinlichkeit einer "Blockierung" (s.u.) möglichst gering zu halten. Damit im Zusammenhang steht die Forderung, daß jeder der beteiligten Rechner möglichst lange das Recht haben sollte, eine globale Transaktion von sich aus abzubrechen ("unilateral abort"). Die Unterstützung einer hohen Robustheit erfordert jedoch i.d.R. zusätzliche Nachrichten und E/A-Vorgänge und geht damit auf Kosten der Leistungsfähigkeit. Da der Fehlerfall (hoffentlich) selten eintrifft, sollte daher der Optimierung des Leistungsverhaltens im Normalbetrieb Vorrang eingeräumt werden.

Bei den vorzustellenden Commit-Protokollen nehmen wir generell an, daß an jedem Knoten ein *Transaktions-Manager* (TM) existiert, der für die Transaktionsverwaltung der lokal ausgeführten (Teil-)Transaktionen verantwortlich ist und der mit den Transaktions-Managern der anderen Knoten im Rahmen des Commit-Protokolls kooperiert. Insbesondere verwaltet jeder TM eine lokale Log-Datei, in der sämtliche Commit-Entscheidungen protokolliert werden. Für Verteilte Datenbanksysteme kann der Transaktions-Manager als Teil der an jedem Knoten laufenden DBVS-Instanz aufgefaßt werden, so daß für Commit-Entscheidungen sowie DB-Änderungen dieselbe Log-Datei verwendet werden kann*.

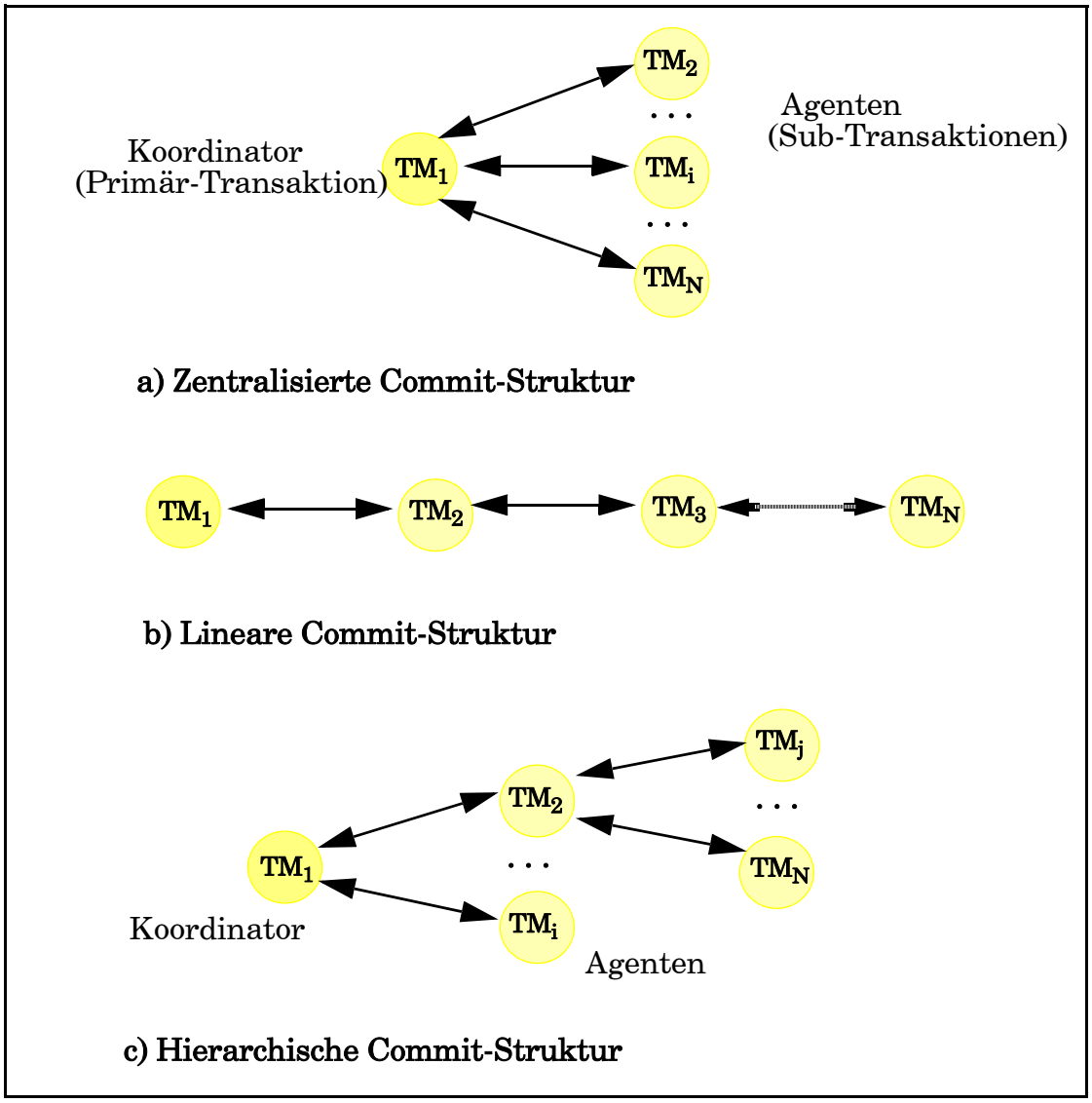
In verteilten Commit-Protokollen kommt dem Transaktions-Manager des Heimatknotens einer Transaktion die Sonderrolle des *Commit-Koordinators* zu. Aufgabe

---

* Im X/Open-Modell erfolgt dagegen eine Trennung zwischen Transaktions-Managern und sogenannten Resource-Managern wie dem DBVS (Kap. 2.1.4, Kap. 11.4.1). Dabei ist der TM im wesentlichen für die Commit-Behandlung zuständig, während Synchronisation und Logging den Resource-Managern überlassen bleiben. Dies ist vorteilhaft zur Unterstützung heterogener Resource-Manager, verlangt jedoch i.a. getrennte Log-Dateien und damit erhöhten Log-Aufwand.

des Koordinators ist die Ermittlung des globalen Commit-Ergebnisses (Commit oder Abort) und dessen Mitteilung an die an der Transaktionsausführung beteiligten Rechner. Die Transaktions-Manager der Rechner, an denen eine Sub-Transaktion ausgeführt wurde, wollen wir als *Agenten* bezeichnen.

Abb. 7-2: Kommunikationsstrukturen für verteilte Commit-Protokolle



Zur Kommunikation zwischen Koordinator und Agenten kommen unterschiedliche Aufrufstrukturen in Betracht, die in Abb. 7-2 gezeigt sind. Bei der zentralisierten Commit-Struktur (Abb. 7-2a) kommuniziert der Koordinator direkt mit jedem Agenten, während zwischen den Agenten i.d.R. keine Kommunikation stattfindet. Der Vorteil liegt vor allem darin, daß die Commit-Behandlung parallel mit allen Agenten durchgeführt werden kann. Die lineare Commit-Struktur (Abb. 7-2b) verlangt dagegen eine Sequentialisierung der Commit-Bearbeitung, erlaubt jedoch Einsparungen in der Nachrichtenanzahl (s.u.). Die hierarchische Struktur (Abb. 7-

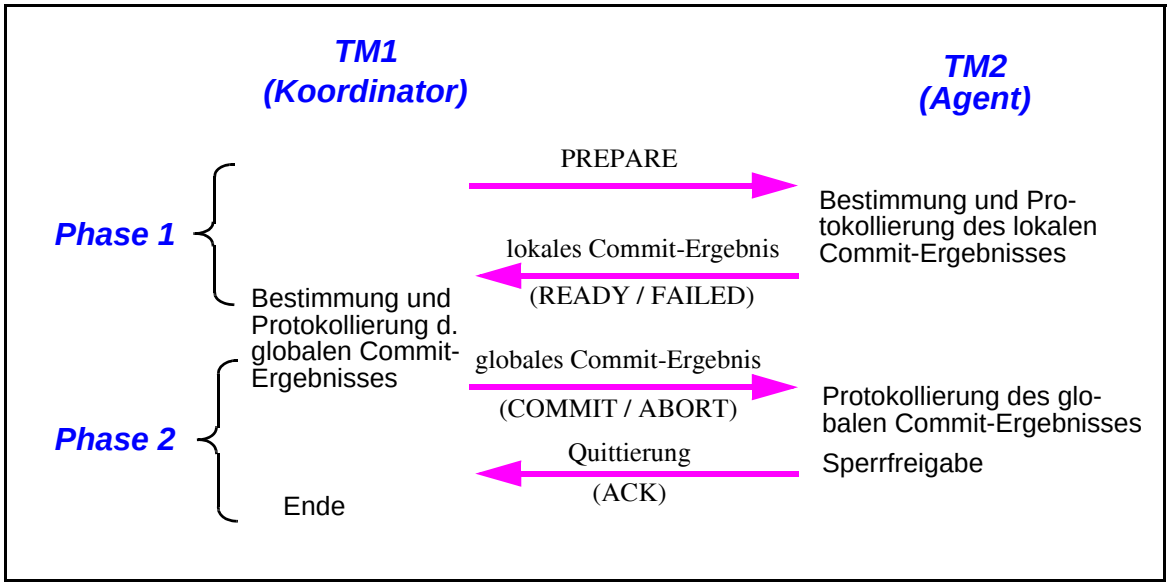
2c) schließlich berücksichtigt die hierarchische Aufrufstruktur innerhalb einer globalen Transaktion, wie sie im Transaktionsbaum repräsentiert ist. Sie kann als allgemeinste Struktur aufgefaßt werden, da sich die beiden ersten Ansätze als Spezialfälle ergeben.

Wir stellen im folgenden zunächst als Basismodell ein Zwei-Phasen-Commit-(2PC)-Protokoll vor, das die beiden eingangs erwähnten Phasen der Commit-Behandlung verteilt realisiert und auf der zentralisierten Kommunikationsstruktur basiert. Danach werden lineare und hierarchische 2PC-Protokolle erläutert. In Kap. 7.2.4 werden dann verschiedene Optimierungen verteilter Commit-Protokolle zur Reduzierung der Nachrichtenanzahl präsentiert. Abschließend behandeln wir noch ein 3-Phasen-Commit-Protokoll, welches vor allem eine höhere Robustheit im Vergleich zu 2PC-Verfahren anstrebt.

7.2.1 Verteiltes Zwei-Phasen-Commit (Basis-Protokoll)

Als Basis-Protokoll betrachten wir ein verteiltes Zwei-Phasen-Commit-Protokoll mit zentralisierter Kommunikationsstruktur zwischen Koordinator und Agenten [Gr78, MLO86]. Der Nachrichtenfluß dieses Protokolls ist in Abb. 7-3 skizziert, wobei die gezeigten Nachrichten für jeden Agenten anfallen. Im einzelnen laufen dabei folgende Schritte ab:

Abb. 7-3: Nachrichtenfluß im 2PC-Basisprotokoll



1. Bei Transaktionsende sendet der Koordinator eine PREPARE-Nachricht gleichzeitig an alle Agenten, um deren lokales Commit-Ergebnis in Erfahrung zu bringen.
2. Nach Empfang der PREPARE-Nachricht sichert der Agent einer erfolgreich zu Ende gekommene Sub-Transaktion deren Wiederholbarkeit durch das Aus-

schreiben von möglicherweise noch ungesicherten Log-Daten sowie eines Prepared-Satzes auf die lokale Log-Datei. Anschließend sendet der Agent eine READY-Nachricht an den Koordinator zurück. Danach wartet der Agent bis der Koordinator den Ausgang der globalen Transaktion (Commit oder Abort) mitteilt.
Für eine gescheiterte Sub-Transaktion werden ein Abort-Satz auf die lokale Log-Datei geschrieben und eine FAILED-Nachricht zum Koordinator geschickt. Der Agent setzt die Sub-Transaktion zurück, wobei auch von ihr gehaltene Sperren freigegeben werden*. Da das Scheitern der globalen Transaktion damit feststeht, wird die Sub-Transaktion daraufhin bereits beendet.

3. Nach Eintreffen aller Antwortnachrichten der Agenten beim Koordinator ist Phase 1 beendet. Haben alle Agenten mit READY geantwortet (und war das lokale Commit-Ergebnis am Koordinator-Knoten auch positiv), schreibt der Koordinator einen Commit-Satz in die lokale Log-Datei, woraufhin die globale Transaktion als erfolgreich beendet gilt. Danach wird eine COMMIT-Nachricht gleichzeitig an alle Agenten gesendet.
   Stimmte mindestens ein Agent mit FAILED, so ist damit auch die globale Transaktion zum Scheitern verurteilt. Der Koordinator schreibt daher einen Abort-Satz auf seinen Log und sendet eine ABORT-Nachricht an alle Agenten, die mit READY gestimmt haben.

4. Ein Agent schreibt nach Eintreffen einer COMMIT-Nachricht ebenfalls einen Commit-Satz auf die Log-Datei und gibt anschließend die Sperren der Sub-Transaktion frei. Bei einer ABORT-Nachricht werden ein Abort-Satz geschrieben und die Sub-Transaktion zurückgesetzt, wobei gehaltene Sperren ebenfalls freigegeben werden.
   Der Agent sendet danach zur Bestätigung noch eine Quittung (ACK-Nachricht) an den Koordinator. Nach Eintreffen aller ACK-Nachrichten beim Koordinator ist die globale Transaktion beendet, was durch einen Ende-Satz in der Log-Datei des Koordinators vermerkt wird.

Das Basisprotokoll erfordert pro Agent 4 Nachrichten, so daß bei einer über N Knoten verteilten globalen Transaktion insgesamt 4*(N-1) Nachrichten zur Commit-Behandlung anfallen. Weiterhin sind für den Koordinator sowie jeden Agenten zwei Schreibvorgänge auf die Log-Datei erforderlich. Bis auf den Ende-Satz sind diese Schreibvorgänge synchron, das heißt, die weitere Verarbeitung wird durch die E/A-Dauer verzögert. Es fallen somit insgesamt 2N-1 solcher synchronen Log-Vorgänge an.

Jeder an einer globalen Transaktion beteiligte Agent hat bis zur lokalen Commit-Entscheidung in Phase 1 das Recht des "unilateral abort", d.h. des einseitigen

* Wir gehen in diesem Kapitel davon aus, daß Sperrverfahren zur Synchronisation verwendet werden.

Transaktionsabbruchs. Dieses Recht wird jedoch nach Senden der READY-Nachricht aufgegeben; stattdessen wird die Verpflichtung übernommen, das globale Commit-Ergebnis des Koordinators zu akzeptieren. Die damit eingeführte Abhängigkeit zum Koordinator ist ein Hauptproblem des 2PC-Protokolls. Denn ein Koordinatorausfall kann dazu führen, daß andere Knoten lange Zeit auf das globale Commit-Ergebnis warten müssen (i.a. bis der Koordinator-Knoten wieder funktionsfähig ist). Da jedoch Sperren für die betroffenen Sub-Transaktionen bis zur globalen Entscheidung zu halten sind, kann dies zu erheblichen Leistungsproblemen ("Blockierungen") für andere Transaktionen führen.

Die Korrektheit des Protokolls im Normalbetrieb ist offensichtlich. Es bleibt also noch zu zeigen, wie unterschiedliche Fehlersituationen korrekt behandelt werden können. Hierzu ist die Betrachtung von Zustandsübergängen während der Commit-Bearbeitung hilfreich, wie sie in Abb. 7-4 für den Koordinator sowie die Agenten gezeigt sind. Die Knoten entsprechen dabei den einzelnen Zuständen, die Verbindungen zwischen ihnen den Zustandsübergängen. Zustandsübergänge sind durch zwei Angaben gekennzeichnet, die den Übergang auslösende Aktion (oben) sowie die dadurch veranlaßten Aktionen (unten).

Die Diagramme repräsentieren eine andere Darstellung des oben beschriebenen 2PC-Protokolls. Neu hinzugekommen sind vor allem die Berücksichtigung von *Timeout*-Ereignissen, über die Rechnerausfälle sowie Kommunikationsfehler abgefangen werden. Der Koordinator (Abb. 7-4a) befindet sich zunächst in einem Initialzustand (INITIAL). Nach Eingang der EOT-Operation wird für die betreffende Transaktion das Commit-Protokoll durch Verschicken der PREPARE-Nachrichten gestartet. Der Koordinator geht daraufhin in einen Wartezustand (WAIT). Nachdem alle Agenten mit READY gestimmt haben, wird das positive Commit-Ergebnis protokolliert und mitgeteilt; der Koordinator befindet sich jetzt im Zustand COMMITTING. Nach Eintreffen aller ACK-Nachrichten wird ein Ende-Satz auf den Log geschrieben, der den Endzustand TERMINATED kennzeichnet. Der Koordinator veranlaßt den Abbruch einer Transaktion, wenn einer der Agenten mit FAILED stimmt oder nicht innerhalb eines spezifizierten Timeout-Intervalls antwortet (Zustand ABORTING). Nach Eingang aller vom Abbruch informierten Agenten wird auch im Fehlerfall ein Ende-Satz auf den Log geschrieben. Wenn für die ACK-Nachrichten eine Timeout-Bedingung eintritt, dann protokolliert der Koordinator anstelle des Ende-Satzes in der Log-Datei, welche Agenten noch nicht geantwortet haben (in Abb. 7-4a nicht gezeigt).

Nach Ausführung einer Sub-Transaktion befindet sich der zugehörige Agent zunächst in einem Wartezustand WAIT, um auf die PREPARE-Nachricht zu warten (Abb. 7-4b). Wenn diese eintrifft, wird das lokale Commit-Ergebnis protokolliert und eine READY-Antwort an den Koordinator geschickt. Der Agent befindet sich danach im Zustand PREPARED, wo er auf das globale Commit-Ergebnis wartet. Trifft dieses ein, dann wird es auf die lokale Log-Datei protokolliert, eine Quittung

an den Koordinator gesendet und der Endzustand COMMITTED bzw. ABORTED eingenommen. Der Agent entscheidet auf ein lokales Abort, wenn die PREPARE-Aufforderung nicht innerhalb eines spezifizierten Timeout-Intervalls eintrifft oder ein sonstiger Fehler vorkommt (Empfang einer ABORT-Nachricht). In diesen Fällen wird direkt in den Zustand ABORTED gewechselt. Trifft in diesem Zustand die PREPARE-Nachricht des Koordinators noch ein, so wird mit FAILED geantwortet.

Beim Agenten droht eine Blockierung, wenn im PREPARED-Zustand eine Timeout-Bedingung einsetzt (in Abb. 7-4b nicht gezeigt). In diesem Fall kann versucht werden, beim Koordinator das globale Commit-Ergebnis nachzufragen, da das Ergebnis möglicherweise aufgrund eines Kommunikationsfehlers nicht empfangen werden konnte. Ist dies erfolglos (z.B. wegen Ausfall des Koordinators), kann erfragt werden, ob einer der anderen Agenten* möglicherweise das globale Commit-Ergebnis noch erhalten hat oder mit FAILED gestimmt hat (in letzterem Fall steht das Scheitern der gesamten Transaktion fest)**. Ist auch das nicht der Fall, muß gewartet werden, bis der Koordinator wieder funktionsfähig wird und das Ergebnis mitteilt ("Blockierung").

Wir diskutieren jetzt, wie die Recovery für globale Transaktionen nach einem Rechnerausfall aussieht. Wesentlich hierzu ist, daß mit dem Log-Inhalt festgestellt werden kann, welcher Commit-Zustand vor dem Rechnerausfall erreicht wurde. Bei der Crash-Recovery für einen *Agenten-Knoten* sind folgende Fälle möglich:

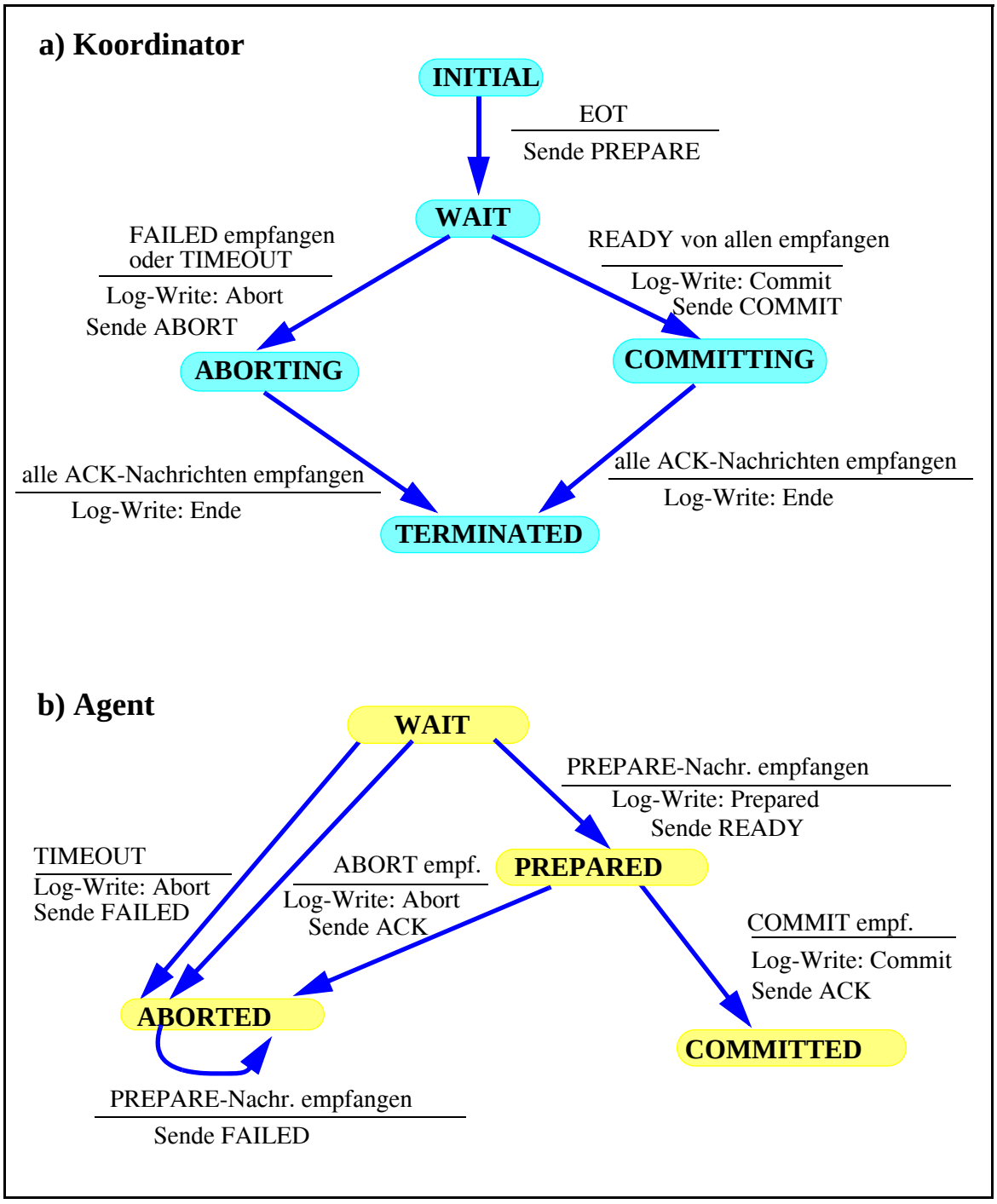
- Wird ein Commit-Satz auf dem Log gefunden (Zustand: COMMITTED), so handelt es sich um eine erfolgreich beendete Transaktion. Es wird eine Redo-Recovery vorgenommen, um Änderungen der Transaktion zu wiederholen, die aufgrund des Rechnerausfalls noch nicht in die physische Datenbank gelangten.
- Falls ein Abort-Satz vorliegt (Zustand: ABORTED), ist die Transaktion zurückzusetzen. Mit einer Undo-Recovery können Änderungen der Transaktion, die bereits in die physische Datenbank gelangten, zurückgenommen werden.
- Liegt ein Prepared-Satz vor (PREPARED-Zustand), dann wird das globale Commit-Ergebnis beim Koordinator nachgefragt. Dieser hält die Information noch, da er von dem ausgefallenen Agenten noch keine ACK-Nachricht erhalten hatte. Sollte der Koordinator zu diesem Zeitpunkt nicht verfügbar sein, so sind wieder die oben erwähnten Aktionen vorzusehen (Nachfragen bei anderen Agenten bzw. Blockierung).
- Liegt keiner der drei Log-Sätze vor, dann wird die Transaktion abgebrochen, da das Commit-Protokoll noch nicht begonnen wurde.

---

* Falls nicht alle Knoten befragt werden sollen, setzt dies voraus, daß der Koordinator z.B. mit der PREPARE-Nachricht mitteilt, welche Agenten noch an der Transaktion beteiligt sind.

** Diese Vorgehensweise wird z.B. in Cincom's Verteiltem DBS Supra-Server unterstützt und dort als "Transaction Partnering" bezeichnet (Kap. 19.11).

Abb. 7-4: Zustandsübergänge beim 2PC-Protokoll



Bei der Crash-Recovery für den *Koordinator-Knoten* bestehen folgende Möglichkeiten:

- Wird ein Ende-Satz auf dem Log gefunden (Zustand: TERMINATED), dann sind keine offenen Sub-Transaktionen mehr möglich. Je nach Commit-Ergebnis wird eine Redo- bzw. Undo-Recovery veranlaßt.
- Im Zustand ABORTING (Abort-Satz, jedoch kein Ende-Satz auf dem Log) wird eine Undo-Recovery vorgenommen. Agenten, die das (negative) Commit-Ergebnis noch

nicht quittiert haben, werden davon jetzt in Kenntnis gesetzt (falls auf dem Log nicht vermerkt wurde, von welchen Agenten die ACK-Nachricht noch aussteht, werden alle informiert).

- Im Zustand COMMITTING (Commit-Satz, jedoch kein Ende-Satz auf dem Log) wird eine Redo-Recovery veranlaßt. Agenten, die das (positive) Commit-Ergebnis noch nicht quittiert haben, werden davon jetzt in Kenntnis gesetzt.
- Liegt keiner der vorhergehenden Fälle vor, dann befand sich die Transaktion zum Zeitpunkt des Koordinatorausfalls in einem Zustand (INITIAL oder WAIT), in dem noch nicht alle lokalen Commit-Ergebnisse empfangen wurden. Es wird daher wie im ABORTING-Fall verfahren.

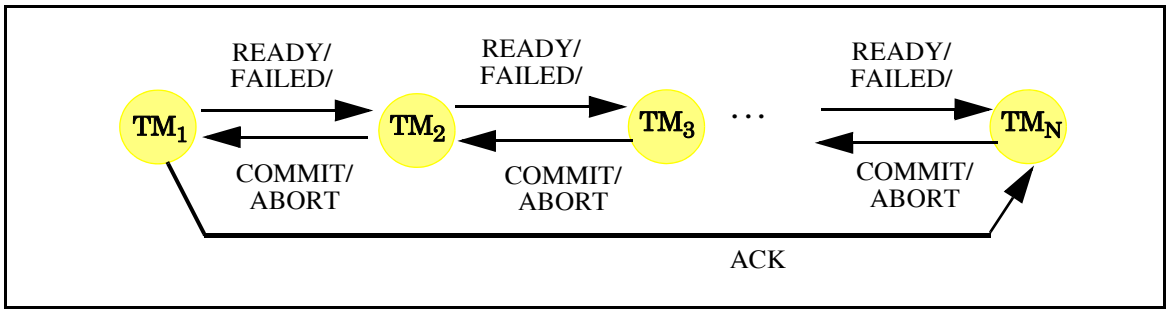
In den Zwischenzuständen wird das Commit-Protokoll jeweils wieder an dem Punkt fortgeführt, an dem die Unterbrechung stattfand. Insbesondere werden die noch ausstehenden Log-Sätze (z.B. das Commit-Ergebnis beim Agenten bzw. der Ende-Satz beim Koordinator) geschrieben, da sie bei einem erneuten Rechnerausfall benötigt werden.

Der Ausfall einer Kommunikationsverbindung wird wie beschrieben über Time-out-Bedingungen behandelt. Für nicht mehr erreichbare Knoten, auch im Rahmen von Netzwerk-Partitionierungen, ergibt sich somit die gleiche Behandlung wie für ausgefallene Rechner. Netzwerk-Partitionierungen bleiben nur dann ohne Auswirkungen, wenn Koordinator und alle Agenten in einer Partition verbleiben.

7.2.2 Lineares Zwei-Phasen-Commit

Beim *linearen 2PC-Protokoll* erfolgt die Commit-Behandlung sequentiell an den N Transaktions-Managern der an der Ausführung einer globalen Transaktion beteiligten Knoten. Wie Abb. 7-5 verdeutlicht, besteht Phase 1 aus einer Vorwärtskommunikation ausgehend vom Koordinator (TM_1) bis zum letzten Agenten TM_N, Phase 2 aus einer Kommunikationssequenz in umgekehrter Reihenfolge von TM_N bis TM_1.

Abb. 7-5: Nachrichtenfluß beim linearen 2PC-Protokoll



Die Commit-Behandlung beginnt im Erfolgsfall damit, daß der Koordinator sich zunächst selbst in den Prepared-Zustand bringt und seine lokale Commit-Entscheidung (READY) an TM_2 weitergibt. Ein Agent bringt sich nach Eintreffen einer READY-Nachricht ebenfalls in den Prepared-Zustand und gibt das READY an

den nächsten Agenten weiter. Der globale Erfolg der Transaktion steht fest, wenn der letzte Agent TM_N das READY empfangen und das Commit der gesamten Transaktion protokolliert hat. Das Commit-Ergebnis wird dann in umgekehrter Reihenfolge an die Agenten mitgeteilt, die dies daraufhin protokollieren und die Sperren freigeben. Nachdem TM_1 das Ergebnis erhalten und protokolliert hat, schickt er noch eine ACK-Nachricht an TM_N, der daraufhin einen Ende-Satz schreibt. Die Transaktion muß abgebrochen werden, sobald einer der Knoten in Phase 1 auf Abbruch entscheidet und entsprechend eine FAILED-Nachricht weitergibt.

Genaugenommen kommt bei dem linearen Commit-Protokoll dem letzten Agenten die Koordinatorrolle zu, da er das globale Commit-Ergebnis verbindlich protokolliert und weitergibt. Der Ansatz wird daher auch gelegentlich als *Transfer der Commit-Koordinierung* bzw. *"Last Agent"-Optimierung* bezeichnet [GR93, SBCM93]. Der Hauptvorteil liegt darin, daß die Nachrichtenanzahl gegenüber dem Basisverfahren mit zentralisierter Kommunikationsstruktur nahezu halbiert wurde. Denn durch den Wegfall separater PREPARE-Nachrichten und von N-2 ACK-Nachrichten verbleiben lediglich 2N-1 Nachrichten. Auf der anderen Seite führt die sequentielle Commit-Bearbeitung für größere N zu signifikanten Antwortzeiterhöhungen. Dieser Nachteil besteht jedoch nicht für den häufig auftretenden Fall N=2, der mit diesem Protokoll effizient bedient wird (3 Nachrichten).

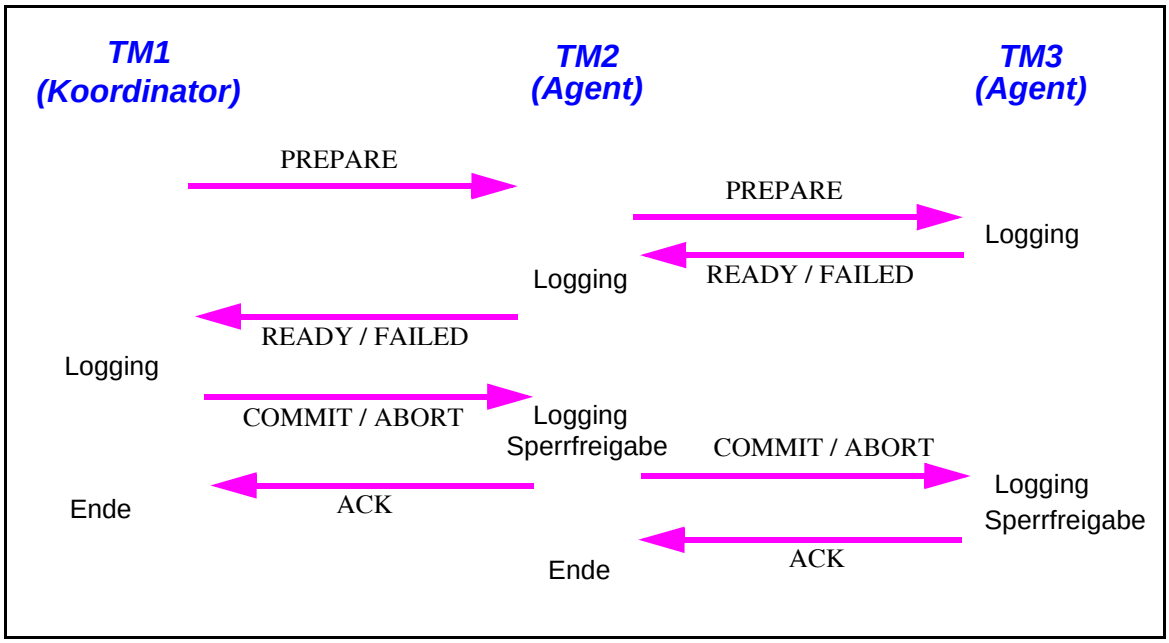
7.2.3 Hierarchische 2PC-Protokolle

Hierarchische Commit-Protokolle stellen eine Verallgemeinerung des Basisprotokolls für hierarchische Aufrufstrukturen dar, die durch Transaktionsbäume beschrieben werden können (Kap. 7.1). Dabei kommuniziert jeder Agent lediglich mit seinem direkten Vorgänger und den direkten Nachfolgern im Baum. Hierarchische Commit-Protokolle können daher auch in Umgebungen eingesetzt werden, in denen ein Agent nicht alle an einer globalen Transaktion beteiligten Knoten kennt, sondern nur die direkt mit ihm in Kontakt stehenden.

Wie Abb. 7-6 zeigt, kann das Basisprotokoll relativ einfach auf die hierarchische Struktur erweitert werden. Für den Wurzelknoten sowie die Blattknoten kommt das Basisprotokoll unverändert zur Anwendung. Eine neue Rolle nehmen dagegen die Zwischenknoten ein, da sie sowohl als Koordinator (gegenüber ihren Nachfolger-Agenten im Transaktionsbaum) als auch als untergeordneter Agent (gegenüber ihrem Vorgänger im Baum) fungieren. Wenn ein Zwischenknoten eine PREPARE-Nachricht erhält, wird diese Aufforderung an alle seine Nachfolger weiterpropagiert. Nach Eingang der lokalen Commit-Entscheidungen der Nachfolger, trifft der Zwischenknoten die Commit-Entscheidung für den gesamten Teilbaum, dessen Koordinator er ist, protokolliert sie in seine Log-Datei und gibt sie an seinen Vorgänger weiter. Im Fehler-Fall (Abort) kann der Zwischenknoten bereits zu diesem Zeitpunkt alle Nachfolger, die mit READY gestimmt haben, vom Scheitern

der Transaktion informieren. In der zweiten Commit-Phase nehmen die Zwischenknoten das Commit-Ergebnis der gesamten Transaktion von ihrem Vorgänger entgegen und protokollieren es. Danach wird das Ergebnis an die Nachfolger weitergeleitet und der Erhalt des Ergebnisses an den Vorgänger quittiert. Zwischenknoten schreiben ebenfalls einen Ende-Satz, nachdem sie alle Quittierungen ihrer Nachfolger erhalten haben.

Abb. 7-6: Nachrichtenfluß bei hierarchischen 2PC-Protokollen



Die Allgemeinheit und Flexibilität hierarchischer Commit-Protokolle geht jedoch auf Kosten einer geringeren Leistungsfähigkeit. Denn es wird dieselbe Nachrichtenanzahl wie im Basisprotokoll benötigt, allerdings führt die geringere Parallelisierung zu einer längeren Bearbeitungsdauer, die proportional zur Höhe des Baumes zunimmt. Weiterhin fällt für Zwischenknoten im Vergleich zu Blattknoten ein zusätzlicher (asynchroner) Log-Vorgang für den Ende-Satz an.

Diese Nachteile gelten auch für optimierte hierachische Commit-Protokolle, wie dem Presumed-Commit- oder dem Presumed-Abort-Ansatz [MLO86]. Eine große Bedeutung hat dabei vor allem das *Presumed-Abort-Protokoll* erlangt, daß in mehreren Produkten und auch Standards (ISO/OSI TP, X/Open DTP) verwendet wird [SBCM93]. Der Name "Presumed Abort" rührt daher, daß wenn in der Log-Datei des Koordinators keine Log-Sätze bezüglich des Commit-Zustandes einer globalen Transaktion vorliegen, auf Abort entschieden wird (während der Crash-Recovery oder bei einer Anfrage zum Transaktionsausgang). Ein Vorteil dieser Festlegung ist, daß der Koordinator den Abort-Satz nicht mehr synchron schreiben muß, sondern lediglich asynchron. Weiterhin können für eine gescheiterte Transaktionen die ACK-Nachrichten eingespart werden, und das Schreiben von Ende-

Sätzen beim Koordinator und in den Zwischenknoten entfällt ebenfalls. Für erfolgreiche globale Transaktionen ergeben sich allerdings keine Einsparungen. Besonders effektiv ist eine weitere Optimierung für lesende Sub-Transaktionen, die im Presumed-Abort-Protokoll vorgenommen wird. Da diese Optimierung jedoch für andere Commit-Protokolle auch nutzbar ist, beschreiben wir sie im folgenden Kapitel (Kap. 8).

7.2.4 Optimierungen von 2PC-Verfahren

Zwei Optimierungen von 2PC-Verfahren haben wir bereits diskutiert. Zum einen die Presumed-Abort-Optimierung und zum anderen die Verlagerung der Commit-Verantwortung zum letzten Agenten (lineares 2PC), welche vor allem für N=2 zu empfehlen ist. Wir beschreiben im folgenden zwei weitere 2PC-Optimierungen, mit denen Nachrichten und/oder (synchrone) Logging-Vorgänge eingespart werden können.

Optimierung lesender Sub-Transaktionen

Für Sub-Transaktionen, die keine Änderungen vorgenommen haben (read only), kann das Commit-Protokoll stark optimiert werden. Für solche Sub-Transaktionen ist weder ein Logging noch eine (Undo/Redo-) Recovery erforderlich. Im Rahmen der Commit-Behandlung sind für die Lese-Sub-Transaktionen daher lediglich die Sperren freizugeben. Dies kann jedoch bereits in Phase 1 eines 2PC-Protokolls erfolgen, da dies erforderlich ist unabhängig davon, ob die globale Transaktion erfolgreich zu Ende kommt oder abgebrochen werden muß. Folglich kann die zweite Commit-Phase für Lese-Sub-Transaktionen eingespart werden. Wenn M der N-1 Sub-Transaktionen Leser sind (M < N), reduziert sich damit die Nachrichtenanzahl im Basisprotokoll um 2M auf 4*(N-1) - 2M Nachrichten. Weiterhin verringert sich die Anzahl von Log-Schreibvorgängen auf 2N-M. Für reine Lesetransaktionen (M=N) werden keinerlei Log-Zugriffe sowie lediglich 2*(N-1) Nachrichten benötigt.

Viele existierende Datenbanksysteme verwenden lediglich “kurze” Lesesperren, die nicht bis zum Transaktionsende, sondern nur für die Dauer einer DB-Operation gehalten werden. Es wird dabei keine Serialisierbarkeit mehr erreicht*, jedoch wird dies in vielen Anwendungen zur Reduzierung der Konflikthäufigkeit in Kauf genommen. In diesem Fall kann die Commit-Behandlung für lesende Sub-Transaktionen sogar vollständig entfallen, da die Lesesperren zum Commit-Zeit-

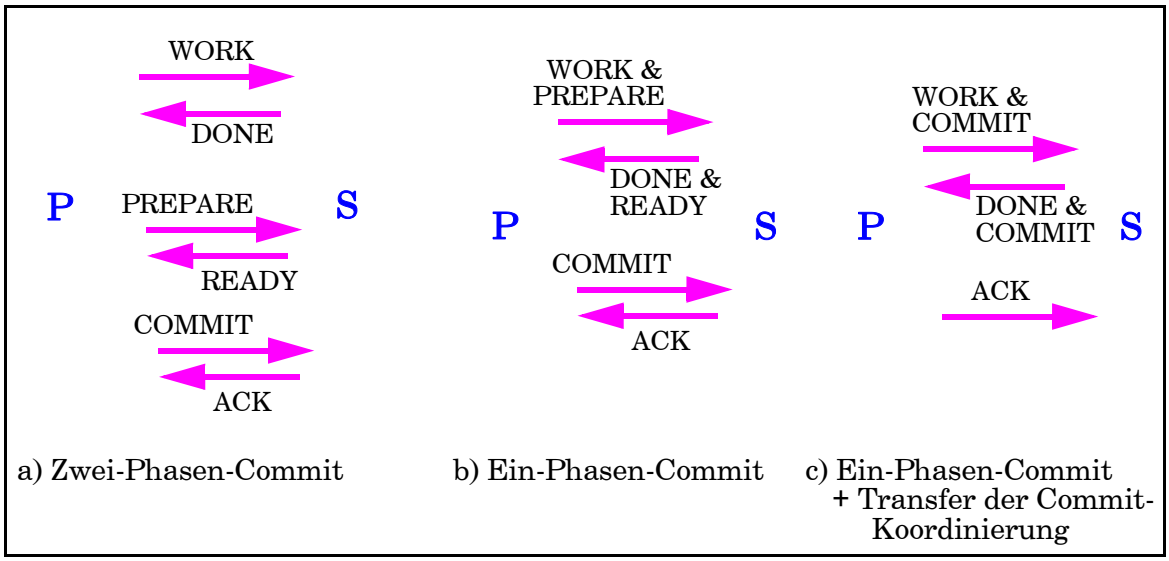
* Man spricht i.a. von Konsistenzebene 2 oder “Cursor Stability”, die in diesem Fall vorliegt. Im SQL92-Standard wird der mit kurzen Lesesperren einhergehende Isolationsgrad als "Read Committed" bezeichnet. Eine mögliche Folge kurzer Lesesperren ist die Anomalie "Non-Repeatable Read". Eine Transaktion kann also beim mehrmaligen Lesen eines Objektes unterschiedliche Zustände sehen, da die zwischenzeitliche Freigabe der Lesesperre eine parallele Änderung des Objektes ermöglicht. Schreibsperren werden generell bis Transaktionsende gehalten.

punkt bereits freigegeben sind. Für reine Lesetransaktionen ist daher kein Commit-Protokoll auszuführen. Anderenfalls (M<N) reduziert sich die Nachrichtenanzahl auf 4*(N-M-1).

1-Phasen-Commit

Bisher wurde angenommen, daß die von Sub-Transaktionen verrichtete "Arbeit" (DB-Operationen bzw. Teiloperationen) getrennt vom Commit-Protokoll durchgeführt wird. Wie Abb. 7-7a für das Basis-2PC-Protokoll zeigt, werden zur eigentlichen Transaktionsbearbeitung Sub-Transaktionen (S) über WORK-Aufrufe von der Primär-Transaktion (P) bzw. einer anderen Sub-Transaktion aufgefordert, bestimmte Operationen zu erledigen. Das Ergebnis wird über eine DONE-Nachricht zurückgeliefert. Das Commit-Protokoll wird dann am Transaktionsende wie beschrieben mit eigenen Nachrichten ausgeführt. Diese Vorgehensweise führt jedoch vor allem für kurze (verteilte) Transaktionen, die z.B. nur eine externe DB-Operation erfordern (etwa eine Überweisungs-Transaktion), dazu, daß ein Großteil der Bearbeitungszeit auf die Commit-Behandlung entfällt

Abb. 7-7: Zwei-Phasen- vs. Ein-Phasen-Commit.



Vor allem in solchen Fällen empfiehlt es sich als Optimierung, die PREPARE-Aufforderung bereits mit dem WORK-Aufruf zu kombinieren (Abb. 7-7b). Die Sub-Transaktion geht dann unmittelbar nach Durchführung ihrer Operationen und vor Rückmeldung an die rufende (Primär-) Transaktion in den Prepared-Zustand. Damit kann die erste Commit-Phase eingespart werden, so daß die eigentliche Commit-Bearbeitung nur noch aus einer Phase zur Mitteilung des Ergebnisses besteht ("1-Phasen-Commit"). Es werden somit pro Agent zwei Nachrichten eingespart. Eine weitere Nachricht kann für N=2 eingespart werden, wenn das Protokoll mit dem Transfer der Commit-Koordinierung kombiniert wird (Abb. 7-7c). In

diesem Fall wird mit dem WORK-Aufruf gleichzeitig die Commit-Koordination an den Agenten der betreffenden Sub-Transaktion übergeben. Nach Ausführung der Sub-Transaktion trifft der Agent daraufhin bereits die globale Commit-Entscheidung und teilt diese mit der DONE-Nachricht mit. Die Primär-Transaktion sendet lediglich noch eine ACK-Nachricht an den neuen Commit-Koordinator.

Das Ein-Phasen-Commit ohne Transfer der Commit-Koordinierung ist nicht auf zwei Rechner beschränkt. Wenn während einer Transaktion mehrere WORK-Aufrufe an denselben Rechner notwendig werden, dann sollte nur für den letzten die Kombination mit der PREPARE-Aufforderung vorgenommen werden, da ansonsten unnötiger Logging-Aufwand eingeführt würde. Ein potentielles Problem liegt darin, daß sich durch den frühzeitigen Übergang in den Prepared-Zustand die Wahrscheinlichkeit einer Blockierung erhöht. Das Ausmaß dieser Gefahr hängt natürlich davon ab, wieviel weitere Arbeit in einer globalen Transaktion noch auszuführen ist, nachdem eine Sub-Transaktion bereits in den Prepared-Zustand übergegangen ist. Für einfache Transaktionen ist dies kein Problem, bei einer größeren Rechneranzahl dagegen kann die Blockierungswahrscheinlichkeit deutlich steigen. Allerdings läßt sich das Ein-Phasen-Commit leicht mit dem Zwei-Phasen-Commit kombinieren. So könnte die Optimierung auf die am Ende einer globalen Transaktion auszuführenden Sub-Transaktionen beschränkt werden.

7.2.5 Drei-Phasen-Commit

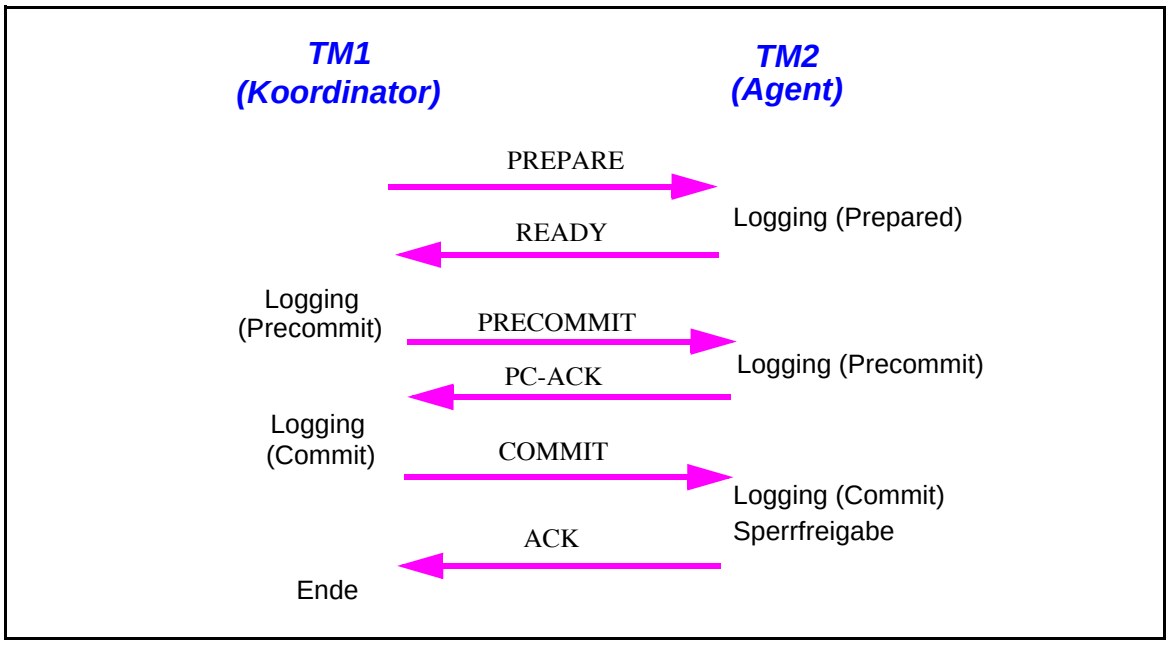
Ein (potentieller) Schwachpunkt der Zwei-Phasen-Commit-Protokolle liegt in der Abhängigkeit zum Koordinator, dessen Ausfall während des Prepared-Zustandes der Agenten zu langen Blockierungen führen kann*. Als Abhilfe wurden daher sogenannte "nicht-blockierende" Commit-Protokolle vorgeschlagen, die das Blockierungsproblem unter Inkaufnahme eines erhöhten Aufwandes abschwächen. Ein solches Verfahren ist das Drei-Phasen-Commit-Protokoll [Sk81, BHG87], das nachfolgend beschrieben wird. Es kann jedoch Blockierungen nur unter der Voraussetzung verhindern, daß keine Netzwerk-Partitionierungen vorkommen und höchstens K < N Rechner gleichzeitig ausfallen, wobei K ein Verfahrensparameter ist.

---

* Zur praktischen Relevanz des Blockierungsproblems gibt es keine Untersuchungen, obwohl nahezu ausschließlich Zwei-Phasen-Commit-Protokolle zur verteilten Commit-Behandlung im Einsatz sind. Allerdings werden in existierenden Systemen z.T. *"heuristische" Commit-Entscheidungen* zur Auflösung von Blockierungen vorgesehen [GR93, SBCM93]. Dabei entscheidet nach einem Koordinatorausfall der Operateur über das Commit-Ergebnis für eine blockierte Sub-Transaktion, oder es werden Default-Entscheidungen getroffen, um die Blockierung aufzulösen. Für derart "behandelte" Transaktionen kann allerdings keine Atomarität mehr garantiert werden. Wird später festgestellt, daß die heuristische Commit-Entscheidung von der des Koordinators abweicht, so kann der Schaden (Inkonsistenz der Datenbank) nur noch durch manuelle Recovery-Maßnahmen behoben bzw. begrenzt werden.

Wir beschreiben das 3PC-Protokoll für eine zentralisierte Kommunikationsstruktur zwischen Koordinator und Agenten. Die dabei anfallenden Nachrichten sind in Abb. 7-8 gezeigt. Man erkennt, daß die erste Phase mit der des 2PC-Protokolls übereinstimmt. Der (nicht gezeigte) Abort-Fall, wenn also wenigstens ein Agent mit FAILED stimmt, wird wie im 2PC-Protokoll behandelt. Eine zusätzliche Phase wird also nur notwendig, wenn alle Agenten mit READY stimmen. In dieser Situation nimmt der Koordinator jetzt zunächst einen Zwischenzustand (Precommit) ein, der mit einem entsprechenden Log-Satz protokolliert wird. Der Koordinator teilt das Precommit allen Agenten mit, die dieses Zwischenergebnis ebenfalls protokollieren und mit einer PC-ACK-Nachricht quittieren. Nachdem die PC-ACK-Nachrichten von K der N-1 Agenten eingetroffen sind, entscheidet der Koordinator auf Commit und schreibt den entsprechenden Log-Satz. Die letzte Phase (Mitteilung und Quittierung des Commit-Resultats) stimmt wiederum mit der des 2PC-Protokolls überein. Mit dem Precommit sichert der Koordinator zu, daß er von sich aus die Transaktion nicht mehr zurücksetzt. Allerdings ist es im Gegensatz zum Commit nach einem Precommit des Koordinators nach dessen Ausfall noch möglich, die Transaktion abzubrechen.

Abb. 7-8: Nachrichtenfluß beim Drei-Phasen-Commit



Wird während der Commit-Behandlung ein Koordinatorausfall erkannt (durch Timeout), so erfolgt die Wahl eines neuen Koordinators. Damit der neue Koordinator die Commit-Behandlung fortführen kann, erfrägt er zunächst von den überlebenden Rechnern den Commit-Zustand bezüglich der betroffenen Transaktion. Wenn einer der Agenten einen Commit-Satz oder Abort-Satz für die Transaktion protokolliert hat, wird das entsprechende Ergebnis global gültig gemacht. Wenn keiner der Agenten einen Abort- oder Commit-Satz, jedoch wenigstens einer einen

Precommit-Zustand vorliegen hat, dann wird das 3PC-Protokoll mit dem Verschikken der PRECOMMIT-Nachrichten fortgesetzt. Anderenfalls wird auf Abbruch der Transaktion entschieden.

Das im 2PC-Protokoll bestehende Blockierungsproblem im Prepared-Zustand eines Agenten ist mit diesem Ansatz gelöst. Denn wenn der Koordinator auf Abort entschieden hat, dies jedoch nicht mehr propagiert wurde, dann kann keiner der Agenten im Precommit-Zustand sein. Der neue Koordinator entscheidet daher richtigerweise auf Transaktionsabbruch. Hatte der ausgefallene Koordinator auf Commit entschieden, so findet der neue Koordinator bei wenigstens einem der überlebenden Agenten einen Precommit-Zustand vor (da nach Voraussetzung neben dem Koordinator höchstens K-1 weitere Rechner ausfallen dürfen und vor dem Commit K Agenten das Precommit quittierten). Allerdings entscheidet der neue Koordinator in dieser Situation nicht unmittelbar auf Commit, da dann der Ausfall des neuen Koordinators auf dasselbe Blockierungsproblem wie beim 2PC führen würde. Stattdessen wird das 3PC-Protokoll mit dem Verschicken der PRECOMMIT-Nachrichten fortgesetzt. Ist der ursprüngliche Koordinator im Precommit-Zustand ausgefallen, so kann die globale Transaktion sowohl abgebrochen als auch erfolgreich beendet werden. Welche Entscheidung vom neuen Koordinator gefällt wird, hängt davon ab, ob noch einer der Agenten den Precommit-Zustand erreicht hat.

Der Preis für diese höhere Robustheit liegt in der signifikanten Zunahme an Nachrichten und Log-Zugriffen. Insgesamt fallen jetzt 6*(N-1) Nachrichten sowie 3N Log-Vorgänge an. Die Voraussetzung, daß keine Netzwerk-Partitionierungen auftreten, ist notwendig, da ansonsten in mehreren Partitionen ein neuer Koordinator gewählt werden könnte. Die einzelnen Koordinatoren könnten dann zu unterschiedlichen Commit-Ergebnissen kommen.

Abschließender Vergleich

In Abb. 7-9 zeigen wir abschließend noch einmal den Nachrichtenbedarf für einige der vorgestellten Commit-Protokolle (mit optimierter Behandlung lesender Sub-Transaktionen). Es wird dabei wie bisher angenommen, daß die Transaktion an N Knoten ausgeführt wurde (N>1), wobei an M (< N) Knoten keine Änderungen erfolgten. Für das Ein-Phasen-Commit sowie das lineare 2PC ergeben sich für lesende Sub-Transaktionen keine Einsparungen. Denn bei langen Lesesperren sind auch für diese Protokolle 2 Nachrichten pro Agent zur Sperrfreigabe erforderlich.

Abb. 7-9: Nachrichtenbedarf verteilter Commit-Protokolle

	allgemein	Beispiel 1 (N=2, M=0)	Beispiel 2 (N=10, M=5)
1-Phasen-Commit	2*(N-1)	2	18
lineares 2PC	2*N-1	3	19
zentralisiertes / hierarchisches 2PC	4*(N-1)-2M	4	26
3-Phasen-Commit	6*(N-1)-4M	6	34

7.3 Integritätssicherung

Die zweite der vier ACID-Eigenschaften, Consistency, verlangt, daß Transaktionen die semantische Integrität der Datenbank bewahren. Hierzu können sogenannte *Integritätsbedingungen* spezifiziert werden, die festlegen, welche Datenbankzustände bzw. Zustandsübergänge korrekt sind, so daß DB-Inhalt und modellierte Miniwelt inhaltlich übereinstimmen. Eine Änderungstransaktion darf nur dann zum Commit kommen, wenn sie alle definierten Integritätsbedingungen einhält. Integritätsbedingungen lassen sich hinsichtlich verschiedener Kriterien klassifizieren, z.B. ihrer Reichweite (Attribut, Tupel, Relation, mehrere Relationen) oder Überprüfbarkeit (sofort oder verzögert am Transaktionsende) [Re87]. Eine Sonderrolle nehmen die modellinhärenten Bedingungen ein, die unabhängig von der jeweiligen Anwendung gewahrt bleiben müssen. Im Relationenmodell sind dies die beiden Relationalen Invarianten, also Eindeutigkeit des Primärschlüssels sowie die referentielle Integrität (Kap. 2.1.1).

Die Überwachung von Integritätsbedingungen in Verteilten DBS kann weitgehend wie im zentralen Fall erfolgen, jedoch erhöht sich der Overhead möglicherweise erheblich durch zusätzlich notwendig werdende Kommunikationsvorgänge. So kann es beim Einfügen eines neuen Tupels (bzw. Änderung des Primärschlüssels) zur Eindeutigkeitsprüfung des Primärschlüssels notwendig werden, sämtliche Knoten zu involvieren, an denen Fragmente der Relation vorliegen. Der Kommunikationsaufwand kann bei horizontaler Fragmentierung nur reduziert werden, wenn die Fragmentierung über Wertebereiche auf dem Primärschlüssel definiert wurde.

Bezüglich der referentiellen Integrität ist zu unterscheiden zwischen Änderungsoperationen auf der referenzierenden (abhängigen) Sohn-Relation S, die den Fremdschlüssel enthält, sowie der referenzierten Vater-Relation V, auf deren Primärschlüssel sich der Fremdschlüssel bezieht. Das Löschen eines S-Satzes sowie das Einfügen eines V-Satzes erfordern keine zusätzlichen Aktionen hinsichtlich

der referentiellen Integrität. Beim Einfügen eines S-Satzes bzw. Änderung des Fremdschlüsselattributs ist zu überprüfen, ob der neue Fremdschlüsselwert definiert ist. Dies erfordert einen zusätzlichen Kommunikationsvorgang, falls das betreffende V-Fragment an einem anderen Knoten vorliegt. Die beim Löschen eines V-Satzes oder Ändern eines Primärschlüssels vorzunehmenden Aktionen können in SQL92 anwendungsspezifisch festgelegt werden [DD92]. Eine Option ist, die Änderungen abzuweisen, sofern noch abhängige S-Tupel vorliegen. Diese Überprüfung erfordert im Worst-Case den Zugriff auf alle S-Fragmente. Dies gilt auch für die Alternativen, in der alle abhängigen Tupel ebenfalls gelöscht bzw. deren Fremdschlüssel geändert werden. Im günstigsten Fall werden jedoch zusätzliche Kommunikationsvorgänge vollständig vermieden; dies ist bei abhängiger horizontaler Fragmentierung möglich (Kap. 5.3.2). Die effiziente Implementierung der zur Überprüfung der referentiellen Integrität an jedem Rechner anfallenden Aufgaben wird in [HR92] diskutiert.

Ähnlich wie die Modellbedingungen erfordern auch die anwendungsbezogenen Integritätsbedingungen zusätzliche Verarbeitungsschritte und Kommunikationsvorgänge in Abhängigkeit der vorliegenden Datenverteilung. Eine Besonderheit ergibt sich bei Integritätsbedingungen, die verzögert am Transaktionsende auszuwerten sind*. Deren Überwachung ist jetzt ins Commit-Protokoll einzubinden, wofür ggf. auch Rechner zu involvieren sind, die an der Transaktionsausführung selbst nicht beteiligt waren. Bei einem Zwei-Phasen-Commit-Protokoll geschieht die Überprüfung in der ersten Commit-Phase, wobei dann der Prepared-Zustand natürlich nur bei erfolgreichem Test eingenommen wird.

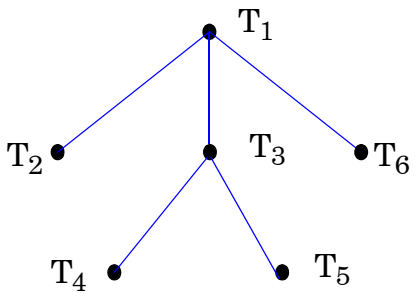
Übungsaufgaben

Aufgabe 7-1: Presumed-Abort - vs. Standard-2PC

Bestimmen Sie die Anzahl von Nachrichten und Log-Schreibvorgängen für Presumed-Abort und folgenden Transaktionsbaum wenn a) die Gesamt-Transaktion erfolgreich ist und alle Sub-Transaktionen Änderungen vornehmen, b) die Gesamt-Transaktion erfolgreich ist und Sub-Transaktionen T_4 und T_6 Leser sind und c) Sub-Transaktion T_2 mit FAILED stimmt und alle Sub-Transaktionen Änderungen vornehmen. Bestimmen Sie zum Vergleich den Aufwand im Basisprotokoll (keine Berücksichtigung der Hierarchie) für die drei

* Dies sind z.B. alle Integritätsbedingungen, die mehrere Relationen betreffen, da in SQL Änderungsoperationen jeweils auf eine Relation beschränkt sind.

Fälle, wobei die Optimierung für Lese-Sub-Transaktionen auch dort eingesetzt werden soll.



Aufgabe 7-2: Nicht-blockierendes 2PC durch Prozeß-Paare

Blockierungen im 2PC-Protokoll können verhindert werden, wenn der Koordinator als Prozeß-Paar fehlertolerant realisiert wird. Dabei sendet der Koordinator-TM sämtliche für die Commit-Bearbeitung relevanten Zwischenzustände an einen Backup-TM in einem anderen Rechner. Nach Ausfall des Koordinators übernimmt der Backup-TM dessen Rolle. Wie hoch ist der Overhead einer solchen Lösung gegenüber einem 3PC-Protokoll ?

8 Synchronisation in Verteilten Datenbanksystemen

Eine Schlüsseleigenschaft von DBS ist, daß viele Benutzer gleichzeitig lesend und ändernd auf die gemeinsamen Datenbestände zugreifen können, ohne daß die Konsistenz der Daten verletzt wird. Die Wahrung der DB-Konsistenz trotz paralleler Zugriffe ist Aufgabe der Synchronisationskomponente, wobei der Mehrbenutzerbetrieb gegenüber den Benutzern verborgen wird (Transparenz der konkurrierenden Verarbeitung, logischer Einbenutzerbetrieb). Werden alle Transaktionen seriell ausgeführt, dann ist der geforderte logische Einbenutzerbetrieb ohne jegliche Synchronisation erreicht, und die DB-Konsistenz ist am Ende jeder Transaktion gewährleistet. Eine strikt serielle Ausführung der Transaktionen zur Umgehung der Synchronisationsproblematik verbietet sich jedoch v.a. aus Leistungsgründen. Denn im Einbenutzerbetrieb könnten die Prozessoren aufgrund langer Transaktionsunterbrechungen, z.B. wegen Kommunikations- oder E/A-Vorgängen, nicht vernünftig genutzt werden.

Beim unkontrollierten Zugriff auf Datenobjekte im Mehrbenutzerbetrieb können eine Reihe von unerwünschten Phänomenen auftreten. Die wichtigsten dieser Anomalien sind verlorengegangene Änderungen ("lost update"), Lesen "schmutziger" Änderungen ("dirty read"), inkonsistente Analyse ("non-repeatable read") sowie sogenannte Phantome [Re87]. Änderungen können verlorengehen, wenn zwei Transaktionen parallel dasselbe Objekt ändern, wobei die zuerst vorgenommene Änderung durch die zweite Transaktion fälschlicherweise überschrieben wird. Das Lesen schmutziger Änderungen, welche von noch nicht zu Ende gekommenen Transaktionen stammen, führt zu fehlerhaften Ergebnissen, wenn die ändernde Transaktion noch zurückgesetzt werden muß und somit ihre Änderungen ungültig werden. Die inkonsistente Analyse sowie das Phantom-Problem betreffen Lesetransaktionen, die aufgrund parallel durchgeführter Änderungen während ihrer Ausführungszeit unterschiedliche DB-Zustände sehen.

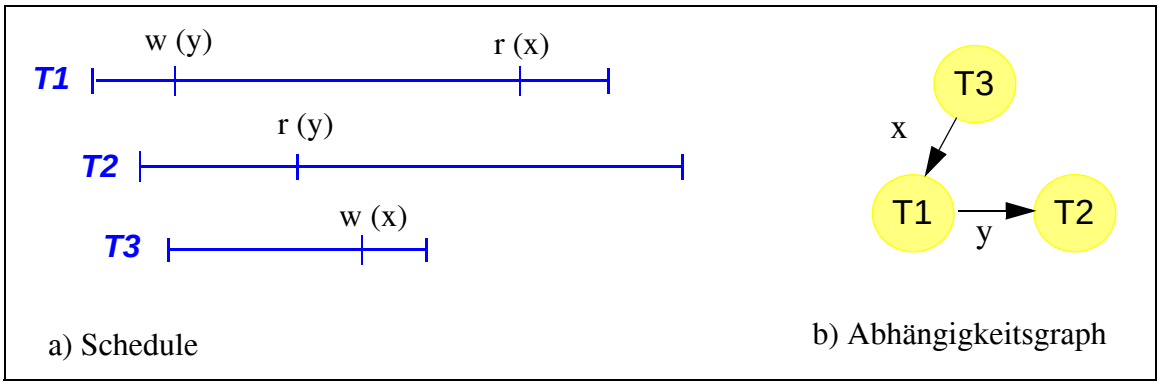
Die genannten Anomalien werden vermieden durch Synchronisationsverfahren, welche das Korrektheitskriterium der *Serialisierbarkeit* erfüllen [EGLT76, BHG87]. Dieses Kriterium verlangt, daß das Ergebnis einer parallelen Transaktionsausführung äquivalent ist zu dem Ergebnis irgendeiner der seriellen Ausführungsreihenfolgen der beteiligten Transaktionen. Äquivalent bedeutet in diesem Zusammenhang, daß für jede der Transaktionen dieselbe Ausgabe wie in der seriellen Abarbeitungsreihenfolge abgeleitet wird und daß der gleiche DB-Endzustand erzeugt wird. Dies erfordert, daß eine Transaktion alle Änderungen der Transaktionen sieht, die vor ihr in der äquivalenten seriellen Ausführungsreihenfolge stehen, jedoch keine der in dieser Reihenfolge nach ihr kommenden Transaktionen. Die zur parallelen Transaktionsbearbeitung äquivalente serielle Ausführungsreihenfolge wird auch als *Serialisierungsreihenfolge* bezeichnet.

Um zu überprüfen, ob ein bestimmter Schedule, d.h. eine Ablauffolge von Transaktionen mit ihren zugehörigen Operationen, serialisierbar ist, kann ein Serialisierbarkeits- oder *Abhängigkeitsgraph* geführt werden. In diesem Graphen treten die einzelnen Transaktionen als Knoten auf und die Abhängigkeiten zwischen Transaktionen als (gerichtete) Kanten. Eine Abhängigkeit zwischen zwei Transaktionen T_i und T_j liegt vor, wenn T_i vor T_j auf dasselbe Objekt zugegriffen hat und die Operationen der beiden Transaktionen nicht reihenfolgeunabhängig (kommutativ) sind. Werden (wie üblich) als Operationen Lese- und Schreibzugriffe betrachtet, dann liegt eine Abhängigkeit vor, wenn wenigstens eine der beiden Operationen eine Schreiboperation darstellt. Es kann gezeigt werden, daß ein Schedule genau dann serialisierbar ist, wenn der zugehörige Abhängigkeitsgraph azyklisch ist. Denn nur in diesem Fall reflektiert der Graph eine partielle Ordnung zwischen den Transaktionen, die zu einer vollständigen Ordnung, die zugleich den äquivalenten seriellen Schedule bestimmt, erweitert werden kann.

Beispiel 8-1

Abb. 8-1a zeigt einen Schedule mit drei Transaktionen, wobei r(x) bzw. w(x) den Lese- bzw. Schreibzugriff der jeweiligen Transaktion auf Objekt x kennzeichnen. Der zugehörige Abhängigkeitsgraph ist in Abb. 8-1b gezeigt, wobei die Kanten zusätzlich mit dem die Abhängigkeit verursachenden Objekt gekennzeichnet sind. Da der Graph keinen Zyklus enthält, ist der gezeigte Schedule serialisierbar. Die Serialisierungsreihenfolge lautet T3 < T1 < T2.

Abb. 8-1: Schedule und Abhängigkeitsgraph (Beispiel)



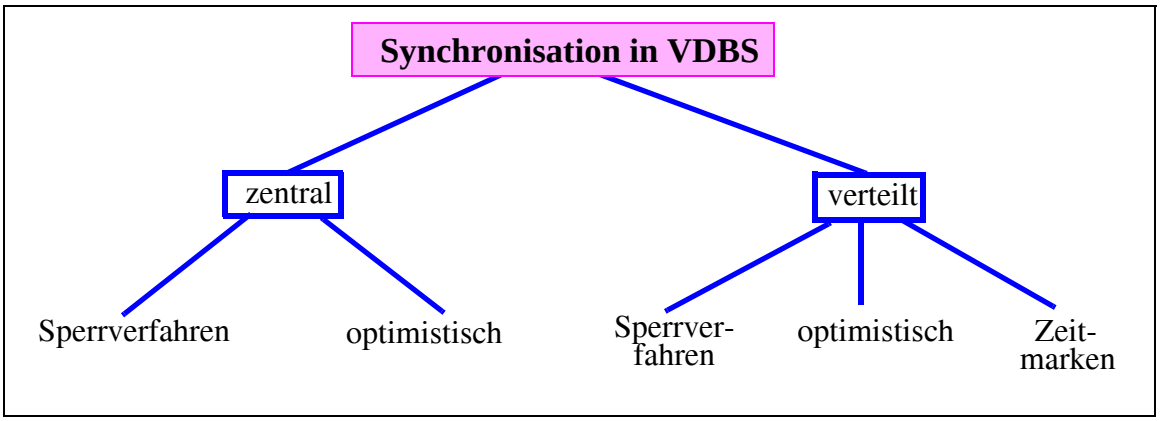
Das Führen von Abhängigkeitsgraphen bietet jedoch keinen praktikablen Ansatz zur Implementierung eines Synchronisationsverfahrens, da hiermit meist erst nachträglich die Serialisierbarkeit von Schedules geprüft werden kann [Pe87]. Weiterhin wäre der Verwaltungsaufwand prohibitiv hoch, zumal auch Abhängigkeiten zu bereits beendeten Transaktionen zu berücksichtigen sind. Zur Synchronisation wird daher auf andere Verfahren zurückgegriffen, für welche nachgewiesen werden konnte, daß sie Serialisierbarkeit gewährleisten. Die Mehrzahl der vorgeschlagenen Verfahren läßt sich einer der drei folgenden Klassen zuordnen: Sperrverfahren, optimistische Protokolle sowie Zeitmarkenverfahren.

Diese Verfahrensklassen kommen auch für Verteilte Datenbanksysteme in Betracht. Als neue Anforderung ergibt sich hier, eine systemweite Serialisierbarkeit aller lokalen und globalen Transaktionen zu erzielen (*globale Serialisierbarkeit*). Hierzu reicht es nicht aus, nur die (lokale) Serialisierbarkeit in jedem Rechner zu gewährleisten, da in den einzelnen Rechnern unterschiedliche Serialisierungsreihenfolgen vorliegen können. Um eine hohe Leistungsfähigkeit zu erhalten, sollte diese Aufgabe wiederum mit möglichst geringem Kommunikationsaufwand erfüllt werden. Weiterhin sollte, wie bereits für zentralisierte DBS, die Synchronisation mit möglichst wenig Blockierungen und Rücksetzungen von Transaktionen erfolgen. Denn diese zur Behandlung von Synchronisationskonflikten verfügbaren Methoden haben beide einen negativen Einfluß auf Durchsatz und Antwortzeiten. Eine weitere Anforderung im verteilten Fall ist eine möglichst hohe Robustheit der Synchronisationsprotokolle gegenüber Fehlern (Rechnerausfall, Kommunikationsfehler). Erweiterungen werden daneben bei replizierten Datenbanken notwendig, um die wechselseitige Konsistenz von Replikaten sicherzustellen und Transaktionen mit aktuellen Objektkopien zu versorgen.

Da replizierte Datenbanken in Kap. 9 behandelt werden, konzentrieren wir uns hier zunächst auf die Synchronisation in partitionierten Datenbanken. Eine grobe Übersicht der hierbei in Betracht kommenden Verfahrensklassen ist in Abb. 8-2 gezeigt. Hierbei wird zwischen verteilten und zentralisierten Verfahren unter-

schieden, wobei letztere auf einem ausgezeichneten Knoten durchgeführt werden. Beide Alternativen bestehen für Sperrverfahren sowie optimistische Protokolle, während für Zeitmarkenverfahren nur eine verteilte Realisierung sinnvoll ist. Wie schon in zentralisierten DBS können die einzelnen Verfahrensklassen auch in Verteilten DBS vielfältig miteinander kombiniert werden (z.B. Sperrverfahren mit optimistischen Protokollen).

Abb. 8-2: Synchronisationsverfahren in Verteilten DBS



Im folgenden diskutieren wir die Realisierung von Sperrverfahren, Zeitmarkenverfahren sowie optimistischen Ansätzen für Verteilte Datenbanksysteme. Danach gehen wir auf die Realisierung einer Mehrversionen-Synchronisation ein, mit der das Ausmaß an Sperrkonflikten reduziert werden kann. Ein solcher Ansatz stellt eine mit allen Verfahrensklassen kombinierbare Optimierung dar, die in kommerziellen (zentralisierten) DBS zunehmend an Bedeutung gewinnt. In Kap. 8.5 werden dann die wichtigsten Alternativen zur Behandlung globaler Deadlocks vorgestellt. Einige zusammenfassende Bemerkungen schließen das Kapitel ab.

8.1 Sperrverfahren in Verteilten DBS

Sperrverfahren sind dadurch gekennzeichnet, daß das DBVS vor dem Zugriff auf ein Objekt für die betreffende Transaktion eine Sperre erwirbt, deren Modus dem Zugriffswunsch entspricht. Zur Sicherstellung der Serialisierbarkeit ist dabei i.a. ein sogenanntes *strikt zweiphasiges Sperrprotokoll* erforderlich, bei dem sämtliche Sperren bis zum Transaktionsende gehalten werden (zweite Commit-Phase) [Gr78]. Im einfachsten Fall wird nur zwischen Lese- und Schreibsperren unterschieden (*RX-Sperrverfahren*). Dabei sind Lese- oder R-Sperren (read locks, shared locks) miteinander verträglich, während Schreib- oder X-Sperren (exclusive locks) weder mit sich selbst noch mit Lesesperren kompatibel sind. So können bei gesetzter R-Sperre auf ein Objekt weitere Leseanforderungen gewährt werden, jedoch keine X-Sperren; bei gesetzter X-Sperre sind alle weiteren Sperranforderungen abzulehnen. Ein Sperrkonflikt führt zur Blockierung der Transaktion, de-

ren Sperranforderung den Konflikt verursacht hat; die Aktivierung der wartenden Transaktionen ist möglich, sobald die unverträglichen Sperren freigegeben sind.

Beispiel 8-2

Für den Schedule in Abb. 8-1a tritt mit einem RX-Protokoll ein Sperrkonflikt auf: für die Lesesperre von T2 auf Objekt y ergibt sich ein Konflikt aufgrund der zuvor gewährten X-Sperre für T1. T2 wird nach Beendigung und Freigabe der Sperren von T1 fortgesetzt. Dagegen verursacht der Lesezugriff von T1 auf x keinen Konflikt, da zu diesem Zeitpunkt T3 bereits beendet ist. Als Serialisierungsreihenfolge ergibt sich T3 < T1 < T2.

8.1.1 Zentrales Sperrprotokoll

Ein naheliegender Ansatz zur Synchronisation in Verteilten DBS liegt darin, sämtliche Sperranforderungen und -freigaben auf einem dedizierten Rechner zu bearbeiten. Als Vorteil ergibt sich, daß die Synchronisation quasi wie in einem zentralisierten DBS abgewickelt werden kann, da im zentralen Knoten stets der aktuelle Synchronisationszustand bekannt ist. Insbesondere kann auch eine Deadlock-Erkennung wie im Ein-Rechner-Fall vorgenommen werden.

Allerdings sprechen eine Reihe schwerwiegender Nachteile gegen einen solchen Ansatz, so daß er für Verteilte DBS als ungeeignet anzusehen ist:

- Jede Sperranforderung einer Transaktion, die nicht auf dem zentralen Knoten läuft, verursacht eine Nachricht, auf die die Transaktion synchron warten muß. Eine solche Nachrichtenhäufigkeit ist für Durchsatz und Antwortzeit gleichermaßen inakzeptabel.
- Der zentrale Knoten stellt einen Engpaß für Leistung und Verfügbarkeit ("single point of failure") dar.
- Es wird keine Knotenautonomie unterstützt.

8.1.2 Verteilte Sperrverfahren

Eine bessere Alternative stellen verteilte Sperrverfahren dar. Hierbei synchronisiert jeder Rechner alle Zugriffe auf die Daten der ihm zugeordneten Datenpartition. Diese lokale Sperrbehandlung erfordert keinerlei zusätzliche Kommunikation (bei fehlender Datenreplikation), da die verteilte Ausführung von Transaktionen und Operationen bereits auf die Datenverteilung abgestimmt wird. Kommunikation fällt also zum Starten der Sub-Transaktionen an; für die Datenzugriffe während der Sub-Transaktionen werden die benötigten Sperren lokal angefordert. Das Freigeben der Sperren erfolgt im Rahmen des Commit-Protokolls (Kap. 7.2). Das größte Problem verteilter Sperrverfahren ist die Behandlung globaler Deadlocks, die die Leistungsfähigkeit entscheidend beeinflussen kann. Auf die hierzu bestehenden Alternativen gehen wir in Kap. 8.5 ein.

8.2 Zeitmarkenverfahren

Bei diesen Verfahren wird die Serialisierbarkeit durch Zeitstempel bzw. -marken an den Datenobjekten überprüft. Die Überprüfungen werden stets an den Speicherungsorten der Daten vorgenommen, so daß sich ein inhärent verteiltes Protokoll ergibt. Weiterhin entfallen eigene Kommunikationsvorgänge zur Synchronisation, ähnlich wie bei verteilten Sperrverfahren. Ein Hauptvorteil gegenüber Sperrverfahren liegt darin, daß keine Deadlocks vorkommen können.

Wir beschränken uns hier auf das einfachste Zeitmarkenverfahren (*Basic Timestamp Ordering* [BHG87]). Dabei bekommt jede Transaktion T bei ihrem BOT eine global eindeutige Zeitmarke *ts (T)* fest zugeordnet. Zur Sicherstellung der globalen Eindeutigkeit dieser Zeitmarken kommen mehrere Methoden in Betracht. Zum Beispiel kann ein dedizierter Knoten (timestamp server) vorgesehen werden, an dem die Zeitstempel verwaltet und angefordert werden. Dieser Ansatz verursacht jedoch Kommunikationverzögerungen und beeinträchtigt die Knotenautonomie. Eine bessere Alternative bildet die Verwendung zweiteiliger Zeitstempel bestehend aus lokaler Uhrzeit und Rechner-ID [La78]. Damit wird die globale Ordnung primär über die lokalen Uhrzeiten festgelegt. Die Rechner-ID wird nur für Transaktionen (verschiedener Rechner) mit übereinstimmender lokaler Uhrzeit benötigt, um eine vollständige Ordnung zu erreichen und die globale Eindeutigkeit sicherzustellen. Der Hauptvorteil eines solchen Ansatzes liegt darin, daß die Zeitstempel an jedem Knoten lokal vergeben werden können. Allerdings ist ohne Synchronisierung der lokalen Uhren [Cr89] nicht gewährleistet, daß die Transaktions-Zeitstempel monoton wachsen. Es kann also sein, daß nach Beendigung einer Transaktion T2 an einem anderen Knoten eine Transaktion mit kleinerem Zeitstempel T1 gestartet wird.

Bei Zeitmarkenverfahren ist die Position einer Transaktion in der Serialisierungsreihenfolge bereits a priori durch ihre BOT-Zeitmarke festgelegt. Konfliktoperationen verschiedener Transaktionen müssen daher stets in der Reihenfolge der Transaktions-Zeitmarken erfolgen. Dies erfordert, daß eine Transaktion alle Änderungen von "älteren" Transaktionen (d.h. Transaktionen mit kleinerem Zeitstempel) sehen muß, jedoch keine Änderungen von "jüngeren" Transaktionen sehen darf. Werden diese Bedingungen verletzt, wird die betroffene Transaktion zurückgesetzt und mit einem neuen Zeitstempel wiederholt.

Die Überprüfung dieser sehr restriktiven Forderungen geschieht mittels Zeitmarken an den Datenobjekten, wobei für jedes Objekt ein Schreib- und ein Lesezeitstempel geführt wird. Der Schreibzeitstempel (write time stamp) *WTS* bzw. der Lesezeitstempel (read time stamp) *RTS* entspricht dabei der Transaktions-Zeitmarke derjenigen Transaktion, die das Objekt zuletzt geändert bzw. gelesen hat. Ein Lesezugriff einer Transaktion T mit Zeitstempel *ts (T)* auf ein Objekt x ist nicht zulässig, wenn gilt: $ts(T) < WTS(x)$.

Das bedeutet, daß keine jüngere Transaktion als T das Objekt zuletzt geändert haben darf. Analog muß für einen Schreibzugriff geprüft werden, daß keine jüngere Transaktion das Objekt bereits geändert oder gelesen hat, es darf also nicht gelten:

$$ts(T) < \text{Max}(RTS(x), WTS(x)).$$

Liegt eine dieser Bedingungen vor, erfolgt die Rücksetzung der zugreifenden Transaktion T.

Beispiel 8-3

Für den Schedule in Abb. 8-1a erfolgen die Zugriffe auf Objekt y in der BOT-Reihenfolge der beiden Transaktionen T1 und T2, so daß kein Konflikt vorliegt. Dagegen greift T1 erst nach der jüngeren Transaktion T3 auf Objekt x zu, so daß T1 zurückgesetzt wird. Als Serialisierungsreihenfolge ergibt sich T2 < T3.

Die Rücksetzgefahr einer Transaktion steigt mit zunehmender Verweildauer im System, da dann entsprechend mehr jüngere Transaktionen auf die noch benötigten Datenobjekte zugreifen können. Damit besteht vor allem für lange Transaktionen eine hohe Rücksetzwahrscheinlichkeit. Weiterhin kann ein "Verhungern" (starvation) einer Transaktion nicht verhindert werden, da für eine bereits zurückgesetzte Transaktion wiederum eine Rücksetzung notwendig werden kann. Dies kann dazu führen, daß bestimmte Transaktionen u.U. nie zu Ende kommen.

Ein weiterer Nachteil ergibt sich daraus, daß "schmutzige" Änderungen einer Transaktion durch Zusatzmaßnahmen gegenüber anderen Transaktionen zu verbergen sind. Für den Schedule in Abb. 8-1a ist so der Zugriff auf Objekt y durch T2 aufgrund der Zeitmarken zwar zulässig. Jedoch stellt die von T1 vorgenommene Änderung von y zu diesem Zeitpunkt eine schmutzige (vorläufige) Änderung dar, da T1 noch kein Commit erreicht hat. In der Tat wird ja T1 später aufgrund des Zugriffs auf Objekt x noch zurückgesetzt, so daß die von ihr vorgenommene Änderung von y zurückgenommen werden muß. Damit T2 nicht die schmutzige Änderung sieht, muß ihr Zugriff auf y bis zum Ende von T1 blockiert werden.

Das Beispiel verdeutlicht, daß nach einer Änderung wie bei Sperrverfahren alle weiteren Zugriffe (die nicht schon wegen des Zeitstempelvergleichs abgewiesen wurden) bis zum Transaktionsende des Änderers verzögert werden müssen. Damit kann ein ähnlich hohes Ausmaß an Blockierungen wie bei Sperrverfahren eingeführt werden, zusätzlich zu den aufgrund der Zeitmarkenvergleiche eingeführten Rücksetzungen. Deadlocks (zyklische Wartebeziehungen zwischen Transaktionen) sind jedoch nicht möglich, da die Objektzugriffe stets in der Reihenfolge der BOT-Zeitstempel durchgeführt werden.

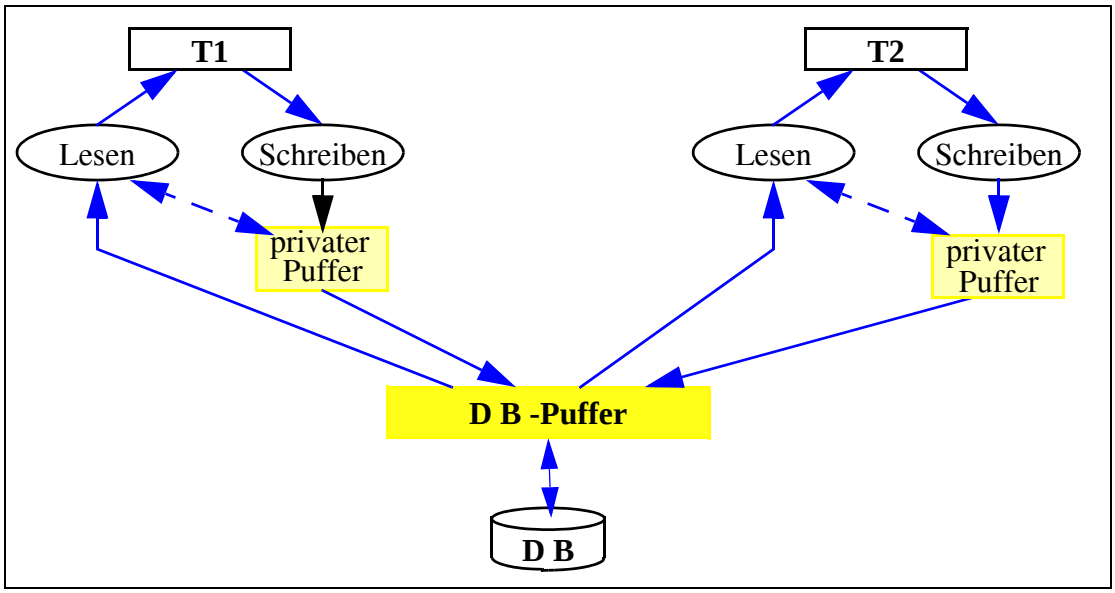
8.3 Optimistische Synchronisation

Optimistische Synchronisationsverfahren gehen von der Annahme aus, daß Konflikte zwischen Transaktionen seltene Ereignisse darstellen und somit das präventive Sperren der Objekte unnötigen Aufwand verursacht [KR81]. Daher greifen diese Verfahren zunächst nicht in den Ablauf einer Transaktion ein, sondern erlauben ein nahezu beliebig paralleles Arbeiten auf der Datenbank. Erst bei Transaktionsende wird überprüft, ob Konflikte mit anderen Transaktionen aufgetreten sind. Gemäß dieser Vorgehensweise unterteilt man die Ausführung einer Transaktion in drei Phasen:

- In der *Lesephase* wird die eigentliche Transaktionsverarbeitung vorgenommen, d.h., es werden Objekte der Datenbank gelesen und modifiziert. Jede Transaktion führt dabei ihre Änderungen auf Kopien in einem ihr zugeordneten *Transaktions-Puffer* durch, der für keine andere Transaktion zugänglich ist.
- Bei EOT wird eine *Validierungsphase* gestartet, in der geprüft wird, ob die beendigungswillige Transaktion mit einer parallel zu ihr laufenden Transaktion in Konflikt geraten ist. Im Gegensatz zu Sperrverfahren, bei denen Blockierungen das primäre Mittel zur Behandlung von Synchronisationskonflikten sind, werden hier Konflikte stets durch Zurücksetzen einer oder mehrerer beteiligter Transaktionen aufgelöst. Es ist so zwar mit mehr Rücksetzungen als bei Sperrverfahren zu rechnen, dafür können aber bei optimistischen Verfahren keine Deadlocks entstehen. Dies ist vor allem für Verteilte DBS ein potentieller Vorteil.
- Die *Schreibphase* wird nur von Änderungstransaktionen ausgeführt, die die Validierungsphase erfolgreich beenden konnten. In dieser Phase wird zuerst die Wiederholbarkeit der Transaktion sichergestellt (Logging), bevor alle Änderungen durch Einbringen in die Datenbank für andere Transaktionen sichtbar gemacht werden.

Abb. 8-3 veranschaulicht die Verwendung von privaten Transaktions-Puffern sowie dem DB-Puffer (Systempuffer), der für alle Transaktionen zugänglich ist. Für Lesezugriffe ist zunächst zu überprüfen, ob das entsprechende Objekt (aufgrund einer vorherigen Änderung) im privaten Transaktions-Puffer vorliegt. Ist dies nicht der Fall, erfolgt der Zugriff zum DB-Puffer; liegt die betreffende Seite auch dort nicht vor, ist sie vom Externspeicher einzulesen. In der Schreibphase werden die Änderungen vom Transaktions-Puffer in den DB-Puffer gebracht, womit sie allen Transaktionen zugänglich werden.

Abb. 8-3: Pufferorganisation bei optimistischer Synchronisation



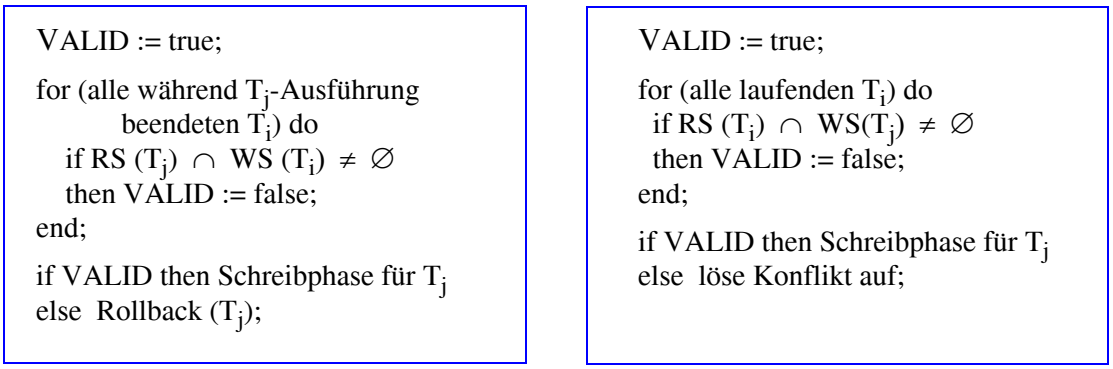
8.3.1 Validierungsansätze

Um die Validierungen durchführen zu können, werden für jede Transaktion T_i während ihrer Lesephase die Namen von ihr gelesener bzw. geänderter Objekte in einem *Read-Set* RS (T_i) bzw. *Write-Set* WS (T_i) geführt. Wir nehmen an, daß vor jeder Änderung das entsprechende Objekt gelesen wird, so daß der Write-Set einer Transaktion stets eine Teilmenge des Read-Sets bildet. Nach [Hä84] lassen sich optimistische Synchronisationsverfahren gemäß ihrer Validierungsstrategie grob in zwei Klassen unterteilen. Bei den rückwärtsorientierten Verfahren (Backward Oriented Optimistic Concurrency Control, *BOCC*) erfolgt die Validierung ausschließlich gegenüber bereits beendeten Transaktionen. Bei den vorwärtsorientierten Verfahren (Forward Oriented Optimistic Concurrency Control, *FOCC*) dagegen wird gegen noch laufende Transaktionen validiert. In beiden Fällen wird durch die Validierung sichergestellt, daß die validierende Transaktion alle Änderungen von zuvor erfolgreich validierten Transaktionen gesehen hat. Damit ist die Serialisierungsreihenfolge durch die Validierungsreihenfolge gegeben.

Im ursprünglichen BOCC-Verfahren nach [KR81] wird bei der Validierung überprüft, ob die validierende Transaktion ein Objekt gelesen hat, das während ihrer Lesephase geändert wurde. Dazu wird in der Validierungsphase der Read-Set der validierenden Transaktion T_j mit den Write-Sets aller Transaktionen T_i verglichen, die während der Lesephase von T_j validiert haben (Abb. 8-4a). Ergibt sich eine Überschneidung mit einem dieser Write-Sets, wird die validierende Transaktion zurückgesetzt, da sie möglicherweise auf veraltete Daten zugegriffen hat (die am Konflikt beteiligten Transaktionen können nicht mehr zurückgesetzt werden, da sie bereits beendet sind). Die Validierungen werden dabei in einem kritischen

Abschnitt durchgeführt, der sicherstellt, daß zu einem Zeitpunkt höchstens eine Validierung vorgenommen wird.

Abb. 8-4: Validierung einer Transaktion T_j bei BOCC und FOCC



a) BOCC b) FOCC

Beispiel 8-4

Für den Schedule in Abb. 8-1a wird mit diesem BOCC-Protokoll zunächst T3 erfolgreich validiert. Bei der nachfolgenden Validierung von T1 wird festgestellt, daß das gelesene Objekt x sich im Write-Set der parallel ausgeführten und bereits validierten Transaktion T3 befindet. Folglich wird T1 zurückgesetzt. Die Validierung von T2 schließlich ist erfolgreich. Es ergibt sich damit folgende Serialisierungsreihenfolge: T3 < T2.

Die skizzierte BOCC-Validierung hat den Nachteil, daß Transaktionen oft unnötigerweise (wegen eines "unechten" Konfliktes) zurückgesetzt werden, obwohl die aktuellen Objektversionen gesehen wurden. Dies ist dann der Fall, wenn auf das von einer parallelen Transaktion geänderte Objekt erst nach dem Einbringen in die Datenbank zugegriffen wurde. So wird in obigem Beispiel (Abb. 8-1a) T1 unnötigerweise zurückgesetzt, weil die Änderung von T3 gesehen wurde. Diese unnötigen Rücksetzungen betreffen v.a. lange Transaktionen, die bereits durch kurze Änderer zum Scheitern gezwungen werden können. Eine Abhilfe des Problems wird jedoch möglich, indem man Änderungszähler oder Versionsnummern an den Objekten führt und eine Rücksetzung nur vornimmt, wenn tatsächlich veraltete Daten gelesen wurden (s. Übungsaufgaben).
Ein weiteres Problem ist auch hier die Gefahr des "Verhungerns", daß also Transaktionen bei der Validierung ständig scheitern. Dies ist v.a. für lange Transaktionen zu befürchten, da sie einen großen Read-Set aufweisen und sich gegenüber vielen Transaktionen validieren müssen. Weiterhin verursacht das späte Zurücksetzen am Transaktionsende ein hohes Maß unnötig verrichteter Arbeit.

Diese Probleme werden von *FOCC-Verfahren* zumindest teilweise umgangen. Bei ihnen wird nicht gegen bereits beendete Transaktionen validiert, sondern gegenüber aktiven Transaktionen. In der Validierungsphase, die nur von Änderungstransaktionen durchzuführen ist, wird untersucht, ob eine der in der Lesephase

befindlichen Transaktionen ein Objekt gelesen hat, das die validierende Transaktion zu ändern im Begriff ist (Abb. 8-4b). In diesem Fall muß der Konflikt durch Zurücksetzen einer (oder mehrerer) der beteiligten Transaktionen aufgelöst werden. Anstatt der validierenden Transaktion können also auch die betroffenen laufenden Transaktionen zurückgesetzt werden, um z.B. den Arbeitsverlust zu verringern. Auch das Verhungern von Transaktionen kann durch eine geeignete Bestimmung der "Opfer" verhindert werden (s. Übungsaufgaben). Im Gegensatz zum (ursprünglichen) BOCC-Ansatz führen bei FOCC daneben nur echte Konflikte zu Rücksetzungen.

Beispiel 8-5

Für den Schedule in Abb. 8-1a wird mit dem FOCC-Protokoll zunächst die Änderungstransaktion T3 erfolgreich validiert, da kein Konflikt mit laufenden Transaktionen vorliegt. Somit wird im Gegensatz zu BOCC richtigerweise kein Konflikt mit T1 festgestellt, da T1 erst nach Abschluß von T3 auf (die aktuelle Version von) x zugreift. Bei der Validierung von T1 wird dann jedoch ein Konflikt mit T2 erkannt (Objekt y), der durch Rücksetzen von T1 oder T2 aufgelöst werden kann. Da T1 bereits beendet ist, empfiehlt sich die Rücksetzung von T2. Als Serialisierungreihenfolge ergibt sich T3 < T1.

In Verteilten DBS kommen auch für die optimistischen Synchronisationsverfahren eine zentrale oder eine verteilte Realisierung in Betracht, die im folgenden diskutiert werden.

8.3.2 Zentrale Validierung

Zur Durchführung der Validierungen sendet eine Transaktion (bei globalen Transaktionen die Primär-Transaktion) am Transaktionsende den vollständigen Read- und Write-Set zu einem zentralen Knoten, der für die Durchführung der Validierungen verantwortlich ist. Dieser meldet dann nach der Validierung das Ergebnis zur (Primär-) Transaktion zurück, woraufhin entweder das Zurücksetzen der Transaktion oder die Schreibphase veranlaßt wird. Bei globalen Transaktionen erfolgt die Schreibphase im Rahmen eines verteilten Commit-Protokolls, das auf ändernde Sub-Transaktionen beschränkt werden kann.

Die zentrale Validierung hat neben der Einfachheit den Vorteil, daß zur eigentlichen Synchronisation nur eine Nachricht (die zur Validierung) erforderlich ist, auf die synchron gewartet werden muß. Damit liegt der Kommunikations-Overhead weit unter dem eines zentralen Sperrverfahrens, bei dem jede Sperranforderung eine synchrone Nachricht an den zentralen Lock-Manager verlangt. Jedoch kann zur Validierung nur eine BOCC-artige Synchronisation verwendet werden; die FOCC-Alternative scheitert daran, daß es unmöglich ist, im zentralen Knoten die aktuellen Read-Sets der in den verschiedenen Knoten laufenden Transaktionen zu kennen. Desweiteren bestehen auch hier wieder die Probleme zentralisierter Lösungen vor allem hinsichtlich Verfügbarkeit sowie Knotenautonomie. Insbeson-

dere ist selbst für lokale Transaktionen Kommunikation zur Validierung erforderlich.

8.3.3 Verteilte Validierung

Beim verteilten Validierungsschema wird jede (Sub-) Transaktion an ihrem ausführenden Rechner validiert. Damit wird für rein lokale Transaktionen, die idealerweise den größten Transaktionsanteil ausmachen, keine Interprozessor-Kommunikation erforderlich. Auch für globale Transaktionen verursacht (wie schon bei verteilten Sperrverfahren) die Synchronisation keine zusätzlichen Nachrichten, da sich Validierungen und Schreibphasen im Rahmen eines Zwei-Phasen-Commit-Protokolls durchführen lassen:

- Die PREPARE-Nachricht, welche die Primär-Transaktion an alle Sub-Transaktionen schickt, dient jetzt gleichzeitig als Validierungsaufforderung. Nach einer erfolgreichen lokalen Validierung sichert jede Sub-Transaktion ihre Änderungen (Prepared-Zustand) und schickt eine READY-Nachricht zur Primär-Transaktion. Bei gescheiterter lokaler Validierung dagegen wird eine FAILED-Nachricht zurückgesendet, und die Sub-Transaktion setzt sich zurück.
- Waren alle lokalen Validierungen erfolgreich, so verschickt die Primär-Transaktion (nach Schreiben des Commit-Satzes) eine COMMIT-Nachricht an alle Sub-Transaktionen, die Änderungen vorgenommen haben. Diese führen daraufhin ihre Schreibphasen durch. Verliefen eine oder mehrere der lokalen Validierungen erfolglos, leitet die Primär-Transaktion durch Verschicken von ABORT-Nachrichten das Zurücksetzen der Transaktion ein.

Diese Vorgehensweise reicht jedoch allein nicht aus, um eine korrekte Synchronisation zu gewährleisten. Denn durch die lokalen Validierungen wird zwar die lokale Serialisierbarkeit in jedem Knoten sichergestellt, nicht aber notwendigerweise die globale Serialisierbarkeit, da die lokale Serialisierungsreihenfolge von Sub-Transaktionen zweier globaler Transaktionen auf verschiedenen Rechnern entgegengesetzt sein kann. Zur Behandlung dieses Problems wurden mehrere Alternativen vorgeschlagen [Ra88a], wobei die wohl einfachste Lösung möglich wird, wenn die Validierungen globaler Transaktionen auf allen Knoten in der gleichen Reihenfolge ausgeführt werden. In diesem Fall entspricht die auf allen Rechnern gleiche Validierungsreihenfolge nicht nur der lokalen, sondern zugleich der globalen Serialisierungsreihenfolge.

Um die gleiche Validierungsreihenfolge auf allen Knoten zu erzwingen, können global eindeutige Transaktions-Zeitstempel verwendet werden. Diese Zeitstempel werden (im Gegensatz zu Zeitmarkenverfahren) beim EOT am Heimatknoten zugewiesen und bestimmen die Position der Transaktion in der globalen Serialisierungsreihenfolge. Der Transaktions-Zeitstempel wird bei den Validierungsaufforderungen mitgeschickt, so daß in jedem Knoten die gleiche Validierungsreihenfolge eingehalten werden kann. "Zu spät" eintreffende Transaktionen, deren Zeitstempel kleiner ist als die von zuletzt lokal validierten Transaktionen, werden

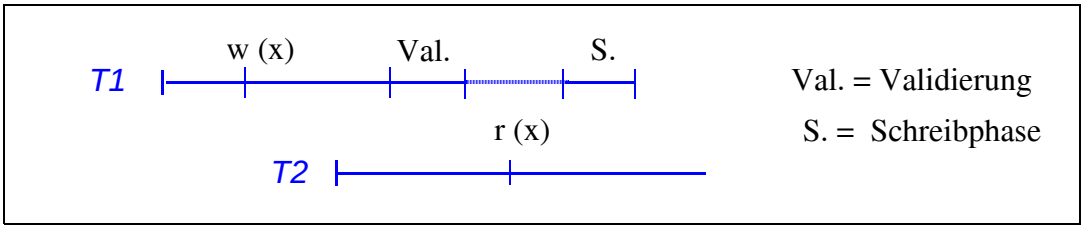
zurückgesetzt. Das Ausmaß solcher Rücksetzungen ist umso geringer, je schneller und gleichmäßiger die Übertragungszeiten zwischen den Rechnern sind und je enger die lokalen Uhren der Rechner synchronisiert sind.

Ein weiteres Problem bei verteilter Validierung entsteht durch die zeitliche Trennung zwischen lokaler Validierung und Schreibphase. Denn im Gegensatz zu zentralisierten DBS ist nun auf einem Knoten nach erfolgreicher lokaler Validierung das Schicksal der globalen Transaktion ungewiß. Die von lokal erfolgreich validierten Sub-Transaktionen beabsichtigten Änderungen sind demnach "unsicher", da sie je nach Validierungsausgang der globalen Transaktion weggeworfen oder in die Datenbank eingebracht werden.

Beispiel 8-6

Im Schedule von Abb. 8-5 will Transaktion T2 auf Objekt x zugreifen, das von der lokal bereits erfolgreich validierten, globalen Änderungstransaktion T1 geändert wurde. Deren Änderung ist jedoch unsicher, da zu diesem Zeitpunkt ihr globales Commit-Ergebnis noch nicht bekannt ist.

Abb. 8-5: Probleme mit "unsicheren" Änderungen



Für die Behandlung dieser *unsicheren Änderungen* in der Lesephase und bei der Validierung lokaler Transaktionen kommen im wesentlichen folgende Alternativen in Betracht [Ra88a]:

1. Bei der ersten Möglichkeit werden unsichere Änderungen nicht beachtet, sondern auf die aktuell gültige, ungeänderte Version zugegriffen. Dies ist allerdings wenig sinnvoll, wenn - wie bisher vorausgesetzt - sich die laufenden Transaktionen in der Serialisierungsreihenfolge nach allen bereits (lokal) validierten Transaktionen einreihen müssen. Denn in diesem Fall wird die auf eine alte Objektversion zugreifende Transaktion zurückgesetzt, wenn die globale Transaktion erfolgreich abschließt, wovon optimistischerweise auszugehen ist. Der Zugriff auf die ungeänderte Objektversion kommt daher allenfalls in Frage, wenn sich die betreffende Transaktion in der Serialisierungsreihenfolge vor der die Änderung beabsichtigende globalen Transaktion einreihen läßt. Die Entscheidung darüber ist im verteilten Fall relativ komplex und soll hier nicht weiter untersucht werden.
2. In einer optimistischeren Alternative wird sofort auf unsichere Änderungen zugegriffen, weil davon ausgegangen wird, daß die globale Transaktion erfolgreich zu Ende kommt. Damit machen die auf unsichere Änderungen zugreifenden Transaktionen ihr Schicksal von dem der globalen Transaktion abhängig. Sinnvollerweise warten dann diese abhängigen Transaktionen vor ihrer lokalen Validierung ab, ob ihr Optimismus berechtigt war, d.h., ob die globalen Transaktionen, deren Änderungen gesehen wurden, erfolgreich waren oder nicht. Würden nämlich diese abhängigen Transaktionen ebenfalls wieder unsichere Änderungen zugänglich machen, könnte eine Rücksetzung einen Domino-Effekt nach sich ziehen.

3. Die dritte Alternative schließlich blockiert den Zugriff auf unsichere Änderungen bis der Ausgang der globalen Transaktion feststeht. Damit werden unnötige Rücksetzungen - wie bei Alternative a) möglich - umgangen, allerdings wird durch das "Sperren" der Objekte auch die Parallelität reduziert. Da die Sperren jedoch nur während der Commit-Behandlung gehalten werden, ist die Wahrscheinlichkeit von Sperrkonflikten entsprechend geringer als bei reinen Sperrverfahren. Deadlocks können dabei nicht entstehen, da die Transaktion, auf die gewartet wird, ihre Lesephase bereits beendet hat und somit selbst nicht mehr auf solche Sperren laufen kann.

Von den genannten Alternativen erscheint der letzte Ansatz am vielversprechendsten. Insbesondere können damit im Falle BOCC-artiger Synchronisation Validierungen und Schreibphasen quasi wie im zentralen Fall erfolgen. Wie in [TR90] beschrieben, können die Sperren weitergehend genutzt werden, um ein mehrfaches Zurücksetzen derselben Transaktion zu verhindern. Dabei werden die von einer Transaktion erworbenen Sperren im Falle einer gescheiterten Validierung beibehalten. Die Sperren sichern zu, daß die erneute Transaktionsausführung erfolgreich ist, sofern keine zusätzlichen Objekte referenziert werden*. Es handelt sich dabei um eine spezielle Kombination von optimistischem Ansatz und Sperrverfahren. Deadlocks können verhindert werden, da die Sperranforderungen in allen Knoten in der Reihenfolge der Transaktions-Zeitstempel bearbeitet werden.

Bei FOCC war im zentralen Fall für Lesetransaktionen keine Validierung erforderlich; mit Erreichen ihres EOT war ihr erfolgreiches Abschneiden gewährleistet. Dies gilt aufgrund der unsicheren Änderungen im verteilten Fall jedoch nicht mehr. Zum einen müssen - wie auch für BOCC - lokale Lesetransaktionen beim Objektzugriff die Sperren globaler Transaktionen beachten. Weiterhin kann es sein, daß eine lesende Sub-Transaktion noch nach ihrer lokalen Beendigung, jedoch vor Abschluß der Gesamttransaktion, zurückgesetzt wird, wenn man im Konfliktfall nicht stets die validierende Transaktion zurücksetzen will. Daher ist auch für eine globale Lesetransaktion eine Validierung notwendig, um sicherzustellen, daß sämtliche Änderungen von zuvor beendeten Transaktionen gesehen wurden.

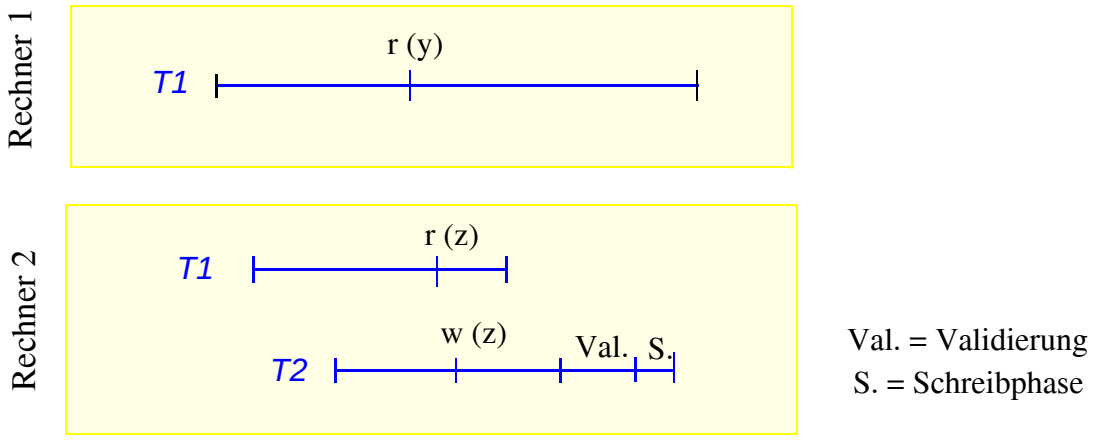
Beispiel 8-7

Im Schedule von Abb. 8-6 führt die globale Lesetransaktion T1 Sub-Transaktionen in Rechner 1 und 2 aus. Die Sub-Transaktion in Rechner 2 ist bei FOCC-Validierung bis zu ihrem Ende in keinen Konflikt verwickelt. Bei der danach stattfindenden Validierung der lokalen Update-Transaktion T2 wird jedoch ein Konflikt für Objekt z entdeckt, da T1 zu diesem Zeitpunkt noch nicht beendet ist. Zur Konfliktbehebung ist entweder T2 oder T1 abzubrechen. Beim EOT der globalen Lesetransaktion T1 ist eine Validierung erforderlich, um festzustellen, ob alle Sub-Transaktionen unbeschadet blieben.

---

* Um dies zu gewährleisten, sind bei der Validierung zusätzlich gesetzte Sperren zu berücksichtigen [TR90].

Abb. 8-6: Probleme mit Lesetransaktionen bei verteilter FOCC-Validierung



8.4 Mehrversionen-Konzept

Mehrversionen-Synchronisationsverfahren (multiversion concurrency control) streben eine Reduzierung an Synchronisationskonflikten an, indem für geänderte Objekte zeitweilig mehrere Versionen geführt werden und Leser ggf. auf ältere Versionen zugreifen. Voraussetzung dabei ist, daß für jede Transaktion vorab Wissen darüber vorliegt, ob sie möglicherweise Änderungen vornimmt. Einer reinen Lesetransaktion T wird dabei während ihrer gesamten Laufzeit eine Sicht auf die Datenbank gewährt, wie sie bei ihrem BOT gültig war; Änderungen, die während ihrer Bearbeitung vorgenommen werden, bleiben für T unsichtbar. Um dies zu realisieren, erzeugt jede erfolgreiche Änderung eine neue Version des modifizierten Objekts; die Versionen werden in einem sogenannten *Versionen-Pool* verwaltet. Im Gegensatz zu Lesetransaktionen greifen Änderungstransaktionen stets auf die aktuelle Version eines Objektes zu. Für ihre Synchronisation kann praktisch jedes der allgemeinen Synchronisationsverfahren verwendet werden [CM86, AS89].

Da mit den Versionen jeder Lesetransaktion der bei BOT gültige (und konsistente) DB-Zustand zur Verfügung gestellt wird, ist für Lesetransaktionen keinerlei Synchronisation mehr erforderlich. Weiterhin brauchen sich andere Transaktionen nicht mehr gegen Lesetransaktionen zu synchronisieren. Damit reduziert sich sowohl die Konfliktwahrscheinlichkeit (und damit die Anzahl von Blockierungen und Rücksetzungen) sowie der Synchronisierungsaufwand (Anzahl von Sperranforderungen, Validierungen etc.). Die Serialisierbarkeit bleibt generell gewahrt.

Beispiel 8-8

Für den Schedule in Abb. 8-1a trat mit einem RX-Protokoll ein Sperrkonflikt auf (Beispiel 8-2). In Kombination mit einem Mehrversionen-Konzept wird der Lesezugriff von T2 auf Objekt y nun ohne Konflikt abgewickelt, da der Zugriff auf die ungeänderte Version erfolgt. Die Serialsierungsreihenfolge lautet jetzt T2 < T3 < T1.

Für die Vorteile, die ein Mehrversionen-Konzept bietet, muß zum einen in Kauf genommen werden, daß Lesetransaktionen (vor allem lange Leser) nicht immer die aktuellen Daten sehen. Zum anderen ist ein erhöhter Speicherplatzbedarf zur Haltung der Versionen sowie ein zusätzlicher Verwaltungsaufwand für den Versionen-Pool erforderlich. Bezüglich der Versionen-Pool-Verwaltung sind vor allem zwei Aufgaben zu behandeln, nämlich Bestimmung der zu lesenden Versionen sowie die Freigabe nicht mehr benötigter Versionen (garbage collection).

Diese Aufgaben lassen sich relativ einfach durch Verwendung von Zeitstempeln lösen. Im zentralen Fall genügt dazu das Führen eines Transaktionszählers *TNC*. Änderungstransaktionen bekommen am Transaktionsende den aktuellen *TNC*-Wert als Commit-Zeitstempel *cts* zugewiesen, anschließend wird *TNC* inkrementiert. Für jede Version eines geänderten Objektes wird ein Schreibzeitstempel *WTS* geführt, der dem Commit-Zeitstempel der ändernden Transaktion entspricht. Für Lesetransaktionen wird dagegen beim Transaktionsbeginn der aktuelle *TNC*-Wert als BOT-Zeitstempel *bts* übernommen. Damit muß einer Lesetransaktion T für den Zugriff auf Objekt x die jüngste Version von x bereitgestellt werden, für die $WTS(x) < bts(T)$ gilt.
Änderungstransaktionen greifen stets auf die aktuellsten Objektversionen zu.

Um feststellen zu können, welche Versionen nicht mehr benötigt werden, wird der BOT-Zeitstempel der ältesten Lesetransaktion *Min-bts* geführt. Eine Version x_i von Objekt x kann gelöscht werden, falls es eine neuere Version x_j gibt, so daß gilt: $WTS(x_i) < WTS(x_j) < Min\text{-}bts$.

Beispiel 8-9

In obigem Beispiel (Schedule von Abb. 8-1a) seien folgende Initialwerte gegeben:
$WTS(x_0) = 0$, $WTS(y_0) = 0$, $TNC = 1$.
Die Lesetransaktion T2 erhält den BOT-Zeitstempel $bts(T2) = 1$; zugleich wird *Min-bts* auf diesen Wert gesetzt. Für den Zugriff auf y wird daher die ungeänderte Version y_0 mit $WTS(y_0) = 0$ ausgewählt. Beim Commit von T3 werden der Commit-Zeitstempel $cts(T3) = 1$ zugewiesen und TNC auf 2 inkrementiert. Für die neue Version von Objekt x (x_1) gilt $WTS(x_1) = 1$. Beim Commit von T1 werden analog *TNC* auf 3 erhöht und eine neue Version y_1 mit $WTS(y_1) = 2$ erzeugt.
Am Ende von T2 wird *Min-bts* angepaßt und überprüft, welche Versionen freigegeben werden können. Da zu diesem Zeitpunkt keine Lesetransaktion mehr läuft, wird *Min-bts* auf den Wert ∞ gesetzt. Die alten Versionen x_0 und y_0 werden freigegeben, da in beiden Fällen neuere Versionen existieren, deren Schreibzeitstempel kleiner als *Min-bts* sind.

Zur effizienten Lokalisierung und Freigabe von Versionen empfiehlt sich die Implementierung des Versionen-Pools in Form eines Ringpuffers, wie in [Ch82, HP87, CM86] näher beschrieben. Leistungsuntersuchungen zeigten, daß mit Mehrversionen-Verfahren die Leistungsfähigkeit in konfliktträchtigen Anwendungen signifikant verbessert werden kann [CM86, HP87]. In mit realen DB-Lasten vorgenommenen Untersuchungen zeigte sich, daß die überwiegende Mehr-

zahl der Objektzugriffe (> 90%) auf die aktuelle Version und die Mehrzahl der restlichen Zugriffe auf die nächstältere Version entfallen [HP87]. Der Umfang des Versionen-Pools kann daher meist klein gehalten werden*. Einige kommerzielle DBS verwenden bereits einen Mehrversionen-Ansatz (Oracle, Rdb, Prime, InterBase); Einzelheiten der Implementierung wurden jedoch nicht veröffentlicht.

Bei der Übertragung der skizzierten Mehrversionen-Synchronisation auf Verteilte DBS ist vor allem zu klären, wie eine korrekte Vergabe von BOT- und Commit-Zeitstempeln gewährleistet werden kann. Eine einfache Lösung dieses Problems besteht darin, den Zähler TNC auf einem dedizierten Knoten zu führen und die BOT- und Commit-Zeitstempel dort anzufordern. Der Commit-Zeitstempel wird dabei erst zugewiesen, nachdem das Durchkommen der Änderungstransaktion sichergestellt ist (nach Phase 1 des Commit-Protokolls). Er wird in der zweiten Commit-Phase allen Sub-Transaktionen zur Vergabe der Versionsnummern für geänderte Objekte mitgeteilt. Am zentralen Knoten können auch für jeden Rechner die BOT-Zeitstempel der ältesten Lesetransaktionen (*Min-bts*) vermerkt werden. Diese Angaben müssen nicht immer aktuell sein, sondern die Rechner können die betreffenden Änderungen asynchron (z.B. gebündelt mit Zeitmarkenanforderungen) mitteilen. Ebenso kann der zentrale Knoten das globale Minimum asynchron an die einzelnen Rechner weiterleiten, damit dort die nicht mehr benötigten Versionen ermittelt werden können.

Eine solche zentralisierte Zeitstempelverwaltung weist jedoch wieder die bekannten Nachteile hinsichtlich Verfügbarkeit und Knotenautonomie auf, zudem entstehen zusätzliche Kommunikationsverzögerungen zur Zeitstempelvergabe. Die lokale Vergabe global eindeutiger Zeitstempel (z.B. bestehend aus lokaler Uhrzeit und Knoten-ID) ist jetzt jedoch problematisch, wenn nur innerhalb eines Rechners - nicht jedoch rechnerübergreifend - monoton wachsende Zeitstempel gewährleistet sind (z.B. aufgrund unsynchronisierter Uhren). Eine Folge davon ist, daß eine Lesetransaktion nicht notwendigerweise alle bei ihrem Start gültigen Änderungen zu sehen bekommt. Dies ist dann möglich, wenn sie einen BOT-Zeitstempel erhält, der kleiner ist als der Commit-Zeitstempel von an anderen Rechnern bereits beendeten Änderern. Noch gravierender ist jedoch, daß eine Änderungstransaktion T1 möglicherweise einen Commit-Zeitstempel erhält, der kleiner ist als der BOT-Zeitstempel einer an einem anderen Knoten zuvor gestarteten Lesetransaktion T2. Wenn T1 ein Objekt ändert, auf das T2 vorher bereits zugegriffen hat, dann ist die Serialisierbarkeit der Transaktionen in der Reihenfolge ihrer Zeitstempel nicht mehr möglich.

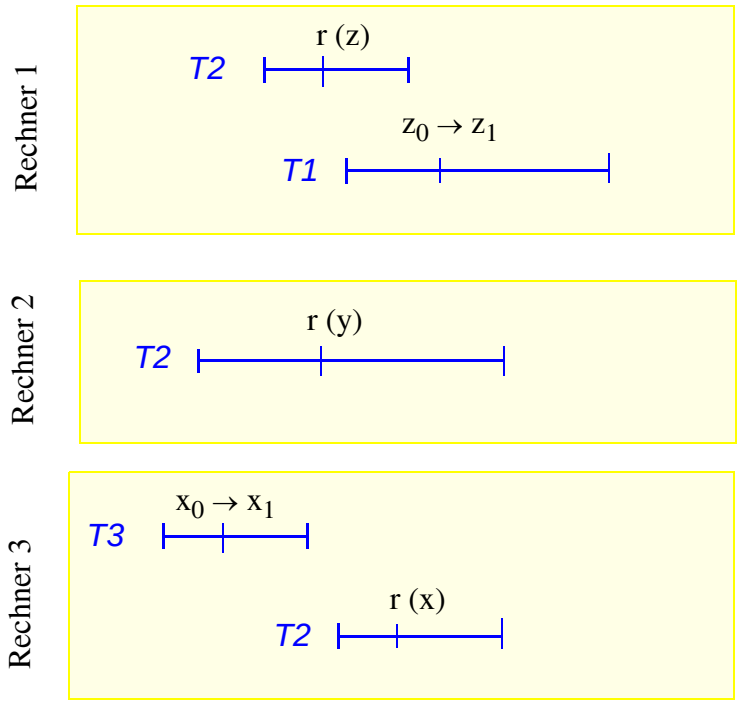
---

* Probleme können durch lange Lesetransaktionen verursacht werden. Kommt es zu einem "Überlauf" des Versionen-Pools sind daher ggf. Rücksetzungen von Lesetransaktionen erforderlich, für die benötigte Versionen nicht mehr weitergeführt werden konnten.

Beispiel 8-10

Im Schedule von Abb. 8-7 werden die globale Lesetransaktion T2 in Rechner 2, die Änderungstransaktionen T1 und T3 in Rechner 1 bzw. Rechner 3 gestartet. Aufgrund unsynchronisierter Zeitstempelvergabe sei *cts (T1)* < *bts (T2)* < *cts (T3)*. Wegen dieser Zeitstempelreihenfolge greift die Sub-Transaktion von T2 in Rechner 3 auf die alte Version x_0 zu, obwohl T3 bereits vor dem Start von T2 die Version x_1 erzeugt hatte. Dies ist zwar unerwünscht, hat jedoch keine Auswirkungen auf die Serialisierbarkeit. Problematisch ist, daß die Sub-Transaktion von T2 in Rechner 1 auf die Objektversion z_0 zugegriffen hat und danach von T1 eine neuere Version z_1 erzeugt wurde, mit $WTS(z_1) = cts(T1) < bts(T2)$. Dies ist nicht zulässig, da die Serialisierungsreihenfolge T1 < T2 aufgrund des vorherigen Zugriffs von T2 auf z_0 nicht mehr eingehalten werden kann.

Abb. 8-7: Probleme mit unsynchronisierter Zeitstempelvergabe



Eine Lösungsmöglichkeit zur Wahrung der Serialisierbarkeit besteht darin, Änderungstransaktionen abzubrechen, falls sie Objektversionen erzeugen, die von bereits ausgeführten Lesetransaktionen hätten gesehen werden müssen. Diese Rücksetzungen können jedoch vermieden werden, wenn bei der Vergabe der Commit-Zeitstempel darauf geachtet wird, daß diese größer sind als die BOT-Zeitstempel aller zuvor an dem Rechner ausgeführten (externen) Lesetransaktionen. Weiterhin muß ein lokal vergebener Commit-Zeitstempel größer sein als der Commit-Zeitstempel von externen Änderungstransaktionen, die an dem Knoten eine Sub-Transaktion ausgeführt haben. Dies ist möglich, da deren Commit-Zeitstempel im Rahmen des Commit-Protokolls propagiert wird. Für globale Änderungstransaktionen müssen diese Bedingungen jedoch an allen Knoten, an denen Sub-Transaktionen ausgeführt wurden, gelten. Daher kann beim EOT zunächst lokal nur ein

vorläufiger Commit-Zeitstempel festgelegt werden, der in Phase 1 des Commit-Protokolls an die beteiligten Rechner propagiert wird. Der endgültige Commit-Zeitstempel wird nach Rückmeldung der beteiligten Knoten festgelegt, so daß er größer ist als alle an den involvierten Rechnern bekannten BOT- und Commit-Zeitstempel. Der endgültige Commit-Zeitstempel wird in der zweiten Commit-Phase an die beteiligten Rechner mitgeteilt. Diese Vorgehensweise erreicht eine Synchronisierung der Zeitstempelvergabe ohne zusätzlichen Kommunikationsaufwand. Eine ähnliche Vorgehensweise wird in [BG83, We87] empfohlen; sie kann als Spezialfall der Generierung konsistenter Zeitstempel nach Lamport [La78] aufgefaßt werden.

Eine solche Vergabe von Commit-Zeitstempel hat Auswirkungen auf Lesetransaktionen, da die Auswahl einer Objektversion vom endgültigen Commit-Zeitstempel abhängt. Weiterhin sind die Änderungen einer Transaktion, die noch keinen endgültigen Commit-Zeitstempel besitzt, unsicher, da die Transaktion möglicherweise noch abgebrochen wird. Die unsicheren Änderungen einer globalen Änderungstransaktion T mit vorläufigem Commit-Zeitstempel *pcts* sind relevant für Lesetransaktionen T_j, für die gilt:

$$bts(T_j) > pcts(T).$$

Hier empfiehlt es sich wiederum, die Lesezugriffe solange zu blockieren, bis der Transaktionsausgang sowie der endgültige Commit-Zeitstempel für T feststeht.

Zur Freigabe von alten Objektversionen muß der BOT-Zeitstempel der ältesten Lesetransaktion im System bekannt sein. Hierzu ist es bei verteilter Realisierung des Mehrversionen-Ansatzes erforderlich, daß jeder Knoten seinen lokalen *Min-bts*-Wert systemweit propagiert (z.B. gebündelt mit Nachrichten für das Commit-Protokoll). Damit kann in jedem Rechner eine Approximation des globalen Minimums geführt werden.

8.5 Deadlock-Behandlung

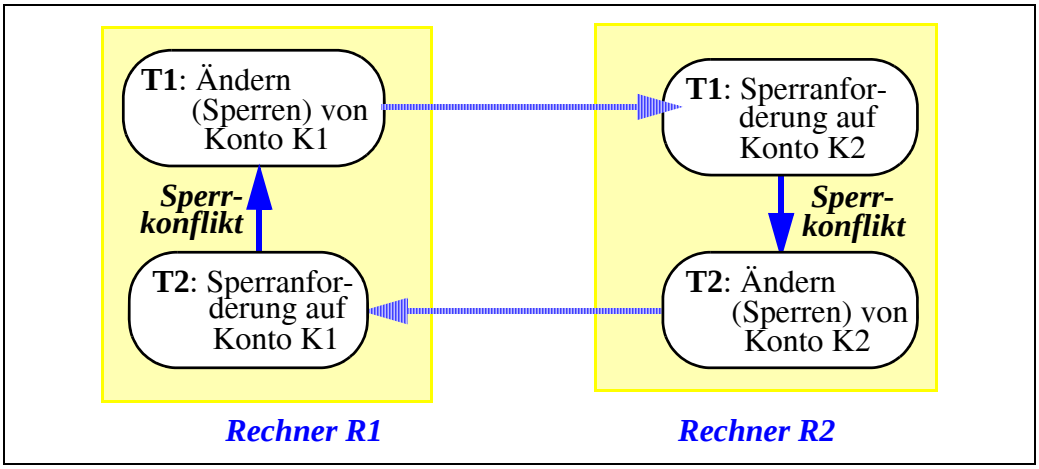
Eine mit Sperrverfahren einhergehende Interferenz ist die Gefahr von Verklemmungen oder Deadlocks, deren charakterisierende Eigenschaft eine zyklische Wartebeziehung zwischen zwei oder mehr Transaktionen ist. Im verteilten Fall können diese Verklemmungen zwischen Transaktionen verschiedener Rechner auftreten, so daß es zu sogenannten *globalen Deadlocks* kommt.

Beispiel 8-11

In einer Bankanwendung kann es zu einem globalen Deadlock zwischen zwei Überweisungstransaktionen kommen, die auf dieselben Konten K1 (an Rechner R1) und K2 (an Rechner R2) zugreifen wollen (Abb. 8-8). Dabei beabsichtigt die in R1 gestartete Transaktion T1 eine Überweisung von K1 nach K2, die Transaktion T2 in R2 eine Überweisung von K2 nach K1. T1 erwirbt zunächst eine Schreibsperre auf K1 und führt die Kontoänderung durch, danach startet sie eine Sub-Transaktion in R2, um auf K2 zuzugreifen. Die entsprechende Sperranforderung führt jedoch zu einem Konflikt, da T2 bereits eine

Schreibsperre auf K2 erworben hat. T2 hat unterdessen eine Sub-Transaktion in R1 gestartet, um Konto K1 zu ändern; diese Sub-Transaktion gerät jedoch in einen Konflikt mit T1. Beide Transaktionen blockieren sich nunmehr gegenseitig; die Situation kann nur durch Rücksetzung einer der Transaktionen behoben werden.

Abb. 8-8: Beispiel eines globalen Deadlocks



Zur Deadlock-Behandlung in zentralisierten sowie in Verteilten DBS kommen einige generelle Strategien in Betracht: Verhütung, Vermeidung, Timeout, Erkennung sowie hybride Strategien. Diese Alternativen werden im folgenden näher diskutiert. Besonderheiten für Verteilte DBS ergeben sich dabei vor allem bezüglich der Deadlock-Erkennung.

8.5.1 Deadlock-Verhütung

Die Deadlock-Verhütung (prevention) ist dadurch gekennzeichnet, daß die Entstehung von Deadlocks verhindert wird, ohne daß dazu irgendwelche Maßnahmen während der Abarbeitung der Transaktionen erforderlich sind. In diese Kategorie fallen v.a. die sogenannten *Preclaiming*-Sperrverfahren, bei denen eine Transaktion alle benötigten Sperren bereits bei BOT anfordern muß. Verklemmungen können dabei umgangen werden, indem jede Transaktion ihre Sperren in einer global festgelegten Reihenfolge anfordert, da dann keine zyklische Wartebeziehungen entstehen können. Die eigentliche Ausführung einer Transaktion kann erst nach Erwerb aller benötigten Sperren beginnen, wobei im Fall von Sperrkonflikten die entsprechenden Wartezeiten in Kauf zu nehmen sind.

Beispiel 8-12

Für den in Abb. 8-8 gezeigten Deadlock (Beispiel 8-11) benötigt Transaktion T1 eine Schreibsperre für die Konten K1 und K2; T2 für K2 und K1. Bei einer festen Reihenfolge für Sperranforderungen ergeben sich somit zum Beginn beider Transaktionen die gleichen Sperranforderungen, z.B. zunächst für K1 und dann für K2. Diejenige Transaktion, deren

Sperranforderung für K1 zuerst bearbeitet wird, kann ohne Behinderung auch die K2-Sperre erwerben. Die zweite Transaktion läuft in einen Sperrkonflikt für K1 und kann erst nach Beendigung der ersten Transaktion mit der Sperrbearbeitung und der eigentlichen Ausführung fortfahren. Dafür wird ein Deadlock umgangen (keine Rücksetzung).

Ein wesentliches Problem der Deadlock-Verhütung ist, daß bei Beginn einer Transaktion i.a. nur Obermengen der zu referenzierenden Objekte (z.B. ganze Relationen) bekannt sind. Der somit verursachte Grad an Sperrkonflikten ist daher in der Regel inakzeptabel. Dies gilt umso mehr, da die Sperren während der gesamten Transaktionslaufzeit zu halten sind. Im verteilten Fall kommt als weiterer Nachteil hinzu, daß für nicht lokal verwaltete Objekte vor Transaktionsbeginn eigene Kommunikationsvorgänge für die Sperranforderungen erforderlich sind.

Wegen dieser Schwächen hat der Preclaiming-Ansatz keine praktische Relevanz für Datenbanksysteme. Wir gehen daher im weiteren bei Sperrverfahren davon aus, daß die Sperren während der Transaktionsverarbeitung (unmittelbar vor einem Objektzugriff) angefordert werden.

8.5.2 Deadlock-Vermeidung

Bei der Deadlock-Vermeidung (avoidance) werden potentielle Deadlocks im voraus erkannt und durch entsprechende Maßnahmen vermieden; im Gegensatz zur Verhütung ist also eine Laufzeitunterstützung zur Deadlock-Behandlung erforderlich. Die Vermeidung der Deadlocks erfolgt generell durch Zurücksetzen von möglicherweise betroffenen Transaktionen. Ein einfacher Vermeidungsansatz wäre, im Falle eines Sperrkonfliktes die in Konflikt geratene Transaktion bzw. den Sperrbesitzer abzubrechen, so daß keine Blockierungen auftreten. Dieser Ansatz scheidet jedoch aufgrund der hohen Anzahl von Rücksetzungen aus. Wir betrachten stattdessen Alternativen, welche weniger Rücksetzungen verursachen und Zeitmarken bzw. Zeitintervalle zur Deadlock-Vermeidung verwenden.

Wait/Die und Wound/Wait

Zwei einfache Verfahren zur Deadlock-Vermeidung sind Wait/Die und Wound/Wait [RSL78]. Bei ihnen wird wie für Zeitmarkenverfahren (Kap. 8.2) jeder Transaktion T bei BOT eine global eindeutige Zeitmarke *ts (T)* zugewiesen, wobei hier die Kombination aus lokaler Uhrzeit und Knoten-ID ausreicht. Im Gegensatz zu den Zeitmarkenverfahren erfolgt jetzt die Synchronisation jedoch mit einem Sperrprotokoll. Die Transaktions-Zeitmarken werden lediglich im Falle eines Sperrkonfliktes herangezogen, um Deadlocks zu vermeiden. Dies kann erreicht werden, wenn vorgeschrieben wird, daß bei einem Sperrkonflikt stets nur die ältere (bzw. die jüngere) der beteiligten Transaktionen warten darf und anderenfalls der Konflikt durch Rücksetzung einer Transaktion aufgelöst wird. Denn dann kann sich keine zyklische Wartebeziehung mehr bilden.

Die bei Wait/Die bzw. Wound/Wait verwendete Behandlung von Sperrkonflikten zeigt Abb. 8-9. Im *Wait/Die-Verfahren* (Abb. 8-9a) darf die die Sperre anfordernde Transaktion T_i nur dann warten, wenn sie älter ist als die Transaktion T_j, welche die Sperre besitzt*. Ist dagegen T_i jünger als T_j, so muß T_i "sterben" (*die*), d.h., sie wird zurückgesetzt. Damit warten bei diesem Ansatz stets ältere Transaktionen auf jüngere, jedoch nicht umgekehrt. Im Wound/Wait-Verfahren (Abb. 8-9b) dagegen warten stets jüngere Transaktionen auf ältere. Ist die anfordernde Transaktion T_i älter als der Sperrbesitzer T_j, so wird jedoch nicht die anfordernde Transaktion, sondern der Sperrbesitzer abgebrochen ("verwundet", *wound*). Es handelt sich dabei also um einen preemptiven Ansatz**. Beide Strategien vermeiden Deadlocks ohne zusätzliche Kommunikationsvorgänge, da die Zeitstempel lokal zugewiesen und die Zeitstempelvergleiche im Konfliktfall lokal durchgeführt werden können. Es kann allerdings zu Rücksetzungen kommen, ohne daß ein Deadlock vorliegt.

Abb. 8-9: Konfliktbehandlung bei Wait/Die und Wound/Wait
(Sperranforderung von T_i gerät in Konflikt mit T_j)

```
if ts (Ti) < ts (Tj)  { Ti älter als Tj }
then  WAIT (Ti)
else Rollback  (Ti)   { "Die" }
```

a) Wait/Die

```
if ts (Ti) < ts (Tj)  { Ti älter als Tj }
then  Rollback (Tj)  { "Wound" }
else WAIT  (Ti)
```

b) Wound/Wait

Beispiel 8-13

Für den Schedule in Abb. 8-1a tritt mit einem RX-Protokoll ein Sperrkonflikt auf (Beispiel 8-2), es liegt jedoch kein Deadlock vor. Für Wait/Die wird der Sperrkonflikt dennoch durch Rücksetzung von T2 aufgelöst, da T2 jünger als T1 ist. Für Wound/Wait dagegen wartet T2 auf die Sperrfreigabe durch T1.

Beispiel 8-14

Für den in Abb. 8-8 gezeigten Deadlock (Beispiel 8-11) sei T1 älter als T2, d.h. *ts (T1) < ts (T2)*. Bei Wait/Die wartet T1 beim Sperrkonflikt in Rechner R2, da sie älter als T2 ist. Dagegen muß T2 beim Sperrkonflikt in R1 sterben. Bei Wound/Wait wird beim Sperrkonflikt von T1 in R2 der jüngere Sperrbesitzer T2 verwundet, also zurückgesetzt. In beiden Fällen wird der Deadlock also durch Rücksetzen der jüngeren Transaktion T2 aufgelöst.

Bei Wait/Die und Wound/Wait werden im Gegensatz zu den Zeitmarkenverfahren (Kap. 8.2) ältere Transaktionen bevorzugt, da im Falle einer Rücksetzung stets die

---

* Sollten mehrere Transaktionen eine inkompatible Sperre halten, müssen diese alle jünger als die anfordernde Transaktion sein.

** Bei Wound/Wait kann auf das Zurücksetzen des Sperrbesitzers verzichtet werden, wenn (lokal) erkannt wird, daß diese Transaktion nicht blockiert ist, da dann kein Deadlock möglich ist. Insbesondere kann das Zurücksetzen entfallen, wenn die Transaktion sich bereits in der Commit-Behandlung befindet und die Sperren daher ohnehin bald freigibt.

jüngere der am Konflikt beteiligten Transaktionen zurückgesetzt wird. Daher empfiehlt es sich generell, bei Wait/Die und Wound/Wait nach einer Rücksetzung die ursprüngliche Zeitmarke beizubehalten, um die Wahrscheinlichkeit weiterer Rücksetzungen zu begrenzen. Die Bevorzugung älterer Transaktionen ist besonders ausgeprägt bei Wound/Wait, wo sogar der Sperrbesitzer abgebrochen wird, falls er jünger ist als die anfordernde Transaktion. Bei Wait/Die ist das Durchkommen älterer Transaktionen auch gesichert, jedoch nimmt mit zunehmendem Alter die Wahrscheinlichkeit von Blockierungen zu. Bei Wait/Die sollte eine zurückgesetzte Transaktion T2 auch erst dann wieder gestartet werden, wenn die in Konflikt stehende, ältere Transaktion T1 beendet ist. Denn anderenfalls würde beim erneuten Zugriff auf das betreffende Objekt wiederum ein Abbruch von T2 erfolgen. Diese Gefahr besteht bei Wound/Wait nicht, da hier der (jüngere) Sperrbesitzer zurückgesetzt wird. Bei einem erneuten Zugriff auf das Objekt während der Wiederausführung erfolgt für die jüngere, anfordernde Transaktion jedoch keine Rücksetzung, sondern sie kann auf die Sperrfreigabe warten.
In der Simulationsstudie [ACM87] zeigte Wound/Wait ein konsistent besseres Leistungsverhalten als Wait/Die.

Dynamische Zeitmarken und Zeitintervalle

Wait/Die und Wound/Wait basieren beide auf statischen Zeitmarken, da sie während der Ausführung der Transaktion nicht mehr geändert werden. In [BEHR82] wurde die Verwendung *dynamischer Zeitmarken* vorgeschlagen, wobei Transaktionen zunächst ohne Zeitmarke gestartet werden. Erst wenn der erste Konflikt mit einer anderen Transaktion erfolgt, wird eine Transaktions-Zeitmarke bestimmt. Dabei kann stets eine Zeitmarke gewählt werden, so daß der Konflikt ohne Rücksetzung überstanden wird. Damit überlebt eine Transaktion zumindest den ersten Sperrkonflikt, womit die Anzahl von Rücksetzungen gegenüber der Verwendung statischer Zeitmarken reduziert wird.

Beispiel 8-15

Für den Schedule in Abb. 8-1a verursachte Wait/Die die Rücksetzung von T2, obwohl kein Deadlock vorlag (Beispiel 8-13). Diese Rücksetzung kann mit dynamischen Zeitmarken vermieden werden, da die Zuordnung der Zeitmarken für T1 sowie T2 erst vorgenommen wird, wenn es aufgrund des Lesezugriffs von T2 zum Sperrkonflikt mit T1 kommt. Im Falle von Wait/Die weist man T2 zu diesem Zeitpunkt eine kleinere Zeitmarke als T1 zu, so daß T2 auf die Sperre warten kann, anstatt zurückgesetzt zu werden.

Eine Verallgemeinerung der Nutzung dynamischer Zeitmarken stellt die Verwendung von Zeitintervallen dar [BEHR82]. Dabei wird jeder Transaktion anstelle einer Zeitmarke ein Zeitintervall zugeordnet, das die mögliche Position der Transaktion in der Serialisierungsreihenfolge eingrenzt. Das zunächst unendliche Zeitintervall wird bei jedem Sperrkonflikt dynamisch verkleinert. Ein leeres Zeitintervall, das eine Rücksetzung erzwingt, ergibt sich für eine Transaktion erst bei

einem Sperrkonflikt mit einer Transaktion mit disjunktem Zeitintervall, welches in falscher zeitlicher Relation zum eigenen Intervall steht. Durch geschickte Wahl der Zeitintervalle kann das Ausmaß unnötiger Rücksetzungen jedoch i.a. geringer als mit statischen oder dynamischen Zeitintervallen gehalten werden [KP85, NW87]. Dafür ergibt sich eine höhere Komplexität der Realisierung.

8.5.3 Timeout-Verfahren

Bei diesem sehr einfachen und billigen Verfahren wird eine Transaktion zurückgesetzt, sobald ihre Wartezeit auf eine Sperre eine festgelegte Zeitschranke (Timeout) überschreitet. Da ein Deadlock von alleine nicht verschwindet, wird irgendwann der Timeout überschritten, so daß jeder Deadlock aufgelöst wird. Das Hauptproblem mit diesem Ansatz ist die geeignete Wahl des Timeout-Wertes. Wird er hoch angesetzt, werden Deadlocks erst nach längerer Zeit aufgelöst, so daß die betroffenen Transaktionen unnötig lange blockiert sind. Ein kleiner Wert dagegen kann zu einer hohen Anzahl unnötiger Rücksetzungen von Transaktionen führen, ohne daß also ein Deadlock vorliegt. Ein akzeptabler Mittelwert hängt von vielen sich ändernden Faktoren ab wie der Lastzusammensetzung, Konfliktwahrscheinlichkeit, Verfügbarkeit und Auslastung der Rechner, Kommunikationsverbindungen etc..

Mehrere Leistungsstudien zeigten, daß der Timeout-Ansatz nur im Falle geringer Konflikthäufigkeit hinreichend stabil ist, ansonsten jedoch eine sehr hohe Anzahl von Rücksetzungen verursacht [ACM87, JTK89, Ra93d]. Dennoch wird der Timeout-Ansatz in vielen kommerziellen DBS eingesetzt, z.B. in Tandem NonStop SQL [Ta89].

8.5.4 Deadlock-Erkennung

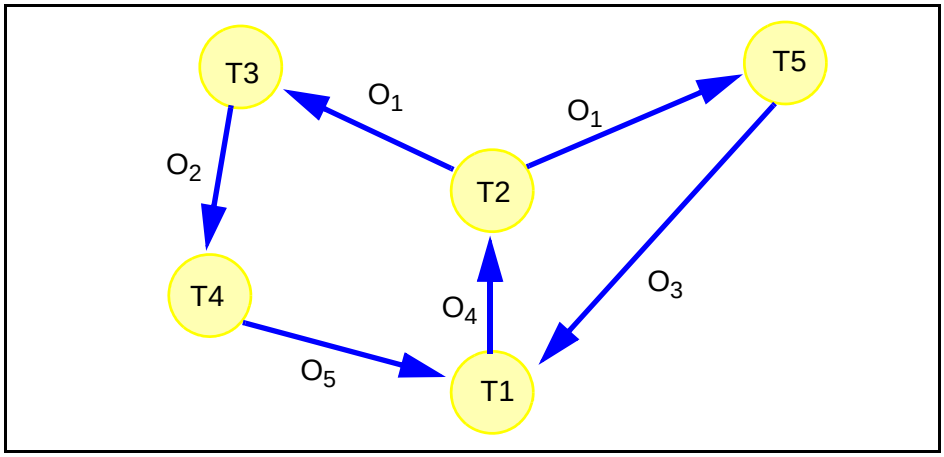
Bei der Deadlock-Erkennung (detection) werden sämtliche Wartebeziehungen aktiver Transaktionen explizit in einem *Wartegraphen* (wait-for graph) protokolliert und Verklemmungen durch Zyklensuche in diesem Graphen erkannt. Im Gegensatz zum Serialisierungs- oder Abhängigkeitsgraphen sind im Wartegraphen nur aktive Transaktionen berücksichtigt, die an einem Sperrkonflikt beteiligt sind. Die Auflösung eines Deadlocks geschieht durch Rücksetzung einer oder mehrerer der am Zyklus beteiligten Transaktionen. Die Deadlock-Erkennung ist zwar im Vergleich zu Vermeidungs- oder Timeout-Strategien am aufwendigsten, dafür kommt sie mit den wenigsten Rücksetzungen aus. Dies bewirkte in quantitativen Untersuchungen [ACM87] die beste Leistungsfähigkeit für zentralisierte DBS, wo die Deadlock-Erkennung auch relativ einfach realisiert werden kann [ACD83, Ji88].

Beispiel 8-16

Abb. 8-10 zeigt ein Beispiel eines Wartegraphen. Die Kanten zwischen den Transaktionen sind mit dem Objekt gekennzeichnet, für das der Konflikt auftrat. Für die Sperranforde-

rung von T2 auf Objekt O_1 ist ein Konflikt mit zwei Transaktionen entstanden (die z.B. beide eine Lesesperren halten). In dem gezeigten Szenario liegen zwei Deadlocks vor: T1 -> T2 -> T3 -> T4 -> T1 sowie T1 -> T2 -> T5 -> T1. Die Rücksetzung von T2 würde beide Deadlocks auflösen.

Abb. 8-10: Beispiel eines Wartegraphen



Die Zyklensuche kann bei jedem Sperrkonflikt vorgenommen werden (continous deadlock detection) oder in periodischen Zeitabständen. Im ersteren Fall wird ein Deadlock zum frühestmöglichen Zeitpunkt aufgelöst; zudem vereinfacht sich die Suche, da i.a. nur ein Teil des Graphen zu berücksichtigen ist. Bei der periodischen Zyklensuche ist stets der ganze Graph zu durchsuchen, dafür kann der Aufwand durch eine seltenere Ausführung begrenzt werden. Allerdings ergibt sich dann wieder eine verzögerte Auflösung von Deadlocks ähnlich wie bei einem Timeout-Ansatz. Bei der Auswahl der "Opfer" können verschiedene Kriterien herangezogen werden, z.B. Minimierung des Arbeitsverlustes oder Einfachheit der Opferbestimmung. Wird bei einem Sperrkonflikt jeweils sofort eine Deadlock-Erkennung vorgenommen, kann ein neuer Deadlock einfach durch Rücksetzung der in den Konflikt geratenen Transaktion umgangen werden. Dies ist im verteilten Fall jedoch nicht mehr möglich.

Die Deadlock-Erkennung in Verteilten DBS ist weitaus aufwendiger als in zentralisierten DBS, da Kommunikationsvorgänge zur Mitteilung von Wartebeziehungen notwendig werden. Zudem ist es schwierig, überhaupt eine korrekte Deadlock-Erkennung zu realisieren, da jeder Knoten nur bezüglich lokaler Transaktionen die aktuelle Wartesituation kennt. Warteinformationen bezüglich externer Transaktionen sind dagegen aufgrund zeitlicher Verzögerungen bei der Übertragung i.a. veraltet. So konnte für viele Verfahrensvorschläge gezeigt werden, daß sie "falsche" Deadlocks (Phantom-Deadlocks) erkennen und somit Transaktionen zurücksetzen, ohne daß ein Deadlock vorliegt [BHG87]. Vielfach werden auch globale Deadlocks mehrfach erkannt, was ebenfalls zu unnötigen Rücksetzungen führt.

Im folgenden diskutieren wir verschiedene Ansätze zur globalen Deadlock-Erkennung, die entweder zentral oder verteilt erfolgen kann.

Zentrale Deadlock-Erkennung

Hierbei wird in einem zentralen Knoten ein globaler Wartegraph geführt und auf Zyklen durchsucht. Um den Kommunikations-Overhead einzugrenzen, schickt jeder Rechner die bei ihm entstehenden Wartebeziehungen in periodischen Zeitabständen an den zentralen Knoten (Nachrichtenbündelung). Dieser vervollständigt damit seinen Wartegraph und startet die Zyklensuche. Da der globale Wartegraph wegen der Nachrichtenverzögerungen i.a. nicht auf dem aktuellsten Stand ist, werden Deadlocks nicht nur verspätet entdeckt, es können auch Phantom-Deadlocks erkannt und damit unnötige Rücksetzungen verursacht werden. Die Sonderrolle des für die Deadlock-Erkennung zuständigen Rechners bewirkt zudem ähnliche Probleme bezüglich Verfügbarkeit und Rechnerautonomie wie bei einem zentralen Sperrverfahren. Nach einem Ausfall des zentralen Knotens könnte jedoch leicht ein anderer dessen Funktion übernehmen bzw. auf ein Timeout-Verfahren umgestellt werden.

Eine zentrale Deadlock-Erkennung wurde in Distributed Ingres vorgenommen [St79].

Verteilte Deadlock-Erkennung

Eine verteilte Deadlock-Erkennung vermeidet die Abhängigkeiten zu einem zentralen Knoten, ist dafür jedoch wesentlich schwieriger zu realisieren. Insbesondere kann es zur Mehrfacherkennung globaler Deadlocks kommen und Warteinformationen sind ggf. über mehrere Rechner zu propagieren, bis ein globaler Deadlock erkannt wird. Ein Überblick zu den zahlreichen Verfahrensvorschlägen findet sich in [El86, Kn87]. Wir beschränken uns hier auf die Beschreibung des im System R* implementierten Verfahrens von Obermarck [Ob82, MLO86]. Es hat den großen Vorteil, daß lediglich eine Nachricht benötigt wird, um globale Deadlocks aufzulösen, an denen zwei Rechner beteiligt sind. Dies ist der wichtigste Typ von globalen Deadlocks, da typischerweise die meisten Deadlocks (> 90%) nur Zykluslänge 2 aufweisen [GHOK81]. Eine zentrale Deadlock-Erkennung hätte bereits zur Übertragung der Wartebeziehungen einen höheren Aufwand erfordert.

Das Obermarck-Verfahren geht davon aus, daß in jedem Rechner ein spezieller Prozeß ("deadlock detector") mit der Deadlock-Erkennung beauftragt ist und mit den entsprechenden Prozessen der anderen Rechner kommuniziert. Jeder dieser Prozesse führt einen lokalen Wartegraphen, in dem sämtliche Wartebeziehungen lokaler und externer Transaktionen hinsichtlich der an diesem Rechner verwalteten Objekte geführt werden. Damit lassen sich lokale Deadlocks sofort und ohne Kommunikation erkennen. Im Wartegraph wird ein spezieller Knoten EXTERNAL geführt, der Wartebeziehungen zu Sub-Transaktionen auf anderen Rechnern kennzeichnet. Für eine externe Sub-Transaktion T_j wird stets eine Wartebeziehung EXTERNAL -> T_j aufgenommen, da möglicherweise an anderen Rechnern auf Sperren von T_j gewartet wird. Eine Wartebeziehung T_i -> EXTERNAL wird

ferner eingeführt, sobald Transaktion T_i eine externe Sub-Transaktion startet, da diese möglicherweise in einen Konflikt gerät. Ein *potentieller* globaler Deadlock wird durch einen Zyklus im lokalen Wartegraphen angezeigt, an dem der EXTERNAL-Knoten beteiligt ist:

$$\text{EXTERNAL} \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow \text{EXTERNAL}$$

Um festzustellen, ob tatsächlich ein globaler Deadlock vorliegt, wird die Warteinformation an den Rechner weitergeleitet, an dem Transaktion T_n die Sub-Transaktion gestartet hatte. Nach Empfang solcher Warteinformationen vervollständigt der zuständige Prozeß seinen Wartegraphen und führt darauf eine Zyklensuche durch. Zur Erkennung eines globalen Deadlocks kann dazu erneut die Weiterleitung der erweiterten Warteinformation an einen anderen Rechner erforderlich sein. Wird ein vollständiger Zyklus festgestellt, erfolgt die Rücksetzung einer der beteiligten Transaktionen. Dabei sollte, wenn möglich, eine lokale Transaktion ausgewählt werden, um den Deadlock möglichst schnell zu beheben [MLO86].

Ein Problem mit dem skizzierten Verfahren liegt darin, daß sich im Falle eines globalen Deadlocks an jedem der beteiligten Rechner ein Zyklus mit dem EXTERNAL-Knoten ergibt. Wenn jetzt jeder dieser Rechner seine Warteinformation zur globalen Deadlock-Erkennung wie oben beschrieben weitergibt, führt dies jedoch zu einem unnötig hohen Kommunikationsaufwand sowie zur mehrfachen Erkennung eines globalen Deadlocks. Zur Abschwächung dieses Problems wird jeder Transaktion T wiederum eine global eindeutige Zeitmarke *ts (T)* mitgegeben. Die Weiterleitung der Warteinformation in obiger Situation wird damit nur dann vorgenommen, wenn $ts(T_n) < ts(T_1)$ gilt. Damit wird bei einem globalen Deadlock mit zwei Rechnern gewährleistet, daß nur einer der beiden Rechner die Warteinformation weiterleitet. Bei mehr als zwei Rechnern ist es jedoch weiterhin möglich, daß ausgehend von mehr als einem Rechner die Warteinformation weitergegeben wird.

Beispiel 8-17

Für den in Abb. 8-8 gezeigten Deadlock (Beispiel 8-11) bildet sich in Rechner R1 folgender Zyklus EXTERNAL -> T2 -> T1 -> EXTERNAL,
in R2 dagegen EXTERNAL -> T1 -> T2 -> EXTERNAL.

Wenn *ts (T1)* < *ts (T2)* gilt, dann sendet lediglich R1 seine Warteinformation an R2, nicht jedoch umgekeht. In R2 wird der Wartegraph vervollständigt und der Zyklus

T1 -> T2 -> T1

entdeckt. Der Deadlock wird durch Rücksetzen der lokalen Transaktion T2 aufgelöst.

Der Obermarck-Algorithmus benötigt maximal $N^* (N-1)/2$ Nachrichten für einen sich über N Rechner erstreckenden globalen Deadlock. Für N=2 fällt also lediglich eine Nachricht an. Allerdings kann es zur Erkennung "falscher" Deadlocks kommen, da die Wartegraphen der einzelnen Rechner i.a. unterschiedliche Änderungszustände reflektieren [El86].

8.5.5 Hybride Strategien

Die Diskussion der vorgestellten Verfahren zur Deadlock-Behandlung zeigte, daß alle Vor- und Nachteile aufweisen. So sprechen vor allem Gründe der Einfachheit sowie das Umgehen zusätzlicher Nachrichten für eine Deadlock-Vermeidung oder einen Timeout-Ansatz. Diese verursachen jedoch u.U. eine hohe Anzahl unnötiger Rücksetzungen. Erkennungsansätze dagegen erlauben i. d. R. die geringste Anzahl von Rücksetzungen, sind jedoch schwierig zu realisieren und führen Kommunikationsaufwand ein. Vielversprechend erscheinen daher hybride Ansätze, um die Vorteile der einzelnen Verfahrensklassen zu vereinen.

Ein möglicher Ansatz hierzu sieht eine explizite Erkennung von lokalen Deadlocks vor, die also nur Objekte eines Rechners betreffen. Zur Behandlung globaler Deadlocks dagegen wird auf einen einfachen und billigen Vermeidungs- oder Timeout-Ansatz zurückgegriffen. Eine solche Vorgehensweise bewahrt den Vorteil der Einfachheit und vermeidet Kommunikation zur Deadlock-Behandlung. Außerdem ist davon auszugehen, daß die meisten Deadlocks nur Transaktionen eines Rechners berühren (und somit explizit erkannt werden), da ein Deadlock wie erwähnt meist nur zwei Transaktionen betrifft und häufig eine hohe Lokalität im Referenzverhalten erreicht wird. Daher ist eine hybride Strategie z.B. mit einem Timeout-Ansatz zur Auflösung globaler Deadlocks umso angebrachter, je höher die erzielbare Lokalität ist.

8.6 Abschließende Bemerkungen

In diesem Kapitel wurden die wichtigsten Ansätze zur Synchronisation in partitionierten (nicht-replizierten) Datenbanken vorgestellt. Es wurden zentralisierte und verteilte Sperrverfahren, Zeitmarkenverfahren sowie optimistische Protokolle diskutiert. Daneben wurden die Realisierung einer Mehrversionen-Synchronisation sowie die Alternativen zur Behandlung globaler Deadlocks erörtert.

Die Leistungsfähigkeit der einzelnen Verfahren ist vor allem bestimmt vom Kommunikationsumfang sowie der Häufigkeit von Blockierungen und Rücksetzungen. Die Kommunikationskosten sprechen bei fehlender Replikation eindeutig für verteilte Ansätze, da bei ihnen die Datenzugriffe während der Ausführung der Sub-Transaktionen lokal synchronisiert werden. Zeitmarkenverfahren und optimistische Protokolle sind deadlock-frei, erfordern jedoch auch zum Teil Blockierungen, um den Zugriff auf schmutzige bzw. unsichere Änderungen zu vermeiden. Sperrverfahren verlangen dagegen eine Deadlock-Behandlung, die eigene Kommunikationsbelastungen (bei einer Erkennung) oder zahlreiche unnötigen Rücksetzungen verursachen kann. Daher ist die für zentralisierte DBS festgestellte Überlegenheit von Sperrverfahren gegenüber anderen Synchronisationsverfahren [ACL87, Pe87] im verteilten Fall nicht notwendigerweise gegeben. Die zahlreichen Leistungsun-

tersuchungen im verteilten Fall weisen leider starke Unterschiede in der Methodik sowie den jeweiligen Annahmen auf, so daß zum Teil widersprechende Resultate erzielt wurden. In der relativ genauen Simulationsstudie [CL88] waren z.B. die Ergebnisse primär vom Umfang an Rücksetzungen bestimmt, so daß ein verteiltes Sperrverfahren mit (zentraler) Deadlock-Erkennung am besten abschnitt. Interessanterweise war ein Sperrverfahren mit Wound/Wait-Deadlock-Vermeidung nicht besser als ein reines Zeitmarkenverfahren.

Die meisten in der Literatur vorgestellten Synchronisationsverfahren wurden nicht implementiert. Kommerzielle DBS verwenden auch im verteilten Fall praktsich ausschließlich Sperrverfahren. Verfahren der anderen Klassen wurden zum Teil in Prototypen realisiert (Zeitmarkenverfahren z.B. in SDD-1 [BG83], optimistische Verfahren z.B. in Jasmin [LWL89]). Diese Implementierungen bieten jedoch i.a. nicht die für die Praxis erforderliche vollständige Funktionalität und Effizienz ("industrial strength"). So sind vor allem für optimistische Protokolle einige signifikante Implementierungsprobleme noch nicht gelöst, insbesondere die Unterstützung feiner Synchronisationsgranulate (kleiner als Seiten) sowie die effiziente Synchronisation auf Indexstrukturen [Hä84, Mo92b]. Ähnliche Schwierigkeiten bestehen jedoch auch bei Mehrversionen-Synchronisationsverfahren.

Übungsaufgaben

Aufgabe 8-1: Verbesserung von BOCC

Entwerfen Sie eine Verbesserung von BOCC, bei der nur echte Konflikte zur Rücksetzung der validierenden Transaktion führen. Hinweis: Verwenden Sie Zeitstempel zur Konflikterkennung.

Aufgabe 8-2: FOCC

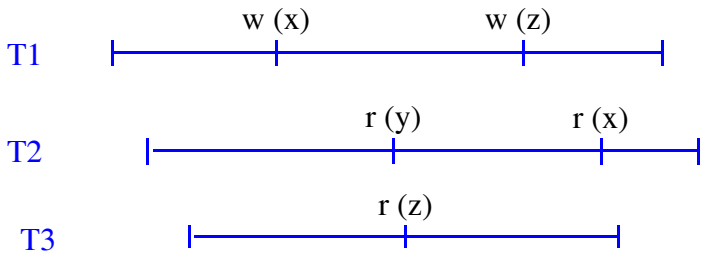
Wie kann bei FOCC das "Verhungern" einer Transaktion verhindert werden bzw. garantiert werden, daß jede Transaktion erfolgreich zu Ende kommt ?

Aufgabe 8-3: Schmutzige vs. unsichere Änderungen

Erläutern Sie den Unterschied zwischen schmutzigen und unsicheren Änderungen. Wieso ist bei optimistischer Synchronisation kein Zugriff auf schmutzige Änderungen möglich ?

Aufgabe 8-4: Vergleich von Synchronisationsverfahren

An einem Rechner sei folgender Schedule von drei Transaktionen mit Zugriff auf ausschließlich lokale Objekte gegeben, wenn keine Synchronisation erfolgt:

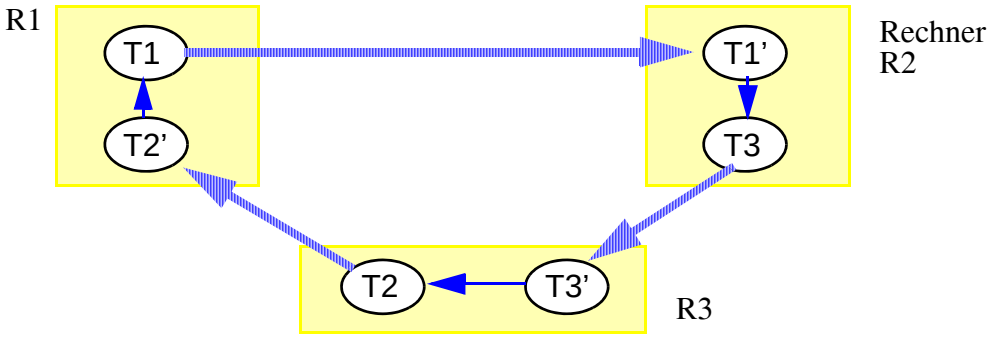


Bestimmen Sie für jedes der folgenden Synchronisationsverfahren die vorkommenden Blockierungen und Rücksetzungen sowie die sich ergebende Serialisierungsreihenfolge:

a. Zeitmarkenverfahren (Basic Timestamp Ordering)
b. BOCC
c. FOCC
d. RX-Sperrverfahren mit Wait/Die-Deadlock-Vermeidung
e. RX-Sperrverfahren mit Wound/Wait-Deadlock-Vermeidung
f. RX-Sperrverfahren mit Deadlock-Erkennung
g. RX-Sperrverfahren mit Mehrversionen-Synchronisation und Deadlock-Erkennung.

Aufgabe 8-5: Deadlock-Erkennung

Wenden Sie den Algorithmus von Obermarck zur Erkennung des folgenden Deadlocks an. Geben Sie jeden Zwischenschritt an (Annahme: *ts (T1)* < *ts (T2)* < *ts (T3)*). Wieviele Nachrichten werden zur Auflösung des Deadlocks benötigt?



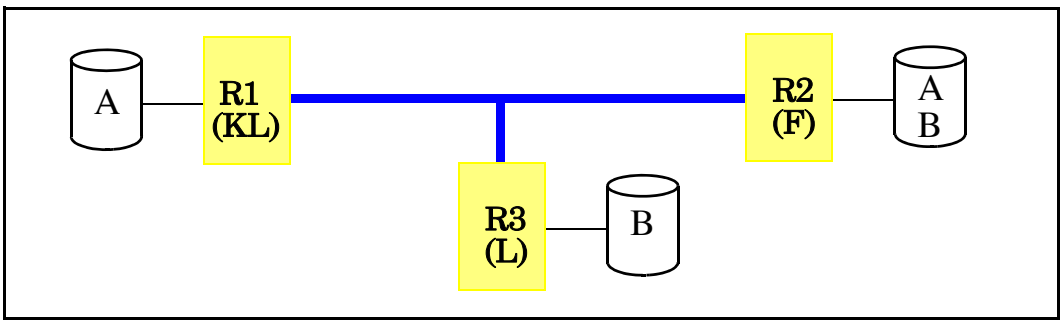
9 Replizierte Datenbanken

Hauptziel beim Einsatz replizierter Datenbanken ist die Steigerung der Verfügbarkeit eines Verteilten DBS. Denn repliziert an mehreren Knoten gespeicherte Daten bleiben auch nach Ausfall eines der Rechner erreichbar. Daneben wird auch eine Verbesserung der Leistungsfähigkeit angestrebt, insbesondere für Lesezugriffe. So lassen sich viele Kommunikationsvorgänge einsparen, indem man Objekte repliziert an allen Knoten speichert, an denen sie häufig referenziert werden (Unterstützung von Lokalität). Zum anderen erhöht die Wahlmöglichkeit unter mehreren Kopien das Potential zur Lastbalancierung, so daß die Überlastung einzelner Rechner eher vermeidbar ist. Unter diesen Gesichtspunkten bieten natürlich voll-redundante Datenbanken, bei denen jeder Knoten eine Kopie der gesamten Datenbank führt, die größten Vorteile; insbesondere können prinzipiell alle Lesezugriffe lokal abgewickelt werden.

Beispiel 9-1

In der Bankanwendung seien die Konten unter den einzelnen Filialen partitioniert, zusätzlich sei jeder Kontosatz in der Bankzentrale in Frankfurt repliziert gespeichert (Abb. 9-1). Dies unterstützt zum einen eine lokale Verarbeitung in den jeweiligen Zweigstellen; zudem können in der Zentrale Auswertungen über mehrere Filialen hinweg lokal berechnet werden. Weiterhin kann nach Ausfall eines Rechners (prinzipiell) weiterhin auf alle Konten zugegriffen werden.

Abb. 9-1: Beispiel replizierter Datenhaltung



Allerdings verursacht das Führen replizierter Daten nicht nur einen erhöhten Speicherplatzbedarf, sondern v.a. hohe Änderungskosten, um alle Replikate bei einer Modifikation zu aktualisieren. Dieser Aktualisierungsaufwand ist in ortsverteilten Systemen wegen der teuren Kommunikation besonders hoch, so daß in der Regel nur eine partielle Replikation in Betracht kommt. Replizierte Datenbanken führen ferner zu einer signifikanten Erhöhung der Implementierungskomplexität von Verteilten DBS, da die Existenz von Replikaten dem Benutzer gegenüber transparent gehalten werden soll (Replikationstransparenz). Das DBS hat somit dafür zu sorgen, daß Änderungen automatisch auf alle Kopien übertragen werden, damit diese untereinander konsistent bleiben. Weiterhin sollte den Transaktionen auch eine konsistente Version der Daten zur Verfügung stehen. Hierzu sind Erweiterungen im Synchronisationsprotokoll erforderlich, um zu verhindern, daß Leser unterschiedliche Änderungszustände zu sehen bekommen und daß Replikate eines Objektes gleichzeitig von mehreren Transaktionen geändert werden.

Die genannten Anforderungen werden von erweiterten Synchronisationsverfahren erfüllt, die das Korrektheitskriterium der *1-Kopien-Serialisierbarkeit* (one copy serializability) [BHG87] unterstützen. Diese lassen nur solche Schedules zu, welche zu serialisierbaren Schedules auf einer nicht-redundanten Datenbank äquivalent sind. Das DBS muß weiterhin dafür sorgen, daß die DB-Konsistenz auch im Fehlerfall gewahrt bleibt, insbesondere nach Netzwerk-Partitionierungen. Denn wenn z.B. in einem Hotelreservierungssystem mit replizierter Datenhaltung eine Netzwerk-Partitionierung auftritt, könnte es ohne Zusatzmaßnahmen vorkommen, daß zwei Kunden dasselbe Zimmer für dieselbe Nacht buchen. Schließlich verlangt die Unterstützung replizierter Datenbanken Erweiterungen im physischen Datenbankentwurf sowie bei der Query-Optimierung und -Verarbeitung.

In diesem Kapitel stellen wir die drei wichtigsten Verfahrensklassen zur Aktualisierung von und Synchronisation auf replizierten Datenbanken vor: sogenannte Write-All-Ansätze (Kap. 9.1), Primary-Copy-Verfahren (Kap. 9.2) sowie Voting-Ansätze (Kap. 9.3). Danach gehen wir noch auf zwei speziellere Ansätze zur Unterstützung replizierter Datenbanken ein, welche von einigen kommerziellen DBS bereits verwendet werden. Dies sind der Einsatz von sogenannten DB-Schnappschüssen (Kap. 9.4) sowie die Nutzung einer DB-Kopie zur Katastrophen-Recovery (Kap. 9.5). Der Einsatz von Datenreplikation in lokal verteilten Mehrrechner-DBS (Parallelen DBS) wird in Kap. 17.4 diskutiert.

9.1 Write-All-Ansätze

Die bekannteste Variante ist die sogenannte Write-All-Read-Any- bzw. *Read-Once-Write-All (ROWA)*-Strategie. Sie verlangt die synchrone Änderung aller Replikate vor Abschluß einer Änderungstransaktion. Dadurch ist gesichert, daß jedes Replikat auf dem aktuellen Stand ist, so daß zum Lesen jedes beliebige aus-

gewählt werden kann. Die Auswahl kann im Hinblick auf minimale Kommunikationskosten oder zur Unterstützung von Lastbalancierung erfolgen. Weiterhin ergibt sich für Lesezugriffe eine erhöhte Verfügbarkeit, da diese durchführbar sind, solange wenigstens eine Kopie erreichbar bleibt.

Diese Vorteile gehen jedoch auf Kosten der Änderungstransaktionen. Zum einen ist es beim Einsatz eines verteilten Sperrverfahrens erforderlich, vor jeder Änderung eine Schreibsperre auf allen Kopien zu erwerben, was einen enormen Zusatzaufwand an Kommunikation einführen kann*. Die betroffenen Knoten sind ferner alle im Commit-Protokoll zu beteiligen. Bei einem Zwei-Phasen-Commit-Protokoll werden zunächst in Phase 1 die Änderungen an alle Knoten übertragen und dort protokolliert, bevor in der zweiten Commit-Phase die Replikate selbst aktualisiert und die Sperren freigegeben werden. Ein weiterer Schwachpunkt liegt darin, daß die Verfügbarkeit für Änderer schlechter ist als bei fehlender Replikation! Denn eine Änderung ist nicht mehr möglich, sobald einer der Rechner ausfällt, an dem ein Replikat des zu ändernden Objekts gespeichert ist.

Beispiel 9-2

Für die Situation aus Beispiel 9-1 (Abb. 9-1) können mit dem ROWA-Protokoll in der Zentralen R2 sämtliche Lesezugriffe ohne Kommunikation auf der lokalen Kopie abgewickelt werden. Ebenso erfordert in den Filialen der Lesezugriff auf lokale Konten (A in R1, B in R3) keine Kommunikation. Jede Änderung eines Kontosatzes erfordert dagegen das Sperren sowie synchrone Aktualisieren beider Kopien. Nach einem Ausfall der Zentralen können daher nur noch Lesezugriffe erfolgen; der Ausfall eines Filialrechners verhindert weitere Änderungen von Konten der entsprechenden Filiale. Lesezugriffe auf lokale Kopien sind nach einem Rechnerausfall weiterhin möglich.

Zur Abschwächung des Verfügbarkeitsproblems wurde die *Write-All-Available*-Variante vorgeschlagen [BHG87], bei der nur die Replikate der verfügbaren Rechner gesperrt und aktualisiert werden müssen. Für einen ausgefallenen Rechner werden die nicht vorgenommenen Modifikationen eigens protokolliert und nach Wiedereinbringen des Rechners nachgefahren. Dieses Verfahren eignet sich allerdings nicht bei Netzwerk-Partitionierungen, da hierbei ein Auseinanderlaufen der in verschiedenen Teilnetzen vorgenommenen Änderungen droht. Ferner bleibt natürlich der hohe Mehraufwand für Änderer bezüglich Sperranforderungen und Commit-Protokoll.

---

* Dieser Aufwand ließe sich mit einem verteilten optimistischen Synchronisationsprotokoll (Kap. 8.3.3) vermeiden, bei dem Synchronisation sowie Aktualisierung der Replikate vollständig ins Commit-Protokoll integriert werden können. Optimistische Synchronisationsverfahren werden in diesem Kapitel jedoch nicht weiter betrachtet.

9.2 Primary-Copy-Verfahren

Primary-Copy-Verfahren [AD76, St79] zielen auf eine effizientere Bearbeitung von Änderungen ab, indem eine Änderungstransaktion bei einer Modifikation nur eines der Replikate, die sogenannte *Primärkopie* (primary copy), zu ändern braucht. Die Aktualisierung der anderen Replikate wird dann asynchron vom Primary-Copy-Rechner aus durchgeführt, i.a. erst nach Ende der ändernden Transaktion ("as soon as possible"). Dabei können Nachrichten eingespart werden, indem der Primärkopien-Rechner mehrere Änderungen gebündelt an die kopienhaltenden Rechner weiterleitet. Dies geht dann jedoch zu Lasten einer verzögerten Anpassung der Replikate. Im allgemeinen werden die Primärkopien unterschiedlicher Objekte an verschiedenen Knoten gespeichert sein, um einen zentralen Engpaß zu vermeiden. Zur Reduzierung von Kommunikationsvorgängen empfiehlt sich dabei natürlich die Zuordnung einer Primärkopie an den Rechner, an dem das Objekt am häufigsten geändert wird.

Insgesamt bestehen mehrere Alternativen bei der Realisierung eines Primary-Copy-Protokolls. Ein Ansatz besteht darin, wie beim ROWA-Protokoll Änderungssperren verteilt bei allen kopienhaltenden Rechnern anzufordern. Am Transaktionsende wird jedoch nur die Primärkopie synchron aktualisiert, so daß eine schnellere Bearbeitung von Änderungstransaktionen als mit dem ROWA-Ansatz erreicht wird. Nach Aktualisierung der Primärkopie geht die Schreibsperre dann quasi in den Besitz des Primary-Copy-Rechners über, der diese bis nach der Aktualisierung der Replikate hält. Das verteilte Sperren aller Kopien bringt den Vorteil mit sich, daß Leser wie im ROWA-Verfahren ein beliebiges Replikat referenzieren (sperren) können, nicht also notwendigerweise die Primärkopie. Die verzögerte Aktualisierung der Replikate kann dafür vermehrte Sperrkonflikte verursachen.

Die Alternative zu diesem Ansatz besteht darin, Schreibsperren nur noch zentral beim jeweiligen Primary-Copy-Rechner anzufordern, um den Aufwand verteilter Schreibsperren zu umgehen. Für Leser besteht jetzt jedoch das Problem, daß die lokalen Kopien aufgrund der verzögerten Aktualisierung möglicherweise veraltet sind. Für die Behandlung der Lesezugriffe bestehen im wesentlichen drei Alternativen, die sich dadurch unterscheiden, wo die Objektzugriffe erfolgen und wo die Sperren angefordert werden (zentral beim Primary-Copy-Rechner oder verteilt/lokal):

1. *Lesezugriff auf Primärkopie*
   In diesem Ansatz referenzieren und sperren Lesetransaktionen die Primärkopie, um den aktuellsten, konsistenten DB-Zustand zu sehen. Damit werden jedoch die Replikate kaum noch genutzt, so daß dieser Ansatz nur in Kombination mit einem der anderen von Interesse erscheint.

2. *Lesezugriff auf lokale Kopien, Sperranforderung beim Primärkopien-Rechner*
   Im Gegensatz zur vorherigen Alternative werden nur die Lesesperren zentral beim Primärkopien-Rechner angefordert, der Objektzugriff erfolgt dagegen lokal. Damit kann der Kommunikationsumfang für Leseoperationen reduziert und der Primärkopien-Rechner entlastet werden. Bei der Anforderung der Lesesperre kann gleichzeitig überprüft werden, ob die für den Lesezugriff vorgesehene Objektkopie bereits auf dem aktuellen Stand ist. Falls nicht, kann entweder auf den Abschluß der Aktualisierung gewartet oder auf eine bereits aktualisierte Kopie ausgewichen werden (z.B. die Primärkopie).
3. *Lokale Abwicklung von Lesezugriffen*
   In diesem Ansatz greifen Leser auf lokale Objekte zu, ohne eine Synchronisierung über den Primärkopien-Rechner vorzunehmen. Die Aktualität der lokalen Kopien ist dabei nicht mehr gewährleistet, wenngleich die Daten i.a. höchstens wenige Sekunden veraltet sein dürften [Gr86]. Gravierender ist jedoch, daß Leser in diesem Fall eine inkonsistente Sicht auf die Datenbank haben, da sie i.d.R. für verschiedene Objekte unterschiedliche Änderungszustände sehen*. Dies kann nur in einfachen Fällen umgangen werden, z.B. wenn eine Transaktion nur ein Objekt oder nur Objekte mit lokaler Primärkopie referenziert. Anderenfalls bleibt für Lesetransaktionen, die den Zugriff auf eine konsistente DB-Version benötigen, eine der beiden vorgenannten Möglichkeiten. Alternativ dazu ist es möglich, das Synchronisationsverfahren so zu erweitern, daß Leser zwar u.U. einen leicht veralteten, aber dennoch konsistenten DB-Zustand sehen (etwa im Rahmen eines Mehrversionen-Konzepts [Wa83]).

Beispiel 9-3

Für die Situation aus Beispiel 9-1 (Abb. 9-1) nehmen wir an, daß die Primärkopien der Kontosätze an den jeweiligen Filialrechnern liegen, das heißt, R1 hält die Primärkopie von A und R3 von B. Damit können in diesen Rechnern sämtliche Lese- und Schreibzugriffe auf lokale Konten ohne Kommunikation abgewickelt werden. Kommunikation fällt lediglich für die Kontenzugriffe in der Zentrale R2 an. Dort vorzunehmende Kontoänderungen erfordern Kommunikation zum Sperren und Aktualisieren der jeweiligen Primärkopie. Die Behandlung von Lesezugriffen in R2 differiert für jede der drei Strategien. Strategie 1 verlangt das Sperren und Lesen der Primärkopie und somit Nachrichten für Objektzugriff und Commit, während Strategie 2 lediglich Nachrichten für die Sperranforderung und -freigabe erfordert. Mit Strategie 3 greift R2 unsynchronisiert auf die lokalen, möglicherweise veralteten Objektkopien zu, was für globale Auswertungen durchaus ausreichend sein kann.

Im Falle einer Netzwerk-Partitionierung können in der Partition mit dem jeweiligen Primary-Copy-Rechner Objekte weiterhin gelesen und geändert werden. In

* Allerdings wird in der Praxis selbst in zentralisierten DBS Lesetransaktionen aus Leistungsgründen meist keine konsistente DB-Sicht gewährt, da Lesesperren i.a. nur kurz gehalten werden. Es liegt dabei jedoch meist eine Wahlmöglichkeit für den Programmierer vor, ob eine derartige Einschränkung der Konsistenz zugelassen wird.

den anderen Partitionen ist i.a. kein weiterer Zugriff möglich; lediglich bei der obigen Strategie 3 ist ein Lesezugriff auf die ggf. veralteten Kopien zulässig.

Der Ausfall eines Rechners macht die an ihm vorliegenden Primärkopien unerreichbar, so daß keine Änderungen bis zur Reintegration des Knotens mehr möglich sind. Um diese lange Unterbrechung zu vermeiden, kann vorgesehen werden, einen neuen Primärkopien-Rechner zu bestimmen. Dies setzt jedoch voraus, daß die neue Primärkopie auf den neusten Stand gebracht werden kann. Im Falle einer Netzwerk-Partitionierung ist dabei darauf zu achten, daß für ein Objekt höchstens in einer der Partitionen ein neuer Primärkopien-Rechner bestimmt wird (z.B. in der Partition mit mehr als der Hälfte aller Kopien).

9.3 Voting-Verfahren

Voting-Verfahren verlangen, daß vor dem Zugriff auf ein repliziert gespeichertes Objekt zunächst eine ausreichende Anzahl an Stimmen (votes) gesammelt werden muß. Wir diskutieren zunächst die einfachste Variante des *Mehrheits-Votierens* [Th79]. Danach wird die Verallgemeinerung gewichteter Voting-Verfahren vorgestellt.

9.3.1 Mehrheits-Votieren (Majority Consensus)

Bei diesem Ansatz erfordert das Ändern eines Objektes, daß die Transaktion eine Mehrheit der Replikate mit einer Schreibsperre belegt und diese Replikate ändert*. Da zu einem Zeitpunkt nur eine Transaktion die Mehrheit der Replikate sperren kann, ist sichergestellt, daß ein Objekt nicht gleichzeitig von mehreren Transaktionen geändert wird. Zum Lesen wird ebenfalls eine Mehrheit der Replikate gesperrt und davon ein aktuelles Replikat referenziert. Die Aktualität der Replikate kann z.B. durch Änderungszähler festgestellt werden. Offenbar ist durch das Sperren einer Mehrheit immer gewährleistet, daß mindestens eines der Replikate auf dem aktuellen Stand ist. Der Vorteil des Verfahrens ist, daß ein Objekt auch nach einem Rechnerausfall oder einer Netzwerk-Partitionierung noch referenzierbar ist, solange eine Mehrheit der Replikate erreichbar ist.

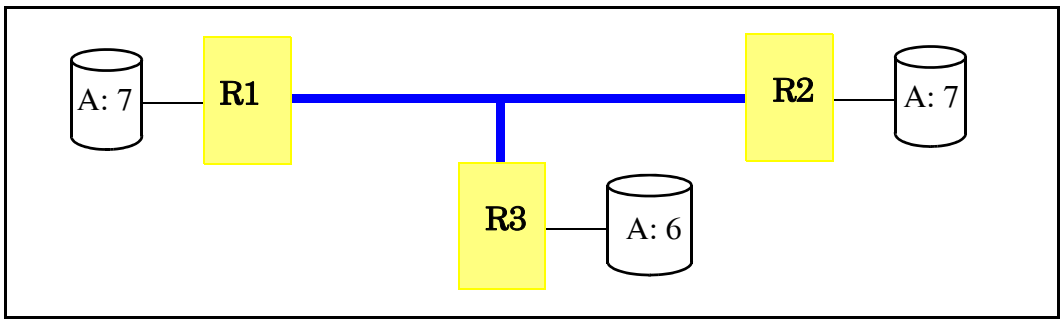
Beispiel 9-4

In Abb. 9-2 existieren drei Kopien von Objekt A. Das Lesen oder Ändern eines Objektes erfordert beim Mehrheits-Votieren somit den Erwerb einer Lese- bzw. Schreibsperre an mindestens zwei Knoten. Abb. 9-2 zeigt die Situation, nachdem eine Transaktion Objekt A an den Knoten R1 und R2 geändert und den Änderungszähler auf 7 inkrementiert hat. Eine Transaktion in R3, die jetzt auf A zugreifen will, muß entweder an R1 oder R2 eine Sperre erwerben, um eine ausreichende Mehrheit zu erhalten. Durch Vergleich der Änderungszähler kann der Zugriff auf die lokale, jedoch veraltete Version von A in R3 verhin-

* In diesem Fall kann die Sperre als "Stimme" interpretiert werden.

dert werden.
Fällt einer der Rechner aus, kann weiterhin auf Objekt A zugegriffen werden, da mit zwei "überlebenden" Knoten noch eine Mehrheit vorliegt. Ebenso kann bei einer Netzwerk-Partitionierung die Verarbeitung in einer aus zwei Knoten bestehenden Partition fortgeführt werden.

Abb. 9-2: Einsatz von Änderungszählern beim Mehrheits-Votieren



Der Hauptnachteil des Mehrheits-Votierens liegt darin, daß jeder Objektzugriff Kommunikation erfordert, um eine Mehrheit zu bilden. Dies gilt auch für Lesezugriffe auf eine lokal vorhandene Kopie, welche z.B. mit dem ROWA-Protokoll ohne Kommunikation abgewickelt werden konnten. Der Schreibaufwand liegt zwischen dem des Primary-Copy-Ansatzes und dem des ROWA-Verfahrens. Für den häufigen Spezialfall mit zwei Kopien (Beispiel 9-1) ist das Mehrheits-Votieren ungeeignet, da es für sämtliche Zugriffe die Involvierung beider Rechner verlangt.

9.3.2 Gewichtetes Votieren (Quorum Consensus)

Diese Nachteile können durch verallgemeinerte Voting-Verfahren abgeschwächt werden [Gi79]. Dabei wird jedem Replikat eines Objektes ein bestimmtes Gewicht (Stimmenanzahl) zugeordnet. Zum Lesen eines Objektes muß nun eine Transaktion erst eine bestimmte Anzahl von Stimmen sammeln (Lese-Quorum), ebenso zum Schreiben (Schreib-Quorum). Wenn v die Gesamtanzahl der für ein Objekt vergebenen Stimmen (votes) ist, und r bzw. w das Lese-Quorum (read quorum) bzw. Schreib-Quorum (write quorum), dann müssen folgende beiden Bedingungen erfüllt sein:

a) $w > v/2$

b) $r + w > v$

Die erste Bedingung garantiert, daß ein Objekt nicht gleichzeitig von mehr als einer Transaktion geändert wird. Weiterhin kann bei einer Netzwerk-Partitionierung die Bedingung in höchstens einer der Partitionen erfüllt werden, so daß parallele Änderungen eines Objekts in mehreren Partitionen ausgeschlossen sind. Die zweite Bedingung verhindert, daß ein Objekt zur gleichen Zeit gelesen und geändert wird. Weiterhin ist damit sichergestellt, daß jedes Lese-Quorum mindestens

ein Replikat enthält, das zum Schreib-Quorum des letzten Änderungszugriffs gehörte und damit auf dem aktuellsten Stand ist.

Das anfangs beschriebene Mehrheits-Votieren ergibt sich offenbar als Spezialfall, wenn man jedem Replikat das gleiche Gewicht zuordnet (z.B. 1 Stimme) und r und w eine einfache Mehrheit darstellen. Durch unterschiedliche Gewichtung der Replikate und Festlegung von r und w kann man aber nun in flexibler Weise sowohl die Kosten eines Lese- bzw. Schreibzugriffs als auch die Verfügbarkeit im Fehlerfall bestimmen. So kann man z.B. an einem Rechner einen besonders schnellen Zugriff auf ein Objekt ermöglichen, indem man dem dort gespeicherten Replikat eine hohe Anzahl von Stimmen zuordnet. Je kleiner r (w) gewählt wird, desto schneller sind Lesezugriffe (Schreibzugriffe) auszuführen, da dann weniger Stimmen gesammelt werden müssen. Entsprechend erhöht sich dadurch auch die Verfügbarkeit der Leseoperation (Schreiboperation), da dann die notwendigen Stimmen oft noch eingeholt werden können, selbst wenn ein(ige) Rechner nicht erreichbar ist (sind). Allerdings geht wegen obiger Bedingung b) die Bevorzugung von Leseoperationen immer auf Kosten der Schreibzugriffe und umgekehrt.

Beispiel 9-5

Objekt A sei an vier Rechnern R1 bis R4 repliziert; die Stimmenverteilung sei <2, 1, 1, 1>, d.h., R1 hält 2 Stimmen, die anderen Rechner jeweils eine (v=5). Wählt man r=3, w=3 sind zwei bis drei Rechner involviert, um einen Lese- oder Schreibzugriff abzuwickeln. Durch die Bevorzugung von R1 genügen zwei Rechner, sobald R1 an dem Quorum beteiligt ist. Der Objektzugriff ist auch noch nach Ausfall jedes der Rechners möglich. Solange R1 verfügbar bleibt, ist der Zugriff auch noch nach Ausfall von zwei der drei anderen Rechner möglich.
Will man Lese- gegenüber Schreibzugriffen bevorzugen, kann man z.B. r=2, w=4 wählen. Damit können Lesezugriffe in R1 lokal abgewickelt werden; dafür sind mindestens drei Rechner (darunter stets R1) an einem Schreibzugriff zu beteiligen. Nach Ausfall von R1 ist das Objekt nicht mehr änderbar.

Die Flexibilität des Voting-Ansatzes wird daran deutlich, daß sich durch spezielle Wahl der Parameter nicht nur das Mehrheits-Votieren, sondern auch das ROWA-Verfahren sowie eines der Primary-Copy-Protokolle als Spezialfälle ergeben:

- Ein ROWA-Verfahren erhält man, indem man jedem Replikat eine Stimme zuordnet und r=1 und w=v (= Anzahl der Replikate) wählt. Dieser Fall unterstützt lokale Lesezugriffe, was beim Mehrheits-Votieren nicht möglich ist.
- Ein Primary-Copy-Verfahren ergibt sich, wenn man der Primärkopie eine Stimme zuweist und allen anderen Replikaten keine Stimme und r=w=1 wählt. Dieses Verfahren verlangt, daß Lesevorgänge an die Primärkopie gerichtet werden. Lokale Kopien ohne Stimmen können wiederum für Lesezugriffe ohne Konsistenzzusicherung genutzt werden.

Auch der Fall von nur zwei Kopien kann mit dem verallgemeinerten Voting-Verfahren besser als beim Mehrheits-Votieren behandelt werden. Wählt man etwa die Stimmenverteilung <2, 1> und r=w=2, dann kann das Lese- bzw. Schreib-Quorum

lokal am ersten Knoten erreicht werden. Ein Objektzugriff am zweiten Rechner erfordert stets Kommunikation, um ein ausreichendes Quorum zu bilden. Die zweite Kopie kann dennoch sinnvoll sein, um den Objektzugriff selbst lokal durchzuführen (sofern die Aktualität der Kopie festgestellt wurde). In unserer Bankanwendung (Beispiel 9-1) könnten so in jeder Filiale lokale Konten ohne Kommunikation gelesen und geändert werden. In der Zentralen wird zwar für jeden Kontozugriff Kommunikation zur Sperrbehandlung notwendig, dafür kann die Verarbeitung selbst auf den lokalen Kopien erfolgen.

Ein Hauptproblem bei Voting-Verfahren liegt in der geeigneten Wahl der Stimmenvergabe sowie der Lese- und Schreib-Quoren. Obwohl es zu dieser Thematik eine Reihe von Arbeiten gibt (z.B. [GB85]), erscheint eine für den Praxiseinsatz geeignete Lösung, die keine oder nur eine geringe Involvierung der Systemverwalter erfordert, nicht in Sicht. In der Literatur wurden daneben verschiedene Erweiterungen wie dynamische, mehrdimensionale oder hierarchische Voting-Verfahren vorgeschlagen, die in [Bo91] überblicksartig diskutiert werden.

9.4 Schnappschuß-Replikation

Die Forderung, sämtliche Replikate eines Objektes stets auf demselben Stand zu halten, verursacht v.a. in geographisch verteilten Systemen sehr hohe Änderungskosten. Weiterhin führen die erweiterten Synchronisationsanforderungen i.a. Mehraufwand an Kommunikation ein. Diese Kosten können durch eine schwächere Form der Datenreplikation, sogenannte Schnappschüsse (snapshots), signifikant reduziert werden [AL80]. Ein *Schnappschuß* entspricht dabei einer materialisierten Sicht, die durch eine DB-Anfrage spezifiziert wird. Das Ergebnis der Anfrage wird als eigenständiges DB-Objekt unter einem benutzerspezifizierten Namen gespeichert. Der Zugriff auf einen Schnappschuß erfolgt mit der jeweiligen DB-Anfragesprache, wobei jedoch nur Lesezugriffe zugelassen werden.

Eine mögliche Schnappschuß-Definition in der Bankanwendung sieht folgendermaßen aus:

```
CREATE SNAPSHOT Underflow AS
SELECT KNR, KTONR, KTOSTAND
FROM KONTO
WHERE KTOSTAND < 0
REFRESH EVERY DAY
```

Diese Anweisung ermittelt alle überzogenen Konten und speichert das Ergebnis in einer eigenen Relation *Underflow*. Das Ergebnis entspricht einer Kopie auf einer Teilmenge der KONTO-Relation, deren Gültigkeit sich auf den Auswertungszeitpunkt bezieht (Schnappschuß). In [AL80] wurde vorgesehen, daß der Benutzer bei der Definition eines Schnappschusses (mit der REFRESH-Option) ein "Auffri-

schen" der Kopie in bestimmten Zeitabständen verlangen kann (stündlich, täglich, wöchentlich etc.). Neben einer derart automatischen Aktualisierung kann auch über eine eigene REFRESH-Anweisung die Schnappschuß-Aktualisierung explizit veranlaßt werden. Für unser Beispiel wird dies erreicht durch die Anweisung

REFRESH SNAPSHOT Underflow.

Das Schnappschuß-Konzept ist relativ einfach und effizient realisierbar und dennoch flexibel. Zur Spezifikation eines Schnappschusses steht die volle Auswahlmächtigkeit der DB-Anfragesprache zur Verfügung, so daß die "Kopien" keineswegs auf Teilmengen der Datenbank beschränkt sind, sondern auch aggregierte Informationen, Verknüpfungen mehrerer Relationen etc. enthalten können. Ebenso kann der Benutzer bzw. DB-Administrator die Aktualisierungsintervalle in flexibler Weise an die Bedürfnisse der jeweiligen Anwendung anpassen. Weiterhin wird ein sehr gutes Leistungsverhalten unterstützt, da Zugriffe auf einen lokal gespeicherten Schnappschuß ohne Kommunikation erledigt werden und eine Lastbalancierung unterstützen (Entlastung des Rechners mit der "Primärkopie"). Weiterhin wird durch die relativ seltene Aktualisierung von Schnappschuß-Kopien der Änderungsaufwand sehr klein gehalten. Ein weiterer Vorteil liegt darin, daß das Synchronisationsproblem replizierter Datenbanken entfällt, da auf Schnappschuß-Relationen nur Leseoperationen zulässig sind.

Auf der anderen Seite ist natürlich die "Qualität" einer Schnappschuß-Kopie geringer als die einer Kopie bei allgemeinen replizierten Datenbanken. Insbesondere sind auf einem Schnappschuß keine Änderungsoperationen möglich und die Informationen sind i.d.R. veraltet (wenn auch in einem kontrollierten Maß). Dies reduziert auch den Wert der Kopie im Fehlerfall (Ausfall des Rechners mit der Primärkopie). Dennoch ist diese schwächere Form der Replikation für viele Anwendungen ausreichend. Man denke etwa an Verzeichnisse wie Ersatzteillisten bei KFZ-Betrieben, Buchkataloge in Bibliotheken oder Telefonbücher. Auf solche Daten wird häufig und vorwiegend lesend zugegriffen, so daß sich eine Replikation zur Einsparung von Kommunikation lohnt. Zudem reicht in diesen Fällen eine periodische Aktualisierung der Kopien vollkommen aus.

Die skizzierte Form von Schnappschuß-Replikation wurde im Prototyp R* realisiert [AL80]. Effiziente Techniken zur Aktualisierung von Schnappschüssen werden in [Li86] vorgestellt. Eine zunehmende Anzahl kommerzieller Produkte unterstützt eine Schnappschuß-Replikation, u.a. von IBM (DataPropagator), DEC (Data Distributor), ASK/Ingres (Replicator), Oracle, Software AG (Entire Transaction Propagator), HP (Allbase/Replicate) etc.

Ein den DB-Schnappschüssen verwandtes Konzept ist das der *Quasi-Kopien* [ABG90], das jedoch für eine Kopienhaltung bzw. Pufferung von Objekten in Workstation/Server-Umgebungen vorgesehen wurde. Die Idee dabei ist, anwendungsspezifisches Wissen zu verwenden, um die Aktualisierung replizierter Objekte zu steuern. Zum Beispiel könnte so verlangt werden, daß Änderungen einer Preisliste

nur dann propagiert werden sollen, sobald sich Preisunterschiede von über 5% ergeben oder sich mehr als 10% aller Preise geändert haben.

9.5 Katastrophen-Recovery

Die Unterstützung einer hohen Verfügbarkeit verlangt eine schnelle Recovery von Transaktionsfehlern, Rechner- oder Externspeicherausfällen. Zentralisierte DBS bieten höchstens eine schnelle Behandlung von Transaktionsfehlern sowie Externspeicherfehlern (z.B. über Spiegelplatten). Rechnerausfälle (sowie Kommunikationsfehler) können i.d.R. am schnellsten durch lokal verteilte Mehrrechner-DBS behandelt werden. Diese Ansätze unterstützen jedoch keine effektive Recovery von "Katastrophen", worunter man die vollständige Zerstörung eines Rechenzentrums z.B. aufgrund eines Erdbebens oder eines Terroranschlags versteht. In diesem Fall sind nämlich i.a. neben den Rechnern auch sämtliche Externspeicher mit der Datenbank sowie den Log-Dateien und Archivkopien zerstört. Der traditionelle Recovery-Ansatz für einen solchen Fall besteht darin, die Archivkopien auf Magnetbändern an einem geographisch weit entfernten Ort vom Rechenzentrum aufzubewahren [GA87]. Im Katastrophenfall wird die Verarbeitung auf der letzten Archivkopie an einem anderen Rechenzentrum fortgesetzt. Dieser Ansatz ist für OLTP-Anwendungen (z.B. zur Flugreservierung oder in Banken) vielfach inakzeptabel, da sämtliche Transaktionen seit Erstellung der letzten Archivkopie verloren sind und es i.a. zu lange dauert, bis die Archivkopie installiert ist und die Verarbeitung fortgesetzt werden kann. Ferner entstehen große Verzögerungen im Kommunikationsnetz, bis sämtliche Terminals mit dem neuen Verarbeitungssystem verbunden sind [GA87].

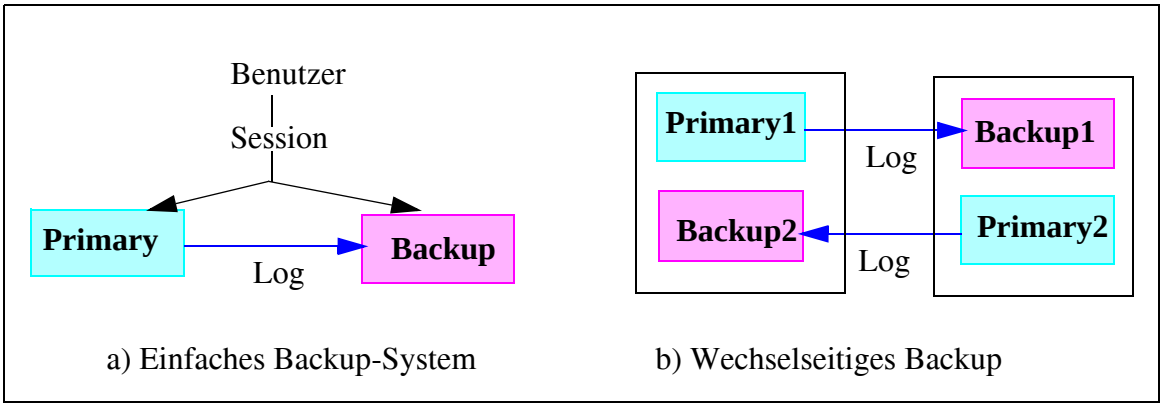
9.5.1 Systemstruktur

Replizierte Datenbanken bieten hier natürlich eine Lösung, wenn die an einem Rechner (bzw. Rechenzentrum) vorliegenden Daten vollständig an anderen, geographisch verteilten Knoten repliziert sind. Dieser allgemeine Ansatz wird jedoch derzeit für OLTP-Anwendungen mit hohen Leistungs- und Verfügbarkeitsanforderungen nicht verfolgt, da die aufwendige Aktualisierung der Replikate signifikante Leistungseinbußen verursachen kann*. Stattdessen strebt man i.a. eine Lösung mit einer begrenzten Form von Replikation an, bei der eine vollständige Kopie der Datenbank an einem geographisch entfernten Backup-Rechenzentrum geführt wird, mit der im Katastrophenfall die Verarbeitung fortgesetzt werden kann. Ein solcher Ansatz wird bereits in kommerziellen Produkten wie Tandems RDF (Remote Data Facility) [Ly90], IBMs RSR (Remote Site Recovery) für IMS, Sybase

* Daneben gibt es noch eine Reihe weiterer Gründe, die gegen ortsverteilte Mehrrechner-DBS sprechen (Kap. 3.1).

(Replication Server) sowie Informix (Data Replication Server) unterstützt. Daneben befaßten sich mehrere konzeptionelle Arbeiten mit der Realisierung einer derartigen Katastrophen-Recovery [GPH90, BT90, KHGP91, MTO93, GR93].

Abb. 9-3: Systemkonfigurationen zur Katastrophen-Recovery [GR93]



Die in diesen Ansätzen unterstellte Systemkonfiguration ist stark vereinfacht in Abb. 9-3a dargestellt. Im Normalbetrieb findet dabei die gesamte DB-Verarbeitung im Primärsystem (Primary) statt, d.h. einem Rechenzentrum mit einem zentralisierten DBS bzw. lokal verteilten Mehrrechner-DBS. Das geographisch entfernte Backup-System ist passiv; es wird nur im Fehlerfall genutzt. Sämtliche Terminals sind mit beiden Rechenzentren verbunden. Neben einer Kopie der Datenbank werden im Backup-System auch Kontrollinformationen zu den offenen Benutzer-Dialogen oder Sessions geführt, damit im Fehlerfall ein schnelles Umschalten der Verbindungen möglich wird. Sämtliche Änderungen des Primärsystems werden unmittelbar an das Backup-System übertragen, so daß die DB-Kopie immer auf dem aktuellen Stand gehalten werden kann. Hierzu werden die Log-Daten vom Primärsystem zum Backup-System übertragen, wobei die Log-Daten im Backup-System nochmals gesichert und auf die DB-Kopie angewendet werden. Dieser Ansatz hat den Vorteil, daß im Primärsystem ein geringer Zusatzaufwand anfällt, da dort die Log-Daten ohnehin erstellt werden. Im Backup-System kann die Aktualisierung der Datenbank durch Anwendung von Log-Daten mit bekannten Recovery-Konzepten und mit vergleichsweise geringem CPU-Aufwand erfolgen.

Nach Ausfall des Primärsystems veranlaßt der Systemverwalter ein Umschalten auf das Backup-System, wo die Verarbeitung typischerweise nach wenigen Sekunden fortgeführt werden kann*. Die Verlagerung der DB-Verarbeitung ins Backup-System kann auch für geplante Unterbrechungen im Primärsystem genutzt wer-

* Eine automatische Umschaltung zwischen Primär- und Backup-System ist i.a. nicht möglich/sinnvoll, da das System z.B. nicht zwischen einem tatsächlichen Ausfall des Primärsystems und einem Ausfall im Kommunikationssystem unterscheiden kann [Ly90, GR93].

den, etwa zur Hardware-Aufrüstung oder Installation neuer Software-Versionen (z.B. für Betriebssystem oder DBS). Weiterhin können Dienstprogramme wie die Erzeugung einer Archivkopie im Backup-System ausgeführt werden und damit das Primärsystem entlasten. Schließlich können auch Lesetransaktionen (z.B. Ad-hoc-Anfragen), die keine aktuelle DB-Sicht benötigen, im Backup-System bearbeitet werden.

Abb. 9-3b zeigt eine erweiterte Systemkonfiguration, bei der beide Rechenzentren im Normalbetrieb genutzt werden können. Hierzu wird eine Partitionierung von Datenbank und Anwendungen in zwei Teilsysteme vorgenommen, wobei Transaktionen aus Leistungsgründen in jedem Teilsystem weitgehend lokal ausführbar sein sollten. Die beiden Primärsysteme laufen in verschiedenen Rechenzentren und werden durch ein eigenes Backup-System im jeweils anderen Rechenzentrum gegenüber Katastrophen gesichert. Eine solche Konfigurierung wird von einigen der genannten Implementierungen unterstützt, z.B. Tandems RDF.

9.5.2 Commit-Behandlung

Eine wesentliche Entwurfsentscheidung bei der Realisierung eines solchen Ansatzes betrifft die Festlegung, ob die Änderungen einer Transaktion sowohl im Primär- als im Backup-System zu protokollieren sind, bevor ein Commit möglich ist. Diese Festlegung führt zur Unterscheidung von 1-sicheren (1-safe) und 2-sicheren (2-safe) Transaktionen [MTO93]. Für ***2-sichere Transaktionen*** ist durch ein verteiltes Commit-Protokoll zwischen Primär- und Backup-System zu gewährleisten, daß die Änderungen einer Transaktion an beiden Systemen gesichert sind, bevor die Commit-Entscheidung getroffen wird*. Für ***1-sichere Transaktionen*** erfolgt dagegen das Commit bereits nach Sicherung der Änderungen im Primärsystem; die Übertragung der Log-Daten an das Backup-System geschieht asynchron. Der Vorteil 2-sicherer Transaktionen liegt darin, daß auch im Katastrophenfall keine Änderungen verlorengehen und die DB-Kopie auf den aktuellsten Zustand gebracht werden kann. Andererseits ergeben sich signifikante Leistungseinbußen, da die Antwortzeit sich um die Übertragungszeit für die Log-Daten sowie die Schreibverzögerung im Backup-System erhöht. Weiterhin ist der Ansatz von der ständigen Verfügbarkeit des Backup-Systems abhängig, so daß sich ohne weitere Vorkehrungen die Verfügbarkeit sogar reduzieren kann. Zur Vermeidung dieser Nachteile unterstützen existierende Systeme eine asynchrone Aktualisierung der Bakkup-Kopie. Der Verlust einiger weniger Transaktionen im Katastrophenfall wird dabei in Kauf genommen.

---

* Hierzu ist kein vollständiges Zwei-Phasen-Commit-Protokoll erforderlich. Es genügt die synchrone Übertragung der Log-Daten in der ersten Commit-Phase [GR93], so daß lediglich zwei zusätzliche Nachrichten (sowie eine Log-E/A) anfallen.

Wünschenswert wäre ein hybrider Ansatz zur Unterstützung von 1- und 2-sicheren Transaktionen. Damit könnte für die Mehrzahl der Transaktionen eine effiziente asynchrone Übertragung der Log-Daten vorgenommen werden. "Wichtige" Transaktionen, z.B. Banktransaktionen mit hohem Geldumsatz, könnten dagegen als 2-sicher eingestuft werden, um ihren Verlust zu vermeiden. Bei der Realisierung eines solchen gemischten Ansatzes ist zu beachten, daß im Primärsystem Abhängigkeiten von 2-sicheren Transaktionen zu zuvor beendeten 1-sicheren Transaktionen bestehen können. Diese Abhängigkeiten müssen auch im Backup-System beachtet werden, um die Rekonstruktion eines konsistenten DB-Zustands zu ermöglichen.

Beispiel 9-6

Im Primärsystem habe Transaktion T2 von Transaktion T1 geänderte Objekte gelesen und für eigene Änderungen verwendet, so daß die Serialisierungsreihenfolge T1 < T2 besteht. Wenn T1 1-sicher, T2 dagegen 2-sicher ist, ist es aufgrund der verzögerten Übertragung der Log-Daten für 1-sichere Transaktionen möglich, daß die Log-Daten von T1 erst nach denen von T2 im Backup-System eintreffen. Probleme gibt es, falls das Primärsystem ausfällt, nachdem die Log-Daten von T2 im Backup-System gesichert sind, jedoch bevor die Log-Daten von T1 abgeschickt wurden. In diesem Fall würde die alleinige Anwendung der Änderungen von T2 im Backup-System einen inkonsistenten DB-Zustand erzeugen.

Die Lösung dieses Problems erfordert, daß die Log-Daten von 1-sicheren Transaktionen, die im Primärsystem vor einer 2-sicheren Transaktion beendet wurden, im Backup-System gesichert sind, bevor das Commit der 2-sicheren Transaktion erfolgt [MTO93]. Dies ist einfach realisierbar, wenn alle Log-Daten innerhalb eines "Log-Stroms" übertragen werden*. In diesem Fall sind die Log-Daten im Backup-System lediglich in der gleichen Reihenfolge anzuwenden, wie sie im Primärsystem generiert wurden.

Liegt am Primärsystem ein Mehrrechner-DBS vor, dann verlangt die Verwendung eines einzigen Log-Stroms jedoch das Mischen der Log-Daten vor der Übertragung, was einen hohen Zusatzaufwand einführen kann (der im Falle von Tandems RDF in Kauf genommen wird**). Für Shared-Nothing besteht die Alternative, daß jeder Rechner die Log-Daten seiner DB-Partition in einem eigenen Log-Strom an das Backup-System sendet, wobei dort dann ebenfalls mehrere Log-Dateien geführt werden. Der Vorteil dieses Ansatzes ist zum einen, daß das Mischen der Log-Daten und damit ein potentieller Engpaß entfällt. Zudem kann eine höhere Übertragungsbandbreite zwischen Primär- und Backup-System genutzt werden. Ein Problem bei Verwendung mehrerer Log-Ströme ist jedoch, daß es im Backup-Sy-

* Hierzu können die Log-Daten im Primärsystem in einem Puffer gesammelt werden, der dann übertragen wird, wenn er gefüllt ist oder das Commit einer 2-sicheren Transaktion ansteht. Damit werden i.a. Log-Daten mehrerer Transaktionen gebündelt übertragen.

** Im Falle eines Mehrrechner-DBS vom Typ "Shared Disk" (DB-Sharing) ist ein solches Mischen lokaler Log-Daten ohnehin erforderlich (s. Kap. 13.3.4).

stem aufgrund der asynchronen Log-Übertragung nicht ohne weiteres möglich ist, einen konsistenten DB-Zustand zu erzeugen, da die Log-Informationen zu den einzelnen DB-Partitionen i.a. einen unterschiedlichen Änderungsstand reflektieren. Die Lösung dieses Problems erfordert zusätzliche Synchronisierungsmaßnahmen im Backup-System; Algorithmen dazu werden in [GPH90] vorgestellt.

9.6 Abschließende Bemerkungen

Trotz der großen Anzahl von Veröffentlichungen über replizierte Datenbanken gibt es kaum vollständige und effiziente Realisierungen. Eine Implementierung erfolgte in einigen wenigen Prototypen von Verteilten DBS [Mo84] und Verteilten Betriebssystemen (z.B. LOCUS). Das einzige kommerziell verfügbare Verteilte DBS, das unseres Wissens replizierte Datenbanken mit vollständiger Replikationstransparenz unterstützt, ist CA:DB-STAR (Kap. 19.10). Dies liegt zum einen an der hohen Implementierungskomplexität sowie den potentiell hohen Kosten zur Aktualisierung der Replikate. Zum anderen verlangen nur wenige geographisch verteilte Anwendungen die vollständige Übereinstimmung sämtlicher Kopien. Aus Praxissicht bedeutender sind daher beschränktere Formen der Replikation wie Primary-Copy-Ansätze mit Lesen veralteter Daten, Schnappschüsse oder DB-Kopien zur Katastrophen-Recovery. Allerdings unterstützen mehrere kommerzielle Systeme die replizierte Speicherung von Katalogdaten, für die meist nur eine geringe Änderungshäufigkeit besteht. In diesem Fall können natürlich die vorgestellten Synchronisationsverfahren ebenfalls eingesetzt werden.

Übungsaufgaben

Aufgabe 9-1: Netzwerk-Partitionierungen

Bewerten Sie der vorgestellten Verfahren zur Synchronisation auf replizierten Datenbanken hinsichtlich Verfügbarkeit nach Netzwerk-Partitionierungen.

Aufgabe 9-2: Datenbankverteilung

Auf Objekt A seien an den Rechnern R1 bis R4 folgende Zugriffshäufigkeiten (pro Sekunde) gegeben:

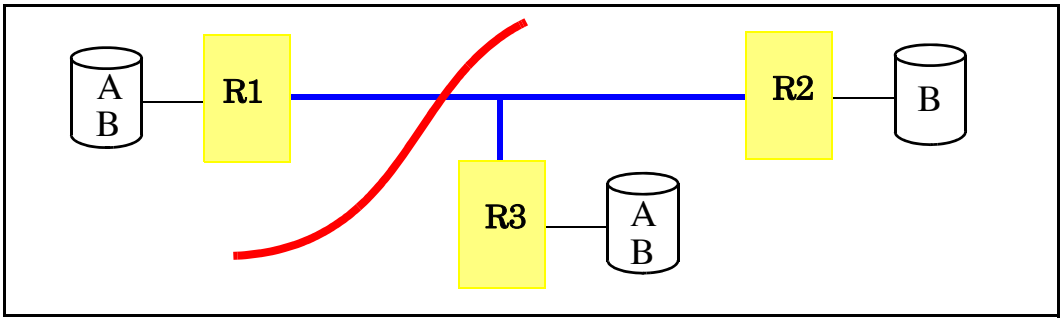
	R1	R2	R3	R4
Lesezugriffe	30	40	50	20
Änderungszugriffe	10	20	-	-

An welchen Knoten empfiehlt sich die Speicherung einer Kopie von A, wenn zur Synchronisation ein ROWA-Protokoll verwendet wird? Es soll dabei nur die Minimierung der Kommunikationskosten erfolgen (keine Lastbalancierung); jede Referenz soll eine eigene Transaktion bilden*.

Aufgabe 9-3: Kommunikationsaufwand (ROWA, Primary-Copy)

Auf Objekt A seien die Zugriffshäufigkeiten wie in der vorherigen Aufgabe gegeben. Ferner sei A an den Knoten R1 bis R3 repliziert gespeichert. Welche Nachrichtenhäufigkeiten entstehen für diese Last und Datenverteilung mit dem ROWA-Protokoll sowie den Primary-Copy-Verfahren, wenn jede Referenz eine eigene Transaktion bildet?

Aufgabe 9-4: Voting-Verfahren



Für die gezeigte Datenverteilung seien für Objekt A die Stimmenverteilung <2, -, 1> mit den Quoren r=2, w=2 und für Objekt B die Stimmenverteilung <1, 3, 1> sowie r=2, w=4 gegeben. Aufgrund eines Fehlers im Netzwerk sei ferner die gezeigte Partitionierung des Systems in zwei Teile eingetreten (Partition P1 bestehend aus R1, Partition P2 mit R2 und R3). In welchen Partitionen können die folgenden drei Transaktionen noch bearbeitet werden (R (x) bzw. W (x) bezeichne den Lese- bzw. Schreibzugriff auf Objekt x):

T1: R (A)

T2: R (B), W (B)

T3: R (A), R (B)

Durch welche Wahl der Stimmenverteilung und Quoren könnte die Transaktion

T4: W (A), W (B)

noch bearbeitet werden ?

Aufgabe 9-5: Katastrophen-Recovery

Das Führen einer DB-Kopie in einem Backup-System zur Katastrophen-Recovery (Kap. 9.5) weist Ähnlichkeiten mit einem Primary-Copy-Ansatz zur Wartung der Replikation (Kap. 9.2) auf. Wo liegen die Unterschiede ? Welche Unterschiede bestehen zur Schnappschuß-Replikation?

---

* Dies ist eine Worst-Case-Annahme hinsichtlich der Nachrichtenhäufigkeit (warum?), die aus Einfachheitsgründen getroffen werden soll.

Teil III
Heterogene Datenbanken

Dieser Teil behandelt die wichtigsten Ansätze zur Unterstützung heterogener Datenbanken. Im einleitenden Kapitel 10 werden zunächst die neuen Anforderungen diskutiert, insbesondere Knotenautonomie und Heterogenität. Ferner wird die Rolle von Standards zur Unterstützung von Interoperabilität und Portabilität erörtert. Die beiden folgenden Kapitel behandeln die wesentlichsten Architekturklassen zur Behandlung von heterogenen Datenbanken: Verteilte Transaktionssysteme (Kap. 11) sowie föderative Mehrrechner-DBS (Kap. 12). Bisherige Implementierungen verfolgen meist den ersten Ansatz, wofür auch eine Reihe von Standards vorliegen, auf die in Kap. 11 ebenfalls eingegangen wird. Föderative DBS streben eine weitergehende Funktionalität an und werden vor allem in der Forschung untersucht.

10 Autonomie und Heterogenität

Verteilte Datenbanksysteme (sowie andere Typen integrierter Mehrrechner-Datenbanksysteme, s. Kap. 3.2) unterstützen die verteilte Verarbeitung auf einer einzigen logischen Datenbank, die durch ein entsprechendes (globales) konzeptionelles Schema beschrieben wird. In der Praxis zeigt sich jedoch daneben zunehmend die Notwendigkeit, in koordinierter Weise auf mehrere selbständige Datenbanken zuzugreifen, die jeweils von einem eigenem DBVS verwaltet werden. Dies ist z.B. in zahlreichen Unternehmen und Institutionen der Fall, in denen meist mehrere Datenbanken in unkoordinierter Weise geführt werden*. Die verschiedenen Datenbanken, welche wir hier als *Lokale DBS (LDBS)* bezeichnen, sind in der Regel unabhängig voneinander entworfen worden und daher auch heterogen (s.u.); sie enthalten jedoch häufig inhaltlich verwandte Informationen. Der isolierte Einsatz der LDBS bringt gravierende Nachteile mit sich, da Benutzer jeweils nur mit einer Datenbank arbeiten können. Damit ist die Verteilung der Daten für die Benutzer voll sichtbar (keinerlei Verteilungstransparenz), und die Datenkonsistenz über mehrere Datenbanken hinweg wird systemseitig nicht überwacht. Sind z.B. einzelne Informationen in verschiedenen Datenbanken repliziert, ist diese Redundanz seitens der Benutzer zu warten.

Beispiel 10-1

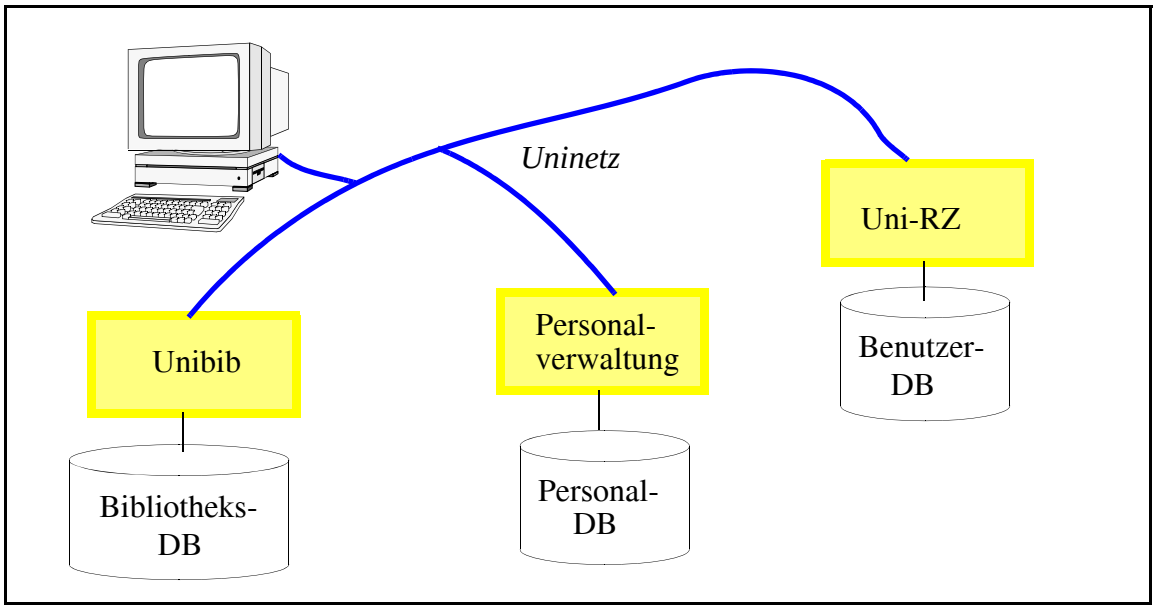
In einer Universitätsumgebung finden sich meist mehrere, unabhängig voneinander betriebene Datenbanken (Abb. 10-1). Bei der Personalverwaltung wird z.B. eine Datenbank über alle Universitätsbediensteten geführt; personenbezogene Daten werden daneben etwa in der Zentralbibliothek sowie im Rechenzentrum für die jeweiligen Benutzer gespeichert. Die unabhängige Verwaltung dieser verwandten Daten führt zu unkontrollierter Redundanz sowie den damit verbundenen Änderungsproblemen. So müssen z.B. Änderungen der Adresse ggf. in allen drei Datenbanken manuell nachgeführt werden. Weiterhin ist es nicht möglich, in einer Transaktion auf die verschiedenen Datenbanken zuzu-

---

* Eine Verschärfung dieser Problematik ergibt sich durch die zunehmende Anzahl von Firmen-Zusammenschlüssen, die auch den Zugriff auf vormals getrennte Datenbestände verlangen.

greifen, z.B. um eine Person beim Ausscheiden aus der Universität aus allen Datenbanken zu löschen

Abb. 10-1: Heterogene Datenbanken in einer Universitätsumgebung



Der entscheidende Unterschied zu den integrierten Mehrrechner-DBS liegt in der notwendigen Unterstützung mehrerer unabhängiger und heterogener Datenbanken. Eine fundamentale Anforderung ist dabei, daß innerhalb einer Transaktion auf mehrere solcher Datenbanken zugegriffen werden kann - unter Wahrung der ACID-Eigenschaften. Wünschenswert ist darüber hinaus, daß selbst innerhalb einer DB-Operation Daten mehrerer Datenbanken bearbeitet werden können, z.B. um globale Join-Berechnungen vorzunehmen. Die Zugriffe auf die einzelnen Datenbanken sollten ferner über eine einheitliche Schnittstelle möglich sein, auch wenn unterschiedliche DBVS an der Verarbeitung beteiligt sind. Dies verlangt die Bereitstellung eines gemeinsamen *Application Programming Interface (API)* zwischen Anwendungen und Datenbanksystemen.

Die Erfüllung dieser Anforderungen wird durch zwei wesentliche Randbedingungen erschwert, nämlich die zu wahrende Knotenautonomie sowie die vorliegende Heterogenität. Diese beiden Aspekte sollen im folgenden genauer diskutiert werden. Danach werden die Rolle von Standards erörtert und das Konzept der "Middleware" eingeführt (Kap. 10.3). Abschließend erfolgt eine Kurzcharakterisierung der wesentlichen Realisierungsalternativen zur Unterstützung heterogener Datenbanken, welche in den weiteren Kapiteln behandelt werden.

10.1 Knotenautonomie

Der Zugriff auf mehrere Datenbanken innerhalb einer Transaktion setzt ein Mindestmaß an Kooperationsbereitschaft der beteiligten Knoten und ihrer DBVS voraus, u.a. um die ACID-Eigenschaften wahren zu können. Daraus folgt unweigerlich eine reduzierte Unabhängigkeit oder Autonomie der Rechner (Knotenautonomie, node autonomy) im Vergleich zu ihrer isolierten Nutzung. Verschiedene Arten der Knotenautonomie lassen sich hierbei unterscheiden (siehe auch [SL90]):

- *Entwurfsautonomie (design autonomy)*
  Jeder Knoten entscheidet selbständig darüber, welcher Anwendungsausschnitt in der Datenbank repräsentiert sein soll (Miniwelt) und wie der logische DB-Entwurf (Namenswahl, Datenrepräsentation, Integritätsbedingungen etc.) und physische DB-Entwurf (Speicherungsstrukturen, Indexwahl etc.) dazu aussehen sollen.
- *Ausführungsautonomie (execution autonomy*)
  Lokale Transaktionen, die auf ausschließlich lokalen Daten arbeiten, sollten unabhängig von anderen Rechnern bearbeitbar sein und durch externe Transaktionen möglichst nicht beeinträchtigt werden. Sub-Transaktionen externer Transaktionen sollten weitgehend wie lokale Transaktionen bearbeitet werden.
- *Kooperationsautonomie (communication + association autonomy)*
  Jeder Knoten kann darüber entscheiden, welche Bereiche der lokalen Datenbank für externe Benutzer (Transaktionen) referenzierbar sind und welche Operationen darauf ausführbar sind. Weiterhin entscheiden die LDBS selbständig darüber, in welchem Umfang eine Kooperation mit anderen LDBS unterstützt wird, z.B. zur globalen Anfragebearbeitung oder Transaktionsverwaltung.
- *Unabhängigkeit hinsichtlich Wahl des DBVS*
  Die Entscheidung, welches DBVS zur Verwaltung einer Datenbank eingesetzt wird, soll unabhängig von anderen Knoten getroffen werden können. Damit wird zugleich die Wahl des Datenmodells (relational, hierarchisch, netzwerkartig, objekt-orientiert etc.), der Anfragesprache (SQL-Dialekt, DL/1...) sowie der internen Realisierung (Transaktionsverwaltung, Query-Optimierung, Zugriffspfadtypen etc.) festgelegt.
- *Unabhängigkeit hinsichtlich Wahl der Ablaufumgebung*
  Jeder Knoten kann die Wahl der Hardware, von Betriebssystem sowie sonstiger Software (TP-Monitor, Kommunikationsprotokolle) auf seine lokalen Bedürfnisse abstimmen.

Kooperation und Knotenautonomie sind jedoch keine Alles-oder-Nichts-Eigenschaften, sondern werden von verschiedenen Ansätzen in unterschiedlichem Ausmaß unterstützt. Selbst bei der Realisierung von Verteilten DBS wurde bereits versucht, der Forderung nach Knotenautonomie Rechnung zu tragen, um eine akzeptable Unterstützung geographisch verteilter Systeme zu erreichen. Daher wurden zentralisierte Lösungansätze etwa zur Katalogverwaltung oder zur Synchronisation als inakzeptabel eingestuft. Auch bei der Namensverwaltung wurde die Wahrung einer hohen Autonomie angestrebt (Kap. 4.4). Auf der anderen Seite unterstützen Verteilte DBS nahezu keine Entwurfsautonomie (bis auf den physi-

schen DB-Entwurf). Zur Gewährleistung von Verteilungstransparenz muß ferner jede DB-Operation an allen beteiligten DBS gestartet werden können*. Wie wir sehen werden, wird auch bei der Unterstützung heterogener Datenbanken Knotenautonomie in unterschiedlichem Umfang erreicht. Die jeweilige Anwendung entscheidet über den erforderlichen Grad an Knotenautonomie und Kooperation und damit über die Eignung verschiedener Realisierungsalternativen.

10.2 Heterogenität

Die diskutierten Formen der Knotenautonomie führen meist zu unterschiedlichen Arten der Heterogenität, welche die Nutzung der einzelnen Datenbanken erschweren. Vor allem drei Arten der Heterogenität sind dabei zu unterscheiden:

- *Heterogenität bezüglich der beteiligten DBVS*
  Aufgrund der Unabhängigkeit hinsichtlich der Wahl des DBVS (s.o.) können sich die LDBS hinsichtlich Hersteller, Version, Datenmodell, Anfragesprache sowie interner Realisierung unterscheiden.
- *Heterogenität in der Ablaufumgebung*
  Aufgrund der Unabhängigkeit hinsichtlich der Wahl der Ablaufumgebung können die einzelnen Rechner Unterschiede bei Hardware (Prozessortyp, Instruktionsumfang, Zeichensätze etc.), Betriebssystemen, TP-Monitoren sowie Kommunikationsprotokollen aufweisen.
- *Heterogenität im DB-Inhalt (semantische Heterogenität)*
  Semantische Heterogenität ist eine Folge der Entwurfsautonomie. Sie kann sich in vielfacher Weise zeigen, insbesondere bei der Benennung und Repräsentation von DB-Objekten. Gleiche oder verwandte Daten können so verschiedene Namen erhalten und in unterschiedlichster Weise repräsentiert werden. Umgekehrt können DB-Objekte gleichen Namens verschiedene Objekte der realen Welt verkörpern. Die Datenwerte verschiedener Datenbanken können zudem widersprüchlich (z.B. aufgrund von Eingabefehlern) oder unvollständig sein. Eine genauere Klassifikation zur semantischen Heterogenität erfolgt in Kap. 12.2.

10.3 Die Rolle von Standards

Die gemeinsame Nutzung heterogener Rechnersysteme setzt zwingend den Einsatz von Standards voraus, um ein ausreichendes Maß an Interoperabilität zu erreichen. Unter *Interoperabilität* versteht man dabei allgemein die Kommunikations- und Kooperationsfähigkeit, so daß ein Programm eines Rechners, auf Programme oder Daten eines anderen Rechners zugreifen kann. Hierzu ist es erforderlich, daß die beteiligten Kommunikationspartner ein gemeinsames Protokoll verwenden, in dem die Formate und Reihenfolgen von Nachrichten sowie deren Bedeutung festgelegt sind. Eine weitere Zielsetzung von Standards ist die Defini-

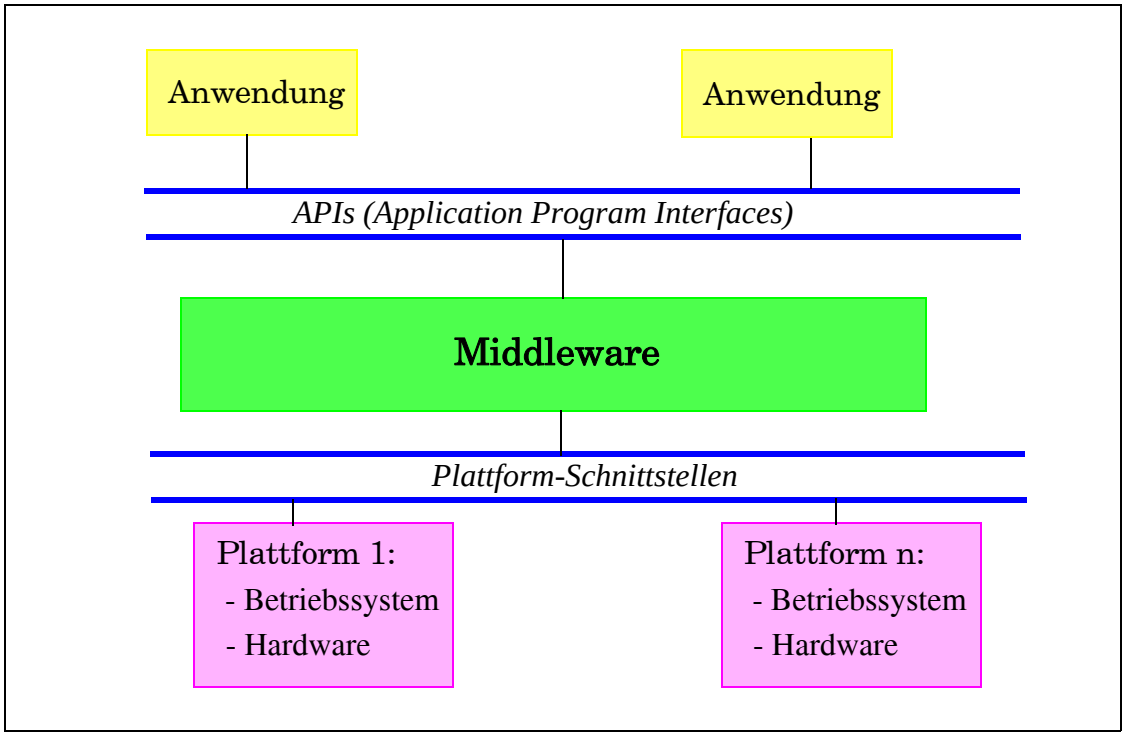
* Eine besonders enge Kooperationsnotwendigkeit wird eingeführt, wenn Relationen über mehrere Knoten hinweg partitioniert oder repliziert gespeichert werden, da dann bereits Operationen auf einer Relation verteilt auszuführen sind.

tion einheitlicher Schnittstellen zur Unterstützung der *Portabilität* von Programmen (Übertragbarkeit auf unterschiedliche Hardware- bzw. Betriebssystem-Plattformen). Um eine weitestgehende Interoperabilität und Portabilität erreichen zu können, sollten die eingesetzten Protokolle und Schnittstellen von möglichst vielen Herstellern unterstützt werden. Systeme, welche solch allgemein akzeptierten und öffentlich zugänglichen Standards folgen, werden auch als *"offene" Systeme* bezeichnet [Se93]*.

Die Menge der Systemdienste zur Unterstützung von Interoperabilität und Portabilität in verteilten und heterogenen Umgebungen wird neuerdings oft als *Middleware* bezeichnet [Be93a]. Der Name resultiert daher, daß diese Dienste (Services) "in der Mitte" zwischen Anwendungen und Betriebssystem anzusiedeln sind (Abb. 10-2). Gegenüber Anwendungen bieten die Middleware-Services eine Reihe von Programmierschnittstellen oder APIs (Application Program Interfaces). Damit werden den Anwendungen mächtige Funktionen angeboten, die die darunterliegenden Plattformen (Hardware, Betriebssystem, Kommunikationsnetzwerk) vollständig verbergen und auch die Behandlung von Verteilung und Heterogenität vereinfachen. Zugleich wird bei Verwendung standardisierter APIs die Portabilität von Anwendungen unterstützt, d.h. ihre Lauffähigkeit auf unterschiedlichen Plattformen. Dies setzt voraus, daß die Middleware-Komponenten selbst auf verschiedenen Plattformen verfügbar sind und eine Interoperabilität zwischen ihnen unterstützt wird.

* "Offen" ist derzeit einer der am meisten gebrauchten und mißbrauchten Begriffe, für den es keine allgemein akzeptierte Definition gibt. Ähnliches gilt allerdings zum Teil auch für andere Schlagwörter, z.B. Interoperabilität oder Middleware.

Abb. 10-2: Stellung von Middleware [Be93a]



Beispiele für Middleware sind u.a. DBVS, Präsentations-Services, Kommunikations-Services, Transaktions-Manager, Name-Service, Dateiverwaltung oder Authentisierungsdienst. Der Middleware zuzurechnen sind auch TP-Monitore, wobei diese meist mehrere Services bieten und auf eine bestimmte Anwendungsklasse, nämlich OLTP (Online Transaction Processing), zugeschnitten sind*. Ältere TP-Monitore realisierten sämtliche Funktionen selbst und nutzten ausschließlich plattformspezifische Funktionen des Betriebssystems (inklusive des Basiskommunikationssystems). Neuere Entwicklungen dagegen basieren selbst wiederum auf standardisierten Middleware-Services z.B. zur Kommunikation (Bsp. OSF DCE) oder Transaktionsverwaltung. Damit werden für diese TP-Monitore Portabilität sowie Realisierungsaufwand verbessert.

Die praktische Umsetzbarkeit des abstrakten Middleware-Konzepts wird derzeit v.a. dadurch beschränkt, daß zahlreiche Systemarchitekturen, Standards bzw. De-facto-Standards vorliegen, die oft nur teilweise miteinander verträglich sind bzw. gar miteinander konkurrieren. Die einzelnen Ansätze stammen von internationalen und nationalen Gremien und Vereinigungen (z.B. ISO, ANSI, DIN, IEEE, ECMA), Hersteller-Konsortien (X/Open, OSF, OMG etc.) sowie Firmen-Allianzen und Einzel-Herstellern. So haben viele größere Unternehmen eigene Middleware-Architekturen definiert, z.B. IBM mit SAA (Systems Application Architecture) und

* Solche spezialisierteren Middleware-Komponenten werden auch als *Frameworks* bezeichnet [Be93a].

OE (Open Enterprise), DEC mit NAS (Network Application Support) oder Software AG mit ISA (Integrated Software Architecture), in die zum Teil bestehende Standards aufgenommen wurden und fehlende Teile durch eigene Komponenten und Schnittstellen-Definitionen ergänzt wurden.

Daß von Gremien wie der ISO verabschiedete Standards keineswegs immer herstellerspezifische Lösungen ablösen können, zeigt sich bezüglich der Kommunikationsarchitektur am Beispiel von OSI (Open Systems Interconnection). Dessen Verbreitung ist aufgrund der Marktdominanz von IBMs SNA sowie TCP/IP immer noch recht begrenzt. Die Folge davon ist u.a., daß in vielen Anwendungen weiterhin verschiedene Netzwerke und Kommunikationsprotokolle im Einsatz sind und zu ihrer Überbrückung Gateways eingesetzt werden müssen. Eine bessere Marktdurchdringung wurde von der ISO mit dem SQL-Standard erreicht, da sich dessen Definition eng an bestehenden Produkten orientierte. Dennoch weichen die meisten SQL-Implementierungen vom Standard ab, z.B. um zusätzliche Funktionen zu realisieren.

Große Bedeutung hinsichtlich der Standardisierung nimmt derzeit die Hersteller-Vereinigung *X/Open* ein, deren Ziel es ist, Industrie-Standards auf Basis allgemein akzeptierter Schnittstellen durchzusetzen. Die Definition einer einheitlichen Anwendungsumgebung (Common Application Environment, CAE) umfaßt APIs für einen Vielzahl von Diensten; die Festlegungen sind in einem "Portability Guide" zusammengestellt (derzeit X/Open Portability Guide 4, XPG4). Große Verbreitung erlangt haben auch die standardisierten Middleware-Dienste Motif (Benutzeroberfläche), DCE (Distributed Computing Environment) und DME (Distributed Management Environment) der *OSF* (Open Software Foundation) [Sc93]. So sind die DCE-Dienste wie Fernaufruf (Remote Procedure Call, RPC), Datendarstellungs-, Zeit- und Namensdienste u.a. bereits auf einer Vielzahl von Plattformen verfügbar und erleichtern damit Interoperabilität und Portabilität der nutzenden Software-Komponenten.

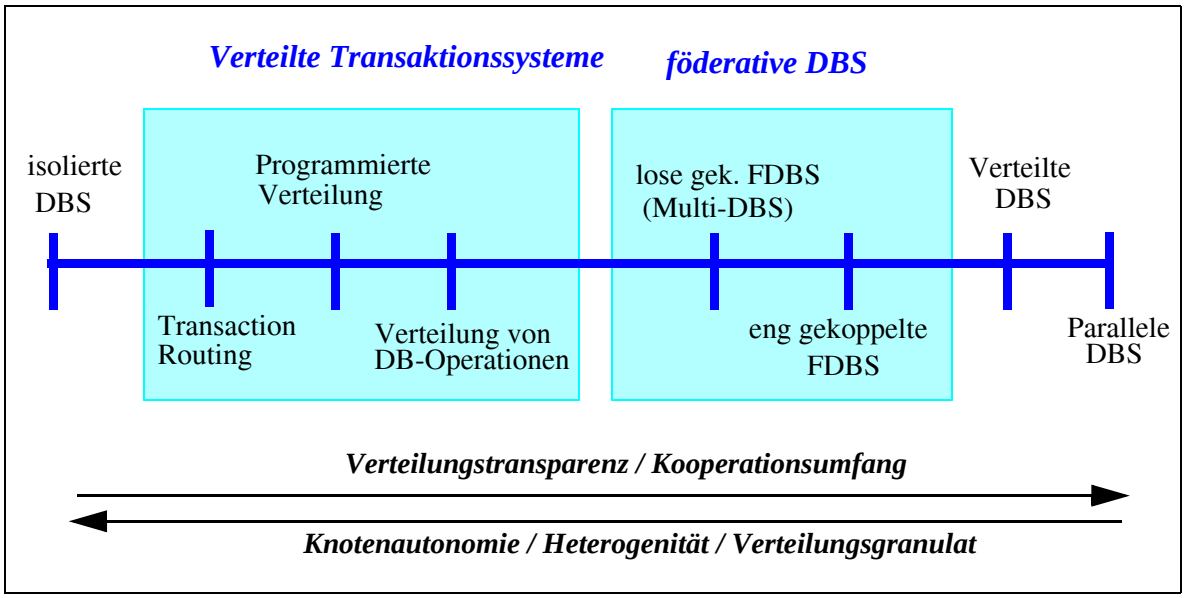
Wir können hier (Kap. 11) nur auf einige Standardisierungen eingehen, die im engeren Zusammenhang mit unserer Problemstellung stehen (X/Open DTP, MIA STDL, ISO RDA, ODBC, IBM DRDA). Einen guten Überblick über die sonstigen Standardisierungsaktivitäten bietet [Se93].

10.4 Realisierungsansätze

Zur Unterstützung heterogener Datenbanken bestehen mehrere Realisierungsalternativen, welche die genannten Anforderungen in unterschiedlichem Ausmaß erfüllen. Generell nehmen wir dazu an, daß pro Knoten ein eigenes LDBS mit einer privaten Datenbank vorliegt, d.h., die Einheit der Datenverteilung sind ganze (logische) Datenbanken. Die Alternativen unterscheiden sich darin, welche Verte-

lungseinheiten hinsichtlich der Transaktionsverarbeitung unterstützt werden. Wir gruppieren die einzelnen Ansätze in zwei Kategorien, welche in den beiden folgenden Kapiteln behandelt werden: Verteilte Transaktionssysteme (Kap. 11) sowie föderative DBS (Kap. 12). Bei Verteilten Transaktionssystemen wird die verteilte Transaktionsverarbeitung weitgehend außerhalb der LDBS realisiert, i.a. unter Kontrolle von TP-Monitoren. Kennzeichnende Eigenschaft dieser Ansätze ist, daß ganze DBS-Anweisungen (DB-Operationen) das feinstmögliche Verteilungsgranulat bilden. Bei föderativen DBS dagegen werden rechnerübergreifende DB-Operationen unterstützt, so daß Teiloperationen als Verteileinheiten möglich werden. Dies setzt eine engere Zusammenarbeit der beteiligten LDBS voraus.

Abb. 10-3: Lösungsspektrum zur Unterstützung heterogener Datenbanken

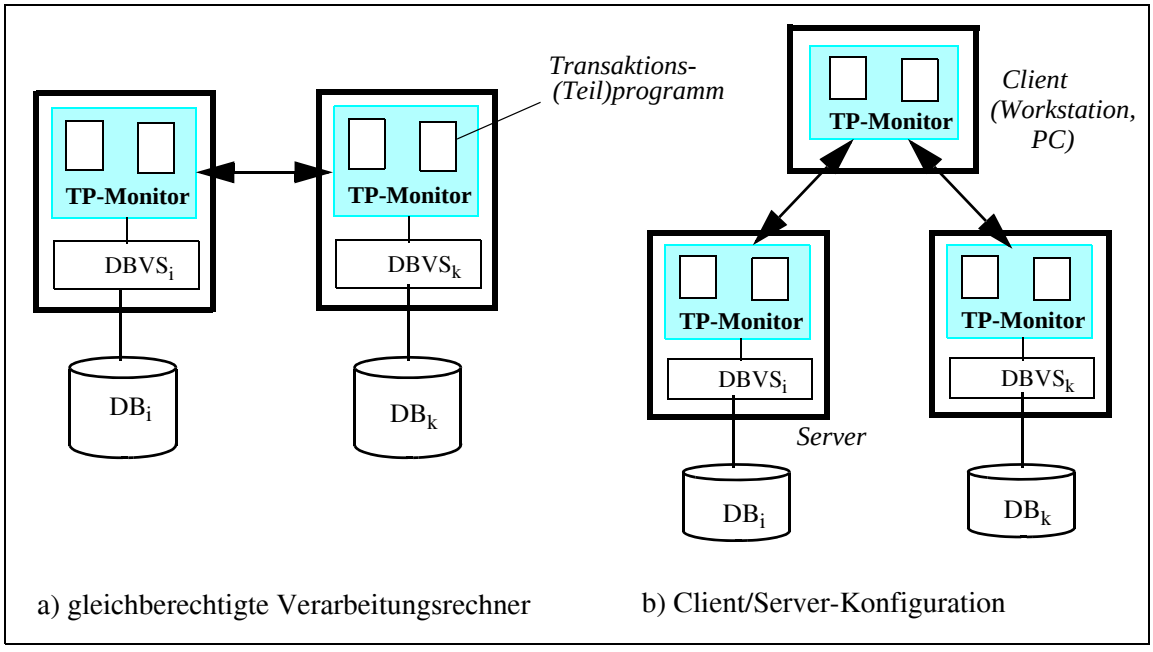


In beiden Fällen existieren mehrere Ausprägungen, wobei sich insgesamt das in Abb. 10-3 gezeigte Lösungsspektrum ergibt. Wie durch die Pfeile angedeutet, nimmt dabei das jeweilige Verteilungsgranulat bezüglich der Verarbeitung von links nach rechts ab und damit der notwendige Kooperationsumfang zwischen den LDBS zu. Mit zunehmendem Kooperationsumfang steigt zugleich der erreichbare Grad an Verteilungstransparenz für den Anwender, umgekehrt sinkt das mögliche Ausmaß an Heterogenität und Knotenautonomie. Das Lösungsspektrum wird begrenzt von dem Extrem "Isolierte DBS" einerseits (keinerlei Kooperation und Verteilungstransparenz, maximale Heterogenität und Autonomie) und den Ansätzen "Verteilte DBS" sowie "Parallele DBS" andererseits (maximale Verteilungstransparenz, geringes Maß an Heterogenität und Autonomie). Die dazwischenliegenden Alternativen sind zur Unterstützung heterogener Datenbanken besser geeignet und werden in den beiden folgenden Kapiteln näher vorgestellt. Bisherige Standardisierungen konzentrierten sich vor allem auf die Verteilungsansätze "Programmierte Verteilung" sowie "Verteilung von DB-Operationen" und werden im

11 Verteilte Transaktionssysteme

In Mehrrechner-Datenbanksystemen kooperieren mehrere DBVS (bzw. DBVS-Prozesse) zur Bearbeitung von DB-Operationen. Eine verteilte Transaktionsverarbeitung kann jedoch auch außerhalb der DBVS erreicht werden, i.a. unter Kontrolle von TP-Monitoren (Kap. 2.1.4). Wir sprechen in diesem Fall von ***Verteilten Transaktionssystemen***. Die verteilte Transaktionsverarbeitung kann zwischen gleichberechtigten Verarbeitungsrechnern stattfinden oder zwischen funktional unterschiedlichen Client- und Server-Rechnern (Abb. 11-1), wobei jeweils ähnliche Verteilungsalternativen bestehen.

Abb. 11-1: Einsatz verteilter Transaktionsysteme (Programmierte Verteilung)



Da sich die verteilte Transaktionsverarbeitung zunehmend in Client/Server-Systemen abspielt, diskutieren wir zunächst kurz die hierbei bestehenden Alternativen. Danach beschreiben wir die wichtigsten, auch zwischen Verarbeitungs- bzw. Server-Rechnern einsetzbaren Verteilungsformen, vor allem die programmierte

Verteilung und die Verteilung von DB-Operationen (Kap. 11.2). In Kap. 11.3 gehen wir dann auf einige Implementierungsaspekte ein, insbesondere die Transaktionsverwaltung sowie Realisierungsalternativen zur Kommunikation. Ebenfalls diskutiert wird der Einsatz von DB-Gateways zur Überbrückung unterschiedlicher Anfragesprachen. In Kap. 11.4 schließlich werden Standardisierungsansätze zur programmierten Verteilung (X/Open DTP, MIA STDL) sowie zur Verteilung von DB-Operationen (ISO RDA, SQL Access, ODBC, IDAPI, IBM DRDA) vorgestellt.

11.1 Transaktionsverarbeitung in Client/Server-Systemen

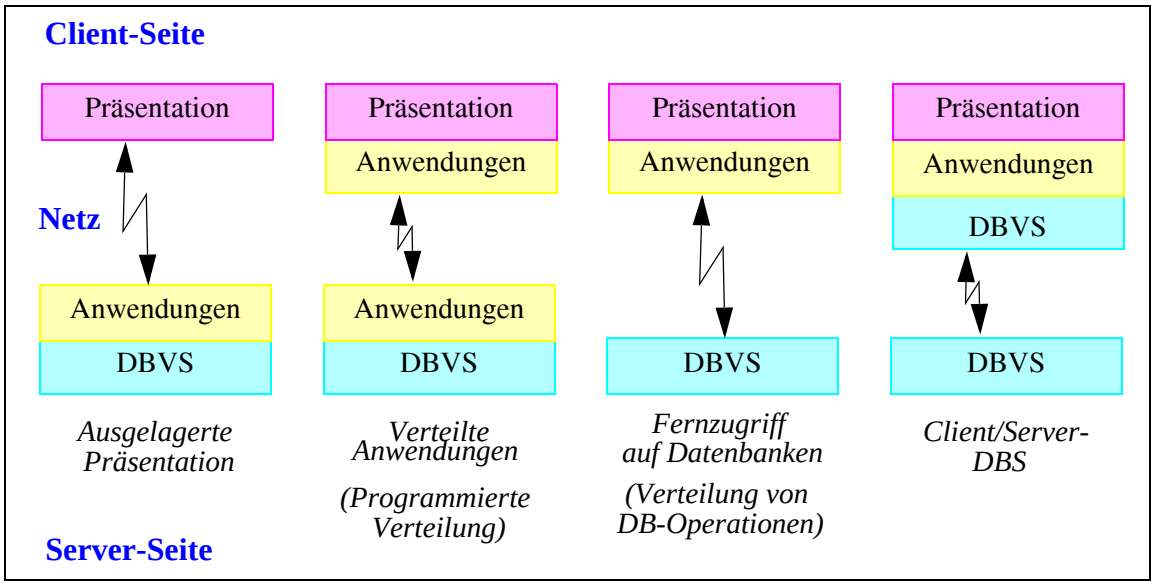
Traditionellerweise erfolgte die Transaktionsverarbeitung (Online Transaction Processing, OLTP) zentralisiert auf allgemeinen Großrechnern, auf die über einfache Terminals zugegriffen wurde. Die Datenbank- und Anwendungsverarbeitung erfolgte - unter Kontrolle von DBS und TP-Monitor - vollständig auf dem Mainframe, ebenso Präsentationsdienste zur Ein-/Ausgabe mit dem Benutzer (Maskenverwaltung, Formatkonversionen, etc.).

Solche zentralisierten Transaktionssysteme (Kap. 2.1.4) werden jedoch zunehmend durch *Client/Server-Transaktionssysteme* abgelöst. Dabei erfolgt eine Zerlegung der Funktionen in Client- und Server-Komponenten, die unterschiedlichen Rechnern zugeordnet werden können. Als Client-Rechner fungieren dabei meist leistungsfähige PCs und Workstations, die fast an jedem Arbeitsplatz verfügbar sind und die einfachen Terminals weitgehend abgelöst haben. Damit lassen sich eine Reihe wesentlicher Vorteile erreichen:

- Durch die Ausführung von Teilen der Transaktionsverarbeitung auf Client-Rechnern können die zentralen Server-Rechner erheblich entlastet werden, so daß insgesamt ein besseres Leistungsverhalten möglich wird (höherer Server-Durchsatz).
- Die Nutzung von PCs und Workstations läßt eine signifikante Verbesserung der Kosteneffektivität gegenüber dem zentralisierten Mainframe-Ansatz erwarten. Dieser Effekt kann noch verbessert werden, indem auch in den Server-Rechnern leistungsfähige Mikroprozessoren eingesetzt werden (z.B. Multiprozessoren oder Parallelrechner auf Mikroprozessorbasis).
- Die Grafikfähigkeiten von Workstations und PCs (Fenstertechniken, etc.) erlauben eine starke Verbesserung der Benutzeroberflächen.
- Es ist relativ einfach, mehrere dedizierte Server für unterschiedliche Aufgaben in die Verarbeitung einzubeziehen. Damit wird auch der Zugriff auf mehrere unabhängige DB-Server (heterogene Datenbanken) möglich.

Für die Aufteilung der zur Transaktionsverarbeitung benötigten Funktionen der Präsentation, Anwendungsprogramme (Transaktionsprogramme) und DB-Verarbeitung bestehen im wesentlichen vier Alternativen, die in Abb. 11-2 dargestellt sind. Jede dieser Verteilungsformen geht einher mit einem unterschiedlichen Aufrufgranulat zwischen Client- und Server-System:

Abb. 11-2: Aufgabenteilung in Client/Server-Transaktionssystemen



- Der erste Ansatz der *ausgelagerten Präsentation* beläßt Anwendungen und DB-Verarbeitung vollständig auf Server-Seite und führt lediglich die Präsentationsaufgaben (Realisierung der graphischen Benutzeroberfläche) im Client-System durch. Dies stellt die minimale Funktionalität dar, die client-seitig ausgeführt werden sollte; sie kann jedoch bereits eine spürbare Entlastung der Server bewirken. Das Aufrufgranulat zum Server sind wie in zentralisierten Transaktionssysteme ganze Transaktionsaufträge.
- Beim Ansatz der *verteilten Anwendungen* werden Anwendungen zum Teil auf den Client-Rechnern ausgeführt, so daß diese weitergehend als bei der ausgelagerten Präsentation genutzt werden können. Allerdings können die Client-Anwendungen nicht unmittelbar mit den Server-DBVS kommunizieren, sondern müssen Anwendungsprogramme auf den Servern starten, um die dort erreichbaren Daten zu referenzieren. Aufrufeinheit ist somit eine Anwendungsfunktion bzw. Transaktionsprogramm, so daß man auch von *programmierter Verteilung* spricht [HM90]. Der Aufruf erfolgt häufig über RPCs (remote procedure calls). Ein Vorteil des Ansatzes liegt darin, daß es einfach möglich ist, Anwendungsfunktionen verschiedener Rechner aufzurufen und somit auf mehrere Datenbanken zuzugreifen.
- Ein *Fernzugriff auf Datenbanken (Verteilung von DB-Operationen)* liegt vor, wenn Client-Anwendungen direkt DB-Operationen an Server-DBS stellen. In diesem Fall werden die Anwendungsfunktionen vollständig auf Client-Seite abgewickelt. Das Aufrufgranulat sind einzelne DB-Operationen (z.B. SQL-Anweisungen). Diese Verteilungsform ist für den Zugriff auf einen Server einfach realisierbar und wird von fast allen DBS unterstützt. Die Zusammenarbeit mit mehreren unabhängigen DB-Servern ist problematischer, findet jedoch auch zunehmend Verbreitung (s.u.).
- Den aufwendigsten Ansatz stellen *Client/Server-DBS* dar, bei denen auch auf Client-Seite eine DB-Verarbeitung erfolgt und das Client-DBVS mit den Server-DBVS kooperiert. Damit liegt nach unserer Terminologie hier bereits ein Mehrrechner-DBS-Ansatz und kein verteiltes Transaktionssystem mehr vor. Ein solcher Ansatz wird v.a. in objekt-orientierten DBS verfolgt, wobei ein DBS funktional auf Client- und Server-Rechner (Workstation/Server-DBS) aufgeteilt wird (Kap. 3.3.1). Es handelt

sich damit um integrierte Mehrrechner-DBS. Auch für föderative DBS kann ein Client/Server-DBS-Ansatz relevant sein, wenn DB-Operationen von einem Client-DBS aus zur verteilten Verarbeitung unter mehreren unabhängigen Server-DBS aufgeteilt werden. In beiden Fällen stellen Teile von DB-Operationen (bzw. Satz- oder Seitenanforderungen) die Aufrufgranulate dar.

Da der erste Ansatz ganze Transaktionsaufträge als Verteileinheiten benutzt und der letzte zu den Mehrrechner-DBS zählt, bestehen im wesentlichen die Alternativen der programmierten Verteilung und der Verteilung von DB-Operationen zur verteilten Transaktionsausführung. Beide Ansätze gestatten den Zugriff auf mehrere heterogene Datenbanken in einer Transaktion und werden im nächsten Abschnitt näher untersucht.

Zuvor sei jedoch noch auf einige Probleme von Client/Server-Transaktionssystemen hingewiesen. Zunächst ergeben sich natürlich Schwierigkeiten durch die Notwendigkeit der Interoperabilität mit mehreren heterogenen DBVS und der Integration bestehender Anwendungen und Datenbanken. Darauf wird im folgenden noch näher eingegangen. Einen Nachteil gegenüber zentralisierten Transaktionssystemen stellt die weitaus komplexere Administration dar. So sind meist sehr viele Client-Rechner zu verwalten, die jeweils eigene System- und Anwendungs-Software ausführen, die auf einem aktuellen und miteinander verträglichen Stand zu halten sind. Die Client-Rechner selbst sind als inhärent unzuverlässig einzustufen, da sie z.B. vom jeweiligen Benutzer abgeschaltet werden können u.ä. Daher sollten kritische Daten (Log-Daten, nicht-private Datenbanken) und Funktionen (Commit-Koordinierung) nur auf zuverlässigen Server-Rechnern vorgehalten werden. Entscheidend für die Zuverlässigkeit des gesamten Systems ist die Unterstützung der transaktionsbasierten Verarbeitung, da sie eine weitgehende Fehlerisolation Anwendungen gegenüber ermöglicht und die Konsistenz der Daten auch im Fehlerfall gewahrt bleibt.

11.2 Alternativen zur verteilten Transaktionsverarbeitung

Wie erwähnt kann eine verteilte Transaktionsverarbeitung in Client/Server-Systemen, aber auch zwischen gleichberechtigten Verarbeitungsrechnern (Servern) erfolgen. In letzterem Fall ergeben sich im wesentlichen die gleichen Verteilungsalternativen, insbesondere die programmierte Verteilung sowie die Verteilung von DB-Operationen. Daneben können auch ganze Transaktionsaufträge zwischen Rechnern verschickt werden, wobei man dann von einem *Transaction Routing* spricht [HM90]. Diese drei Verteilungsformen sollen im folgenden im Hinblick auf die Unterstützung heterogener Datenbanken näher diskutiert werden, wobei der Ansatz des Transaction Routing nur der Vollständigkeit wegen berücksichtigt wird. Wir unterstellen dabei, daß in jedem Rechner ein TP-Monitor vorliegt, der den Anwendungen Funktionen (APIs) zur Kommunikation und Transaktionsverwaltung bereitstellt sowie zum Zwecke der verteilten Transaktionsbearbeitung

mit den TP-Monitoren anderer Rechner kooperiert*. Der TP-Monitor soll ferner die Lokalisierung von entfernten Transaktionsprogrammen und DBS durchführen und damit Ortstransparenz unterstützen.

11.2.1 Transaction Routing

In diesem Fall werden ganze Transaktionsaufträge zwischen den Verarbeitungsrechnern bzw. zwischen Client- und Server-Rechner verteilt. Wenn nur diese Verteilform unterstützt wird, also keine weitere Verteilung mit feinerem Verarbeitungsgranulat erfolgt, muß eine Transaktion vollkommen auf einem Rechner ausgeführt werden, und es können somit nur die lokal erreichbaren Datenbanken referenziert werden. Diese Verteilform setzt daher voraus, daß jedes Transaktionsprogramm nur an einem Rechner ausführbar ist, wobei der TP-Monitor die Programmzuordnung kennt und somit einen Auftrag an den Rechner weiterleiten kann, wo das entsprechende Programm vorliegt. Dennoch ist diese Verteilform allein nicht ausreichend, da keine echt verteilte Transaktionsausführung erfolgt. Transaction Routing wird u.a. durch die TP-Monitore CICS und IMS-TM (MSC-Komponente: Multiple Systems Coupling) von IBM unterstützt [Wi89, SUW82].

11.2.2 Programmierte Verteilung (Verteilte Anwendungen)

Hierbei wird die Verteilung auf Ebene der Anwendungsprogramme realisiert, indem von einem Client-Rechner aus externe Anwendungsprogramme aufgerufen werden, um auf dort vorliegende Datenbanken zuzugreifen (Abb. 11-1). Das Verteilungsgranulat "Teilprogramm" bietet eine hervorragende Unterstützung heterogener DBS sowie einer hohen Knotenautonomie. Denn an jedem Knoten kann ein unabhängiges LDBS vorliegen, wobei keine direkte Kooperation zwischen den LDBS stattfindet. Der Datenbankaufbau sowie die Anfragesprache eines LDBS sind nur für lokale Anwendungen sichtbar und damit völlig transparent für externe Benutzer. Auch die DB-Administration ist relativ einfach, da die Vergabe von Zugriffsrechten an externe Benutzer sich auf lokale Teilprogramme bezieht und nicht auf einzelne Datenbankobjekte.

Beispiel 11-1

Die Überweisung eines Geldbetrags zwischen Konten zweier Banken B1 und B2 könnte stark vereinfacht durch folgendes Programm realisiert werden:

```
Eingabedaten lesen (Konten K1, K2; Banken B1, B2; Betrag Delta)
BEGIN-Transaction;
  CALL Debit (B1, K1, Delta, ...);
  CALL Credit (B2, K2, Delta, ...);
```

* Anstelle eines TP-Monitors können auch andere Middleware-Komponenten zur Erfüllung dieser Aufgaben verwendet werden.

```
        COMMIT-Transaction;
        Ausgabenachricht ausgeben;
```

Die Überweisung wird also durch Aufruf der zwei Teilprogramme Debit (Abbuchung) und Credit (Zubuchung) realisiert, die auf unterschiedlichen Rechnern ausgeführt werden. Für den Nutzer dieser Funktionen sind Ort, Aufbau und Anfragesprache der dahinterstehenden Datenbanken transparent.

Jedes an einer derartigen Transaktionsbearbeitung beteiligte Teilprogramm kann nur lokale Daten eines Rechners referenzieren, wenn keine weitere Verteilung durch die DBS erfolgt. Ortstransparenz läßt sich dabei erzielen, wenn der TP-Monitor eine Name-Server-Funktion zur Lokalisierung der Programme bietet. Verteilungstransparenz kann für den Anwendungsprogrammierer dagegen i.a. nicht vollständig erreicht werden, da er die verfügbaren Teilprogramme (Anwendungsfunktionen) anderer Rechner kennen muß, um sie korrekt aufzurufen. Gegebenenfalls sind zur Realisierung einer verteilten Anwendung auch eigens Teilprogramme für entfernte Datenbanken zu entwickeln, wobei dann die jeweilige Anfragesprache und Schemainformationen benutzt werden müssen.

Die meisten TP-Monitore unterstützen die programmierte Verteilung (CICS [Wi89], UTM-D [Go90], Tuxedo [Fe93], Encina [Sh93, Wi93], ACMS [Tr93], Top End [Sm93] etc.). Dieser Ansatz wird sogar vielfach mit verteilter Transaktionsverarbeitung gleichgesetzt. Neben der Unterstützung einer hohen Autonomie sowie heterogener DBS liegt der Vorteil v.a. in der aus Systemsicht vergleichsweise einfachen Realisierbarkeit einer verteilten Transaktionsverarbeitung. Dazu sind v.a. ein rechnerübergreifendes Commit-Protokoll sowie die Inter-Programm-Kommunikation zu realisieren (s. Kap. 11.3).

11.2.3 Verteilung von DB-Operationen (Fernzugriff auf Datenbanken)

Bei dieser Verteilungsform kooperieren die Anwendungsprogramme direkt mit mehreren LDBS, um auf ihre Datenbanken zuzugreifen. Dies ermöglicht größere Freiheitsgrade zur Realisierung verteilter Transaktionen als bei der programmierten Verteilung. Das Verteilungsgranulat "DB-Operation" bedeutet, daß jede DB-Operation nur Daten eines Rechners referenzieren darf (keine Kooperation zwischen den DBS). Die Verteilung einzelner DB-Operationen verursacht jedoch einen höheren Kommunikationsaufwand als bei der programmierten Verteilung. Besonders aufwendig können Leseoperationen werden, die große Ergebnismengen zurückliefern. Denn diese Ergebnismengen werden seitens der Anwendung üblicherweise satzweise abgerufen (Cursor-Konzept). Muß jedes Ergebnistupel einzeln von einem anderen Rechner angefordert werden, entsteht ein extremer Kommunikationsbedarf, der i.a. nicht akzeptabel sein dürfte (s. Übungsaufgaben).

Zudem ergibt sich eine sehr komplexe Anwendungsprogrammierung, da nur eine geringe Verteilungstransparenz unterstützt wird. Insbesondere muß i.a. explizit

mit jedem an der Verarbeitung zu beteiligenden DBS eine logische Verbindung aufgebaut werden. Operationen über mehrere Datenbanken hinweg (z.B. Join-Berechnung) sind explizit auszuprogrammieren. Ferner muß der Anwendungsprogrammierer mit mehreren DB-Schemata arbeiten, so daß die semantische Heterogenität vollständig sichtbar ist. Weiterhin führt die Weitergabe von Schemaangaben an externe Knoten zu einer reduzierten Datenunabhängigkeit der Anwendungen und beeinträchtigt die Knotenautonomie. Ortstransparenz läßt sich erreichen, wenn in den DB-Operationen logische Namen verwendet werden bzw. eine Synonym-Funktion (Kap. 4.4) unterstützt wird.

Beispiel 11-2

Die Realisierung der Geldüberweisung aus dem vorhergehenden Beispiel könnte (wiederum stark vereinfacht) mit dieser Verteilungsform folgendermaßen realisiert werden:

```
Eingabedaten lesen (Konten K1, K2; Banken B1, B2; Betrag Delta)
BEGIN-Transaction;
  CONNECT  TO  R1.DB1 ...  (Verbindung mit DBVS von Bank B1)
  UPDATE  ACCOUNT SET  BALANCE = BALANCE - :Delta
    WHERE  ACCT_NO = :K1;
  CONNECT  TO  R2.DB2 ... (Verbindung mit DBVS von Bank B2)
  UPDATE  GIROKTO SET  KSTAND = KSTAND + :Delta
    WHERE  KNUMMER = :K2;
COMMIT-Transaction;
Ausgabenachricht ausgeben;
Verbindungen abbauen (DISCONNECT);
```

Vor dem Zugriff auf ein bestimmtes LDBS muß der Benutzer zunächst eine Verbindung mit ihm aufbauen und sich authentifizieren (CONNECT). Vereinfachenderweise wurde angenommen, daß beide an der Verarbeitung beteiligte LDBS SQL unterstützen. Trotzdem erkennt man an den Objektangaben der UPDATE-Anweisungen, daß der Programmierer unterschiedliche Schemaangaben berücksichtigen muß (ACCOUNT, BALANCE, ACCT_NO vs. GIROKTO, KSTAND, KNUMMER). Dies erschwert die Programmierung und reduziert die Datenunabhängigkeit gegenüber externen Schemaänderungen. Umgekehrt bewirkt aus Datenbanksicht die Weitergabe von Informationen zum DB-Aufbau an andere Rechner eine Einschränkung der Knotenautonomie, die gerade bei Bankanwendungen kaum toleriert werden dürfte.

Um den Programmierer nicht auch noch mit unterschiedlichen Anfragesprachen zu konfrontieren, ist als Mindestanforderung die Unterstützung einer gemeinsamen Anfragesprache zur Formulierung der DB-Operationen zu verlangen. Dies ist gegeben, wenn alle beteiligten LDBS homogen sind (gleiche Produktversion desselben Herstellers). Daher wurde die Verteilung einzelner DB-Operationen auch zunächst für solch homogene Fälle unterstützt. Beispiele sind CICS Function Request Shipping (Verteilung von DL/1-Operationen), UDS-D oder Sesam-DCN, wobei die Weiterleitung der Operationen zum Teil über die DBVS erfolgt. Im Falle unterschiedlicher DBVS-Hersteller bestehen jedoch in der Anfragesprache mehr oder weniger große Unterschiede, selbst wenn alle SQL unterstützen. Zur Über-

brückung solcher Unterschiede ist daher der Einsatz von DB-Gateways erforderlich (s. Kap. 11.3.3).

Abschließend sei darauf hingewiesen, daß die Unterscheidung zwischen programmierter Verteilung und Verteilung von DB-Operationen bei Unterstützung von *gespeicherten Prozeduren* ("stored procedures") durch das DBS weitgehend entfällt. In diesem Fall werden nämlich Anwendungsfunktionen im DBS verwaltet, die mit einer DB-Operation aufgerufen werden können. In Implementierungen wie Sybase (Kap. 19.6) können darüber hinaus innerhalb einer gespeicherten Prozedur Prozeduren anderer DBVS aufgerufen werden*. In der Weiterentwicklung des SQL-Standards (SQL3) ist eine Unterstützung gespeicherter Prozeduren (Persistent Storage Modules, PSM) vorgesehen, wobei für diesen Teil bereits 1995 mit einer ISO-Standardisierung zu rechnen ist. Zur Realisierung solcher Prozeduren wurde eine Vielzahl von Programmiersprachenkonstrukten im SQL-Standard aufgenommen. Daneben können jedoch auch in einer anderen Programmiersprache realisierte Prozeduren durch das DBS verwaltet und ausgeführt werden.

11.3 Realisierungsaspekte

Wir diskutieren zunächst die Realisierung der Transaktionsverwaltung und Inter-Programm-Kommunikation, wie sie in Verteilten Transaktionssystemen seitens der TP-Monitore (bzw. alternativer Middleware-Komponenten) zu unterstützen ist. Im Anschluß betrachten wir die Rolle von DB-Gateways, welche v.a. bei der Verteilung der DB-Operationen benötigt werden.

11.3.1 Transaktionsverwaltung

Die globale Transaktionsverwaltung verlangt von dem TP-Monitor vor allem die Durchführung eines verteilten Commit-Protokolls. Daneben diskutieren wir noch kurz, inwieweit die DB-Konsistenz und Isolation gewahrt bleiben.

Commit-Behandlung

Um die Alles-oder-Nichts-Eigenschaft einer verteilten Transaktion zu garantieren, müssen die TP-Monitore ein gemeinsames verteiltes Commit-Protokoll unterstützen. Die LDBS sind an dem Protokoll über die TP-Monitore indirekt beteiligt, so daß ein hierarchisches Commit-Protokoll (Kap. 7.2.3) erforderlich wird. Eine Erweiterung bei der LDBS-Realisierung betrifft damit die Notwendigkeit, eine Commit-Initiierung "von außen" zu unterstützen. Damit wird allerdings eine Verringerung der lokalen Autonomie in Kauf genommen, da der Commit-Koordinator auf

* Die Unterstützung von gespeicherten Prozeduren, Multi-Tasking sowie Kommunikationsfunktionen im DBS wird gelegentlich auch als "TP-Lite" bezeichnet, wobei das DBS einen Teil der TP-Monitor-Funktionen abdeckt [Gr93].

einem anderen Rechner residieren kann, dessen Ausfall den Zugriff auf lokale Daten möglicherweise für unbestimmte Zeit blockiert.

Die Kooperation zwischen heterogenen TP-Monitoren sowie zwischen TP-Monitoren und LDBS verlangt natürlich die Einigung auf ein gemeinsames Commit-Protokoll. De-facto-Standard ist hierzu derzeit das hierarchische Zwei-Phasen-Commit-Protokoll innerhalb des SNA-Protokolls LU 6.2 von IBM. LU (Logical Unit) 6.2, auch APPC (Advanced Program-to-Program Communication) genannt, dient innerhalb von SNA zur transaktionsgeschützten Kommunikation zwischen gleichberechtigten Programmen [Gr83]. Eine sehr ähnliche Funktionalität wie LU 6.2 wird von dem neuen OSI-Standard TP (Transaction Processing) angeboten [GR93], der im X/Open-Standard zur verteilten Transaktionsverarbeitung vorgesehen ist (s.u.). LDBS, welche die XA-Schnittstelle von X/Open unterstützen, können in das Commit-Protokoll einbezogen werden.

Ein anderer "Ansatz" zur verteilten Transaktionsverwaltung, der vielfach von Systemen verfolgt wird, die kein globales Commit-Protokoll unterstützen, besteht darin, keine verteilten Änderungstransaktionen zuzulassen bzw. Änderungen auf höchstens einen Rechner zu beschränken. In diesem Fall ist nämlich zur Sicherstellung der Atomarität kein globales Commit-Protokoll erforderlich. Globale Serialisierbarkeit wird jedoch nicht erreicht, da die Synchronisation (z.B. Sperrfreigabe) zwischen Sub-Transaktionen unkoordiniert erfolgt (s. Übungsaufgaben). Dafür sind praktisch keine Erweiterungen zur Transaktionsverwaltung erforderlich und eine hohe Autonomie der LDBS bleibt erhalten.

In Client/Server-Transaktionssystemen besteht das Problem, daß Client-Rechner wie PCs und Workstations aufgrund ihrer Unzuverlässigkeit zur Commit-Koordinierung nicht verwendet werden sollten. Denn fallen sie während der kritischen Commit-Phase aus, sind Teile der am Commit beteiligten Datenbanken auf unbestimmte Zeit blockiert. Aus diesem Grund muß bei der Commit-Operation einer Client-Anwendung die Koordinatorfunktion auf einen der Server-Rechner migrieren [Pa91]. Eine solche Funktionalität wird bereits von einigen Systemen unterstützt.

Konsistenz und Isolation

Die Unterstützung des Commit-Protokolls durch die TP-Monitore und LDBS garantiert allein noch nicht die Einhaltung der ACID-Eigenschaften für verteilte Transaktionen, sondern lediglich die globale Atomarität (A) und Dauerhaftigkeit (D). Die Sicherstellung der globalen Datenbankkonsistenz (C) würde die automatische Überwachung LDBS-übergreifender Integritätsbedingungen erfordern. Dies kann ohne DBS-Unterstützung offenbar nicht erreicht werden, so daß bei fehlender Kooperation der LDBS die globale Integritätsüberwachung durch die Anwendungen vorzunehmen ist*.

Etwas besser sieht es hinsichtlich der Isolation aus. Denn wie in [BST90] gezeigt, ist die globale Serialisierbarkeit der Transaktionsverarbeitung gewährleistet, wenn jedes der beteiligten LDBS neben dem gemeinsamen Zwei-Phasen-Commit-Protokoll ein striktes Zwei-Phasen-Sperrverfahren (lange Sperren) zur Synchronisation einsetzt. Die Auflösung globaler Deadlocks erfolgt dann am einfachsten mit einem Timeout-Verfahren. Jedoch selbst dann wird eine (einfache) Erweiterung der LDBS notwendig, wenn zur Auflösung lokaler Deadlocks ein anderes Verfahren verwendet wurde (z.B. explizite Erkennung von Deadlocks). Der Einfachheit des Timeout-Ansatzes stehen daneben potentielle Leistungsprobleme gegenüber (Kap. 8.5.3).

11.3.2 Kommunikation

Neben der Transaktionsverwaltung ist der TP-Monitor vor allem auch für die Kommunikation von Anwendungsprogrammen mit dem Endbenutzer, dem DBS und anderen Anwendungsprogrammen verantwortlich, wobei in zentralisierten Transaktionssystemen sämtliche Eigenschaften der eingesetzten Kommunikationsmechanismen dem Programmierer verborgen bleiben. Diese Kommunikationsunabhängigkeit sollte natürlich auch in verteilten Transaktionssystemen erhalten bleiben, bei denen unterschiedliche TP-Monitore, LDBS und Kommunikationsprotokolle beteiligt sein können. Zur Ausführung entfernter Teilprogramme stellen derzeitige TP-Monitore Transaktionsanwendungen vor allem drei unterschiedliche Ansätze zur Verfügung: Konversationen, RPC (Remote Procedure Call) sowie persistente Warteschlangen. Daneben ist zu unterscheiden zwischen synchroner und asynchroner Ausführung der entfernten Teilprogramme.

Weitverbreitet ist die Kommunikation über Konversationen im Rahmen eines sogenannten *"Peer-to-Peer"-Protokolls*. Dabei sind zwischen den Kommunikationspartnern zunächst Verbindungen aufzubauen, bevor eine Konversation (Dialog, Nachrichtenaustausch) über Sende- und Empfangsoperationen erfolgt [Be90, Cy91]. Dieser Ansatz wird von dem SNA-Protokoll LU6.2 sowie ISO OSI-Protokollen verfolgt sowie den darauf aufbauenden APIs zur Kommunikation (z.B. CPI-C). Er ist sehr flexibel, da nahezu beliebige Kooperationsformen realisiert werden können (z.B. können mehrere Teilprogramme asynchron gestartet werden u.ä.). Die resultierende Programmierschnittstelle ist jedoch sehr komplex und fehleranfällig. Ferner wird keine Ortstransparenz erreicht, da für lokale und entfernte Teilprogramme unterschiedliche Aufrufmechanismen verwendet werden (lokaler Prozeduraufruf vs. Kommunikation über Konversationen). TP-Monitore wie CICS oder UTM, die auf LU 6.2 aufbauen, verfolgen diesen Ansatz.

---

* Föderative DBS versuchen diese Aufgabe durch eine zusätzliche DBS-Komponente zwischen Anwendungen und LDBS zu lösen.

Die Alternative besteht in der Verwendung von *Remote Procedure Calls (RPC)* zum Starten externer Teilprogramme [Sc92, GR93]. Hierbei besteht eine Client-/Server-Beziehung zwischen den Kommunikationspartnern, wobei das rufende Programm (Client) jeweils nur einen Aufruf absetzt und stets eine Antwort vom gerufenen Programm (Server) entgegennimmt. Der Aufruf der Server-Funktionen ist meist synchron. Die Anwendungsprogrammierung wird gegenüber der Kommunikation über Konversationen erheblich vereinfacht, da eine einheitliche Schnittstelle zum Aufruf lokaler und entfernter Programme verwendet werden kann. Der RPC-Mechanismus (TP-Monitor) ist verantwortlich für Lokalisierung und Aufruf der Programme, wobei ggf. eine Kommunikation und Anpassung von Parameterformaten erfolgt. Einige neuere TP-Monitore wie Encina nutzen zur RPC-Abwicklung die Middleware-Komponente OSF DCE (s.o.), allerdings erweitert mit einer Transaktionssemantik ("transaktionsgeschützter RPC") [GR93].

Zur asynchronen Ausführung von Teilprogrammen unterstützen einige TP-Monitore das Konzept der *"queued transactions"* [GR93]. Dabei werden Programmaufrufe innerhalb *persistenter Warteschlangen* ("recoverable/reliable queues") verwaltet, die vom TP-Monitor zu bestimmten Zeitpunkten abgearbeitet werden. Das rufende Programm wartet dabei nicht auf das Ende (bzw. den Start) der auszuführenden Funktion, sondern beendet sich bereits, nachdem ihm die spätere Ausführung des Programms (nach Einfügung in die Warteschlange) zugesichert wurde. Der TP-Monitor garantiert dazu, daß das betreffende Programm genau einmal ausgeführt wird (Exactly-Once-Semantik). Operationen auf den Warteschlangen (Einfügen, Ausfügen) sind Teil der ACID-Transaktion und i.a. durch die Anwendungsprogramme vorzunehmen.
Persistente Warteschlangen sind i.a. nicht für Dialogaufgaben nutzbar, die eine direkte Bearbeitung sämtlicher Teilschritte erfordern. Dennoch weist dieses Konzept vor allem zur Transaktionsverarbeitung in heterogenen Systemen eine Reihe signifikanter Vorteile auf. Denn die verzögerte Ausführbarkeit nicht-lokaler Programmaufrufe bedeutet eine Erhöhung der Knotenautonomie (Ausführungsautonomie). Weiterhin ergibt sich eine erhöhte Unabhängigkeit gegenüber der Verfügbarkeit der Knoten, an denen eine Funktion gestartet werden soll. Denn ist der betreffende Rechner zunächst nicht erreichbar, kann der TP-Monitor des Ursprungsknotens automatisch den Aufruf zu einem späteren Zeitpunkt wiederholen. Bei synchroner Programmausführung ist dagegen eine Fehlerbehandlung im aufrufenden Programm vorzusehen. Die asynchrone Programmausführung im Rahmen von "queued transactions" erlaubt auch in vielen Fällen die zuverlässige Zerlegung einer verteilten Transaktion in mehrere kürzere Transaktionen, was z.B. für Überweisungsaufträge zwischen Banken in der Praxis Verwendung findet [BHM90]. Damit ergibt sich auch eine erhebliche Reduzierung der Sperrdauer, wodurch das Sperrproblem lang-lebiger Transaktionen umgangen wird. Auf der anderen Seite können durch den TP-Monitor nicht behebbare Fehler während der asynchronen Programmausführung (z.B. aufgrund eines Programmierfehlers) zu

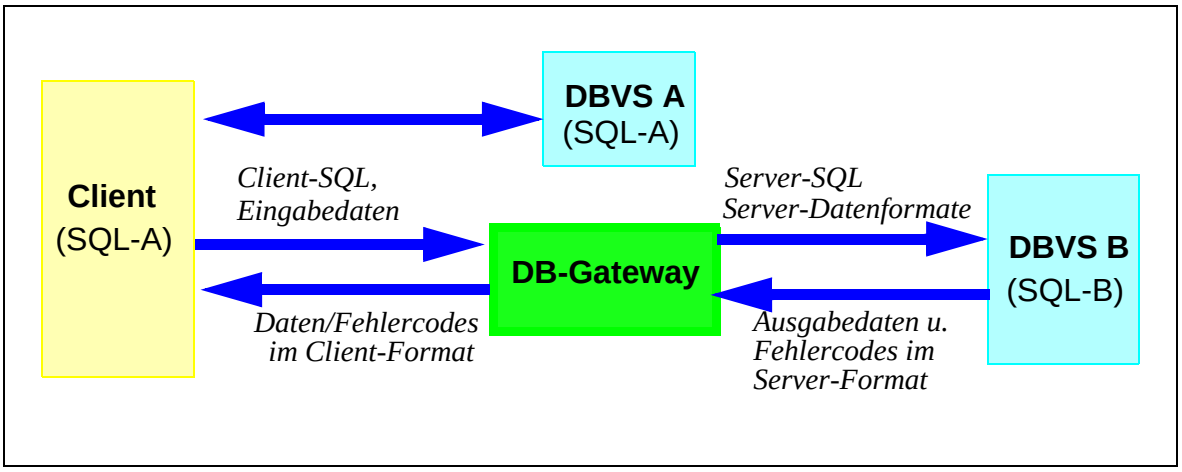
Inkonsistenzen führen, die durch manuelle Recovery-Aktionen auszugleichen sind. Bei synchronen Aufrufen würde dagegen im Fehlerfall die gesamte Transaktion abgebrochen.

11.3.3 Datenbank-Gateways

Die überwiegende Mehrzahl derzeitiger DBVS unterstützt die Anfragesprache SQL, wodurch der einheitliche Zugriff auf mehrere LDBS - wie beim Ansatz der Verteilung von DB-Operationen gefordert - bereits erheblich vereinfacht wird. Trotzdem zeigt sich, daß praktisch keine SQL-Schnittstelle mit der anderer Hersteller übereinstimmt und zum Teil beträchtliche Abweichungen zum ISO SQL-Standard vorliegen. Dies geht zum Großteil darauf zurück, daß die ersten SQL-Standardisierungen der ISO (SQL-86, SQL-89) sehr unvollständig waren und viele kritische Punkte wie Behandlung von Integritätsbedingungen, Katalogaufbau oder Fehler-Codes nicht festlegten.

Zur Angleichung dieser Unterschiede, wie sie beim integrierten Zugriff auf DBVS mehrerer Hersteller erforderlich wird, werden DB-Gateways eingesetzt. Das Einsatzprinzip von DB-Gateways ist in Abb. 11-3 veranschaulicht. Dabei soll im Client-Programm (Anwendung oder Tool, z.B. Spreadsheet-Programm) auf SQL-Datenbanken der Hersteller A und B zugegriffen werden. Wenn der Client als Anfragesprache den SQL-Dialekt des Herstellers A (SQL-A genannt) verwendet, ist für den Zugriff auf dessen DBVS keine Konversion erforderlich. Für den Zugriff auf das DBVS des Herstellers B dagegen erfolgt eine Anpassung der SQL-Befehle und Datenformate an die von Hersteller B unterstützte Schnittstelle SQL-B. Umgekehrt transformiert das Gateway Ausgabedaten sowie Fehler-Codes in das Format von Hersteller A, das vom Client erwartet wird.
Ein Gateway kann prinzipiell entweder auf dem Client- oder dem Server-Rechner laufen. Generell führt der Gateway-Einsatz natürlich zu einer Erhöhung der Bearbeitungszeiten im Vergleich zum direkten DBVS-Zugriff. Ferner können Gateways leicht zu Durchsatz-Engpässen werden, insbesondere auf Server-Seite.

Abb. 11-3: Einsatz von DB-Gateways



Ein weiteres Problem beim Einsatz von Gateways besteht darin, daß die Zahl unterschiedlicher Gateways prinzipiell quadratisch mit der Zahl der Hersteller zunimmt, zwischen denen eine Zusammenarbeit unterstützt werden soll. Dies erfordert einen extrem hohen Aufwand zur Realisierung und Wartung dieser Gateways, da diese i.a. bei jeder Versionsänderung einer der beteiligten Produkte anzupassen sind. Dieses Problem könnte natürlich durch die Verwendung eines gemeinsamen SQL-Standards von allen Herstellern umgangen werden, was jedoch schon aus Kompatibilitätsgründen mit früheren Versionen der einzelnen DBVS nicht möglich ist. Eine starke Reduzierung der notwendigen Gateway-Anzahl läßt sich jedoch bereits erreichen, wenn client-seitig eine standardisierte SQL-Version (SQL-API) verwendet wird. In diesem Fall ist nämlich nur noch ein Gateway pro Hersteller erforderlich, um die Abbildung vom SQL-Standard in den lokalen SQL-Dialekt vorzunehmen. Die Festlegung einer solchen portierbaren SQL-Version erfolgte durch die X/Open, wobei der Sprachumfang die in den meisten SQL-Implementierungen vorkommenden Konstrukte umfaßt und auch im ISO-SQL-Standard fehlende Aspekte wie die einheitliche Definition von Fehler-Codes berücksichtigt.

Die Mehrzahl von DB-Gateways unterstützt den Zugriff auf SQL-DBVS, da hier aufgrund der weitgehenden Übereinstimmung in der Funktionalität verschiedener Systeme eine relativ einfache Realisierung möglich wird. Es wird jedoch i.a. lediglich "dynamisches SQL" unterstützt, wobei Anfragen erst zur Ausführungszeit übersetzt und optimiert werden. Dies kann vor allem für vorgeplante Anwendungsfunktionen eine erhebliche Performance-Einbuße bedeuten. Einige Gateway-Anbieter (DBVS-Hersteller) unterstützen SQL-Zugriffe auch auf Dateisysteme sowie nicht-relationale DBVS wie IMS, die selbst kein SQL anbieten. In diesem Fall wird das Gateway um die DBVS-Funktionalität des jeweiligen Herstellersystems erweitert und das nicht-relationale System lediglich als weitere

Dateizugriffsschnittstelle aufgefaßt [SB90]. Es läßt sich dabei jedoch i.a. nur ein beschränkter Befehlsumfang von SQL realisieren.

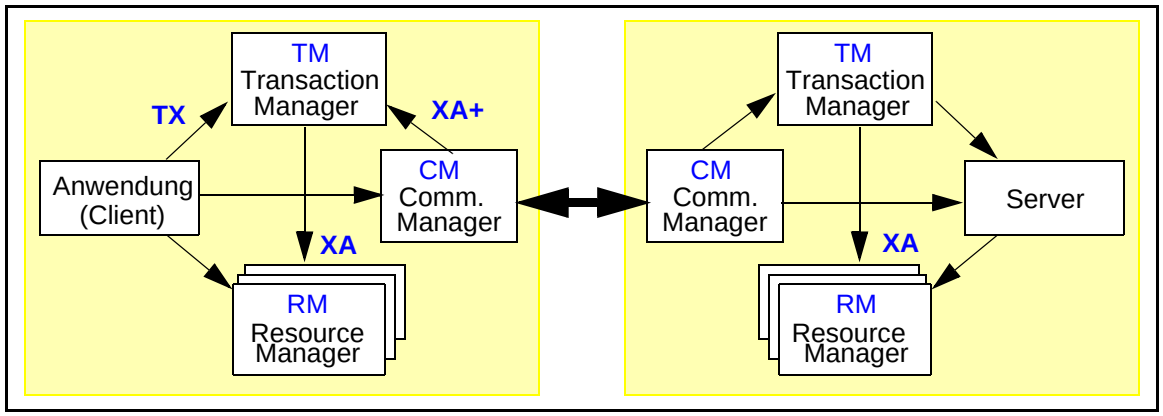
11.4 Standardisierungsansätze

Zunächst werden zwei Standardisierungsansätze zur verteilten Transaktionsbearbeitung vorgestellt, welche v.a. eine programmierte Verteilung unterstützen: X/Open DTP (Kap. 11.4.1) sowie MIA (Kap. 11.4.2). Die weiteren Ansätze betreffen die Verteilung von DB-Operationen. Hierzu diskutieren wir zunächst den ISO-Standard RDA (Kap. 11.4.3) und die darauf aufbauende Festlegung SQL Access (Kap. 11.4.4). Im Anschluß werden noch Standardisierungsanstrengungen einzelner Hersteller vorgestellt: ODBC (Microsoft) und IDAPI (Kap. 11.4.5) sowie DRDA von IBM (Kap. 11.4.6). DRDA umfaßt dabei mehrere Kooperationsformen, darunter die Verteilung einzelner DB-Operationen.

11.4.1 X/Open Distributed Transaction Processing (DTP)

Hauptziel von X/Open DTP ist die Standardisierung von Systemschnittstellen zur verteilten Transaktionsbearbeitung [GR93]. Dabei wird das in Abb. 11-4 gezeigte Modell eines verteilten Transaktionssystems zugrundegelegt. Man erkennt, daß in jedem der Rechner die Komponenten eines zentralisierten Transaktionssystems (Kap. 2.1.4) vorgesehen sind, also ein Transaction-Manager (TM), verschiedene Resource-Manager (RM) sowie Anwendungsprogramme. Neu hinzugekommen ist ein Communication-Manager (CM) pro Knoten, der einem speziellen Resource-Manager zur Durchführung von Kommunikationsaufgaben entspricht.

Abb. 11-4: X/Open-Modell eines verteilten Transaktionssystems [GR93]



Wesentlicher Teil der Standardisierung ist die Festlegung der Schnittstellen zwischen diesen Komponenten. Das API (Application Programming Interface) umfaßt dabei neben Funktionen zur Transaktionsverwaltung (TX-Schnittstelle) und Resource-Manager-Aufrufen zusätzlich noch die Kommunikationsschnittstelle zum

CM. Zur Kommunikation ist dabei sowohl die Unterstützung von Konversationen (basierend auf der SAA-Schnittstelle CPI-C von IBM) als von RPCs (basierend auf MIA RTI, s.u.) vorgesehen. Als Schnittstelle zwischen Anwendungen und DBS-Resource-Managern wurde von X/Open eine Teilmenge von SQL festgelegt, die von den meisten SQL-Implementierungen unterstützt wird. Dies war trotz der ISO-Standardisierung von SQL erforderlich, da existierende Implementierungen eine Vielzahl von Abweichungen zum Standard aufweisen.

Die Schnittstellen XA+ sowie XA dienen zur Abwicklung des Commit-Protokolls und sind für die Anwendungen nicht sichtbar (kein API-Bestandteil). Das Commit-Protokoll läuft zwischen den TMs der an einer Transaktionsausführung beteiligten Rechner ab, wobei die Kommunikation wiederum über die Communication-Manager abgewickelt wird. Als Commit-Protokoll ist dabei der ISO-Standard OSI TP vorgesehen; die X/Open-Schnittstelle XA+ legt fest, welche Aufrufe hierfür zwischen TM und CM zu verwenden sind. Die Resource-Manager sind über die XA-Schnittstelle am Commit-Protokoll beteiligt.

Zur Ausführung einer Transaktion teilt die Anwendung zunächst dem lokalen TM den Transaktionsbeginn mit, woraufhin dieser eine globale Transaktions-ID vergibt. Diese ID wird bei allen folgenden TM-, RM- und CM-Aufrufen der Transaktion mitgegeben. Nach Aufruf eines RM meldet sich dieser (über die XA-Schnittstelle) ebenfalls beim lokalen TM an, damit diesem bekannt ist, wer am Commit-Protokoll zu beteiligen ist. Für den Aufruf eines entfernten Programms (Abb. 11-4) ist von der Anwendung ein CM-Aufruf erforderlich. Die CM-Komponente informiert daraufhin den lokalen TM, daß sich die betreffende Transaktion auf einen anderen Rechner ausweitet (Schnittstelle XA+). Am zweiten Rechner erfolgt von der dortigen CM-Komponente eine Anmeldung der Transaktion an den lokalen TM sowie der Aufruf des entsprechenden Programms. Nachdem im Client-Rechner das Anwendungsprogramm dem lokalen TM das Commit mitteilt, führt dieser das Commit-Protokoll mit allen lokalen RMs sowie externen TMs aus, die an der Transaktion beteiligt waren.

Das X/Open-Modell sieht einen TP-Monitor nicht explizit vor, jedoch würde dieser typischerweise die Funktionalität des TM und des CM umfassen. Daneben bieten TP-Monitore jedoch noch eine Reihe weiterer Dienste (z.B. Programmverwaltung, Lastbalancierung, Authentifikation, Präsentationsunterstützung etc.), die im X/Open-Modell (noch) nicht explizit definiert sind. Heterogene DBS sind als Resource-Manager problemlos integrierbar, sofern sie die XA-Schnittstelle unterstützen. Eine zunehmende Anzahl von DBVS (Informix, Oracle etc.) und TP-Monitoren (Encina, CICS/6000, UTM-D etc.) bietet eine solche XA-Kompatibilität.

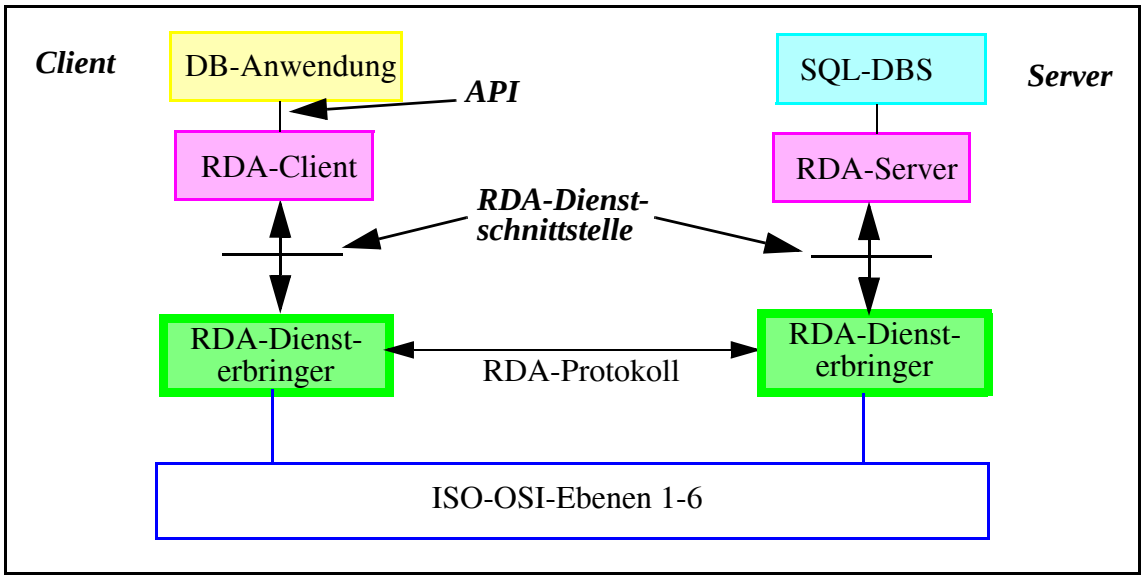
11.4.2 Multivendor Integration Architecture (MIA)

Eine ähnliche, jedoch umfassendere Zielsetzung wie X/Open DTP verfolgt das Herstellerkonsortium MIA (Multivendor Integration Architecture). Die Bildung dieses Konsortiums wurde von der japanischen Telefongesellschaft NTT veranlaßt und umfaßt Hersteller wie DEC, IBM, Fujitsu, Hitachi und NEC. Die Standardisierungsaktivitäten betreffen APIs, Kommunikationsprotokolle sowie Endbenutzer-Schnittstellen, wobei zum Teil auf andere (De-facto-) Standards zurückgegriffen wurde (u.a. bei den Sprachen SQL, C, FORTRAN und COBOL). Speziell zur verteilten Transaktionsverarbeitung wurde als Teil des APIs die Sprache STDL (Structured Transaction Definition Language) definiert [Be93b]. Diese Sprache unterstützt zur Kommunikation transaktionsgeschützte RPCs sowie persistente Warteschlangen ("queued transactions"). Das RPC-Protokoll, RTI (Remote Task Invocation) genannt, entspricht einer Integration des DCE RPC-Mechanismus' sowie dem OSI TP-Standard zur Commit-Behandlung. RTI wurde mittlerweile auch als RPC-Mechanismus für X/Open DTP übernommen. Bei der Festlegung der Protokolle und Schnittstellen wurde darauf geachtet, daß diese von den Produkten (TP-Monitoren) der beteiligten Hersteller innerhalb kurzer Zeit unterstützt werden können.

11.4.3 Remote Database Access (RDA)

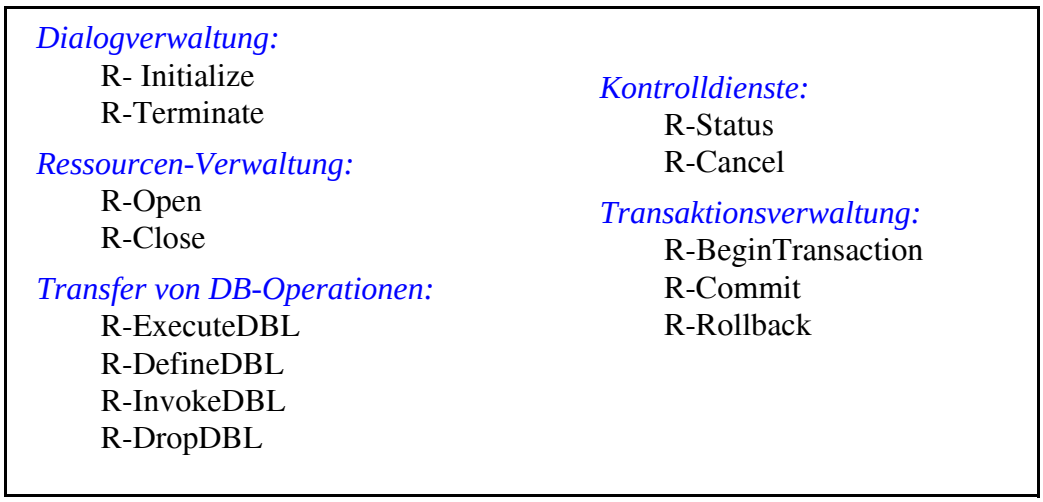
RDA ist ein ISO OSI-Kommunikationsstandard zur Verteilung von DB-Operationen in heterogenen Systemumgebungen [La91, Pa91]. Er ermöglicht einem Programm auf einem Client-Rechner, DB-Operationen auf einem DBVS eines Server-Rechners auszuführen, unabhängig von den zugrundeliegenden Hardware- und Betriebssystem-Plattformen. Die Auswahl des Ziel-DBVS kann dabei bis zur Ausführungszeit des Anwendungsprogramms verzögert werden, sofern die beteiligten Rechner die OSI-Kommunikationsprotokolle befolgen. Die RDA-Standardisierungen begannen bereits 1985, zunächst im Rahmen der ECMA (European Computer Manufacturers Association); die Verabschiedung als ISO-Standard erfolgte 1992. Die RDA-Spezifikation besteht dabei aus einem generischen Teil sowie aus auf einzelne DB-Anfragesprachen zugeschnittenen Spezialisierungen. Derzeit besteht eine Spezialisierung lediglich für SQL, wobei vereinfachend vorausgesetzt wird, daß die beteiligten Kommunikationspartner den ISO-SQL-Standard befolgen. Zudem kann in der aktuellen Fassung des RDA-Standards innerhalb einer Transaktion nur auf eine Datenbank zugegriffen werden, jedoch ist eine Erweiterung auf mehrere Server geplant

Abb. 11-5: Einsatz des RDA-Protokolls.



Wie bereits in Kap. 2.2.2 erwähnt, ist RDA Teil der Anwendungsebene (Schicht 7) des ISO OSI-Referenzmodells zur Kommunikation in offenen Systemen. Wie Abb. 11-5 zeigt, ruft ein Anwendungsprogramm nicht direkt die RDA-Funktionen ("Dienste") zur Kommunikation mit einem SQL-Server auf. Zur Erhöhung der Portabilität werden die Funktionen eines APIs verwendet, welche dann durch eine Zwischenkomponente (hier RDA-Client genannt) in Aufrufe der RDA-Dienstschnittstelle umgesetzt werden. Auf der Server-Seite werden umgekehrt die eingehenden RDA-Nachrichten über einen RDA-Server-Prozeß in Aufrufe an das SQL-DBS umgesetzt. Der Rücktransport von Ergebnissen oder Statusmeldungen erfolgt analog. Das API sowie die Schnittstelle zwischen RDA-Server und SQL-DBS sind im Rahmen des RDA-Standards nicht festgelegt. Dieser umfaßt die Definition der RDA-Dienstschnittstelle sowie des eigentlichen Kommunikationsprotokolls zwischen den RDA-Diensterbringern.

Abb. 11-6: RDA-Kommunikationsdienste [Ar91]



Die Funktionen der RDA-Dienstschnittstelle sind in Abb. 11-6 nach Gruppen unterteilt zusammengestellt. Da RDA wie andere ISO OSI-Protokolle ein verbindungsorientiertes Protokoll darstellt, werden explizite Dienste zum Auf- und Abbau von Dialogen zwischen Client und Server bereitgestellt. Die Anweisungen zur Ressourcen-Verwaltung dienen zum "Öffnen" bzw. "Schließen" einer Datenbank bzw. Teilen der Datenbank, auf die zugegriffen werden soll. Nach Öffnen der notwendigen Ressourcen können mit der Funktion R-ExecuteDBL die eigentlichen Datenbankoperationen einer bestimmten DB-Sprache (z.Zt. nur SQL) übermittelt werden. Diese Befehle müssen dann im Server erst übersetzt werden, was sehr aufwendig sein kann. Daneben ist es möglich, DB-Operationen vorab zu übermitteln (R-DefineDBL), um sie bereits vor ihrer Ausführung zu übersetzen und zu optimieren. Die Ausführung solcher DB-Befehle wird dann später durch die Funktion R-InvokeDBL erreicht; mit R-Drop können vordefinierte DB-Operationen wieder gelöscht werden. Mit R-Status kann der Status einer zuvor abgesetzten DB-Operation nachgefragt werden, z.B. wenn deren Ausführungszeit unerwartet lange dauert; mit R-Cancel lassen sich solche Operationen abbrechen. Daneben werden drei Dienste zur Transaktionsverwaltung bereitgestellt (Starten, Commit und Abbruch einer Transaktion). Die RDA-Operationen werden bezogen auf die Client-Anwendung stets asynchron ausgeführt, so daß eine Weiterarbeit vor Beendigung einer Operation möglich ist und mehrere Operationen initiiert werden können.

Das RDA-Kommunikationsprotokoll legt im Detail die Formate aller RDA-Nachrichten fest. Ferner werden die Regeln und Bedingungen für den Nachrichtenaustausch bestimmt, insbesondere die zulässigen Reihenfolgen, in der die einzelnen RDA-Nachrichten übermittelt werden können. So wird z.B. verlangt, daß R-Initialize allen anderen RDA-Operationen vorangehen muß. RDA ist keine in sich geschlossene Norm, sondern greift auf andere OSI-Standards zurück. Dies sind insbesondere das Ebene-6-Protokoll ASN.1 (Abstract Syntax Notation One) zur her-

stellerneutralen Datenrepräsentation sowie ACSE (Association Control Service Element) zur Verbindungsverwaltung. Bei Erweiterung des RDA-Protokolls auf den Mehr-Server-Fall soll für das dann erforderliche verteilte Commit-Protokoll der ISO-Standard TP verwendet werden.

11.4.4 SQL Access

Die Beschreibung des RDA-Standards zeigt, daß dieser keine Schnittstellen zum Anwendungsprogramm sowie zum DBS festlegt. Insbesondere wird vorausgesetzt, daß Client und Server den ISO-SQL-Standard befolgen, was jedoch in der Praxis unrealistisch ist. Um eine praktische Nutzbarkeit des RDA-Standards zu ermöglichen, schlossen sich 1989 zahlreiche Hersteller von DB-Tools und SQL-DBS (außer IBM) zu dem Konsortium *SQL Access Group (SAG)* zusammen. Die SAG definierte die auf RDA aufbauende Spezifikation *SQL Access* für den Fernzugriff auf SQL-Datenbanken [Ar91, Ba91]. Daneben wurden die Spezifikationen von den beteiligten Unternehmen prototypisch realisiert, um ihre praktische Eignung für die Zusammenarbeit zwischen Anwendungen und SQL-Datenbanken in heterogenen Umgebungen zu demonstrieren. Eine erste derartige Demonstration (basierend auf Version 1 von SQL Access) fand bereits 1991 statt, wobei eine Interoperabilität zwischen 10 unterschiedlichen Client- und 9 Server-Systemen nachgewiesen wurde [Ar91].

SQL Access besteht aus einem API- sowie einem FAP-Teil (Formats and Protocols). Als FAP wurde der RDA-Standard gewählt*. Das API umfaßt eine normierte Teilmenge des SQL-Sprachumfangs, der in den meisten SQL-Implementierungen verfügbar ist. Um eine ausreichende Funktionalität bereitzustellen, waren jedoch eine Reihe von Erweiterungen im Vergleich zum ISO SQL-Standard bzw. existierenden SQL-Implementierungen erforderlich. So wurden eigene Befehle definiert zum Auf- und Abbau von Verbindungen mit einem SQL-Server (CONNECT bzw. DISCONNECT) und zur Auswahl unter zuvor etablierten Verbindungen (SET CONNECTION). Weiterhin wurde eine einheitliche Menge von Fehler-Codes festgelegt, um eine herstellerunabhängige Fehlerbehandlung zu unterstützen. Neben einem API für direkt eingebettete SQL-Anweisungen wurde auch ein *Call-Level-Interface* (SQL/CLI) festgelegt**. Die von der SAG definierten SQL-APIs wurden von der X/Open übernommen; das CLI wird auch im künftigen SQL-Standard (SQL3) verwendet.

Abb. 11-7 veranschaulicht die Abbildung von Operationen des APIs auf die RDA-Dienstschnittstelle, wie sie von dem RDA-Client (Abb. 11-5) vorgenommen werden könnte. Auf der Server-Seite erfolgt eine ähnliche Umsetzung eingehender Aufrufe auf Operationen des lokalen SQL-DBS durch den RDA-Server. Der RDA-Ser-

* Bei Festlegung von SQL Access V1 war RDA noch kein ISO-Standard. Vielmehr gingen die Erfahrungen mit SQL Access in beträchtlichem Maße in die RDA-Standardisierung ein.

ver umfaßt damit auch die Funktionalität eines DB-Gateways, um die Abbildung zwischen dem normierten SAG-SQL und lokalem SQL vorzunehmen.

Abb. 11-7: Umsetzung von API-Operationen von SQL Access in RDA-Aufrufe

Anwendung	***erzeugte RDA-Aufrufe***
CONNECT TO "INGRES@a.b.c.d" AS "Ingres" USER "E_Rahm"	*A-Assoc (...)* *R-Initialize ("E_Rahm", ...)* *R-Open ("INGRES@a.b.c.d", ...)*
INSERT INTO Pers VALUES (...)	*R_BeginTransaction (...)* *R_ExecuteDBL ("INSERT", ...)*
COMMIT WORK	*R_Commit (...)*
DISCONNECT "Ingres"	*R_Close ("Ingres", ...)* *R_Terminate (...)*

Wie RDA so ist auch die erste Version von SQL Access auf die Interaktion mit einem SQL-DBS pro Transaktion beschränkt. In künftigen Versionen wird diese Restriktion jedoch aufgehoben. Weiterhin wird daran gearbeitet, die RDA-Funktionalität auf Basis von TCP/IP zu realisieren sowie den SQL-Sprachumfang zu erweitern (in Anlehnung an den SQL-92-Standard).

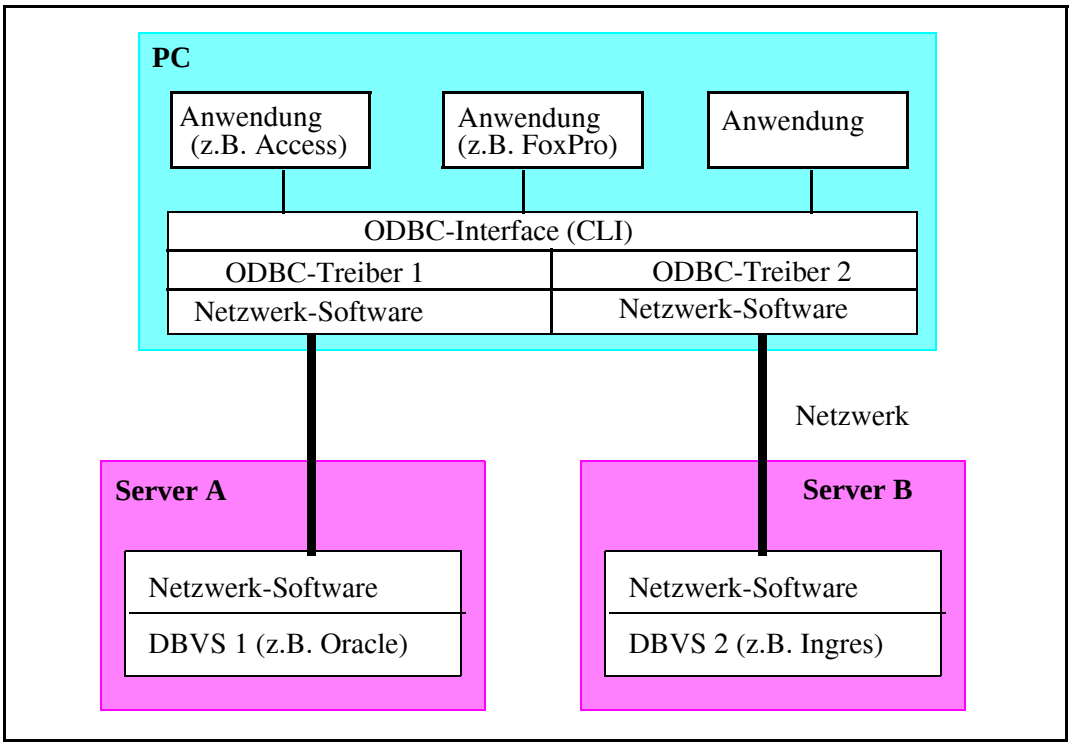
11.4.5 ODBC und IDAPI

Um Client-Anwendungen - insbesondere auf PCs - den Zugriff auf mehrere heterogene DBVS zu ermöglichen, definierte die Fa. Microsoft das ODBC-Konzept (Open Database Connectivity) [Je93]. Es basiert auf dem SQL-CLI (Call Level Interface) der SQL Access Group. Über dieses API können somit PC-Anwendungen bzw. -Tools in einheitlicher Weise auf verschiedene SQL-Server zugreifen. Dabei erfolgt die Anpassung an das jeweilige Server-SQL durch spezielle ODBC-Treiber auf Client-Seite (Abb. 11-8). Diese Treiber entsprechen in der Funktionalität DB-Gateways und sind für die SQL-DBVS transparent. Allerdings erfolgt die Kommunikation zwischen Client und Server nicht über standardisierte Kommunikationspro-

** Direkt in eine Programmiersprache eingebettete SQL-Anweisungen können wie Anweisungen der jeweiligen Sprache verwendet werden (mit einem Präfix EXEC SQL versehen); ihre Übersetzung erfolgt i.a. durch spezielle Prä-Compiler. Bei einem Call-Level-Interface werden SQL-Anweisungen im Rahmen externer Bibliotheksaufrufe spezifiziert, die erst beim Binden des ausführbaren Programmes aufgelöst werden. Damit wird eine Portierung von Anwendungen ohne Neuübersetzung ermöglicht. Andererseits wird die direkte Einbettung i.a. als benutzerfreundlicher angesehen. Gelegentlich werden APIs mit direkter Einbettung auch als *syntaktische APIs* und Call-Level-Interfaces als *prozedurale APIs* bezeichnet [GR93].

tokolle wie RDA, so daß eine Abbildung der ODBC-Operationen an das jeweils vorliegende Netzwerk erforderlich wird. Durch die Dominanz von Microsoft im PC-Markt und der großen Anzahl von PC-Anwendungen ist in Zukunft mit einer großen Bedeutung von ODBC zu rechnen. Dies zeigt sich auch an der wachsenden Anzahl verfügbarer ODBC-Treiber. Andere Hersteller (Oracle, DEC etc.) bieten auch zunehmend Produkte auf Basis von SAG-Spezifikationen an, um den Zugriff auf Datenbanken anderer Hersteller zu unterstützen.

Abb. 11-8: Zugriff auf heterogene Datenbanken mit ODBC



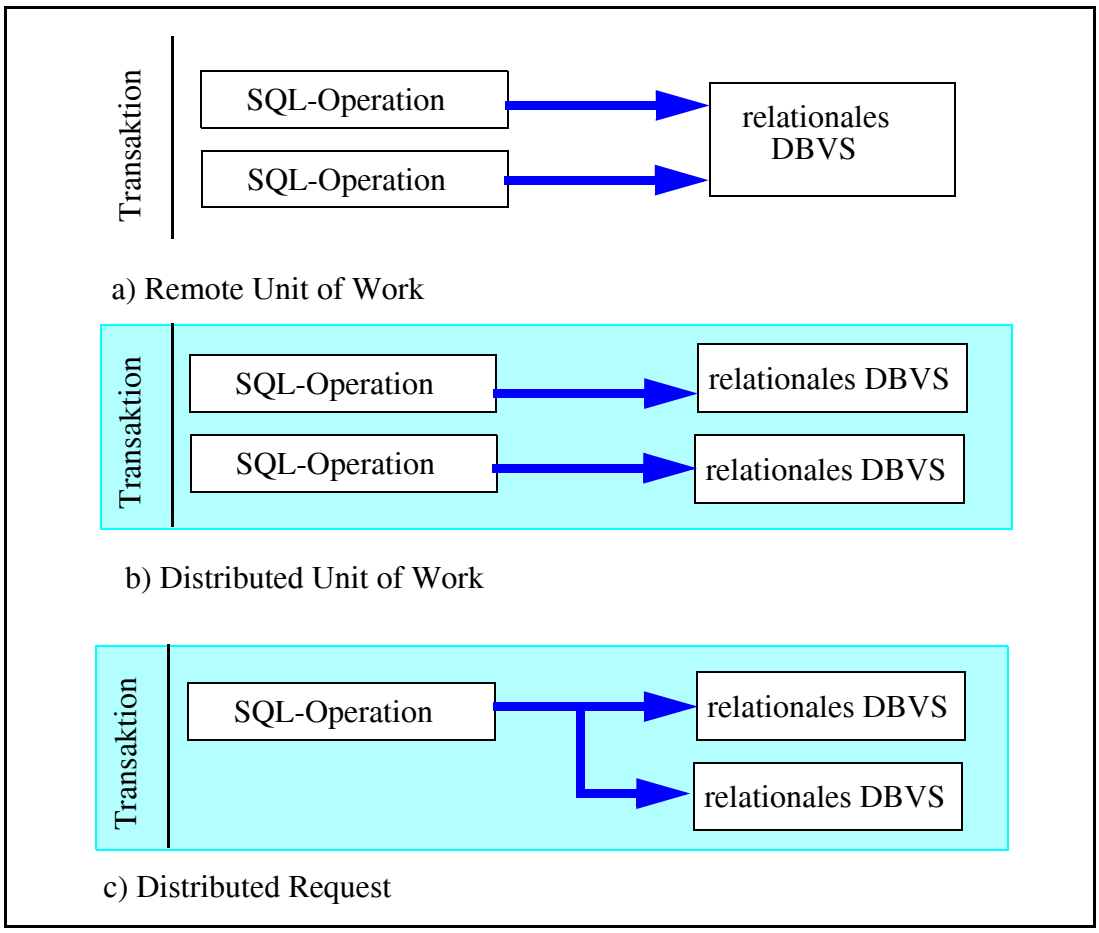
Als Konkurrenzansatz zu ODBC definierten die Hersteller Borland, IBM, Novell und WordPerfect ein eigenes API namens IDAPI (Integrated Database Application Programming Interface). Dabei soll neben SQL auch ein standardisierter Zugriff auf satzorientierte DBS unterstützt werden (Navigational Call Level Interface).

11.4.6 IBM Distributed Relational Database Architecture (DRDA)

Im Rahmen seiner Systemarchitektur SAA (System Application Architecture) definierte IBM 1990 die sogenannte "Distributed Relational Database Architecture (DRDA)" [Da93]. Es handelt sich dabei um die Festlegung verschiedener Kooperationsformen für den Zugriff auf mehrere relationale SQL-Datenbanksysteme. Dabei wird primär der integrierte Zugriff auf die SQL-Datenbanksysteme von IBM unterstützt, nämlich DB2/MVS (MVS/ESA-Betriebssystem), DB2/6000 (AIX), DB2/2 (OS/2), SQL/DS (VM und VSE) und SQL/400 (OS/400). Da die DRDA-Spe-

zifikationen jedoch öffentlich dokumentiert wurden, können auch DBS anderer Hersteller in die Verarbeitung eingebunden werden, sofern sie die DRDA-Protokolle befolgen.

Abb. 11-9: DRDA-Stufen



DRDA unterscheidet drei Stufen für den Datenbankzugriff in verteilten Umgebungen: "Remote Unit of Work" (RUOW), "Distributed Unit of Work" (DUOW) und "Distributed Request". Dabei ist "Unit of Work" die IBM-Bezeichnung für Transaktion. Die einzelnen Verteilungsformen können folgendermaßen charakterisiert werden:

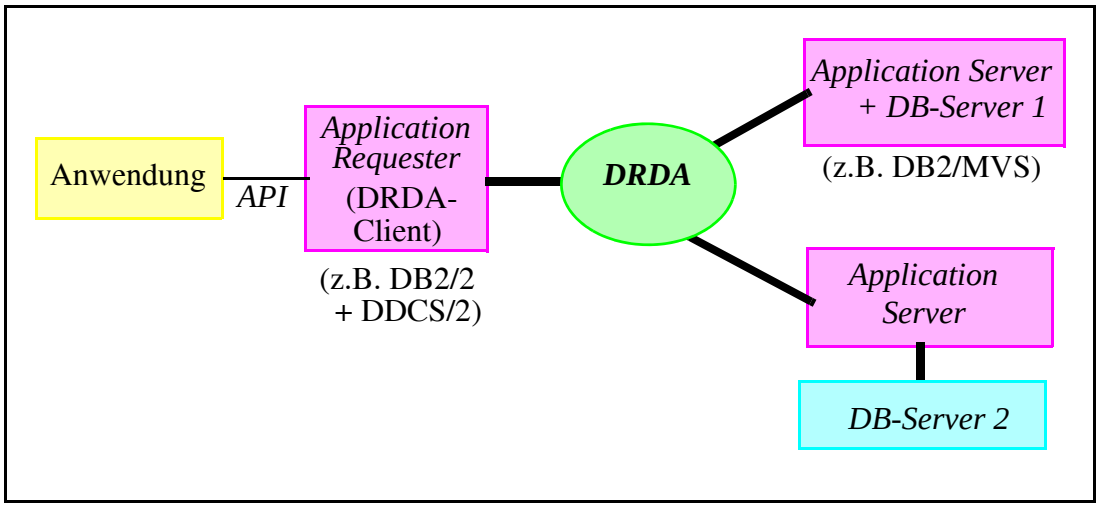
- *Stufe 1: Remote Unit of Work (Abb. 11-9a)*
  Hierbei kann pro Transaktion nur auf Datenbanken an einem entfernten Knoten zugegriffen werden, wobei jede SQL-Anweisung einzeln an den entsprechenden Rechner gesendet wird. Da meist nur ein DBVS pro Rechner vorliegt, sind Transaktionen somit i.a. auf eine Datenbank beschränkt, ähnlich wie in der derzeitigen Fassung von RDA. Eine Unterstützung heterogener Datenbanken wird nur insoweit geboten, daß neben dem entfernten DBVS auch auf lokale DBS auf Client-Seite zugegriffen werden kann.
- *Stufe 2: Distributed Unit of Work (Abb. 11-9b)*
  Diese Kooperationsform realisiert die "Verteilung von DB-Operationen" in der allge-

meinen Form mit mehreren entfernten Datenbanken pro Transaktion. Jede SQL-Anweisung ist dabei auf eine Datenbank beschränkt. Zur Gewährleistung der Alles-oder-Nichts-Eigenschaft (insbesondere von Änderungstransaktionen) an mehreren Datenbanken ist ein globales Commit-Protokoll zu unterstützen. Die Anwendung muß die Verteilung der Datenbanken kennen und SQL-Befehle direkt an die jeweiligen DBVS richten.

- *Stufe 3: Distributed Request (Abb. 11-9c)*
  Dieser Ansatz ermöglicht die verteilte Ausführung einer SQL-Operation auf mehreren lokalen und/oder entfernten Datenbanken. Sämtliche Verteilungsaspekte werden außerhalb der Anwendungen realisiert.

Stufe 3 wird von den erwähnten SQL-DBVS von IBM derzeit noch nicht unterstützt, Stufe 2 lediglich von DB2/MVS (seit Version 3.1) und DB2/6000 und DB2/2 (seit Version 2). Die anderen DBVS bieten zur Zeit nur Stufe 1. Die Realisierung des verteilten Commit-Protokolls, das ab Stufe 2 erforderlich ist, basiert auf LU6.2. Globale Deadlocks werden über einen Timeout-Ansatz aufgelöst.

Abb. 11-10: Client-Server-Kooperation bei DRDA



Ähnlich wie RDA spezifiziert DRDA kein API (SQL), sondern die genauen Kommunikationsprotokolle und Nachrichtenformate für die Verteilung von DB-Operationen. Die Abbildung von SQL-Operationen einer Anwendung auf die DRDA-Nachrichten erfolgt über einen DRDA-Client, auch *Application Requester* genannt. Dieser kommuniziert mit einem *Application Server*, der mit dem jeweiligen DBVS zusammenarbeitet (Abb. 11-10). DRDA legt dabei die Protokolle zwischen Application Requester und Application Server sowie zwischen Application Server und DB-Server (DBVS) fest. Die Funktionen von Application Requester bzw. Server müssen nicht notwendigerweise durch eigene Komponenten (Gateways) erbracht werden, sondern sind in DRDA-kompatiblen DBS meist schon integriert*. Damit spezifiziert DRDA zugleich ein Protokoll zum Austausch von SQL-Operationen (bzw. Teiloperationen bei Stufe 3) und Ergebnissen zwischen

DRDA-DBVS. Die Unterstützung der DRDA-Protokolle verlangt erhebliche Systemerweiterungen am DBVS und schränkt somit dessen Autonomie und Heterogenität ein. Dies gilt in besonderem Maße für Stufe 3 von DRDA, welche eine ähnlich enge Kooperation zwischen den einzelnen DBS verlangt wie in (homogenen) Verteilten DBS.

Übungsaufgaben

Aufgabe 11-1: Beschränkung globaler Änderungstransaktionen

a) Wieso ist bei Beschränkung globaler Transaktionen auf eine ändernde Sub-Transaktion keine globale Serialisierbarkeit gewährleistet, wenn jedes LDBS ein striktes Zwei-Phasen-Sperrprotokoll einsetzt, jedoch kein globales Commit-Protokoll vorgenommen werden soll?

b) Sind dabei globale Deadlocks möglich?

Aufgabe 11-2: Gateway-Anzahl

Es soll eine Interoperabilität zwischen 10 DB-Tools und 5 SQL-DBVS jeweils unterschiedlicher Hersteller über DB-Gateways unterstützt werden. Wieviele unterschiedliche Gateways sind hierzu zu realisieren

a) ohne Nutzung einer gemeinsamen SQL-Sprache

b) bei Nutzung einer gemeinsamen (Zwischen-) SQL-Sprache, jedoch davon abweichenden SQL-Dialekten bei allen Herstellern

c) bei Unterstützung der gemeinsamen SQL-Sprache durch die Tool-Hersteller, nicht jedoch durch die DBVS?

Aufgabe 11-3: Abruf großer Ergebnismengen

SQL-Anfragen mit großen Ergebnismengen können einen extremen Kommunikationsaufwand verursachen, wenn jeder Ergebnissatz einzeln über das Netz angefordert wird (FETCH-Anweisung beim Cursor-Konzept). Wie könnte der Kommunikationsaufwand reduziert werden ?

---

* Die DBVS DB2/2 und DB2/6000 erfordern jedoch eigene Zusatzkomponenten DDCS/2 bzw. DDCS/6000 (Distributed Database Connection Services) für die Weitergabe von SQL-Operationen an entfernte DRDA-DBVS.

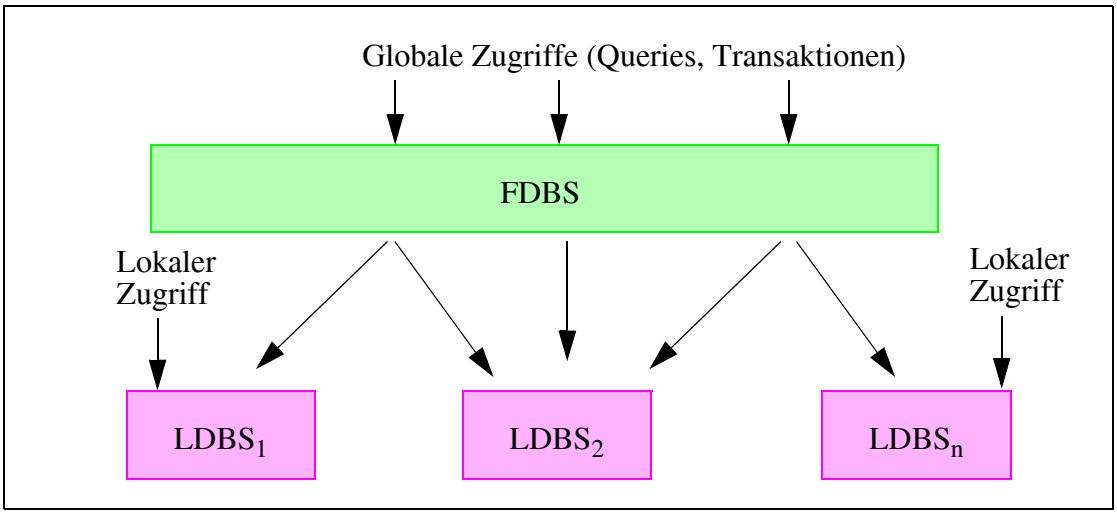
12 Föderative Datenbanksysteme

Die vorgestellten Ansätze der programmierten Verteilung sowie der Verteilung von DB-Operationen gestatten eine weitgehende Heterogenität und Autonomie der LDBS. Praxisorientierte Standardisierungsanstrengungen der X/Open und SAG erlauben die Verwendung einer gemeinsamen SQL-Anfragesprache auf unterschiedlichen DBS. Einschränkungen der Autonomie einzelner LDBS betreffen vor allem die Transaktionsverwaltung, z.B. durch die Unterstützung der XA-Schnittstelle zur verteilten Commit-Behandlung nach X/Open DTP. Außerdem kann innerhalb einer DB-Operation nicht auf mehrere Datenbanken zugegriffen werden, und es besteht nur ein geringer Grad an Verteilungstransparenz. Insbesondere wird keine Unterstützung zur Behandlung semantischer Heterogenität geboten.

Föderative Mehrrechner-DBS streben eine Behebung der letztgenannten Einschränkungen an. Unter föderativen Mehrrechner-DBS (*FDBS*) verstehen wir eine Menge autonomer, möglicherweiser heterogener LDBS, welche in begrenztem Umfang kooperieren, um externen Benutzern einen kontrollierten Zugriff auf Teile ihrer Daten zu gestatten. Dabei sollten die erwähnten Nebenanforderungen wie Verteilungstransparenz, Verbergen der Heterogenität (einheitliche Anfragesprache, einheitliches Datenmodell, automatische Behandlung von semantischer Heterogenität), DB-Operationen auf mehreren Datenbanken sowie das Transaktionskonzept möglichst erfüllt werden.
Die Architektur von FDBS basiert i.a. auf einem Zusatzebenen-Ansatz, da die LDBS weitgehend unverändert bleiben sollen (Abb. 12-1). Aufgabe der Zusatzschicht "FDBS" ist es, in der gemeinsamen Anfragesprache formulierte globale Anfragen bzw. Transaktionen, welche mehrere LDBS betreffen, zu bearbeiten. Hierzu erfolgt eine Zerlegung dieser Anfragen in von den LDBS lokal ausführbare Teilanfragen sowie (für Leseoperationen) das Zusammenführen der Teilergebnisse. Damit kann dann z.B. auch eine verteilte Join-Berechnung über unabhängige Datenbanken erfolgen. Die Zusatzschicht umfaßt daneben weitere globale Aufgaben, etwa zur Transaktionsverwaltung. Die Realisierung der Zusatzebene

kann zentralisiert oder verteilt erfolgen. So könnte auf jedem LDBS-Knoten auch eine FDBS-Komponente vorliegen, die mit den FDBS-Komponenten anderer Rechner kommuniziert. Alternativ dazu ist eine FDBS-Ausführung auf dedizierten Rechnern möglich.

Abb. 12-1: Grobarchitektur von föderativen DBS



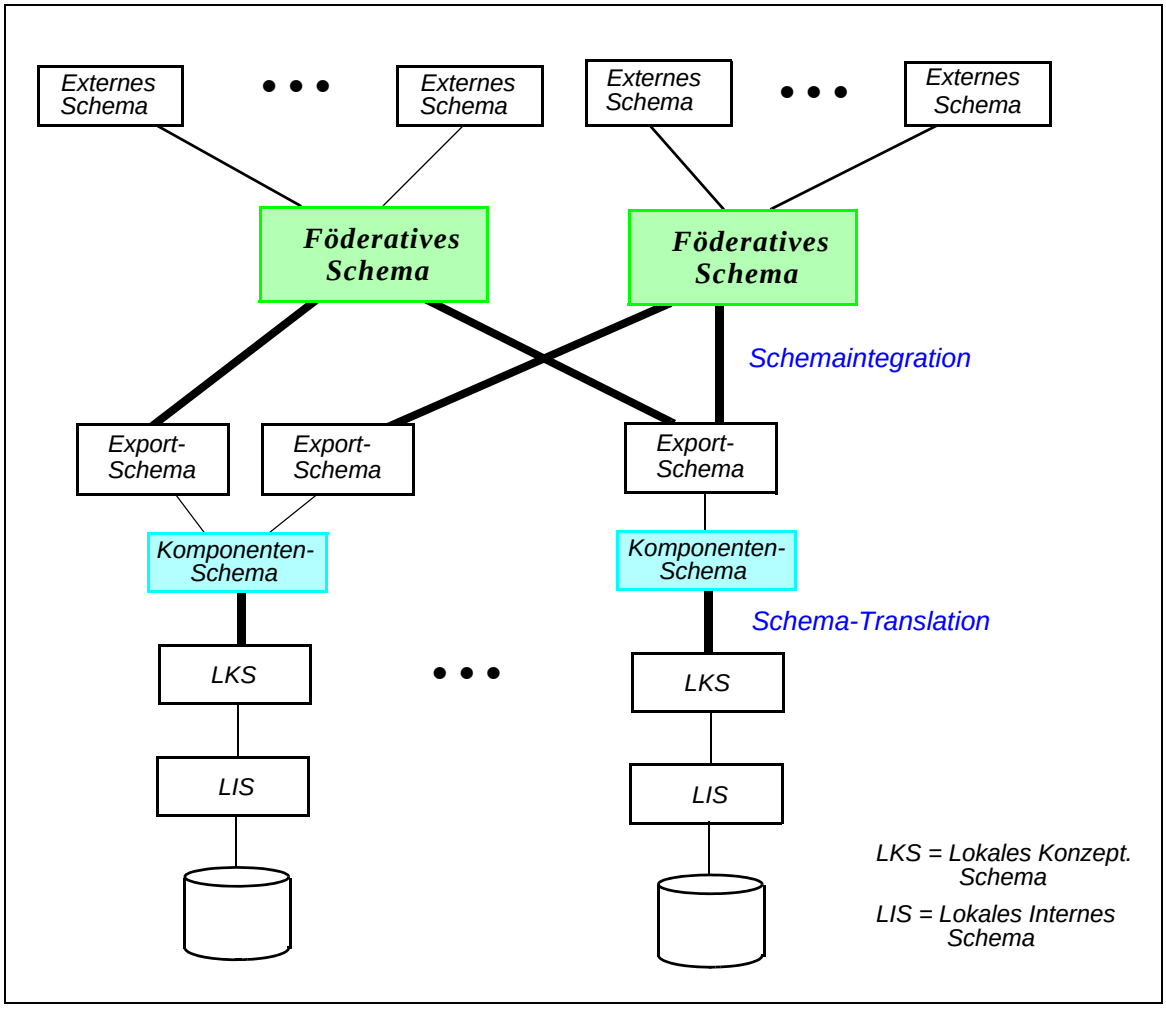
Bei der Realisierung von föderativen DBS lassen sich grob zwei Klassen unterscheiden, die wir nach [SL90] als eng bzw. lose gekoppelte FDBS bezeichnen wollen. *Eng gekoppelte FDBS* streben eine weitgehende Verteilungstransparenz an, in dem Benutzern gegenüber ein globales konzeptionelles Schema bzw. föderatives Schema angeboten wird, welches die Existenz mehrerer LDBS verbirgt. Dies setzt eine sogenannte Schemaintegration (s.u.) voraus, während der auch eine Behandlung von semantischer Heterogenität erfolgt. Die LDBS können sich im allgemeinen Fall zu mehreren Föderationen, mit jeweils einem eigenen föderativen Schema, zusammenschließen, um unterschiedlichen Benutzergruppen verschiedene Sichtweisen auf die Datenbanken zu ermöglichen. Bei *lose gekoppelten FDBS* dagegen, welche häufig auch als Multi-DBS bezeichnet werden, wird keine Schemaintegration angestrebt. Vielmehr bleibt hier die Unterscheidung mehrerer Datenbanken für den Benutzer sichtbar. Er bekommt jedoch Hilfsmittel (insbesondere eine Anfragesprache) zur Verfügung gestellt, um in einfacher und mächtiger Weise auf die verschiedenen Datenbanken zuzugreifen.

Im nächsten Unterkapitel betrachten wir zunächst eine allgemeine Schemaarchitektur für FDBS, welche eng und lose gekoppelte FDBS abdeckt. Anschließend stellen wir unterschiedliche Formen der semantischen Heterogenität sowie den Prozeß der Schemaintegration vor. Danach wird die Nutzung lose gekoppelter FDBS diskutiert. Abschließend behandeln wir kurz die Transaktionsverwaltung in FDBS. Überblicke zu FDBS-Prototypen bzw. -Produkten finden sich u.a. in [LMR90, Th90, BHP92].

12.1 Schemaarchitektur

Die Schemaarchitektur eines zentralisierten DBS nach ANSI/SPARC (Kap. 2.1.2) sowie die Schemaarchitektur von Verteilten DBS (Kap. 4.2) unterstützen keine ausreichend hohe Autonomie und Heterogenität der LDBS, wie für föderative DBS erforderlich. Eine besser geeignete Schemaarchitektur, welche für unterschiedliche Typen von FDBS anwendbar ist, zeigt Abb. 12-2. Dabei existiert an jedem Knoten, wie bei zentralisierten und Verteilten DBS, ein internes sowie ein konzeptionelles Schema (LIS bzw. LKS). Ebenso greifen Benutzer i.a. über externe Schemata auf die Datenbanken zu. Neu hinzugekommen sind drei Schematypen: Komponenten-, Export- sowie föderative Schemata.

Abb. 12-2: Schemaarchitektur von föderativen DBS [SL90]



Komponenten-Schemata sind erforderlich, wenn die LDBS unterschiedliche Datenmodelle verwenden; sie dienen daher zur Behandlung von Heterogenität (bei den Datenmodellen). Die Komponenten-Schemata basieren auf einem gemeinsamen Datenmodell (Common Data Model, CDM). Jedes lokale konzeptionelle Schema eines anderen Datenmodells wird durch eine *Schema-Translation* in ein

äquivalentes konzeptionelles Schema des CDM (das Komponenten-Schema) transformiert. In der Praxis ist das relationale Datenmodell als ein geeigneter Kandidat für das CDM anzusehen, da bereits ein Großteil der Datenbanken relational aufgebaut ist (und die Schema-Translation damit entfällt) und auch für nicht-relationale Datenbanken (hierarchische und netzwerkartige Datenbanken) meist eine Transformation ins Relationenmodell möglich ist [HK89]. Als sinnvolle Alternative werden vielfach auch semantische Datenmodelle (z.B. Entity-Relationship-Modelle) bzw. objekt-orientierte Modelle angesehen, da sie eine hohe Modellierungsmächtigkeit unterstützen, welche im Transformationsprozeß (sowie zur Schemaintegration) genutzt werden kann [SL90]. Allerdings besteht hier aus praktischer Sicht das Problem, daß eine Vielzahl solcher Modelle existiert und keines dieser Modelle eine signifikante Marktbedeutung erreicht hat. Zudem liegen meist noch Defizite hinsichtlich der Unterstützung mächtiger mengenorientierter DB-Operationen vor.

Export-Schemata werden an jedem Knoten auf dem Komponenten-Schema der lokalen Datenbank definiert. In ihnen wird festgelegt, welche Objekte der lokalen Datenbank im Rahmen einer Föderation externen Benutzern zugänglich gemacht werden sollen (ein Export-Schema pro Föderation). Weiterhin können die zulässigen Operationen durch entsprechende Zugriffsrestriktionen eingeschränkt werden. Die Export-Schemata dienen damit zur Unterstützung der LDBS-Autonomie, insbesondere der Kooperationsautonomie.

Ein *föderatives Schema* umfaßt die Schemaangaben mehrerer Export-Schemata, und zwar von den an der Föderation beteiligten LDBS. Weiterhin enthält das föderative Schema Angaben zur Datenverteilung, um Operationen auf dem föderativen Schema auf die einzelnen LDBS abbilden zu können (diese Angaben könnten auch in einem separaten Verteilungsschema geführt werden, analog zu Verteilten DBS). Im allgemeinen Fall können unterschiedliche Föderationen gebildet werden, um die Bedürfnisse verschiedener Benutzergruppen abzudecken. Mit *externen Schemata* kann eine weitergehende Einschränkung der sichtbaren Daten und der zulässigen Operationen für einzelne Benutzer erreicht werden. Benutzer, die lediglich auf eine lokale Datenbank zugreifen, tun dies weiterhin über (in Abb. 12-2 nicht gezeigte) externe Schemata, welche direkt auf das betreffende LKS abgebildet werden.

Die Rolle der föderativen Schemata (sowie der externen Schemata) hängt davon ab, ob ein eng oder ein lose gekoppeltes FDBS realisiert werden soll. Im Falle einer engen Kopplung wird versucht, die Unterscheidung zwischen mehreren Datenbanken durch eine Schemaintegration aufzuheben. In diesem Fall entspricht das föderative Schema einem *globalen konzeptionellen Schema*, mit dem eine weitgehende Verteilungstransparenz - ähnlich wie bei Verteilten DBS - erreicht werden soll. Die Schemaintegration ist transparent für die Benutzer und wird durch *globale Datenbankadministratoren (GDBA)* vorgenommen. Das globale Schema be-

steht dabei nicht nur aus der Vereinigung der einzelnen Export-Schemata, sondern es wird vor allem eine Behandlung der semantischen Heterogenität vorgenommen.
Im Falle der lose gekoppelten FDBS erfolgt keine Schemaintegration, sondern die Unterscheidung mehrerer Datenbanken bleibt für den Benutzer sichtbar. Jeder Benutzer erzeugt sich aus den benötigten Export-Schemata selbst ein föderatives Schema, das nun auch als *Import-Schema* bezeichnet wird. Dies geschieht typischerweise durch Operationen einer speziellen Multi-DB-Anfragesprache, mit der auch Beziehungen zwischen Objekten verschiedener Export-Schemata definiert werden können (s.u.). Zusätzliche externe Schemata sind bei der losen Kopplung i.a. nicht erforderlich, da das Import-Schema bereits auf die spezifischen Bedürfnisse eines Benutzers abgestimmt werden kann.

Im folgenden diskutieren wir zunächst, welche Arten der semantischen Heterogenität in FDBS vorliegen können. Danach skizzieren wir den Prozeß der Schemaintegration in eng gekoppelten FDBS sowie den Einsatz einer Multi-DB-Anfragesprache in lose gekoppelten FDBS. Abschließend gehen wir auf die Transaktionsverwaltung in FDBS ein.

12.2 Semantische Heterogenität

Selbst wenn alle LDBS identisch sind, also keine Heterogenität in den Datenmodellen sowie der internen Realisierung besteht, bleibt das schwierige Problem der semantischen Heterogenität (Kap. 10.2) zu lösen. Dieses ist in erster Linie eine Folge der Entwurfsautonomie, welche den unabhängigen Entwurf des logischen (und physischen) Aufbaus der einzelnen Datenbanken gestattet. Im Rahmen der Schemaintegration wird versucht, die semantische Heterogenität zu beheben und eine möglichst widerspruchsfreie Datenbankstruktur anzubieten. Bei lose gekoppelten FDBS ist die Behandlung semantischer Heterogenität Aufgabe der Benutzer. Bevor auf diese Ansätze eingegangen wird, sollen hier zunächst die wichtigsten Formen der semantischen Heterogenität näher vorgestellt werden.

Nach [KS91] äußert sich semantische Heterogenität in Form von Schemakonflikten sowie Datenkonflikten. *Schemakonflikte* können weiterhin unterteilt werden in Namenskonflikte, strukturelle Konflikte sowie Konflikte bei Integritätsbedingungen. *Datenkonflikte* beruhen entweder auf unterschiedlicher Datenrepräsentation oder fehlenden bzw. widersprüchlichen Datenbankinhalten. Diese Konfliktarten sollen im folgenden näher diskutiert werden. Wir nehmen dazu an, daß die LDBS das relationale Modell unterstützen (bzw. dieses als CDM verwendet wird). Einige der Konflikte können mit folgendem Beispiel illustriert werden.

Beispiel 12-1

Wir betrachten zwei relationale Datenbanken UNIBIB und STADTBIB zur Verwaltung bibliographischer Angaben. Es werden lediglich die Relationen- und Attributnamen spezifiziert, wobei Primärschlüssel durch Unterstreichung gekennzeichnet sind.

UNIBIB	PUBLIKATION (Pubnr, Titel, Typcode) BUCHPUB (Pubnr, Verlag, Ejahr, #Exemplare, ISBN) VERFASSER (Pubnr, Vname) SCHLAGWORT (Pubnr, Sname)
STADTBIB	BUCH (ISBN, Titel, Autor, Vnr, Jahr, Preis, Standort) VERLAG (Vnr, Vname, Adresse)

Beide Datenbanken weisen ähnliche Informationen auf, jedoch bestehen in der Informationsstrukturierung und -benennung sowie im Umfang einige Unterschiede. In der Datenbank STADTBIB werden so lediglich Bücher verwaltet, während in der UNIBIB unterschiedliche Publikationen Aufnahme finden (jeweils mit einem Typcode gekennzeichnet). Außerdem werden in der UNIBIB Schlagworte unterstützt; Angaben zum Buchpreis liegen nur in der STADTBIB vor. Weitere Unterschiede werden im Text aufgezeigt.

Für relationale Datenbanken beziehen sich Schemakonflikte v.a. auf Relationen und Attribute. *Namenskonflikte* treten in zwei Varianten auf: Synonyme und Homonyme. *Synonyme* liegen vor, wenn zwei identische bzw. semantisch äquivalente Objekte (Relationen, Attribute) unterschiedliche Namen tragen. Unterschiedliche Objekte mit demselben Namen dagegen repräsentieren *Homonyme*. In obigem Beispiel sind die Attribute "Verlag" in BUCHPUB (UNIBIB) und "Vname" in VERLAG (STADTBIB) Synonyme, ebenso "Ejahr" (BUCHPUB) und "Jahr" (BUCH). Die Attribute "Vname" in VERFASSER bzw. in VERLAG sind dagegen Homonyme (Verfasser- vs. Verlagsname).

Strukturelle Konflikte treten in vielfältiger Weise auf. So können semantisch äquivalente Informationen (Objekte, Beziehungen) entweder als Relationen oder als Attribute repräsentiert werden. Dabei ist eine Abbildung auf unterschiedlich viele Relationen sowie unterschiedlich viele Attribute möglich. Im Beispiel liegt so in STADTBIB eine eigene Relation VERLAG vor, während in UNIBIB "Verlag" ein einfaches Attribut ist. Umgekehrt werden Angaben zu Autoren in UNIBIB innerhalb einer eigenen Relation geführt (VERFASSER), um mehrere Autoren pro Publikation gleichrangig berücksichtigen zu können, während in STADTBIB ein einfaches Attribut (Autor) verwendet wird. Buchangaben sind primär in den Relationen BUCHPUB bzw. BUCH repräsentiert, welche unterschiedlich viele Attribute aufweisen. Insgesamt sind die bibliographischen Angaben über unterschiedlich viele (4 vs. 2) Relationen verstreut.

Konflikte bezüglich Integritätsbedingungen beziehen sich auf Relationenebene auf Unterschiede bei der Definition der Primärschlüssel und weiteren Schlüsselkandidaten, die Festlegung der Fremdschlüssel sowie unterschiedliche Aktionen zur Wartung der referentiellen Integrität sowie sonstige anwendungsspezifische Integritätsbedingungen, wie sie in SQL92 mit der CHECK-Klausel definiert wer-

den können. Auf Attributebene können Unterschiede beim Datentyp, dem Wertebereich, bei Default-Werten, bezüglich der Zulässigkeit von Nullwerten sowie allgemeinen CHECK-Bedingungen Probleme bereiten.

Im Beispiel liegen unterschiedliche Primärschlüssel vor. In UNIBIB wird hierzu eine in der anderen Datenbank unbekannte "Pubnr" verwendet, während in STADTBIB Bücher über ihre ISBN identifiziert werden. Dieses Attribut existiert auch in UNIBIB, könnte dort jedoch optional besetzt werden (bei Zulässigkeit des Nullwertes). Dann wären möglicherweise übereinstimmende Bücher nicht ausfindig zu machen. Auch bei den Attributen bezüglich Titel-, Autoren- oder Verlagsangaben könnten unterschiedliche Datentypen bzw. sonstige Abweichungen bezüglich Integritätsbedingungen vorliegen, welche die Vergleichbarkeit einschränken.

Datenkonflikte betreffen Unterschiede hinsichtlich der DB-Inhalte, welche auf Schemaebene nicht erkennbar sind. Eine Konfliktursache liegt dabei in der Verwendung unterschiedlicher Datenrepräsentationen, wie sie auch bei übereinstimmenden Datentypen und Wertebereichen möglich ist. So kann bei Preisangaben die Vergleichbarkeit durch Verwendung unterschiedlicher Währungen und Mehrwertsteuer-Berücksichtigung erschwert werden. Für Namen (z.B. Autorennamen) können unterschiedliche Regelungen vorliegen bezüglich der Reihenfolge von Vor- und Nachnamen, der Verwendung von Trennzeichen zwischen Namensteilen, der Behandlung akademischer Titel, etc. So dürfte es z.B. schwierig sein, automatisch zu erkennen, daß die Angaben "Prof. Dr. Theo Härder", "Härder, Theo" und "HAERDER THEO" jeweils dieselbe Person bezeichnen.
Ähnliche Datenkonflikte entstehen durch falsche bzw. unvollständige DB-Inhalte, wie sie z.B. durch Eingabefehler oder unzureichende Informationen entstehen können (z.B. Autorennamen "T. Härder", "Harder, Th."). Widersprüchliche DB-Inhalte ergeben sich auch, wenn repliziert vorliegende Daten nicht zeitgleich geändert werden (z.B. Verlagsadresse).

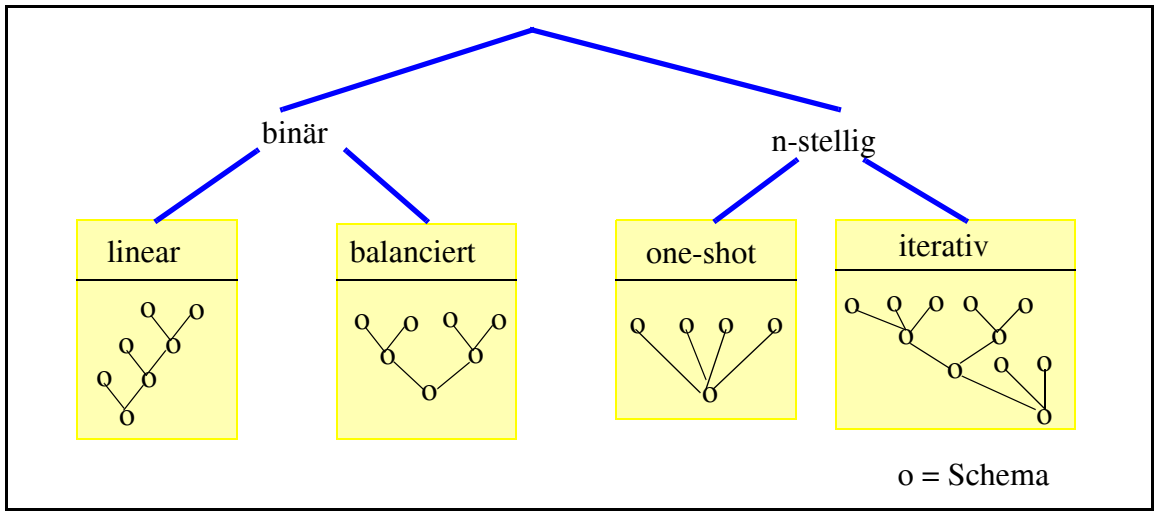
12.3 Schemaintegration

Ziel der Schemaintegration in eng gekoppelten FDBS ist es, aus mehreren Export-Schemata ein globales konzeptionelles Schema zu erstellen. Damit soll die Unterscheidung mehrerer unabhängiger Datenbanken für den Benutzer des globalen Schemas verborgen werden. Die Schemaintegration erfolgt durch einen oder mehrere GDBA. Dieser Prozeß ist, sofern er überhaupt erfolgreich durchgeführt werden kann, weitgehend manuell vorzunehmen. Denn vor allem die durch die semantische Heterogenität verursachten Konflikte können nur zu einem geringen Teil automatisch behandelt werden. Allerdings soll der Integrationsprozeß durch geeignete Tools für die GDBA soweit als möglich erleichtert werden.

In [BLN86] wurden zur Schemaintegration unterschiedliche Integrationsstrategien gegenübergestellt (Abb. 12-3). Dabei wird zwischen binären und n-stelligen

Ansätzen unterschieden je nachdem, ob jeweils zwei oder mehrere (n) Schemata pro Integrationsschritt zusammengeführt werden. Bei der n-stelligen Vorgehensweise kann versucht werden, alle Schemata auf einmal zu integrieren ("one shot"), oder mehrere iterative Integrationsschritte vorzunehmen. In letzterem Fall werden die Zwischenergebnisse vorhergehender Integrationsschritte mit noch nicht berücksichtigten Schemata zusammengeführt. Binäre Strategien erfordern bei mehr als zwei Schemata stets mehrere Integrationsschritte, welche linear oder balanciert (bzw. nicht-linear) ausgeführt werden können (Abb. 12-3). Bei der Beschränkung auf je zwei Schemata kann jeweils nur eine Teilmenge der Schemainformation berücksichtigt werden. Dafür vereinfacht sich der Integrationsprozeß, was auch eine Tool-Unterstützung erleichtert.

Abb. 12-3: Strategien zur Schemaintegration



Die eigentliche Schemaintegration umfaßt die Phasen der Vorintegration, Erkennung und Behebung von Schemakonflikten sowie Mischen und Restrukturierung der Schemaangaben [BLN86]. Im Rahmen der *Vorintegration* werden die Integrationsstrategie (s.o.) sowie die Reihenfolge, in der die Schemata integriert werden, festgelegt. Weiterhin werden in den beteiligten Schemata die Schlüsselkandidaten bestimmt, um etwaige Abhängigkeiten zwischen den Schemata erkennen zu können. Zudem werden potentiell äquivalente Wertebereiche sowie *Transformationsabbildungen* (mapping functions) zwischen ihnen (falls erforderlich) festgelegt. Transformationsfunktionen sind auch zur Angleichung unterschiedlicher Repräsentationen vorzusehen (Behebung von Datenkonflikten). Damit kann z.B. eine Umwandlung zwischen numerischen und nicht-numerischen Datentypen oder eine Vereinheitlichung von Preisangaben unterschiedlicher Währungen erfolgen. Um Änderungen auf derart transformierten Daten zu ermöglichen, sind Umkehrfunktionen zu spezifizieren, welche die Transformation rückgängig machen.

In der Phase der *Konflikterkennung und -behandlung* sind die aufgrund semantischer Heterogenität entstandenen Probleme festzustellen und zu beheben. Dabei ist die Behandlung von Namenskonflikten am unproblematischsten, da hierzu die Umbenennung der betroffenen Objekte genügt. Die Erkennung von Homonymen ist dabei am einfachsten und leicht automatisierbar. Dagegen existiert für die Behandlung struktureller Konflikte sowie von Konflikten hinsichtlich Integritätsbedingungen keine allgemein anwendbare Vorgehensweise. Dies muß manuell durch den GDBA geschehen unter Berücksichtigung der Anwendungssemantik. Allenfalls für häufig vorkommende Transformationsschritte (z.B. Abbildung von Attributen nach Relationen und umgekehrt) kann eine Teilunterstützung erfolgen.

Noch weniger automatisierbar ist die abschließende Phase des *Mischens und Restrukturierens*. Dabei sollen die Angaben der einzelnen Schemata zusammengeführt werden, so daß keine Informationen verlorengehen (Vollständigkeit). Das erzeugte Schema soll ferner minimal sein, das heißt, möglichst keine Redundanz aufweisen, sowie für den Benutzer leicht verständlich sein. Die Beseitigung der Redundanz ist besonders wesentlich für Änderungsoperationen, da die in den lokalen Datenbanken vorliegende Redundanz nun nicht mehr vom Benutzer zu warten ist. Vielmehr kann durch automatische Anwendung der Änderungen auf allen betroffenen Kopien die globale Konsistenz der Daten sichergestellt werden.

Beispiel 12-2

Für die beiden Datenbanken aus Beispiel 12-1 soll eine Schemaintegration vorgenommen werden. Durch die Beschränkung auf zwei Schemata entfällt die Wahl einer Integrationsstrategie. Zur Vorintegration nehmen wir an, daß für das Attribut ISBN in BUCHPUB keine Nullwerte zugelassen sind (Schlüsselkandidat). Nach Auflösung der Namenskonflikte (*kursiv* gezeigte Attribute) erhalten wir folgende Relationen:

PUBLIKATION (Pubnr, Titel, Typcode)
BUCHPUB (Pubnr, *Vname*, *Jahr*, #Exemplare, ISBN)
VERFASSER (Pubnr, *Autor*)
SCHLAGWORT (Pubnr, Sname)

BUCH (ISBN, Titel, Autor, Vnr, Jahr, Preis, Standort)
VERLAG (Vnr, Vname, Adresse)

Die Behebung der anderen Konflikte wird dadurch erschwert, daß die beiden Datenbanken auf Primär- bzw. Fremdschlüsseln basieren, die in der anderen Datenbank unbekannt sind. So ist "Pubnr" nur in der ersten, "Vnr" nur in der zweiten Datenbank bekannt. Ähnliches gilt für die einfachen Attribute "Typcode", "#Exemplare" oder "Standort", die keine übergreifende Bedeutung haben. Probleme bereitet auch die unterschiedliche Behandlung von Autoren.
Zur Behandlung dieser Schwierigkeiten nehmen wir an, daß die Unterschiede über vom GDBA zu spezifizierende Transformationsfunktionen angeglichen werden können. Insbesondere sei es möglich, zu einer Buch-ISBN einen Pubnr-Wert sowie zu einem Verlagsnamen einen Vnr-Wert zu bestimmen. Liegen der ISBN- bzw. Vname-Wert bereits in der BUCHPUB- bzw. VERLAG-Relation vor, ergibt sich die Zuordnung aus dem Inhalt dieser Relationen. Anderenfalls sind durch die Transformationsabbildung neue Nummern zu generieren, die noch nicht vergeben sind. Mit einer weiteren Transformationsfunktion sei es möglich, aus dem Attribut "Autor" in BUCH die Namen der einzelnen Verfasser zu extra-

hieren und, falls erforderlich, in dieselbe Repräsentation wie für das Attribut "Autor" in VERFASSER zu überführen. Unter diesen Voraussetzungen kann folgender Integrationsansatz gewählt werden, der dem Benutzer den direkten Zugriff auf den gemeinsamen Buchbestand der zwei Bibliotheken gestattet:

PUBLIKATION (Pubnr, Titel, Typcode)
BUCHP (Pubnr, Vnr, Jahr, Preis, Standort-STADT, #Ex-UNI, ISBN)
VERFASSER (Pubnr, Autor)
SCHLAGWORT (Pubnr, Sname)
VERLAG (Vnr, Vname, Adresse).

Dabei wurden die Attribute der BUCH-Relation unter Anwendung der Transformationen für ISBN und Autor auf die neue Relation BUCHP sowie die Relationen PUBLIKATION (Titel) und VERFASSER (Autor) abgebildet. Die Angaben der Relation BUCHPUB finden sich weitgehend in BUCHP; lediglich der Verlagsname wird jetzt in VERLAG geführt (Anwendung der Transformationsfunktion auf Vname). Die Attribute "Standort" und "#Exemplare" wurden umbenannt, um anzudeuten, daß sie sich auf jeweils eine bestimmte Bibliothek beziehen. Das globale Schema ist vollständig, minimal und redundanzfrei, da jedes Attribut (bis auf die Fremdschlüssel "Pubnr" und "Vnr") genau einmal vorkommt.

Eine Reihe neuerer Forschungsarbeiten favorisiert zur Schemaintegration ein objektorientiertes Datenmodell für das globale Schema [Ah91, HD92, FL93]. Denn die im Vergleich zum Relationenmodell größere Modellierungsmächtigkeit erleichtert die Behandlung von Schema- und Datenkonflikten. Als besonders hilfreich werden Abstraktionskonzepte wie die Generalisierung angesehen, mit der hierarchische Teilmengenbeziehungen zwischen Satztypen (Objektklassen) definiert werden können*. Weiterhin lassen sich durch Definition objekttypspezifischer Methoden (Funktionen) mächtige Transformationen realisieren, wie sie z.B. in obigem Beispiel bereits vorausgesetzt wurden. Auf der anderen Seite entsteht durch die Wahl eines mächtigen (objekt-orientierten) Datenmodells das Problem, daß Konzepte und Operationen des globalen Modells von den LDBS nicht unterstützt werden. Dieses Problem kann jedoch auch bei einem relationalen globalen Datenmodell auftreten, wenn die LDBS etwa auf dem hierarchischen Datenmodell beruhen. In solchen Fällen muß i.a. eine eingeschränkte Funktionalität in Kauf genommen werden (womit die Verteilungstransparenz beeinträchtigt wird).

Ein weiteres (bereits erwähntes) Problem der Schemaintegration liegt darin, daß dieser Prozeß weitgehend manuell erfolgen muß. Um semantische Konflikte beheben zu können, sind seitens der GDBA genaue Kenntnisse über den Aufbau aller an einer Föderation beteiligten Datenbanken erforderlich. Dieses zentralisierte Wissen dürfte jedoch vor allem bei Datenbanken unterschiedlicher Unternehmen bzw. Organisationen oft nicht vorliegen bzw. preisgegeben werden, da hiermit eine Einschränkung der Autonomie verbunden ist. Generell wird der manuelle Ansatz

* In Beispiel 12-2 entspricht so der Satztyp BUCHP einer Teilmenge (Spezialisierung) des Satztyps PUBLIKATION. Die Definition einer Generalisierungshierarchie zwischen diesen Satztypen impliziert, daß Attribute und sonstige Eigenschaften von PUBLIKATION auch für BUCHP-Objekte gelten (“vererbt” werden).

der Schemaintegration bei einer größeren Anzahl von Datenbanken schnell unpraktikabel. Schwierigkeiten bereiten dabei vor allem Änderungen in den lokalen Schemata, die aufgrund der Autonomie jederzeit möglich sind. Sofern diese Änderungen die Export-Schemata betreffen, ist eine teilweise Wiederholung der Schemaintegration vorzunehmen, was einen enormen Verwaltungsaufwand verursacht.

12.4 Einsatz einer Multi-DB-Anfragesprache

Die Probleme der Schemaintegration lassen in vielen Fällen lose gekoppelte FDBS als sinnvollere Alternative erscheinen, insbesondere bei einer großen Anzahl von Datenbanken sowie Datenbanken unterschiedlicher Unternehmen. Zur Erhöhung der Autonomie erfolgt keine Definition eines globalen Schemas, sondern der Benutzer sieht mehrere Datenbanken. Er kann jedoch mit einer einheitlichen Multi-DB-Anfragesprache auf die einzelnen Datenbanken zugreifen, wobei pro Operation Daten verschiedener Knoten verknüpft werden können. Ferner soll die Sprache eine Unterstützung zur Behandlung semantischer Heterogenität bieten. Derzeit folgen bereits einige große Information-Retrieval-Systeme diesem Ansatz, wobei Suchanfragen über viele unabhängige Datenbanken formuliert werden können [LMR90]. Im deutschen Bildschirmtext-System ist es auch möglich, mit einer einheitlichen Benutzeroberfläche auf zahlreiche Datenbanken lesend zuzugreifen, jedoch kann dabei nur mit jeweils einer Datenbank gearbeitet werden.

Um eine ausreichende Flexibilität für den Zugriff auf mehrere Datenbanken zu erhalten, sollte eine relationale Multi-DB-Anfragesprache nach [LMR90] u.a. folgende Eigenschaften unterstützen:

- Unterstützung erweiterter Objektnamen, welche den logischen Namen der lokalen Datenbank beinhalten. Damit wird auch die Eindeutigkeit von Objektnamen sichergestellt.
- Definition und explizite Benennung von *"Multi-Datenbanken"* durch Spezifikation der zu beteiligenden lokalen Datenbanken.
- Erzeugen, Ändern und Löschen von Relationen im Import-Schema einer Multi-DB. Für die Attribute einer Relation sollen dabei automatische Konversionen von Datentypen unterstützt werden. Man spricht hierbei auch von *dynamischen Attributen*, da für sie die Datenwerte zur Zugriffszeit dynamisch über eine Transformationsfunktion konvertiert werden.
- Lese- und Änderungsoperationen über mehrere lokale Datenbanken. Insbesondere sollen verteilte Joins über Attribute mit kompatiblen Wertebereichen möglich sein.
- Definition von Sichten (Views) auf Multi-Datenbanken.
- Definition von Abhängigkeiten zwischen Datenbanken durch den DBA, z.B. um äquivalente bzw. kompatible Wertebereiche oder Änderungsabhängigkeiten spezifizieren zu können. *Änderungsabhängigkeiten* (update dependencies) können zur Überwachung von globalen Integritätsbedingungen sowie zur automatischen Wartung

von Redundanz genutzt werden (Abweisung bestimmter Änderungsoperationen bzw. automatische Durchführung von Folgeänderungen).

Diese Eigenschaften werden (bis auf Änderungsabhängigkeiten) von dem Sprachvorschlag MSQL unterstützt, der eine Erweiterung von SQL darstellt und im MRDSM-Prototyp verwendet wird [LMR90]. Das folgende Beispiel illustriert die Verwendung einiger MSQL-Anweisungen.

Beispiel 12-3

Für die lokalen Datenbanken UNIBIB und STADTBIB aus Beispiel 12-1 kann mit folgender MSQL-Anweisung die Multi-Datenbank BIB erzeugt werden (MSQL-Schlüsselwörter sind *kursiv* gehalten):

CREATE MULTIDATABASE BIB (UNIBIB, STADTBIB).

Um alle Bücher zu ermitteln, die in beiden Datenbanken vorliegen, könnte folgende Join-Anfrage verwendet werden:

USE BIB
SELECT y.Titel
FROM UNIBIB.BUCH% x, STADTBIB.BUCH% y
WHERE x.ISBN = y.ISBN.

Die USE-Anweisung dient dabei zur Festlegung der Datenbank(en), auf die sich die Query bezieht. Das %-Zeichen in den Relationennamen dient als Abkürzung für eine beliebige Zeichenfolge. Damit bleiben Umbenennungen der lokalen Relationennamen ohne Auswirkungen auf die Anfrage, solange das Präfix "BUCH" erhalten bleibt (Erhöhung der Autonomie).

Die folgende Sicht-Definition illustriert den Einsatz eines dynamischen Attributs zur Umwandlung von Preisangaben für Bücher amerikanischer Verlage, welche durch eine Verlagsnummer von über 1000 gekennzeichnet seien.

CREATE VIEW US-BUCH (ISBN, Titel, Autor, DMPreis)
USE BIB
D-COLUMN (Preis) DMPreis = Preis * 1,70
SELECT ISBN, Titel, Autor, Preis
FROM STADTBIB.BUCH
WHERE Vnr > 1000.

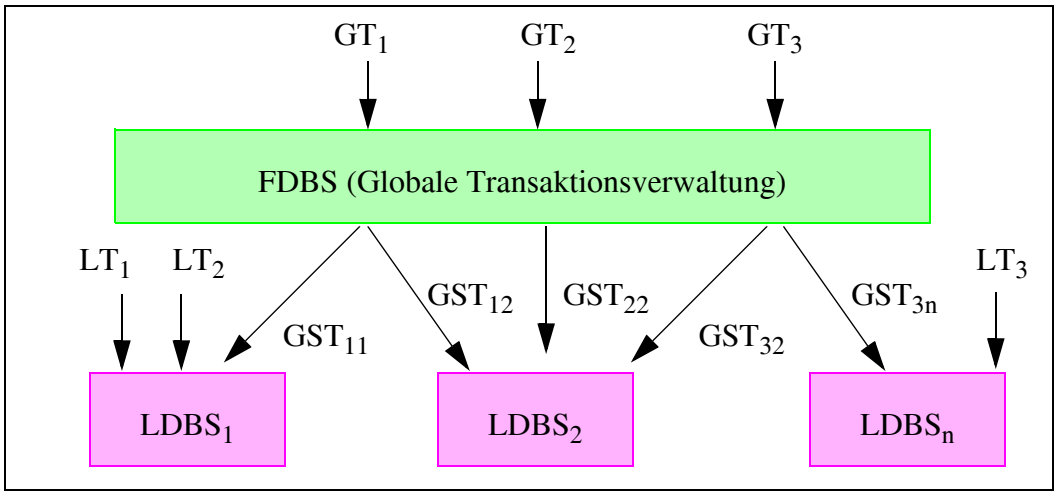
Die Unterstützung von Sichten auf Multi-Datenbanken ermöglicht eine Erhöhung der Verteilungstransparenz. Aufgrund der Änderungsproblematik auf Sichten sind dabei jedoch Änderungen nur sehr eingeschränkt möglich. Auch beim direkten Zugriff auf die einzelnen Datenbanken bereiten Änderungen Probleme, da der Benutzer weitgehend für die Überprüfung globaler Integritätsbedingungen sowie für die Redundanzwartung verantwortlich ist. Dies kann durch die Definition von Abhängigkeiten zwischen Datenbanken erleichtert werden, setzt jedoch die konsistente Spezifikation dieser Abhängigkeiten durch einen globalen DBA voraus, was wiederum die Autonomie beeinträchtigt. Aus diesen Gründen sind derzeit meist nur Lesezugriffe auf Multi-Datenbanken möglich.

Damit der Ansatz der lose gekoppelten FDBS eine größere Bedeutung erlangen kann, ist eine Standardisierung der Multi-DB-Anfragesprache erforderlich. Die im Rahmen von SQL Access vorgenommenen Spracherweiterungen von SQL gehen in diese Richtung, jedoch bleibt die Funktionalität noch weit hinter oben genannten Forderungen zurück. Insbesondere kann innerhalb einer DB-Operation nicht auf mehrere Datenbanken zugegriffen werden.

12.5 Transaktionsverwaltung

Die Unterstützung von globalen Änderungstransaktionen in FDBS wirft große Probleme hinsichtlich der Transaktionsverwaltung auf, wenn keine Erweiterung der LDBS in Kauf genommen wird. Gemäß der zweigeteilten Architektur von FDBS (Abb. 12-1) erfolgt auch die Transaktionsverwaltung zweistufig: neben einer lokalen Transaktionsverwaltung in jedem LDBS liegt eine globale Transaktionsverwaltung in der FDBS-Zusatzschicht vor (Abb. 12-4). Globale Transaktionen (GT) werden von der globalen Transaktionsverwaltung in eine Folge von lokal ausführbaren Sub-Transaktionen (GST) zerlegt. Die LDBS kennen bei vollständiger Autonomie keinen Unterschied zwischen lokalen Transaktionen (LT) und GST. Ferner soll keine Kooperation zwischen den LDBS und der globalen Transaktionsverwaltung sowie unter den LDBS erfolgen. Schließlich kann bei den lokalen Transaktionsverwaltungen Heterogenität vorliegen, insbesondere bei der Synchronisation (z.B. können unterschiedliche Sperrverfahren oder eine optimistische Synchronisation im Einsatz sein).

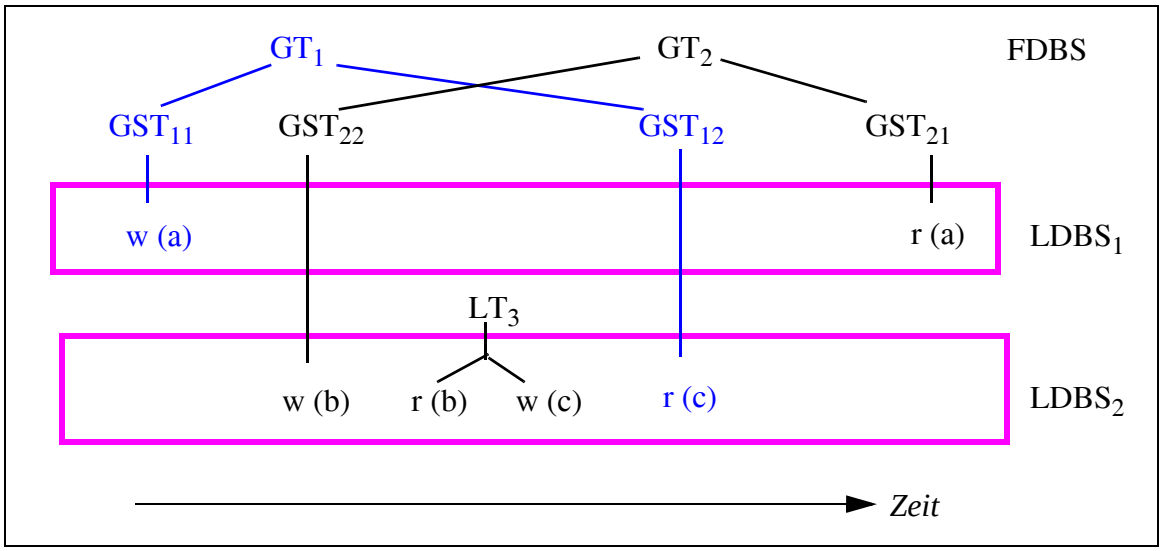
Abb. 12-4: Transaktionsverwaltung in FDBS



Unter diesen Voraussetzungen ist es äußerst schwer, die ACID-Eigenschaften für globale Transaktionen zu wahren. Die automatische Unterstützung globaler Integritätsbedingungen (Eigenschaft C) setzt deren Spezifikation durch den GDBA voraus, so daß sie prinzipiell vom FDBS bei der Abarbeitung globaler Änderungs-

operationen überwacht weren können. Die Eigenschaften A, I und D verlangen die Gewährleistung der globalen Serialisierbarkeit sowie die Realisierung eines korrekten Commit-Protokolls zwischen den an der Föderation beteiligten Knoten. Die Realisierung eines globalen Commit-Protokolls wird bereits dadurch nahezu unmöglich, da die LDBS keinen Unterschied zwischen lokalen Transaktionen und globalen Sub-Transaktionen machen. Daher nimmt ein LDBS am Ende einer GST bereits das Commit für diese (Sub-) Transaktion vor, so daß deren Änderungen zu diesem Zeitpunkt dauerhaft und sichtbar werden. Ein Rücksetzen dieser Sub-Transaktion, wie beim Scheitern der globalen Transaktion erforderlich, ist damit prinzipiell nicht mehr möglich. Die globale Transaktionsverwaltung kann höchstens noch versuchen, die Änderungen der Sub-Transaktion durch eine sogenannte Kompensationstransaktion zurückzunehmen [SW91]. Die lokale Sichtbarkeit von aus globaler Sicht ungültigen Änderungen kann damit jedoch nicht verhindert werden.

Abb. 12-5: Synchronisationsproblem in FDBS



Die Entkopplung von globaler und lokaler Transaktionsverwaltung verursacht ähnlich schwerwiegende Probleme für die Synchronisation. Zur Wahrung der globalen Serialisierbarkeit bereiten dabei vor allem Konflikte mit lokalen Transaktionen Schwierigkeiten, da diese der globalen Transaktionsverwaltung unbekannt sind. So ist im Beispiel von Abb. 12-5 aus globaler Sicht keine Verletzung der Serialisierbarkeit erkennbar, da nur das Objekt a von beiden globalen Transaktionen referenziert wird. Da GT_1 als erstes auf a zugreift, besteht die Abhängigkeit (der Konflikt) $GT_1 < GT_2$.
Auch liegt in beiden LDBS keine Verletzung der lokalen Serialisierbarkeit vor. Jedoch führen die Zugriffe der lokalen Transaktion LT_3 zu der indirekten (transitiven) Abhängigkeit $GT_2 < LT_3 < GT_1$
zwischen den beiden globalen Transaktionen. Damit besteht eine zyklische Abhän-

gigkeit zwischen GT_1 und GT_2, so daß die globale Serialisierbarkeit verletzt wird. Dies wird jedoch nicht erkannt, da die globale Transaktionsverwaltung die transitive Abhängigkeit nicht mitbekommt.

Die Lösung der genannten Probleme erfordert Abstriche in einem oder mehreren der folgenden Bereiche:

- Funktionale Beschränkungen, z.B. bezüglich der Unterstützung globaler Änderungstransaktionen.
- Reduzierung der LDBS-Autonomie, insbesondere in Form von vorzunehmenden Erweiterungen der lokalen Transaktionsverwaltung.
- Beschränkung der Heterogenität, z.B. Verwendung identischer Synchronisations- und Commit-Protokolle.
- Reduzierung der ACID-Zusicherungen, z.B. Verzicht auf globale Serialisierbarkeit.

Es ist klar, daß diese Punkte auf verschiedenste Weise berücksichtigt werden können. Einfache "Ansätze" bestehen so darin, keine globalen Änderungstransaktionen zuzulassen bzw. auf eine ändernde Sub-Transaktion zu beschränken (Kap. 11.3.1). Für den allgemeinen Fall mit beliebig vielen ändernden Sub-Transaktionen besitzt der ebenfalls in Kap. 11.3.1 diskutierte Ansatz zur Transaktionsverwaltung für die Praxis die größte Bedeutung. Er sieht vor, daß alle LDBS ein striktes Zwei-Phasen-Sperrprotokoll sowie einen Timeout-Ansatz zur Behandlung globaler Deadlocks unterstützen. Ferner müssen die LDBS an einem globalen Commit-Protokoll teilnehmen (über die XA-Schnittstelle). Mit diesen relativ geringen LDBS-Erweiterungen werden die Eigenschaften A, I und D für globale Transaktionen sichergestellt. Ferner ergibt sich auch für die globale Transaktionsverwaltung eine einfache Realisierung, da diese im wesentlichen auf die Durchführung des globalen Commit-Protokolls beschränkt ist. Weitere Vorteile sind, daß außer für das Commit-Protokoll keine zusätzlichen Nachrichten zur globalen Transaktionsverwaltung anfallen und daß jeder Knoten als globaler Commit-Koordinator fungieren kann (verteilte Realisierung der globalen Transaktionsverwaltung). Die in Kauf zu nehmenden Einschränkungen sind eine reduzierte Autonomie (LDBS-Erweiterungen, Abhängigkeit zum globalen Commit-Koordinator) sowie weitgehender Verzicht auf Heterogenität bei der lokalen Transaktionsverwaltung.

In der Forschung wurde in den letzten Jahren eine Vielzahl anderer Ansätze zur Transaktionsverwaltung in FDBS vorgeschlagen, die einen höheren Grad an Autonomie und Heterogenität anstreben. Die praktische Tauglichkeit vieler Vorschläge ist jedoch fraglich, da sie häufig auf zweifelhaften Annahmen basieren; zudem wurde kaum eines der vorgeschlagenen Verfahren tatsächlich implementiert. So wurde vielfach ein zentralisierter globaler Transaktions-Manager unterstellt, über den sämtliche globale Transaktionen ausgeführt werden müssen. Dies ist vor

allem für geographisch verteilte Systeme aus Leistungs- und Autonomiegründen eine inakzeptable Lösung. Ferner wurde häufig angenommen, daß der globale Transaktionsverwalter zur Synchronisation die Objektreferenzen (i.a. Sätze oder Seiten) globaler Transaktionen genau kennt. Selbst dies kann jedoch bei entkoppelten LDBS nicht vorausgesetzt werden, da die genauen Objektreferenzen sich i.a. erst bei der Abarbeitung im LDBS ergeben. Andere Ansätze verursachen (aufgrund pessimistischer Konfliktannahmen) große Einschränkungen hinsichtlich der Parallelität zwischen globalen Transaktionen, so daß das Leistungsverhalten stark beeinträchtigt wird. Schließlich betrachten eine Reihe von Arbeiten lediglich die Synchronisationsproblematik, so daß die in engem Zusammenhang stehende Commit-Behandlung ungelöst bleibt. Wir wollen im folgenden stellvertretend zwei Vorschläge näher diskutieren, nämlich Commit-Ordnung und Quasi-Serialisierbarkeit. Eine Übersicht zu weiteren Ansätzen bietet [VG93].

Commit-Ordnung [Raz92, Raz93]

Das Prinzip der Commit-Ordnung (Commitment Ordering, CO) ist ein allgemeiner Ansatz zur Gewährleistung von Serialisierbarkeit, der sich vorteilhaft auf föderative DBS übertragen läßt. Ein Transaktions-Schedule erfüllt die *CO-Eigenschaft*, wenn die Commit-Operationen von in Konflikt stehenden Transaktionen in der gleichen Reihenfolge ausgeführt werden wie die Konflikt-Operationen selbst. In [Raz92] wurde gezeigt, daß die Erfüllung der CO-Eigenschaft eine hinreichende Bedingung für Serialisierbarkeit ist. Mit einem strikten Zwei-Phasen-Sperrprotokoll wird die CO-Eigenschaft offenbar immer garantiert, jedoch läßt sich dies auch für andere, deadlock-freie (z.B. optimistische) Synchronisationsverfahren erreichen.

Beispiel 12-4

Von den beiden Schedules ... $w_1(x), r_2(x), c_1, c_2$

und ... $w_1(x), r_2(x), c_2, c_1$

erfüllt lediglich der erste die CO-Eigenschaft, da die Commit-Operationen (c_i) der beiden in Konflikt stehenden Transaktionen T_1 und T_2 in der gleichen Reihenfolge ausgeführt werden, wie ihre Konflikt-Operationen auf Objekt x. Der zweite Schedule ist serialisierbar ($T_1 < T_2$), obwohl die CO-Eigenschaft nicht erfüllt ist. Dies zeigt, daß die CO-Bedingung keine notwendige Voraussetzung für Serialisierbarkeit darstellt.
Bei einem Sperrverfahren ist der zweite Schedule nicht möglich, da beim Lesezugriff von T_2 ein Sperrkonflikt mit T_1 entsteht, der erst nach dem Commit von T_1 (Sperrfreigabe) behoben wird (c_2 muß somit nach c_1 kommen). Bei einem nicht-blockierenden, CO-konformen Synchronisationsprotokoll wird für den zweiten Schedule bei der Operation c_2 erkannt, daß eine Verletzung der CO-Eigenschaft droht. Dies kann z.B. durch Abbruch von T_1 verhindert werden.

Für föderative DBS ist es nun einfach, die globale Serialisierbarkeit sicherzustellen, sofern jedes LDBS ein Synchronisationsverfahren verwendet, das die CO-Eigenschaft bezüglich der bei ihm ausgeführten (Sub-) Transaktionen gewährleistet

(diese Bedingung allein ist aber offenbar nicht ausreichend, da die lokale Serialisierbarkeit in jedem Knoten keine globale Serialisierbarkeit garantiert). Dazu genügt es, daß jedes LDBS an einem globalen Commit-Protokoll teilnimmt (z.B. einem verteilten Zwei-Phasen-Commit-Protokoll), so daß die LDBS eine externe Commit-Entscheidung akzeptieren. Während der lokalen Commit-Behandlung einer Transaktion T müssen dabei alle Transaktionen abgebrochen werden, die aufgrund lokal erfolgter Konflikte vor T hätten beendet werden müssen. Im Beispiel von Abb. 12-5 würde so beim globalen Commit von GT_1 (nach Beendigung von GST_{12}) in $LDBS_2$ erkannt werden, daß GST_{22} noch nicht beendet ist, obwohl diese Transaktion in der lokalen CO-Reihenfolge vor GST_{12} steht. Durch Abbruch von GST_{22} (und damit von GT_2) wird das Problem behoben und die globale Serialisierbarkeit gewahrt.

Dieses Protokoll stellt offenbar eine Verallgemeinerung des oben diskutierten Ansatzes dar, bei dem neben der Teilnahme an einem globalen Commit-Protokoll vorausgesetzt wurde, daß jedes LDBS ein striktes Zwei-Phasen-Sperrprotokoll verwendet. Der skizzierte CO-Ansatz erlaubt jedoch Heterogenität bei der lokalen Synchronisationskomponente, solange die CO-Eigenschaft gewahrt bleibt. Insbesondere können deadlock-freie Synchronisationsverfahren unterstützt werden. In Kauf zu nehmen sind wiederum Autonomie-Einschränkungen aufgrund des globalen Commit-Protokolls sowie (möglicherweise) Erweiterungen der LDBS zur Gewährleistung der CO- Eigenschaft.

Bei der Überprüfung der CO-Eigenschaft erfolgt im Basisverfahren keine Unterscheidung zwischen lokalen Transaktionen und globalen Sub-Transaktionen. Wenn in den LDBS jedoch eine Unterscheidung zwischen diesen beiden Transaktionstypen möglich ist, kann dies dazu genutzt werden, die CO-Überprüfung auf globale Transaktionen zu beschränken, sofern das lokale Synchronisationsprotokoll lokale Serialisierbarkeit garantiert [Raz93]. Mit diesem erweiterten Protokoll kann eine verringerte Abbruchhäufigkeit für lokale Transaktionen erwartet werden.

Quasi-Serialisierbarkeit

Eine Reihe von Forschungsarbeiten versuchte, die Transaktionsverwaltung in FDBS durch Unterstützung schwächerer Korrektheitskriterien als der globalen Serialisierbarkeit zu erleichtern. Ein solches Kriterium ist die Quasi-Serialisierbarkeit [DE89, DEK91]. *Quasi-Serialisierbarkeit* verlangt, daß alle lokalen Schedules serialisierbar sind und die Ausführung globaler Transaktionen äquivalent ist zu einer quasi-seriellen Ausführung dieser Transaktionen, bei der globale Transaktionen sequentiell ausgeführt werden. Die Quasi-Serialisierungsreihenfolge ist durch die äquivalente quasi-serielle Ausführungsfolge gegeben und bezieht sich ausschließlich auf globale Transaktionen.

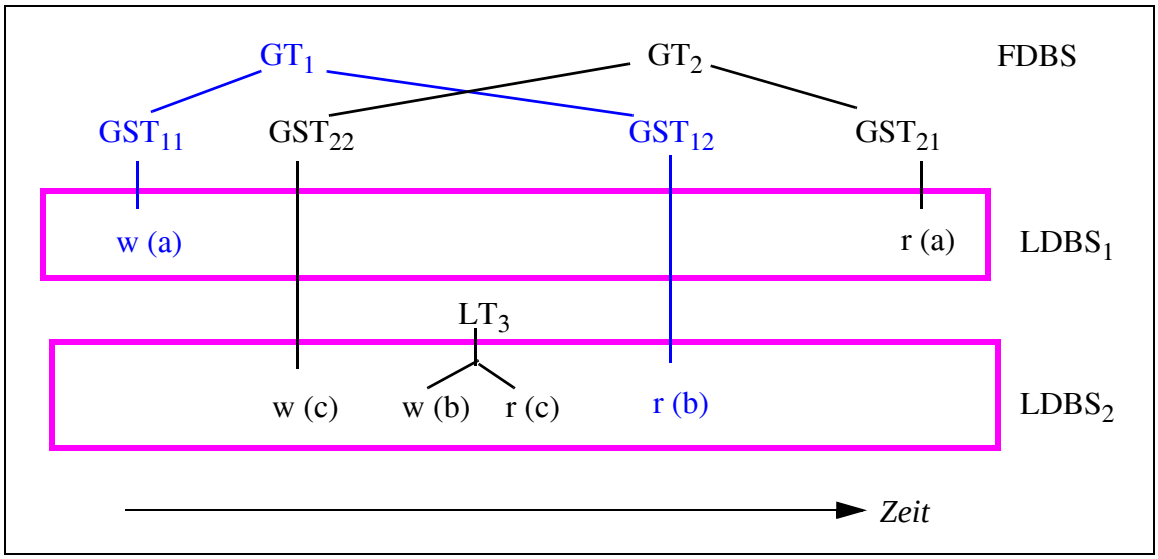
Beispiel 12-5

Die in Abb. 12-6 gezeigte Ausführung von Transaktionen ist nicht serialisierbar, da in LDBS1 die Abhängigkeit $GT_1 < GT_2$ und in LDBS2 die Abhängigkeiten $GT_2 < LT_3 < GT_1$ existieren. Jedoch liegt Quasi-Serialisierbarkeit vor, da die lokalen Schedules serialisierbar sind und der Schedule in LDBS2 äquivalent ist zu folgender Ausführungsreihenfolge, bei der GST_{12} vor GST_{22} ausgeführt wird:

$$w_3\,(b),\ r_1\,(b),\ w_2\,(c),\ r_3\,(c).$$

Dabei bezieht sich der Subskript 3 auf die lokale Transaktion LT_3, die Subskripte 1 bzw. 2 auf die globalen Sub-Transaktionen GST_{12} bzw. GST_{22}.

Abb. 12-6: Quasi-serialisierbare (jedoch nicht serialisierbare) Transaktionsausführung



Somit ist die gezeigte Ausführung der beiden globalen Transaktionen äquivalent zu einer (quasi-)seriellen Ausführung, bei der GT_1 vor GT_2 ausgeführt wird. Der indirekte Konflikt in LDBS2 mit der lokalen Transaktion LT_3 ist also für die Quasi-Serialisierbarkeit irrelevant, da es möglich war, eine äquivalente Ausführungsfolge zu finden, bei der sämtliche Operationen von GT_1 vor denen von GT_2 ausgeführt wurden.

Dies ist jedoch im Beispiel von Abb. 12-5 nicht der Fall, da dort in LDBS2 relevante Abhängigkeiten mit der lokalen Transaktion auftreten, welche eine echte Wechselwirkung zwischen den globalen Sub-Transaktionen bewirkt. Denn die Änderungen von GST_{22} werden dort über die lokale Transaktion indirekt für die Sub-Transaktion GST_{12} sichtbar, so daß keine äquivalente Umkehrung ihrer Ausführungsreihenfolge möglich ist.

Ein einfacher Ansatz zur Gewährleistung von Quasi-Serialisierbarkeit besteht darin, globale Transaktionen stets seriell auszuführen, also jeweils höchstens eine globale Transaktion zuzulassen. Dies ist jedoch aus Leistungsgründen offenbar inakzeptabel, da beim Start globaler Transaktionen mit sehr langen Verzögerungszeiten zu rechnen ist. In [DEK91] wurde ein weniger restriktiver Realisierungsansatz vorgeschlagen, bei dem berücksichtigt wird, auf welche Datenbanken die glo-

balen Transaktionen zugreifen. Insbesondere können ohne Verletzung der Quasi-Serialisierbarkeit solche globalen Transaktionen gleichzeitig zugelassen werden, die auf höchstens einer lokalen Datenbank gemeinsam operieren*. Globale Transaktionen, die auf alle lokalen Datenbanken zugreifen wollen, müssen damit jedoch weiterhin seriell ausgeführt werden (der in Abb. 12-6 gezeigte quasi-serialisierbare Ablauf wird somit nicht zugelassen). Der Ansatz erfordert keine LDBS-Erweiterungen und ermöglicht unterschiedliche lokale Synchronisationsverfahren, solange diese die lokale Serialisierbarkeit gewährleisten. Ferner können keine globalen Deadlocks auftreten. Andererseits sind für globale Transaktionen bis zur Zulassung immer noch erhebliche Blockierungszeiten möglich. Außerdem wird eine zentralisierte globale Transaktionsverwaltung vorausgesetzt, über die sämtliche globalen Transaktionen zu starten und zu beenden sind. Schließlich behandelt der Ansatz lediglich den Isolationsaspekt und nicht die Atomarität und Dauerhaftigkeit globaler Transaktionen (Commit-Behandlung).

Übungsaufgaben

Aufgabe 12-1: Schemaarchitektur eines lose gekoppelten FDBS

Geben Sie die Schemaarchitektur eines lose gekoppelten FDBS an, bei dem nur relationale LDBS zum Einsatz kommen.

Aufgabe 12-2: Semantische Heterogenität

Im Universitätsbeispiel seien z.B. für die Personal-DB und Bibliotheks-DB jeweils ein relationales DBVS im Einsatz mit Personendaten in je einer Relation. Dabei sei (auszugsweise) folgender Relationenaufbau gegeben:

PERSONAL-DB:

Relation PERSONAL	*(PNR*	*INTEGER,*	*{Personalnummer}*
	PNAME	*CHAR (40),*	
	ANSCHRIFT	*CHAR (50), ...)*	

BIBLIOTHEK-DB:

Relation BENUTZER (	*BNR*	*INTEGER,*	*{Benutzernummer}*
	BNAME	*CHAR (50),*	
	WOHNORT	*CHAR (30),*	
	PLZ	*INTEGER,*	
	STRASSE	*CHAR (30), ...)*	

Welche Arten semantischer Heterogenität liegen vor ?

* Eine überlappende Ausführung globaler Transaktionen ist damit auf einen Knoten beschränkt, so daß die lokale Serialisierungsreihenfolge die Quasi-Serialisierungsreihenfolge bestimmt.

Aufgabe 12-3: Schemaintegration

Führen Sie für das Beispiel in der vorherigen Aufgabe eine Schemaintegration durch.

Aufgabe 12-4: Query-Optimierung in FDBS

Inwieweit kollidiert die Optimierung globaler Anfragen mit der Autonomie der LDBS ?

Aufgabe 12-5: Quasi-Serialisierbarkeit

GT_1 und GT_2 seien globale, LT_3 und LT_4 lokale Transaktionen. Ist folgende Ausführungsreihenfolge serialisierbar, quasi-serialisierbar oder keines von beiden (warum) ?

LDBS1: r_3 (a), w_2 (a), w_1 (b), r_3 (b)

LDBS2: w_2 (c), r_4 (c), w_4 (d), r_1 (d).

Teil IV
Shared-Disk-Datenbanksysteme

Teil IV behandelt Mehrrechner-Datenbanksysteme vom Typ "Shared-Disk" oder sogenannte DB-Sharing-Systeme. Nach unserer Klassifikation in Kap. 3 verkörpert Shared-Disk einen allgemeinen Architekturtyp von Parallelen DBS, welche durch eine lokale Rechneranordnung und enge Kooperation der DBVS-Instanzen mehrerer Rechner gekennzeichnet sind (integrierte, homogene Mehrrechner-DBS). Zunächst betrachten wir in Kap. 13 die Architektur von lose und nahe gekoppelten Shared-Disk-Systemen. Daneben werden die wichtigsten Systemfunktionen diskutiert, für die neue Lösungsansätze erforderlich sind. Die beiden darauffolgenden Kapitel befassen sich mit den wichtigsten Realisierungsalternativen für zwei leistungskritische Funktionen in Shared-Disk-Systemen, nämlich der globalen Synchronisation (Kap. 14) sowie der Kohärenzkontrolle (Kap. 15). In diesen Kapiteln wird die Parallelisierung innerhalb von Transaktionen (Intra-Transaktionsparallelität) außer acht gelassen, da diese Gegenstand von Teil V ist und zudem erst in jüngster Vergangenheit von einigen Shared-Disk-Systemen unterstützt wird.

13 Architektur von Shared-Disk-DBS

Shared-Disk (SD)-Systeme streben - wie alle Parallele DBS - vor allem die Erfüllung folgender Anforderungen an (Kap. 1.2): hohe Transaktionsraten, kurze Antwortzeiten, Skalierbarkeit, hohe Verfügbarkeit, Verteilungstransparenz, einfache Administration und hohe Kosteneffektivität. Nicht unterstützt werden dagegen dezentrale Organisationsstrukturen, hohe Autonomie der Rechnerknoten und heterogene Datenbanken. Dies zeigt sich auch an der Architektur von SD-Systemen, die homogen aufgebaut ist und keine geographische Verteilung der Rechner vorsieht.

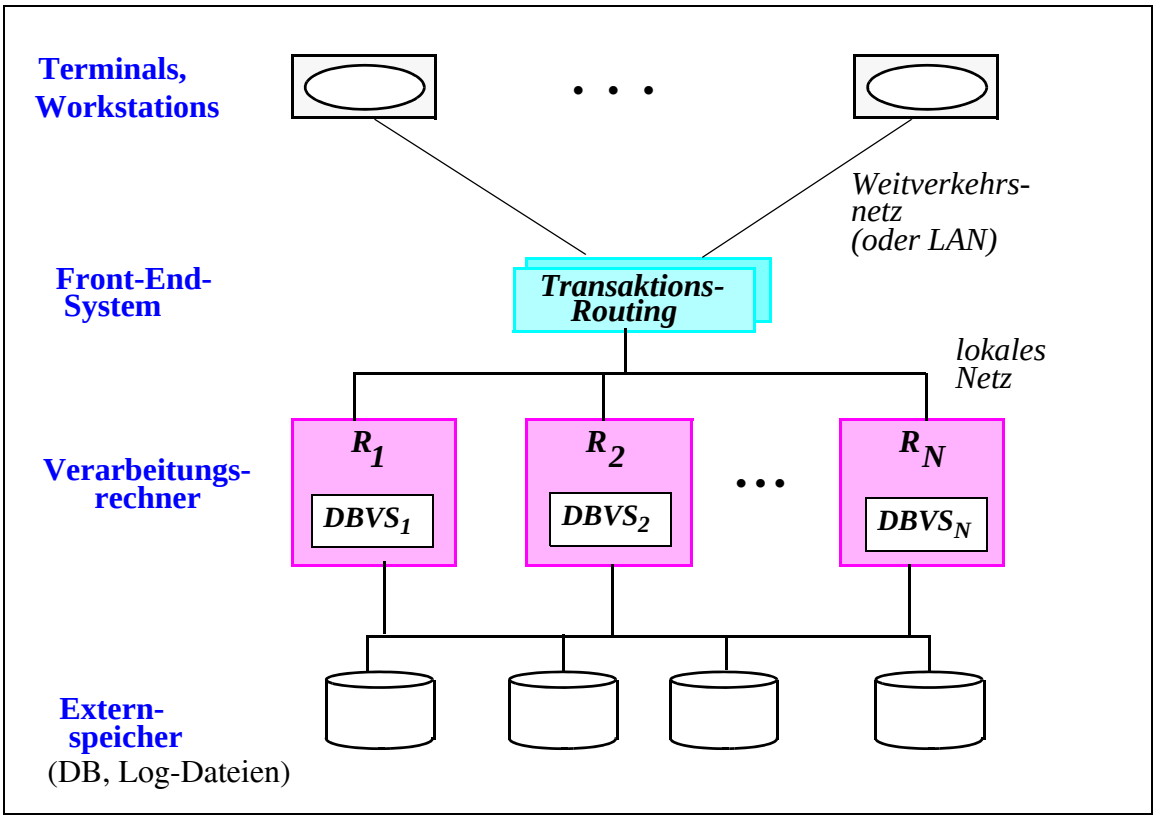
Wir betrachten zunächst den Aufbau lose gekoppelter SD-Systeme, in denen die Kooperation zwischen Rechnern über Nachrichtenaustausch realisiert wird. Von besonderem Interesse hierbei ist die Anbindung der Verarbeitungsrechner an die Externspeicher. Danach ziehen wir einen groben Vergleich zwischen Shared-Disk und Shared-Nothing, wie er sich aus der Architektur dieser Alternativen ergibt. In Kap. 13.3 diskutieren wir die wichtigsten DBS-Funktionen, die für SD neue Realisierungsansätze gegenüber zentralisierten bzw. verteilten DBS erfordern. Abschließend betrachten wir in Kap. 13.4 Alternativen zur Realisierung nah gekoppelter SD-Systeme, insbesondere die Nutzung gemeinsamer Halbleiterspeicher.

13.1 Grobarchitektur lose gekoppelter Shared-Disk-Systeme

Abb. 13-1 zeigt den Grobaufbau eines lose gekoppelten SD-Systems. Es umfaßt N Verarbeitungsrechner, welche mehrere Prozessoren aufweisen können. In jedem Rechner laufen eigene Kopien von Anwendungs- und System-Software, insbesondere eine identische Version des DBVS. Die DBVS-Instanzen arbeiten eng zusammen und bieten den Anwendungen gegenüber vollkommene Verteilungstransparenz. Die Rechner sind lokal innerhalb eines Clusters (i.a. in einem Maschinenraum) angeordnet und über ein Hochgeschwindigkeitsnetzwerk gekoppelt. Wie von der Shared-Disk-Architektur gefordert, ist jeder Rechner und damit jede DBVS-Instanz an alle Externspeicher angebunden. Der gemeinsame Zugriff be-

steht dabei nicht nur auf die physische Datenbank, sondern auch für die Log-Dateien, um nach Rechnerausfällen die Recovery durch überlebende Rechner vornehmen zu können (Kap. 13.3.4).

Abb. 13-1: Prinzipieller Aufbau eines Shared-Disk-Systems



Die Anbindung von Terminals (bzw. Workstation oder PCs) an die Verarbeitungsrechner erfolgt in großen Transaktionssystemen über ein Weitverkehrsnetzwerk, ist jedoch auch über ein lokales Netz (LAN) möglich. Die Terminals sollten nicht statisch je einem Rechner zugeordnet sein, da sich dadurch zwangsweise eine ungünstige Lastbalancierung ergibt. Vielmehr ist eine dynamische Lastverteilung zu unterstützen, bei der die Zuordnung von Transaktionsaufträgen zu Verarbeitungsrechnern (Transaktions-Routing) vom aktuellen Systemzustand abhängig gemacht wird. Wie in Abb. 13-1 angedeutet, kann diese Aufgabe durch Front-End-Rechner gelöst werden.

Wesentlich für eine hohe Leistungsfähigkeit sowie Skalierbarkeit von SD-Systemen ist, daß eine große Anzahl von Rechnern mit allen Platten verknüpft werden kann. In konventionellen E/A-Subsystemen, wie sie etwa in IBM-Großrechner-Konfigurationen vorliegen, bestehen jedoch diesbezüglich relativ enge Grenzen. Diese Architekturen sind durch Platten und Platten-Controller gekennzeichnet, die jeweils nur eine feste Anzahl von Zugriffspfaden erlauben. In der Regel kann dabei eine bestimmte Platte von zwei Controllern aus erreicht werden; ein Platten-

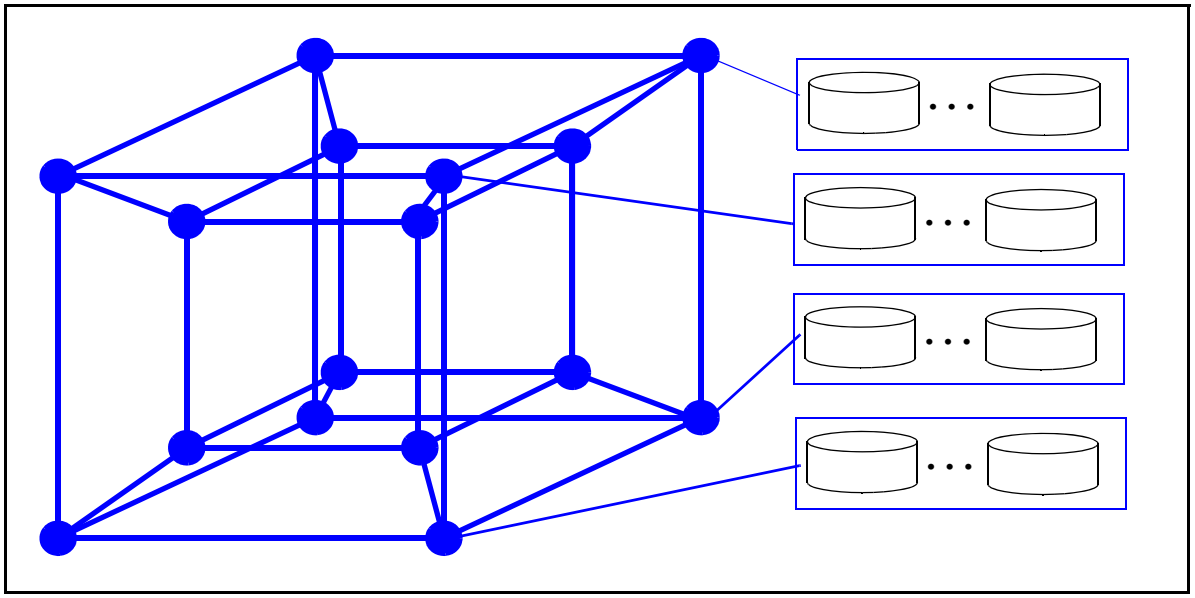
Controller ist meist nur an bis zu 8 bis 16 E/A-Prozessoren anschließbar, die wiederum einem Rechner fest zugeordnet sind. Damit sind solche Shared-Disk-Konfigurationen auf maximal 16 Rechnerknoten beschränkt. Dies stellt für Großrechner, welche selbst wiederum aus mehreren Prozessoren bestehen können, meist keine signifikante Einschränkung dar. Allerdings sind damit hohe Systemkosten und vergleichsweise geringe Kosteneffektivität vorgegeben.

Die Alternative besteht in einem nachrichtenbasierten E/A-Subsystem, bei dem E/A-Prozessoren und Platten-Controller über ein allgemeines Verbindungsnetzwerk gekoppelt sind und über Nachrichtenaustausch kommunizieren. Damit werden auch sämtliche Seitentransfers zwischen Haupt- und Externspeicher im Rahmen von Nachrichten abgewickelt. Ein solcher Ansatz ermöglicht eine prinzipiell unbeschränkte Anzahl von Rechnerknoten, wobei auch preisgünstige Mikroprozessoren verwendet werden können. Allerdings wird ein sehr leistungsfähiges und skalierbares Übertragungsnetzwerk erforderlich, dessen Kapazität mit der Rechneranzahl wächst.

Die VaxCluster-Systeme von DEC [KLS86] verfolgten als erstes eine solche nachrichtenbasierte E/A-Schnittstelle, wobei bis zu 96 Rechner und Platten-Controller miteinander verknüpft werden können. Höhere Rechneranzahlen verlangen i.d.R. mehrstufige Verbindungsnetzwerke, um ein skalierbares Leistungsverhalten zu erreichen. Beispiele hierfür sind Hypercube-Konfigurationen, welche aus 2^N Rechnerknoten bestehen, wobei jeder Knoten nur mit N Nachbarn direkt verbunden ist. Zur Kommunikation sind somit ggf. Zwischenknoten zu involvieren; die maximale Distanz zwischen zwei Knoten ist jedoch auf N begrenzt. Die Anzahl der Verbindungen und damit die Übertragungskapazität steigt proportional mit der Rechneranzahl. Wie in Abb. 13-2 gezeigt, kann die Plattenanbindung dadurch realisiert werden, daß N der Knoten als Platten-Controller fungieren, unter denen alle Platten aufgeteilt werden. Da diese N Knoten von allen restlichen Rechnern erreichbar sind, kann von jedem Knoten aus auf sämtliche Platten zugegriffen werden. Allerdings ergeben sich dabei i.a. langsamere Zugriffszeiten als bei direkter Plattenanbindung ohne Zwischenknoten.

Neben den Magnetplatten können auch weitere Externspeicher von allen Rechnern genutzt werden. Insbesondere sind durch die Controller verwaltete Platten-Caches für alle Knoten zugreifbar, wodurch sich das E/A-Verhalten durch Einsparung von Plattenzugriffen verbessern läßt. Denn jede im Platten-Cache gepufferte Seite vermeidet das Lesen von Platte. Bei nicht-flüchtigen Platten-Caches, die i.a. über eine Reservebatterie realisiert werden, können zudem auch Schreibzugriffe stark optimiert werden [Ra92a]. Ähnliche Performance-Gewinne können mit anderen seitenadressierbaren Halbleiterspeichern wie Solid-State-Disks erzielt werden. Die im Vergleich zu Platten hohen Kosten solcher Speicher dürften sich in Shared-Disk-Systemen eher als in zentralisierten Systemen amortisieren.

Abb. 13-2: 4-dimensionaler Hypercube mit gemeinsamer Plattenanbindung



13.2 Shared-Disk vs. Shared-Nothing

Während in SD-Systemen jede DBVS-Instanz Zugriff auf die gesamte physische DB besitzt, ist in Shared-Nothing (SN)-Systemen eine Partitionierung der Datenbank unter mehreren Rechner erforderlich. Aus dieser grundlegenden Unterscheidung resultieren eine Reihe gewichtiger Vorteile für SD:

- Die Systemverwaltung wird erheblich erleichtert, da die Bestimmung der Datenverteilung in SN-Systemen eine schwierige und aufwendige Aufgabe darstellt. Besonders problematisch ist die Festlegung einer DB-Verteilung für nicht-relationale DBS (z.B. objekt-orientierte DBS oder hierarchische bzw. Netzwerk-DBS) aufgrund der starken Vernetzung zwischen Datenobjekten. Für solche DBS erscheint der SD-Ansatz wesentlich attraktiver als SN.
- Rechnerausfälle bleiben für SD ohne Auswirkungen auf die Externspeicherzuordnung, da die überlebenden Rechner weiterhin sämtliche Daten erreichen können. In SN-Systemen drohen dagegen Verfügbarkeitseinbußen, wenn die Daten eines ausgefallenen Rechners nicht mehr erreichbar sind. Ein einfacher Ansatz bei lokaler Verteilung sieht vor, daß ein überlebender Rechner die Partition des ausgefallenen Knotens vollständig übernimmt. Der Nachteil dabei ist jedoch, daß dieser dann mit hoher Wahrscheinlichkeit zum Engpaß wird, da er die Zugriffe auf zwei Partitionen bearbeiten muß. Eine aufwendige Alternative besteht in der Replikation von Daten an mehreren Knoten (Kap. 9, Kap. 17.4).
- Eine Erhöhung der Rechneranzahl erfordert bei SN die Neubestimmung der Datenbankverteilung, um die zusätzlichen Rechner nutzen zu können. Zudem ist das physische Umverteilen für große Datenbanken ein sehr aufwendiger Vorgang, der auch die Verfügbarkeit der betroffenen Daten beeinträchtigt. Diese Probleme entfallen bei SD, so daß sich Vorteile bezüglich der Erweiterbarkeit des Systems ergeben. Insbesondere sind keine Änderungen für den physischen DB-Aufbau beim Übergang vom Ein-Rechner- auf den Mehr-Rechner-Fall erforderlich, was wiederum eine starke Vereinfachung für die Administration bedeutet.

- In SN-Systemen bestimmt die DB-Verteilung für viele Operationen, wo diese auszuführen sind, unabhängig davon, ob diese Rechner bereits überlastet sind oder nicht. Damit bestehen kaum Möglichkeiten, die Auslastung der Rechner dynamisch zu beeinflussen. Vielmehr ist mit stark schwankender Rechnerauslastung zu rechnen, da zu einem bestimmten Zeitpunkt die einzelnen DB-Partitionen i.a. ungleichmäßig referenziert werden. Bei SD dagegen kann eine DB-Operation von jedem Rechner (DBVS) bearbeitet werden, da er Zugriff auf die gesamte Datenbank besitzt. Dies ergibt ein weit höheres Potential zur dynamischen Lastverteilung und -balancierung. Damit können auch bei schwankendem Lastprofil ungleiche Rechnerauslastung sowie die damit verbundenen Leistungseinbußen umgangen werden [YD94].
- Die fehlende, statische DB-Aufteilung ergibt für SD auch erhöhte Freiheitsgrade zur Parallelisierung innerhalb von Transaktionen. Darauf wird in Teil V näher eingegangen.

Für die SN-Architektur spricht, daß es für sie hardwareseitig einfacher ist, eine große Anzahl von Prozessoren zu unterstützen, da jede Platte nur mit einem Rechner zu verbinden ist. Ferner ist ein weniger leistungsstarkes Verbindungsnetzwerk ausreichend, da es lediglich zur Kommunikation zwischen den Rechnern, jedoch nicht für Externspeicherzugriffe (Seitentransfers) verwendet wird.

SD und SN unterscheiden sich natürlich auch hinsichtlich der technischen Probleme, die jeweils zu lösen sind (Kap. 3.1.2). Während die SN-Probleme in anderen Teilen des Buchs behandelt werden, diskutieren wir im nächsten Teilkapitel die SD-spezifischen Probleme, wobei auch auf Unterschiede zu SN eingegangen wird.

13.3 Neue Realisierungsanforderungen

Die Diskussion zeigt, daß die Shared-Disk-Architektur zahlreiche Vorteile hinsichtlich der von Parallelen DBS zu erfüllenden Anforderungen aufweist. Die Nutzung dieser Freiheitsgrade setzt jedoch geeignete Lösungen für SD-spezifische Probleme voraus. Dies betrifft vor allem die Funktionen Synchronisation, Kohärenzkontrolle, Lastverteilung sowie Logging/Recovery, die im folgenden diskutiert werden. Wie wir sehen werden, beeinflussen sich die einzelnen Systemkomponenten auch wesentlich stärker als in zentralisierten DBS oder SN-Systemen. Diese Abhängigkeiten sind somit bei der Entwicklung geeigneter Realisierungsansätze von vorneherein zu berücksichtigen.

13.3.1 Globale Synchronisation

Die Synchronisation der Rechner beim Zugriff auf die gemeinsame Datenbank ist bei SD eine offenkundige Notwendigkeit, um die globale Serialisierbarkeit (Kap. 8) der bearbeiteten Transaktionen sicherstellen zu können. Da wegen der losen Rechnerkopplung die Synchronisation nur durch Nachrichtenaustausch zwischen den Rechnern möglich ist, ergibt sich ein weitaus höherer Synchronisationsaufwand als in zentralisierten Systemen, wo alle benötigten Datenstrukturen im

Hauptspeicher erreichbar sind. In Verteilten DBS bzw. Shared-Nothing-Systemen kann die Synchronisation bei verteilten Protokollen ohne zusätzliche Kommunikation erfolgen, wenn jeder Rechner die Zugriffe auf seine Datenpartition lokal synchronisiert (Kap. 8.1.2). Höchstens zur Erkennung globaler Deadlocks fallen dabei eigene Nachrichten an.

Selbst bei einem sehr schnellen Kommunikationssystem ist z.B. das Anfordern einer Sperre bei einem anderen Rechner um Größenordnungen langsamer als eine lokale Sperrbearbeitung. Denn allein das Senden und Empfangen der dazu notwendigen Nachrichten verursacht eine weitaus höhere CPU-Belastung, da diese Operationen meist Tausende von Instruktionen erfordern. Dazu kommen i.a. noch Prozeßwechselkosten, da die Verzögerungen für die Transaktion zu lang sind, um während der Wartezeit auf eine Antwort im Besitz des Prozessors zu bleiben. Für die Leistungsfähigkeit besonders kritisch sind *synchrone Nachrichten*, für die eine Transaktion bis zum Eintreffen einer Antwortnachricht unterbrochen werden muß. Denn diese verursachen nicht nur Durchsatzeinbußen (Kommunikations-Overhead), sondern erhöhen direkt die Antwortzeiten. Verlängerte Antwortzeiten wiederum bewirken eine Zunahme an Synchronisationskonflikten, da z.B. Sperren länger gehalten werden, was zur Erhöhung der Konfliktwahrscheinlichkeit sowie der mittleren Wartezeit auf gehaltene Sperren führt. Ein Hauptziel effizienter Synchronisationsprotokolle für Shared-Disk ist daher, das Ausmaß synchroner Nachrichten zu minimieren. Wie wir sehen werden, ist hierzu auch eine enge Abstimmung mit der Kohärenzkontrolle sowie der Lastverteilung erforderlich.

Ein leistungsfähiges Synchronisationsprotokoll muß generell mit möglichst wenig Blockierungen und Rücksetzungen von Transaktionen auskommen. Dies kann durch eine Vielzahl von Maßnahmen unterstützt werden, wie Verwendung feiner Synchronisationsgranulate (z.B. Satzsperren), Sonderbehandlung für Verwaltungsdaten oder andere Hot-Spot-Objekte, Mehrversionen-Verfahren oder reduzierte Konsistenzzusicherungen [Ra88b]. Diese Ansätze sind jedoch für Shared-Disk aufgrund von Wechselwirkungen mit der Kohärenzkontrolle zum Teil erheblich schwieriger zu realisieren als in zentralisierten DBS oder Shared-Nothing-Systemen. So kann es bei Verwendung von Satzsperren vorkommen, daß eine Seite parallel in verschiedenen Rechnern geändert wird. Dies führt jedoch dazu, daß keiner der Rechner nach seiner Änderung die aktuelle Version der Seite besitzt und die Gefahr besteht, daß die Änderungsstände von Kopien einer Seite immer weiter auseinanderlaufen.

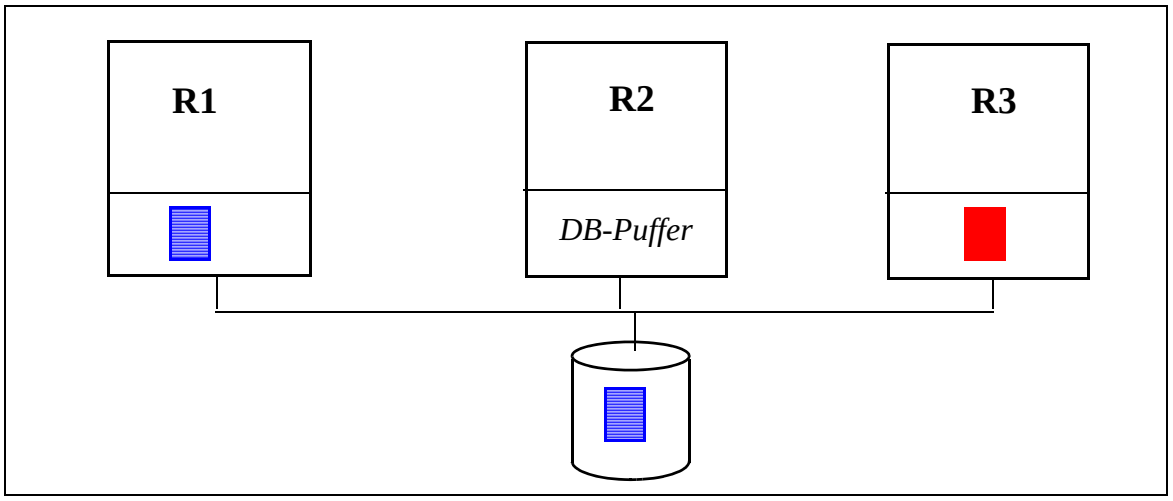
Eine weitere Anforderung an ein geeignetes Synchronisationsprotokoll für Shared-Disk ist Robustheit gegenüber Fehlern im System. Insbesondere muß nach Rechnerausfällen eine korrekte Fortsetzung der Synchronisation gewährleistet sein, um eine hohe Verfügbarkeit zu erreichen.

In Kap. 14 werden wir ausführlich auf die Realisierung von Synchronisationsverfahren für Shared-Disk eingehen.

13.3.2 Kohärenzkontrolle

Jede DBVS-Instanz eines Shared-Disk-Systems führt einen DB-Puffer im lokalen Hauptspeicher, um die Anzahl von Plattenzugriffen gering zu halten. Da jeder Rechner auf die gesamte DB zugreifen kann, ist es möglich, daß Kopien derselben DB-Seiten gleichzeitig in mehreren Knoten gepuffert werden. Dies führt jedoch zu einem Invalidierungsproblem, da bei einer Seitenänderung im DB-Puffer eines Rechners Kopien der Seite in anderen Knoten einen veralteten und somit ungültigen Zustand aufweisen (Pufferinvalidierung). Im Beispiel von Abb. 13-3 verursacht die Änderung einer Seite in Rechner R3 eine Pufferinvalidierung in R1. Die ursprüngliche Seitenversion auf Externspeicher ist ebenso veraltet. Die Behandlung solcher invalidierten Seiten ist die Aufgabe der Kohärenzkontrolle, wobei sicherzustellen ist, daß jeder Transaktion die aktuellen DB-Objekte zur Verfügung gestellt werden. Dies ist wiederum mit möglichst wenig Kommunikationsaufwand und geringer Antwortzeiterhöhung zu erledigen.

Abb. 13-3: Seiteninvalidierungen in Shared-Disk-Systemen



Ähnliche Kohärenzprobleme bestehen bei Multiprozessoren auf Ebene der Hardware-Caches [St90] sowie in anderen verteilten Systemen, insbesondere in Workstation/Server-Systemen und DSM (Distributed Shared Memory)-Systemen [Ra93b]. In Shared-Nothing-Systemen stellt sich die Kohärenzproblematik nicht, da jedes DBVS i.a. nur Seiten der lokalen DB-Partition puffert, so daß es zu keiner replizierten Pufferung von Seiten kommt. Ähnlichkeiten bestehen jedoch zu replizierten Datenbanken (Kap. 9), wo auch an mehreren Knoten replizierte DB-Objekte zu warten sind. Diese Replikation ist jedoch statisch und vorgeplant sowie auf Ebene der Externspeicher, während für Shared-Disk eine dynamische Replikation auf Hauptspeicherebene vorliegt.

In SD-Systemen erfordert die Kohärenzkontrolle, invalidierte Seitenkopien in den DB-Puffern zu erkennen bzw. von vorneherein zu vermeiden. Weiterhin müssen

die Änderungen im ganzen System propagiert werden, um an jedem Rechner die neuesten Objektversionen verfügbar zu machen. Der Austausch geänderter Seiten kann über die gemeinsamen Platten erfolgen, indem zunächst ein Ausschreiben in die physische DB erfolgt, von wo dann andere Rechner die aktuelle Version einer Seite lesen. Wesentlich schneller ist allerdings der direkte Austausch geänderter Seiten über das Kommunikationsnetz, wenngleich dabei die physische DB auf dem veralteten Stand bleibt.

Die replizierte Speicherung von DB-Seiten in mehreren Puffern unterstützt das Lastbalancierungspotential von SD, da somit dieselben Objekte in verschiedenen Knoten parallel bearbeitet werden können. Andererseits ergibt sich damit systemweit betrachtet i.d.R. eine Verringerung der Trefferraten gegenüber zentralisierten DBS bzw. Shared-Nothing-Systemen. Denn dort wird eine Seite nur einmal gepuffert, so daß bei gleicher Gesamt-Hauptspeichergröße insgesamt mehr unterschiedliche Seiten gepuffert werden können. Pufferinvalidierungen führen zu einer weiteren Verschlechterung des E/A-Verhaltens, da veraltete Kopien nicht wiederverwendet werden können. Die Anzahl von Pufferinvalidierungen steigt sowohl mit der Änderungsrate, der Puffergröße sowie der Rechneranzahl (da N-1 von N Rechnern Invalidierungen verursachen können). Letztere Abhängigkeit kann für änderungsintensive Lasten zu einer beeinträchtigten Skalierbarkeit von Shared-Disk-Systemen führen. Dies gilt jedoch primär für einfache Lastverteilungsansätze, die das Referenzverhalten nicht berücksichtigen, so daß sich die Zugriffe auf bestimmte DB-Bereiche gleichermaßen über alle N Rechnern verteilen (s.u.).

Die wichtigsten Ansätze zur Kohärenzkontrolle in Shared-Disk-Systemen werden in Kap. 15 behandelt.

13.3.3 Lastverteilung

Ziel der Lastverteilung ist die Allokation von Transaktionsaufträgen zu Verarbeitungsrechnern (Transaktions-Routing), so daß eine effiziente Transaktionsbearbeitung erreicht wird. Diese Aufgabe sollte unter Berücksichtigung des aktuellen Systemzustandes durchgeführt werden, insbesondere der Verfügbarkeit und Auslastung der einzelnen Verarbeitungsrechner*. Denn nur so kann eine effektive Lastbalancierung zur Reduzierung von Überlastsituationen erreicht werden.

---

* Die Freiheitsgrade der Lastverteilung sind natürlich eingeschränkt, wenn bestimmte Transaktionsprogramme nur an einer Teilmenge der Rechner vorliegen. Wir unterstellen daher die Verfügbarkeit jedes Transaktionsprogramms an allen Rechnern. Für Shared-Disk-Systeme ist dies ohne replizierte Speicherung der Programme möglich, indem diese auf den gemeinsamen Platten gehalten werden. Einzelne DB-Operationen können ohnehin von jeder DBVS-Instanz bearbeitet werden.

Neben der Lastbalancierung sollte die Lastverteilung jedoch auch eine Transaktionsverarbeitung mit einem Minimum an Kommunikations-, E/A- und Synchronisationsverzögerungen unterstützen. Ein allgemeiner Ansatz hierzu ist ein sogenanntes *affinitätsbasiertes Transaktions-Routing* [YCDI87, Ra92b]. Dabei wird eine Lastverteilung angestrebt, so daß Transaktionen verschiedener Rechner möglichst unterschiedliche DB-Bereiche referenzieren. Hierzu gilt es, Transaktionstypen mit einer Affinität zu denselben DB-Bereichen möglichst denselben Verarbeitungsrechnern zuzuweisen, wodurch eine hohe *rechnerspezifische Lokalität* im Referenzverhalten erreicht wird. Das für solche Entscheidungen benötigte Wissen über das Referenzverhalten ist zumindest für häufig ausgeführte Online-Transaktionen bekannt bzw. kann über Monitoring ermittelt werden.

Lastverteilungverfahren wie die wahlfreie Auswahl der Verarbeitungsrechner (random routing), welche das Referenzverhalten nicht berücksichtigen, sind wesentlich einfacher zu realisieren als ein affinitätsbasierter Ansatz. Dafür bringt die Unterstützung rechnerspezifischer Lokalität gewichtige Vorteile für das Leistungsverhalten eines Shared-Disk-Systems, die im Rahmen mehrerer Leistungsanalysen nachgewiesen wurden [YCDI87, Ra93d]:

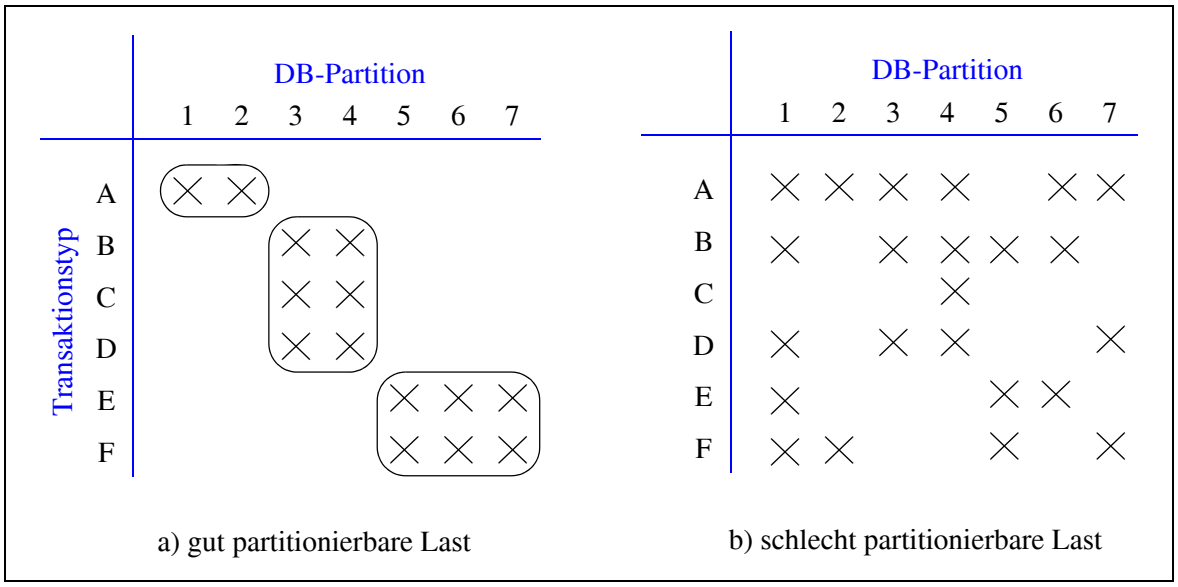
- Eine hohe Lokalität kann natürlich von der Pufferverwaltung jedes Rechners zur Reduzierung physischer E/A-Vorgänge genutzt werden. Für Shared-Disk bewirkt die rechnerspezifische Lokalität darüber hinaus, daß relativ wenige Seiten repliziert in mehreren Rechnern gepuffert sind, so daß sich die Trefferraten gegenüber wahlfreier Lastverteilung verbessern.
- Das Ausmaß an Pufferinvalidierungen kann stark eingeschränkt werden, wenn Änderungen bestimmter Seiten nur an einem bzw. wenigen Rechnern erfolgen. Gegenüber wahlfreier Lastverteilung verbessert dies die Trefferraten und reduziert die Notwendigkeit, geänderte Seiten im System zu propagieren. Damit wird auch die Skalierbarkeit auf eine große Anzahl von Rechnern nachhaltig verbessert.
- Einige Synchronisationsverfahren für Shared-Disk sind in der Lage, rechnerspezifische Lokalität zur Reduzierung des Kommunikationsaufwandes zu nutzen (Kap. 14). Damit ergibt sich auch für diese kritische Funktion ein besseres Leistungsverhalten als bei wahlfreier Lastverteilung.
- Eine weitere Konsequenz rechnerspezifischer Lokalität ist, daß Synchronisationskonflikte zumeist zwischen Transaktionen desselben Rechners auftreten. Diese können jedoch wesentlich schneller erkannt und behandelt werden (lokale Blockierung/Rücksetzung von Transaktionen, lokale Deadlock-Erkennung, Warten auf Sperrfreigabe einer lokalen Transaktionen u.ä.) als rechnerübergreifende Konflikte, die zusätzliche Kommunikationsverzögerungen bewirken.

Inwieweit Lastbalancierung und hohe rechnerspezifische Lokalität erreicht werden können, hängt stark von den Referenzmerkmalen der zu bearbeitenden Transaktionslast ab. Im Idealfall, der in Abb. 13-4a skizziert ist, kann die Last für N Rechner in N Lastgruppen (Transaktionstypen) partitioniert werden, die (weitgehend) disjunkte DB-Bereiche berühren. Durch Zuweisung jeder Lastgruppe zu einem anderen Verarbeitungsrechner wird dann ein Optimum an rechnerspezifi-

scher Lokalität erzielt. Eine günstige Lastbalancierung würde darüber hinaus erfordern, daß jede Lastgruppe etwa den gleichen Bearbeitungsaufwand erfordert.

Vielfach liegen jedoch weniger günstige Verhältnisse vor, wie sie beispielhaft in Abb. 13-4b gezeigt sind. In solchen Fällen existieren Transaktionstypen, die nahezu alle DB-Bereiche referenzieren, sowie DB-Bereiche, die von fast allen (wichtigen) Transaktionstypen referenziert werden. Eine affinitätsbasierte Lastverteilung ist hier zwar auch möglich, jedoch läßt sich nicht verhindern, daß verschiedene Rechner dieselben DB-Bereiche referenzieren. Die Situation verschärft sich noch für dominierende Transaktionstypen, deren Verarbeitung auf nur einem Rechner aufgrund hoher Ausführungsrate bzw. hohen Ressourcen-Bedarfs dessen Überlastung zur Folge hätte. Die notwendige Aufteilung solcher Transaktionstypen auf mehrere Rechner führt jedoch nahezu zwangsweise zu Lokalitätseinbußen. Dies kann höchstens (etwa unter Berücksichtigung von Eingabeparametern) durch eine Aufteilung in mehrere Sub-Transaktionstypen, die unterschiedliche DB-Teile berühren, vermieden werden. In Bankanwendungen könnten so Konto-Transaktionen (Einzahlung, Abhebung etc.) über die Kontonummer aufgeteilt werden, so daß jeder Rechner Transaktionen/Konten eines anderen Wertebereiches für die Kontonummern bearbeitet.

Abb. 13-4: Referenzverteilung von Transaktionslasten



Zur Implementierung einer Routing-Strategie bestehen vielzählige Alternativen, die in [Ra92b] klassifiziert und beschrieben werden. Aus Effizienzgründen empfiehlt sich für Online-Transaktionen ein tabellengesteuerter Ansatz, bei dem für einen Transaktionstyp aus einer Routing-Tabelle der bevorzugte Verarbeitungsrechner ermittelt wird. Bei der Bestimmung der Routing-Tabelle sind Referenzierungsmerkmale sowie geschätzter Betriebsmittelbedarf von Transaktionstypen zu berücksichtigen. Eine Neubestimmung der Routing-Tabelle kann in bestimmten

Situationen vorgenommen werden, insbesondere bei veränderter Rechneranzahl (Rechnerausfall, neuer Rechner) oder stark geändertem Lastprofil.

Allerdings zeigten empirische Untersuchungen bereits, daß mit einem tabellengesteuerten Ansatz allein meist keine effektive Lastbalancierung erreicht werden kann [Ra88b]. Denn die Berechnung der Routing-Tabelle basiert auf Erwartungswerten über mittlere Ankunftsraten und Instruktionsbedarf pro Transaktionstyp, während die tatsächliche Lastsituation von den Mittelwerten meist signifikant abweicht. Zudem sind die durch Kommunikations-, E/A-, und Sperrverzögerungen bedingten Auswirkungen i.a. in jedem Rechner verschieden und bei der Berechnung der Routing-Tabelle kaum vorhersehbar. Aus diesen Gründen empfiehlt sich ein gemischter Ansatz zum Transaktions-Routing, bei dem aus einer vorbestimmten Tabelle der unter Lokalitätsaspekten bevorzugte Rechner bestimmt wird, daneben jedoch auch die aktuelle Lastsituation berücksichtigt wird. Ist z.B. der bevorzugte Rechner zum Zuweisungszeitpunkt bereits völlig überlastet, kann eine Zuteilung zu einem weniger belasteten Rechner erfolgen, der auch teilweise Lokalität unterstützt.

13.3.4 Logging und Recovery

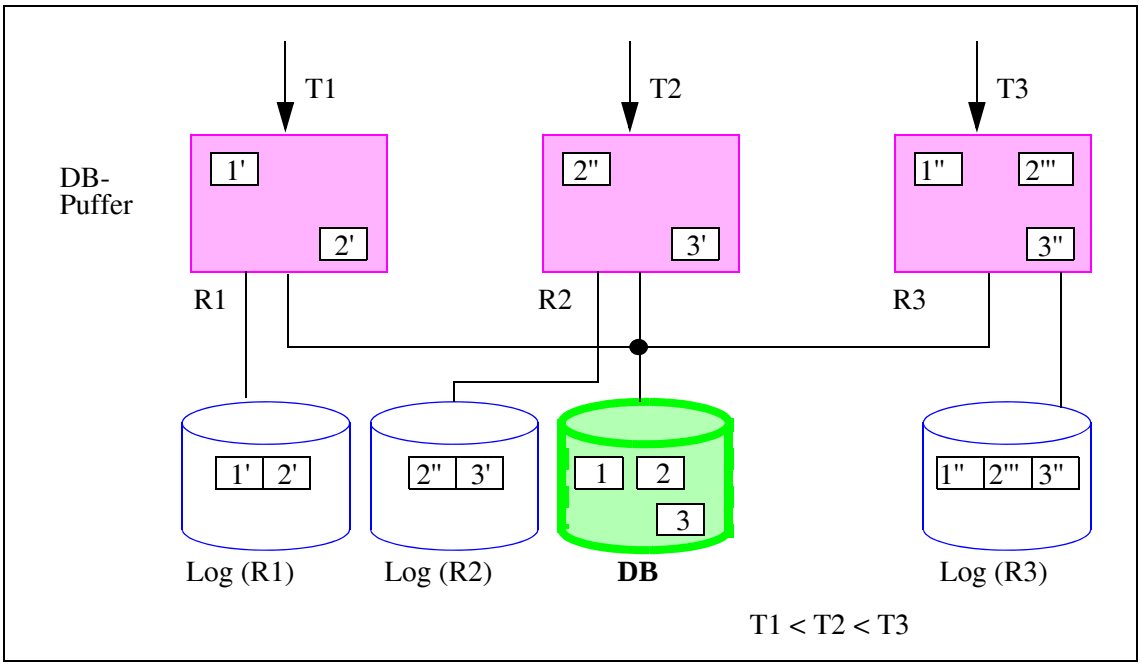
Jeder Verarbeitungsrechner protokolliert die DB-Änderungen der bei ihm ausgeführten Transaktionen in eine *lokale Log-Datei.* Mit der lokalen Log-Datei können dann sowohl Rücksetzungen einzelner Transaktionen als auch die Recovery nach Ausfall eines Rechners vorgenommen werden. Das Führen der lokalen Log-Datei kann mit den aus zentralisierten DBS bekannten Techniken erfolgen [GR93]. Die Behandlung von Plattenfehlern und ggf. auch von Rechnerausfällen verlangt jedoch zusätzlich die Erstellung einer *globalen Log-Datei*, in der sämtliche Änderungen im System in chronologischer Reihenfolge protokolliert sind. Dies ist erforderlich, da eine bestimmte Seite i.a. an mehreren Rechnern geändert wird. Zur Rekonstruktion des neuesten Seitenzustands im Fehlerfall sind daher die an verschiedenen Rechnern vorgenommenen Änderungen in der ursprünglichen Reihenfolge zu wiederholen.

Beispiel 13-1

Im Szenario von Abb. 13-5 werden die drei DB-Seiten 1, 2 und 3 von drei Transaktionen verschiedener Rechner geändert, wobei die Serialisierungsreihenfolge T1 < T2 < T3 sein soll. So ändert zunächst T1 die Seiten 1 und 2 (neue Versionen 1' und 2') in Rechner R1, danach T2 die Seiten 2 und 3 in R2; abschließend ändert T3 in R3 alle drei Seiten. Die Änderungen werden jeweils in den lokalen Log-Dateien der Rechner protokolliert. Die physische DB enthält noch die alten Seitenversionen, da unterstellt wurde, daß geänderte Seiten direkt zwischen den Rechnern ausgetauscht werden. Nach einem Ausfall der gezeigten DB-Platte sind somit alle Änderungen seit dem letzten Einlesen zu wiederholen. Dies verlangt die Anwendung der Log-Daten aller lokalen Log-Dateien in der ursprünglichen Reihenfolge, wie es mit einer globalen Log-Datei unterstützt wird.

Selbst die Crash-Recovery erfordert aufgrund des direkten Austauschs geänderter Seiten die Anwendung der globalen Log-Datei. Fällt etwa R3 aus, so sind die dort ausgeführten Änderungen erfolgreicher Transaktionen zu wiederholen. Dazu genügt es jedoch nicht, lediglich die lokalen Log-Daten anzuwenden, da die Änderungen in R3 auf zuvor ausgeführten Änderungen der anderen Rechner basieren, die jedoch noch nicht in der physischen Datenbank reflektiert sind. Daher müssen diese zuvor auch wiederholt werden, wie mit einem globalen Log möglich.

Abb. 13-5: Beispiel-Szenario zur Recovery in SD-Systemen



Die direkte Erstellung einer globalen Log-Datei auf Plattenspeicher ist i.a. zu aufwendig, da zusätzliche Schreibvorgänge am Transaktionsende notwendig würden und zudem das aktuelle Log-Ende aufgrund der hohen Schreibfrequenz leicht zum Engpaß würde. Daher ist es vorzuziehen, die globale Log-Datei asynchron zur Transaktionsverarbeitung durch Mischen der lokalen Log-Dateien zu erstellen. Die Mehrzahl existierender SD-Implementierungen sieht hierfür einen Offline-Ansatz vor, bei dem der Systemverwalter durch Starten eines Dienstprogrammes die Erstellung bzw. Vervollständigung der globalen Log-Datei veranlaßt. Von Nachteil dabei ist jedoch, daß im Fehlerfall manuelle Eingriffe notwendig werden und lange Zeit vergehen kann, bis die benötigten globalen Log-Daten vorliegen. Eine bessere Verfügbarkeit wird mit einer Online-Erstellung der globalen Log-Datei erreicht, bei der im laufenden Betrieb bereits das Mischen der Log-Daten erfolgt. Ein solcher Ansatz ist jedoch schwer zu realisieren und beeinträchtigt das Leistungsverhalten im Normalbetrieb (zusätzliche Kommunikations- und E/A-Vorgänge, großer Log-Umfang).

Um ein korrektes Mischen zu ermöglichen, sind die Log-Sätze der lokalen Log-Dateien mit geeigneten Zeitstempeln oder Änderungszählern zu kennzeichnen. Ein

in IBM-Systemen verfolgter Ansatz ist die Nutzung einer speziellen Hardware-Uhr (Sysplex Timer), welche von allen Rechnern zugreifbar ist und systemweit eindeutige Zeitstempel liefert, die in die Log-Sätze aufgenommen werden. Damit brauchen die Log-Sätze nur noch nach diesen Zeitstempel sortiert zu werden. Von Nachteil sind dabei jedoch die relativ hohen Kosten für die Uhr. Ein alternativer Ansatz besteht im Führen seitenspezifischer Änderungszähler, die bei jeder Änderung inkrementiert werden. Damit lassen sich ohne Spezial-Hardware die Änderungen für jede Seite auch systemweit ordnen, allerdings nicht seitenübergreifend [Lo90]. Eine weitere Alternative besteht darin, an jedem Rechner einen lokalen Änderungszähler zu führen, der mit den Zählern anderer Rechner synchronisiert wird [MN92b, DYJ94]. Ein Ansatz dazu ist, bei jedem Seitentransfer von Rechner A nach B den aktuellen Zählerstand von A mitzuübergeben. Liegt er über dem Zählerstand von B, wird dieser auf den höheren Wert von Rechner A gebracht. Dadurch ist sichergestellt, daß alle Seitenänderungen in B einen höheren Zeitstempel erhalten als die zuvor in A vorgenommenen Änderungen. Ein solcher Ansatz entspricht der von Lamport vorgeschlagenen Methode zur Uhrensynchronisation in verteilten Systemen [La78].

Die Recovery nach Ausfall eines Rechners muß, um eine möglichst störungsfreie Fortsetzung der Verarbeitung zu erlauben (Freigabe gehaltener Sperren u.ä.), von einem oder mehreren der überlebenden Rechner mit der lokalen Log-Datei des ausgefallenen durchgeführt werden. Dabei sind alle durch den Rechnerausfall verlorengegangenen DB-Änderungen erfolgreich beendeter Transaktionen zu rekonstruieren. Diese Redo-Recovery muß - wie das Beispiel gezeigt hat - ggf. mit einer globalen Log-Datei erfolgen. Transaktionen, die durch den Rechnerausfall unterbrochen wurden, sind mit der lokalen Log-Datei zurückzusetzen (Undo-Recovery), wobei ggf. gehaltene Sperren freizugeben sind. Während der Crash-Recovery müssen außerdem ggf. verlorengegangene Datenstrukturen rekonstruiert werden, um eine korrekte Fortsetzung von Synchronisation und Kohärenzkontrolle zu gewährleisten. Die Einzelheiten dafür hängen natürlich stark von den jeweils verwendeten Protokollen ab [MN91, Ra91a].

Die Katastrophen-Recovery für Shared-Disk-Systeme kann im Prinzip wie in Kap. 9.5 dargestellt mit einem geographisch entfernten Backup-System erfolgen. Dabei empfiehlt sich jedoch die Übertragung der globalen Log-Daten, um die DB-Kopie im Backup-System nachführen zu können. Dies setzt eine Online-Erzeugung der globalen Log-Datei voraus.

13.4 Nah gekoppelte Shared-Disk-Systeme

Ziel einer nahen Rechnerkopplung ist, den hohen Kommunikationsaufwand einer losen Kopplung zu reduzieren, indem bestimmte Funktionen über globale Halbleiterspeicher bzw. Spezialprozessoren realisiert werden (Kap. 3.1). Dabei soll der Zugriff auf die gemeinsam benutzten Systemkomponenten sehr schnell sein, so daß ein synchroner Zugriff möglich wird. Dabei bleibt die CPU während des Zugriffs zur Umgehung von Prozeßwechselkosten belegt. Dies erfordert in der Regel spezielle Maschinenbefehle für den Zugriff.

Wir diskutieren zunächst, welche generellen Nutzungsmöglichkeiten für eine nahe Kopplung in Shared-Disk-Systemen bestehen. Anschließend betrachten wir dazu verschiedene Realisierungsalternativen, insbesondere die Verwendung von instruktions- und seitenadressierbaren Halbleiterspeichern sowie von Spezialprozessoren (Kap. 13.4.2). Abschließend wird der Einsatz eines sogenannten Globalen Erweiterten Hauptspeichers (GEH) erörtert. Die 1994 eingeführten Shared-Disk-Architekturen IBM Parallel Sysplex und Parallel Transaction Server verwenden eine nahe Kopplung, auf die in Kap. 19.1.2 eingegangen wird.

13.4.1 Einsatzformen der nahen Kopplung

Die allgemeinste Einsatzform eines gemeinsamen Speichers liegt darin, die *Kommunikation* über ihn abzuwickeln. In diesem Fall, werden die "Nachrichten" vom Sender in den gemeinsamen Speicher geschrieben; das "Empfangen" der Nachricht erfolgt durch Lesen von diesem Speicher. Der Lesezugriff erfolgt entweder periodisch oder aufgrund einer speziellen Signalisierung seitens des ersten Rechners. Eine solche speicherbasierte Kommunikation kann bei entsprechend schnellen Zugriffszeiten (z.B. wenigen Mikrosekunden) vor allem bei sehr großen Datentransfers wesentlich effizienter als ein allgemeines Kommunikationsprotokoll sein. Die Realisierung sollte durch das Betriebssystem erfolgen, so daß der Ansatz nicht nur der DB-Verarbeitung zugute kommt. Weiterhin bleibt dann dem DBS die nahe Kopplung vollkommen verborgen; d.h. nachrichtenbasierte Protokolle für Synchronisation, Kohärenzkontrolle etc. können weiterhin eingesetzt werden.

Eine andere allgemeine Einsatzmöglichkeit ist die Nutzung seitenadressierbarer Halbleiterspeicher zur Verbesserung des E/A-Verhaltens. Besonders interessant sind hierfür nicht-flüchtige Halbleiterspeicher, da mit ihnen sowohl Lese- als auch Schreibzugriffe auf die Platten eingespart werden können. Insbesondere können solche Halbleiterspeicher zur permanenten *Allokation ganzer Dateien* verwendet werden, um für sie sämtliche Plattenzugriffe zu umgehen. Aufgrund der hohen Speicherkosten ist dies nur für leistungskritische Dateien kosteneffektiv, z.B. Log-Dateien oder häufig geänderte DB-Dateien. Die Alternative besteht in der Pufferung von DB-Seiten im Rahmen eines globalen Dateipuffers bzw. DB-Puffers. Von einem *globalen DB-Puffer* sprechen wir, wenn die Verwaltung in Abstimmung mit

dem DBS erfolgt; ein Dateipuffer liegt vor bei Verwaltung durch das Betriebssystem bzw. den Speicher-Controller. Für Seiten, die in einem solchen Puffer vorliegen, kann der Lesezugriff auf Platte eingespart werden. Ferner können bei Nicht-Flüchtigkeit Schreibzugriffe zunächst nur in den Puffer erfolgen, von wo aus die Seitenkopien auf Platte asynchron aktualisiert werden. In Shared-Disk-Systemen kann der globale Puffer auch zum schnellen Austausch geänderter Seiten genutzt werden. Die Nutzung eines Dateipuffers ist aufgrund der fehlenden Abstimmung mit der Hauptspeicherpufferung des DBS i.a. weniger effektiv als ein globaler DB-Puffer [Ra93a].

Die nahe Kopplung kann für eine Reihe weiterer, Shared-Disk-spezifischer Nutzungsformen eingesetzt werden, wobei eine Unterstützung durch die DBVS-Realisierung erforderlich ist. So können in einem gemeinsamen Halbleiterspeicher globale Datenstrukturen für folgende Aufgaben genutzt werden:

- Die globale Synchronisation läßt sich fast wie in zentralisierten DBS realisieren, wenn eine globale Sperrtabelle im gemeinsamen Speicher geführt wird, mit der sämtliche Sperranforderungen bearbeitet werden. Damit kann ein aufwendiges, nachrichtenbasiertes Protokoll vermieden werden, was i.a. eine signifikante Leistungsverbesserung ermöglicht. In ähnlicher Weise können zur Kohärenzkontrolle benötigte Angaben in globalen Datenstrukturen verwaltet werden.
- Die Realisierung einer dynamischen Lastverteilung wird erleichtert, wenn dazu benötigte Informationen zur aktuellen Last- und Verteilsituation im gemeinsamen Speicher verwaltet werden. Ferner können dort zur besseren Lastbalancierung gemeinsame Auftragswarteschlangen geführt werden, auf die jeder Rechner zugreifen kann.
- Bei nicht-flüchtigem Speicher kann die Erstellung einer globalen Log-Datei vereinfacht werden, indem die Log-Daten aller Rechner direkt im Speicher gesammelt werden. Aufgrund der hohen Zugriffsgeschwindigkeit fallen die Verzögerungen dafür kaum ins Gewicht, so daß eine direkte Erstellung des globalen Logs möglich wird.

13.4.2 Realisierungsalternativen

Zur Ausgestaltung der nahen Kopplung bestehen mehrere Möglichkeiten, welche die prinzipiellen Nutzungsformen in unterschiedlichem Maße abdecken. Dies sind der Einsatz instruktions- und seitenadressierbarer Halbleiterspeicher sowie von Spezialprozessoren, die wir im folgenden diskutieren.

Instruktionsadressierbare Halbleiterspeicher

Beim Einsatz eines gemeinsamen Halbleiterspeichers ergeben sich die größten Freiheitsgrade, wenn dieser wie die Hauptspeicher instruktionsadressierbar ist. Damit steht jedem Rechner ein mächtiger Satz an Zugriffsmöglichkeiten zur Verfügung, die eine direkte Änderung der Speicherinhalte sowie die Realisierung beliebig komplexer Datenstrukturen ermöglichen. Diese Flexibilität erlaubt die Un-

terstützung aller angesprochenen Einsatzformen, sofern - wie für einige Aufgaben erforderlich - zusätzlich Nicht-Flüchtigkeit gewährleistet wird.

Allerdings ergeben sich mit einem solchen Ansatz weitgehend die gleichen Probleme wie mit dem gemeinsamen Hauptspeicher in eng gekoppelten Systemen (Kap. 3.1). Insbesondere besteht eine unzureichende Fehlerisolierung sowie die Gefahr von Engpässen, so daß der Ansatz als problematisch einzustufen ist.

Seitenadressierbare Halbleiterspeicher

Die Daten seitenadressierbarer Halbleiterspeicher sind wie für Magnetplatten nicht direkt manipulierbar. Der Datenzugriff verlangt vielmehr, die betroffenen Seiten in den Hauptspeicher zu bringen. Änderungen erfolgen zunächst in den Seitenkopien im Hauptspeicher und müssen explizit zurückgeschrieben werden. Damit ergibt sich für diese Speicher eine größere Isolierung gegenüber Hardware- und Software-Fehlern als bei Instruktionsadressierbarkeit, insbesondere gegenüber Rechnerausfällen.

Die seitenorientierte Zugriffsschnittstelle begrenzt jedoch auch die Nutzungsmöglichkeiten. Denn im wesentlichen wird nur eine Verbesserung des E/A-Verhaltens möglich, ähnlich wie es solche Speicher bereits in zentralisierten Systemen zulassen (Allokation ganzer Dateien, zusätzliche Pufferung von Seiten) [Ra92a]. Außerdem können diese Effekte auch ohne nahe Kopplung durch Nutzung von Platten-Caches und Solid-State-Disks (SSD) erreicht werden, auf die alle Rechner Zugriff haben. So können SSDs zur permanenten Allokation von Dateien genutzt werden, während Platten-Caches die Realisierung eines globalen Dateipuffers ermöglichen. Über die Platten-Caches können geänderte Seiten auch wesentlich schneller zwischen den Rechnern als über Magnetplatte ausgetauscht werden. Eine nahe Kopplung liegt dabei nicht vor, da die Zugriffszeiten auf diese Speicher im Bereich von 1-5 ms pro Seite liegen, so daß kein synchroner Zugriff möglich ist.

Synchrone Seitenzugriffe werden jedoch von sogenannten erweiterten Hauptspeichern unterstützt, die Zugriffszeiten von 10-50 Mikrosekunden pro Seite bieten [Ra93a]. Die Seitentransfers zwischen Hauptspeicher und erweiterten Hauptspeichern erfolgen durch eigene Maschinenbefehle. Die Verwaltung des erweiterten Hauptspeicher geschieht wie für den Hauptspeicher in erster Linie durch Betriebssystem-Software der Verarbeitungsrechner, also nicht durch eigene Controller, wie es für SSD und Platten-Cache der Fall ist. Bei nicht-flüchtiger Auslegung dieser Speicher und Kopplung mit allen Verarbeitungsrechnern können alle Einsatzformen von SSDs und Platten-Caches auch realisiert werden, jedoch mit verbessertem Leistungsverhalten. Weitergehende Nutzungsformen verlangen jedoch eine erweiterte Zugriffsschnittstelle. Wir werden in Kap. 13.4.3 eine Realisierungsmöglichkeit dafür betrachten; ein ähnlicher Ansatz wird im neuen IBM Parallel Sysplex verfolgt (Kap. 19.1.2).

In [DIRY89, DDY91] wurde die Verwendung eines gemeinsamen, seitenadressierbaren Halbleiterspeichers für Shared-Disk-Systeme untersucht; die Architektur wurde als *Shared Intermediate Memory (SIM)* bezeichnet. Der Zwischenspeicher diente lediglich zur globalen Pufferung von DB-Seiten. In [DDY91] stand die Untersuchung verschiedener Pufferstrategien im Mittelpunkt, wobei vor allem die Aufnahme geänderter Seiten im Zwischenspeicher als empfehlenswert angesehen wird. In [YD94] wurde ein Leistungsvergleich zwischen SIM, lose gekoppelten SD-Systemen und SN vorgenommen, wobei der nah gekoppelte SD-Ansatz eindeutig am besten abschnitt. Gegenüber lose gekoppelten SD-Systemen war hierfür das bessere E/A-Verhalten sowie der schnelle Austausch geänderter Seiten ausschlaggebend. Gegenüber SN schlug die stärkere Unempfindlichkeit gegenüber ungünstiger Partitionierbarkeit der Last bzw. Daten sowie gegenüber Lastschwankungen positiv zu Buche.

Spezialprozessoren

Eine nahe Kopplung über Spezialprozessoren liegt vor, wenn bestimmte globale Systemfunktionen in Hardware oder Mikrocode realisiert werden, um ihre schnellere Bearbeitung zu erreichen. Die Grenzen zur nahen Kopplung über gemeinsame Halbleiterspeicher sind dabei fließend, da der Spezialprozessor natürlich auch einen Speicher verwaltet. Umgekehrt kann ein gemeinsamer Halbleiterspeicher von einem "intelligenten" Controller verwaltet werden, der spezielle Funktionen an seiner Schnittstelle anbietet, die in Hardware bzw. Mikrocode realisiert werden.

Für Shared-Disk kommen Spezialprozessoren vor allem zur Realisierung eines globalen Sperrverfahrens in Betracht ("lock engine"). Hier werden an der Schnittstelle Funktionen zum Setzen und Freigeben einer Sperre (Lock, Unlock) angeboten, die durch den Spezialprozessor mit einer globalen Sperrtabelle bearbeitet werden. Durch die Verwendung solcher Operationen wird auch eine hohe Isolierung zwischen den Rechnern gewahrt, da sie die Speicherinhalte (Sperrtabelle) nicht direkt modifizieren können.

Eine hardware-gestützten Synchronisation für Shared-Disk-Systeme wird bereits seit langem durch die sogenannte "Limited Lock Facility" unterstützt, welche eine Mikrocode-Option für IBM-Platten-Controller ist [BDS79]. Sperranforderungen und -freigaben erfolgen über spezielle Kanalbefehle und können häufig mit E/A-Operationen kombiniert werden; die Gewährung einer Sperre wird über einen Interrupt mitgeteilt. Dieser Ansatz wird im Spezial-Betriebssystem TPF genutzt, das in vielen großen Reservierungssystemen eingesetzt wird [Sc87]. Der Overhead zur Synchronisation ist durch die Verwendung der E/A-Schnittstelle immer noch relativ hoch (keine synchronen Aufrufe). Zudem weist der Ansatz nur eine geringe Funktionalität auf (nur exklusive Sperren auf Rechnerebene), so daß ein Großteil des Sperrprotokolls zusätzlich durch Software in den Verarbeitungsrechnern zu realisieren ist. Überlegungen zur Realisierung leistungsfähigerer Lock-Engines

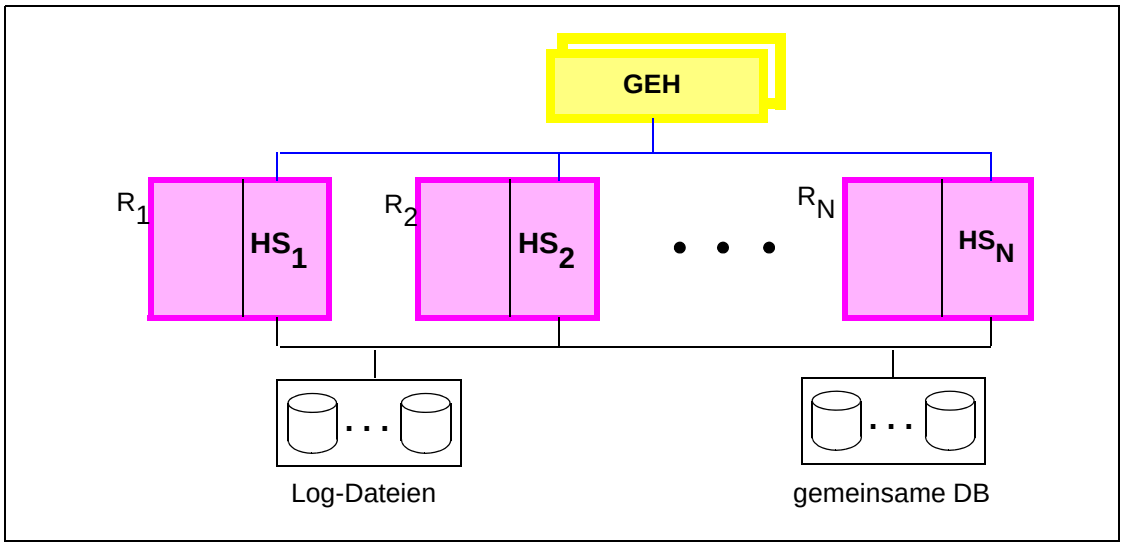
finden sich in [Ro85]. In [Se84] wird die Realisierung einer fehlertoleranten Lock-Engine, welche aus mehreren Prozessoren besteht, betrachtet.

Spezialprozessoren verursachen hohe Entwicklungskosten für sehr spezielle Aufgaben, die nur wenigen Anwendungen zugute kommen. Weiterhin können nur relativ einfache Synchronisationsprotokolle hardwaremäßig realisiert werden, so daß nicht immer eine ausreichend hohe Funktionalität gewahrt wird. Dennoch sind die Operationen umfangreicher als einfache Lese- und Schreibzugriffe auf Halbleiterspeicher, so daß sich erhöhte Bearbeitungszeiten sowie eine stärkere Engpaßgefahr ergeben.

13.4.3 Nutzung eines Globalen Erweiterten Hauptspeichers

Der Zugriff auf erweiterte Hauptspeicher im zentralen Fall besteht lediglich aus Lese- und Schreiboperationen auf Seiten. Diese einfache Schnittstelle begrenzt jedoch die Einsatzmöglichkeiten im verteilten Fall zu sehr, vor allem hinsichtlich der Realisierung von Kommunikationsaufgaben sowie zur Ablage globaler Datenstrukturen. In [Ra93a] wurde daher die Verwendung eines Globalen Erweiterten Hauptspeichers (GEH) mit erweiterter Zugriffsschnittstelle vorgeschlagen. Die Stellung des GEH innerhalb eines Shared-Disk-Systems ist in Abb. 13-6 dargestellt.

Abb. 13-6: Einsatz eines Globalen Erweiterten Hauptspeichers (GEH)



Wie für den erweiterten Hauptspeicher im zentralen Fall kann auf den GEH über Maschineninstruktionen synchron zugegriffen werden. Neben Seitenzugriffen unterstützt der GEH jedoch noch ein weiteres Zugriffsgranulat, sogenannte Einträge, um z.B. einfache globale Datenstrukturen zu realisieren. Die Eintragsgröße kann dabei das Mehrfache eines Einheitsgranulats (z.B. ein Doppelwort) sein. Neben Lese- und Schreiboperationen auf Einträgen existiert eine Compare&Swap-

Operation auf dem Einheitsgranulat, um Zugriffe mehrerer Rechner auf GEH-Datenstrukturen synchronisieren zu können. Eintragszugriffe können wesentlich schneller als Seitenzugriffe abgewickelt werden (z.B. 1-5 Mikrosekunden), da eine weit geringere Datenmenge zu transferieren ist. Trotz der erweiterten Zugriffsschnittstelle liegt keine Instruktionsadressierbarkeit auf dem GEH vor, da eine direkte Änderung von GEH-Einträgen nicht möglich ist. Stattdessen müssen Einträge wie Seiten zur Auswertung und Änderung in den Hauptspeicher gebracht werden. Aus Fehlertoleranzgründen kann der GEH nicht-flüchtig ausgelegt sein; zudem ist eine Duplizierung der Speicherinhalte in unabhängigen Partitionen möglich, um GEH-Ausfälle behandeln zu können (analog zu Spiegelplatten).

Ein solcher Speicher ist wesentlich allgemeiner nutzbar als Spezialprozessoren. Wie in [Ra93a] ausgeführt, können damit insbesondere die allgemeinen Nutzungsformen der speicherbasierten Kommunikation sowie zur E/A-Optimierung realisiert werden. Aber auch die in Kap. 13.4.1 genannten Shared-Disk-spezifischen Einsatzformen sind realisierbar, insbesondere die effiziente Synchronisation über eine globale Sperrtabelle sowie die Erstellung einer globalen Log-Datei. Das Sperrprotokoll erfordert beim doppelten Führen der Sperrinformationen weniger als 10 GEH-Zugriffe zum Setzen und Freigeben einer Sperre, so daß dafür weniger als 50 Mikrosekunden anfallen. Damit konnte ein weit besseres Leistungsverhalten als mit einem nachrichtenbasierten Protokoll erzielt werden [Ra93a, Ra93c]. Auch die Realisierung der globalen Log-Datei ist sehr einfach und effizient möglich [Ra91b].

Allerdings lassen sich mit Einträgen nur einfache Datenstrukturen mit vertretbarem Aufwand realisieren. Die fehlende Instruktionsadressierbarkeit verlangt vor allem für variabel lange Datenstrukturen, wie etwa zur Pufferverwaltung erforderlich, eine umständliche Realisierung mit einer hohen Anzahl von GEH-Zugriffen. Zudem stellt natürlich auch der GEH Spezial-Hardware dar, die zusätzliche Kosten verursacht.

Übungsaufgaben

Aufgabe 13-1: Abhängigkeiten zwischen Systemfunktionen

Diskutieren Sie die Abhängigkeiten zwischen Synchronisation, Lastverteilung und Crash-Recovery in Shared-Disk-Systemen.

Aufgabe 13-2: GEH-Synchronisation

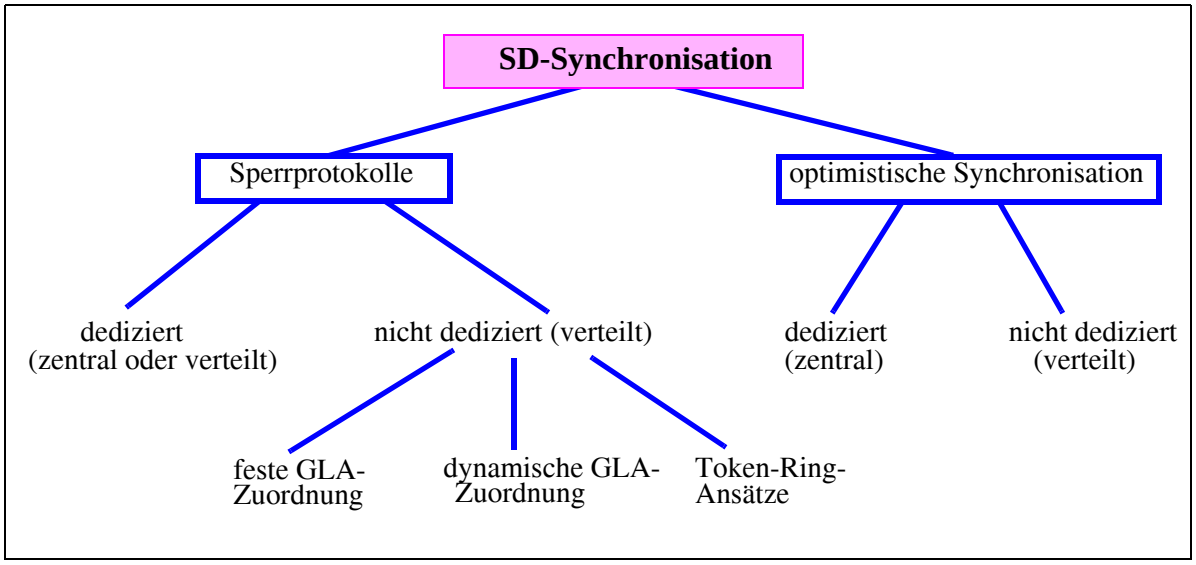
Welche GEH-Zugriffe sind notwendig zum Setzen und Freigeben einer Sperre, wenn im GEH eine globale Sperrtabelle zur Synchonisation verwendet wird ? Wie erhöht sich die Zugriffshäufigkeit bei doppelter Speicherung der Sperrtabelle ?

14 Synchronisation in Shared-Disk-DBS

Die Leistungsfähigkeit von Shared-Disk-Systemen ist wesentlich von den gewählten Algorithmen zur globalen Synchronisation und Kohärenzkontrolle abhängig, da der Kommunikationsaufwand zur Transaktionsbearbeitung weitgehend durch diese beiden Funktionen bestimmt ist. Obwohl zwischen beiden Aufgaben enge Abhängigkeiten bestehen und integrierte Lösungen anzustreben sind, separieren wir hier aus didaktischen Gründen die Beschreibung der wichtigsten Lösungsansätze. Dies erleichtert zum einen das Verständnis der grundlegenden Alternativen. Zum anderen kann damit das gesamte Lösungsspektrum, das sich durch die geeignete Kombination der Teillösungen ergibt, besser veranschaulicht werden. Schließlich können so bestehende Implementierungen innerhalb des Lösungsspektrums besser eingeordnet und beurteilt werden.

Zur Synchronisation betrachten wir in diesem Kapitel Sperrverfahren sowie optimistische Protokolle, die jeweils unter zentraler oder verteilter Kontrolle ablaufen können (Abb. 14-1). Wesentlicher ist jedoch die Unterscheidung, ob die globale Synchronisation dediziert auf ausgezeichneten Rechnern erfolgt oder nicht. In letzterem Fall liegt stets ein verteiltes Protokoll vor, bei dem jeder Verarbeitungsrechner an der globalen Synchronisation beteiligt ist. Die Darstellung geht besonders ausführlich auf die Realisierung der Sperrverfahren ein, da z.Zt. nur sie in Shared-Disk-Implementierungen verwendet werden. Zeitmarkenverfahren werden nicht betrachtet, da sie bereits für Verteilte DBS als relativ wenig sinnvoll eingestuft wurden (Kap. 8.2). Für Shared-Disk würden sie zudem keine Nachrichteneinsparungen gegenüber Sperrverfahren erlauben, jedoch zu einer erhöhten Anzahl von Transaktionsrücksetzungen führen.

Abb. 14-1: Synchronisationsverfahren für Shared-Disk-Systeme



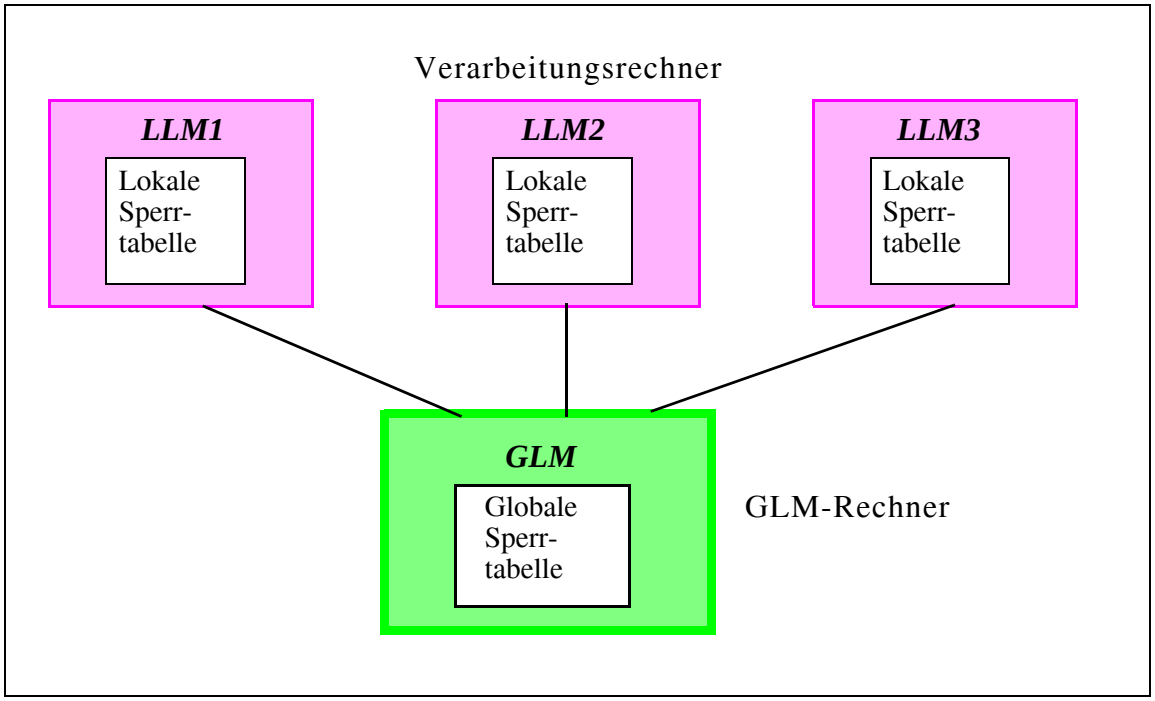
Die meisten Sperrprotokolle für Shared-Disk unterstellen, daß für jedes DB-Objekt ein *Globaler Lock-Manager (GLM)* existiert, der die globalen Sperren für das Objekt verwaltet. Die globale Sperrverantwortung oder *GLA (global lock authority)* kann dabei einem oder mehreren dedizierten Rechnern zugeordnet bzw. unter allen Verarbeitungsrechnern verteilt werden. In letzterem Fall wird noch unterschieden, ob die Zuordnung fest vorbestimmt wird oder dynamisch erfolgt (Abb. 14-1). Ein weiterer Ansatz basiert auf einer logischen Token-Ring-Topologie und verlangt die Gewährung einer Sperre durch mehrere Knoten.

Wir beschreiben zunächst dedizierte Sperrverfahren, die sowohl zentral als auch verteilt realisiert werden können. Danach werden in Kap. 14.2 verschiedene Optimierungen zur Reduzierung des Kommunikationsaufwandes eingeführt, die für alle Sperrverfahren einsetzbar sind. Die drei nachfolgenden Abschnitte behandeln nicht-dedizierte, verteilte Sperrverfahren mit fester und dynamischer GLA-Zuordnung sowie Token-Ring-Ansätze. Optimistische Synchronisationsverfahren werden in Kap. 14.6 diskutiert. Eine zusammenfassende Wertung der Synchronisationsverfahren erfolgt im nächsten Kapitel (Kap. 15) zusammen mit den Verfahren zur Kohärenzkontrolle (Kap. 15.6).
Bei der Beschreibung der Sperrverfahren unterstellen wir strikte Zwei-Phasen-Sperrprotokolle mit langen Lese- und Schreibsperren (Kap. 8.1). Auf die Behandlung von globalen Deadlocks wird nicht eingegangen, da hierfür die gleichen Techniken wie für Verteilte DBS anwendbar sind (Kap. 8.5).

14.1 Globale Sperrverwaltung auf dedizierten Rechnern

In dedizierten Sperrprotokollen erfolgt die globale Sperrverarbeitung auf ausgezeichneten Rechnern, die meist nur zur Synchronisation verwendet werden. Im Falle eines *zentralen Sperrverfahrens* besitzt ein Rechner die globale Sperrverantwortung für die gesamte Datenbank, so daß nur ein GLM existiert (Abb. 14-2). Der GLM verwaltet eine globale Sperrtabelle, mit der Sperranforderungen und -freigaben aller Rechner bearbeitet werden. Daneben existiert in jedem Verarbeitungsrechner ein *Lokaler Lock-Manager (LLM)*, der sämtliche Sperren der bei ihm laufenden Transaktionen innerhalb einer lokalen Sperrtabelle verwaltet.

Abb. 14-2: Globale Sperrverwaltung auf einem dedizierten Rechner



Sperranforderungen und -freigaben von Transaktionen werden zunächst an den LLM gestellt, der dann mit dem GLM zusammenarbeitet. Im einfachsten Fall sind sämtliche Sperroperationen an den GLM-Rechner zu schicken, was jedoch einen extrem hohen Kommunikationsaufwand und starke Antwortzeiterhöhungen verursacht. Insbesondere sind 2 Nachrichten pro Sperranforderung erforderlich; die Sperrfreigabe am Transaktionsende kann dagegen mit einer Nachricht erfolgen.

Ein offenkundiges Problem bei einem zentralen Sperrverfahren ist, daß ein einziger GLM-Rechner leicht zum Engpaß wird und somit Durchsatz und Erweiterbarkeit des Systems beeinträchtigt. Dieser Nachteil kann jedoch relativ leicht behoben werden, indem die globale Sperrverantwortung auf mehrere dedizierte Rech-

ner verteilt wird. Ein einfacher Ansatz dazu, der u.a. im Shared-Disk-System von Oracle (Parallel Server) verfolgt wird, ist die feste Aufteilung der GLA über eine Hash-Funktion, welche zu jedem Objektbezeichner den zuständigen GLM-Rechner bestimmt. Neben der Engpassgefahr wird damit auch die Verfügbarkeit verbessert, da ein GLM-Ausfall nur noch einen Teil der Datenbank betrifft (s. Übungsaufgaben).

Die Verwendung mehrerer dedizierter Rechner löst natürlich nicht das Problem der hohen Bearbeitungskosten von Sperranforderungen, wenn diese stets vom zuständigen GLM-Rechner zu bearbeiten sind. Im Falle von Spezialprozessoren (lock engines) zur GLM-Realisierung (Kap. 13.4.2), welche auch ein dediziertes Sperrprotokoll unterstützen, wird der hohe Kommunikationsaufwand durch spezielle Maschinenbefehle umgangen. Bei loser Kopplung kann der Kommunikations-Overhead durch Einsatz einer *Nachrichtenbündelung* reduziert werden. Dabei werden mehrere Sperranforderungen verschiedener Transaktionen eines Verarbeitungsrechners zusammen übertragen. Dies ermöglicht Einsparungen vor allem bei einem zentralen Sperrverfahren, wo sämtliche Anforderungen an einen GLM zu richten sind. Allerdings ergibt sich damit für die Antwortzeiten eine zusätzliche Verschlechterung, da eine Sperranforderung erst übertragen wird, wenn mehrere Anforderungen zusammen gekommen sind (bzw. ein Timeout abgelaufen ist).

Eine wirkungsvollere Optimierung ist, die Anzahl globaler Sperranforderungen zu zu senken, da dies sowohl die Antwortzeiten als auch den Kommunikations-Overhead verbessert. Dazu stellen wir im nächsten Kapitel (Kap. 15) zwei Ansätze vor.

14.2 Techniken zur Einsparung globaler Sperranforderungen

Um die Anzahl globaler Sperranforderungen zu reduzieren, ist es notwendig, daß möglichst viele Sperren ausschließlich lokal - von den LLMs der Verarbeitungsrechner - behandelt werden. Hierzu betrachten wir zwei allgemeine Ansätze, nämlich die Nutzung sogenannter Autorisierungen sowie von hierarchischen Sperren. Beide Techniken sind nicht nur bei dedizierten Sperrprotokollen anwendbar, sondern im Prinzip bei allen Sperrverfahren für Shared-Disk.

14.2.1 Lese- und Schreibautorisierungen

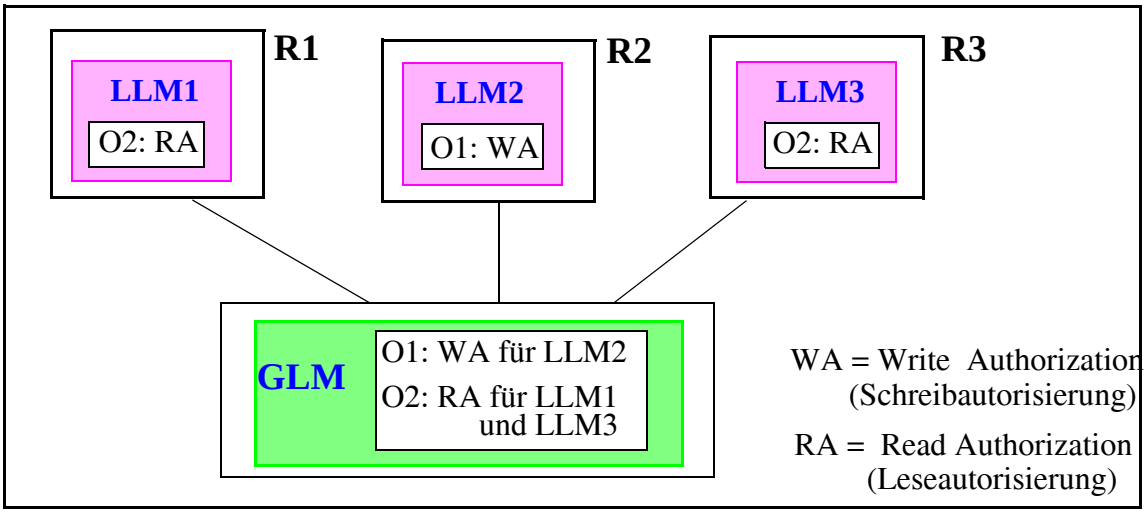
Die Anzahl globaler Sperranforderungen läßt sich reduzieren, indem man die lokalen Sperrverwalter einzelner Rechner autorisiert, Sperren ggf. lokal zu verwalten, ohne also Kommunikation mit dem GLM-Rechner vornehmen zu müssen. Dabei lassen sich zwei Arten von Autorisierungen unterscheiden, die vom globalen Sperrverwalter den lokalen Sperrverwaltern zugewiesen werden können:

- Eine *Schreibautorisierung* ermöglicht es dem lokalen Sperrverwalter (LLM), sowohl Schreib- als auch Lesesperren für das betreffende Objekt lokal zu vergeben. Eine solche Autorisierung wird vom globalen Sperrverwalter erteilt, wenn zum Zeitpunkt einer Sperranforderung kein weiterer Rechner eine Sperre auf dem betreffenden Objekt angefordert hat. Eine Schreibautorisierung kann jeweils nur einem Rechner (LLM) zuerkannt werden. Eine Rückgabe der Schreibautorisierung ist erst erforderlich, wenn ein anderer Rechner dasselbe Objekt referenzieren will.
- Eine *Leseautorisierung* ermöglicht es dem lokalen Sperrverwalter, Lesesperren für das betreffende Objekt lokal zu vergeben. Eine solche Autorisierung wird vom globalen Sperrverwalter erteilt, wenn zum Zeitpunkt einer Sperranforderung kein weiterer Rechner eine Schreibsperre auf dem betreffenden Objekt angefordert hat. Im Gegensatz zur Schreibautorisierung können mehrere Rechner eine Leseautorisierung für dasselbe Objekt halten. Eine Rückgabe der Leseautorisierung(en) ist erst erforderlich, sobald ein Rechner einen Schreibzugriff auf das betreffende Objekt durchführen will.

Beispiel 14-1

Den Einsatz dieser Autorisierungen veranschaulicht Abb. 14-3. Dabei besitzt Rechner R2 (genauer: der lokale Sperrverwalter LLM2) eine Schreibautorisierung für Objekt O1, in den Rechnern R1 und R3 liegt je eine Leseautorisierung für Objekt O2 vor. Damit können in R1 sämtliche Sperranforderungen und -freigaben bezüglich O1 ohne Kommunikationsverzögerungen behandelt werden, in R1 und R3 sind sämtlich Lesesperren auf O2 lokal gewährbar. Erfolgt jedoch eine Referenz auf O1 zum Beispiel in Rechner R1, dann führt die entsprechende Sperranforderung an den globalen Sperrverwalter (GLM) zum Entzug der Schreibautorisierung an R2. In diesem Fall entstehen für die Sperranforderung in R1 zusätzliche Verzögerungen und Nachrichten (insgesamt 4 Nachrichten). Ebenso kann eine Schreibsperre auf O2 erst gewährt werden, nachdem die beiden Leseautorisierungen entzogen wurden.

Abb. 14-3: Einsatz von Schreib- und Leseautorisierungen



Zwischen den eingeführten Autorisierungen und regulären Sperren bestehen wesentliche konzeptionelle Unterschiede. Sperren werden für einzelne Transaktionen angefordert und freigegeben, während die Autorisierungen den Rechnern bzw. LLMs zugewiesen und entzogen werden. Autorisierungen werden daher auch über

das Ende der Transaktion hinaus gehalten, deren Sperranforderung zur Erteilung einer Autorisierung führte. Damit soll die lokale Synchronisierung späterer Transaktionsausführungen an dem betreffenden Rechner ermöglicht werden.

Obwohl durch den Entzug von Schreib- bzw. Leseautorisierungen zusätzliche Nachrichten eingeführt werden, zeigen Leistungsuntersuchungen, daß die Nachrichteneinsparungen demgegenüber i.a. deutlich überwiegen [Ra93d]. Dabei ist die Effektivität der Schreibautorisierungen jedoch davon abhängig, inwieweit durch die Lastverteilung ein hohes Maß an rechnerspezifischer Lokalität (Kap. 13.3.3) erreicht werden kann. Weiterhin kann für häufig referenzierte Objekte nicht erwartet werden, daß sie für längere Zeit nur an einem Rechner referenziert werden. Für solche Objekte können Schreibautorisierungen vielmehr zu zahlreichen Entzugsverzögerungen führen. Die Effektivität von Leseautorisierungen ist dagegen weitgehend unabhängig von rechnerspezifischer Lokalität, da diese gleichzeitig an verschiedenen Rechnern gehalten werden können. Allerdings können sie nur für Objekte mit vorwiegend lesendem Zugriff ihre Wirksamkeit entfalten. Da die Kosten eines Entzuges von Lese- und Schreibautorisierungen sehr hoch sind, sollten diese freiwillig aufgegeben werden, wenn sie längere Zeit nicht mehr benötigt wurden.

Die Diskussion zeigt, daß ein selektiver Einsatz von Lese- und Schreibautorisierungen zu empfehlen ist. Werden beide Autorisierungstypen unterstützt, sind auch Konversionen zwischen ihnen möglich. Wird z.B. eine Schreibautorisierung aufgrund einer Lesesperre entzogen, kann eine Konversion in eine Leseautorisierung erfolgen. Die wichtigsten Fälle bei der Sperrbehandlung sind in Abb. 14-4 zusammengestellt. Die obere Tabelle spezifiziert die LLM-Aktionen nach einer Lese- oder Schreibsperranforderung in Abhängigkeit vom lokalen Sperrstatus. Sperrstatus NL (no lock) bedeutet dabei, daß noch keine Sperreintrag für das Objekt vorliegt, während RA bzw. WA die Existenz einer Lese- oder Schreibautorisierung anzeigen. Der Eintrag "GLM" bedeutet, daß die Sperranforderung vom GLM zu bearbeiten ist (Kommunikation). Der globale Teil der Sperrbehandlung ist in der unteren Tabelle gezeigt. Dabei bedeutet Zustand NL, daß das Objekt im ganzen System noch nicht in Bearbeitung war, während WA (RA) anzeigt, daß (wenigstens) ein anderer Rechner eine Schreib- bzw. Leseautorisierung für das betreffende Objekt hält. Der Sperrstatus "sonstige" kennzeichnet eine Konfliktsituation, da Sperren, aufgrund bereits wartender Anforderungen jedoch keine Autorisierungen vergeben sind.

Abb. 14-4: Lokale und globale Sperraktionen mit Lese- und Schreibautorisierungen

	lokaler Sperrstatus (LLM)			
	NL	RA	WA	sonstige
Lesesperr-anforderung	GLM	gewähre Sperre	lokal (1)	GLM
Schreibsperr-anforderung	GLM	GLM	lokal (1)	GLM

	globaler Sperrstatus (GLM)			
	NL	RA	WA	sonstige
Lesesperr-anforderung	gewähre RA	gewähre RA	entziehe WA (2)	globaler Konflikt (warte)
Schreibsperr-anforderung	gewähreWA	entziehe RA (3)	entziehe WA (4)	globaler Konflikt (warte)

NL = no lock
WA = write authorization
RA = read authorization

(1) lokale Sperrkonflikte möglich
(2) wenn mögl., RA-Zuweisung (Abstufung von WA nach RA)
(3) wenn mögl., WA-Zuweisung (Umwandlung von RA nach WA)
(4) wenn mögl., WA-Zuweisung

14.2.2 Hierarchische Sperren und Autorisierungen

Die Leistungsfähigkeit eines Sperrprotokolls ist generell stark vom jeweiligen Sperrgranulat abhängig. Ein feines Granulat (z.B. Satzsperren) erlaubt eine geringe Anzahl von Konflikten zwischen Transaktionen und damit eine hohe Parallelität. Auf der anderen Seite entsteht damit ein hoher Verwaltungsaufwand, da sehr viele Objekte zu sperren und eine große Anzahl von Sperreinträgen zu verwalten sind. Grobe Sperrgranulate (z.B. Satztypen) weisen die umgekehrten Eigenschaften auf. Um diese Gegensätze zu überbrücken, werden üblicherweise hierarchische Sperrverfahren eingesetzt, die zwei oder mehr Objektgranularitäten unterstützen [GR93]. Damit können für lange Transaktionen, die sehr viele Sperren anfassen, grobe und für kurze Transaktionen feine Sperrgranulate verwendet werden. Um eine Verträglichkeit beider Ansätze zu gewährleisten, müssen beim Sperren feiner Granulate die zugehörigen gröberen Granulate mit sogenannten *Anwartschaftssperren* (intention locks) belegt werden. Diese Anwartschaftssperren zeigen an, daß die Transaktion auf einer tieferen Ebene der Objekthierarchie explizite Sperren gesetzt hat. Die Anwartschaftssperren verhindern daneben, daß auf den gröberen Granulaten (höheren Ebene der Sperrhierarchie) Sperren gewährt werden, die mit denen auf den feinen Granulaten nicht verträglich sind.

Für Shared-Disk ist die Unterstützung eines hierarchischen Sperrprotokolls von besonderer Bedeutung. Denn die reduzierte Sperranzahl bei Verwendung grober Sperrgranulate bedeutet auch eine entsprechende Verbesserung in der Anzahl glo-

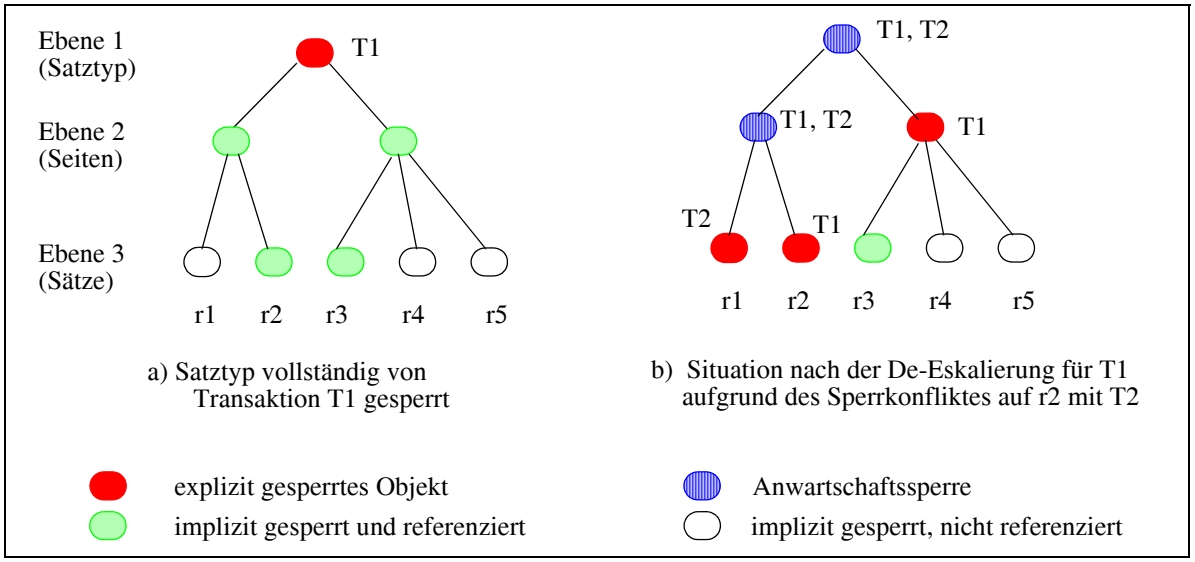
baler Sperranforderungen. Weitere Kommunikationseinsparungen sind möglich durch Anwendung der Lese- und Schreibautorisierungen auf mehreren Ebenen der Objekthierarchie. Hält z.B. ein Rechner eine Schreibautorisierung (Leseautorisierung) auf einem Satztyp, so gilt diese implizit für alle Seiten und Sätze dieses Satztyps, so daß für sie eine lokale Synchronisation möglich wird. Da diese Autorisierungen für alle Transaktionen eines Rechners gelten, ergeben sich weitergehende Einsparungen als mit hierarchischen Sperren. Denn bei diesen kommt die Verwendung grober Sperrgranulate nur jeweils einer Transaktion zugute.

Damit die mit groben Sperrgranulaten möglichen Kommunikationseinsparungen nicht zu Lasten zahlreicher Sperrkonflikte gehen, empfiehlt sich die dynamische Wahl des Sperrgranulats in Abhängigkeit der Konfliktlage [Jo91]. Bei mehreren hierarchisch geordneten Objektgranularitäten wird eine Sperre stets auf der höchsten Ebene angefordert, für die dies ohne Konflikt möglich ist. Wird etwa eine Satztypsperre auf diese Weise erworben, sind alle Objekte der darunterliegenden Ebenen implizit gesperrt und ohne Kommunikation referenzierbar. Tritt danach ein Konflikt für das grobe Sperrgranulat ein, wird für den Sperrbesitzer eine Verfeinerung seines Sperrgranulates (*De-Eskalierung*) vorgenommen, bis kein Konflikt mehr vorliegt oder das feinste Granulat erreicht wird (z.B. Satzsperren). Voraussetzung dafür ist, daß auch nach Erwerb einer Sperre für ein grobes Granulat alle folgenden Referenzen auf den feineren Granulaten in der lokalen Sperrtabelle aufgezeichnet werden. Damit wird also der lokale Wartungsaufwand gegenüber einem Sperransatz auf dem feinsten Sperrgranulat nicht reduziert, jedoch können globale Sperranforderungen eingespart werden.

Beispiel 14-2

Im Szenario von Abb. 14-5a hält Transaktion T1 zunächst eine explizite Sperre auf dem ganzen Satztyp, womit sämtliche Seiten und Sätze des Satztyps implizit gesperrt sind. Der zuständige LLM vermerkt, daß T1 die Sätze r2 und r3 referenziert hat, obwohl für sie keine explizite Sperranforderung notwendig ist. Wenn eine andere Transaktion T2 den Satz r1 bearbeiten will, beginnt sie mit ihrer Sperranforderung auf der Satztypebene, wo ein Konflikt erkannt wird. Dies veranlaßt die De-Eskalierung für T1, womit die in Abb. 14-5b gezeigte Situation entsteht. Die Satzsperre für r1 konnte an T2 gewährt werden, da die Satzzugriffe von T1 damit nicht kollidieren. T1 hält nun explizite Sperren auf der Satz- und Seitenebene sowie Anwartschaftssperren auf den höheren Ebenen. Das Beispiel verdeutlicht, daß der Ansatz die gleiche Parallelität wie mit Satzsperren ermöglicht. Dennoch lassen sich zahlreiche Nachrichten einsparen, solange keine Konflikte auftreten.

Abb. 14-5: Hierarchisches Sperren mit De-Eskalierung



Der vorgestellte Ansatz kann in analoger Weise für Lese- und Schreibautorisierungen genutzt werden. Jedoch wird eine De-Eskalierung von Autorisierungen nur für globale Konflikte erforderlich. Für Transaktionssperren auf groben Granulaten erfolgt dagegen bereits eine De-Eskalierung für einen lokalen Sperrkonflikt, wodurch für die betroffenen Objekte keine weiteren Nachrichteneinsparungen mehr möglich sind.

Beispiel 14-3

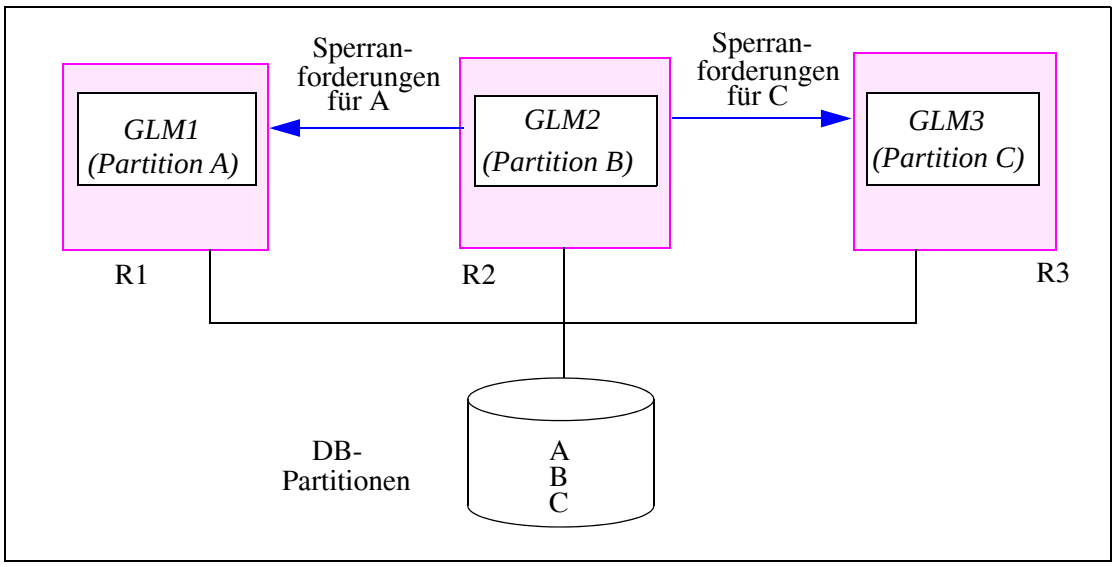
Wenn im Szenario von Abb. 14-3 der Konflikt mit der Schreibautorisierung auf O1 nicht auf der feinsten Objektebene stattfindet, kann der GLM eine De-Eskalierung für O1 verlangen. LLM2 gibt daraufhin die Schreibautorisierung für O1 zurück, behält jedoch alle Schreibautorisierungen für alle in O1 enthaltenen Objekte, die in Rechner R2 zwischenzeitlich referenziert wurden und für die kein Konflikt mit der Transaktion besteht, welche die De-Eskalierung ausgelöst hat.

14.3 Verteilte Sperrverfahren mit fester GLA-Zuordnung

Die restlichen Sperrverfahren, die noch zu behandeln sind, verwenden keine dedizierten Rechner zur globalen Synchronisation, sondern führen diese auf allen Verarbeitungsrechnern durch. Dies ergibt eine homogene Systemarchitektur, was Vorteile hinsichtlich Lastbalancierung und Verfügbarkeit ergibt; insbesondere entfällt bei der Behandlung von Rechnerausfällen die Unterscheidung zwischen Verarbeitungsrechner und GLM-Rechner. Ein weiterer großer Vorteil liegt darin, daß eine lokale Bearbeitung von Sperranforderungen und -freigaben für diejenigen Objekte möglich wird, für die der Verarbeitungsrechner die globale Sperrverantwortung besitzt. Dies wird sowohl von einer festen als auch der dynamischen Zuordnung der GLA unterstützt.

Der Ansatz mit einer festen GLA-Zuordnung wird in [Ra86b] als *Primary-Copy-Sperrverfahren* bezeichnet, obwohl das Verfahren nur bedingt mit dem gleichnamigen Verfahren für replizierte Datenbanken (Kap. 9.2) vergleichbar ist*. Wie in Abb. 14-6 gezeigt, existiert dabei in jedem Rechner ein GLM, der für eine Partition der Datenbank die globale Sperrverantwortung (GLA) besitzt. So können in Rechner R2 sämtliche Sperren auf der lokalen Partition B ohne Kommunikation behandelt werden. Anforderungen für Objekte der Partitionen A und C dagegen sind an die zuständigen Rechner R1 bzw. R3 zu stellen. Die GLA-Verteilung ist vorab festgelegt und allen Rechnern bekannt.

Abb. 14-6: GLA-Aufteilung beim Primary-Copy-Sperrverfahren



Ein einfacher Ansatz zur GLA-Allokation ist wieder die Nutzung einer Hash-Funktion, die jeden Objektbezeichner auf einen der N Rechner (GLM) abbildet. Damit sind jedoch nur relativ geringe Kommunikationseinsparungen möglich, da im Durchschnitt nur 1 von N Sperranforderungen lokal behandelt werden kann, so daß im Mittel 2 - 2/N Nachrichten pro Sperranforderung anfallen. Der Anteil lokaler Sperren läßt sich erhöhen, wenn GLA-Aufteilung und Lastverteilung aufeinander abgestimmt werden. Dies erfordert i.a. eine logische DB-Partitionierung, bei der die GLA-Zuordnung für Satztypen bzw. horizontale Fragmente (Kap. 5.3) erfolgt. Weiterhin ist ein darauf abgestimmtes affinitätsbasiertes Routing (Kap. 13.3.3) vorzunehmen, so daß Transaktionen vorzugsweise an den Rechnern bearbeitet werden, an denen die meisten ihrer Sperren lokal bearbeitet werden können. Die damit unterstützte und auf die GLA-Verteilung ausgerichtete rechner-

* Eine bessere Rechtfertigung für den Namen ergibt sich bei Berücksichtigung der Kohärenzkontrolle (Kap. 15.3.3).

spezifische Lokalität bewirkt direkte Kommunikationseinsparungen und ermöglicht eine weitgehend lokale Transaktionsverarbeitung.

Es ist möglich, das Primary-Copy-Verfahren mit Schreib- und Leseautorisierungen zu erweitern, ähnlich wie in Kap. 14.2 vorgestellt. Damit wird auch an den Verarbeitungsrechnern, die nicht die GLA für ein Objekt besitzen, eine lokale Synchronisierung möglich, sofern eine ausreichende Autorisierung vorliegt. Allerdings wird damit der Nutzen einer lokalen GLA abgeschwächt, da dann auch Sperranforderungen auf der lokalen Partition nur mit Kommunikationsverzögerungen bearbeitet werden können, wenn zuvor extern vergebene Autorisierungen zurückzunehmen sind*.

Beispiel 14-4

Im Beispiel von Abb. 14-6 habe Rechner R2 eine Schreibautorisierung für Objekt A1 von Partition A sowie eine Leseautorisierung für Objekt C1 von Partition C. Damit können sämtliche Zugriffe auf A1 in R2 lokal bearbeitet werden, obwohl R1 die GLA für dieses Objekt besitzt. Dafür müssen auch im GLA-Rechner R1 Zugriffe auf A1 verzögert werden, bis R2 die Schreibautorisierung zurückgegeben hat. In R2 können daneben Lesezugriffe auf C1 lokal behandelt werden, obwohl R3 für dieses Objekt zuständig ist. Ein Entzug durch R3 wird erst bei einer Schreibanforderung veranlaßt, wodurch auch für Transaktionen in R3 Verzögerungen entstehen können.

Der Einsatz von Schreib- und Leseautorisierungen empfiehlt sich generell bei einem einfachen, hash-basierten Ansatz zur GLA-Verteilung, da hierbei über eine lokale GLA nur geringe Einsparungen möglich werden. Soll dagegen mit GLA- und Lastverteilung ein hohes Maß an lokaler Sperrverarbeitung erreicht werden, ist v.a. der Einsatz von Schreibautorisierungen nicht zu empfehlen, da diese - wie die Nutzung einer lokalen GLA - auf rechnerspezifischer Lokalität aufbauen. Das Konzept der Schreibautorisierungen weist jedoch den Nachteil der Instabilität auf, da eine solche Autorisierung entzogen wird, sobald ein anderer Rechner auf das Objekt zugreifen will. Die GLA-Zuordnung dagegen ist stabil, so daß ein Objekt am GLA-Rechner stets lokal synchronisiert werden kann, unabhängig von der sonstigen Referenzverteilung im System. Zudem entfallen analoge Nachrichten und Verzögerungen wie beim Entzug von Schreibautorisierungen, da die GLA-Zuordnung nicht entzogen wird.

Die Vergabe von Leseautorisierungen erscheint demgegenüber auch für das Primary-Copy-Sperrverfahren empfehlenswert (für Objekte mit vorwiegendem Lesezugriff). Dieses Konzept erfordert nämlich keine rechnerspezifische Lokalität. Es ist vielmehr in der Lage, die Abhängigkeiten zur günstigen Partitionierbarkeit der

* Die Vergabe von Schreib- und Leseautorisierungen ist nur für Rechner notwendig, die für das Objekt nicht die GLA halten, da der GLM-Rechner generell lokal über die Sperrvergabe entscheiden kann.

Transaktionslast und der Datenbank zu reduzieren [Ra93d]. Insbesondere können die gleichen Objekte in mehreren Rechnern parallel bearbeitet und lokal synchronisiert werden, solange Lesezugriffe vorliegen.

Der Hauptnachteil bei der Nutzung einer lokalen GLA im Rahmen des Primary-Copy-Sperrverfahrens liegt darin, daß ähnlich wie für Verteilte DBS bzw. Shared-Nothing-Systeme eine DB-Partitionierung gefunden werden muß, die eine weitgehend lokale Verarbeitung von Transaktionen auf N Rechnern zuläßt. Es ist jedoch zu beachten, daß diese logische DB-Partitionierung nur für Synchronisationszwekke verwendet wird, daß jedoch weiterhin jeder Rechner auf die gesamte DB zugreifen kann. Dies erleichtert zum einen die Anpassung der Partitionierung, z.B. bei geänderter Rechneranzahl oder stark wechselndem Transaktionsprofil, da hiermit keine physische Umverteilung der Daten einhergeht. Zudem bleibt das hohe Lastbalancierungspotential der Shared-Disk-Architektur bestehen, da weiterhin jeder Knoten alle DB-Operationen ausführen kann.

14.4 Verteilte Sperrverfahren mit dynamischer GLA-Zuordnung

Das Problem des Primary-Copy-Sperrverfahrens, eine möglichst geeignete DB-Partitionierung zu finden, entfällt bei dynamischer GLA-Zuordnung. Hierbei wird keine vordefinierte GLA-Zuordnung angewendet, sondern derjenige Rechner, der ein Objekt erstmalig referenziert, erhält automatisch die globale Sperrverantwortung. Da jeder Rechner weiß, für welche Objekte er die GLA hält, können für diese Objekte alle Sperranforderungen und -freigaben lokal bearbeitet werden. Für die sonstigen Objekte muß jedoch erst herausgefunden werden, ob und an welchem Rechner bereits ein zuständiger GLM vorliegt. Diese *GLM-Lokalisierung* erfordert i.a. zusätzliche Nachrichten gegenüber einem Ansatz mit fester GLA-Zuordnung, die allen Rechnern bekannt ist.

Nachdem ein Rechner die GLA für ein Objekt erhalten hat, behält er die GLM-Funktion i.a. mindestens solange, bis die von ihm vergebenen Sperren wieder freigegeben sind. Dies garantiert, daß die Sperrfreigabe (bzw. Sperrkonversion) vom gleichen Rechner vorgenommen wird wie die Sperrvergabe, so daß eine erneute GLM-Lokalisierung entfällt. Nach Freigabe der letzten Transaktionssperre kann die GLA für das Objekt auch aufgegeben werden, so daß bei späterer Referenz ein neuer GLM-Rechner bestimmt wird. Alternativ dazu kann die GLA beibehalten werden, um nachfolgende Referenzen am gleichen Rechner ohne Verzögerungen bearbeiten zu können. Sollte der spätere Zugriff auf das Objekt von einem anderen Rechner aus erfolgen, kann eine *GLA-Migration* an diesen Rechner vorgesehen werden, um dort eine lokale Verarbeitung zu ermöglichen. Generell wird es durch die Dynamik der GLA-Zuordnungen schwieriger als bei fester GLA-Zuordnung,

durch die Lastverteilung ein hohes Maß an lokaler Sperrverarbeitung zu ermöglichen.

Zur GLM-Lokalisierung könnte - ähnlich wie bei fester GLA-Zuordnung - die Verteilungsinformation repliziert an jedem Knoten geführt werden. Damit muß jedoch jede Änderung in der GLA-Zuordnung im ganzen System propagiert werden, was einen enormen Aufwand verursachen kann. Günstiger erscheint daher, die GLA-Zuordnung in einem nicht replizierten Directory zu verwalten. Das Directory kann dabei zentral an einem Knoten oder partitioniert an mehreren Rechnern vorliegen. Im letzteren Fall kann etwa wieder über eine Hash-Funktion festgelegt werden, welcher Rechner für ein Objekt die GLA-Zuordnung führt. Diese Indirektion führt jedoch zu bis zu 4 Nachrichten für eine Sperranforderung. Denn zunächst ist der zuständige Directory-Knoten zu konsultieren, um den GLM-Rechner zu ermitteln (2 Nachrichten), bevor die eigentliche Sperre angefordert werden kann*. Die Anpassung der Directory-Information bei Aufgabe oder Migration der GLA erfordert i.a. ebenfalls zusätzliche Nachrichten.

Die Kosten der GLM-Lokalisierung sind jedoch auch stark davon abhängig, für welche Objektgranularitäten die GLA-Zuordnung erfolgt. Insbesondere kann im Rahmen eines hierarchischen Sperrprotokolls die GLA-Vergabe auf grobe Granulate wie Satztypen beschränkt werden. Damit können in dem Rechner, der die GLA für einen Satztyp (Relation) bekommen hat, sämtliche zugehörigen Seiten und Sätze lokal synchronisiert werden. An anderen Rechnern wird eine GLM-Lokalisierung zudem nur für Satztypen erforderlich; für die untergeordneten Objekte ist damit der zuständige GLM-Rechner bereits ermittelt. Allerdings kann die Verwendung grober GLA-Granulate leicht zu einer ungleichmäßigen Unterstützung der lokalen Sperrvergabe sowie ungleicher Verteilung des Sperraufwandes an den einzelnen Rechnern führen.

Beispiel 14-5

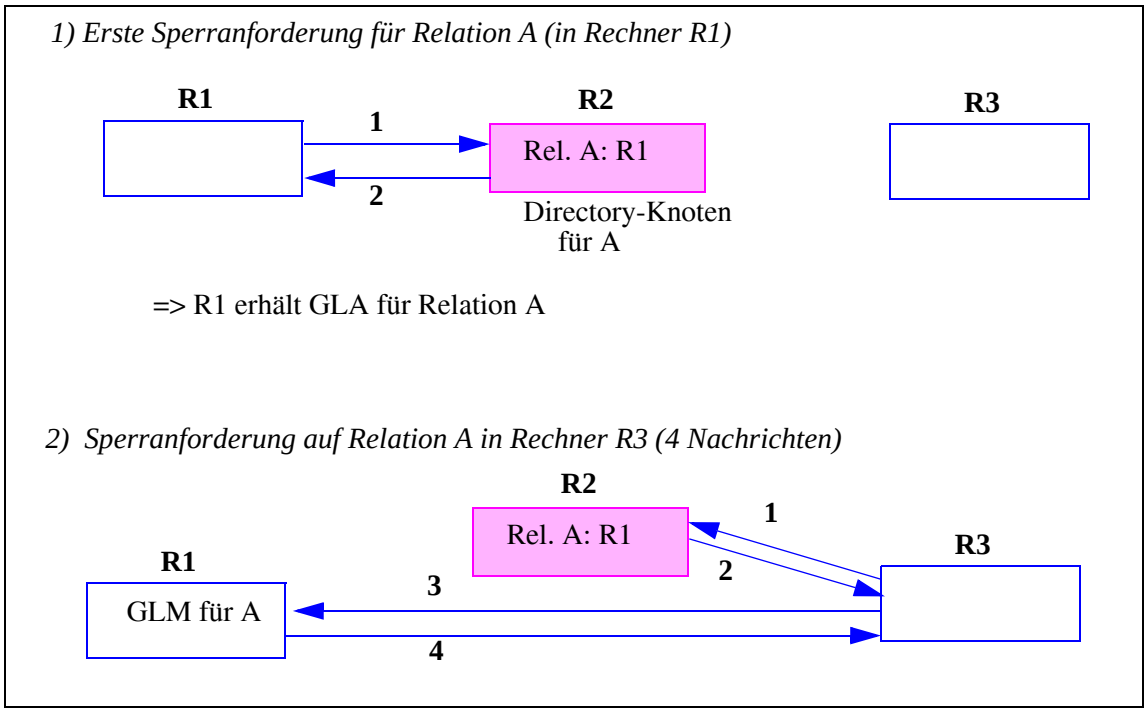
Im Szenario von Abb. 14-7 wird Relation A zunächst in R1 erstmalig referenziert. Dazu wird der für A zuständige Directory-Rechner R2 konsultiert, wobei festgestellt wird, daß für A noch kein GLM vorliegt. Deshalb überträgt R2 die GLA für A an R1 und erweitert seine Directory-Information entsprechend. In R1 können dann die Sperren für A fortan lokal bearbeitet werden. Im weiteren Verlauf der Transaktionsverarbeitung soll Relation A in Rechner R3 referenziert werden (Schritt 2 in Abb. 14-7). Für die zugehörige Sperranforderung werden 4 Nachrichten erforderlich, nämlich zwei zur GLM-Lokalisierung und zwei zum Sperrerwerb selbst. Wenn keine Sperren für A mehr vergeben bzw. angefordert sind, kann R1 die GLA aufgeben, wozu die Directory-Information in R2 zurückzusetzen ist. Bei einer Migration der GLA für A, z.B. nach R3, ist die Directory-Information in R2 ebenso anzupassen.

---

* Einsparungen ergeben sich, wenn Transaktionsrechner, Directory-Knoten und GLM-Rechner nicht verschieden sind.

Ein solcher Ansatz mit dynamischer GLA-Zuordnung wird in DEC VaxCluster-Systemen unterstützt, und zwar durch den sogenannten Distributed Lock Manager (DLM) des VMS-Betriebssystems, der von den Shared-Disk-DBS genutzt wird [KLS86, ST87, Jo91]. Die GLM-Lokalisierung basiert auf einem über eine Hash-Funktion partitionierten Directory.

Abb. 14-7: Sperrszenario mit dynamischer GLA-Zuordnung

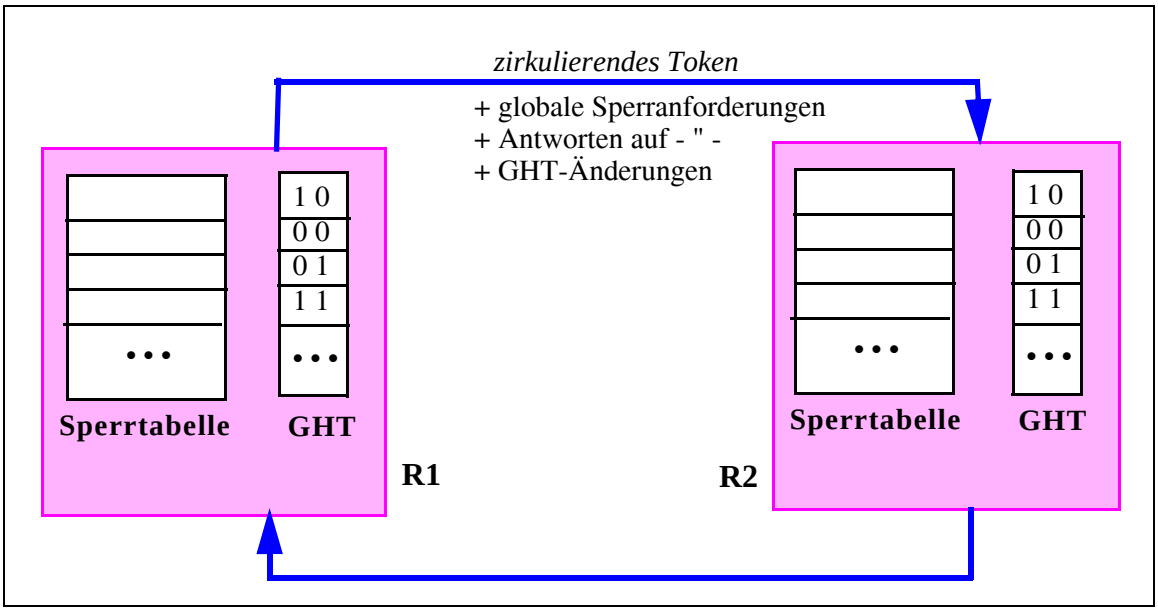


14.5 Token-Ring-Sperrprotokolle

Eine Alternative zur Nutzung eines Globalen Lock-Managers ist, globale Sperranforderungen durch alle Rechner zu gewähren. Ein solcher Ansatz kann durch eine logische Token-Ring-Topologie erreicht werden, bei der globale Sperranforderungen zusammen mit einem Token reihum an alle Rechner weitergeleitet werden. Dabei überprüft ein Rechner nach Eingang des Tokens, ob die globalen Sperranforderungen anderer Rechner mit den lokal laufenden Transaktionen verträglich sind. Die Ergebnisse dieser Überprüfungen werden dann mit dem Token an den nächsten Rechner in der Ring-Reihenfolge übergeben, so daß nach einem Ringumlauf das Ergebnis der Sperranforderung feststeht. Der Kommunikationsaufwand eines solchen Aufwandes ist relativ gering, da eine Nachrichtenbündelung quasi im Protokoll eingebaut ist (Übertragung mehrerer Sperranforderungen und Antworten in einer Nachricht). Der Kommunikationsaufwand läßt sich noch weiter re-

duzieren, indem das Token an jedem Rechner eine bestimmte Zeit festgehalten wird, um einen größeren Bündelungseffekt zu erzielen. Allerdings liegt für globale Sperranforderungen eine sehr hohe Bearbeitungsdauer vor, die mit der Token-Verweilzeit sowie der Rechneranzahl zunimmt. Token-Ring-Ansätze sind daher allenfalls für sehr wenige Rechner von Interesse.

Abb. 14-8: Prinzip des Pass-the-Buck-Protokolls von IMS



Ein solches Token-Ring-Verfahren namens *Pass the Buck* wird von IMS Data Sharing eingesetzt [SUW82, Yu87]. Es ist auf zwei Rechner beschränkt. Um die Anzahl globaler Sperranforderungen zu reduzieren, wird dabei eine spezielle Globale Hash-Tabelle (GHT) verwendet, die an beiden Knoten repliziert ist. Darin werden für jede Hash-Klasse der Sperrtabelle zwei Bits (interest bits) geführt, die anzeigen, welche der beiden Rechner "Interesse" für Objekte der Hash-Klasse haben (Abb. 14-8). Interesse eines Rechners liegt dabei vor, sobald eine seiner Transaktionen für ein Objekt der Hash-Klasse eine Sperranforderung gestellt hat. Zeigt die Hash-Tabelle, daß nur der lokale Rechner Interesse hat, dann können die zugehörigen Objekte sofort lokal synchronisiert werden. So zeigt die Bitkombination 10 (bzw. 01) an, daß nur in Rechner 1 (bzw. Rechner 2) Sperranforderungen für Objekte der Hash-Klasse vorliegen. Damit wird an Rechner 1 (bzw. 2) eine lokale Sperrvergabe für sämtliche Objekte der Hash-Klasse möglich, da ein Konflikt mit Transaktionen des anderen Rechners ausgeschlossen ist*. GHT-Änderungen sind nur bei Token-Besitz möglich. Sie werden mit dem Token an den anderen Rechner übertragen, damit dieser seine Version der Hash-Tabelle aktualisieren kann.

Die GHT realisiert offenbar Schreibautorisierungen auf Ebene von Hash-Klassen. Im Prinzip ist es auch bei diesem Sperrprotokoll möglich, Schreib- und Leseautorisierungen für andere Objektgranularitäten zur weitergehenden Einsparung globaler Sperranforderungen zu nutzen [HR85, Ra88b].

14.6 Optimistische Synchronisation

Optimistische Synchronisationsverfahren für zentralisierte DBS sowie Verteilte DBS wurden in Kap. 8.3 eingeführt. Für Shared-Disk sind diese Verfahren von besonderem Interesse, da sie gegenüber Sperrverfahren starke Kommunikationseinsparungen versprechen. Denn Kommunikation zur Synchronisation fällt nur am Transaktionsende zur globalen Validierung an. Der Aufwand dafür ist im Gegensatz zu den Sperrprotokollen auch weitgehend unabhängig vom Referenzverhalten der Last, der Lastverteilung sowie der Rechneranzahl, so daß eine bessere Skalierbarkeit möglich ist. Allerdings werden wir sehen (Kap. 15), daß die Kohärenzkontrolle für optimistische Synchronisationsverfahren einen höheren Kommunikationsaufwand als für Sperrverfahren erfordert. Zudem bestehen natürlich die generellen Schwierigkeiten optimistischer Verfahren, insbesondere die Gefahr vieler Transaktionsrücksetzungen sowie einige ungelöste Implementierungsprobleme wie die Unterstützung von feinen Synchronisationsgranulaten und Zugriffspfaden (Kap. 8.6).

Ähnlich wie in Verteilten DBS kann die Validierung zentral an einem dedizierten Rechner als auch verteilt an den Verarbeitungsrechnern erfolgen. Beide Ansätze sollen im folgenden nur kurz angesprochen werden; ausführlichere Beschreibungen finden sich in [Ra87, Ra88b].

14.6.1 Zentrale Validierung

Die zentrale Validierung kann für Shared-Disk praktisch genauso wie in Verteilten DBS realisiert werden (Kap. 8.3.2). Am Ende jeder Transaktion werden ihr Read- und Write-Set an den zentralen Validierungsknoten gesendet, der entscheidet, ob ein Konflikt mit anderen Transaktionen aufgetreten ist. Dabei ist nur eine BOCC-artige Validierung möglich, so daß im Konfliktfall die validierende Transaktion abzubrechen ist. Zur Synchronisation entstehen damit lediglich zwei Nachrichten pro Transaktion. Wie empirische Simulationsstudien gezeigt haben, ist der damit verbundene Kommunikationsaufwand meist deutlich geringer als mit (optimierten) globalen Sperrverfahren [Ra93d]. Der Validierungsaufwand pro Transak-

* Selbst bei einem 00-Eintrag kann eine lokale Sperrvergabe erfolgen, jedoch nur bei Token-Besitz. In IMS wird in diesem Fall aus Recovery-Gründen die Sperrvergabe zusätzlich verzögert, bis die GHT des anderen Rechners aktualisiert wurde.

tion ist ebenfalls relativ gering, so daß selbst mit einem einzigen Validierungsrechner hohe Transaktionsraten möglich sind.

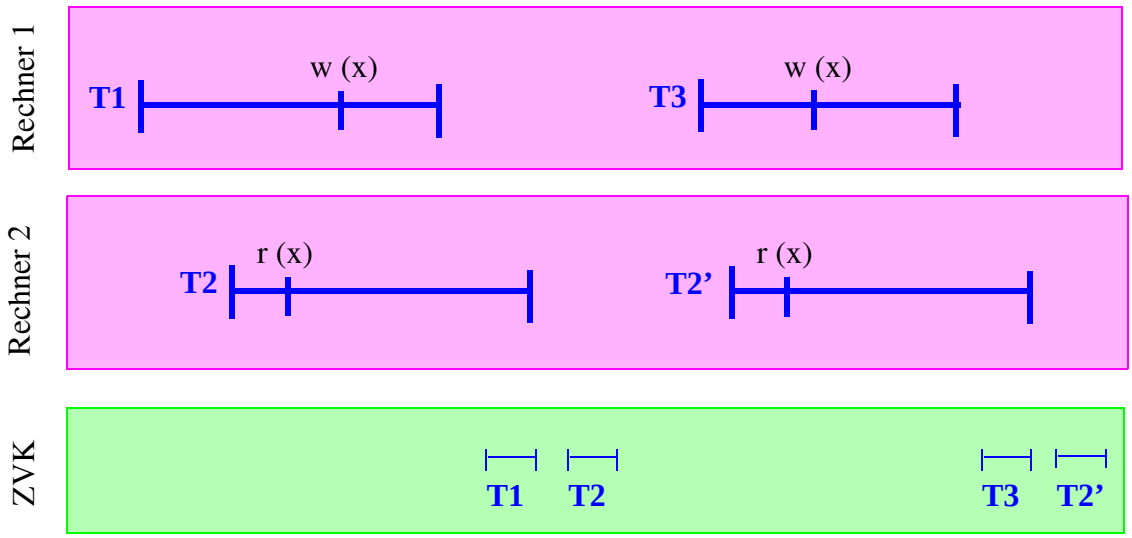
Um das mehrfache Scheitern derselben Transaktion zu verhindern, kann eine Kombination mit einem zentralen Sperrverfahren vorgenommen werden. Hierzu wird ausgenutzt, daß vor allem kürzere Transaktionen bei der wiederholten Ausführung mit hoher Wahrscheinlichkeit die gleichen Objekte wie bei der ersten Ausführung referenzieren. Die betreffenden Objekte sind im zentralen Rechner natürlich bekannt, da sie im Read-Set bzw. Write-Set der Transaktion vermerkt sind. Für die Neuausführung einer erfolglos validierten Transaktion erfolgt nun die Umstellung auf eine pessimistische Synchronisation. Dazu wird vor dem erneuten Start für jedes von der Transaktion gelesene bzw. geänderte Objekt eine Lese- bzw. Schreibsperre am zentralen Knoten gesetzt, was ohne zusätzliche Kommunikation möglich ist. Diese Sperren verhindern, daß andere Transaktionen die Objekte in unverträglicher Weise bearbeiten. Dazu wird eine erweiterte Validierung notwendig, welche die gesetzten Sperren berücksichtigt und bei unverträglichem Zugriff die validierende (optimistisch synchronisierte) Transaktion abbricht. Somit ist die erfolgreiche Bearbeitung der zweiten Transaktionsausführung garantiert, wenn keine zusätzlichen Objekte referenziert werden*. Ferner wird ein Verhungern von Transaktionen, wie ansonsten bei BOCC-artiger Synchronisation möglich (Kap. 8.3), verhindert.

Beispiel 14-6

Abb. 14-9 zeigt die parallele Verarbeitung mehrerer Transaktionen in zwei Verarbeitungsrechnern sowie die zugehörigen Validierungen am zentralen Validierungsknoten (ZVK). Nach der erfolgreichen Validierung von T1 wird bei der Validierung von T2 ein Konflikt für Objekt x erkannt, so daß T2 erneut an Rechner 2 ausgeführt werden muß (Transaktion T2'). Bei rein optimistischer Synchronisation scheitert jedoch auch die zweite Ausführung T2', da eine Transaktion T3 zwischenzeitlich Objekt x erfolgreich ändern konnte. Bei Kombination mit einem Sperrverfahren dagegen wird am zentralen Rechner nach der erfolglosen Validierung von T2 eine Schreibsperre für x gesetzt. Dies führt dazu, daß die Validierung von T3 scheitert, während die zweite Ausführung T2' erfolgreich zu Ende kommt.

* Für während der Wiederholung erstmals referenzierte Objekte ist entweder eine Validierung durchzuführen, oder es wird sicherheitshalber für sie eine Sperre am zentralen Rechner bereits während der Transaktionsausführung eingeholt.

Abb. 14-9: Szenario zur zentralen Validierung



Die Sperren können nicht nur zur erfolglosen Validierung optimistisch synchronisierter Transaktionen führen, sondern auch zu Sperrkonflikten (Blockierungen) mit anderen Transaktionen, deren Validierung gescheitert ist. Deadlocks lassen sich jedoch umgehen, da sämtliche Sperren vor der erneuten Transaktionsausführung angefordert werden, ähnlich einem Preclaiming-Ansatz (Kap. 8.5.1). Die Wiederausführung einer Transaktion kommt auch meist mit geringerem E/A-Aufwand aus, da die benötigten Objekte vielfach noch aufgrund der ersten Ausführung im Hauptspeicher gepuffert sind.

14.6.2 Verteilte Validierung

Verteilte Validierungsverfahren führen für Shared-Disk zu einem höheren Kommunikationsaufwand als zentrale Ansätze, da eine Transaktion i.a. an allen Verarbeitungsrechnern zu validieren ist, um alle Konflikte erkennen zu können. Dies kann mit einem Token-Ring-Ansatz erreicht werden, bei dem Validierungsaufträge mit dem Token reihum an alle Rechner gestellt werden, ähnlich wie für Token-Ring-Sperrverfahren. Diese Sequentialisierung der Validierungen führt jedoch wieder zu langen Bearbeitungszeiten und ist nur für eine geringe Rechneranzahl ausgelegt. Eine Alternative besteht in einem Broadcast-Ansatz, mit dem die Validierung einer Transaktion gleichzeitig an allen Rechnern gestartet wird. Hierbei entsteht jedoch ein sehr hoher Kommunikationsaufwand. Eine Verbesserung wird möglich, wenn unter den Rechnern eine feste Aufteilung der Synchronisationsverantwortung vorgenommen wird, ähnlich wie beim Primary-Copy-Sperrverfahren. Denn in diesem Fall braucht eine Transaktion nur noch an denjenigen Rechnern

zu validieren, welche für wenigstens eines der bearbeiteten Objekte die globale Synchronisationsverantwortung besitzt [Ra87].

Die verteilte Validierung wirft daneben ähnliche Probleme auf wie in Verteilten DBS (Kap. 8.3.3). Insbesondere empfiehlt sich zur Wahrung der globalen Serialisierbarkeit wieder, die Validierungen an allen Rechnern in der gleichen Reihenfolge zu bearbeiten. Für die Behandlung "unsicherer" Änderungen bestehen ebenso die gleichen Lösungsmöglichkeiten wie für Verteilte DBS (Kap. 8.3.3).

Übungsaufgaben

Aufgabe 14-1: Zentrale Sperrprotokolle

In Verteilten DBS wurden zentrale Sperrverfahren als ungeeignet eingestuft. Warum kommen sie für Shared-Disk eher in Betracht?

Aufgabe 14-2: GLM-Ausfall

Wie kann nach einem GLM-Ausfall die DB-Verarbeitung korrekt fortgesetzt werden?

Aufgabe 14-3: Dedizierte Sperrverfahren

Welche Vorteile bietet ein zentrales Sperrprotokoll gegenüber einem verteilten Ansatz auf dedizierten Rechnern?

Aufgabe 14-4: Lese- und Schreibautorisierungen

Auf ein Objekt O ergebe sich folgende Verteilung von R- und X-Sperranforderungen in den Rechnern R1, R2 und R3, wobei jede Sperranforderung durch eine andere Transaktion verursacht sei.

Zeitpunkt	R1	R2	R3
t1	X		
t2	R		
t3	X		
t4		R	
t5			R
t6	R		
t7		R	
t8			R
t9			X

Mit einem einfachen zentralen Sperrprotokoll sind 18 Nachrichten zur Behandlung der 9 Sperranforderungen erforderlich.

Wieviele Nachrichten werden benötigt, falls

- nur Schreibautorisierungen (keine Leseautorisierungen)
- nur Leseautorisierungen (keine Schreibautorisierungen)

- Lese- und Schreibautorisierungen unterstützt werden?

Welche Faktoren bestimmen die Nachrichtenanzahl?

Wie ändert sich das Bild, wenn auch die Nachrichten zur Sperrfreigabe berücksichtigt werden? Eine Sperre soll dabei mit 1 Nachricht (unquittiert) freigegeben werden.

Aufgabe 14-5: Primary-Copy-Sperrverfahren

Für das Beispiel in Aufgabe 14-4 soll ein Primary-Copy-Sperrprotokoll zur globalen Synchronisation eingesetzt werden, wobei R1 die GLA für das Objekt halte. Wieviele Nachrichten sind für die Anforderung sowie Freigabe der Sperren erforderlich

- ohne Leseautorisierungen
- mit Leseautorisierungen?

Aufgabe 14-6: Dynamische GLA-Zuordnung

Für das Beispiel in Aufgabe 14-4 soll ein verteiltes Sperrverfahren mit dynamischer GLA-Zuordnung eingesetzt werden.

- Wieviele Nachrichten sind für die Anforderung sowie Freigabe der Sperren erforderlich, wenn R3 der für das Objekt zuständige Directory-Knoten ist und die GLA am ersten GLM-Rechner verbleibt?
- Ist die Verwendung von Schreib- bzw. Leseautorisierungen sinnvoll?
- Ist eine GLA-Migration sinnvoll, wenn eine externe Sperranforderung eintrifft und beim GLM-Rechner ansonsten keine Sperranforderung vorliegt?

15 Kohärenzkontrolle

Aufgabe der Kohärenzkontrolle in Shared-Disk-Systemen ist die mit der dynamischen Hauptspeicher-Replikation von DB-Seiten einhergehenden Inferenzen zu behandeln (Kap. 13.3.2). Hierzu sind im wesentlichen zwei Teilaufgaben zu erfüllen:

- *Erkennen bzw. Vermeiden von Pufferinvalidierungen*
  Da auf veraltete Objekte im DB-Puffer eines Rechners nicht zugegriffen werden darf, müssen diese Pufferinvalidierungen entweder erkannt bzw. von vorneherein vermieden werden.
- *Propagieren von DB-Änderungen*
  Änderungen werden zunächst im Hauptspeicher eines Rechners vorgenommen, müssen jedoch den Transaktionen aller Rechner zugänglich gemacht werden. Dies erfordert die Propagierung von Änderungen zwischen den Rechnern. Ebenso müssen die Änderungen natürlich auch in die auf Externspeicher vorliegende physische Datenbank eingebracht werden.

Für beide Teilprobleme, die mit einem Minimum an Performance-Einbußen zu lösen sind, bestehen mehrere Lösungsalternativen, die in diesem Kapitel vorgestellt werden. Viele der Techniken sind dabei nicht auf Shared-Disk-DBS beschränkt, sondern können auch in Workstation/Server-Umgebungen genutzt werden, wo die replizierte Objektpufferung auf Workstation-Seite zu analogen Invalidierungsproblemen führt [CFLS91].

Zwischen Kohärenzkontrolle und Synchronisation bestehen enge Abhängigkeiten. So erfolgen natürlich sämtliche Objektzugriffe im Puffer unter der Kontrolle der Synchronisation, so daß durch geeignete Erweiterungen dieser Funktion auch die Zugriffe auf veraltete Objektkopien verhindert bzw. erkannt werden können. Für Sperrverfahren wird es somit möglich, den Zugriff auf invalidierte Daten zu umgehen, da vor jedem Objektzugriff eine Sperre anzufordern ist. Für optimistische Synchronisationsverfahren kann dies dagegen i.a. nicht gewährleistet werden, da Objektzugriffe zunächst unsynchronisiert erfolgen. Hier wird spätestens bei der Validierung der Zugriff auf veraltete Objekte entdeckt, was jedoch zur Rücksetzung der betroffenen Transaktion führt. Eine schnelle Beseitigung von Pufferinvalidierungen ist daher wesentlich zur Begrenzung der Rücksetzhäufigkeit.

Eine weitere wichtige Abhängigkeit betrifft das Synchronisationsgranulat. Da Seiten die Einheiten des Transfers zwischen Extern- und Hauptspeicher sowie der Hauptspeicherpufferung darstellen, vereinfacht sich die Kohärenzkontrolle bei einer Synchronisation auf Seitenebene (oder gröberen Granulaten). Weiterhin ergeben sich in diesem Fall effiziente Lösungsmöglichkeiten zur Kohärenzkontrolle mit geringem Kommunikationsbedarf. Bei feineren Granulaten wie Satzsperren besteht das Problem, daß verschiedene Sätze derselben Seite parallel in verschiedenen Rechnern geändert werden können. Damit sind die Seitenkopien beider Rechner nicht vollständig aktuell, wodurch sich die Konsistenzwahrung der physischen Datenbank sowie die Propagierung von Änderungen verkomplizieren. Die folgenden Ausführungen werden daher eine *Synchronisation auf Seitenebene* unterstellen; auf die Verwendung feinerer Granulate wird in Kap. 15.5 eingegangen.

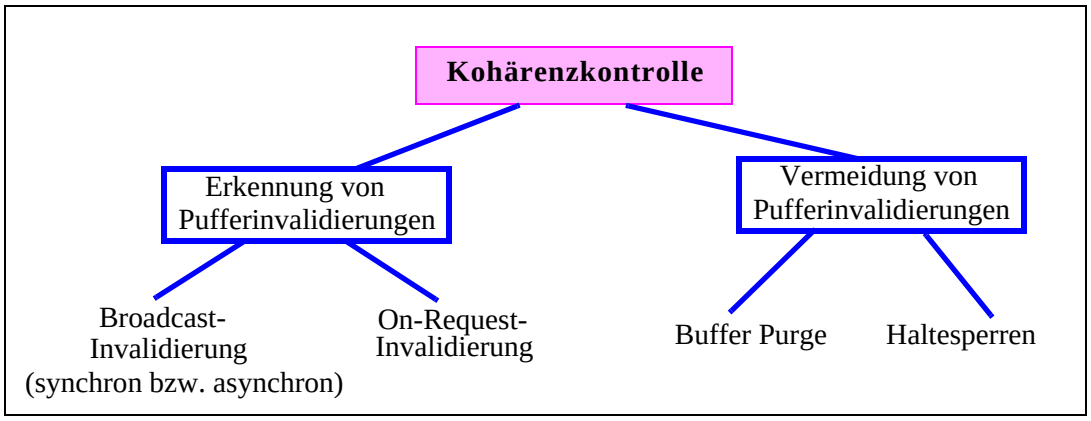
Wir geben zunächst einen Überblick über das Lösungsspektrum der Kohärenzkontrolle, indem wir für die beiden genannten Teilprobleme jeweils mehrere Alternativen spezifizieren. In den darauffolgenden Kapiteln 15.2 bis 15.4 werden dann die drei wichtigsten Verfahrensklassen zur Erkennung/Vermeidung von Pufferinvalidierungen im Detail vorgestellt, wobei jeweils auf verschiedene Kombinationsmöglichkeiten zur Propagierung von Änderungen sowie zur Synchronisation eingegangen wird. Danach behandeln wir Möglichkeiten zur Kohärenzkontrolle beim Einsatz von Satzsperren. Abschließend geben wir eine zusammenfassende Wertung der Verfahren zur Kohärenzkontrolle sowie ihrer Kombination mit Synchronisationsverfahren (Kap. 15.6).

15.1 Verfahrensüberblick

15.1.1 Behandlung von Pufferinvalidierungen

Zur Erkennung/Vermeidung von Pufferinvalidierungen bestehen im wesentlichen die in Abb. 15-1 gezeigten Alternativen, welche in den folgenden Kapiteln noch näher vorgestellt werden. Erkennungsansätze sind dadurch charakterisiert, daß bei ihnen - im Gegensatz zu Vermeidungsstrategien - Pufferinvalidierungen möglich sind, diese jedoch entdeckt und aus dem Puffer entfernt werden. Bei der Broadcast-Invalidierung (Kap. 15.2) werden Änderungen über Broadcast-Nachrichten allen Rechnern mitgeteilt, so daß Pufferinvalidierungen frühzeitig erkannt werden. Dieser Ansatz ist für alle Synchronisationsmethoden anwendbar. Eine On-Request-Invalidierung (Kap. 15.3) ist dagegen nur für Sperrverfahren möglich. Dabei wird bei der Anforderung einer globalen Sperre - ohne Mehraufwand an Kommunikation - entschieden, ob eine gepufferte Objektkopie noch gültig ist oder nicht.

Abb. 15-1: Alternativen zur Erkennung/Vermeidung von Pufferinvalidierungen



Ein *Buffer-Purge*-Ansatz vermeidet Pufferinvalidierungen, indem DB-Seiten bereits am Transaktionsende aus dem Puffer entfernt werden, um ihre Invalidierung durch andere Transaktionen zu vermeiden. Ein solcher Ansatz kann jedoch zu einem sehr schlechten E/A-Verhalten führen, da im Puffer nur noch Lokalität innerhalb von Transaktionen zur Einsparung von Externspeicherzugriffen nutzbar ist. Ferner müssen sämtliche Änderungen am Transaktionsende in die physische Datenbank durchgeschrieben werden (Force-Strategie, s.u.). Der Buffer-Purge-Ansatz führt aufgrund dieser Beschränkungen meist zu einem inakzeptablen Leistungsverhalten und soll hier nicht weiter betrachtet werden*. Eine effektivere Methode zur Vermeidung von Pufferinvalidierungen ist der Einsatz von Haltesperren (Kap. 15.4). Dabei bleiben Seiten über das Transaktionsende hinaus gepuffert, jedoch werden sie durch spezielle Haltesperren (retention locks) vor ihrer Invalidierung geschützt. Dieser Ansatz ist nur in Verbindung mit pessimistischer Synchronisation möglich.

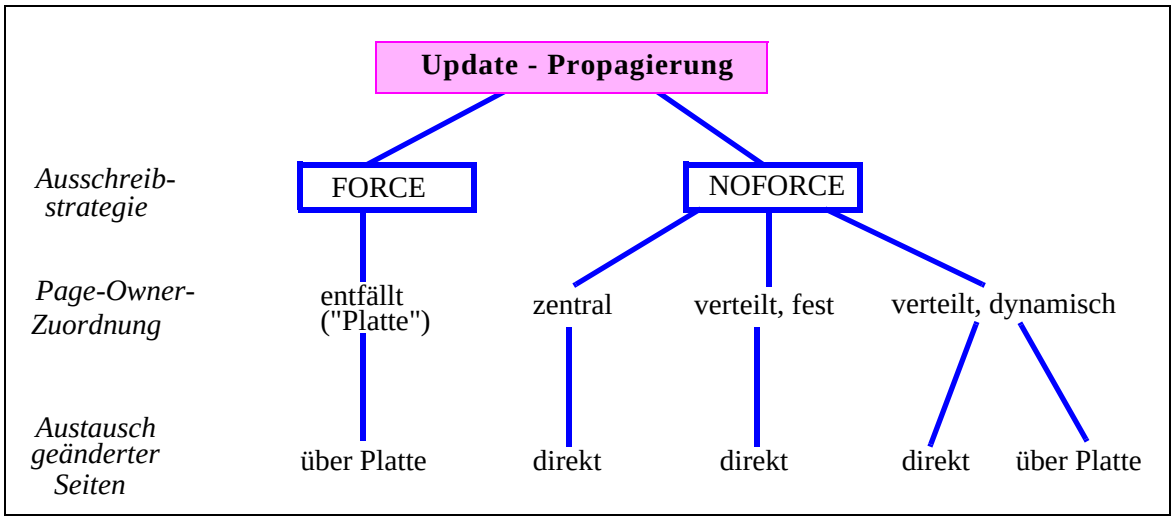
15.1.2 Update-Propagierung

Zur Propagierung von Änderungen bestehen auch mehrere Alternativen, die in Abb. 15-2 klassifiziert sind. Man kann dabei zwei Arten der Update-Propagierung unterscheiden, nämlich eine vertikale und eine horizontale. Die vertikale Propagierung von Änderungen bezieht sich auf den Transfer von Änderungen zwischen Haupt- und Externspeicher, während die horizontale Propagierung den Transfer von Objekten zwischen den Rechnern betrifft.

---

* Eine andere allgemeine Vermeidungsstrategie wäre, nur Seiten zu puffern, die nicht geändert werden. Da jedoch i.a. jede DB-Seite änderbar ist, kommt ein solcher "Ansatz" ebenfalls nicht in Betracht.

Abb. 15-2: Alternativen zur Propagierung von Änderungen



Force vs. Noforce

Für die vertikale Update-Propagierung bzw. die Ausschreibstrategie bestehen die zwei gleichen Alternativen wie für zentralisierte DBS, nämlich ein Force- oder Noforce-Ansatz [HR83]. Die *Force-Strategie* verlangt, daß alle von einer Transaktion geänderten Seiten spätestens am Transaktionsende in die physische Datenbank auf Externspeicher ausgeschrieben werden. Dieser Ansatz bedingt i.a. ein sehr ungünstiges Leistungsverhalten, da die Schreibvorgänge zu signifikanten Antwortzeitverschlechterungen sowie hohem E/A-Overhead führen können. Auf der anderen Seite vereinfacht sich die Behandlung von Rechnerfehlern, da die Änderungen aller erfolgreicher Transaktionen bereits eingebracht sind, so daß eine Redo-Recovery entfällt. Für Shared-Disk ergibt sich ferner eine erhebliche Vereinfachung der Kohärenzkontrolle. Denn das Durchschreiben der Änderung garantiert, daß die letztgültige Version geänderter Seiten in der physischen Datenbank auf Externspeicher vorliegt! Da jeder Rechner direkten Zugriff auf die physische Datenbank hat, ist bei Force das Problem der Update-Propagierung somit bereits durch die Ausschreibsstrategie gelöst. Der Austausch geänderter Seiten zwischen Rechnern erfolgt also stets über die gemeinsamen Platten.

Die *Noforce*-Alternative erlaubt i.d.R. ein signifikant besseres Leistungsverhalten als Force, da geänderte Seiten erst verzögert und asynchron ausgeschrieben werden, so daß die Ausschreibverzögerungen am Transaktionsende wegfallen. Die E/A-Häufigkeit wird reduziert, da auch für Schreibvorgänge Lokalität im Puffer genutzt werden kann, um mehrere Änderungen verschiedener Transaktionen zu akkumulieren. Dafür ist jetzt für Shared-Disk eine aufwendigere Crash-Recovery (Kap. 13.3.4) sowie komplexere Kohärenzkontrolle zu unterstützen. Da die physische DB i.a. nicht auf dem aktuellen Stand ist, muß ermittelt werden, an welchen

Rechnern die aktuellen Versionen geänderter Seiten vorliegen. Daneben ist eine explizite horizontale Update-Propagierung zu unterstützen. Eine weitere Schwierigkeit liegt darin, daß eine Seite in mehreren Rechnern als geändert gegenüber der Version auf Platte vorliegen kann. Damit kann eine Ausschreibkoordinierung erforderlich werden, um das Ausschreiben einer veralteten Seitenversion zu verhindern, da dies zum Verlust von Änderungen führen könnte.

Page-Owner-Zuordnung

Zur Lösung dieser Probleme wird üblicherweise das Konzept des *Page-Owners* eingesetzt. Dabei fungiert für jede (gegenüber der physischen Datenbank) geänderte Seite genau einer der Rechner als Page-Owner, der im wesentlichen zwei Aufgaben wahrnimmt. Zum einen versorgt er andere Rechner auf Anforderung mit der aktuellen Version der Seite. Zum anderen ist er für das Ausschreiben der geänderten Seite verantwortlich, wodurch eine Ausschreibkoordinierung mit anderen Rechnern entfällt. Beide Aufgaben entfallen offenbar, wenn sich die aktuelle Version der Seite in der physischen DB befindet, so daß in diesem Fall kein Page-Owner erforderlich ist (bzw. die Platte könnte als Page-Owner betrachtet werden).

Abb. 15-2 zeigt, daß für die Zuordnung der Page-Owner-Funktion zu den Rechnern ähnliche Alternativen bestehen wie für die GLA-Zuordnung für Sperrverfahren (Kap. 14):

- *Zentraler Page-Owner*
  Ein einzelner Rechner fungiert als Page-Owner für die gesamte Datenbank, so daß er sämtliche Änderungen bereitstellen muß. Dies erfordert jedoch auch, daß die Änderungen aller Rechner zunächst an ihn zu übertragen sind, damit er diese Aufgabe erfüllen kann. Dies macht den zentralen Knoten mit hoher Wahrscheinlichkeit zum Engpaß.
- *Verteilte, feste Page-Owner-Zuordnung*
  Mehrere bzw. alle Rechner können als Page-Owner auftreten, wobei vorab festgelegt ist, für welche DB-Partition ein Rechner die Page-Owner-Funktion hat. Diese Zuordnung ist allen Rechnern bekannt, so daß der für eine Seite zuständige Page-Owner ohne Kommunikation ermittelt werden kann.
- *Verteilte, dynamische Page-Owner-Zuordnung*
  Der Rechner, an dem eine Seite zuletzt geändert wurde, wird zum Page-Owner. Der Vorteil gegenüber der festen Zuordnung liegt darin, daß geänderte Seite nicht erst zum zuständigen Page-Owner transferiert werden müssen. Dafür ist jedoch der Page-Owner für eine bestimmte Seite i.a. unbekannt, so daß eine Lokalisierung notwendig wird, um die aktuellste Version der Seite bei ihm anfordern zu können.

Von diesen Alternativen ist die dynamische Page-Owner-Zuordnung am universellsten einsetzbar; ihre Realisierung wird in den nachfolgenden Kapiteln noch genauer betrachtet. Der erste Ansatz hat für Shared-Disk nur geringe Relevanz, liegt jedoch i.a. in Workstation/Server-Systemen vor, wo der Server-Rechner als zentraler Page-Owner für alle Workstations fungiert. Die zweite Alternative kann

sehr effektiv mit dem Primary-Copy-Sperrverfahren kombiniert werden (s. Kap. 15.3).

Alternativen für Seitentransfers

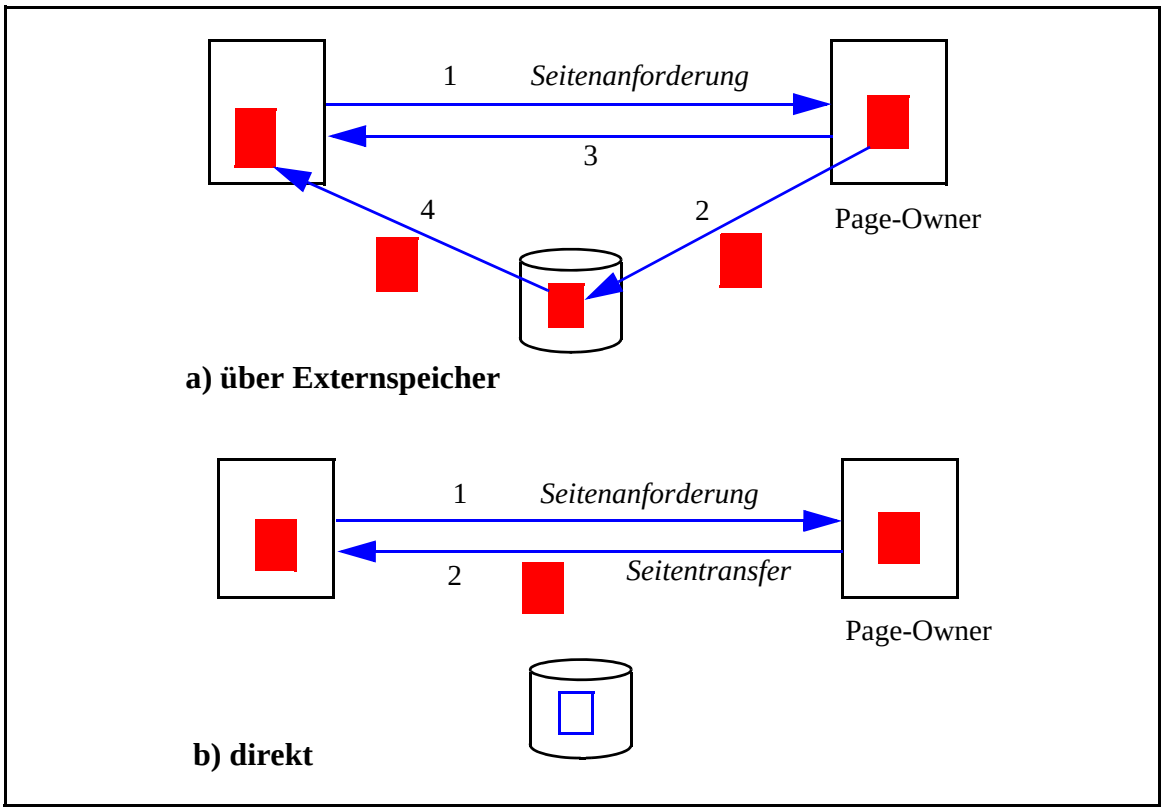
Das letzte Klassifikationsmerkmal von Abb. 15-2 ist ebenfalls nur für Noforce relevant und betrifft den Austausch geänderter Seiten zur horizontalen Update-Propagierung. Dabei sind im wesentlichen zwei Alternativen zu unterscheiden, nämlich den Austausch geänderter Seiten über die gemeinsamen Externspeicher sowie der direkte Austausch über das Kommunikationsnetz. Wie Abb. 15-3 verdeutlicht, ist dabei der Seitenaustausch über Platte signifikant langsamer als der direkte Transfer. Denn im ersteren Fall schreibt der Page-Owner nach Eintreffen einer Seitenanforderung (page request) zunächst die Seite aus und signalisiert dann dem anfordernden Rechner, daß die Seite von Externspeicher gelesen werden kann. Somit sind neben zwei Nachrichten zwei synchrone Plattenzugriffe erforderlich, so daß Verzögerungen von über 30 ms zu erwarten sind. Beim direkten Seitentransfer dagegen entfallen die Externspeicherzugriffe, da die Seite bereits in der Antwortnachricht auf die Seitenanforderung zurückgeliefert wird. Ein solcher Seitenaustausch dürfte innerhalb von 1-5 ms abgewickelt werden, also um rund eine Größenordnung schneller*.

Allerdings kann der Seitenaustausch über Externspeicher erheblich beschleunigt werden, wenn die gemeinsamen Platten mit einem nicht-flüchtigen Platten-Cache ausgestattet sind. Denn dann fallen für die Externspeicherzugriffe lediglich die Cache-Zugriffszeiten an, wodurch eine Gesamtverzögerung im Bereich von 4-8 ms erreicht werden kann. Noch kürzere Transferzeiten sind möglich bei naher Kopplung über einen schnellen, gemeinsamen Halbleiterspeicher (Kap. 13.4). Der Austausch geänderter Seiten über Externspeicher bringt zudem erhebliche Vorteile hinsichtlich der Crash-Recovery mit sich. Denn damit ist nach einem Rechnerausfall gewährleistet, daß eine Redo-Recovery auf die Änderungen des ausgefallenen Rechners beschränkt werden, welche in dessen lokaler Log-Datei protokolliert sind. Denn die Änderungen anderer Rechner sind aufgrund des Seitenaustauschs über Externspeicher bereits in der physischen DB enthalten. Bei direktem Seitenaustausch dagegen enthält die physische Datenbank i.a. noch nicht die Änderungen anderer Rechner (Abb. 15-3b). Diese müssen daher im Rahmen der Crash-Re-

* Direkte Seitentransfers erlauben auch eine Verbesserung des E/A-Verhaltens, da es schneller möglich ist, eine Seite im Hauptspeicher eines anderen Rechners anzufordern, als eine Seite von Platte zu lesen. Diese Seitentransfer können dabei im Prinzip auch für ungeänderte Seiten genutzt werden, sofern bekannt ist, an welchen Rechnern sie vorliegen. In einigen Systemen ist jedoch der CPU-Overhead für Seitenübertragungen zwischen Rechnern höher als für einen E/A-Vorgang.

covery ebenfalls wiederholt werden, was die Anwendung einer globalen Log-Datei verlangt (Kap. 13.3.4).

Abb. 15-3: Austausch geänderter Seiten



Neben Seitentransfers *vom* Page-Owner zu einem anfordernden Rechner können auch Seitentransfers *zum* Page-Owner notwendig werden. Dies ist bei zentralisierter und fester Page-Owner-Zurordnung notwendig, wenn die Änderung außerhalb des Page-Owner-Rechners stattfand. Für diese Transfers kommt nur die direkte Übertragung in Betracht (da ansonsten kein Page-Owner mehr benötigt würde). Daraus folgt, daß ein Seitenaustausch über Platte nur bei dynamischer Page-Owner-Zuordnung sinnvoll ist (Abb. 15-2).

15.2 Broadcast-Invalidierung

Bei diesem Ansatz wird am Ende jeder Änderungstransaktion eine Broadcast-Nachricht an alle anderen Rechner verschickt, in der mitgeteilt wird, welche Seiten von der Transaktion geändert wurden. Daraufhin überprüft jeder Rechner, ob in seinem Puffer Kopien der geänderten Seiten vorliegen, und entfernt diese gegebenenfalls, da sie nun veraltet sind. Damit werden invalidierte Seiten frühzeitig aus dem Puffer eliminiert und der Zugriff auf sie verhindert. Von Nachteil ist jedoch der sehr hohe Kommunikationsaufwand, der quadratisch mit der Rechner-

zahl N zunimmt. Denn zum einen wächst der Kommunikationsbedarf pro Transaktion proportional mit N, zum anderen steigt die Transaktionsrate linear mit N. Damit schränkt die Broadcast-Invalidierung die Skalierbarkeit stark ein und kommt nur für kleine Konfigurationen in Betracht.

Im folgenden diskutieren wir zunächst Unterschiede in der Realisierung für Sperrverfahren sowie optimistische Synchronisationsverfahren. Danach stellen wir Alternativen zur Update-Propagierung vor.

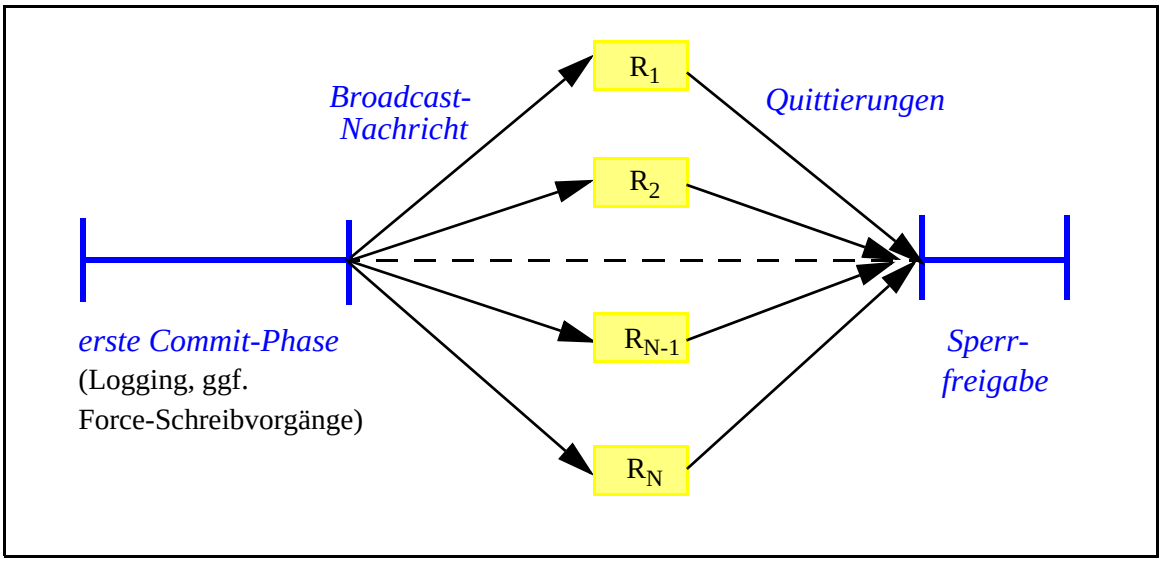
15.2.1 Zusammenwirken mit der Synchronisation

Abb. 15-4 zeigt die durch die Broadcast-Invalidierung erforderlichen Erweiterungen der Commit-Behandlung beim Einsatz von Sperrverfahren. Das Verschicken der Broadcast-Nachricht erfolgt dabei nach der ersten Commit-Phase, wenn die Wiederholbarkeit und damit der Erfolg der Änderungstransaktion sichergestellt ist. Erst nachdem alle Rechner das Eintreffen und die Bearbeitung der Broadcast-Nachricht quittiert haben, kann die Sperrfreigabe erfolgen. Denn eine frühere Sperrfreigabe würde es ermöglichen, daß die Sperre an einem anderen Rechner erworben und eine veraltete Seitenkopie im Puffer referenziert wird. Da Änderungstransaktionen synchron auf das Eintreffen der Quittierungen warten müssen, bezeichnen wir diesen Ansatz als *synchrone Broadcast-Lösung*. Der offensichtliche Nachteil neben dem hohen Kommunikations-Overhead ist die starke Erhöhung der Antwortzeiten und Sperrhaltezeiten. Dennoch verfolgten die ersten Shared-Disk-Implementierungen (IMS Data Sharing [SUW82], Computer Console [WIH83] etc.) einen solch einfachen Ansatz, und zwar in Verbindung mit einer Force-Ausschreibstrategie, welche die Commit-Bearbeitung zusätzlich stark verlängert.

Für optimistische Synchronisationsverfahren wird der Zugriff auf veraltete Seiten erst während der Validierung erkannt und führt zur Rücksetzung der betroffenen Transaktion. Die Broadcast-Benachrichtigungen sind wesentlich zur Begrenzung der Rücksetzhäufigkeit, da sie eine frühzeitige Eliminierung von Pufferinvalidierungen gestatten. Allerdings kann damit der Zugriff auf veraltete Daten nicht ausgeschlossen werden, da der Objektzugriff unsynchronisiert erfolgt und die erfolgreiche Beendigung einer Änderungstransaktion alle zuvor bereits referenzierten Objektversionen nachträglich invalidiert. Daher genügt bei optimistischer Synchronisation auch eine *asynchrone Broadcast-Invalidierung*, bei der die Broadcast-Nachricht erst nach Ende der Änderungstransaktion verschickt wird, ohne also ihre Antwortzeit zu verlängern. Die Broadcast-Nachrichten können auch dazu genutzt werden, Transaktionen aller Rechner abzubrechen, die bereits auf nunmehr invalidierte Objektversionen zugegriffen haben [Ra87]. Durch den frühzeitigen

Abbruch dieser zum Scheitern verurteilten Transaktionen kann unnötige Arbeit zu ihrer Beendigung eingespart werden.

Abb. 15-4: Synchrone Broadcast-Invalidierung



15.2.2 Propagierung von Änderungen

Implementierungen der Broadcast-Invalidierung verwendeten in der Regel eine Force-Ausschreibstrategie. In diesem Fall kann die aktuelle Version einer Seite stets von der physischen DB gelesen werden, so daß weitere Maßnahmen zur Kohärenzkontrolle entfallen. Bei Noforce dagegen sind Zusatzmaßnahmen zur Propagierung von Änderungen erforderlich, wozu sich ein dynamischer Page-Owner-Ansatz (Kap. 15.1.2) empfiehlt. Dies bedeutet, daß der Rechner, an dem die letzte Änderung einer Seite erfolgte, zu deren Page-Owner wird und damit verantwortlich ist, anderen Rechnern Kopien der Seite zur Verfügung zu stellen sowie die Seite auszuschreiben. Diese Aufgaben migrieren an einen anderen Rechner, wenn dort eine erneute Änderung der Seite erfolgt.

Die wesentliche Aufgabe bei diesem Ansatz besteht in der Lokalisierung des Page-Owners. Hierbei bestehen ähnliche Möglichkeiten wie zur GLM-Lokalisierung bei dynamischen GLA-Zuordnung (Kap. 14.4). Allerdings kann jetzt die Lokalisierung ohne zusätzliche Nachrichten erreicht werden! Hierzu betrachten wir im folgenden zwei Möglichkeiten, wobei die erste auf die Broadcast-Invalidierung ausgerichtet und sowohl bei Sperrverfahren als auch optimistischer Synchronisation anwendbar ist. Der zweite Ansatz zur Page-Owner-Lokalisierung ist nur für Sperrverfahren möglich, jedoch auch bei einer On-Request-Invalidierung sowie beim Haltesperren-Ansatz einsetzbar.

Im Falle der Broadcast-Invalidierung kann jeder Rechner in einer replizierten *Page-Owner-Tabelle* vermerken, welcher Rechner für eine geänderte Seite zuständig ist. Diese Tabelle kann ohne zusätzliche Kommunikation geführt werden, indem in den Broadcast-Nachricht mitgeteilt wird, wo die Seitenänderung stattfand und daher die Page-Owner-Funktion vorliegt. Die Page-Owner-Tabelle wird immer konsultiert, sobald eine zu referenzierende Seite nicht im Hauptspeicher gepuffert ist. Besteht für die Seite ein Page-Owner-Eintrag, wird eine Seite bei dem vermerkten Rechner angefordert; ansonsten wird die Seite von der physischen Datenbank eingelesen. Ausschreibvorgänge durch den Page-Owner können mit anderen Broadcast-Nachrichten des Rechners asynchron an alle Rechner mitgeteilt werden. Daraufhin werden die entsprechenden Einträge der Page-Owner-Tabellen gelöscht, da die Seiten nun vom Externspeicher gelesen werden können. Diese Vorgehensweise begrenzt den Umfang der Page-Owner-Tabellen und ist notwendig zur weitgehenden Vermeidung von Seitenanforderungen, die vom Page-Owner nicht mehr befriedigt werden können.

Für Sperrverfahren, die einen Globalen Lock-Manager-Ansatz verfolgen, ist eine noch effizientere Lösung zur Page-Owner-Lokalisierung möglich. Denn in diesem Fall kann für geänderte Seiten der Page-Owner in der globalen Sperrtabelle vermerkt werden, so daß die replizierte Speicherung der Page-Owner-Angaben entfällt. Die Nutzung der Page-Owner-Information ist auch ohne zusätzlichen Kommunikationsvorgänge möglich. Denn der GLM kann bei der Antwort auf eine Sperranforderung mitteilen, welcher Rechner als Page-Owner fungiert. Die Eintragung eines Page-Owners ist zusammen mit der Bearbeitung einer Schreibsperre möglich, die für eine Änderung benötigt wird. Nach Ausschreiben einer geänderten Seite durch den Page-Owner kann dies wiederum asynchron an den GLM-Rechner mitgeteilt werden, woraufhin der Eintrag zurückgesetzt wird. Beispiele zum Einsatz dieser Page-Owner-Lokalisierung werden später vorgestellt (Beispiel 15-3, Beispiel 15-6).

Die Erweiterung der globalen Sperrtabelle für Aufgaben der Kohärenzkontrolle ist ein sehr effektiver Optimierungsansatz, die weitergehend nutzbar ist. Dies wird bei den noch vorzustellenden Alternativen der On-Request-Invalidierung sowie den Haltesperren getan. Jedoch auch für die synchrone Broadcast-Invalidierung läßt sich der Kommunikationsaufwand reduzieren, wenn in der globalen Sperrtabelle genau vermerkt wird, welche Seiten wo gepuffert sind. Bei der Gewährung einer Schreibsperre kann der GLM dann mitteilen, wo die betreffende Seite gepuffert ist, so daß am Transaktionsende nur noch diese Rechner benachrichtigt werden*. Dieser Ansatz entspricht somit einer *Multicast-Invalidierung*; in [DY93] wurde er als *selektive Notifikation* bezeichnet. Die Wartung der Pufferbelegung in der globalen Sperrtabelle kann weitgehend ohne zusätzliche Nachrichten geschehen, indem

das Einlagern einer Seite mit der Sperranforderung signalisiert wird. Ausschreibvorgänge können wiederum asynchron und kombiniert mit anderen Sperrnachrichten mitgeteilt werden. Dennoch bleibt ein enormer Wartungsaufwand, da jede Änderung in der Pufferbelegung eines Rechners in der globalen Sperrtabelle nachgeführt werden muß. Dieser Aufwand steigt mit der Puffergröße sowie der Rechneranzahl.

15.3 On-Request-Invalidierung

Der hohe Kommunikationsaufwand einer Broadcast-Invalidierung kann für Sperrverfahren umgangen werden, wenn die globale Sperrtabelle auch um Informationen zur Erkennung von Pufferinvalidierungen erweitert wird. Dies erlaubt es dem GLM bei der Bearbeitung einer Sperranforderung ("on request"), die vor jedem Objektzugriff erforderlich ist, darüber zu entscheiden, ob eine in dem Rechner gepufferte Kopie der Seite noch aktuell ist [Ra86b]. Damit können veraltete Seiten völlig ohne zusätzliche Kommunikationsvorgänge und Antwortzeitverschlechterungen erkannt werden, wodurch eine weit effizientere Kohärenzkontrolle als mit der Broadcast-Invalidierung erreicht wird. Die Information zur Erkennung von Pufferinvalidierungen ist bei Änderung einer Seite anzupassen. Auch dies kann ohne Zusatzkommunikation zusammen mit der Freigabe der Schreibsperre vorgenommen werden. Von Nachteil gegenüber der Broadcast-Invalidierung ist, daß invalidierte Seiten erst bei erneutem Zugriffswunsch entdeckt und somit später aus dem Puffer entfernt werden. Dies kann zu schlechteren Trefferraten führen, da der Puffer durch nicht nutzbare Seiten belegt wird.

Wir diskutieren im folgenden zwei Realisierungsansätze zur On-Request-Invalidierung, nämlich die Nutzung von Versionsnummern sowie von Invalidierungsvektoren. Zur Einfachheit unterstellen wir dabei zunächst eine Force-Strategie, bei der die aktuelle Seite von der physischen Datenbank gelesen werden kann. Anschließend betrachten wir dann die Integration der Update-Propagierung für Noforce, und zwar sowohl bei dynamischer als auch fester Page-Owner-Zuordnung.

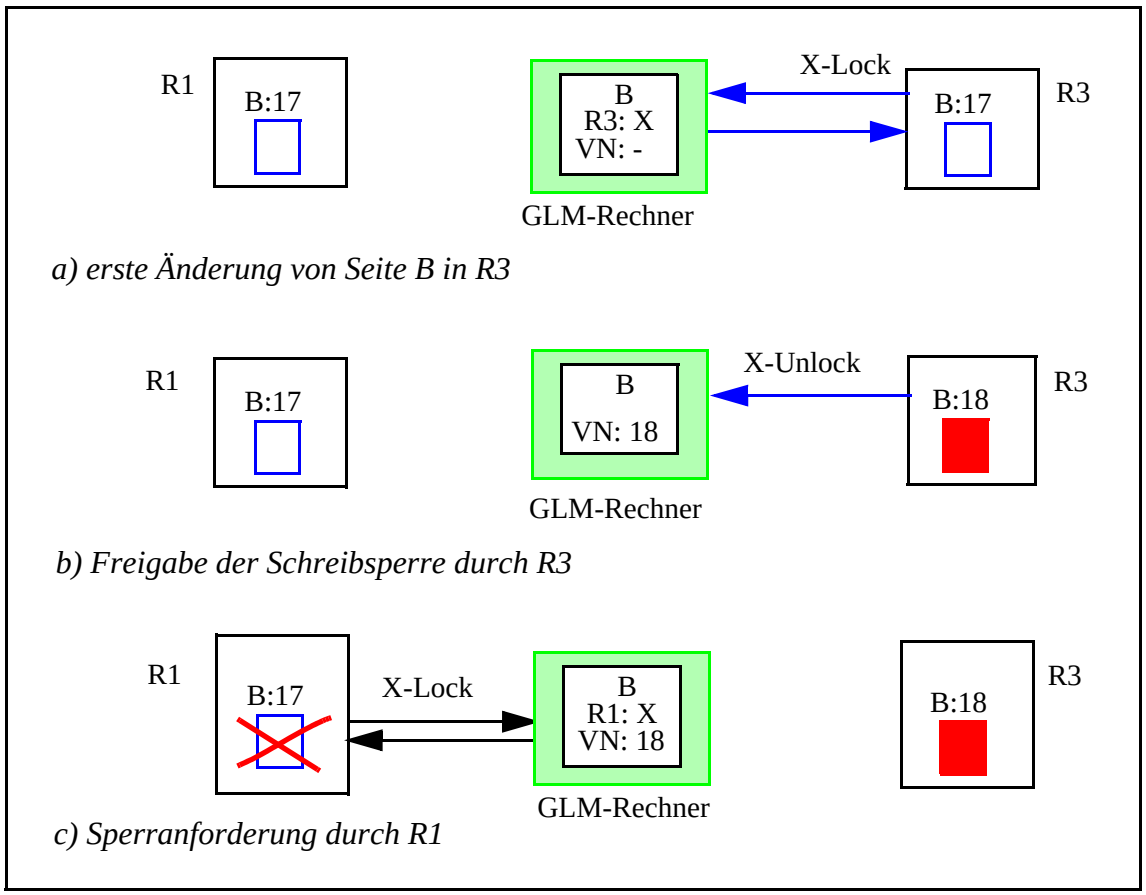
15.3.1 Versionsnummern und Invalidierungsvektoren

Ein einfacher Ansatz zur Erkennung invalidierter Seiten, der u.a. für DEC VaxCluster eingesetzt wird [KLS86], ist die Verwendung einer Versionsnummer bzw. eines Änderungszählers pro Seite (page sequence number). Die Versionsnummer wird im Seitenkopf geführt und bei jeder Änderung inkrementiert. Bei Freigabe

* Die Angaben zur Pufferbelegung können auch für Anforderungen ungeänderter Seiten genutzt werden, die im Puffer eines der Rechner vorliegen, um E/A-Vorgänge zu umgehen.

der Schreibsperre für eine erfolgreich durchgeführte Änderung wird der aktuelle Wert der Versionsnummer dem GLM mitgeteilt und in der globalen Sperrtabelle gespeichert. Bei der Anforderung einer Sperre ist zunächst zu prüfen, ob eine Kopie der Seite lokal gepuffert ist und - wenn ja - mit welcher Versionsnummer. Der GLM kann dann bei der Sperrbearbeitung durch Zeitstempelvergleich feststellen, ob die Kopie invalidiert ist oder nicht, und das Ergebnis mit der Sperrgewährung mitteilen.

Abb. 15-5: On-Request-Invalidierung mit Versionsnummern



Beispiel 15-1

Im Beispiel von Abb. 15-5 ist die Seite B zunächst ungeändert im Hauptspeicher der beiden Rechnern R1 und R3 gepuffert. Nachdem der GLM einer Transaktion in R3 eine Schreibsperre gewährt hat, liegt in der globalen Sperrtabelle der in Abb. 15-5a gezeigte Sperreintrag für B vor. Es ist darin vermerkt, daß eine X-Sperre an Rechner R3 gewährt ist; das Feld für die Versionsnummer (VN) ist unbelegt, da noch keine Änderung erfolgte. Nach erfolgreicher Änderung der Seite in R3 wird dort die Versionsnummer von 17 auf 18 erhöht und dieser Wert mit Freigabe der X-Sperre dem GLM mitgeteilt, der die Versionsnummer in die globale Sperrtabelle übernimmt (Abb. 15-5b). Bei einer Sperranforderung von R1 wird dem GLM mitgeteilt, daß die lokale Kopie von B die Versionsnummer 17 aufweist. Der GLM erkennt damit die Pufferinvalidierung in R1 und teilt dies bei der Gewährung der X-Sperre R1 mit, woraufhin die Seite eliminiert wird (Abb. 15-5c). Die aktuelle

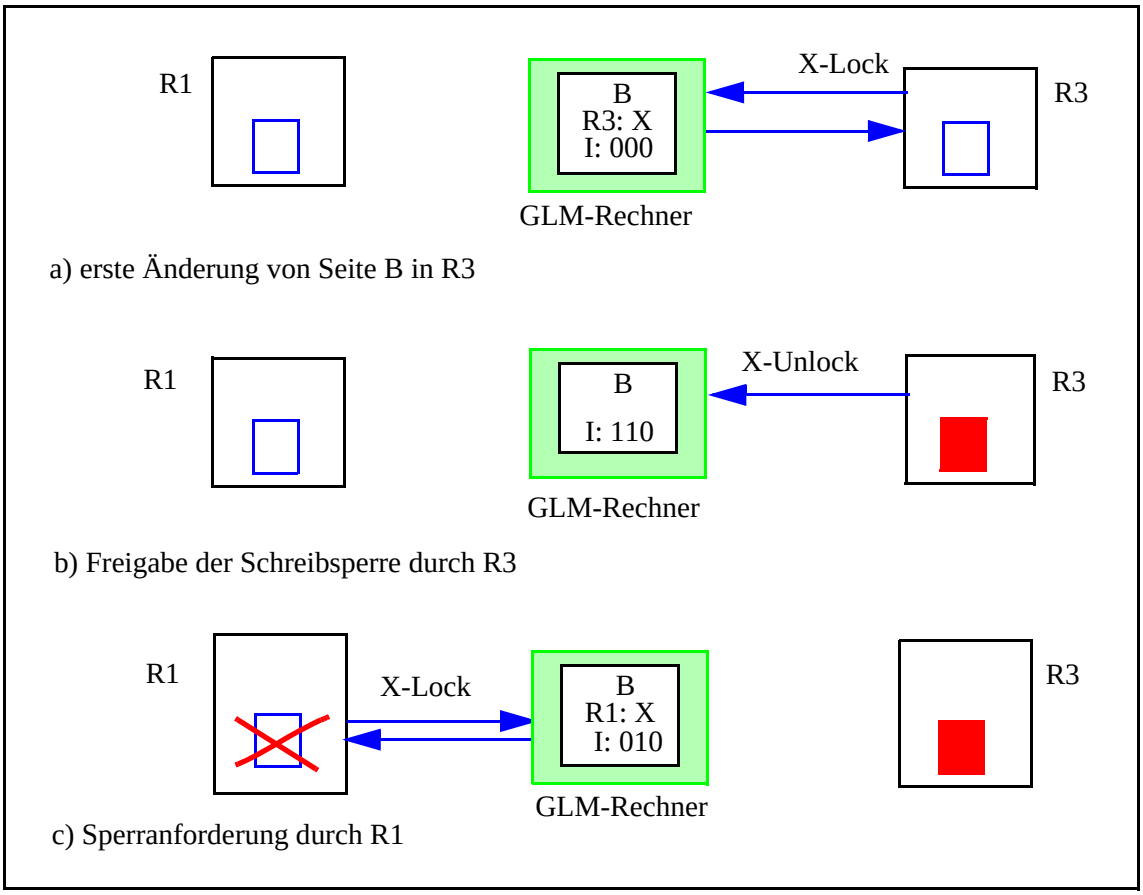
Version der Seite kann bei Force von Platte eingelesen werden, da hierbei R3 vor Freigabe der Schreibsperre die geänderte Seite ausschreiben mußte.

Die On-Request-Invalidierung kann alternativ mit sogenannten Invalidierungsvektoren realisiert werden [Ra86b]. Diese Invalidierungsvektoren werden für geänderte Seiten anstelle der Versionsnummern in der globalen Sperrtabelle geführt und bestehen aus je einem Invalidierungsbit pro Rechner. Das Invalidierungsbit für einen Rechner zeigt an, ob dort *möglicherweise* eine invalidierte Seite vorliegt. Der Wert I(j) =1 bedeutet dabei, daß für die betreffende Seite eine Invalidierung in Rechner j möglich ist, für I(j) = 0 ist sie dagegen ausgeschlossen. Diese Information kann geführt werden, da nach einer Seitenänderung zunächst nur der ändernde Rechner eine aktuelle Version der Seite hat. Für alle anderen Rechner, welche die ungeänderte Seitenversion gepuffert haben, liegt dagegen eine Invalidierung vor. Da der Aufwand der genauen Pufferbelegung nicht geführt werden soll, ist dem GLM nicht bekannt, welche Rechner tatsächlich Kopien der Seiten halten. Daher wird das Invalidierungsbit für alle Rechner außer dem ändernden gesetzt, um eine mögliche Invalidierung anzuzeigen. Dies ist ausreichend, da beim Anfordern einer Sperre geprüft wird, ob eine Kopie der Seite vorliegt. Ist dies der Fall und ist das Invalidierungsbit für den Rechner gesetzt, so liegt eine Invalidierung vor. Der Vorteil gegenüber der Verwendung von Versionsnummern liegt vor allem darin, daß in den Seiten selbst keine Invalidierungsangaben mehr zu führen sind. Dies macht diesen Ansatz auch besser geeignet zur Kohärenzkontrolle bezüglich feineren Objektgranularitäten als Seiten, z.B. zur Objektpufferung in objekt-orientierten DBS.

Beispiel 15-2

Abb. 15-6 zeigt die Bearbeitung der Sperranforderungen aus dem vorherigen Beispiel, wenn anstelle von Versionsnummern Invalidierungsvektoren eingesetzt werden. Man erkennt, daß die Versionsnummern bei den Seiten selbst nicht mehr erforderlich sind. Der Invalidierungsvektor I hat für die unterstellten drei Rechner den für ungeänderte Seiten gültigen Initialwert 000. Nach Änderung der Seite in R3 wird I nach 110 abgeändert, da zu diesem Zeitpunkt nur R3 eine gültige Version von B besitzt, eine etwaige Kopie der Seite in R1 und R2 jedoch veraltet ist. Bei der folgenden Sperranforderung von R1 wird aufgrund des gesetzten Invalidierungsbits für diesen Rechner die Invalidierung erkannt. Der GLM setzt das Invalidierungsbit für R1 zurück, wodurch I den Wert 010 annimmt, da R1 die invalidierte Seite entfernt und die aktuelle Version von B erhält (bei Force durch Einlesen von Platte).

Abb. 15-6: On-Request-Invalidierung mit Invalidierungsvektoren



Die in der globalen Sperrtabelle vorliegenden Angaben zur Kohärenzkontrolle sind im Falle der On-Request-Invalidierung auch dann zu führen, wenn für die betreffenden Seiten zeitweilig keine Sperranforderungen vorliegen. Damit stellt sich das Problem, die Größe der globalen Sperrtabelle zu begrenzen. Wenn in der globalen Sperrtabelle die genaue Pufferbelegung aller Rechner geführt würde, könnte ein Sperreintrag gelöscht werden, sobald die Seite in keinem der Rechner mehr gepuffert ist. Wird aus Aufwandsgründen auf die Wartung der genauen Pufferbelegung verzichtet, können in periodischen Abständen Sperreinträge gelöscht werden, auf die schon längere Zeit nicht mehr zugegriffen wurde. Damit dadurch keine relevanten Informationen verlorengehen, kann der GLM zuvor die betroffenen Seiten in einer Broadcast-Nachricht bekanntgeben, so daß die Rechner bei ihnen noch vorliegende, veraltete Kopien entfernen können. Diese Broadcast-Nachrichten verursachen nur einen geringen Aufwand und beeinflussen nicht die Antwortzeiten laufender Transaktionen.

15.3.2 Dynamische Page-Owner-Zuordnung

Bei einer Noforce-Ausschreibstrategie, die der Force-Alternative vorzuziehen ist, stellt sich wiederum die Frage der Update-Propagierung. Wie schon in Kap. 15.2.2 diskutiert, läßt sich bei dynamischer Page-Owner-Zuordnung die Page-Owner-Lokalisierung einfach und ohne zusätzliche Kommunikation in das Sperrprotokoll integrieren, indem der Page-Owner ebenfalls in der globalen Sperrtabelle geführt wird. Der GLM kann damit bei der Sperrgewährung den zuständigen Page-Owner mitteilen, von dem dann die aktuelle Version der Seite angefordert wird. Eine solche Seitenanforderung wird nur notwendig, wenn ein Page-Owner in der globalen Sperrtabelle vermerkt ist (d.h., die Seitenversion auf Platte nicht aktuell ist) und eine invalidierte oder keine Version der Seite im anfordernden Rechner gepuffert ist.

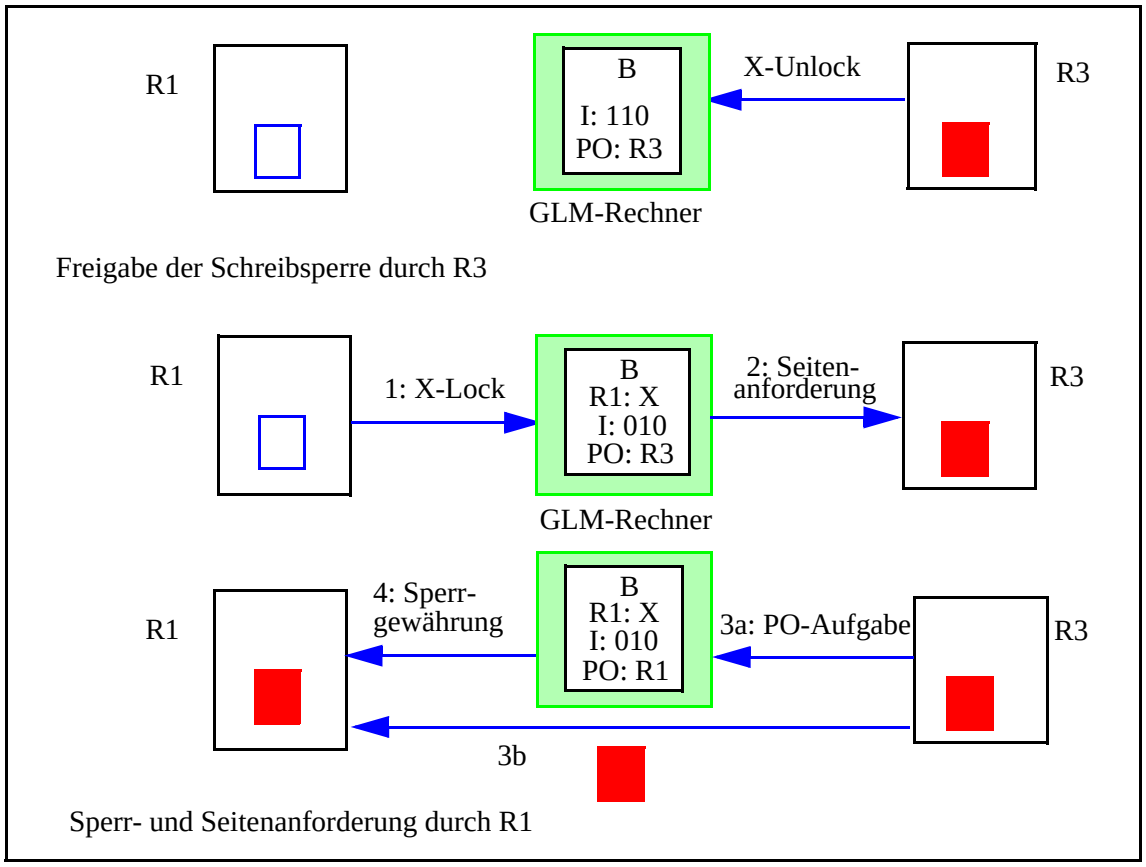
Die Seitenanforderung an den Page-Owner ist analog zum Lesen der Seite von Platte, kann jedoch bei direktem Seitentransfer deutlich schneller abgewickelt werden. Dennoch ist die zweifache Verzögerung der Transaktion zur Sperranforderung und der Seitenanforderung unbefriedigend, zumal damit i.d.R. 4 Nachrichten verbunden sind. Eine leichte Verbesserung wird erreicht, wenn der GLM die Seitenanforderung für die Transaktion an den Page-Owner weitergibt, da somit eine frühere Übertragung der Seite veranlaßt werden kann. Eine weitergehende Verbesserung wird bei fester Page-Owner-Zuordnung möglich (s.u.).

Die Änderung einer Seite in verschiedenen Rechnern bedingt bei dynamischer Page-Owner-Zuordnung eine Migration der Page-Owner-Funktion. Dies verlangt nicht nur Anpassungen in der globalen Sperrtabelle, sondern auch in den Verarbeitungsrechnern, da sie wissen müssen, für welche Seiten sie die Page-Owner-Funktion besitzen. Somit wird auch der explizite Entzug der Page-Owner-Funktion notwendig, bevor ein neuer Rechner als Page-Owner auftreten kann. Die Migration der Page-Owner-Funktion läßt sich vielfach mit dem Seitentransfer vom alten zum neuen Page-Owner kombinieren, so daß zusätzliche Verzögerungen weitgehend vermieden werden. Bei einem Austausch geänderter Seiten über Externspeicher wird die Page-Owner-Funktion mit dem Ausschreiben ohnehin hinfällig; erfolgt danach eine Änderung in einem anderen Rechner, wird dieser zum neuen Page-Owner. Beim direkten Austausch geänderter Seiten kann damit auch ein expliziter Transfer der Page-Owner-Funktion erfolgen. Da der Seitentransfer mit der Sperrgewährung verknüpft ist, sollte in der globalen Sperrtabelle in diesem Fall ein Rechner bereits bei der Gewährung einer Schreibsperre als Page-Owner eingetragen werden. Bei erstmaliger Änderung einer Seite im System genügt die Anpassung der Page-Owner-Angabe bei Freigabe der Schreibsperre.

Beispiel 15-3

Abb. 15-7 verdeutlicht die Funktionsweise der dynamischen Page-Owner-Zuordnung für das Beispiel aus Abb. 15-6, wobei eine On-Request-Invalidierung über Invalidierungsvektoren sowie ein direkter Austausch geänderter Seiten unterstellt ist. Der Page-Owner-Eintrag (PO) in der globalen Sperrtabelle ist für die ungeänderte Seite B zunächst unbelegt und wird für die erste Änderung bei der Freigabe der Schreibsperre auf R3 gesetzt. Bei der nachfolgenden Anforderung einer Schreibsperre durch R1 erkennt der GLM die Pufferinvalidierung aufgrund des gesetzten Invalidierungsbits. Da R3 als Page-Owner geführt ist, stellt der GLM direkt die Seitenanforderung an ihn (Nachricht 2); zugleich wird die Aufgabe der Page-Owner-Funktion verlangt, da in R1 eine Änderung der Seite beabsichtigt ist. Rechner R3 übeträgt daraufhin die Seite an R1 und signalisiert gleichzeitig die Aufgabe der Page-Owner-Funktion an den GLM. Der GLM trägt R1 als neuen Page-Owner ein und teilt dies R1 mit der Gewährung der Schreibsperre mit (Nachricht 4). Damit konnte die Page-Owner-Migration vollkommen mit dem Seitentransfer und der Sperrgewährung kombiniert werden.

Abb. 15-7: On-Request-Invalidierung mit dynamischer Page-Owner-Zuordnung



15.3.3 Feste Page-Owner-Zuordnung

Die feste Zuordnung der Page-Owner-Funktion zu einem oder mehreren Rechnern verlangt, daß eine Seitenänderung an den zuständigen Rechner übertragen wird, damit dieser anderen Rechnern die aktuellen Objektversionen zur Verfügung stellen kann. Dieser Mehraufwand gegenüber einer dynamischen Page-Owner-Zuordnung kann jedoch durch eine noch bessere Kombination von Synchronisation und

Kohärenzkontrolle kompensiert werden. Dies trifft für Sperrverfahren zu, bei denen die globale Sperrverantwortung fest unter den Verarbeitungsrechnern oder dedizierten GLM-Rechnern aufgeteilt wird. Wird nämlich für die Page-Owner-Zuordnung die gleiche DB-Aufteilung wie für die GLA-Zuordnung gewählt, lassen sich sämtliche Seitentransfers mit Sperrnachrichten kombinieren:

- Seitentransfers zum Page-Owner können mit der Nachricht zur Freigabe der Schreibsperre kombiniert werden!
- Seitentransfers vom Page-Owner zum anfordernden Rechner können mit der Nachricht zur Gewährung einer Sperre kombiniert werden!

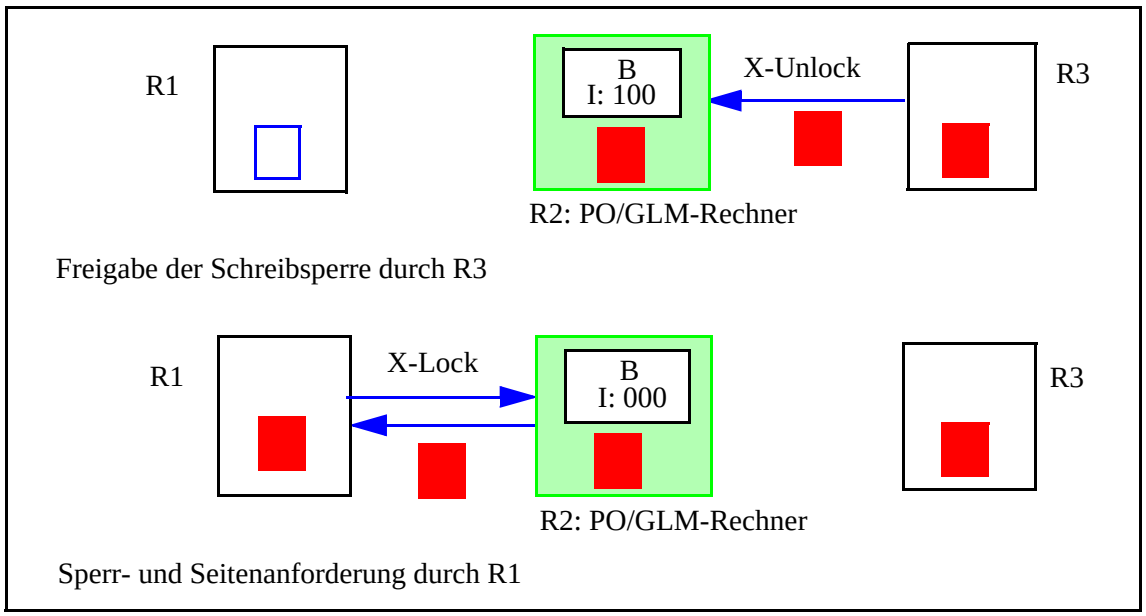
Besonders bedeutsam im Vergleich zu dynamischer Page-Owner-Zuordnung ist der zweite Punkt, da nun Seitenanforderungen keine zusätzlichen Verzögerungen mehr verursachen. Weiterhin sind in der globalen Sperrtabelle keine Angaben mehr zum aktuellen Page-Owner zu führen, und die mit der Migration des Page-Owners verbundenen Komplikationen entfallen.

Beispiel 15-4

Abb. 15-8 zeigt für unser Beispiel die Propagierung geänderter Seiten bei einer solchen auf die GLA-Verteilung abgestimmten, festen Page-Owner-Zuordnung. Dabei sei Rechner R2 sowohl GLM als auch Page-Owner für Seite B. Damit wird mit der Freigabe der Schreibsperre durch R3 die geänderte Seite an R2 übergeben und dort im Puffer aufgenommen. Dies hat auch Auswirkungen auf die Verwendung von Invalidierungsvektoren zur On-Request-Invalidierung. Wie gezeigt, wird jetzt I auf den Wert 100 gesetzt, da in R2 durch die Übernahme der aktuellen Seitenversion keine Invalidierung für B möglich ist. Bei der Sperranforderung von R1 wird die Invalidierung erkannt und die aktuelle Seite direkt mit der Sperrgewährung zurückgeliefert. Ein Vergleich mit Abb. 15-7 zeigt, daß zwei Nachrichten eingespart werden und die mit der Sperranforderung zusammenfallende Seitenanforderung wesentlich schneller bearbeitet wird. Dafür ist der Aufwand der Seitenübertragung bei Freigabe der Schreibsperre in Kauf zu nehmen.

Abb. 15-8: On-Request-Invalidierung mit fester Page-Owner-Zuordnung

Bezogen auf die Anzahl zur Kohärenzkontrolle benötigter Nachrichten ist das Verfahren optimal, da weder zur Erkennung von Pufferinvalidierungen noch zur Update-Propagierung zusätzliche Nachrichten anfallen. Weiterhin können Seitenanforderungen mit minimalem Aufwand bearbeitet werden. Demgegenüber verursachen die Seitenübertragungen zum Page-Owner einen Zusatzaufwand, da natürlich die Übertragung einer "großen" Nachricht einen höheren Kommunikationsaufwand verursacht als eine einfache Nachricht zur Sperrfreigabe. Diese Merkmale gelten sowohl für eine Page-Owner/GLA-Aufteilung unter den Verabeitungsrechnern als auch bei dedizierten Rechnern. In letzterem Fall verursachen die Seitentransfers jedoch ein besonders hohes Übertragungsvolumen. Denn dabei muß jede Seitenänderung zum Page-Owner/GLM-Rechner übertragen werden, obwohl sie dort für die Transaktionsverarbeitung nicht genutzt werden. Der Auf-



wand hierfür ist ähnlich dem einer Force-Strategie, wo auch alle Seitenänderungen am Transaktionsende zu propagieren sind (allerdings auf Platte). Die ständige Aufnahme neuer Seitenänderungen verlangt weiterhin ein ständiges Ausschreiben zuvor gepufferter Seiten und kann dazu führen, daß nur relativ wenige Seiten vom Page-Owner/GLM-Rechner direkt mit der Sperrgewährung an den anfordernden Rechner bereitgestellt werden können. Dies gilt natürlich vor allem bei nur einem Page-Owner/GLM-Rechner (zentraler Page-Owner), der zudem leicht zum Systemengpaß wird.

Günstiger ist dagegen die kombinierte Page-Owner/GLA-Verteilung unter allen Verarbeitungsrechnern, da sich dann Lokalität im Referenzverhalten auch zur Einsparung von Seitentransfers nutzen läßt. Dies trifft vor allem für das *Primary-Copy-Sperrverfahren* (Kap. 14.3) zu, bei dem eine logische DB-Aufteilung zur GLA-Allokation verwendet wird. Durch ein darauf abgestimmtes affinitätsbasiertes Transaktions-Routing kann damit rechnerspezifische Lokalität direkt zur Einsparung von globalen Sperranforderungen genutzt werden. Wird nun dieselbe DB-Partitionierung zur Page-Owner-Zuordnung verwendet, erfolgen idealerweise die meisten Zugriffe auf eine DB-Partition bereits am zuständigen GLM/Page-Owner-Rechner. Für diese Zugriffe entfallen somit die Seitenanforderungen, da der eigene Rechner als zuständiger Page-Owner bereits die aktuelle Seite hat oder diese von der physischen DB gelesen werden kann*. Für Änderungen, die am zuständigen Rechner erfolgen, entfallen ebenso die Seitentransfers bei Freigabe der Schreibsperre. Ein weiterer Pluspunkt ist, daß die Anzahl der Pufferinvalidierungen re-

* Die Bezeichnung "Primary Copy" ist nunmehr berechtigt, da der GLM-Rechner sämtliche Primärkopien seiner DB-Partition führt und verwaltet.

duziert wird, da diese nur noch für Seiten möglich sind, die zur Partition eines anderen Rechners gehören.

15.4 Einsatz von Haltesperren

Ein Buffer-Purge-Ansatz vermeidet in Verbindung mit Sperrverfahren Pufferinvalidierungen, da gepufferte Seiten am Transaktionsende (vor Freigabe der Sperre) aus dem Puffer eliminiert werden. Das E/A-Verhalten eines solchen Ansatzes ist jedoch nicht akzeptabel, da er eine Force-Strategie impliziert (hoher Schreibaufwand) und die Puffer nur schlecht zur Einsparung lesender Plattenzugriffe genutzt werden. Ein besserer Vermeidungsansatz ist dagegen, Seiten auch nach Transaktionsende im Puffer zu belassen, sie jedoch durch spezielle Haltesperren (retention locks) vor der Invalidierung durch andere Rechner zu schützen [HR85, Ra88b, CFLS91, DY92]. Somit muß für jede gepufferte Seite entweder eine reguläre Transaktionssperre oder eine Haltesperre bestehen. Eine durch eine Haltesperre geschützte Seite kann nicht unbemerkt in einem anderen Rechner geändert werden, da die hierfür erforderliche Schreibsperre mit der in der globalen Sperrtabelle vermerkten Haltesperre in Konflikt gerät. Der Konflikt führt dazu, daß der GLM die Aufgabe der Haltesperre verlangt. Durch Eliminieren der Seite vor Freigabe der Haltesperre wird die Invalidierung der Seite verhindert. Damit werden von Invalidierungen betroffene Seiten noch früher aus dem Puffer entfernt als bei einer Broadcast-Invalidierung.

Das Konzept der Haltesperren läßt sich vorteilhaft mit der Verwendung von Lese- und Schreibautorisierungen (Kap. 14.2.1) verknüpfen. Denn diese Autorisierungen werden für gesperrte Objekte auch nach Transaktionsende vom jeweiligen Rechner beibehalten, um Lokalität im Referenzverhalten zur lokalen Sperrvergabe zu nutzen. Die Autorisierungen sind ebenfalls beim GLM vermerkt und müssen bei einem Konflikt mit anderen Rechnern explizit zurückgenommen werden. Da Haltesperren sich auf Seiten beziehen, können diese nun durch Schreib- und Leseautorisierungen auf Seitenebene realisiert werden. Daraus resultieren zwei Arten von Haltesperren, die neben der Vermeidung von Pufferinvalidierungen auch die lokale Vergabe und Freigabe von Seitensperren erlauben:

- Die *WA-Haltesperre* (write authorization) bezieht sich auf geänderte Seiten im Puffer. Sie kann nur in einem Rechner vorliegen und garantiert, daß kein anderer Rechner die Seite gepuffert hat bzw. Zugriffe auf die Seite vornehmen kann. Sie gewährleistet die Gültigkeit einer gepufferten Seitenkopie und erlaubt eine lokale Vergabe/Freigabe von Lese- und Schreibsperren.
- Die *RA-Haltesperre* (read authorization) kann gleichzeitig in mehreren Rechnern vorliegen und garantiert, daß kein anderer Rechner die Seite ändert bzw. eine WA-Haltesperre besitzt. Sie garantiert die Aktualität einer gepufferten Seitenkopie und gestattet die lokale Behandlung von Lesesperren.

Zur Update-Propagierung beschränken wir unsere Diskussion auf den kompliziertesten Fall, nämlich Noforce mit dynamischer Page-Owner-Zuordnung. Dabei hängt das Zusammenspiel zwischen Page-Owner sowie Haltesperren davon ab, ob geänderte Seiten über Externspeicher oder direkt ausgetauscht werden. Beide Fälle sollen im folgenden genauer betrachtet werden.

Austausch geänderter Seiten über die gemeinsamen Platten

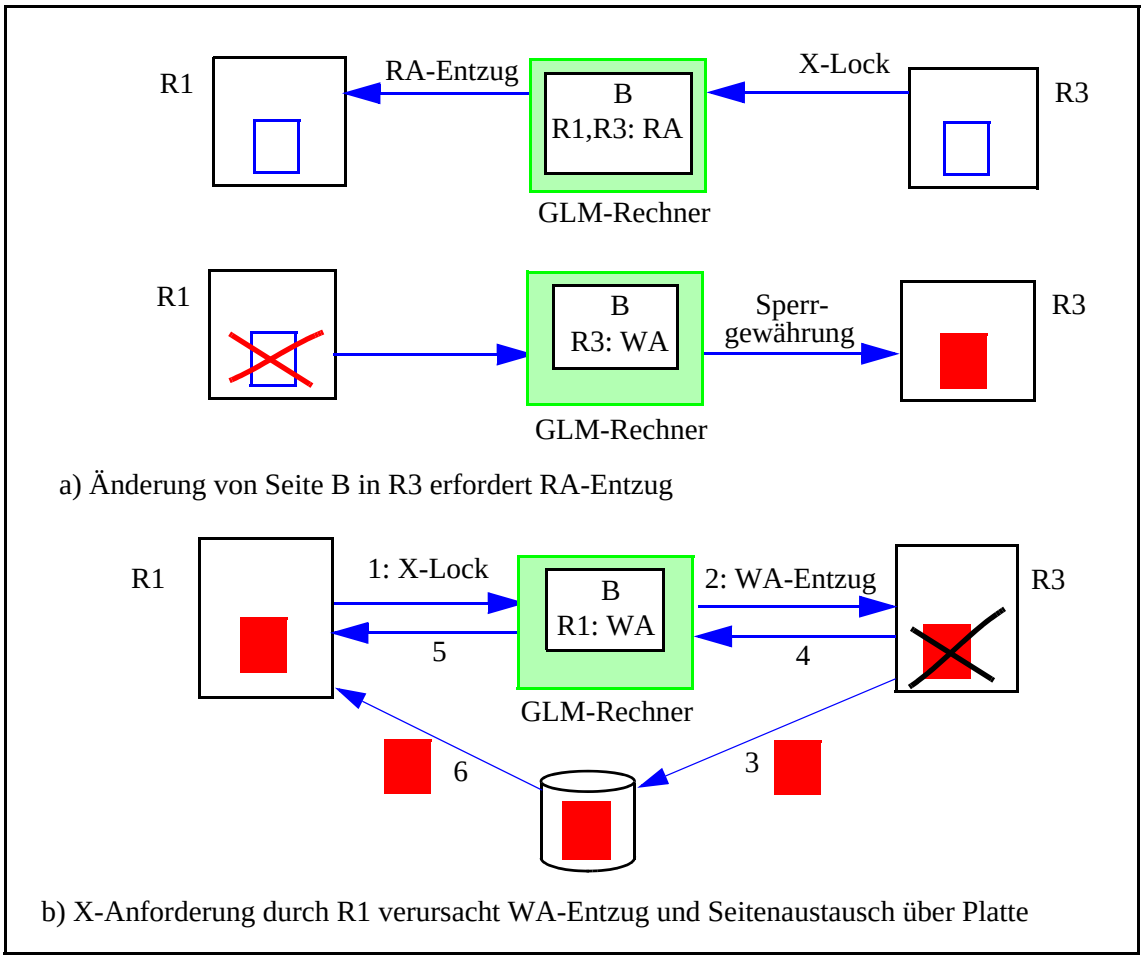
Bei diesem Ansatz, der im Shared-Disk-System von Oracle verwendet wird, entspricht der Page-Owner dem Besitzer einer WA-Haltesperre. Der GLM braucht daher in der globalen Sperrtabelle neben den Haltesperren den Page-Owner nicht zusätzlich zu führen. Aufgrund der Semantik der WA-Haltesperre kann eine geänderte Seite nur in einem Rechner gepuffert sein, nämlich beim Page-Owner, wo die Änderung stattfand. Ein Seitenzugriff in einem anderen Rechner führt dazu, daß der GLM die Aufgabe der WA-Haltesperre sowie einen Seitentransfer über Platte verlangt. Der Page-Owner schreibt daraufhin die Seite aus und gibt danach die WA-Sperre sowie die Page-Owner-Funktion auf. Im Rechner, der den WA-Entzug veranlaßt hat, kann die aktuelle Seite dann von Platte eingelesen werden. Ein Vorteil liegt darin, daß Seitenanforderungen stets mit dem Entzug einer WA-Haltesperre zusammenfallen und somit keine zusätzlichen Nachrichten verursachen.

Eine RA-Haltesperre ist bei diesem Ansatz nur möglich, wenn die Seite in keinem der Rechner geändert vorliegt. Damit garantiert die RA-Haltesperre nicht nur die Aktualität der gepufferten Seitenkopie, sondern auch der Seitenversion in der physischen DB.

Beispiel 15-5

Das zur Illustrierung der On-Request-Invalidierung verwendete Szenario soll nun auch zur Verdeutlichung des Haltesperreneinsatzes dienen (Abb. 15-9). Zunächst lag eine ungeänderte Kopie der Seite B in den Rechnern R1 und R3 vor, was jetzt erfordert, daß diese Rechner eine RA-Haltesperre für die Seite halten. Die Durchführung einer Seitenänderung in R3 verlangt daher zunächst den Entzug der RA-Haltesperre in R1, wobei die Seite vor Rückgabe der Haltesperre aus dem Puffer eliminiert wird, um ihre Invalidierung zu vermeiden (Abb. 15-9a). R3 erhält vom GLM eine Schreibautorisierung/WA-Haltesperre und wird damit auch zum Page-Owner für B. Die Freigabe der Schreibsperre ist eine lokale Aktion in R3. Die Anforderung einer Schreibsperre in R1 führt zum Entzug der WA-Haltesperre in R3, der zugleich mit einer Seitenanforderung verbunden ist. Nach Ausschreiben der Seite eliminiert R3 die Seite aus dem Puffer (um ihre Invalidierung durch R1 zu vermeiden) und gibt die WA-Haltesperre zurück. Der GLM vergibt eine WA-Haltesperre und damit die Page-Owner-Funktion an R1, der jedoch vor der Durchführung seiner Änderung zunächst die Seite von Platte einlesen muß (Abb. 15-9b).

Abb. 15-9: Haltesperreneinsatz bei Austausch geänderter Seiten über Platte

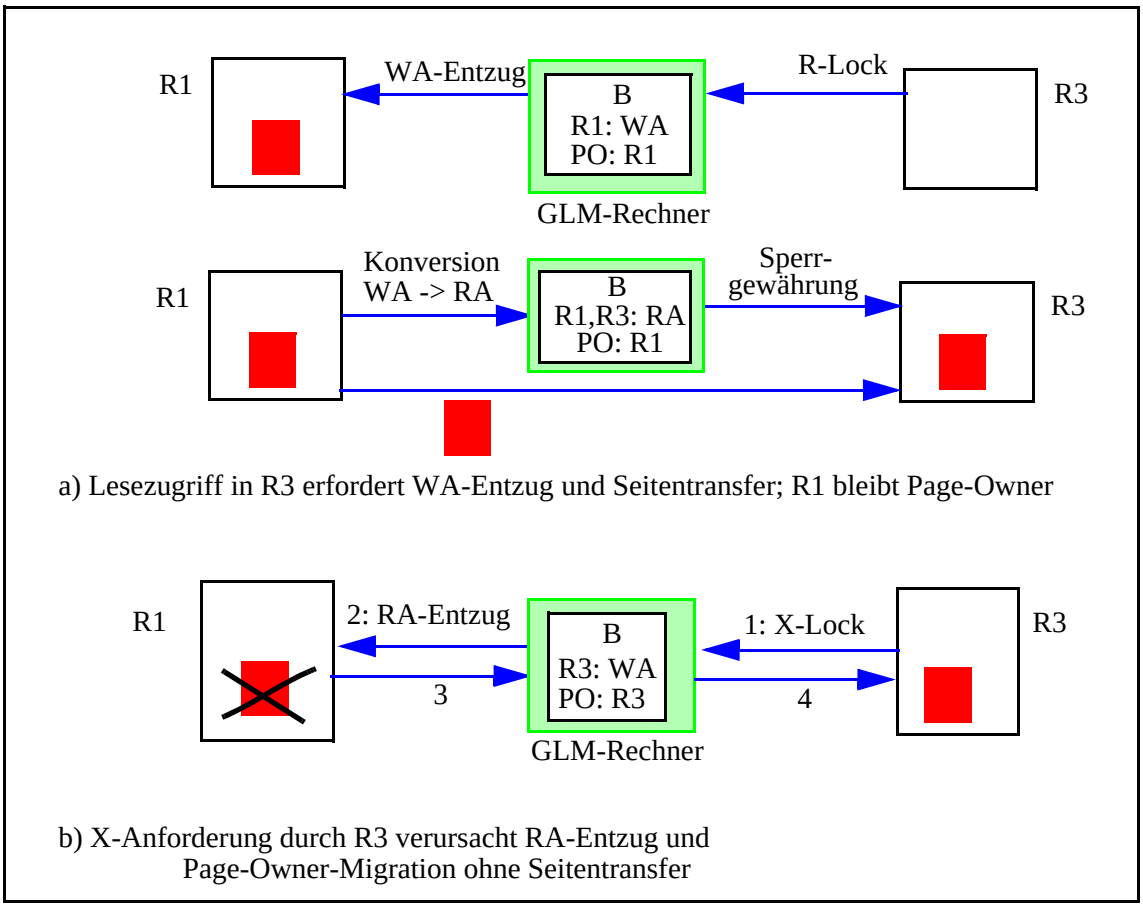


Direkter Austausch geänderter Seiten

Der Hauptunterschied zum vorhergehenden Fall liegt darin, daß beim Entzug einer WA-Haltesperre die Seite vom Page-Owner nicht ausgeschrieben wird. Stattdessen wird die Seite direkt zum anfordernden Rechner übertragen, was eine wesentliche Beschleunigung gegenüber dem Seitenaustausch über Platte ergibt (Einsparung von zwei Plattenzugriffen). Im Beispiel von Abb. 15-9b kann so der Seitentransfer von R3 nach R1 parallel zu den Nachrichten zur Mitteilung des WA-Entzugs sowie der Sperrgewährung erfolgen.

Eine Folge des direkten Seitenaustauschs ist, daß eine geänderte Seite im System vorliegen kann, auch wenn keine WA-Haltesperre vergeben ist. Dies ist etwa beim Entzug der WA-Haltesperre aufgrund eines Lesezugriffs der Fall. So kann für eine geänderte Seite am Page-Owner-Rechner auch nur eine RA-Haltesperre vorliegen, während beim Seitenaustausch über Platte eine RA-Haltesperre nur für ungeänderte Seiten möglich ist. Diese Änderungen machen es notwendig, in der globalen Sperrtabelle den Page-Owner explizit zu führen, unabhängig von der Vergabe der Haltesperren.

Abb. 15-10: Haltesperreneinsatz bei direkten Seitentransfers .



Beispiel 15-6

In Fortsetzung des vorherigen Beispiels liegt in der Ausgangssituation von Abb. 15-10 Seite B geändert in R1 vor; R1 ist Page-Owner und besitzt eine WA-Haltesperre. Eine Leseanforderung von R3 bewirkt nun einen direkten Seitentransfer von R1 nach R3. R1 muß die WA-Haltesperre aufgeben, jedoch ist eine Konversion in eine RA-Haltesperre möglich, da B in R3 nur gelesen wird. Daher behält R1 auch die Page-Owner-Funktion, und die Seite bleibt in R1 gepuffert. Wenn anschließend in R3 eine Änderung der Seite vorgenommen werden soll (Abb. 15-10b), ist zuvor ein Entzug der RA-Haltesperre in R1 notwendig, wobei die Seite auch aus dem Puffer entfernt wird. Die Page-Owner-Funktion geht nach R3 über. In diesem Fall ist mit der Page-Owner-Migration kein Seitentransfer verbunden, da R3 bereits die aktuelle Version der Seite vorliegen hat.

Die Beispiele zeigen zur Illustrierung der Funktionsweise Worst-Case-Szenarien im Hinblick auf den Nachrichtenbedarf, während bei entsprechender Lokalität im Referenzverhalten jedoch viele Nachrichten entfallen. Bei den Haltesperren ist zudem zu beachten, daß nur wenige Nachrichten über die zur Synchronisation mit Lese/Schreibautorisierungen benötigten hinaus erforderlich sind. Die zur Vermeidung der Pufferinvalidierungen erforderliche vorsorgliche Eliminierung von ge-

pufferten Seiten fällt stets zusammen mit dem Entzug einer Lese/Schreibautorisierung. Auch fallen der Entzug der Page-Owner-Funktion sowie Seitenanforderungen meist mit dem Entzug von Autorisierungen/Haltesperren zusammen*. Um das Ausmaß der aufwendigen Entzugsvorgänge zu reduzieren ist eine freiwillige Rückgabe einer Haltesperre angebracht, sobald die Seite aus dem Puffer verdrängt wird. Denn dies zeigt an, daß die Seite schon längere Zeit nicht mehr referenziert wurde. Bei direktem Seitenaustausch kann zudem eine RA-Haltesperre ohnehin nicht mehr zur lokalen Sperrbehandlung genutzt werden, nachdem die Seite verdrängt wurde. Denn in diesem Fall ist die Gültigkeit der Seite in der physischen DB nicht gewährleistet, so daß der GLM-Rechner ohnehin zu befragen ist, ob die Seite von einem anderen Rechner anzufordern ist.

15.5 Unterstützung von Satzsperren

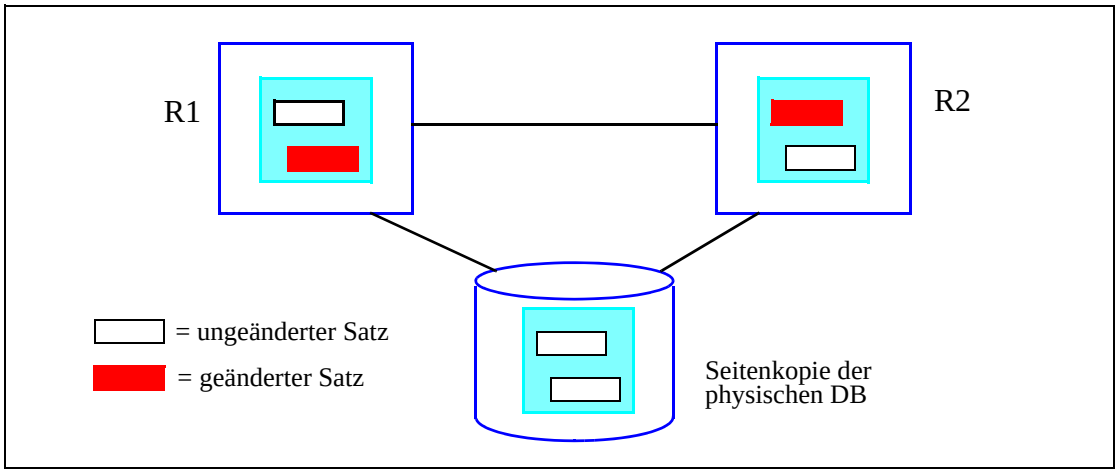
Bei der bisherigen Beschreibung wurde stets eine Synchronisation auf Seitenebene unterstellt. Wenngleich dies für viele Anwendungen ausreichen dürfte, kann das Ausmaß an Synchronisationskonflikten damit nicht generell niedrig genug gehalten werden. Viele DBS unterstützen daher Satzsperren sowie andere Maßnahmen zur Reduzierung der Konfliktrate.

Der Einsatz von Satzsperren in Shared-Disk-Systemen wirft jedoch besondere Kohärenzprobleme auf. Wie in Abb. 15-11 illustriert, können dann nämlich verschiedene Sätze derselben Seite in verschiedenen Rechnern geändert werden. Damit ist die Kopie der Seite an jedem der Rechner nur partiell aktuell; die Version der Seite in der physischen Datenbank enthält zunächst keine der Änderungen. Das Ausschreiben der partiell aktuellen Seiten ist nicht zulässig, da dies zum Verlust gültiger Änderungen führen könnte. Folglich muß eine Seite vollständig aktualisiert werden, bevor ein Ausschreiben erfolgt. Ein "Mischen" von Änderungen verschiedener Rechner ist jedoch nicht nur aufwendig, sondern oft gar nicht möglich, da Satzänderungen vielfach Änderungen in der Seitenstruktur verursachen (Löschen von Sätzen, Änderungen in der Satzlänge etc.).

---

* Dies ist bei Austausch geänderter Seiten über Platte stets der Fall. Bei direktem Seitenaustausch sind separate Seitenanforderungen nur für Lesezugriffe erforderlich, wenn der Page-Owner lediglich eine RA-Haltesperre hält.

Abb. 15-11: Kohärenzproblem beim Einsatz von Satzsperren



Die Probleme werden i.d.R. durch die Unterstützung beschränkter Formen von Satzsperren umgangen. Die einfachste Möglichkeit dazu besteht darin, Satzsperren im Rahmen eines hierarchischen Verfahrens nur innerhalb der Verarbeitungsrechner zu unterstützen, jedoch globale Sperranforderungen auf Seitenebene (oder gröberen Granulaten) zu stellen. Der Vorteil liegt darin, daß die Protokolle zur globalen Sperrverwaltung und Kohärenzkontrolle ungeändert bleiben können. Zudem wird vermieden, daß die Anzahl globaler Sperranforderungen zunimmt, wie bei der Verwendung eines feineren globalen Sperrgranulats zu erwarten. Dennoch wird durch die lokale Verwendung von Satzsperren bereits eine reduzierte Konflikthäufigkeit unterstützt. Dies gilt insbesondere im Zusammenhang mit einer affinitätsbasierten Transaktionsverteilung, mit der ein hohes Maß rechnerspezifischer Lokalität angestrebt wird. Denn in diesem Fall sind vor allem lokale Konflikte zu erwarten, denen bereits mit lokalen Satzsperren wirksam begegnet werden kann.

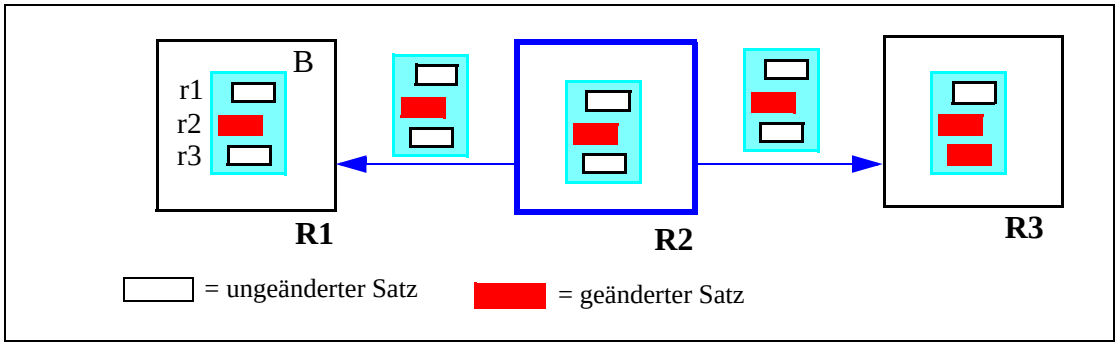
Eine weitere Reduzierung der Konflikthäufigkeit wird mit globalen Satzsperren erreicht, wobei jedoch zu einem Zeitpunkt Änderungen einer Seite stets auf einen Rechner beschränkt werden. Ein solches Verfahren wurde in [MN91] beschrieben. Neben transaktionsspezifischen Satzsperren muß dabei für Änderungen an dem betreffenden Rechner eine spezielle Update (U)-Sperre für die Seite vorliegen. Der Besitzer der U-Sperre fungiert zugleich als Page-Owner für die Seite. U-Sperren sind mit Lesesperren kompatibel, so daß eine geänderte Seite in einem anderen Rechner gepuffert und lesend bearbeitet werden kann (die Satzsperren garantieren dabei, daß unterschiedliche Sätze referenziert werden). Verschiedene Sätze einer Seite können sogar von mehreren parallelen Transaktionen an verschiedenen Rechnern geändert werden, wobei die Änderungen jedoch über die U-Sperre serialisiert werden. Die U-Sperre (Page-Owner-Funktion) kann dabei vor Aufgabe der

Satzsperre einer Transaktion mit der geänderten Seite selbst an einen anderen Rechner weitergegeben werden.

Beispiel 15-7

Abb. 15-12 zeigt ein Szenario zu diesem Protokoll, in dem drei parallele Transaktionen in den Rechnern R1, R2 und R3 verschiedene Sätze derselben Seite B bearbeiten. Zunächst wird Satz r2 in R2 geändert, wofür eine Satzsperre für r2 sowie eine U-Sperre für B beim (nicht gezeigten) GLM anzufordern ist. Danach wird eine Leseanforderung in R1 für Satz r1 gewährt, wobei die zu diesem Zeitpunkt beim Page-Owner R2 vorliegende Version von B an R1 übertragen wird. Eine Änderung von Satz r3 in R3 wird ebenfalls zugelassen; die Anforderung der benötigten U-Sperre führt jedoch zu einem Konflikt mit R2 und resultiert in der Übertragung der Page-Owner-Funktion (U-Sperre) sowie der Seite an R3. Dort liegt nun die aktuelle Version der Seite vor, wobei aufgrund der Serialisierung der Schreibvorgänge kein explizites Mischen der Änderungen erforderlich war.

Abb. 15-12: Serialisierung von Seitenänderungen bei Satzsperren



Es ist zu beachten, daß die notwendige Erkennung von Pufferinvalidierungen auf Satz- oder Seitenebene erfolgen kann, etwa über ein On-Request-Invalidierungsverfahren. Die Erkennung auf Seitenebene verursacht einen geringeren Verwaltungsaufwand beim GLM, führt jedoch zu vermehrten Seitentransfers. So wird damit bei einem Zugriff auf Satz r2 in Rechner R1 eine Pufferinvalidierung festgestellt und die aktuelle Seite beim Page-Owner R3 angefordert, obwohl die in R1 vorliegende Seitenkopie den aktuellen Satz enthält.

Das skizzierte Verfahren erlaubt einen hohen Parallelitätsgrad, jedoch führen die globalen Satzsperren i.a. zu einem weit höheren Kommunikationsaufwand als mit Seitensperren. Dies wird dadurch verschärft, da die U-Sperre im Gegensatz zu Haltesperren nicht zur lokalen Sperrvergabe genutzt werden kann (da sie mit Lesesperren anderer Rechner verträglich ist). Der Transfer von Änderungen nicht beendeter Transaktionen hat auch Recovery-Implikationen, da bei einem Transaktionsabbruch die Undo-Recovery erfordern kann, bereits migrierte Seiten an den Transaktionsrechner zurückzuholen, um die Log-Daten anwenden zu können.

Ein unbeschränktes Sperren auf Satzebene ohne Serialisierung von Änderungen ist möglich für Änderungen, welche die Struktur der Seite unverändert lassen (z.B. Änderung bestehender Sätze oder Einträge) [Ra88b, MNS91]. In diesem Fall ist ein explizites Mischen der Änderungen notwendig, um die aktuelle Version der Seite zu erhalten. Das parallele Ändern einer Seite in verschiedenen Rechnern be-

dingt, daß ein dynamischer Page-Owner-Ansatz nicht mehr anwendbar ist, so daß eine feste Zuordnung der Page-Owner-Funktion erforderlich wird. Jede Satzänderung ist somit an den Page-Owner zu übertragen, so daß er stets die neueste Seitenversion besitzt. Wird der Page-Owner auch zur globalen Sperrbehandlung herangezogen, so können die Änderungen wiederum mit der Sperrfreigabe übergeben werden. Damit erfordert das Mischen keinen zusätzlichen Kommunikationsaufwand. Vorteilhaft ist ferner, daß nicht ganze Seiten, sondern nur Sätze zu übertragen sind, wodurch sich eine deutliche Senkung des Übertragungsumfanges gegenüber Seitensperren erreichen läßt.

Eine noch weitergehende Reduzierung der Konfliktgefahr als mit Satzsperren wird möglich durch spezielle Synchronisationsverfahren, welche die Semantik bestimmter Änderungsoperationen ausnutzen (z.B. Kompatibilität von Inkrement-/Dekrement-Änderungen auf numerischen Attributen) [ON86]. Die Übertragung solcher Verfahren auf Shared-Disk-Systeme wird in [Hä88b, MNS91] diskutiert.

15.6 Zusammenfassende Bewertung

Die Beschreibung der Verfahren zur Kohärenzkontrolle zeigte das Zusammenspiel zwischen Erkennung/Vermeidung von Pufferinvalidierungen, Propagierung von Änderungen sowie der Synchronisation. Eine enge Abstimmung dieser Teilaufgaben im Rahmen eines integrierten Ansatzes erlaubt zudem erhebliche Einsparungen im Kommunikationsumfang. Zur Illustration des Lösungsspektrums sind in Abb. 15-13 noch einmal die wichtigsten Alternativen für die drei Teilprobleme aufgeführt. Da die meisten der Verfahren miteinander kombiniert werden können, ergeben sich über 100 verschiedene Realisierungsansätze. Die Anzahl erhöht sich weiter, wenn man bei den einzelnen Verfahrensklassen noch genauere Unterscheidungen vornimmt (z.B. Verwendung von Lese- und/oder Schreibautorisierungen, Art der Validierung, synchrone/asynchrone Broadcast-Invalidierung etc.). Die Aufstellung kann auch dazu dienen, bestehende Implementierungen bzw. Realisierungsvorschläge einzuordnen und somit besser zu bewerten. Dabei ist zur vollständigen Charakterisierung die Spezifikation aller drei Teilstrategien wesentlich. IMS Data Sharing verwendet z.B. die Verfahren S5/P1/U1 (Token-Ring-Sperrverfahren / Broadcast-Invalidierung / Force), während Oracle die Kombination S2/P4/U4 verfolgt.

Die Vielzahl an Kombinationsmöglichkeiten verlangt eine vergleichende Bewertung, um zu den vielversprechendsten Ansätzen zu kommen. Diese wurde in den zurückliegenden Kapiteln zum Teil bereits vorgenommen, soll jedoch hier noch einmal zusammengefaßt werden. Die Einschätzungen basieren zum Teil auf quantitativen Leistungsanalysen, die in großer Zahl für Shared-Disk-Systeme durchge-

führt wurden (u.a. [HR85, YCDI87, Yu87, Bh88, Ra88a, DIRY89, DDY91, DY92, DY93, Ra93c, Ra93d, YD94]).

Abb. 15-13: Verfahrensspektrum zur Synchronisation und Kohärenzkontrolle in Shared-Disk-Systemen

Synchronisation

- S1: Zentrales Sperrprotokoll
- S2: Verteiltes, dediziertes Sperrprotokoll
- S3: Feste GLA-Zuordnung, nicht dediziert (Primary-Copy-Sperrverfahren)
- S4: Dynamische GLA-Zuordnung
- S5: Token-Ring-Sperrprotokolle
- S6: Optimistisch: zentrale Validierung
- S7: Optimistisch: verteilte Validierung

Behandlung von Pufferinvalidierungen

- P1: Broadcast-Invalidierung
- P2: Multicast-Invalidierung
- P3: On-Request-Invalidierung
- P4: Haltesperren

Update-Propagierung

- U1: Force
- U2: Dedizierte(r) Page-Owner (PO)
- U3: Feste PO-Zuordnung, nicht dediziert
- U4: Dynamische PO-Zuordnung / Austausch geänderter Seiten über Externspeicher
- U5: Dynamische PO-Zuordnung / direkter Seitentransfer

Die erste Festlegung betrifft die Wahl zwischen optimistischer Synchronisation und Sperrverfahren. Optimistische Verfahren (Kap. 14.6) erlauben eine Synchronisation mit geringem Kommunikationsaufwand, insbesondere bei zentraler Validierung, der zudem weitgehend unabhängig von den Lastmerkmalen und der Strategie zur Lastverteilung ist. Die Kohärenzkontrolle dagegen ist weniger effektiv lösbar als für Sperrverfahren, da zur Erkennung von Pufferinvalidierungen nur eine (asynchrone) Broadcast-Invalidierung anwendbar ist. Diese führt jedoch einen sehr hohen Aufwand ein und eignet sich nicht für Konfigurationen mit gro-

ßer Rechneranzahl. Nachteilig ist ferner die hohe Anzahl von Transaktionsrücksetzungen, auch wenn ein Verhungern von Transaktionen durch Kombination mit Sperrverfahren i.a. verhindert werden kann. Sperrverfahren eignen sich schließlich besser zur Unterstützung feiner Synchronisationsgranulate (Satzsperren) und werden in kommerziellen DBS nahezu ausschließlich verwendet.

Von den Sperrverfahren können Token-Ring-Ansätze ausgeschlossen werden, da sie nur für wenige Rechner in Betracht kommen. Damit bleiben zur Synchronisation im wesentlichen die Alternativen S1 bis S4, wobei S1 noch als Spezialfall von S2 aufgefaßt werden kann. Diese Sperrverfahren basieren alle auf der Nutzung Globaler Lock-Manager (GLM), was eine effiziente Integration der Kohärenzkontrolle erlaubt. Dabei sind die Ansätze der Broadcast- und Multicast-Invalidierung aufgrund ihres hohen Kommunikationsbedarfs sowie starker Antwortzeitverschlechterung für Änderungstransaktionen als nicht empfehlenswert einzustufen (ebenso ein Buffer-Purge-Ansatz). Zur Behandlung von Pufferinvalidierungen eignen sich somit primär die Ansätze der On-Request-Invalidierung sowie der Haltesperren. Bezüglich der Update-Propagierung sind Noforce-Ansätze mit direkten Seitentransfers am effizientesten, wobei jedoch dedizierte Page-Owner oft eine hohe Anzahl von Seitenübertragungen verursachen. Am aussichtsreichsten erscheinen daher die Alternativen U3 und U5. Der Austausch geänderter Seiten über Externspeicher (sowie Force) vereinfachen dagegen die Crash-Recovery. Ihr Leistungsverhalten läßt sich zudem durch nicht-flüchtige Platten-Caches stark verbessern.

Der Einsatz von Haltesperren kommt vor allem für Sperrverfahren in Betracht, die Lese- und Schreibautorisierungen verwenden. Weil der Einsatz von Schreibautorisierungen der Nutzung einer lokalen GLA entgegenläuft (Kap. 14.3), sind Haltesperren dabei vor allem für dedizierte Sperrverfahren von Interesse (S1/S2). In diesem Fall kommt vor allem eine dynamische Page-Owner-Zuordnung in Frage, da sich die dedizierten GLM-Rechner weniger als Page-Owner eignen. Bei fester GLA-Zuordnung unter den Verarbeitungsrechnern (Primary-Copy-Sperrverfahren) ist eine sehr effiziente Kohärenzkontrolle möglich mit einer On-Request-Invalidierung sowie mit einer festen Page-Owner-Zuordnung, die mit der GLA-Verteilung übereinstimmt (Kap. 15.3.3). Bei dynamischer GLA-Zuordnung kann der Einsatz von Schreibautorisierungen die Nutzung einer lokalen GLA ebenfalls beschränken, so daß auch hier eine On-Request-Invalidierung Vorteile gegenüber Haltesperren verspricht. Eine feste Page-Owner-Zuordnung ist hierbei weniger interessant, da sie sich nicht mit der dynamischen GLA-Zuordnung kombinieren läßt. Insgesamt erscheinen somit folgende Kombinationen zur Synchronisation/Kohärenzkontrolle am aussichtsreichsten:

- S1/S2 + P4 + U4/U5 (Kap. 15.4)
- S3 + P3 + U3 (Kap. 15.3.3)
- S4 + P3 + U4/U5 (s. Übungsaufgaben).

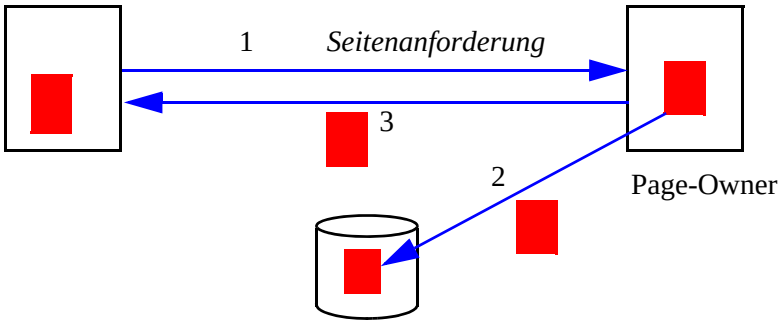
Von diesen Alternativen sprechen wesentliche Vorteile für den zweiten Ansatz, einem Primary-Copy-Sperrverfahren mit übereinstimmender GLA- und Page-Owner-Zuordnung. Die stabile Zuordnung der Verantwortlichkeiten erleichtert ein affinitätsbasiertes Transaktions-Routing zur Unterstützung von rechnerspezifischer Lokalität, welche für alle Sperrverfahren wesentlich zur Einsparung von Kommunikationsvorgängen ist. Zudem entfällt der hohe Aufwand für Migrationsvorgänge von Schreibautorisierungen/Haltesperren bzw. zur Lokalisierung des GLM. Die Kohärenzkontrolle kann ohne jegliche Zusatznachrichten gelöst werden, wobei auch Seitentransfers stets mit Sperrnachrichten kombinierbar sind. Bei dynamischer Page-Owner-Zuordnung verursachen Seitenanforderungen dagegen zusätzliche Verzögerungen. Die On-Request-Invalidierung erfordert einen geringeren Verwaltungsaufwand als Haltesperren, da für jede im System gepufferte Seite eine Haltesperre (oder Transaktionssperre) bestehen muß und somit ein zu wartender Eintrag in der globalen Sperrtabelle des GLM. Auf der anderen Seite verspricht die Vermeidung von Pufferinvalidierungen durch Haltesperren eine bessere Puffernutzung und somit bessere Trefferraten. Ein spezifisches Problem des Primary-Copy-Sperrverfahrens liegt in der Notwendigkeit, eine logische DB-Partitionierung zur GLA/Page-Owner-Zuordnung bestimmen zu müssen.

Wie bereits in Kap. 13.4 ausgeführt, kann das Leistungsverhalten eines Shared-Disk-Systems durch eine nahe Rechnerkopplung signifikant verbessert werden. Wenn z.B. die Synchronisation über eine spezielle Lock-Engine oder eine globale Sperrtabelle in einem synchron zugreifbaren Halbleiterspeicher erfolgt, kann diese Aufgabe oft mit vernachlässigbarer Verzögerung erledigt werden. Ferner können Seitentransfers schnell über einen globalen, nicht-flüchtigen Puffer abgewikkelt werden, der auch zur Verbesserung des E/A-Verhaltens genutzt werden kann. Das Leistungsverhalten wird somit wesentlich unabhängiger vom Referenzverhalten der Last sowie der Lastverteilung. Dennoch bleibt auch hier eine affinitätsbasierte Lastverteilung wesentlich, da sie zumindest das E/A-Verhalten beeinflußt (lokale Trefferraten, Ausmaß an Pufferreplikation und Invalidierungen). Nachteile der nahen Kopplung sind jedoch hohe Hardware-Kosten sowie eine potentiell beeinträchtigte Erweiterbarkeit auf zahlreiche Rechner.

Übungsaufgaben

Aufgabe 15-1: Seitentransfers

Als Kompromiß zwischen einem Austausch geänderter Seiten über Platte und direktem Seitentransfer wurde in [MN91] das in der Abbildung gezeigte Verfahren vorgeschlagen. Dabei schreibt der Page-Owner nach Eintreffen einer Seitenanforderung die geänderte Seite synchron aus, schickt jedoch die Seite selbst mit der Antwort auf die Seitenanforderung an den anfordernden Rechner. Wo liegen die Vor- und Nachteile gegenüber den beiden Basisverfahren für Seitentransfers?

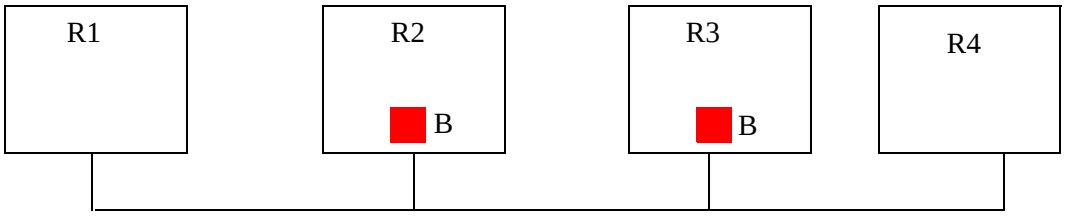


Aufgabe 15-2: On-Request-Invalidierung und Autorisierungen

Eine On-Request-Invalidierung verlangt vor dem Objektzugriff die Prüfung durch den GLM, ob eine gepufferte Seite noch aktuell ist. Mit Lese-/Schreibautorsierungen sollen jedoch Sperren lokal vergeben werden, um Kommunikation mit dem GLM einzusparen. Funktioniert die On-Request-Invalidierung auch bei Einsatz dieser Autorisierungen?

Aufgabe 15-3: On-Request-Invalidierung

Seite B sei in R2 und R3 in der gültigen Version gepuffert.



In dieser Situation sollen nacheinander folgende Operationen ausgeführt werden:

1. Schreibzugriff auf B in Rechner R1
2. Lesezugriff auf B in Rechner R4
3. Schreibzugriff auf B in Rechner R2.

Zur Erkennung von Pufferinvalidierungen sollen Invalidierungsvektoren verwendet werden (Ausgangswert I = 0000); zur Update-Propagierung ein Noforce-Ansatz mit direkten Seitentransfers. Untersuchen Sie die Bearbeitung obiger Aktionen für die folgenden beiden Protokolle

a) dediziertes Sperrprotokoll (ohne Autorisierungen) mit dynamischer Page-Owner-Zuordnung

b) Primary-Copy-Sperrverfahren (ohne Leseautorisierung) mit fester Page-Owner-Zuordnung gemäß der GLA-Verteilung (R2 sei der GLA-Rechner für B).

Geben Sie in beiden Fällen die anfallenden Zwischenschritte an, insbesondere Änderungen des Invalidierungsvektors, der Page-Owner-Zuordnung sowie notwendige Seitentransfers. Wieviele Nachrichten für Sperranforderungen und -freigabe sowie Seitentransfers und wieviele Plattenzugriffe werden in beiden Fällen für die drei Seitenzugriffe benötigt?

Aufgabe 15-4: Haltesperren

Geben Sie an, wie das Beispiel der vorhergehenden Aufgabe bearbeitet wird, falls ein dediziertes Sperrprotokoll mit Haltesperren (Lese- und Schreibautorisierungen) eingesetzt wird. Zu Beginn liege eine RA-Haltesperre in R2 und R3 vor. Der Seitenaustausch soll wiederum über das Kommunikationsnetz erfolgen.Wie ist der hohe Nachrichtenbedarf zu erklären?

Aufgabe 15-5: Kohärenzkontrolle bei dynamischer GLA-Zuordnung

Wie kann die Kohärenzkontrolle bei einem verteilten Sperrprotokoll mit dynamischer GLA-Zuordnung aussehen? Welche Möglichkeiten bestehen, den Kommunikationsaufwand zu reduzieren?

Aufgabe 15-6: Kohärenzkontrolle in Workstation/Server-DBS

Welche Verfahren von dem Shared-Disk-Lösungsspektrum (Abb. 15-13) kommen zur Synchronisation und Kohärenzkontrolle in Workstation/Server-DBS in Betracht? Es soll dabei ein *Page-Server-Ansatz* vorliegen, mit einer Synchronisation auf Seitenebene sowie einer Seitenpufferung in den Workstations sowie im Server. Welche Techniken zur Reduzierung des Kommunikationsumfanges können genutzt werden?

Aufgabe 15-7: Kohärenzkontrolle bei der Katalogverwaltung

In Kapitel 4.3 wurde erläutert, daß für die Katalogverwaltung in Verteilten DBS eine Pufferung nicht-lokaler Katalogdaten zur Einsparung von Kommunikationsvorgängen sinnvoll ist. Es wurden zwei Verfahren zur Kohärenzkontrolle skizziert, die in den Prototypen SDD-1 bzw. R* verwendet wurden. Wo sind diese Ansätze in der eingeführten Klassifikation von Verfahren zur Kohärenzkontrolle einzuordnen?

Teil V
Parallele DB-Verarbeitung

In diesem Teil behandeln wir in drei Kapiteln die parallele DB-Verarbeitung, wobei es vor allem um die Realisierung von Intra-Transaktionsparallelität geht. Im einleitenden Kapitel (Kap. 16) werden zunächst die Metriken Speedup und Scaleup zur Bewertung von Parallelverarbeitung eingeführt sowie unterschiedliche Architekturen Paralleler DBS sowie generelle Parallelisierungsformen diskutiert. In Kap. 17 untersuchen wir die Datenverteilung in Parallelen DBS. Die parallele Anfrageverarbeitung ist schließlich Gegenstand von Kap. 18.

16 Einführung in Parallele DBS

Parallele DBS ermöglichen die Datenbankverarbeitung auf Parallelrechnern, so daß die Verarbeitungskapazität zahlreicher Prozessoren zur Leistungsteigerung genutzt werden kann. Dabei kann man grob zwei Arten der parallelen DB-Verarbeitung unterscheiden: Inter- und Intra-Transaktionsparallelität. Inter-Transaktionsparallelität betrifft die parallele Ausführung mehrerer unabhängiger Transaktionen oder DB-Anfragen. Diese Form der Parallelität wird bereits von zentralisierten DBS sowie von allen Mehrrechner-DBS unterstützt (Mehrbenutzerbetrieb) und ist Voraussetzung für ein akzeptables Durchsatzverhalten. Dieses Kapitel widmet sich schwerpunktmäßig der Intra-Transaktionsparallelität, also der Parallelverarbeitung innerhalb einer Transaktion. Hauptziel hierbei ist die Verkürzung der Bearbeitungszeit von Transaktionen bzw. Anfragen, deren sequentielle Bearbeitung inakzeptable Antwortzeiten verursachen würde.

Eine treibende Kraft für die zunehmende Notwendigkeit von Intra-Transaktionsparallelität sind die ständig wachsenden Datenvolumina, auf denen DB-Operationen abzuwickeln sind. Die größten, kommerziell genutzten relationalen Datenbanken umfassen bereits heute mehrere Terabyte (TB), wobei einzelne Relationen über 100 Gigabyte (GB) belegen [Pi90]. Die sequentielle Verarbeitung von DB-Operationen ist demgegenüber sehr langsam, trotz ständig schneller werdender Hardware. So ist das sequentielle Einlesen der Daten von Magnetplatten typischerweise auf etwa 5 MB/s beschränkt. Auch die Ausführungszeiten relationaler Operatoren auf hauptspeicherresidenten Daten ist relativ langsam. So sind für die einfachste Operation, die sequentielle Suche im Rahmen eines Relationen-Scans, nach [GHW90] etwa 1000 Instruktionen pro Satz anzusetzen. Für einen Prozessor von 100 MIPS und eine Satzlänge von 100 B können somit lediglich 10 MB/s sequentiell durchsucht werden. Komplexere Operationen wie Sortieren oder Join-Bildung sind typischerweise um mindestens eine Größenordnung langsamer (< 1 MB/s). Die sequentielle Verarbeitung einer relationalen DB-Operation auf 1 TB würde somit Bearbeitungszeiten von mehreren Tagen erfordern - und dies auch nur, wenn keine Behinderungen mit anderen Transaktionen auftreten (Einbenut-

zerbetrieb). Offensichtlich sind solche Bearbeitungszeiten in den allermeisten Fällen nicht tolerierbar.

Eine Verschärfung der Problematik ergibt sich für sogenannte Nicht-Standard-Anwendungen, die von relationalen DBS typischerweise nicht adäquat abgedeckt werden. Multimedia-Anwendungen verlangen so den Zugriff auf sehr große Datenmengen mit sehr restriktiven Antwortzeitvorgaben, z.B. um digitalisierte Filme in hoher Qualität abzuspielen oder um Videoaufnahmen in Echtzeit digital zu speichern. Ingenieuranwendungen, z.B. bei CAD (Computer-Aided Design) oder CASE (Computer-Aided Software Engineering), verursachen umfangreiche Operationen auf komplex strukturierten Objekten mit einem Berechnungsaufwand, der den einfacher relationaler Operatoren um ein Vielfaches übersteigt. In diesen Bereichen ist der Einsatz von Intra-Transaktionsparallelität daher umso dringlicher, um für den Dialogbetrieb ausreichend kurze Bearbeitungszeiten zu ermöglichen.

Die Erlangung kurzer Antwortzeiten durch Intra-Transaktionsparallelisierung wird dann besonders erschwert, wenn mehrere komplexe Anfragen gleichzeitig zu bearbeiten sind bzw. gleichzeitig ein hoher Durchsatz für kurze OLTP-Transaktionen erreicht werden muß. Dies ist vielen Anwendungsbereichen sicherlich wünschenswert bzw. notwendig, wird jedoch von derzeitigen Implementierungen paralleler DBS noch nicht zufriedenstellend gelöst. Dieser Aspekt ist Gegenstand laufender Forschungsarbeiten (s. Kap. 18.4.2).

Neben diesen Leistungsforderungen stellen sich an Parallele Datenbanksysteme weitere der bereits in Kap. 1.2 diskutierten Anforderungen. Dies ist zum einen eine hohe Skalierbarkeit, so daß durch Einsatz von Intra-Transaktionsparallelität die Antwortzeiten möglichst linear mit der Anzahl der Prozessoren verkürzt werden kann. Ferner soll für OLTP-Anwendungen die Transaktionsrate linear mit der Prozessoranzahl steigen. Zum anderen sind die Unterstützung einer hohen Verfügbarkeit sowie einer guten Kosteneffektivität von großer Wichtigkeit für Parallele DBS. Keine Rolle dagegen spielen Forderungen nach Knotenautonomie, Unterstützung heterogener DBS oder dezentraler Organisationsstrukturen.

Im folgenden führen wir zunächst zwei allgemeine Leistungsmaße für Parallelverabeitung ein, nämlich Speedup und Scaleup. Danach diskutieren wir die Architektur Paralleler DBS (Kap. 16.2) und klassifizieren verschiedene Typen von Intra-Transaktionsparallelität (Kap. 16.3).

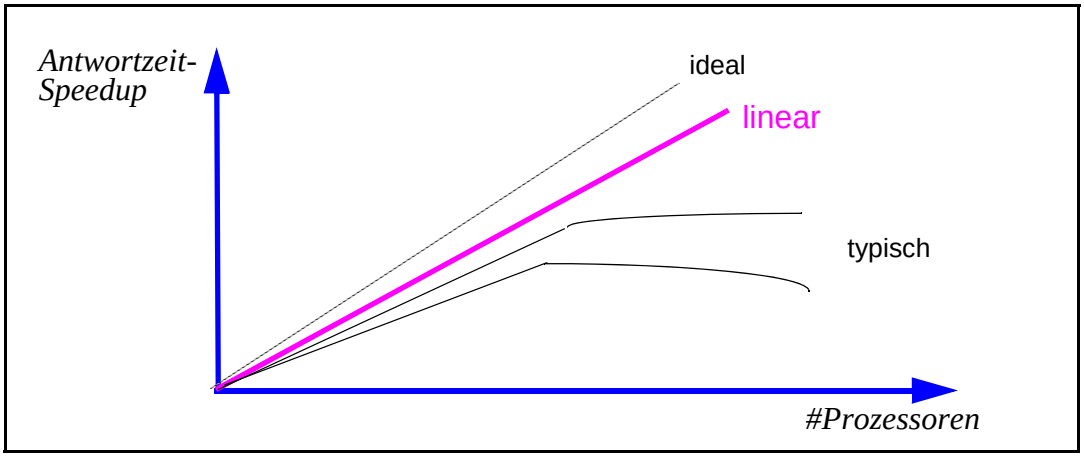
16.1 Speedup und Scaleup

Speedup ist ein allgemeines Maß zur Bestimmung, in welchem Umfang die Leistungsfähigkeit von Computersystemen durch eine bestimmte Optimierung verbessert wird [HP90]. Üblicherweise wird Speedup mit Hinblick auf die Verkürzung von Antwortzeiten verwendet. Mit dem Antwortzeit-Speedup kann insbesondere die Effektivität einer Intra-Transaktionsparallelisierung bestimmt werden, indem festgestellt wird, in welchem Maß sich die Antwortzeit einer bestimmten Transaktion bzw. Operation durch Parallelisierung verbessert hat. Dabei gilt:

$$\text{Antwortzeit-Speedup (n)} = \frac{\text{Antwortzeit bei sequentieller Verarbeitung}}{\text{Antwortzeit bei paralleler Verarbeitung auf n Prozessoren}}$$

Abb. 16-1 zeigt, welche Verlaufsformen für den Antwortzeit-Speedup unterschieden werden können. Idealerweise wird bei n Prozessoren ein Speedup-Wert von n erzielt. Bei einem linearen Antwortzeit-Speedup kann die Antwortzeit auch proportional zur Prozessoranzahl verbessert werden, jedoch i.a. auf einem geringeren Niveau*. Typischerweise läßt sich jedoch die Antwortzeit nur bis zu einer bestimmten Prozessoranzahl verkürzen. Eine weitere Erhöhung der Prozessorzahl führt dann ggf. zu einer Reduzierung des Speedups, also einer Zunahme der Antwortzeit.

Abb. 16-1: Idealer, linearer und typischer Antwortzeit-Speedup



Die suboptimale Speedup-Entwicklung basiert auf mehreren Ursachen. Zunächst ist der maximal mögliche Speedup durch den Anteil einer Transaktion bzw. Operation begrenzt, der überhaupt parallelisierbar ist (begrenzte inhärente Paralleli-

* In bestimmten Fällen ist jedoch ein super-linearer Speedup erreichbar, wobei die Antwortzeit bei n Prozessoren um mehr als den Faktor n verbessert wird. Dies ist z.B. durch E/A-Einsparungen möglich, die darauf basieren, daß die Gesamt- Hauptspeicherkapazität mit der Prozessorzahl wächst, während die DB-Größe gleich bleibt.

tät). Besteht zum Beispiel die Antwortzeit einer Transaktion nur zu 5% aus nicht-parallelisierbaren (sequentiellen) Verarbeitungsanteilen, so ist der maximal möglich Speedup auf 20 beschränkt, unabhängig davon, wieviele Prozessoren eingesetzt werden. Diesen Zusammenhang verdeutlicht *Amdahls Gesetz*, welches den Speedup berechnet, wenn nur ein bestimmter Antwortzeitanteil durch eine Optimierung verkürzt werden kann [HP90]:

$$\text{Antwortzeit-Speedup} = \frac{1}{(1 - F_{opt}) + \frac{F_{opt}}{S_{opt}}}$$

F_{opt} = Anteil der optimierten Antwortzeitkomponente ($0 \leq F_{opt} \leq 1$)
S_{opt} = Speedup für optimierten Antwortzeitanteil

Desweiteren sind es vor allem folgende Faktoren, die den Antwortzeit-Speedup und damit die Skalierbarkeit einer Anwendung beeinträchtigen können [DG92]:

- *Startup- und Terminierungskosten*
  Das Starten und Beenden mehrerer Teiloperationen in verschiedenen Prozessen/Rechnern verursacht einen Overhead, der mit dem Parallelitätsgrad zunimmt. Da umgekehrt die pro Teiloperation zu verrichtende Nutzarbeit (z.B. Anzahl zu verarbeitender Sätze) sinkt, vermindert sich der relative Gewinn einer Parallelisierung mit wachsendem Parallelitätsgrad.
- *Interferenz*
  Die Erhöhung der Prozeßanzahl führt zu verstärkten Wartezeiten auf gemeinsam benutzten Systemressourcen. Vor allem der durch die Parallelisierung eingeführte Kommunikations-Overhead kann sich negativ bemerkbar machen, insbesondere im Mehrbenutzerbetrieb (Inter-Transaktionsparallelität). Neben Wartezeiten auf physischen Ressourcen (CPU, Hauptspeicher, Platten, etc.) kann es auch verstärkt zu Sperrkonflikten zwischen unabhängigen Transaktionen kommen.
- *Skew (Varianz der Ausführungszeiten)*
  Die langsamste Teiloperation bestimmt die Bearbeitungszeit einer parallelisierten Operation. Varianzen in den Ausführungszeiten, z.B. aufgrund ungleichmäßiger Daten- oder Lastverteilung oder Sperrkonflikten, führen daher zu Speedup-Einbußen. Das Skew-Problem nimmt i.a. auch mit wachsendem Parallelitätsgrad (Rechneranzahl) zu und beschränkt daher die Skalierbarkeit.

Der Speedup bestimmt den Einfluß der Prozessoranzahl auf eine konstante Problemgröße, nämlich die Ausführung einer bestimmten Transaktion bzw. Operation auf einer Datenbank fester Größe. Dagegen soll bei der *Scaleup*-Metrik die Problemgröße, hier die DB-Größe, linear mit der Prozessoranzahl erhöht werden [DG92]. Dabei kann zwischen Antwortzeit- und Durchsatz-Scaleup unterschieden werden. Der *Antwortzeit-Scaleup* (batch scaleup) bestimmt die Antwortzeitveränderung bei Einsatz von n Rechnern und n-facher Datenbankgröße verglichen

mit dem 1-Rechner-Fall. Dabei soll im Mehrrechnerfall trotz des höheren Datenvolumens aufgrund der Parallelisierung einer Operation möglichst die gleiche Antwortzeit wie bei einem Rechner erreicht werden (Antwortzeit-Scaleup = 1). Der *Durchsatz-Scaleup* bestimmt das Verhältnis zwischen der Transaktionsrate auf n Rechnern (auf einer n-fach großen Datenbank) gegenüber der Transaktionsrate auf einem Rechner. Dabei sollte idealerweise der Scaleup-Wert n erreicht werden (lineares Durchsatzwachstum). In OLTP-Benchmarks des TPC (Transaction Processing Performance Council) wird die Skalierungsregel für die DB-Größe bei der Bestimmung der Transaktionsraten verlangt [Gr93].

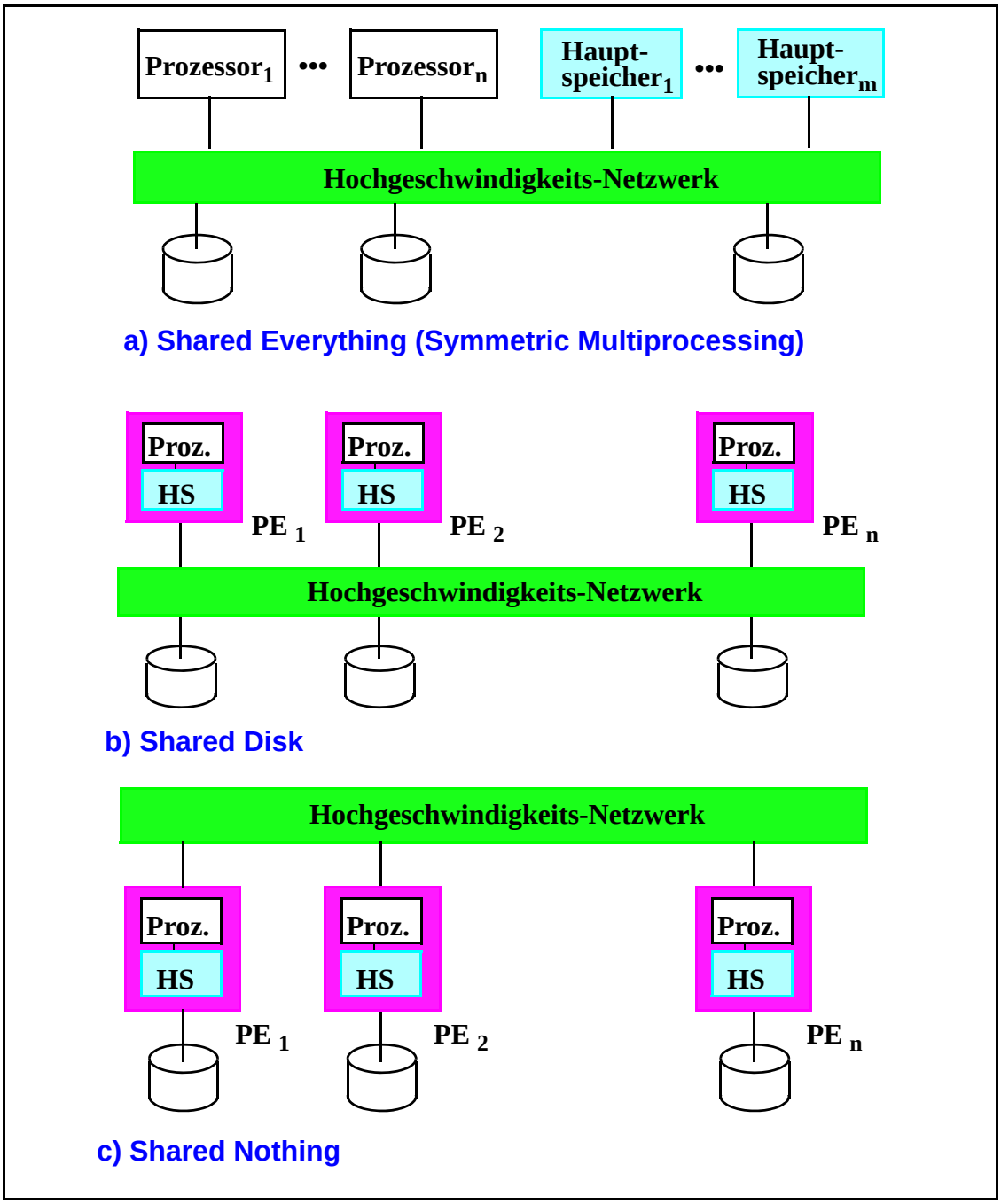
16.2 Architektur von Parallelen DBS

Die Diskussion verschiedener Typen von Mehrrechner-DBS in Kap. 3.4 hat gezeigt, daß die Anforderungen an Parallele DBS (hohe Leistung, Skalierbarkeit, hohe Verfügbarkeit, Kosteneffektivität) am besten von lokal verteilten, integrierten Mehrrechner-Datenbanksystemen erfüllt werden. Die Nutzung mehrerer Prozessoren (*Verarbeitungsparallelität*) wird vor allem durch die drei Architekturklassen Shared-Nothing, Shared-Disk sowie (mit Abstrichen) Shared-Everything ermöglicht (Abb. 16-2). Daneben sind auch hybride Architekturen denkbar, z.B. Shared-Nothing- oder Shared-Disk-Systeme, bei denen jeder Knoten vom Typ Shared-Everything ist [Va93a, Va93b]. Statt der in diesem Buch verwendeten Architekturbezeichnungen findet man auch häufig die Begriffe SMP (Symmetric Multiprocessing) und MPP (Massive Parallel Processing). SMP-Systeme entsprechen Multiprozessoren und somit Shared-Everything. MPP-Systeme sind lose gekoppelte Mehrrechnersysteme, welche sowohl dem Shared-Nothing- oder Shared-Disk-Ansatz folgen können. Üblicherweise geht man dabei von Parallelrechnern aus, die aus Hunderten bis Tausenden von Prozessor-Elementen (PE) bestehen.

Die lokale Verteilung innerhalb eines Clusters ist aus Leistungsgründen als obligatorisch anzusehen (Kap. 3.1.1). Dies gestattet den Einsatz eines skalierbaren Hochgeschwindigkeits-Netzwerkes, dessen Übetragungskapazität proportional zur Prozessoranzahl gesteigert werden kann. Eine sehr schnelle Inter-Prozessor-Kommunikation ist für die effiziente Zerlegung einer Transaktion in zahlreiche Teiltransaktionen sowie für die Übertragung großer Datenmengen entscheidend. Weiterhin kann bei lokaler Verteilung am ehesten eine dynamische Lastbalancierung erreicht werden, wobei der aktuelle Systemzustand berücksichtigt wird. Dies betrifft sowohl die initiale Verteilung eintreffender Transaktionen und Anfragen unter den Verarbeitungsrechnern (Transaktions-Routing) als auch die Rechnerzuordnung von Teilaufträgen während der parallelisierten Abarbeitung. Die Lastbalancierung wird auch dadurch erleichtert, daß Parallele DBS auf homogenen Architekturen basieren, so daß eine bestimmte DB-Funktion prinzipiell von jedem Prozessor ausgeführt werden kann*.

.

Abb. 16-2: Shared-Everything, Shared-Disk, Shared-Nothing



Wesentlich für die Kosteneffektivität von Parallelen DBS ist der weitgehende Verzicht auf Spezial-Hardware, da diese i.a. hohe Kosten verursacht. Stattdessen sollte soweit wie möglich Standard-Hardware zum Einsatz kommen, insbesondere weit verbreitete Mikroprozessoren für die CPUs. Durch die software-mäßige Parallelisierung der DB-Verarbeitung können dann sowohl die geringen Kosten als

* Für Shared-Nothing gilt dies aufgrund der Abhängigkeiten zur Datenverteilung nur eingeschränkt.

auch die enormen Leistungssteigerungen der Standard-Hardware (Mikroprozessoren) unmittelbar genutzt werden. Dieser Aspekt wurde bei der Realisierung älterer DB-Maschinen ignoriert und ist Hauptgrund für deren geringe Relevanz (Kap. 3.3). Eine software-basierte Parallelisierung kann jedoch durchaus innerhalb eines auf DB-Verarbeitung beschränkten Back-End-Systems (DB-Maschine, Server) erfolgen (Beispiele: Teradata, IBM Parallel Query Server). Für OLTP-Anwendungen auf allgemeinen Verarbeitungsrechnern kann dies jedoch zu hohen Kommunikationskosten führen, wenn jede (einfache) DB-Operation eine Kommunikation mit dem Back-End-System erfordert. Für Anfragen, die große Ergebnismengen zurückliefern, entsteht ebenfalls ein hoher Kommunikationsaufwand mit dem Back-End-System, wenn die Ergebnisse satzweise abgerufen werden

Intra-Transaktionsparallelität wurde bisher vor allem für Shared-Everything- (Multiprozessoren) und für Shared-Nothing-Systeme untersucht sowie innerhalb von Prototypen und kommerziellen DBS realisiert. Die Nutzung von Multiprozessoren zur Parallelverarbeitung kann dabei als erster Schritt aufgefaßt werden, da er i.a. auf relativ wenige Prozessoren begrenzt ist (Kap. 3.1). Die Verwendung eines gemeinsamen Hauptspeichers erleichtert dabei die Parallelisierung, da eine effiziente Kommunikation zwischen Prozessen einer Transaktion und eine einfachere Lastbalancierung möglich wird. Prototyp-Realisierungen paralleler Shared-Everything-DBS sind u.a. XPRS [SKPO88], Volcano [Gra94] und DBS3 [ZZB93]. Auch kommerzielle DBS nutzen zunehmend Multiprozessoren für Intra-Transaktionsparallelität, z.B. DB2 [Ha90] oder Informix [Dav92].

Shared-Nothing-Systeme, aber auch Shared-Disk-Systeme, bieten eine größere Erweiterbarkeit als Shared-Everything, so daß eine entsprechend weitergehende Parallelisierung unterstützt werden kann. Bei der Realisierung lokal verteilter Shared-Nothing-DBS können viele der Konzepte von Verteilten DBS übernommen werden, z.B. zur Transaktionsverarbeitung (Commit-Behandlung, Synchronisation). Dabei ergibt sich aufgrund der schnelleren Kommunikation i.a. eine effizientere Bearbeitung als im ortsverteilten Fall*. Bezüglich der Katalog- und Namensverwaltung spielt die Wahrung einer hohen Knotenautonomie keine Rolle mehr, so daß einfachere Lösungen möglich werden. Zum Beispiel kann eine vollständige Replikation der Katalogdaten vorgesehen werden. Notwendige Erweiterungen zur Unterstützung von Intra-Transaktionsparallelität betreffen vor allem die Bestimmung der Datenverteilung sowie die Anfragebearbeitung; darauf wird in diesem Kapitel ausführlich eingegangen. Intra-Transaktionsparallelität wird bereits seit mehreren Jahren in den beiden Shared-Nothing-Systemen Teradata (Kap. 19.8) und Tandem NonStop-SQL (Kap. 19.7) unterstützt; neuere Entwicklungen bestehen für Sybase (Kap. 19.6) und DB2/6000 (Kap. 19.1.3). Daneben verfolgen

* Auch können mächtigere Kommunikationsprimitive wie Broadcast- oder Multicast-Kommunikation vorteilhaft genutzt werden, z.B. zur parallelen Commit-Behandlung.

zahlreiche Prototypen den Shared-Nothing-Ansatz, z.B. Gamma [De90], Bubba [Bo90], EDS [Sk92], Prisma [Ap92] und Arbre [LY89].

Demgegenüber stehen die Untersuchungen zur Intra-Transaktionsparallelität in Shared-Disk-Systemen noch am Anfang [Ra93e]. Existierende Implementierungen sind bisher weitgehend auf Inter-Transaktionsparallelität beschränkt. Eine initiale Unterstützung von Intra-Transaktionsparallelität auf Basis von Oracles "Parallel Server" (Kap. 19.2.2) wurde für den KSR1-Parallelrechner (Kendall Square) realisiert [RMW93]. In der allgemeinen Produktversion 7.1 ist ebenfalls eine beschränkte Form von Intra-Transaktionsparallelität vorgesehen. Daneben wird in dem neuen IBM Shared-Disk-System Parallel Query Server für DB2 (Kap. 19.1.3) eine Parallelisierung von SQL-Anfragen unterstützt.

Parallele DBS müssen nicht nur Verarbeitungsparallelität, sondern auch *E/A-Parallelität* unterstützen, insbesondere um große Datenmengen parallel von mehreren Platten einzulesen. Dies setzt auch in Shared-Everything- und Shared-Disk-DBS eine entsprechende Verteilung der Daten (Relationen) über mehrere Platten voraus. Dies kann transparent für das DBS erfolgen, und zwar innerhalb sogenannter *Disk-Arrays*, welche in der jüngsten Vergangenheit starke Bedeutung erreicht haben [PGK88, Ra93a, WZ93]. Disk-Arrays bestehen intern aus mehreren Platten, können jedoch logisch wie eine Platte angesprochen werden, so daß ihr Einsatz prinzipiell keine Änderungen in der nutzenden Software erfordert. Durch den Einsatz von E/A-Parallelität soll das Leistungsverhalten gegenüber einzelnen Platten jedoch deutlich verbessert werden. Ferner sind automatische Methoden zur Fehlerbehandlung vorgesehen, um eine hohe Ausfallsicherheit zu gewährleisten. Ein weiterer Vorteil liegt in der Verwendung kleiner Platten (z.B. mit einem Durchmesser von 3,5 Zoll), welche eine verbesserte Kosteneffektivität gegenüber großen Platten aufweisen. Der Kostenvorteil wird jedoch durch die Notwendigkeit spezialisierter Platten-Kontroller wieder kompensiert, welche für die Organisation der E/A-Parallelität sowie die Fehlerbehandlung verantwortlich sind.

Die Realisierung Paralleler DBS mit solchen Disk-Arrays wurde bisher noch nicht ausreichend untersucht und ist zumindest problematisch. Denn außerhalb des DBS kann eine Datenverteilung nur auf physischen Objektgranulaten wie Blöcken erfolgen. Wünschenswert dagegen wäre eine logische Definition der Datenverteilung über Datenwerte, um so die Bearbeitung bestimmter Operationen auf eine Teilmenge der Platten begrenzen zu können. Ferner kann eine Parallelisierung ohne genaue Kenntnis der Datenverteilung dazu führen, daß parallele Teiloperationen auf dieselben Platten zugreifen und somit E/A-Engpässe erzeugt werden*. Zudem stellen Disk-Arrays Spezial-Hardware dar und weisen eine Reihe weiterer

* Die Umgehung dieser Probleme setzt also u.a. voraus, daß dem DBS Angaben zur Datenverteilung innerhalb von Disk-Arrays bekanntgemacht werden (Abschwächung der "Geräteunabhängigkeit").

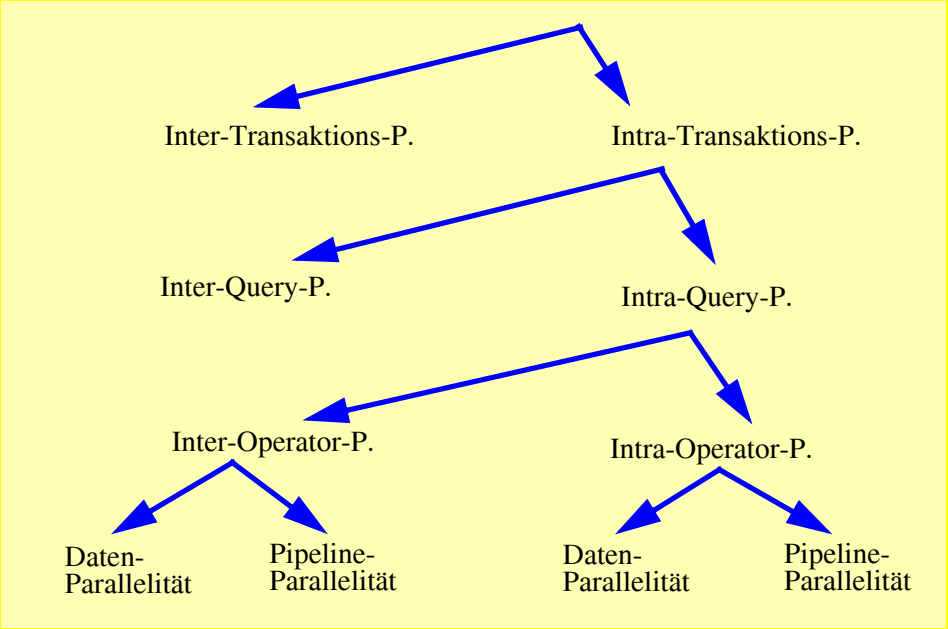
Probleme für die DB-Verarbeitung auf (z.B. ist die Erzeugung von Archivkopien sehr problematisch) [GHW90]. Wir werden daher diesen Ansatz hier nicht weiter betrachten, sondern annehmen, daß die Verteilung der Daten über mehrere Platten den DBVS bekannt ist und über Datenbankwerte definiert werden kann.

16.3 Arten der Parallelverarbeitung

Parallelverarbeitung für DB-Anwendungen kann auf unterschiedliche Weise klassifiziert werden. Wir betrachten hier drei Unterscheidungskriterien:

- Parallelität innerhalb von bzw. zwischen unterschiedlichen Verarbeitungsgranulaten (Transaktionen, DB-Operationen, Operatoren)
- Daten- vs. Pipeline-Parallelität
- Verarbeitungs- vs. E/A-Parallelität.

Abb. 16-3: Arten der parallelen DB-Verarbeitung



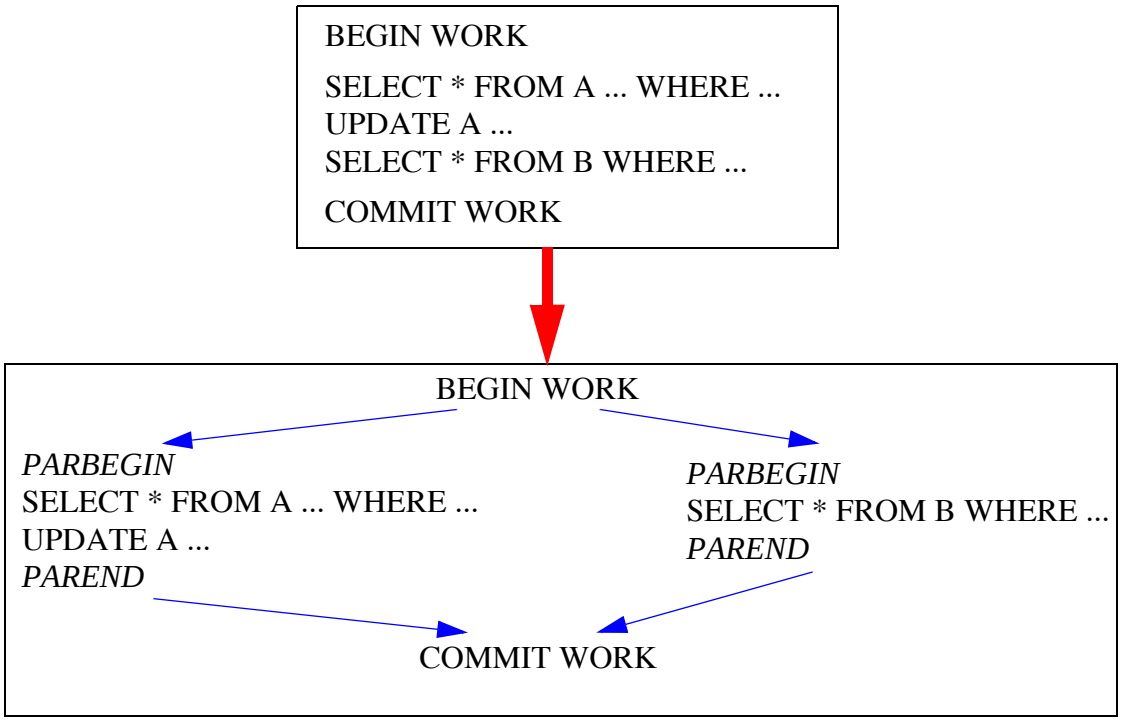
Transaktions-, Query- und Operatorparallelität

Die aus den ersten beiden Kriterien resultierenden Typen der Parallelverarbeitung sind in Abb. 16-3 dargestellt. Auf die Unterscheidung von Inter- und Intra-Transaktionsparallelität wurde bereits zu Beginn des Kapitels kurz eingegangen. *Inter-Transaktionsparallelität* (Mehrbenutzerbetrieb) ist als unabdingbar zu betrachten, um Durchsatzforderungen zu erfüllen. Außerdem könnte im Einbenutzerbetrieb die verfügbare CPU-Kapazität sehr vieler Prozessoren kaum ausgeschöpft werden, so daß eine inakzeptable Kosteneffektivität die Folge wäre. *Intra-Transaktionsparallelität* ist notwendig, um die Bearbeitungszeiten aufwendige-

rer Transaktionen zu verkürzen. Die weiteren Typen der Parallelisierung betreffen nur noch Intra-Transaktionsparallelität.

Da eine Transaktion i.a. aus mehreren DB-Operationen oder Queries besteht, kann zwischen Inter- und Intra-Query-Parallelität unterschieden werden*. Bei *Inter-Query-Parallelität* werden verschiedene DB-Operationen derselben Transaktion parallel bearbeitet. Dabei sind jedoch Präzedenzabhängigkeiten zwischen den Operationen zu beachten, um sicherzustellen, daß nur unabhängige Operationen parallel zueinander ausgeführt werden. Dies erfordert i.a. eine explizite Festlegung durch den Programmierer, welche Operationen parallel bearbeitet werden können (s. Beispiel in Abb. 16-4). Neben der Verkomplizierung der Programmierung kann mit diesem Ansatz auch meist nur eine geringe Antwortzeitverbesserung erwartet werden. Denn der maximale Parallelisierungsgrad (Speedup) ist durch die Anzahl der DB-Operationen eines Transaktionsprogramms begrenzt; bei Abhängigkeiten zwischen DB-Operationen läßt sich selbst dieser Wert nicht erreichen. Von Vorteil ist, daß für das DBS diese Form der Intra-Transaktionsparallelität einfach zu unterstützen ist, da prinzipiell keine Erweiterungen hinsichtlich der Query-Optimierung erforderlich sind. Dennoch unterstützen derzeitige DBS noch keine Inter-Query-Parallelität, wohl auch deshalb, da im SQL-Standard noch keine Anweisungen zur Parallelisierung vorgesehen sind.

Abb. 16-4: Parallelisierung von Transaktionsprogrammen (Inter-Query-Parallelität)



* Für Ad-Hoc-Anfragen ist Inter-Query-Parallelität offenbar nicht anwendbar. Für solche Anfragen sind Intra-Transaktions- und Intra-Query-Parallelität gleichbedeutend.

Relationale DBS mit ihren deskriptiven und mengenorientierten Anfragesprachen wie SQL ermöglichen die Nutzung von Parallelarbeit innerhalb einer DB-Operation (*Intra-Query-Parallelität*). Die Bearbeitung einer DB-Operation erfordert i.a. die Ausführung mehrerer relationaler Basisoperatoren wie Selektion, Projektion, Join etc., deren Ausführungsreihenfolge durch einen Operatorbaum beschrieben werden kann (Kap. 6.2). Damit lassen sich zwei weitere Arten der Parallelisierung unterscheiden, nämlich Inter- und Intra-Operatorparallelität. In beiden Fällen erfolgt die Parallelisierung vollkommen automatisch durch das DBS und somit transparent für den Programmierer und DB-Benutzer. Dies ist ein Hauptgrund für den Erfolg paralleler DB-Verarbeitung und stellt einen wesentlichen Unterschied zur Parallelisierung in anderen Anwendungsbereichen dar, die i.a. eine sehr schwierige Programmierung verlangen [Re92].

Beim Einsatz von *Inter-Operatorparallelität* werden verschiedene Operatoren einer DB-Operation parallel ausgeführt*. Auch hier bestehen Präzedenzabhängigkeiten zwischen den einzelnen Operatoren, die die Parallelität einschränken; jedoch sind diese im Gegensatz zur Inter-Query-Parallelität dem DBS (Query-Optimierer) bekannt. Der erreichbare Parallelitätsgrad ist in jedem Fall durch die Gesamtanzahl der Operatoren im Operatorbaum begrenzt. Bei der Intra-Operatorparallelität schließlich erfolgt die parallele Ausführung der einzelnen Basisoperatoren.

Daten- vs. Pipeline-Parallelität

Wie Abb. 16-3 zeigt, kann sowohl Inter- als auch Intra-Operatorparallelität auf Daten- oder Pipeline-Parallelität basieren**. *Datenparallelität* erfordert eine Partitionierung der Daten, so daß verschiedene Operatoren bzw. Teiloperatoren auf disjunkten Datenpartitionen arbeiten. So können z.B. Selektionsoperatoren auf verschiedenen Relationen parallel ausgeführt werden (Inter-Operatorparallelität). Eine Selektion auf einer einzelnen Relation läßt sich auch parallelisieren, wenn die Relation in mehrere Fragmente partitioniert wird (Intra-Operatorparallelität). Diese Form der Parallelisierung hat den Vorteil, daß der Parallelitätsgrad proportional zur Relationengröße erhöht werden kann.

Pipeline-Parallelität sieht eine überlappende Ausführung verschiedener Operatoren bzw. Teiloperatoren vor, um die Ausführungszeit zu verkürzen. Für benachbarte Operatoren im Operatorbaum bzw. Teiloperatoren eines Basisoperators, zwischen denen eine Erzeuger-Verbraucherbeziehung besteht, werden dabei die Ausgaben des Erzeugers im Datenflußprinzip an den Verbraucher weitergeleitet. Dabei wird die Verarbeitung im Verbraucherprozeß nicht verzögert, bis der Erzeuger die gesamte Eingabe bestimmt hat. Vielmehr werden die Sätze der Ergebnis-

* Gelegentlich wird Inter-Operatorparallelität als *Funktionsparallelität* bezeichnet.

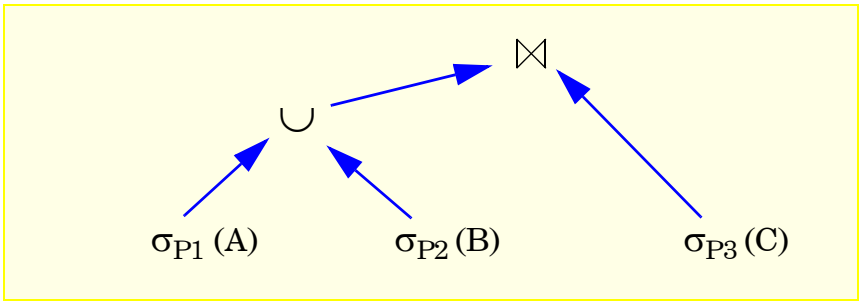
** Datenparallelität wird auch als *horizontale*, Pipeline-Parallelität als *vertikale Parallelität* bzw. *Datenfluß-Parallelität* bezeichnet.

menge fortlaufend weitergeleitet, um eine frühzeitige Weiterverarbeitung zu ermöglichen. Pipeline-Parallelität kommt primär zur Realisierung von Inter-Operatorparallelität in Betracht. Für komplexere Operatoren (z.B. Join) ist sie prinzipiell jedoch auch zur Realisierung von Intra-Operatorparallelität anwendbar.

Beispiel 16-1

Für den in Abb. 16-5 gezeigten Operatorbaum läßt sich Pipeline-Parallelität zwischen allen gezeigten Operatoren nutzen. So kann jedes Ergebnistupel der drei Selektionsoperatoren sofort an den nächsten Operator im Baum geleitet werden, um dort eine Weiterverarbeitung zu ermöglichen. Ergebnistupel des Vereinigungsoperators können ferner sofort an den Join-Operator geschickt werden. Daneben lassen sich natürlich die drei Selektionsoperatoren parallel zueinander ausführen, da sie unterschiedliche Relationen betreffen (Datenparallelität). Zudem kann ggf. Datenparallelität für jede Selektion genutzt werden (Intra-Operatorparallelität).

Abb. 16-5: Beispiel eines parallelisierbaren Operatorbaumes



Pipeline-Parallelität verursacht jedoch einen hohen Kommunikations-Overhead, wenn jedes Ergebnistupel einzeln weitergegeben wird, vor allem wenn die Operatoren in Prozessen verschiedener Rechner ausgeführt werden (Shared-Nothing bzw. Shared-Disk). Ohne Pipeline-Parallelität verzögert sich jedoch der Start eines Operators bis die Ergebnismengen der Vorgängeroperatoren vollständig vorliegen. Zudem kann es dabei zu erheblichem Mehraufwand an E/A im Falle großer Ergebnismengen kommen, wenn diese auf Platte zwischengespeichert werden müssen. Ein Kompromiß besteht darin, den Kommunikationsaufwand der Pipeline-Parallelität dadurch zu beschränken, indem jeweils mehrere Ergebnistupel gebündelt übertragen werden.

Dennoch sind die mit Pipeline-Parallelität erreichbaren Speedup-Werte meist gering, da in relationalen DB-Operationen selten mehr als 10 Operatoren auf diese Weise überlappt ausgeführt werden können [DG92]. Dies ist auch dadurch bedingt, daß einige Operatoren zur Unterbrechung einer Pipeline führen, da sie die vollständigen Ergebnismengen von Vorgängeroperatoren benötigen, bevor sie ein Ergebnis weitergeben können. Dies sind insbesondere Sortieroperatoren sowie Operatoren zur Duplikateliminierung oder Berechnung von Aggregatfunktionen. Schließlich bestehen oft erhebliche Unterschiede in den Ausführungszeiten ver-

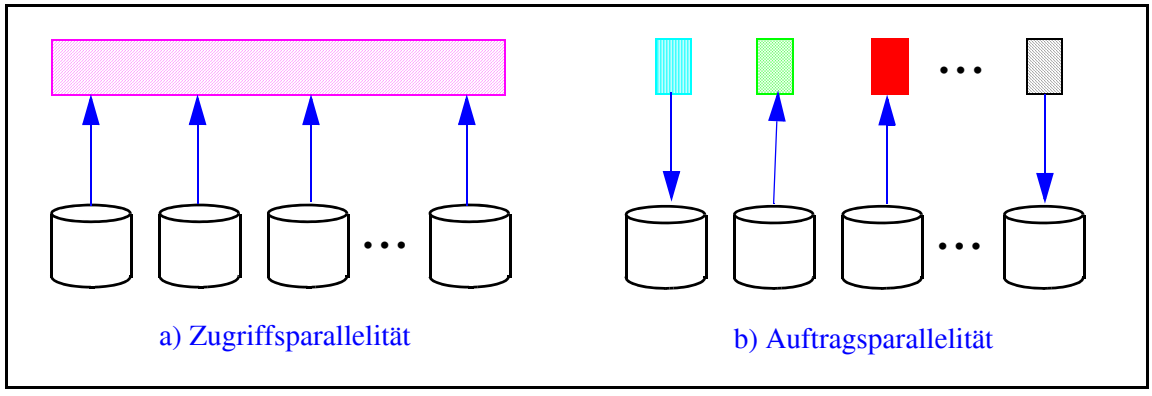
schiedener Operatoren, so daß der erreichbare Speedup stark beeinträchtigt wird (Skew-Effekt).

Verarbeitungs- vs. E/A-Parallelität

Die bisher diskutierten Parallelisierungsformen entsprechen unterschiedlichen Arten von Verarbeitungsparallelität, die sich vor allem auf die Nutzung mehrerer Prozessoren bezieht. Diese Ansätze können jedoch nur dann ihre Wirksamkeit entfalten, wenn sie durch eine entsprechende E/A-Parallelität beim Externspeicherzugriff unterstützt werden. Denn ansonsten würde die Sequentialisierung der Plattenzugriffe sämtliche Parallelitätsgewinne bei den CPU-bezogenenen Verarbeitungsanteilen wieder zunichte machen. So führt z.B. die parallele Abwicklung von Teiloperationen, welche den Zugriff auf dieselbe Platte erfordern, primär zu erhöhten Wartezeiten beim Plattenzugriff, statt zu einer Verkürzung der Antwortzeit.

Man kann grob zwei Arten von E/A-Parallelität unterscheiden, welche durch eine geeignete Datenverteilung über mehrere Platten zu unterstützen sind. Zum einen ist es notwendig, E/A-Vorgänge auf große Datenmengen parallel von mehreren Platten zu bedienen, um kurze Zugriffszeiten zu erhalten (Abb. 16-6a). Diese Art der E/A-Parallelität wurde in [WZ93] als *Zugriffsparallelität* bezeichnet und kommt vor allem bei der Intra-Operatorparallelität zum Tragen. Daneben sollte ein möglichst hoher Durchsatz bzw. hohe E/A-Raten für unabhängige E/A-Aufträge erzielt werden, indem diese möglichst von verschiedenen Platten bedient werden (Abb. 16-6b). Diese sogenannte *Auftragsparallelität* ist für Inter-Transaktionsparallelität erforderlich, jedoch auch für Inter-Query- sowie Inter-Operatorparallelität. Beide Formen der E/A-Parallelität dienen primär der Datenparallelität, also dem parallelen Zugriff auf disjunkte Datenmengen. Pipeline-Parallelität arbeitet dagegen auf Zwischenergebnissen während der Anfrageverarbeitung und kommt idealerweise ohne Externspeicherzugriffe aus.

Abb. 16-6: Zugriffs- vs. Auftragsparallelität



Übungsaufgaben

Aufgabe 16-1: Antwortzeit-Speedup

Ein Relationen-Scan auf einer Relation mit 10 Millionen Tupel verursache bei einem Rechner (10 MIPS) eine Bearbeitungszeit von 150 Sekunden (davon 50 Sekunden für E/A).

- Welcher Antwortzeit-Speedup wird erreicht bei einem Prozessor von 100 MIPS ?
- Welcher Antwortzeit-Speedup ergibt sich bei Nutzung von Datenparallelität auf 10 (100) Rechnern mit jeweils 10 (1) MIPS ? Die durch die Verteilung eingeführte Kommunikationsverzögerung soll bei n Rechnern 50 + n*10 ms betragen.

Aufgabe 16-2: Parallelisierung von Transaktionsprogrammen

In den Benchmarks TPC-A und TPC-B wird folgendes Programm zur Realisierung einer Kontenbuchung (Debit-Credit) zugrundegelegt.

```
BEGIN WORK
UPDATE ACCOUNT SET balance = balance + :delta WHERE acct_no = :acctno;
SELECT balance INTO :abalance FROM ACCOUNT WHERE acct_no = :acctno;
UPDATE TELLER SET balance = balance + :delta WHERE teller_no = :tellerno;
UPDATE BRANCH SET balance = balance + :delta WHERE branch_no = :branchno;
INSERT INTO HISTORY ( ...) VALUES ( ...);
COMMIT WORK;
```

Wie könnte eine Parallelisierung zur Nutzung von Inter-Query-Parallelität aussehen ?

17 Datenverteilung in Parallelen DBS

Voraussetzung für die Nutzung von Datenparallelität ist eine geeignete Datenverteilung, so daß mehrere Prozesse auf disjunkten Datenbereichen parallel arbeiten können. Zu unterstützen sind dabei beide Arten der E/A-Parallelität, also Zugriffs- und Auftragsparallelität. Während in Shared-Everything- und in Shared-Disk-Systemen lediglich eine Verteilung der Daten über mehrere Platten zu finden ist, erfordert Shared-Nothing zugleich eine Verteilung der Daten unter den Verarbeitungsrechnern. Die Datenverteilung hat in dieser Architektur daher auch direkten Einfluß auf den Kommunikations-Overhead und ist daher von besonderer Bedeutung für die Leistungsfähigkeit.
Wir konzentrieren uns daher weitgehend auf die Bestimmung der Datenverteilung für Shared-Nothing, die ähnlich wie für Verteilte DBS die Schritte der Fragmentierung und Allokation erfordert (Kap. 5). Um jedoch eine effektive Parallelisierung erreichen zu können, sind diese Aufgaben enger aufeinander abzustimmen. Insbesondere empfiehlt sich vor der Fragmentierung bereits die Festlegung des Verteilgrades einer Relation. Die eigentliche Allokation (Zuordnung von Fragmenten zu Rechnern) ist danach relativ einfach möglich. Nach der Diskussion dieser drei Teilschritte untersuchen wir noch, inwieweit eine replizierte Datenhaltung von Interesse ist. Am Ende des Kapitels wird dann kurz auf die Datenverteilung bei Shared-Everything und Shared-Disk eingegangen.

17.1 Bestimmung des Verteilgrades

Der Verteilgrad (degree of declustering) D einer Relation legt fest, über wieviele Partitionen (Rechner, Platten) diese verteilt wird. Er ist von entscheidender Bedeutung für die Leistungsfähigkeit, da vor allem Selektionsoperationen häufig auf allen D Partitionen auszuführen sind. Bei der Bestimmung von D sind teilweise widersprüchliche Forderungen wie Unterstützung von Intra-Transaktionsparallelität, geringer Kommunikationsaufwand sowie Lastbalancierung zu berücksichtigen (Kap. 5.7). So ist zur Unterstützung von Intra-Transaktionsparallelität prinzipiell eine "breite" Verteilung einer Relation über viele Knoten wünschenswert.

Dies wird etwa mit dem einfachen Ansatz des "full declustering" erreicht, bei dem eine Relation über alle N Knoten bzw. Platten verteilt wird (D=N). Für sehr große Relationen könnte ein kleinerer Verteilgrad auch eine ungünstige Lastbalancierung verursachen, da nicht die Kapazität des gesamten Systems genutzt würde und in den Rechnern, auf die die Bearbeitung beschränkt wird, verstärkte Behinderungen für andere Transaktionen hervorgerufen würden. Auf der anderen Seite führt jedoch ein hoher Verteilgrad zwangsweise zu einem hohen Kommunikationsaufwand zum Starten von Teiloperationen sowie zum Zurückliefern von Ergebnissen. Dieser Aufwand kann für kleinere Relationen sowie für selektive Anfragen, die z.B. lediglich einen Satz als Ergebnis liefern, prohibitiv teuer sein. Für solche Relationen sollte daher eine Allokation zu einer Teilmenge der Rechner erfolgen. Selektive Anfragen (auch auf großen Relationen) sollten auf möglichst wenige Knoten beschränkt werden können (Unterstützung von Lokalität).

Die Diskussion zeigt, daß unterschiedliche Anfragetypen i.a. unterschiedliche Verteilgrade erfordern. Bei der Datenverteilung muß daher für Shared-Nothing-Systeme ein Kompromißwert festgelegt werden, der eine "durchschnittliche" Anfrage gut bedient. Um den optimalen Verteilgrad für eine bestimmte Anfrage bestimmen zu können, ist eine Abschätzung ihrer Antwortzeit in Abhängigkeit des Verteilgrades, der bei Shared-Nothing meist auch dem Parallelitätsgrad p entspricht, vorzunehmen. Nach [WFA93] läßt sich für den Einbenutzerbetrieb die Antwortzeit R einer über p Rechner parallelisierten Anfrage unter bestimmten Annahmen mit folgender Formel charakterisieren*:

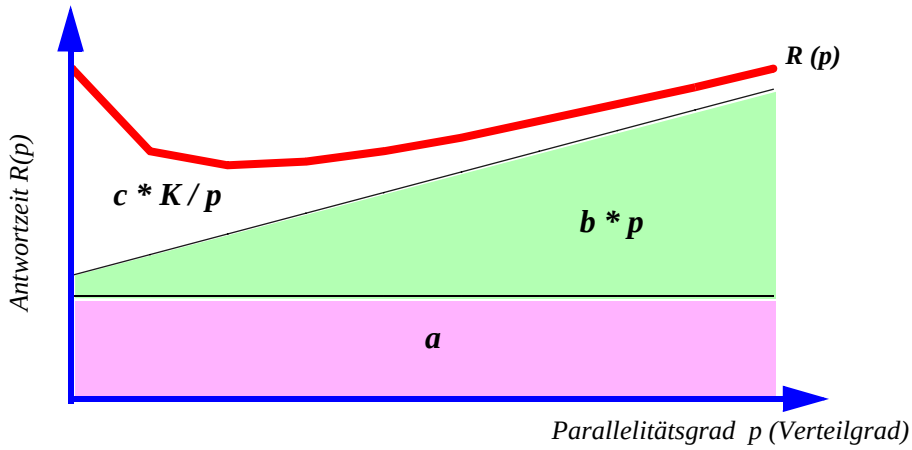
$$R(p) = a + b \times p + \frac{c \times K}{p}$$

Demnach setzt sich die Antwortzeit aus drei Komponenten zusammen, die in Abb. 17-1 separiert dargestellt sind. Der konstante Anteil a umfaßt die nicht parallelisierbaren Anteile der Anfrage (Initialisierungskosten u.ä.). Die zweite Antwortzeitkomponente entspricht der Zeit zum Starten und Beenden der p Teiloperationen; dieser Aufwand steigt i.a. linear mit dem Parallelisierungs- und Verteilgrad. Die eigentlichen Nutzarbeit wird durch $c * K$ abgeschätzt und soll proportional zu der Anzahl zu verarbeitender Tupel K sein. Die Bearbeitungsdauer dieses Anteil wird durch die Parallelisierung idealerweise linear verkürzt. Die Koeffizienten a, b, c und K sind u.a. abhängig von DB- und Anfragemerkmalen, wie Relationengröße, Nutzbarkeit eines Indextyps, Selektivität der Anfrage etc. Weiterhin gehen die benötigten Instruktionen für gewisse Operationen (Kommunikation, etc.), sowie Hardware-Merkmale (CPU-, Platten-, Netzwerkgeschwindigkeit) ein. Auf die Einzelheiten der Festlegung soll hier verzichtet werden; eine genaue Beschreibung für Selektionsoperationen (mit und ohne Indexnutzung) sowie für Join-Anfragen findet sich in [Ma93].

---

* Dabei wird insbesondere eine gleichmäßige Verteilung der Daten unter den Partitionen sowie eine gleichmäßige Referenzverteilung unterstellt.

Abb. 17-1: Antwortzeitzusammensetzung in Abhängigkeit vom Verteilgrad



Wurde für eine Anfrage in dieser Weise eine Antwortzeitabschätzung vorgenommen, läßt sich der optimale Verteilgrad dadurch bestimmen, indem man berechnet, welcher p-Wert die Antwortzeitfunktion R (p) minimiert. Durch Bilden der ersten Ableitung von R (p) und Gleichsetzen mit Null ergibt sich:

$$p_{opt} = \sqrt{\frac{c \times K}{b}}$$

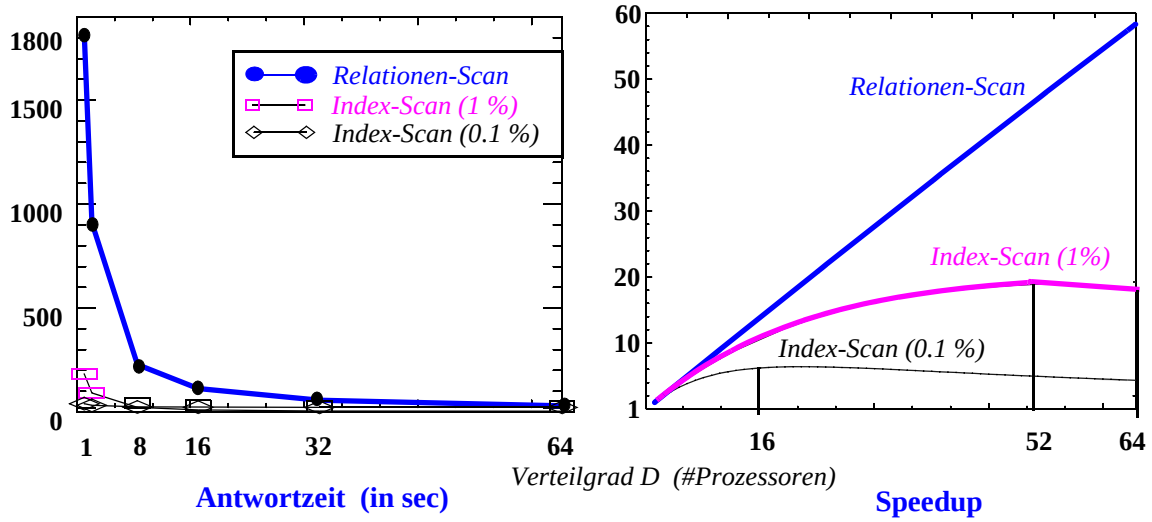
Die Formel zeigt, daß das Optimum unabhängig vom konstanten Anteil a ist, sondern im wesentlichen vom Verhältnis zwischen der Nutzarbeit $c * K$ und dem Parallelisierungs-Overhead b bestimmt wird. Um den Verteilgrad D der Relation festzulegen, bestimmt man in der skizzierten Weise den optimalen Wert p_{opt} für die wichtigsten Anfragetypen. Daraus wird dann ein gewichteter Mittelwert gebilden, bei dem sich das Gewicht eines Anfragetyps aus seinem geschätzten Anteil an der Gesamtlast ergibt.

Beispiel 17-1

Abb. 17-2 zeigt die mit den Formeln aus [Ma93] bestimmte Antwortzeitentwicklung sowie die zugehörigen Speedup-Werte für drei unterschiedliche Selektionsoperationen auf einer Relation von 1 Million Tupel. Man erkennt, daß die Parallelisierung für die verschiedenen Anfragetypen unterschiedlich effektiv ist und unterschiedliche Optima bezüglich des Verteilgrades bestehen. Besonders effektiv ist die Parallelisierung, wenn kein Index genutzt werden kann (Relationen-Scan); hier wird ein nahezu linearer Speedup erzielt. Für eine unterstellte Prozessoranzahl von 64 gilt für diesen Anfragetyp $p_{opt} = 64$. Im Falle der Index-Scans kann die Verarbeitung auf die sich qualifizierenden Tupel beschränkt werden, so daß bessere Antwortzeiten als beim Relationen-Scan erreicht werden. Hier sinkt der Nutzen der Parallelisierung mit zunehmender Selektivität der Anfrage; im Beispiel gilt p_{opt}=52 (1% Selektivität) bzw. p_{opt} = 16 (0,1%). Wenn die beiden Index-Scans jeweils 40% der Zugriffe auf die Relation ausmachen und für 20% der Zugriffe ein vollständiger Relationen-Scan erforderlich ist, ergibt sich als gewichteter Mittelwert

D = 0,4*16 + 0,4*52 + 0,2*64 = 40.

Abb. 17-2: Bestimmung des optimalen Verteilgrades für drei Anfragetypen



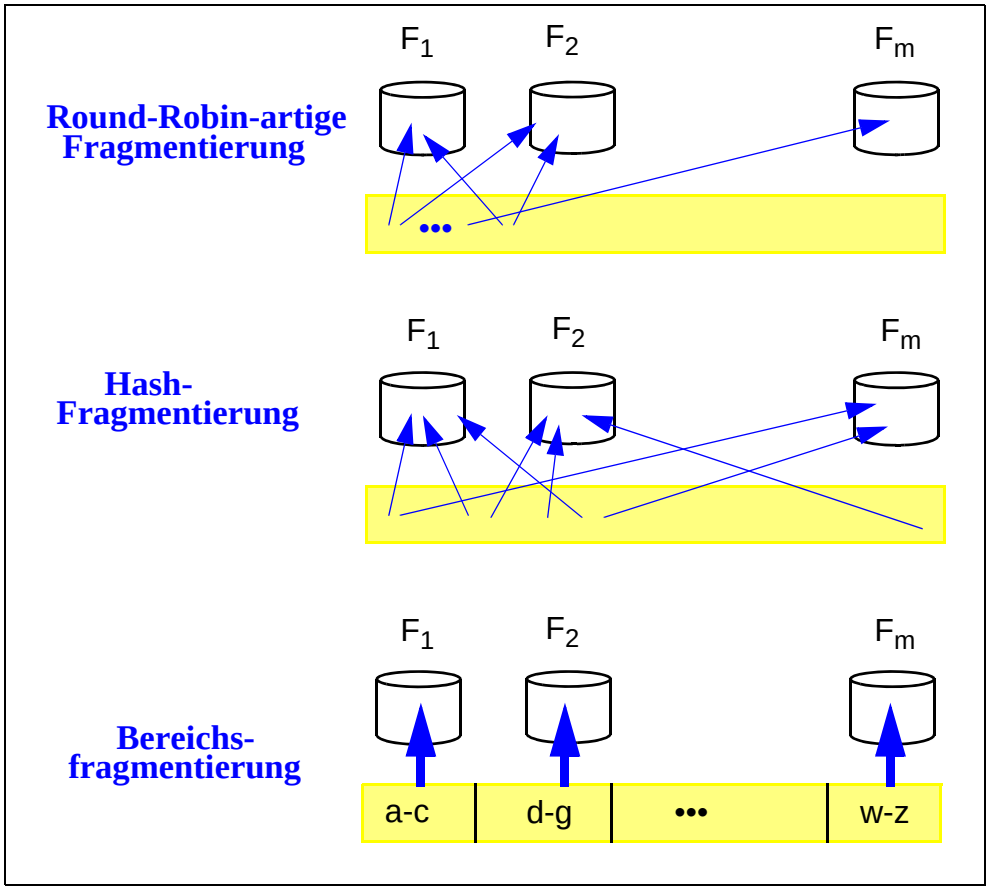
17.2 Fragmentierung

Alle parallelen Shared-Nothing-DBS basieren auf einer horizontalen Fragmentierung von Relationen, da diese eine disjunkte Zerlegung der Daten definiert, die unmittelbar für Datenparallelität genutzt werden kann (Kap. 5.3.3). Ferner läßt sich dabei im Gegensatz zur vertikalen Fragmentierung auch leicht eine sehr große Anzahl von Fragmenten und somit eine weitgehende Parallelisierung erreichen. Im wesentlichen finden drei Fragmentierungsansätze Anwendung für parallele Shared-Nothing-Systeme: Round-Robin, Hash-Fragmentierung sowie Bereichsfragmentierung (Abb. 17-3), die nachfolgend näher diskutiert werden. Wir nehmen dabei an, daß M Fragmente zu bestimmen seien, wobei im allgemeinen M=D gilt*. Die Zuordnung der Fragmente zu Rechnern ist Aufgabe der Allokation (s.u.).

Bei der *Round-Robin-artigen Fragmentierung* werden die Sätze einer Relation reihum den M Fragmenten zugewiesen, so daß der i-te Satz der Relation dem Fragment (i MOD M) + 1 zugeordnet wird. Damit wird eine Aufteilung der Relation in M gleich große Fragmente erreicht. Dies führt zu einer günstigen Lastbalancierung, falls die einzelnen Sätze der Relation mit gleicher Wahrscheinlichkeit referenziert werden. Da für die Datenverteilung keine Attributwerte berücksichtigt werden, ist jedoch i.a. für jede Anfrage auf der Relation eine Bearbeitung aller M Fragmente erforderlich.

* Es kann jedoch auch eine weitergehende Fragmentierung vorgenommen werden (M > D), um größere Freiheitsgrade zur Bildung von Partitionen zu erhalten.

Abb. 17-3: Alternativen zur horizontalen Fragmentierung [DG92]



Die Ansätze der Hash- und Bereichsfragmentierung versuchen diesen Nachteil abzuschwächen, indem sie die Datenverteilung über die Werte eines Attributs (bzw. einer Menge von Attributen) definieren. Dieses *Fragmentierungs-* bzw. *Verteilattribut* ist beliebig wählbar, jedoch wird meist der Primärschlüssel verwendet. Bei der *Hash-Fragmentierung* wird die Zerlegung der Relation durch eine Hash-Funktion auf dem Verteilattribut festgelegt, die jeden Satz auf eines der M Fragmente abbildet. Die Verteilung der Attributwerte sowie die Güte der Hash-Funktion bestimmen, ob eine ähnlich gleichmäßige Datenverteilung wie bei Round-Robin erreicht werden kann. Von Vorteil ist, daß die Bearbeitung von exakten (exact match) Anfragen bezüglich des Verteilattributes, welche nur Sätze zu einem festgelegten Attributwert als Ergebnis liefern, durch Anwendung der Hash-Funktion auf einen Rechner beschränkt werden kann (minimaler Kommunikationsaufwand). Auch für Join-Berechnungen, bei denen das Join-Attribut mit dem Verteilattribut übereinstimmt, ergeben sich signifikante Einsparmöglichkeiten (s. Kap. 18.3.2).

Die *Bereichsfragmentierung* verwendet disjunkte und vollständige Wertebereichsunterteilungen auf dem Verteilattribut anstelle einer Hash-Funktion. Damit lassen sich wie für eine Hash-Fragmentierung Exact-Match-Anfragen bezüg-

lich des Verteilattributes auf ein Fragment (1 Rechner, 1 Platte) beschränken. Daneben können aber auch Bereichsanfragen (range queries) bezüglich des Verteilattributs auf die relevante Teilmenge der Fragmente beschränkt werden, so daß für sie ein geringerer Kommunikationsaufwand als bei Hash-Fragmentierung möglich wird. Ein weiterer Vorteil liegt darin, daß i.a. auch bei ungleichmäßiger Werteverteilung eine Zerlegung in etwa gleich große Fragmente erreichbar ist [DNSS92]. Andererseits erfordert die Bereichsfragmentierung genaue Kenntnisse über die Werteverteilungen und ist daher i.a. schwieriger festzulegen als eine Hash-Fragmentierung.

Beispiel 17-2

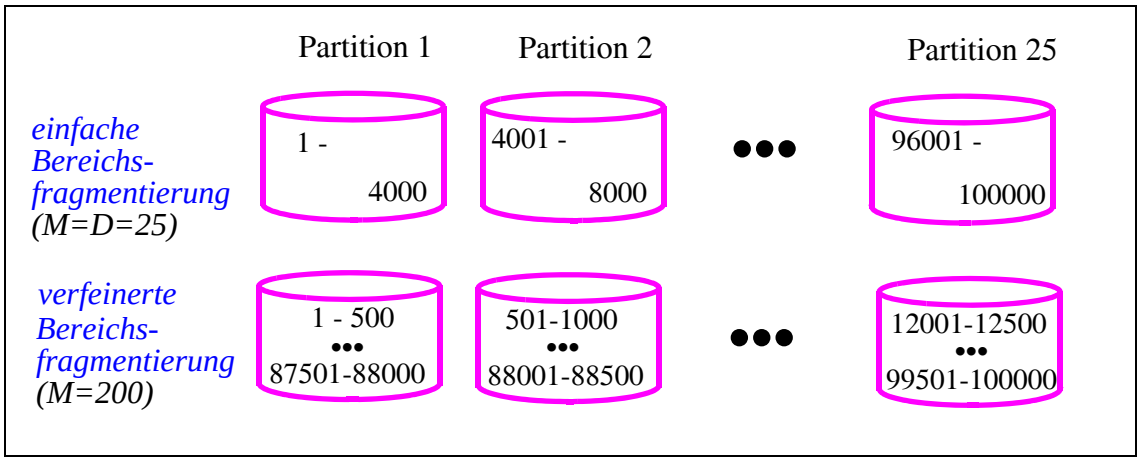
Die KONTO-Relation einer Bankanwendung soll über den Primärschlüssel KTONR in 25 Fragmente zerlegt werden. Im Falle einer Hash-Fragmentierung könnte dies z.B. durch die Hash-Funktion h (KTONR) = 1 + (KTONR MOD 25) erfolgen. Eine Bereichsfragmentierung muß dagegen auf die vergebenen Werte für KTONR abgestimmt werden, um sicherzustellen, daß jede Kontonummer berücksichtigt wird und die einzelnen Fragmente möglichst gleichviele Sätze umfassen. Exact-Match-Anfragen (z.B. KTONR = 4711) können in beiden Fällen auf ein Fragment beschränkt werden. Bereichsanfragen über die Kontonummer (z.B. KTONR < 200.000) lassen sich lediglich bei der Bereichsfragmentierung auf eine Teilmenge der Fragmente eingrenzen, was vor allem für selektive Anfragen vorteilhaft ist. Für die Hash-Fragmentierung wird in diesem Fall der Zugriff auf alle 25 Fragmente erforderlich. Kontenzugriffe über den Namen des Kontoinhabers (anstatt über die Kontonummer) müssten in beiden Fällen auf allen Fragmenten bearbeitet werden.

Im Falle der Bereichsfragmentierung kann die Beschränkung auf M=D Fragmente eine ungünstige Parallelisierung und Lastbalancierung für Bereichsanfragen verursachen. Denn damit werden Bereichsanfragen auf dem Fragmentierungsattribut auf die minimale Anzahl von Prozessoren $s*D$ beschränkt, wobei s der mittlere Anteil sich qualifizierender Tupel sei ($0 \leq s \leq 1$). Wenn für diese Bereichsanfragen jedoch $p_{opt} > s*D$ gilt, wird zwar ein geringer Kommunikationsaufwand, aber aufgrund einer zu geringen Parallelisierung u.U. eine schlechtere Antwortzeit als bei Verarbeitung auf D Knoten wie bei der Hash-Fragmentierung erreicht. Eine in [GD90] beschriebene Abhilfemöglichkeit besteht darin, eine Verfeinerung der Fragmentierung vorzunehmen, so daß die mittlere Ergebnismenge von $s*K$ Tupeln (K = Kardinalität der Relation) auf p_{opt} Fragmente verteilt wird. Dies kann durch eine Fragmentierung erreicht werden, bei der im Mittel etwa $s*K/p_{opt}$ Tupel pro Fragment entfallen, so daß insgesamt $M = p_{opt}/s$ Fragmente zu bestimmen sind. Werden "aufeinanderfolgende" Fragmentbereiche jeweils verschiedenen Prozessoren zugeordnet, kann so für Bereichsanfragen auf dem Fragmentierungsattribut die optimale Parallelität erreicht werden. Der Preis für diese Flexibilität gegenüber einer einfachen Bereichsfragmentierung (M=D) und einer Hash-Fragmentierung liegt vor allem in der aufwendigen Bestimmung der Fragmentierung sowie der aufgrund der potentiell sehr großen Anzahl von Fragmenten teureren Verwaltung der Verteilungsinformation.

Beispiel 17-3

Für die KONTO-Relation aus dem vorherigen Beispiel sei K=100.000, und für Bereichsanfragen über KTONR seien s = 0.05 und p_{opt}=10. Ferner soll D=25 gelten. Bei der einfachen Bereichsfragmentierung (Abb. 17-4 oben) sowie der Hash-Fragmentierung werden somit 25 Fragmente mit jeweils etwa 4000 Konten bestimmt. Die einfache Bereichsfragmentierung begrenzt Bereichsanfragen auf KTONR, welche im Mittel 5000 Sätze betreffen, auf 2 Fragmente (Prozessoren), während bei der Hash-Fragmentierung alle 25 Prozessoren involviert werden. Bei der verfeinerten Fragmentierung (Abb. 17-4 unten) werden M= 10/0.05 = 200 Fragmente mit jeweils etwa 500 Tupeln gebildet. Diese werden dann wie gezeigt reihum den 25 Prozessoren zugeordnet, so daß jede Partition 8 Fragmente umfaßt. Bereichsanfragen auf KTONR werden somit im Mittel auf 10 bis 11 Prozessoren verteilt, so daß der optimale Parallelitätsgrad unterstützt wird.

Abb. 17-4: Einfache vs. verfeinerte Bereichsfragmentierung



Eine Einschränkung bei Hash- sowie Bereichsfragmentierung ist, daß nur Anfragen bezüglich des Verteilattributs auf eine Teilmenge der Fragmente (Rechner) eingeschränkt werden können, für alle anderen Anfragen dagegen alle Fragmente zu involvieren sind. Eine Verbesserungsmöglichkeit bietet eine *mehrdimensionale Bereichsfragmentierung*, bei der mehrere Attribute gleichberechtigt zur Definition der Fragmentierung verwendet werden [GDQ92]. Über ein relativ kompaktes (Grid-)Directory kann dabei für jedes der beteiligten Fragmentierungsattribute festgestellt werden, welche Fragmente relevante Tupel für eine Bereichs- oder Exact-Match-Anfrage enthalten. Damit lassen sich für Anfragen auf mehreren Attributen Kommunikationseinsparungen gegenüber einer Verarbeitung auf allen Fragmenten erzielen. Auf der anderen Seite ergibt sich bezogen auf ein Attribut ein Mehraufwand an Kommunikation gegenüber einer eindimensionalen Fragmentierung auf diesem Attribut. Daher ist eine mehrdimensionale Bereichsfragmentierung i.a. nur sinnvoll, wenn über mehrere Attribute in etwa gleich häufig zugegriffen wird.

Beispiel 17-4

Abb. 17-5 zeigt eine zweidimensionale Bereichsfragmentierung der KONTO-Relation über die Attribute KTONR und KNAME (Kundenname). Diese erlaubt Exact-Match- und Bereichsanfragen bezüglich jedes der beiden Attribute auf 5 Rechner zu begrenzen, falls jedes der Fragmente einem eigenen Rechner zugeordnet ist. Eine einfache Bereichsfragmentierung könnte Anfragen für eines der Attribute auf 1 Rechner eingrenzen, würde jedoch für Anfragen bezüglich des anderen Attributes alle 25 Rechner involvieren. Wenn beide Attribute mit gleicher Häufigkeit referenziert werden, ergibt dies einen Durchschnitt von 13 Rechnern pro Anfrage gegenüber 5 für die zweidimensionale Bereichsfragmentierung. Erfolgen die Kontenzugriffe dagegen in 90% der Fälle über KTONR und nur in 10% über den Kundennamen, schneidet die eindimensionale Fragmentierung über KTONR besser ab (im Mittel 3,4 Knoten pro Anfrage gegenüber 5).

Abb. 17-5: Mehrdimensionale Bereichspartitionierung (Beispiel)

KTONR

KNAME	< 20.000	< 40.000	< 60.000	< 80.000	< 100.000
A-E	1	2	3	4	5
F-J	6	7	8	9	10
K-O	11	12	13	14	15
P-S	16	17	18	19	20
T-Z	21	22	23	24	25

17.3 Allokation

Aufgabe der Allokation ist die Zuordnung der Fragmente zu Partitionen und Rechnern. Diese Aufgabe ist nun einfacher lösbar als bei Verteilten DBS, wo aufgrund großer Entfernungen zwischen den Knoten die Unterstützung einer hohen Lokalität ein Hauptziel war. Dazu war es vor allem erforderlich, die Datenzuordnung auf die an den verschiedenen Knoten gestarteten Transaktionstypen abzustimmen (Kap. 5.7). In lokal verteilten Shared-Nothing-DBS können dagegen eintreffende Transaktionen und Anfragen mit gleichem Aufwand an jedem der Rechner gestartet werden. Als Hauptziele sind neben der Begrenzung des Kommunikationsaufwandes vor allem Intra-Transaktionparallelität sowie Lastbalancierung zu unterstützen. Bei der Festlegung des Verteilgrades sowie der Fragmentierung wurde diesen Faktoren bereits Rechnung getragen, wobei jedoch vor allem eine Minimierung von Antwortzeiten angestrebt wurde. Bei der Allokation soll nun die Lastbalancierung unterstützt werden, um ein gutes Durchsatzverhalten zu fördern.

Bei der Allokation einer Relation sind zunächst D Rechner auszuwählen, denen die Fragmente der Relation zugeordnet werden. Dies ist für D=N trivial, anderenfalls erfolgt die Auswahl mit Hinblick auf eine Lastbalancierung, so daß auf jeden Rech-

ner möglichst gleich viele Zugriffe entfallen. Wenn von einer Gleichverteilung der Zugriffe ausgegangen werden kann, genügt es hierzu, jedem Rechner in etwa gleich viele Tupel zuzuweisen. Anderenfalls müssen die Zugriffshäufigkeiten explizit berücksichtigt werden (s. [CABK88]). Bei der Auswahl der Rechner können auch weitere Faktoren eine Rolle spielen, z.B. wenn zur Unterstützung der Join-Verarbeitung zusammengehörige Fragmente verschiedener Relationen derselben Partition zugewiesen werden sollen. Die Zuordnung der M Fragmente unter den D ausgewählten Prozessoren ist für M=D wiederum trivial. Für M>D, was bei einer Bereichsfragmentierung sinnvoll sein kann, genügt i.a. eine Round-Robin-artige Zuordnung (ähnlich wie in Abb. 17-4).

Die Verteilung einer Relation über D Rechner verlangt auch eine analoge Partitionierung von Indexstrukturen. Dabei wird pro Partition ein eigener Teilindex geführt, der alle Schlüsselwerte enthält, die in den Sätzen des jeweiligen Knotens vorkommen. Die Verwendung und Wartung dieser Indexstrukturen ist damit wie für zentralisierte DBS möglich.

17.4 Replikation

Ähnlich wie in Verteilten DBS (Kap. 9) kann in lokal verteilten Shared-Nothing-Systemen die Fehlertoleranz gegenüber Rechnerausfällen durch replizierte Speicherung der Daten erhöht werden. Im lokalen Fall spielen jedoch die Robustheit gegenüber Netzwerk-Partitionierungen sowie die Unterstützung von geographischer Zugriffslokalität eine untergeordnete Rolle. Dafür soll jetzt die Replikation auch zur schnellen Behandlung von Plattenfehlern sowie zur Verbesserung der Lastbalancierung genutzt werden. Wir stellen dazu im folgenden drei Ansätze vor, die in Produkten bzw. Prototypen realisiert wurden: Spiegelplatten, verstreute Replikation sowie verkettete Replikation. Alle drei Ansätzen verwenden für replizierte Daten jeweils zwei Kopien, so daß sich der Speicherplatzbedarf verdoppelt*. Zur Aktualisierung der Replikate gehen wir von einer einfachen Write-All-Strategie aus, bei der beide Kopien jeweils parallel geändert werden.

Spiegelplatten

Der Einsatz von Spiegelplatten (mirrored disks, shadowed disks) ist bereits in zentralisierten DBS zur schnellen Behandlung von Plattenfehlern weitverbreitet. Dabei wird eine "logische" Platte physisch durch zwei Platten mit identischem Inhalt realisiert. Im Falle eines Plattenfehlers kann die Verarbeitung mit der überlebenden Kopie nahezu unterbrechungsfrei fortgesetzt werden**. Schreibzugriffe wer-

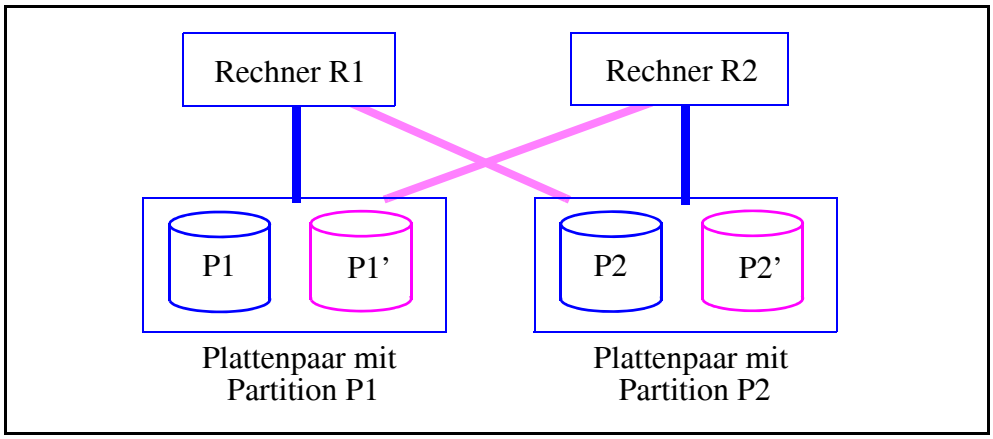
* Eine Verallgemeinerung auf mehr als zwei Kopien ist leicht möglich, wird in der Praxis aus Kostengründen jedoch nicht verfolgt.

** Der traditionelle Ansatz zur Platten-Recovery über Archivkopien und Archiv-Log erfordert dagegen manuelle Eingriffe und kann zu langen Unterbrechungszeiten führen.

den parallel auf beiden Kopien ausgeführt, so daß sich i.a. nur eine geringe Erhöhung der Schreibdauer verglichen mit einer einfachen Platte ergibt. Auf der anderen Seite können Lesezugriffe erheblich beschleunigt werden, da stets die Platte mit der geringsten Armpositionierungszeit verwendet werden kann [Bi89]. Alternativ dazu kann eine Lastbalancierung unterstützt werden, indem Lesezugriffe zur Reduzierung von Wartezeiten gleichmäßig auf beide Platten verteilt werden. Spiegelplatten erlauben somit eine Verdoppelung der E/A-Rate und Bandbreite für Lesezugriffe. Die Verwaltung der Spiegelplatten erfolgt i.a. transparent für das DBS durch Betriebssystem-Software (Dateisystem) oder durch die Platten-Controller.

Diese Eigenschaften von Spiegelplatten bleiben auch beim Einsatz innerhalb von Shared-Nothing-Systemen erhalten, wie er etwa in Tandem-Systemen seit langem praktiziert wird [Ka78]. Dabei ist jedes Plattenpaar mit zwei Rechnern verbunden, wobei jedoch zu jedem Zeitpunkt nur von einem Rechner aus auf ein Plattenpaar zugegriffen wird (fett markierte Verbindung in Abb. 17-6). Es liegt somit keine rechnerübergreifende Replikation der Daten vor, und Lese- und Schreiboperationen sowie die Behandlung von Plattenfehlern können wie im zentralen Fall abgewickelt werden. Nach Ausfall eines Rechners R1 wird dessen Partition P1 vollständig von dem Rechner R2 übernommen, der mit den Platten von R1 verbunden ist. Damit bleibt der Zugriff auf P1 weiterhin möglich (nachdem R2 die Crash-Recovery für R1 durchgeführt hat). Allerdings ist während der Ausfallzeit mit einer ungünstigen Lastbalancierung zu rechnen, da der übernehmende Rechner R2 nun die Zugriffe auf zwei Partitionen zu verarbeiten hat. Für diesen Rechner ist somit eine Überlastung sehr wahrscheinlich. Auf der anderen Seite wird eine sehr hohe Datenverfügbarkeit unterstützt, da der Zugriff auf Partition P1 erst dann unmöglich wird, wenn sowohl R1 als auch R2 ausfallen.

Abb. 17-6: Einsatz von Spiegelplatten bei Shared-Nothing (Tandem)



Die auf jeweils einen Rechner beschränkte Nutzung von Spiegelplatten bedeutet, daß diese Form der Replikation keine Vorteile für die Crash-Recovery mit sich

bringt. Denn die Übernahme der Partition eines ausgefallenen Rechners durch einen überlebenden Rechner ist offenbar auch ohne Spiegelplatten anwendbar. Wäre dagegen die Kopie einer Partition bereits im Normalbetrieb einem anderen Knoten zugeordnet, würde die Aktualisierung der entfernten Kopie Kommunikation erfordern und damit teurer werden. Die Nutzung beider Kopien zur Lastbalancierung müßte zudem durch das DBS unterstützt werden und wäre wesentlich aufwendiger zu realisieren als die rechnerlokale Lastbalancierung für lesende Plattenzugriffe.

Verstreute Replikation

Zur Crash- und Platten-Recovery wird im Shared-Nothing-System von Teradata der Ansatz der verstreuten Replikation (interleaved declustering) verfolgt [Te83]. Damit soll vor allem nach Ausfall eines Rechners eine bessere Lastbalancierung erreicht werden als durch vollständige Übernahme einer Partition durch einen zweiten Rechner. Voraussetzung dazu ist, daß die Kopien zu Daten einer Partition auf mehrere andere Knoten verteilt werden. Im Teradata-System ist es dazu möglich, Rechner-Gruppen von jeweils G Knoten zu bilden. Die Kopien zu Daten eines Rechners R werden dann gleichmäßig unter den G-1 anderen Rechnern der Gruppe verteilt, zu der R gehört. Nach Ausfall des Rechners verteilen sich die Zugriffe dann gleichmäßig unter diesen Knoten (sofern eine Gleichverteilung der Zugriffe vorliegt), so daß pro Rechner lediglich eine Mehrbelastung um den Faktor G/G-1 eintritt. Nach einem Rechnerausfall bleiben sämtliche Daten verfügbar; selbst Mehrfachfehler können toleriert werden, solange verschiedene Gruppen betroffen sind, d.h. nicht mehr als ein Knoten pro Gruppe ausfällt. Teradata unterstützt Gruppengrößen von 2 bis 16 Knoten; Relationen können jedoch über mehrere Gruppen verteilt werden (D≥ G).

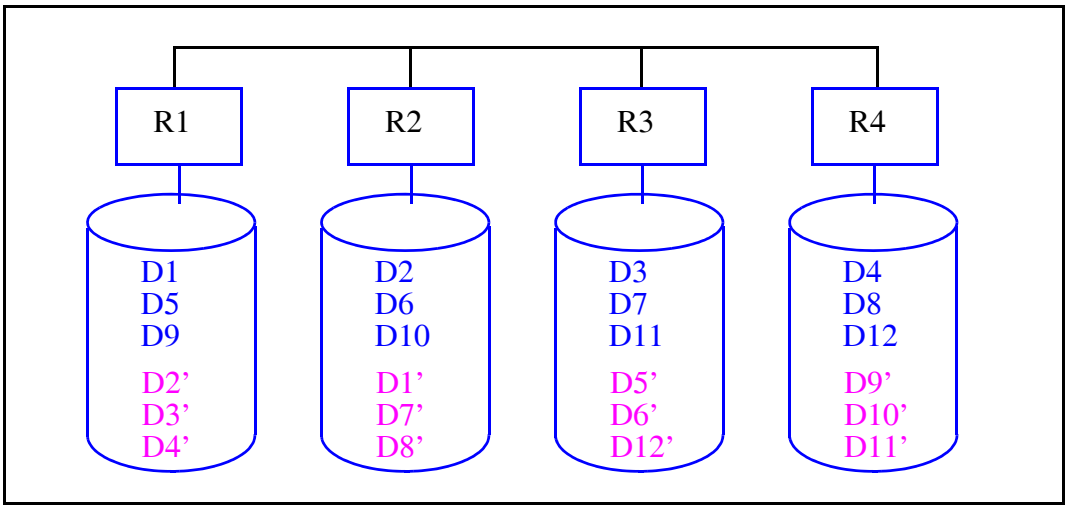
Beispiel 17-5

Abb. 17-7 zeigt ein Beispiel für "Interleaved Declustering" und G=D=4. Die Datenmengen Di bezeichnen dabei einzelne Sätze bzw. Fragmente einer Partition, Di' entspricht der Kopie von Di. Man erkennt, daß die Kopien jeder Partition gleichmäßig über die drei anderen Knoten der Gruppe verteilt sind. Fällt z.B. Rechner R2 aus, so können die Zugriffe auf dessen Daten von den drei überlebenden Rechnern R1, R3 und R4 fortgeführt werden. Erst bei einem weiteren Rechnerausfall kann auf eine Teilmenge der Daten nicht mehr zugegriffen werden

Die Einführung von Gruppen erlaubt einen flexiblen Kompromiß zwischen dem Grad der Lastbalancierung im Fehlerfall und der Datenverfügbarkeit. Denn eine große Gruppe (z.B. G=D) erlaubt eine breite Verteilung der Last des ausgefallenen Rechners. Dafür besteht jedoch eine relativ hohe Wahrscheinlichkeit, daß zwei Rechner gleichzeitig ausfallen, woraufhin dann eine Datenpartition nicht mehr erreichbar ist. Umgekehrt besteht für kleine Gruppen eine sehr hohe Verfügbarkeit,

jedoch eine ungünstigere Lastbalancierung. So ergibt sich im Extremfall G=2 das ungünstige Lastbalancierungsverhalten von Spiegelplatten.

Abb. 17-7: Datenverteilung mit verstreuter Replikation (G=D=4).



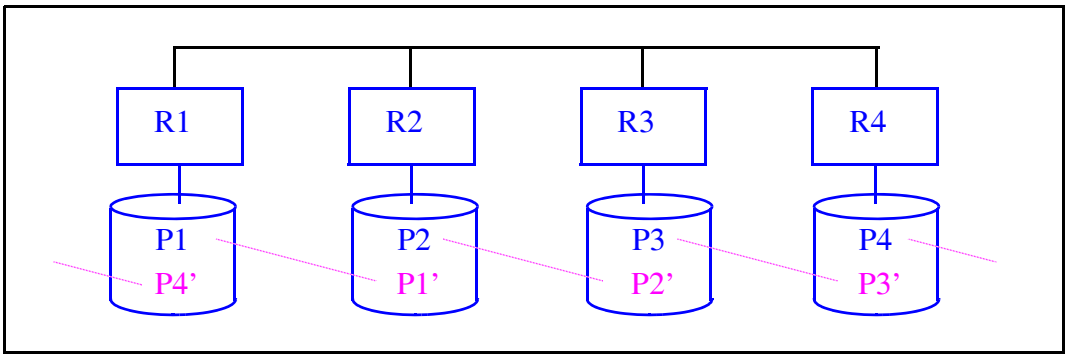
Eine Beschränkung der Teradata-Implementierung liegt darin, daß die Kopien nur im Fehlerfall verwendet werden. Damit wird die Replikation im Gegensatz zu den Spiegelplatten im Tandem-Ansatz also nicht zur Leistungsverbesserung (Lastbalancierung) im Normalbetrieb genutzt. Die effektive Verwendung der Kopien im Normalbetrieb würde von den DBS eine Auswahl der Verarbeitungsrechner unter Berücksichtigung der aktuellen Auslastung erfordern (dynamische Lastbalancierung).

Verkettete Replikation

Der Ansatz der verketteten Replikation (chained declustering), der im Rahmen des Gamma-Projekts untersucht wurde [HD90], versucht die hohe Verfügbarkeit von Spiegelplatten sowie die günstige Lastbalancierung der verstreuten Replikation zu vereinen. Dabei ist es wie bei der verstreuten Replikation möglich, die D Partitionen einer Relation auf mehrere Gruppen von je G Rechnern zu verteilen. Die Datenverteilung innerhalb einer Gruppe erfolgt jetzt jedoch wie in Abb. 17-8 illustriert. Dabei ist die Kopie einer Partition Pi dem jeweils "nächsten" Knoten in der Gruppe, Rj, zugeordnet, wodurch eine logische Verkettung der Knoten entsteht*. Der Vorteil liegt darin, daß Mehrfachausfälle in einer Gruppe im Gegensatz zur verstreuten Replikation nicht notwendigerweise die Datenverfügbarkeit reduzieren, sondern nur dann, wenn zwei benachbarte Knoten ausfallen. Fällt z.B. in der Konfiguration von Abb. 17-8 Rechner R2 aus, dann bleiben auch nach Ausfall von R4 alle Daten erreichbar. Wie bei Teradata werden die Kopien Pi' im Normalbetrieb nicht zur Lastbalancierung eingesetzt, sondern nur im Fehlerfall genutzt.

* Bei fortlaufender Numerierung der Knoten gilt j = (i MOD G) + 1.

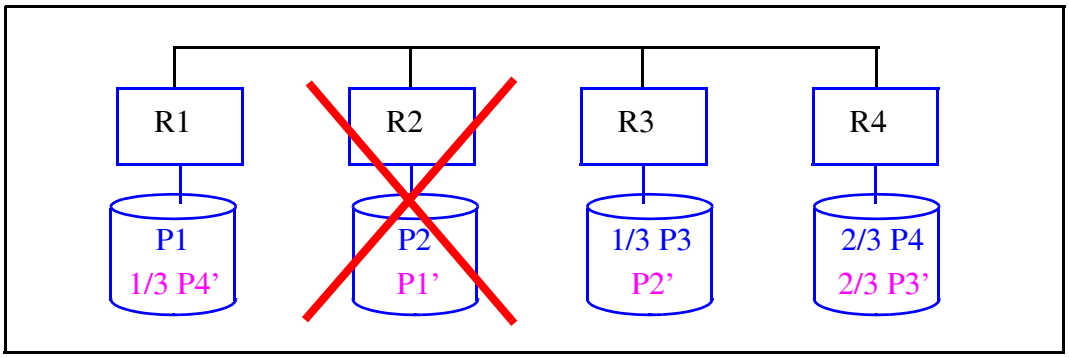
Abb. 17-8: Datenverteilung mit verketteter Replikation (G=D=4)



Nach Ausfall eines Rechners mit Partition Pi ist es mit dieser Datenverteilung notwendig, daß der logisch nächste Rechner Rj sämtliche Pi-Zugriffe auf der Kopie Pi' bearbeitet. Damit dadurch nicht eine ähnlich schlechte Lastbalancierung wie bei Spiegelplatten verursacht wird, sieht der Ansatz der verketteten Replikation vor, daß die Lesezugriffe auf den sonstigen Partitionen so unter den verbleibenden Rechnern der Gruppe umverteilt werden, daß sich eine in etwa gleichmäßige Belastung aller Rechner ergibt.

Die prinzipielle Vorgehensweise dazu ist in Abb. 17-9 für das Beispiel aus Abb. 17-8 veranschaulicht. Damit nach dem Ausfall von Rechner R2 eine gleichmäßige Auslastung der verbleibenden drei Rechner erreicht wird, darf die Zugriffshäufigkeit jedes Rechner nur um etwa ein Drittel steigen. Da Rechner R3 bereits sämtliche Zugriffe auf der Kopie von P2, P2', auszuführen hat, werden nun die Zugriffe auf P3 auf R3 und R4 verteilt, so daß nur noch ein Drittel der Zugriffe durch R3 erfolgt und zwei Drittel durch R4 (auf der Kopie P3'). In ähnlicher Weise werden die Zugriffe auf P4 auf die zwei Rechner R4 und R1 verteilt. Diese Änderungen der Zugriffsverteilung erfordern keine Umverteilung der Daten, sondern lediglich eine Anpassung von Verteilungsinformation. So ist in Abb. 17-9 die Partition P3 weiterhin vollständig an den Knoten R3 und R4 gespeichert, und beide Kopien werden bei jeder Änderung aktualisiert. Die Objekte von P3 werden jedoch intern zwei Sub-Partitionen zugeordnet, die ein bzw. zwei Drittel von P3 umfassen. Lesezugriffe auf Objekte der ersten Sub-Partition werden damit von R3, die auf der zweiten Sub-Partition von R4 bearbeitet. Bei einer Gleichverteilung der Zugriffe innerhalb von Partitionen und bei gleicher Zugriffsfrequenz pro Partition kann damit eine gleichmäßige Lastverteilung erreicht werden.

Abb. 17-9: Verkettete Replikation: Lastbalancierung im Fehlerfall



17.5 Datenverteilung bei Shared-Everything und Shared-Disk

Für Shared-Everything und Shared-Disk bezieht sich die Festlegung der Datenverteilung lediglich auf die Platten, nicht jedoch auf die einzelnen Prozessoren. Der Verteilgrad einer Relation bestimmt daher nicht in dem Maße wie bei Shared-Nothing den Overhead zur Parallelisierung, da alle Prozessoren ohne Kommunikationsverzögerung auf Daten aller Platten zugreifen können. Somit ist auch die Verteilung einer großen Relation über alle Platten eher vertretbar, um ein Maximum an E/A-Parallelität zu unterstützen. Für Relationen-Scans kann die hohe E/A-Parallelität einfach genutzt werden, unabhängig von der verwendeten Fragmentierungsstrategie. Dabei ist zur Vermeidung von Plattenengpässen lediglich darauf zu achten, daß pro Anfrage nicht mehrere Prozessoren von einer Platte lesen. Dies gestattet dennoch die Möglichkeit, die Anzahl von Prozessoren (den Parallelitätsgrad) p dynamisch zu wählen. Denn bei einem Verteilgrad D muß lediglich gelten

$$p * k = D,$$

wobei k die Anzahl der Platten angibt, die pro Teilanfrage zu bearbeiten ist. So kann für eine über 100 Platten verteilte Relation ein Relation-Scan von p= 1,2, 4, 5, 10, 20, 25, 50 oder 100 Teilanfragen parallel bearbeitet werden, während bei Shared-Nothing der Parallelitätsgrad i.a. durch den Verteilgrad (100) bestimmt ist.

Die Wahl eines hohen Verteilgrades geht bei Shared-Everything und Shared-Disk auch nicht zu Lasten selektiver Anfragen, die etwa nur eine Seite betreffen. Denn sofern ein Index zur Bestimmung der relevanten Seite genutzt werden kann, läßt sich die Bearbeitung problemlos auf einen Prozessor sowie eine Platte beschränken. Generell ist es bei diesen Architekturen möglich, Anfragen zur Minimierung des Kommunikationsaufwandes auf einen Prozessor zu beschränken.

Problematischer ist dagegen die Parallelisierung von Operationen, welche nur eine Teilmenge einer Relation betreffen, deren sequentielle Bearbeitung jedoch zu hohe Antwortzeiten verursacht. Die Schwierigkeit liegt in der Bestimmung einer

Parallelisierung, welche garantiert, daß parallele Sub-Transaktionen nicht auf die selben Platten zugreifen. Hierzu ist eine dem DBS bekannte, logische Datenverteilung zu wählen, etwa auf Basis einer Hash- oder Bereichsfragmentierung. Damit kann das DBS wenigstens Anfragen auf dem Verteilattribut wiederum auf eine Teilmenge der Platten beschränken und sicherstellen, daß höchstens eine Sub-Transaktion auf eine Platte zugreift.

Beispiel 17-6

Eine Relation soll über folgende Bereichspartitionierung auf Attribut A über 100 Platten verteilt sein:

A: (1 - 10.000; 10.001 - 20.000; 20.001 - 30.000; ... 990.001 - 1.000.000)

Eine Bereichsanfrage für A-Werte zwischen 70.000 und 220.000 kann damit von 15 (5, 3, 1) parallelen Teilanfragen bearbeitet werden, die jeweils 1 (3, 5, 15) der 100 Platten bearbeiten.

Die diskutierten Replikationsformen Spiegelplatten, verstreute und verkettete Replikation können für Shared-Everything und Shared-Disk zur schnellen Behandlung von Plattenfehlern genutzt werden [CK89, WZ93]. Die Replikation kann dabei für logische Objekte (Dateien, Sätze) oder auf physischer Ebene (Segmente, Seiten) außerhalb des DBS realisiert werden. Bei physischer Verwaltung der Kopien durch die Platten-Controller könnten auch bei verstreuter bzw. verketteter Replikation die Kopien (ähnlich wie für Spiegelplatten) einfach zur Lastbalancierung im Normalbetrieb genutzt werden [WZ93].

Übungsaufgaben

Aufgabe 17-1: Bestimmung des Verteilgrades

Für eine Relation von 10 Millionen Tupeln soll der Verteilgrad für ein Shared-Nothing-System mit 100 Knoten bestimmt werden. Es sollen im wesentlichen zwei Typen von Operationen auf der Relation ausgeführt werden: Relationen-Scans (30% der Operationen) sowie Index-Scans mit einer mittleren Selektivität von 1%. Bestimmen Sie den "optimalen" Verteilgrad für die Relation, wenn gilt a = 50 ms, b = 5 ms und c = 0,05 ms (siehe Kap. 17.1 zur Bedeutung von a, b und c).

Aufgabe 17-2: Verteilattribut

Warum empfiehlt sich meist die Wahl des Primärschlüssels als Verteilattribut?

Aufgabe 17-3: Verteilinformationen in Indexstrukturen

Eine Beschränkung der vorgestellten Fragmentierungsalternativen liegt darin, daß nur Anfragen auf den Fragmentierungsattributen auf eine Teilmenge der Rechner eingegrenzt werden kann. Inwieweit wäre es für andere Attribute sinnvoll, die Speicherungs-

orte der einzelnen Werte in einer erweiterten Indexstruktur zu vermerken, um damit Anfragen bezüglich dieser Attribute auf wenige Knoten beschränken zu können ?

Aufgabe 17-4: Verstreute Replikation

Geben Sie analog zu Abb. 17-7 ein Beispiel zur verstreuten Replikation an mit D=8 und G=4 (zwei Gruppen mit je 4 Rechnern).

Aufgabe 17-5: Verkettete Replikation

Die Daten D1, D2, ... D12 einer Relation seien wie in Abb. 17-7 über vier Knoten verteilt (G=D=4). Wie sieht die Zuordnung der Kopien Di' im Falle der verketteten Replikation aus? Welche Zugriffsverteilung nach Ausfall von Rechner R3 garantiert eine gleichmäßige Lastbalancierung ? Geben Sie dazu für jedes Di bzw. Di' den Rechner an, der die Zugriffe darauf bearbeitet.

18 Parallele Anfragebearbeitung

Wir diskutieren zunächst allgemeine Aspekte der parallelen Anfragebearbeitung gegenüber der Anfragebearbeitung in Verteilten DBS. Danach stellen wir parallele Implementierungen der wichtigsten relationalen Operatoren vor, insbesondere von Selektion, Projektion, Sortierung (Kap. 18.2) und Join (Kap. 18.3). Abschließend diskutieren wir einige aktuelle Probleme der parallelen Anfragebearbeitung.

18.1 Allgemeine Vorgehensweise

Die Anfragebearbeitung in Parallelen DBS weist große Ähnlichkeiten mit der Anfrageverarbeitung in Verteilten DBS (Kap. 6) auf. Insbesondere fallen zur Erstellung eines Ausführungsplans die drei Phasen der Anfragetransformation, Daten-Lokalisierung und Optimierung an. Die Berücksichtigung unterschiedlicher Parallelisierungsarten betrifft vor allem die Phase der Optimierung. Dabei kann die in Verteilten DBS aufgrund der geforderten Knotenautonomie vorgenommene Trennung zwischen globaler und lokaler Optimierung entfallen. Dennoch ist die Erstellung eines guten parallelen Ausführungsplanes weitaus komplexer als die Bestimmung sequentieller Pläne. Denn dabei ist für jeden Operator die Parallelisierungsstrategie festzulegen, welche Entscheidungen über den Parallelitätsgrad, die Zuteilung von Betriebsmitteln (Prozessoren, Hauptspeicher) sowie der Implementierungsstrategie verlangt. Weiterhin wird die Festlegung der Ausführungsreihenfolge von der Parallelisierung stark beeinflußt, z.B. um ein möglichst hohes Maß an Inter-Operatorparallelität nutzen zu können.

Um die Komplexität der Query-Optimierung und damit die Optimierungsdauer zu reduzieren, wird häufig ein zweistufiger Ansatz verfolgt, bei dem für eine DB-Operation zunächst ein optimaler sequentieller Ausführungsplan (Operatorbaum) erstellt wird, für den dann in einem zweiten Schritt die Parallelisierung vorgenommen wird. Dieser Ansatz vereinfacht zwar die Optimierung, kann aber natürlich zu suboptimalen Lösungen führen [Pi90]. Andere Ansätze reduzieren die Komplexität durch strukturelle Beschränkungen des Suchraumes, z.B. in dem nur lineare

Operatorbäume berücksichtigt werden, die jedoch auch den erreichbaren Grad an Inter-Operatorparallelität begrenzen. Ein anderer Ansatz zur Reduzierung des Optimierungsaufwandes besteht darin, die Suchstrategie einzuschränken. In kommerziellen DBS wird nämlich meist eine nahezu vollständige Bewertung aller möglichen Pläne (exhaustive search) vorgenommen, z.B. mit Verfahren der dynamischen Programmierung [Se79]. Der Aufwand solch enumerativer Methoden zur Optimierung paralleler Ausführungspläne ist jedoch für komplexere Operationen, die z.B. mehr als 10 Relationen betreffen, nicht mehr akzeptabel. Zur Reduzierung des Aufwandes wurden zahlreiche heuristische und zufallsgesteuerte Suchstrategien vorgeschlagen [LVZ93]. Wir werden einige dieser Ansätze im Zusammenhang mit Mehr-Wege-Joins diskutieren (Kap. 18.3.4).

Als Kostenfunktion zur Optimierung empfiehlt sich die Abschätzung der Antwort- bzw. Bearbeitungszeit, da deren Verkürzung Ziel von Intra-Transaktionsparallelität ist. Dazu sind die Kommunikationsanteile in Abhängigkeit der vorliegenden Datenverteilung (bei Shared-Nothing) sowie die CPU- und E/A-bezogenen Kosten zu bestimmen. Zur Übersetzungszeit lassen sich allerdings lediglich Antwortzeiten im Einbenutzerbetrieb abschätzen, die sich im Mehrbenutzerbetrieb aufgrund von Behinderungen durch parallel zu bearbeitende Transaktionen und Anfragen stark erhöhen können. Auch Entscheidungen wie Parallelitätsgrad und Prozessorallokation sollten im Mehrbenutzerfall idealerweise vom aktuellen Systemzustand abhängig gemacht werden, was dynamische Ansätze für Parallelisierung und Scheduling verlangt.

18.2 Parallelisierung unärer relationaler Operatoren

Im folgenden diskutieren wir die Parallelisierung von Selektion, Projektion, Aggregatfunktionen sowie der Sortierung. Wir gehen dabei von einer horizontalen Zerlegung der Relationen über mehrere Platten/Knoten aus. Verfahren zur parallele Join-Berechnung werden im nächsten Kapitel (Kap. 18.3) vorgestellt.

18.2.1 Selektion

Die Selektion (Restriktion) ist der mit Abstand am häufigsten auszuführende Operator, so daß seiner Parallelisierung eine besonders hohe Bedeutung für die Leistungsfähigkeit eines Parallelen DBS zukommt. Kann die Suchbedingung durch einen Index auf eine Teilmenge der Relation eingegrenzt werden, spricht man von einem *Index-Scan*; im Extremfall wird kein oder nur ein Tupel referenziert (z.B. Exact-Match-Anfrage auf dem Primärschlüssel). Anderenfalls ist ein *Relationen-Scan* erforderlich, der den Zugriff auf jedes Tupel der Relation erfordert. Im folgenden betrachten wir die Parallelisierung für die verschiedenen Architekturtypen.

Für Shared-Nothing ist die Parallelisierung der Selektion sehr einfach und durch die Datenverteilung bestimmt. Denn wenn eine Relation horizontal in die n Partitionen R1, R2, ... Rn aufgeteilt ist, gilt:

$$R = \cup\, Ri \qquad (1 \le i \le n)$$

und

$$\sigma_P (R) = \cup\, \sigma_P (Ri) \qquad (1 \le i \le n).$$

Die Selektion kann also parallel auf den n *Datenknoten* der Relation, denen jeweils eine Partition zugeordnet ist, durchgeführt werden. Das Gesamtergebnis ergibt sich durch die anschließende Vereinigung der n Teilergebnisse. Im Falle einer Hash- oder Bereichsfragmentierung lassen sich bestimmte Anfragen auf dem Verteilattribut auf eine Teilmenge der Rechner beschränken (s. Kap. 17.2). Alle anderen Selektionen sind jedoch auf sämtlichen n Datenknoten auszuführen, auch im Falle eines Index-Scans. Der Index-Scan reduziert in diesem Fall lediglich die pro Partition auszuwertende Satzmenge, jedoch nicht die Kommunikationskosten zum Starten der n Teilanfragen, zum Zurücksenden der Teilergebnisse sowie zur Teilnahme am Commit-Protokoll. Für selektive Anfragen kann dieser Kommunikationsaufwand im Vergleich zur eigentlichen Nutzarbeit sehr ungünstig werden.

Bei Shared-Everything und Shared-Disk ist die Parallelisierung von Selektionsoperationen auf die Datenverteilung auf Platte abzustimmen, um zu verhindern, daß parallel auszuführende Teilselektionen auf die selben Platten zugreifen (Kap. 17.5). Der große Vorteil im Vergleich zu Shared-Nothing liegt darin, daß der Parallelitätsgrad durch die Datenverteilung noch nicht vorgegeben ist. Vielmehr kann dieser je nach Anfragetyp oder Systemauslastung variiert werden. So können selektive Anfragen zur Minimierung des Parallelisierungs-Overheads stets sequentiell auf einem Prozessor bearbeitet werden. Relationen-Scans dagegen lassen sich problemlos auf mehreren Prozessoren parallel bearbeiten, wobei die Datenverteilung (Anzahl der Platten, auf denen die Relation allokiert wurde) lediglich den maximalen Parallelitätsgrad vorgibt. Im Gegensatz zu Shared-Nothing, wo Selektionsoperationen auf Basisrelationen stets auf den Datenknoten ausgeführt werden, sind bei Shared-Everything und Shared-Disk die ausführenden Rechner nicht durch die Datenverteilung vorbestimmt, sondern frei wählbar. Dies ergibt insbesondere hohe Freiheitsgrade zur dynamischen Lastbalancierung.

Beispiel 18-1

Die KONTO-Relation einer Bank sei über 25 Partitionen (Platten) verteilt; ferner sei ein Index auf KTONR und KNAME (Kundenname) definiert. Exact-Match-Anfragen zu beiden Attributen können für Shared-Everything und Shared-Disk auf jeweils einem Prozessor und der minimalen Plattenanzahl bearbeitet werden. Bei Shared-Nothing erfordert dagegen der Zugriff über KNAME die Involvierung aller Rechner, falls die Datenverteilung über KTONR definiert wurde (Beispiel 17-2). Selbst im Falle einer zweidimensionalen Bereichsfragmentierung sind bei Shared-Nothing mehrere Knoten zu involvieren (im Mittel 5 nach Beispiel 17-4), so daß ein deutlich höherer Kommunikationsaufwand entsteht. Relationen-Scans, z.B. um die Summe aller Kontostände zu berechnen, können auch bei Shared-Disk und Shared-Everything parallel auf allen Platten durchgeführt werden.

Shared-Everything erlaubt einen geringeren Overhead zum Starten der Teilanfragen und zum Mischen der Teilergebnisse als Shared-Disk und Shared-Nothing. Dafür ist dort der Parallelitätsgewinn durch die i.a. recht geringe Prozessoranzahl beschränkt. Für Shared-Disk kann gegenüber Shared-Nothing ggf. Kommunikationsaufwand eingespart werden, indem Teilergebnisse nicht über Nachrichten zurückgeliefert, sondern in temporären Dateien auf den gemeinsamen Platten (bzw. in einem gemeinsamen Halbleiter-Speicherbereich) abgelegt werden. Dies empfiehlt sich für große Ergebnismengen, die im Empfangsrechner nicht vollständig im Hauptspeicher gehalten werden können, so daß eine Speicherung auf Externspeicher ohnehin erforderlich ist. Für kleinere Zwischenergebnisse ist dagegen auch bei Shared-Disk die Rücklieferung über das Kommunikationssystem vorzuziehen.

18.2.2 Projektion

Die Parallelisierung dieses Operators kann analog zur Selektion erfolgen. Denn für eine in n horizontale Partitionen zerlegte Relation R gilt:

$$\pi_A (R) = \cup\ \pi_A (Ri) \qquad (1 \leq i \leq n).$$

In vielen Fällen werden Projektion und Selektion ohnehin zusammen ausgeführt, um die relevanten Daten nur einmal von Platte zu lesen und den Umfang der Ergebnismengen zu reduzieren. Bei der Bearbeitung einer DB-Operation werden beide Operatoren meist als erstes durchgeführt (auf den Basisrelationen), da sie zu einer starken Reduzierung der Datenmenge führen, die ggf. durch andere Operatoren weiterzuverarbeiten ist.

Die Projektion verlangt ggf. eine Eliminierung von Duplikaten, die i.a. durch Sortieren der Ergebnismenge erfolgt (s.u.). Alternativ dazu kann die Erkennung von Duplikaten hash-basiert erfolgen [Gra93]. Dabei wird jedes Tupel über eine Hash-Funktion in eine Hash-Tabelle abgebildet. Die Erkennung von Duplikaten beschränkt sich damit auf die Tupel einer Hash-Klasse, was bei wenigen Tupeln pro Hash-Klasse (sehr viele Hash-Klassen) billig möglich ist. Durch Aufteilung der Hash-Klassen unter mehrere Prozessoren kann die Duplikateliminierung zudem leicht parallelisiert werden.

18.2.3 Aggregatfunktionen

Die Berechnung von Aggregat- bzw. Built-In-Funktionen (MIN, MAX, SUM, COUNT, AVG) ist bei horizontaler Partitionierung ebenfalls leicht parallelisierbar. Dazu nehmen wir an, daß Q (R) ein Attribut der Relation R sei, auf dem die Built-In-Funktion anzuwenden sei. Die parallele Berechnung von Extremwerten (MIN, MAX) ist dann stets problemlos möglich, da gilt:

$$MIN(Q(R)) = MIN(MIN(Q(R_{1)}), \ldots, MIN(Q(R_n)))$$

und

$$MAX(Q(R)) = MAX(MAX(Q(R_{1)}), \ldots, MAX(Q(R_n))).$$

Das bedeutet, daß auf jeder Partition Ri parallel das lokale Minimum bzw. Maximum berechnet werden kann. Abschließend wird dann aus diesen lokalen Ergebnissen das globale Minimum bzw. Maximum bestimmt. Die parallele Berechnung von SUM, COUNT und AVG ist in analoger Weise möglich, wenn keine Duplikateliminierung erforderlich ist:

$$SUM(Q(R)) = \Sigma\ SUM(Q(R_i))$$

$$COUNT(Q(R)) = \Sigma\ COUNT(Q(R_i))$$

$$AVG(Q(R)) = SUM(Q(R)) / COUNT(Q(R))$$

Anderenfalls ist, wie bei der Projektion diskutiert, zuerst die Entfernung von Duplikaten vorzunehmen.

18.2.4 Parallele Sortierung

Die Sortierung stellt einen häufig benötigten Operator dar. Sie wird nicht nur bei der sortierten Ausgabe von Ergebnismengen notwendig, sondern auch zur Realisierung anderer Funktionen wie Duplikat-Eliminierung oder Join-Berechnung (Sort-Merge-Join). Da die zu sortierenden Datenmengen meist nicht vollständig im Hauptspeicher gehalten werden können, sind für DBS interne Sortierverfahren (z.B. Quicksort) nicht ausreichend. Es werden deswegen externe Verfahren eingesetzt, bei denen Zwischenergebnisse innerhalb von temporären Dateien auf Externspeicher ausgelagert werden. Hierzu wird zunächst in einer *Sortierphase* die Eingabe in mehrere Teile oder Läufe (runs) zerlegt, die sortiert und in temporären Dateien gespeichert werden. Die einzelnen Läufe werden dann in einer *Mischphase* sukzessive zu größeren, sortierten Läufen gemischt, bis schließlich ein einziger Lauf erzeugt ist, der die sortierte Ausgabemenge repräsentiert.

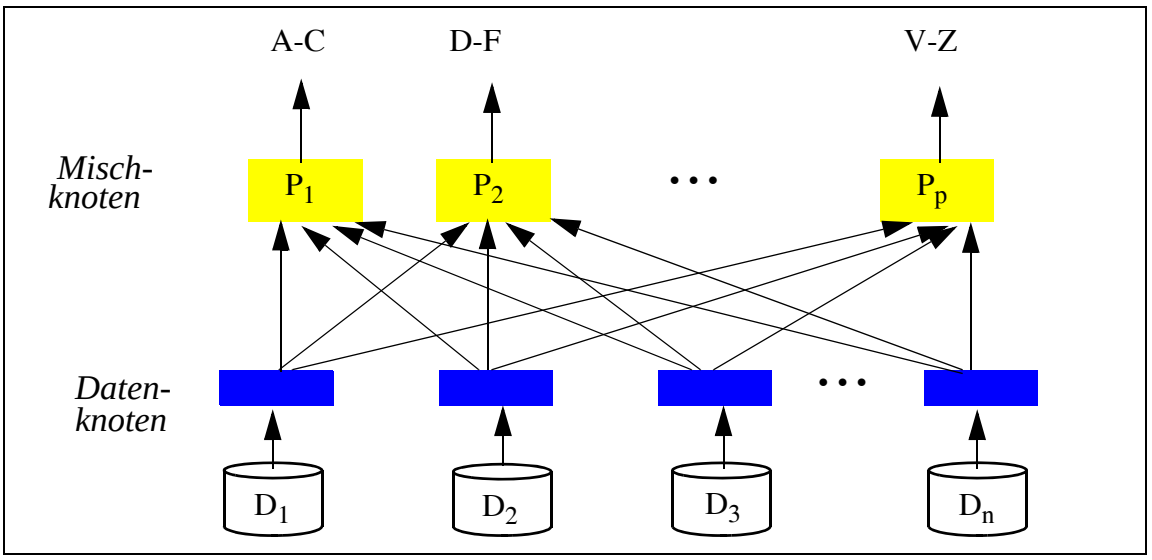
Da die Sortierung in vielen DBS die teuerste Operation überhaupt ist, stellt sie einen besonders geeigneten Kandidaten zur Parallelisierung dar. Zur Nutzung von Datenparallelität sollte dabei die Eingabe bereits über mehrere Partitionen/Rechner verteilt sein (multiple input). Ebenso ist nach [LY89] eine Partitionierung der sortierten Ausgabe (multiple output) sehr wichtig, um die Verzögerung zur Erzeugung eines sequentiellen Ausgabestromes zu vermeiden. Ferner sollten die Sortier- und Mischphasen parallel abgewickelt werden. Zur Reduzierung des Kommunikationsaufwandes sollte dazu jedes Tupel höchstens einmal über das Netzwerk verschickt werden [Gra93]. In der folgenden Diskussion unterstellen wir eine Shared-Nothing-Architektur. Eine Übertragung der Ansätze auf Shared-Disk und Shared-Everything ist jedoch leicht möglich.

Ein einfacher Ansatz zur parallelen Sortierung einer partitionierten Relation sieht vor, die Partitionen an den Datenknoten parallel einzulesen und lokal zu sor-

tieren. Danach werden die so erzeugten Läufe an einen einzigen *Mischknoten* geschickt, wo durch Mischen das sortierte Gesamtergebnis erzeugt wird. Dieser Ansatz hat jedoch den offensichtlichen Nachteil, daß nur die erste Phase parallel arbeitet, während das Mischen sowie die Ergebnisausgabe sequentiell an einem Knoten erfolgen. Dieser Nachteil kann durch folgenden Ansatz behoben werden.

Dabei erfolgt zunächst wieder das parallele Einlesen und Sortieren der verschiedenen Partitionen der Relation, die dann jedoch unter mehrere Mischknoten aufgeteilt werden*. Diese dynamische Datenumverteilung durch Verschicken der Tupel wird über eine (dynamische) Bereichspartitionierung auf dem Sortierattribut gesteuert. Dabei wird für p Mischprozessoren der Wertebereich des Sortierattributs vollständig in p disjunkte Intervalle zerlegt, so daß etwa gleich viel Tupel pro Intervall entfallen. Ein Tupel, dessen Sortierattributwert dem i-ten Intervall angehört, wird dann an den i-ten Mischknoten geschickt. Damit enthält jeder Mischprozessor alle Tupel des ihm zugeordneten Wertebereichsintervalls. Die einzelnen Mischknoten mischen die bei ihnen eingehenden Tupelströme parallel und unterstützen eine partitionierte Ausgabe an den Benutzer. Dabei wird zunächst das sortierte Ergebnis des ersten Mischprozessors bereitgestellt, dann das des zweiten usw. Der Algorithmus, der parallel in allen Phasen arbeitet, ist in Abb. 18-1 veranschaulicht. Dabei wurde die Sortierung auf einem Namensattribut unterstellt. Nicht gezeigt sind Plattenzugriffe für temporäre Dateien, die sowohl an den Daten- als auch an den Mischknoten notwendig werden können.

Abb. 18-1: Dynamische, bereichsbasierte Datenumverteilung zur parallelen Sortierung



In [LY89] wurde eine Verfeinerung dieses Ansatzes vorgestellt, bei dem zur Reduzierung des Kommunikationsaufwandes nicht die vollständigen Tupel zu den

* Das Einlesen von Externspeicher entfällt, wenn die zu sortierenden Daten als Ausgabe zuvor ausgeführter Operatoren noch im Hauptspeicher vorliegen.

Mischknoten geschickt werden, sondern lediglich die Sortierschlüsselwerte sowie die Nummer des zugehörigen Datenknotens. Die Ergebnistupel werden dann von den Datenknoten bereitgestellt, wobei die von den Mischknoten ermittelte Sortierreihenfolge festlegt, von welchem Datenknoten das jeweils nächste Ergebnistupel zu verwenden ist. Zur Bestimmung der dynamischen Bereichsfragmentierung wurde ferner vorgesehen, daß jeder Datenknoten nach der lokalen Sortierung die bei ihm vorliegende Werteverteilung einem Koordinatorknoten mitteilt. Dieser bestimmt aus den lokalen Werteverteilungen eine globale Bereichsfragmentierung, die allen Datenknoten mitgeteilt und zum Verschicken der Daten (Sortierschlüssel) zu den Mischknoten verwendet wird.

Die Beschreibung der parallelen Sortierung verdeutlicht, daß die sequentiellen Basisoperatoren für Sortieren und Mischen in den Daten- und Mischknoten weiterhin zur Anwendung kommen. Die Parallelisierung basiert zum einen auf der partitionierten Datenverteilung zum Lesen der Eingabedaten und zum anderen auf der dynamischen Datenumverteilung zur Parallelisierung des Mischvorgangs. Eine sehr ähnliche Vorgehensweise kann auch zur Parallelisierung anderer Operatoren angewendet werden, insbesondere zur Join-Berechnung (s.u.). Damit ist es auch relativ einfach möglich, aus einem sequentiellen einen parallelen Ausführungsplan zu erzeugen. Im Gamma-System genügten so im wesentlichen zwei zusätzliche Operatoren, Split und Merge, zur Realisierung der parallelen Query-Bearbeitung [DG92]. Die Split-Operation realisiert die Aufteilung eines Datenstromes in mehrere Teilmengen, die verschiedenen Rechnern bzw. Prozessen zugeordnet werden, z.B. über eine Bereichs- oder Hash-Fragmentierung gesteuert. Die Merge-Operation nimmt dagegen das Mischen mehrerer Datenströme vor, die z.B. durch zuvor ausgeführte Operatoren auf verschiedenen Partitionen einer Relation erzeugt werden. Im Volcano-System sind die Split- und Merge-Funktionen durch einen einzigen Operator, dem sogenannten Exchange-Operator, realisiert [Gra94].

18.3 Parallele Joins

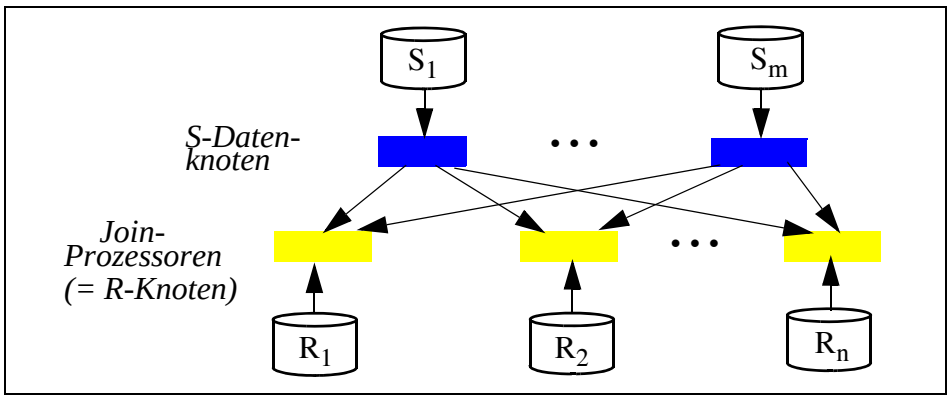
Die Wichtigkeit der Verbundoperation sowie ihre oft aufwendige Berechnung führten dazu, daß eine große Anzahl von parallelen Implementierungen vorgeschlagen wurde. Wir stellen zunächst zwei allgemeine Ansätze zur parallelen Join-Verarbeitung vor, welche auf einer dynamischen Umverteilung der Eingaberelationen basieren und zur lokalen Join-Berechnung unterschiedliche sequentielle Algorithmen zulassen (Nested-Loop-, Sort-Merge- oder Hash-Join). Der erste Ansatz basiert dabei auf einer dynamischen Replikation einer der Relationen (Kap. 18.3.1), während der zweite Ansatz eine dynamische Partitionierung der Eingabedaten vorsieht und nur für Equi-Joins (Gleichverbund) anwendbar ist (Kap. 18.3.2). Danach gehen wir genauer auf parallele Join-Algorithmen ein, die Hash-Joins zur

lokalen Join-Berechnung benutzen (Kap. 18.3.3). Abschließend diskutieren wir die Parallelisierung von Mehr-Wege-Joins (Kap. 18.3.4).

18.3.1 Join-Berechnung mit dynamischer Replikation

Wir betrachten den Verbund zwischen den Relationen R und S, wobei R in n Partitionen R_1, R_2, ..., R_n und S in m Partitionen S_1, S_2, ..., S_m unterteilt sei. Ferner sei S die kleinere der beiden Relationen.

Abb. 18-2: Parallele Join-Berechnung mit dynamischer Replikation von S



a) Prinzip

1. *Koordinator: initiiere Join auf allen R_i (i=1 ... n) und allen S_j (j =1 .. m)*
2. Scan-Phase: *in jedem S-Knoten führe parallel durch:*
   lies lokale Partition S_j und sende sie an jeden Knoten R_i (i= 1...n)
3. Join-Phase: *in jedem R-Knoten mit Partition R_i führe parallel durch (i = 1..n):*
   - *$S := \cup S_j$ (j=1..m)*
   - *berechne $T_i := R_i \bowtie S$ (impliziert Lesen von R_i)*
   - *schicke T_i an Koordinator*
4. *Koordinator: empfange und mische alle T_i*

b) Algorithmus

Wir beschreiben den Ansatz der dynamischen Replikation zunächst für Shared-Nothing. Dabei wird vorgesehen, daß die Join-Berechnung parallel an den n Datenknoten der R-Relation stattfindet und jede S-Partition zur Join-Berechnung an jeden R-Knoten geschickt wird (Abb. 18-2a). Damit wird die kleinere S-Relation an jedem R-Knoten vollständig repliziert. Das Gesamtergebnis ergibt sich durch Vereinigung der lokalen Join-Ergebnisse der R-Knoten. Im einzelnen fallen damit die in Abb. 18-2b gezeigten Schritte an. Nach Starten der Join-Anfrage durch einen Koordinatorknoten erfolgt in einer *Scan-Phase* das Lesen der Eingabe* sowie die Umverteilung der Daten. Danach folgt die *Join-Phase* sowie das Zurückliefern der lokalen Join-Ergebnisse an den Koordinator.

Der Ansatz verursacht offenbar einen sehr hohen Kommunikationsaufwand, da jede S-Partition vollständig an n Rechner zu schicken ist. Dieser Aufwand steigt proportional mit dem R-Verteilungsgrad n sowie der S-Größe und kann somit für große Relationen extrem teuer werden. Zudem entsteht ein hoher Join-Aufwand, da in jedem R-Knoten ein Verbund mit der vollständigen S-Relation vorzunehmen ist. Der Hauptvorteil liegt darin, daß der Ansatz für beliebige Join-Bedingungen anwendbar ist, also nicht nur für Equi-Joins. Die lokale Join-Berechnung in Schritt 3 kann prinzipiell mit jedem sequentiellen Ansatz erfolgen, wobei jedoch für Nicht-Equi-Joins i.a. ein Nested-Loop-Join notwendig wird [ME92]*. Der Ansatz der dynamischen Replikation wird daher gelegentlich auch als *paralleler Nested-Loop-Join* bezeichnet [ÖV91].

Der Algorithmus nutzt Datenparallelität zum parallelen Einlesen der R- und S-Partitionen sowie zur parallelen Join-Berechnung in den R-Knoten. Daneben läßt sich Pipeline-Parallelität zwischen der Scan- und der Join-Phase vorteilhaft einsetzen, insbesondere wenn ein Nested-Loop-Ansatz zur lokalen Join-Berechnung zur Anwendung kommt. Denn das Verschicken einzelner S-Tupel an die R-Rechner kann unmittelbar nach Bearbeitung an den Datenknoten erfolgen, ohne daß also zuvor die gesamte S-Partition gelesen wurde. Ebenso kann die Join-Berechnung in den R-Knoten mit eingehenden S-Tupeln sofort durchgeführt werden, ohne daß auf das Eintreffen aller S-Sätze gewartet wird. Darüber hinaus können neu berechnete Tupel des Verbundergebnisses direkt an den Koordinatorknoten weitergeleitet werden. Die Nutzung der Pipeline-Parallelität reduziert auch die Notwendigkeit, Daten in temporären Dateien zwischenzuspeichern, und erlaubt somit E/A-Einsparungen.

Für *Shared-Everything* kann die Replikation der S-Partition sowie die Kommunikation für Datenumverteilungen entfallen, da Tupel der S-Relation im gemeinsamen Hauptspeicher von allen Prozessoren referenziert werden können. Es genügt somit ein einmaliges Einlesen der S-Tupel, wobei die S-Tupel parallel mit den n R-Partitionen abgeglichen werden. Wenn das mehrfache Lesen eines S-Tupels verhindert werden soll, muß es solange gepuffert bleiben, bis der Vergleich mit allen R-Partitionen beendet ist. Bei annähernd gleichmäßigem Fortgang der Verarbeitung in den Join-Prozessen dürfte der zur S-Pufferung benötigte Speicherumfang relativ gering gehalten werden können. Der eigentliche Join-Aufwand ist für Shared-Everything so hoch wie für Shared-Nothing, da jede R-Partition mit der vollständigen S-Relation zu vergleichen ist.

---

* Als Eingabe für die Join-Verarbeitung kommen anstelle vollständiger Basisrelationen auch die Ergebnisse von zuvor ausgeführten Operatoren in Frage. Insbesondere werden vor der Join-Ausführung i.a. alle anwendbaren Selektionen und Projektionen zur Reduzierung des Kommunikations- und Verarbeitungsumfangs ausgewertet (in der Scan-Phase).

* Beim Nested-Loop-Join wird jedes Tupel der ersten Relation mit jedem Tupel der zweiten Relation verglichen, so daß der Berechnungsaufwand quadratisch mit der Relationengröße zunimmt.

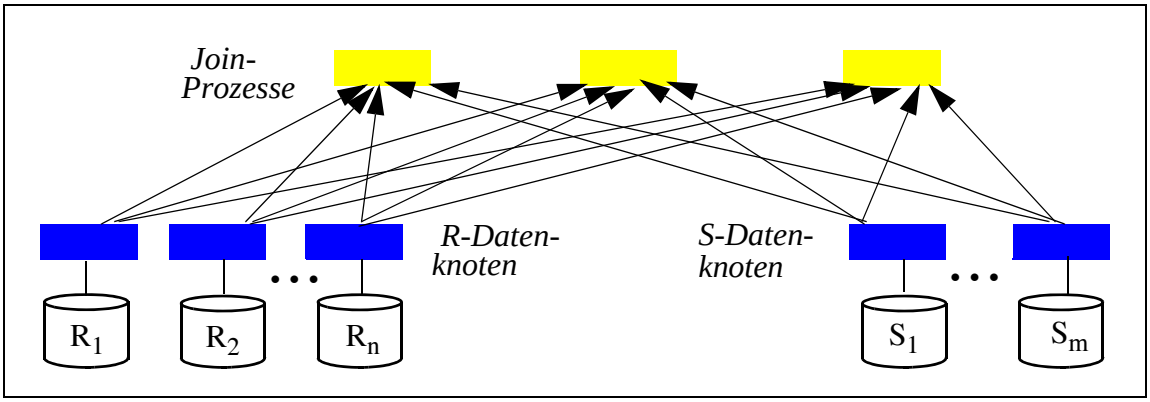
Auch für *Shared-Disk* kann die aufwendige Kommunikation zur Datenumverteilung prinzipiell umgangen werden, da jeder Prozessor direkt auf die S-Platten zugreifen kann. Allerdings erfolgt für Shared-Disk das Lesen von Platte bei nachrichtenbasierter E/A-Schnittstelle auch durch Kommunikation zwischen den Verarbeitungsrechnern sowie E/A-Prozessoren (Platten-Controllern). So kommt die in Abb. 18-2a skizzierte Vorgehensweise auch für Shared-Disk zur Anwendung, wobei lediglich E/A-Prozessoren an die Stelle der S-Datenknoten treten. Der Unterschied zu Shared-Nothing besteht darin, daß dort die S-Datenknoten "aktiv" sind und S-Tupel zur Weiterverarbeitung an die Join-Rechner schicken. Die E/A-Prozessoren bei Shared-Disk dagegen sind "passiv" und liefern die S-Tupel (S-Seiten) erst nach Aufforderung (Lese-E/A) durch die Join-Rechner. Das n-fache Lesen der S-Relation dürfte jedoch ähnlich teuer sein wie die Datenumverteilung bei Shared-Nothing.

Der gleichzeitige Zugriff auf die S-Relation durch mehrere Join-Prozessoren birgt für Shared-Disk zudem die Gefahr von Plattenengpässen. Diese Problem kann durch gezielte Nutzung von Platten-Caches in den E/A-Prozessoren abgeschwächt werden, ähnlich wie durch den gemeinsamen Hauptspeicher bei Shared-Everything. Der E/A-Prozessor liest dabei idealerweise die S-Tupel lediglich einmal von Platte und puffert sie im Platten-Cache, bis alle Join-Prozessoren die S-Tupel angefordert haben. Der Vorteil des Shared-Disk-Ansatzes gegenüber Shared-Nothing liegt darin, daß an den Join-Rechnern ein Zwischenspeichern von S-Tupeln in temporären Dateien definitiv vermieden werden kann. Bei Shared-Nothing dagegen kann dies notwendig werden, wenn die S-Tupel schneller an die Join-Rechner geliefert als sie dort verarbeitet werden. Außerdem ist die Anzahl der Join-Rechner für Shared-Disk prinzipiell frei wählbar, während sie bei Shared-Nothing durch den R-Verteilungsgrad n festgelegt ist.

18.3.2 Join-Berechnung mit dynamischer Partitionierung

Im häufigen Fall des Equi-Joins (Gleichverbund) wird eine parallele Join-Berechnung ohne Replikation einer Relation möglich. Stattdessen genügt eine partitionierte Umverteilung der Eingabedaten, so daß jedes Tupel nur an einen anstatt an n Join-Rechner verschickt wird. Allerdings sind dabei im allgemeinen Fall beide Relationen umzuverteilen, während die Replikation auf die kleinere Relation beschränkt war. Ein Vorteil liegt darin, daß die Anzahl der Join-Prozessoren im allgemeinen Fall frei wählbar ist. Wir beschreiben den Ansatz im folgenden für Shared-Nothing; eine Übertragung auf Shared-Everything und Shared-Disk ist ähnlich möglich wie oben diskutiert. Zunächst betrachten wir den allgemeinen Ansatz mit einer Umverteilung beider Eingaberelationen. Danach werden Spezialfälle diskutiert, wenn für eine oder beide Relationen das Join-Attribut als Verteilattribut vorliegt.

Abb. 18-3: Parallele Join-Berechnung mit dynamischer Partitionierung der Eingaberelationen



a) Prinzip

1. *Koordinator: initiiere Join auf allen R- , S- und Join-Knoten*
2. Scan-Phase
   in jedem R-Knoten führe parallel durch:
   lies lokale Partition R_i und
   sende jedes R-Tupel an zuständigen Join-Rechner

   in jedem S-Knoten führe parallel durch:
   lies lokale Partition S_j und
   sende jedes S-Tupel an zuständigen Join-Rechner
3. Join-Phase: *in jedem Join-Knoten k führe parallel durch (k=1...p):*
   - $R_k' := \cup R_{ik}$ *(i=1..n)* *(Menge der von Join-Knoten k empfangenen R-Tupel)*
   - $S_k' := \cup S_{jk}$ *(j=1..m)* *(Menge der von Join-Knoten k empfangenen S-Tupel)*
   - *berechne* $T_k := R_k' \bowtie S_k'$
   - *schicke* T_k *an Koordinator*
4. *Koordinator: empfange und mische alle* T_k

b) Algorithmus

Wir gehen wiederum von einem Join zwischen R und S mit n bzw. m Partitionen aus. Die Join-Berechnung soll nun parallel auf p Prozessoren stattfinden. Dazu lesen die Datenknoten von R und S zunächst in der Scan-Phase ihre Partitionen parallel ein und verteilen die R- bzw. S-Tupel gemäß derselben Verteilungsfunktion f auf dem Join-Attribut unter den p Join-Prozessoren (Abb. 18-3a). Die Verteilungsfunktion bildet jeden Wert des Join-Attributs auf eine Zahl zwischen 1 und p ab, die den zuständigen Join-Prozessor kennzeichnet. Damit ist gewährleistet, daß R- und S-Tupel mit übereinstimmendem Join-Attributwert demselben Join-Prozessor zugeordnet werden. Der eigentliche Verbund kann somit an den Join-Prozessoren wieder parallel mit einem beliebigen lokalen Verfahren durchgeführt werden (Join-Phase). Das Gesamtergebnis der Join-Operation erhält man erneut durch Mischen der Teilergebnisse aller Join-Prozesse.

Die genannten Schritte sind im Algorithmus in Abb. 18-3b noch einmal dargestellt. In Schritt 3 wurden die an Join-Knoten k aufgrund der Umverteilung entstehenden temporären Partitionen mit R_k' und S_k' (k=1 ... p) bezeichnet, da sie sich anders als die an den Datenknoten vorliegenden Partitionen R_i bzw. S_j (i=1..n; j=1..m) zusammensetzen. Beim Einsatz von Pipeline-Parallelität ist es nicht notwendig, daß die in Schritt 3 gezeigten Vereinigungsoperatoren vor der Join-Berechnung vollständig ausgeführt sind (d.h., alle Eingabetupel vorliegen). Dies wäre jedoch erforderlich, wenn ein Sort-Merge-Ansatz zur lokalen Join-Berechnung verwendet werden soll, da hierfür die Eingaberelationen zunächst sortiert werden müssen*.

Als Verteilungsfunktion zur dynamischen Datenumverteilung kommt eine Hash- oder eine Bereichspartitionierung in Betracht. Die Hash-basierte Verteilung läßt sich über eine Hash-Funktion einfach realisieren und wird in einigen existierenden Systemen verwendet (Teradata, Tandem). Jedoch kann es damit leicht zu Skew-Effekten kommen, wenn eine "schiefe" Werteverteilung für das Join-Attribut vorliegt. Bei bekannter Werteverteilung kann dieser Nachteil durch eine Bereichspartitionierung eher verhindert werden, indem die Wertebereichsintervalle so gewählt werden, daß jeweils etwa gleich viel Tupel pro Partition entfallen. Diese Bestimmung einer solchen Partitionierung ist dafür wieder entsprechend aufwendig [DNSS92].

Der Ansatz der dynamischen Partitionierung weist ein hohes Potential zur Lastbalancierung auf. Denn sowohl die Anzahl der Join-Prozessoren p als auch die Auswahl der Join-Rechner selbst sind dynamisch wählbare Parameter, die in Abhängigkeit des Systemzustandes festgelegt werden können. So wurde in [RM93] festgestellt, daß es sich im Mehrbenutzerbetrieb zur Reduzierung des Kommunikationsaufwandes empfiehlt, die Anzahl der Join-Knoten umso geringer zu wählen, je höher die Prozessoren ausgelastet sind. Hierzu wurde eine Heuristik angegeben, die als Default-Wert für p den optimalen Parallelitätsgrad des Einbenutzerbetriebes verwendet. Dieser wird dynamisch proportional zur mittleren CPU-Auslastung reduziert, v.a. im Auslastungsbereich von über 50%. Zur Join-Verarbeitung selbst wurden daneben die p am geringsten ausgelasteten Rechner verwendet.

Spezialfälle

Der Nachteil des allgemeinen Ansatzes liegt darin, daß beide Eingaberelationen vollständig umverteilt werden müssen, was für große Relationen einen sehr hohen

* Da beim Sort-Merge-Join eine Sortierung der Eingaberelationen auf dem Join-Attribut vorliegt, kann ein Equi-Join durch einmaliges Lesen beider Relationen berechnet werden. Das Lesen erfolgt schritthaltend in beiden Relationen, wobei zu einem Join-Attribut-Wert der ersten Relation jeweils die zugehörigen Treffer in der zweiten Eingabe ermittelt werden [ME92].

Kommunikationsaufwand verursachen kann. Eine deutliche Reduzierung des Kommunikationsumfangs ist jedoch möglich, wenn für eine der beiden Relationen, z.B. R, die Verteilung auf dem Join-Attribut definiert wurde, d.h. das Verteilungsmit dem Join-Attribut übereinstimmt. In diesem Fall ist nur eine Umverteilung der zweiten Relation S notwendig, wenn die Join-Verarbeitung auf den R-Knoten erfolgt. Die Funktion zur dynamischen Umverteilung von S muß dabei mit der Fragmentierungs- und Allokationsstrategie für R übereinstimmen (Hash- oder Bereichspartitionierung), um sicherzustellen, daß zusammengehörige Tupel auch an dem selben R-Rechner verarbeitet werden. Allerdings geht die Reduzierung des Kommunikationsaufwandes einher mit dem Verlust dynamischer Lastbalancierungsmöglichkeiten, da die Join-Rechner nicht mehr frei wählbar sind.

Der Kommunikationsaufwand zur Datenumverteilung wird vollständig umgangen, wenn die Verteilungsattribute beider Relationen mit dem Join-Attribut übereinstimmen und eine übereinstimmende Fragmentierung und Allokation verwendet wird. Dabei müssen die zusammengehörigen Tupel beider Relationen jeweils demselben Datenknoten zugeordnet sein. Dies wird durch eine abgeleitete horizontale Fragmentierung unterstützt (Kap. 5.3).

18.3.3 Parallele Hash-Joins

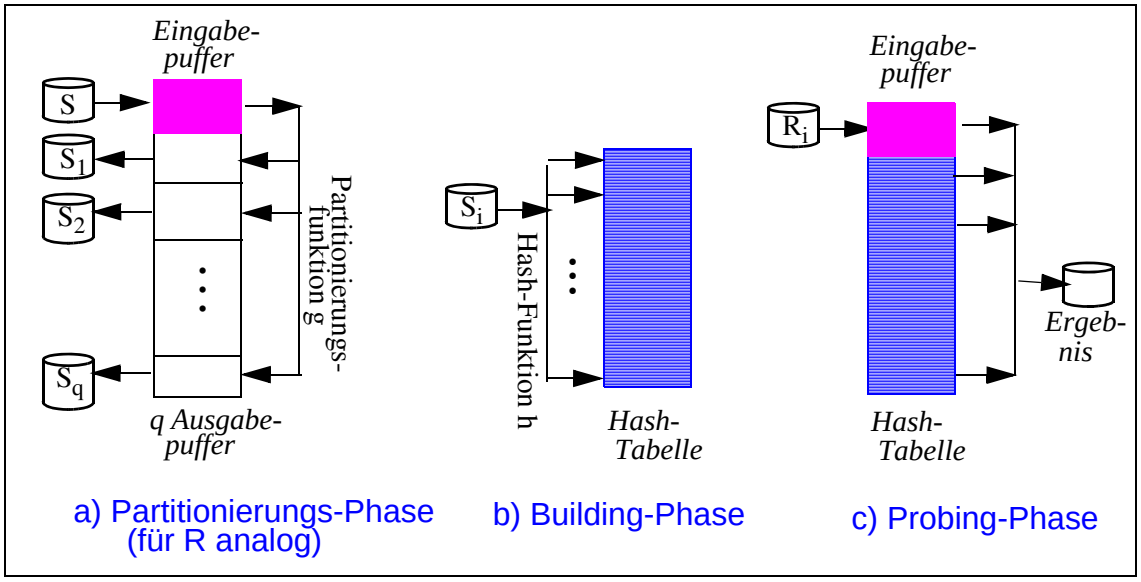
Hash-Joins gestatten eine effiziente Bearbeitung von Equi-Joins, da sie nur einen geringen CPU-Aufwand erfordern und große Hauptspeicher zur E/A-Reduzierung nutzen können. Wir beschreiben zunächst verschiedene Varianten der sequentiellen Hash-Join-Bearbeitung, für die dann eine parallele Realisierung für Shared-Nothing angegeben wird.

Sequentielle Realisierungen

Das *Basisverfahren* des sequentiellen Hash-Joins ist anwendbar, wenn die kleinere (innere) Relation S - nach Anwendung möglicher Selektionen und Projektionen - vollständig im Hauptspeicher gehalten werden kann. In diesem Fall besteht der Hash-Join aus zwei Phasen. In Phase 1 (building phase) wird die innere Relation S in eine Hash-Tabelle im Hauptspeicher gebracht, wobei eine Hash-Funktion h auf das Join-Attribut angewendet wird. Damit wird jedes Tupel von S einer bestimmten Hash-Klasse zugeordnet. In Phase 2 (probing phase) wird die zweite (äußere) Relation R gelesen. Für jeden R-Satz wird dabei wieder die Hash-Funktion h auf das Join-Attribut angewendet, und es wird überprüft, ob in der somit ermittelten Hash-Klasse S-Tupel mit übereinstimmendem Join-Attributwert vorliegen. Diese Bestimmung der Verbundpartner erfordert nur den Vergleich mit den S-Tupeln einer Hash-Klasse, so daß bei einer großen Anzahl von Hash-Klassen der Suchraum signifikant eingeschränkt wird. Die Kosten der Join-Berechnung sind damit sehr gering und linear zur Relationengröße. Die Bearbeitungsdauer ist im wesentlichen durch das Einlesen der beiden Relationen bestimmt.

Dieses optimale Leistungsverhalten ist jedoch nicht mehr erreichbar, wenn die innere Relation zu groß für eine vollständige Hauptspeicherallokation ist. In diesem Fall ist eine Überlaufbehandlung erforderlich, wofür es zahlreiche Alternativen gibt. Wir beschränken unsere Betrachtungen auf Ansätze, welche auf einer Partitionierung der Relationen basieren. Diese Idee wurde zuerst für die GRACE-Datenbankmaschine realisiert [Ki83]. Beim *GRACE-Hash-Join* wird vor der eigentlichen Join-Bearbeitung eine Partitionierungs-Phase durchgeführt (Abb. 18-4a). In dieser Phase wird zunächst die innere Relation S gelesen und mittels einer auf dem Join-Attribut angewendeten Partitionierungsfunktion g (Hash- oder Bereichspartitionierung) in q Partitionen S_1 bis S_q unterteilt, so daß jede S-Partition im Hauptspeicher gehalten werden kann. Zur Partitionierung werden im Hauptspeicher q Ausgabepuffer von mindestens 1 Seite reserviert, welche auf Externspeicher ausgeschrieben werden, sobald sie keine weiteren S-Tupel mehr aufnehmen können. Das Ausschreiben der erzeugten Partitionen auf temporäre Dateien erfolgt somit weitgehend asynchron. Nach Partitionierung der S-Relation erfolgt eine analoge Zerlegung von R in q Partitionen R_1 bis R_q unter Anwendung derselben Partitionierungsfunktion. Die Partitionierung garantiert, daß die Verbundpartner von Partition S_i nur in der Partition R_i vorliegen können (für alle i, $1 \le i \le q$). Daher genügt es zur eigentlichen Join-Berechnung, wie in Abb. 18-4b,c illustriert, das Hash-Join-Basisverfahren auf jede der q zusammengehörigen S- und R-Partitionen anzuwenden (Einlesen von S_i in eine Hauptspeicher-Hash-Tabelle; Lesen von R_i mit Bestimmung der Verbundpartner).

Abb. 18-4: GRACE-Hash-Join



Eine Verbesserung des GRACE-Joins stellt der sogenannte *Hybrid-Hash-Join* dar [De84]. Er zeichnet sich durch eine bessere Hauptspeichernutzung während der Partionierungs-Phase aus. Dazu wird der von den Pufferbereichen nicht benötigte

Hauptspeicherplatz bereits zur Allokation der Hash-Tabelle für die erste Partition S_1 genutzt. Die Join-Berechnung für die erste Partition kann somit bereits beim ersten Lesen der Relationen S und R erfolgen, so daß für diese Partition keine zusätzlichen E/A-Vorgänge anfallen. Die Partitionierungsfunktion sollte so gewählt werden, daß die erste Partition den für die Hash-Tabelle verfügbaren Platz maximal ausnutzt, um das E/A-Verhalten mit zunehmender Hauptspeichergröße zu verbessern. Der Idealfall, daß die innere Relation komplett im Hauptspeicher Platz findet, ergibt sich dann als Spezialfall.

Der Hybrid-Hash-Join verlangt jedoch i.a. eine ungleichmäßige Partitionierung, um den Hauptspeicher für die erste Partition weitgehend nutzen zu können. Zudem ist die maximale Nutzung des Hauptspeichers im Einbenutzerbetrieb zwar sinnvoll, kann allerdings im Mehrbenutzerbetrieb gleichzeitig laufende Transaktionen höherer Priorität stark benachteiligen. Diese Nachteile können durch einen *adaptiven Hash-Join* umgangen werden, der auf der gleichförmigen Zerlegung in q möglichst gleichgroße Partitionen aufbaut. Ein solcher Ansatz ist der in [PCL93] vorgestellte "Partially Preemptible Hash Join" (PPHJ). Dabei werden für die ersten k der q S-Partitionen die Hash-Tabellen im Hauptspeicher gehalten, für die q-k anderen lediglich ein Ausgabepuffer von einer Seite. Somit läßt sich für die k ersten Partitionen die Join-Berechnung unmittelbar ausführen, und nur für die restlichen Partitionen werden E/A-Vorgänge auf temporäre Dateien nötig. Weiterhin kann die Anzahl der im Hauptspeicher resident gehaltenen S-Partitionen dynamisch variiert werden. So wird deren Anzahl verringert, wenn die Hash-Tabellen wegen ungenauer Schätzungen doch nicht vollständig Platz im Hauptspeicher finden oder wenn aufgrund von Speicheranforderungen durch andere Transaktionen der für die Join-Bearbeitung verfügbare Platz sinkt. Für die betroffenen Partitionen werden dann die Tupel der Hash-Tabelle auf eine temporäre Datrei ausgeschrieben und der Speicherplatz bis auf eine Seite für den Ausgabepuffer freigegeben. Umgekehrt können bei wachsender Hauptspeicherverfügbarkeit während der Partitionierungs-Phase einige bereits auf Platte geschriebene S-Partitionen wieder in den Hauptspeicher gebracht werden, um die weitere Join-Bearbeitung direkt auf ihnen vorzunehmen.

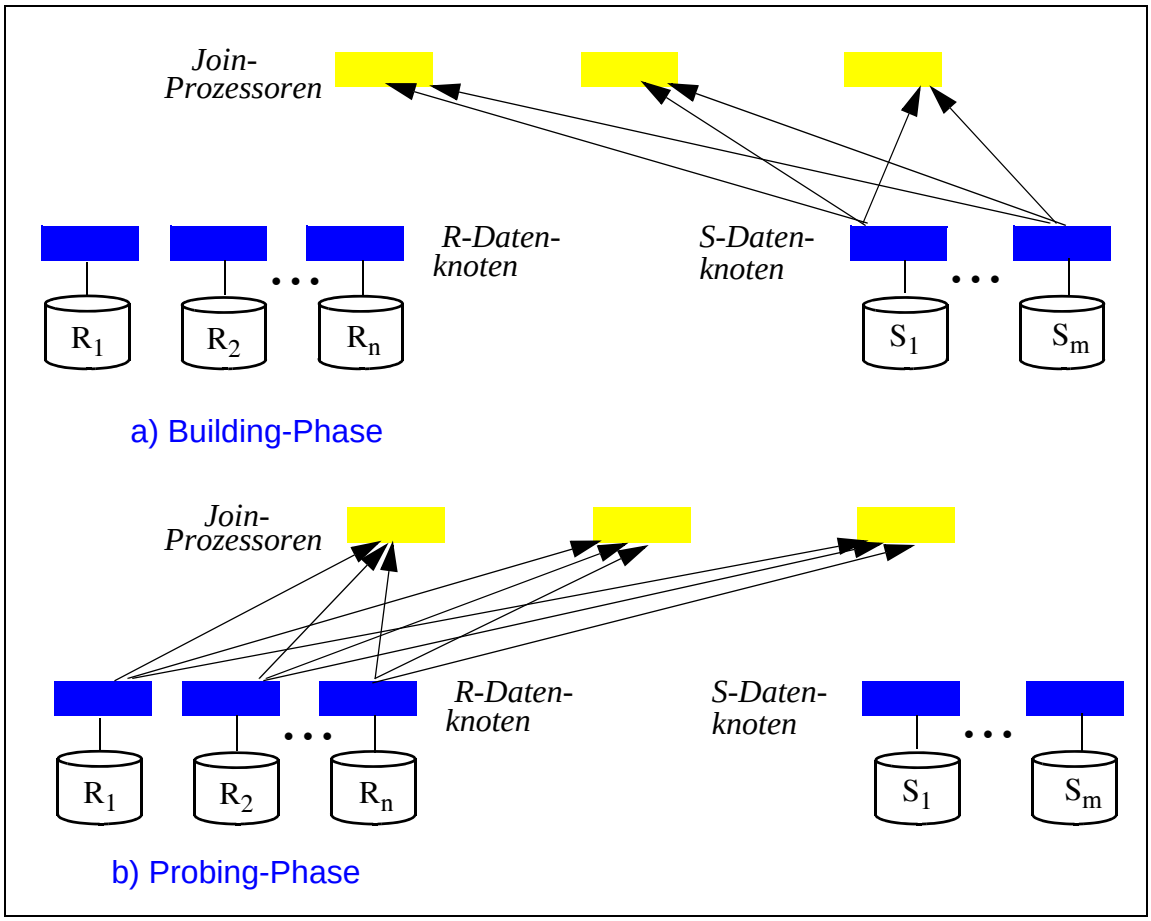
Eine allgemeine Optimierungsmöglichkeit zur Reduzierung von E/A-Vorgängen auf temporären Dateien besteht in der Nutzung von Bitvektoren. Der *Bitvektor* basiert dabei auf einer auf dem Join-Attribut anzuwendenden Hash-Funktion (z.B. die zur Join-Verarbeitung genutzte Funktion h) und enthält für jede Hash-Klasse ein Bit. Während des Einlesens der S-Relation in der Partitionierungs-Phase wird die Hash-Funktion auf jedes S-Tupel angewendet und das zugehörige Bit gesetzt. Dies kann dann zur Reduzierung der größeren Relation R genutzt werden, da nur noch solche R-Tupel in der weiteren Verarbeitung zu berücksichtigen sind, für die das zugehörige Bit gesetzt ist. Durch diese Verkleinerung der R-Partitionen können oft zahlreiche E/A-Vorgänge eingespart und die Join-Bearbeitung

beschleunigt werden. Ein hoher Filtereffekt kann bereits erwartet werden, wenn die Anzahl der Bits doppelt so hoch liegt wie die Tupelanzahl der inneren Relation S [Gra93].

Parallele Realisierungen

Da Hash-Joins zur Berechnung von Equi-Joins eingesetzt werden, eignet sich vor allem eine Parallelisierung mit dynamischer Partitionierung der Eingaberelationen (Kap. 18.3.2). Die unterschiedliche Bearbeitung der inneren und der äußeren Relation im Rahmen der Building- und Probing-Phasen erfordert jedoch, diese beiden Phasen weiterhin nacheinander auszuführen (Abb. 18-5). Es wird also zunächst nur die S-Relation an ihren m Datenknoten parallel eingelesen und unter den p Join-Prozessoren umverteilt. Falls die einzelnen S-Partitionen der Join-Rechner ausreichend klein sind (Basisverfahren), werden sie zudem direkt in Hauptspeicher-Hash-Tabellen gebracht. Danach wird das parallele Lesen und Umverteilen der äußeren Relation gestartet. Im Basisverfahren kann in den Join-Rechnern zudem sofort das Probing vorgenommen werden.

Abb. 18-5: Parallele Hash-Join-Berechnung mit dynamischer Partitionierung der Eingaberelationen



Die Sequentialisierung der beiden Phasen kann als Nachteil gegenüber anderen lokalen Join-Methoden wie Sort-Merge aufgefaßt werden, bei denen beide Relationen parallel verarbeitet werden. Allerdings kann bei Sort-Merge-Joins keine Pipeline-Parallelität zwischen den Daten- und den Join-Knoten genutzt werden. Denn die lokale Join-Bearbeitung ist für Sort-Merge i.a. zu verzögern, bis beide Eingaben vollständig vorliegen, da diese zunächst sortiert werden müssen. Hash-Joins können dagegen Pipeline-Parallelität in beiden Phasen (Building- und Probing-Phasen) einsetzen, was im Rahmen von Mehr-Wege-Joins noch mehr von Vorteil ist (s.u.). Außerdem ist die Sequentialisierung der beiden Phasen Voraussetzung für die Nutzung von Bitvektoren (Hash-Filtern), welche nun auch zu einer signifikanten Reduzierung des Kommunikationsaufwandes zur Datenumverteilung nutzbar sind [SD90]. Hierzu wird an jedem S-Datenknoten vor der Umverteilung ein Bitvektor analog zum zentralen Fall ermittelt, welche dann über eine logische AND-Operation zu einem gemeinsamen Bitvektor für die gesamte Relation verknüpft werden. Dieser Vektor wird dann an die R-Datenknoten übermittelt, womit die Datenumverteilung auf diejenigen R-Tupel beschränkt werden kann, für die das entsprechende Bit gesetzt ist und die daher potentiell am Verbund teilnehmen.

Die dynamische Partitionierung der Eingaberelationen zur Parallelisierung der Join-Bearbeitung ähnelt offenbar der Partitionierung zur Überlaufbehandlung im zentralen Fall. Denn in beiden Fällen wird über eine Hash- oder Bereichspartitionierung eine Zerlegung der beteiligten Relationen in mehrere Partitionen vorgenommen, wobei die Join-Bearbeitung für eine Partition der inneren Relation auf genau eine Partition der äußeren Relation beschränkt wird. Dies gestattet nun eine unmittelbare Parallelisierung. Zudem kann im verteilten Fall aufgrund des (im Mittel um den Faktor p) verringerten Join-Umfangs pro Knoten mit einer höheren Wahrscheinlichkeit das Basisverfahren angewendet werden, bei dem keine Überlaufbehandlung notwendig ist*.

Wenn die S-Partitionen in den Join-Rechnern jedoch nicht vollständig im Hauptspeicher Platz finden, wird eine zusätzliche dynamische Partitionierung zur Überlaufbehandlung erforderlich. Diese kann prinzipiell vor der Umverteilung an den Datenknoten oder nach der Umverteilung an den Join-Rechnern erfolgen. Wird die Überlaufbehandlung an den Join-Rechnern vorgenommen, ergibt sich folgende Vorgehensweise:

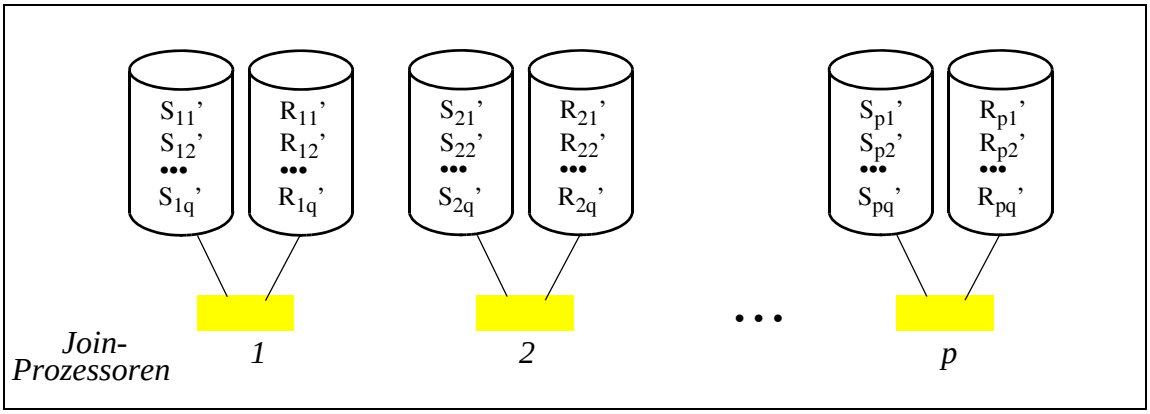
1. *Paralleles Lesen und Umverteilen von S*
   an jedem S-Knoten j (j=1, ..., m) führe parallel durch:
   lies lokale S-Partition S_j und sende jedes S-Tupel an zuständigen Join-Rechner (Anwendung der Verteilungsfunktion f auf dem Join-Attribut)

* Die Speicherung der inneren Relation im Rahmen von Hash-Tabellen entspricht einer weiteren Partitionierung, die den Suchraum zur Bestimmung der Verbundpartner eingrenzt.

2. *Paralleles Partitionieren der S-Partitionen*
   in jedem Join-Knoten k führe parallel durch (k=1...p)
   - $S_k' := \cup S_{jk}$ (j=1..m) (Menge der von Join-Knoten k empfangenen S-Tupel)
   - Partitionierung in q Sub-Partitionen zur Überlaufbehandlung S_{k1}' bis S_{kq}' durch Anwendung einer Partitionierungsfunktion g auf dem Join-Attribut
3. *Paralles Lesen und Umverteilen von R*
   an jedem R-Knoten i (i=1..n) führe parallel durch:
   lies lokale R-Partition R_i und sende jedes R-Tupel an zuständigen Join-Rechner (Verteilungsfunktion f)
4. *Paralleles Partitionieren der R-Partitionen*
   in jedem Join-Knoten k führe parallel durch (k=1...p)
   - $R_k' := \cup R_{ik}$ (i=1..n) (Menge der von Join-Knoten k empfangenen R-Tupel)
   - Partitionierung in q Sub-Partitionen zur Überlaufbehandlung R_{k1}' bis R_{kq}' durch Anwendung einer Partitionierungsfunktion g auf dem Join-Attribut

   Nach diesem Schritt ergibt sich die in Abb. 18-6 gezeigte Situation.
5. *Parallele Join-Berechnung*
   in jedem Join-Knoten k führe parallel durch (k=1...p):
   für *l* = 1, 2 bis q führe nacheinander aus:
   berechne $S_{kl}' \bowtie R_{kl}'$
   schicke Ergebnis an Koordinator

Abb. 18-6: Zweistufige dynamische Partitionierung der Eingaberelationen (Überlaufbehandlung an den Join-Rechnern)



Der gezeigte Algorithmus nutzt Datenparallelität an den Datenknoten zum parallelen Einlesen (aufgrund der statischen Datenpartitionierung) sowie an den Join-Knoten zum parallelen Umverteilen, Partitionieren zur Überlaufbehandlung, sowie zur Join-Berechnung (dynamische Datenpartitionierung). Umverteilung und Überlaufbehandlung der beiden Eingaberelationen sind hier prinzipiell auch parallel möglich (Schritte 1 und 2 parallel zu den Schritten 3 und 4). Pipeline-Parallelität kann zwischen den Daten- und Join-Knoten genutzt werden, wobei auch das Ausschreiben der temporären Partitionen S_{kl}' und R_{kl}' an den Join-Rechnern über Ausgabepuffer weitgehend asynchron realisierbar ist.

Die Überlaufbehandlung an den Join-Rechnern bringt den Vorteil mit sich, daß dort praktisch jeder lokale Ansatz gewählt werden kann. So entspricht der ge-

zeigte Algorithmus der parallelen Version eines GRACE-Hash-Joins. Die Realisierung eines parallelen Hybrid-Hash-Joins ist analog möglich. Hierzu wird in jedem Join-Rechner k bei der Überlaufbehandlung die jeweils erste S-Partition S_{k1}' im Hauptspeicher gehalten, um E/A-Vorgänge einzusparen und mit den zugehörigen R-Tupeln die Join-Berechnung sofort vornehmen zu können (Voraussetzung hierfür ist jedoch wieder die Sequentialisierung von S- und R-Bearbeitung). In ähnlicher Weise können auch adaptive Hash-Verfahren in den Join-Rechnern angewendet werden.

Alternativ zu der beschriebenen Vorgehensweise kann eine Überlaufbehandlung bereits an den Datenknoten erfolgen (s. Übungsaufgaben). Ein Vorteil dabei ist die Vermeidung von E/A-Vorgängen an den Join-Rechnern, so daß selbst Rechner ohne Platten (diskless nodes) zur Join-Berechnung nutzbar werden.

Auch für die parallele Hash-Join-Berechnung kann die Anzahl der Join-Prozessoren sowie ihre Auswahl wieder dynamisch - in Abhängigkeit der aktuellen Systemauslastung - festgelegt werden. Die Untersuchung [RM95] zeigte, daß hierfür vor allem die aktuelle Hauptspeicherverfügbarkeit der einzelnen Rechner berücksichtigt werden sollte, um eine Zuordnung zu finden, die mit möglichst wenigen E/A-Vorgängen zur Überlaufbehandlung auskommt. Dazu ist es bei hoher Hauptspeicherauslastung empfehlenswert, eine höhere Anzahl von Join-Prozessoren zu wählen, um den Speicherbedarf pro Knoten zu reduzieren. Dies ist die entgegengesetzte Vorgehensweise wie zur Behandlung von CPU-Engpässen, bei denen sich zur Reduzierung des Kommunikations-Overheads eine Reduzierung des Parallelitätsgrades empfiehlt.

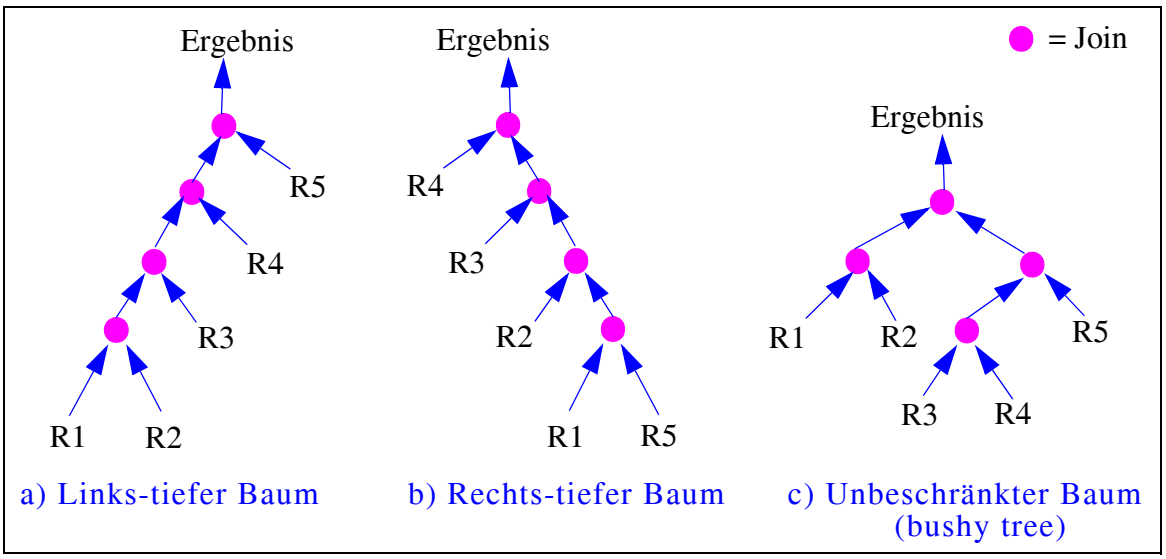
18.3.4 Mehr-Wege-Joins

Zur Parallelisierung komplexer Anfragen, deren Operatorbäume zahlreiche Knoten umfassen, fand vor allem die Optimierung von Mehr- bzw. N-Wege-Joins große Beachtung in der Literatur [Ze89, SD90, LST91,CYW92, CLYY92, ZZB93]. Auf die Optimierung solcher Anfragen in Verteilten DBS wurde bereits in Kap. 6.5.4 eingegangen. Ein besonderes Optimierungsproblem besteht darin, eine äquivalente und möglichst günstige Reihenfolge von Zwei-Wege-Joins zu finden. Das Optimierungsproblem verschärft sich bei paralleler Join-Berechnung noch erheblich. Denn hierzu sind Festlegungen sowohl für Intra- als auch Inter-Operatorparallelität zu treffen, insbesondere auf welchen Prozessoren die Join-Bearbeitung zu welchen Zeitpunkten erfolgen soll. Weiterhin soll sowohl Daten- als auch Pipeline-Parallelität genutzt werden.

Zur Reduzierung des Optimierungsproblems betrachten die meisten Implementierungen und Verfahrensvorschläge lediglich lineare Operatorbäume, bei denen ein Mehr-Wege-Join durch eine lineare Folge von Zwei-Wege-Joins realisiert wird. Hierbei kann unterschieden werden zwischen den in Abb. 18-7 gezeigten links-tie-

fen (left deep) und rechts-tiefen (right deep) Operatorbäumen [Ze89, SD90]*. *Unbeschränkte Operatorbäume* (bushy trees, Abb. 18-7c) bieten dagegen offensichtlich ein größeres Optimierungspotential als lineare Bäume. Insbesondere kann mit ihnen Datenparallelität weitgehender genutzt werden, da auf unterschiedlichen Relationen arbeitende Teilbäume vollständig parallel ausführbar sind (z.B. R1 ⋈ R2 und R3 ⋈ R4 in Abb. 18-7c). Allerdings ist der Optimierungsaufwand für unbeschränkte Operatorbäume komplexer Anfragen vielfach prohibitiv.

Abb. 18-7: Berechnungsfolgen für N-Wege-Joins (N=5)



Die Unterschiede zwischen links- und rechts-tiefen Bäumen treten vor allem bei der Verwendung von Hash-Joins zu Tage. Beim *links-tiefen Baum* (Abb. 18-7a) ist die zum Probing benutzte, rechte Eingabe jeder Join-Operation eine Basisrelation. Die Join-Berechnung erfolgt in N Schritten, wobei im ersten Schritt die Hash-Tabelle für die erste Relation ("linker" Operand) aufgebaut wird und in den N-1 folgenden Schritten das Probing mit den restlichen Basisrelationen erfolgt. Die Ergebnistupel einer Join-Berechnung werden schubweise an den jeweils nächsten Join-Knoten gesendet, der damit die Hash-Tabelle im Hauptspeicher aufbaut. Dort kann die eigentliche Join-Berechnung (Probing-Phase) erst gestartet werden, wenn das komplette Ergebnis des vorausgehenden Joins vorliegt. Der Einsatz von Pipeline-Parallelität zwischen Join-Operatoren ist somit auf jeweils zwei Knoten beschränkt. Von Nachteil ist ferner, daß die Größe der Hash-Tabellen von der Selektivität der einzelnen Join-Operationen abhängt, so daß der Hauptspeicherbedarf oft nur ungenau bekannt ist.

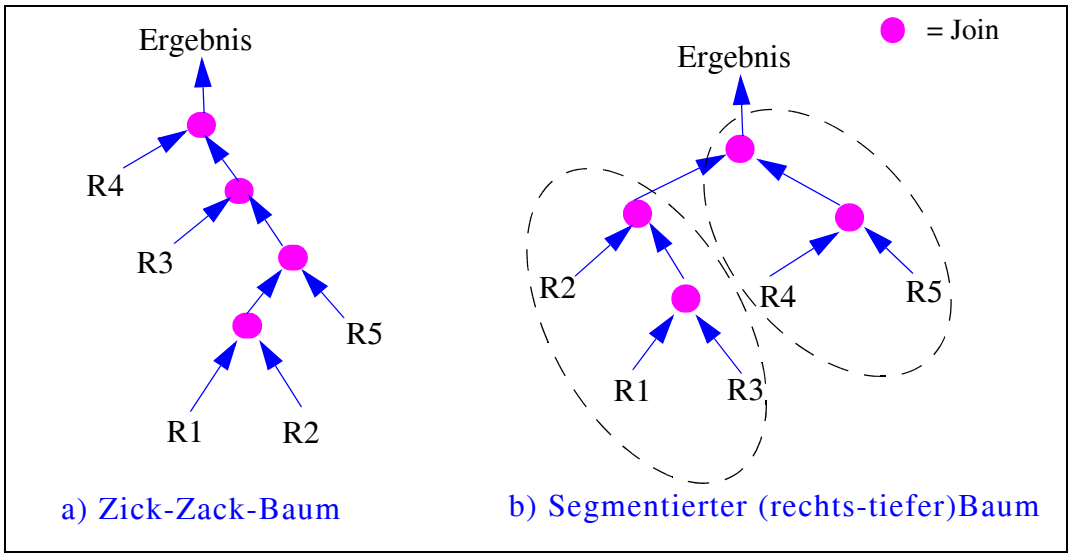
Beim *rechts-tiefen Baum* (Abb. 18-7b) wird durch verstärkte Nutzung von Parallelität eine Join-Bearbeitung in zwei Schritten erreicht. Dabei werden im ersten

* Von "tiefen" Operatorbäumen spricht man generell, wenn alle inneren Knoten des Baumes wenigstens einen Blattknoten (Basisrelation) als Eingabe besitzen [IK91].

Schritt N-1 der N Basisrelationen ("linke" Operanden) parallel in die Hauptspeicher der Join-Rechner geladen. Im zweiten Schritt wird dann sukzessive die Join-Berechnung mit der N-ten Relation unter Nutzung von Pipeline-Parallelität durchgeführt. Dieser Ansatz hat jedoch den Nachteil eines sehr hohen Speicherbedarfs. Gestattet jedoch die Hauptspeicherkapazität das vollständige Laden der N-1 Relationen ergeben sich deutliche Vorteile gegenüber links-tiefen Bäumen [SD90]. Der geringere Speicherbedarf links-tiefer Bäume ist dagegen vor allem bei hoher Join-Selektivität sehr ausgeprägt.

Links- und rechts-tiefe Bäume stellen Extremansätze hinsichtlich des Ressourcenbedarfs sowie der Anzahl benötigter Verarbeitungsschritte dar. Als Alternative dazu wurden in [ZZB93] sogenannte *Zick-Zack-Bäume* vorgeschlagen, welche lineare (tiefe) Operatorbäume darstellen, bei denen jedoch für Teilbäume zwischen links-tiefer und rechts-tiefer Vorgehensweise gewechselt werden kann. So kommt im Beispiel von Abb. 18-8a zunächst ein links-tiefer Teilbaum zur Auswertung, um den Join zwischen R1 und R2 zu berechnen. Das Ergebnis dieser Join-Operation bildet die "linke" Eingabe zu einem rechts-tiefen Teilbaum, ebenso wie die Relationen R3 und R4, die parallel in die Hauptspeicher der betreffenden Rechner gebracht werden. Die abschließende Probing-Phase mit R5 kann erst beginnen, nachdem das Join-Ergebnis von R1 ⋈ R2 vollständig vorliegt. Eine solche Kompromißlösung ermöglicht einen reduzierten Ressourcenbedarf gegenüber rechts-tiefen sowie einen höheren Parallelitätsgrad gegenüber links-tiefen Bäumen. Der Suchraum zur Optimierung erhöht sich dafür, da rechts- bzw. links-tiefe Bäume als Spezialfälle weiterhin möglich sind.

Abb. 18-8: Alternative Baumstrukturen für N-Wege-Joins (N=5)



Eine ähnliche Zielstellung wie mit Zick-Zack-Bäumen wird mit *segmentierten rechts-tiefen Bäumen* [CLYY92, STY93] verfolgt. Diese teilen den Operatorbaum in eine Menge rechts-tiefer Teilbäume (Segmente) auf, für welche die "linken" Einga-

berelationen jeweils hauptspeicherresident gehalten und die daher parallel eingelesen werden können. Es handelt sich dabei im Prinzip um eine Verallgemeinerung von Zick-Zack-Bäumen, wobei nicht nur "tiefe", sondern beliebige Baumstrukturen (bushy trees) zugelassen werden, wie das Beispiel in Abb. 18-8b auch zeigt. Zur Vereinfachung von Optimierung und Scheduling-Entscheidungen wird jedoch zu einem Zeitpunkt jeweils nur ein Segment bearbeitet, also die Parallelität gegenüber unbeschränkten Operatorbäumen reduziert. Im Beispiel von Abb. 18-8b wird die Berechnung von R4 ⋈ R5 somit erst begonnen, wenn die Join-Berechnung des linken Segmentes beendet ist.

Die Eingrenzung des Suchraumes durch strukturelle Beschränkungen der Operatorbäume stellt nur einen Ansatz zur Reduzierung des Optimierungsaufwandes für komplexe Anfragen (z.B. Mehr-Wege-Joins) dar. Eine Alternative besteht in der Wahl einer effizienten Suchstrategie, die den hohen Aufwand für eine enumerative Bewertung nahezu aller Pläne umgeht, wie er mit dem klassischen Ansatz der dynamischen Programmierung [Se79] entsteht. Dies kann u.a. mit zufallsgesteuerten Suchstrategien wie Simulated Annealing oder iterativer Verbesserung erreicht werden, die eine bestimmte Ausgangslösung bis zum Erreichen eines lokalen Optimums verbessern. Auch wenn damit nicht immer das globale Optimum erreicht wird, findet sich mit geringem Aufwand meist eine gute Lösung. In [LVZ93] wurde gezeigt, daß die Anwendung solcher Suchstrategien auf einem unbeschränkten Suchraum (bushy trees) vielfach wirkungsvoller ist als der Einsatz enumerativer Suchstrategien auf einem beschränkten Suchraum.

18.4 Probleme der parallelen DB-Verarbeitung

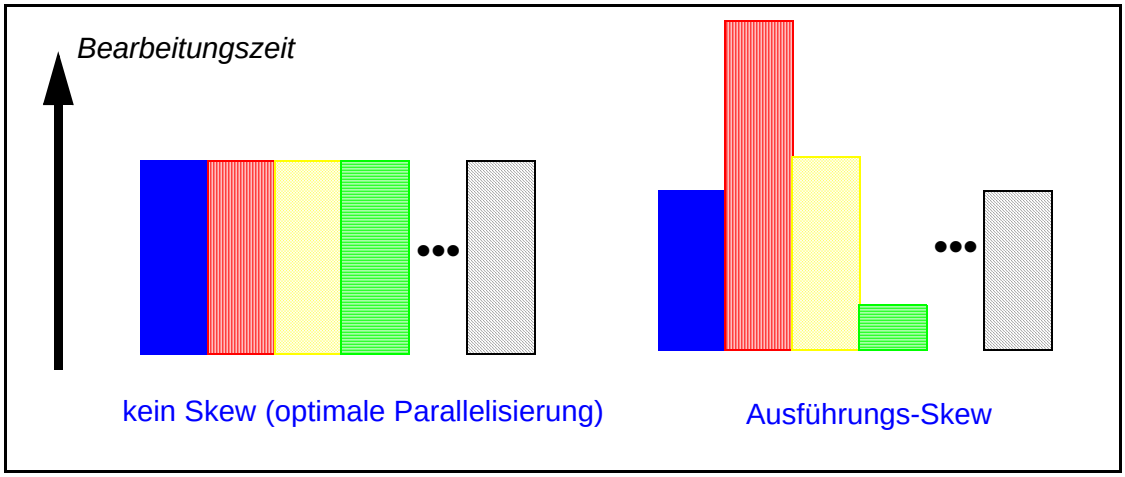
Abschließend gehen wir noch etwas genauer auf zwei Problembereiche bei der parallelen Datenbankverarbeitung ein, die zum Teil bereits angesprochen wurden. Zunächst diskutieren wir die Beeinträchtigung des Leistungsverhaltens durch sogenannte Skew-Effekte, die zu starken Schwankungen in den Bearbeitungszeiten einzelner Teilanfragen führen können. Danach wird untersucht, welche Probleme beim gemeinsamen Einsatz von Inter- und Intra-Transaktionsparallelität auftreten.

18.4.1 Skew-Behandlung

Die effektive Parallelisierung einer Operation in mehrere Teilanfragen verlangt, daß jede Teilanfrage möglichst gleich schnell bearbeitet wird, da die gesamte Bearbeitungszeit durch die langsamste Teiloperation bestimmt ist. Die Varianz in den Bearbeitungszeiten, die wir als *Ausführungs-Skew* (execution skew) bezeichnen, ist jedoch oft nur schwer zu begrenzen und beeinträchtigt somit Speedup und Skalierbarkeit (Abb. 18-9). Ausführungs-Skew geht vielfach auf *Daten-Skew* zurück, der vorliegt, wenn einzelne Teiloperationen unterschiedlich große Daten-

mengen zu verarbeiten haben. Daten-Skew wiederum ist oft eine Folge ungleicher Werteverteilungen in der Datenbank (attribute value skew [WDJ91]). In der folgenden Diskussion unterstellen wir Einbenutzerbetrieb, wo die Umgehung von Ausführungs-Skew bereits ein großes Problem darstellt. Im Mehrbenutzerbetrieb verschärfen sich die Skew-Probleme, da das damit einhergehende Ausmaß an Behinderungen zwischen Transaktionen an verschiedenen Rechnern i.a. differiert.

Abb. 18-9: Einfluß von Ausführungs-Skew



Der Einsatz von Pipeline-Parallelität (Inter-Operatorparallelität) ist besonders anfällig für Ausführungs-Skew, da die von einzelnen Operatoren zu verarbeitenden Datenmengen stark differieren können. Die Datenmengen sind zudem durch die konkreten Datenbankinhalte und Anfragecharakteristika bestimmt, welche von der Anfrageoptimierung jedoch nicht beeinflußbar und deren Auswirkungen auf die Größe von Zwischenergebnisse oft nur ungenau abschätzbar sind.

Im Falle von Datenparallelität (Intra-Operatorparallelität) äußert sich Daten-Skew meist in der Form von *Partitions-Skew,* der vorliegt, wenn Teiloperationen unterschiedlich große Datenpartitionen bearbeiten, die statisch oder dynamisch definiert sein können. Für parallele Join-Verfahren (Kap. 18.3) kann sich Partitions-Skew in der Scan-Phase, während der dynamischen Umverteilung sowie in der Join-Phase negativ bemerkbar machen. Dabei können nach [WDJ91] insgesamt vier Arten von Partitions-Skew unterschieden werden, die auch kombiniert auftreten können:

- *Datenverteilungs-Skew (tuple placement skew)*
  Die statische Datenverteilung unter mehreren Rechnern (bzw. Platten) führte zu unterschiedlich großen Partitionen, so daß Leseoperationen (Scans) darauf unterschiedlich lange andauern. Bei Definition der Datenverteilung auf einem Verteilungsattribut (Hash- bzw. Bereichsfragmentierung, Kap. 17.2) kann dies durch eine ungleichmäßige Werteverteilung für dieses Attribut und/oder eine ungeeignete Verteilungsfunktion hervorgerufen werden.

- *Selektivitäts-Skew*
  Die Scan-Operationen auf den einzelnen Partitionen weisen unterschiedliche Selektivitätsfaktoren auf, so daß sich unterschiedlich viele Tupel qualifizieren. Ein Beispiel hierfür ist etwa eine Bereichsanfrage auf dem Verteilungsattribut.
- *Umverteilungs-Skew (redistribution skew)*
  Die dynamische Umverteilung der Ergebnisse der Scan-Phase führt zu unterschiedlich großen Join-Operanden für verschiedene Join-Prozesse. Dies ist möglich aufgrund einer schiefen Werteverteilung für das Join-Attribut und/oder einer ungeeigneten Verteilungsfunktion zur dynamischen Partitionierung der Daten.
- *Join-Produkt-Skew*
  Die Join-Selektivität differiert zwischen den Join-Knoten, so daß sich unterschiedlich viele Sätze für das Join-Ergebnis qualifizieren.

Selektivitäts-Skew kann dynamisch kaum beeinflußt werden, da er durch die jeweilige Anfrage sowie die gewählte Datenverteilung bestimmt ist. Voraussetzung zur Vermeidung von *Datenverteilungs- und Umverteilungs-Skew* ist eine möglichst genaue Kenntnis der Werteverteilung für das Attribut, auf dem die Verteilungsfunktion anzuwenden ist. Die Häufigkeit bestimmter Attributwerte kann entweder vollständig in Form von Histogrammen geführt oder stichprobenartig über Sampling-Verfahren ermittelt werden. Zur Definition der statischen Datenverteilung empfiehlt sich die Nutzung von Histogrammen, welche durch Lesen aller Sätze ermittelt und im Katalog gespeichert werden können. Durch Definition einer auf die Werteverteilung abgestimmten Bereichspartitionierung kann dann Datenverteilungs-Skew umgangen werden, insbesondere wenn der Primärschlüssel als Verteilattribut gewählt wird (keine replizierten Attributwerte).

Für die dynamische Umverteilung kann die Werteverteilung i.a. vorab nicht abgeschätzt werden, so daß sie dynamisch zu ermitteln ist. Der Aufwand hierfür ist relativ gering für Sort-Merge-Joins, wenn bereits vor der Umverteilung eine Sortierung an den Datenknoten erfolgt. Die Werteverteilung kann dann nämlich während des Sortierens bestimmt und an einem ausgezeichneten Knoten für alle Datenknoten kombiniert werden (ähnlich wie für parallele Sortierverfahren, Kap. 18.2.4). Anderenfalls muß über ein Sampling-Ansatz eine Approximation der Werteverteilung bestimmt werden. Der Aufwand hierfür kann nach [DNSS92] relativ gering gehalten werden, jedoch ist eine effektive Stichprobenerhebung erst möglich, wenn die umzuverteilenden Datenmengen vollständig vorliegen [Gra93]. Dies impliziert, daß Pipeline-Parallelität zwischen Scan- und Join-Phase nicht mehr nutzbar ist.

Umverteilungs-Skew läßt sich bei (in Annäherung) bekannter Werteverteilung für das Join-Attribut auch wieder am besten durch eine Verteilung über eine Bereichspartitionierung erreichen [DNSS92], da hiermit in etwa gleich große Partitionsgrößen gebildet werden können. Weiterhin läßt sich auch der Extremfall, daß das gehäufte Auftretens eines einzigen Wertes zu Skew führt, behandeln, während

eine Hash-Funktion alle Sätze mit übereinstimmenden Join-Attributwert dem gleichen Join-Rechner zuordnet. Die Lösung besteht darin, Tupel mit dem betreffenden Join-Attributwert w mehreren Join-Knoten zuzuordnen (überlappende Bereiche), so daß die einzelnen Partitionen in etwa die gleiche Größe annehmen. In diesem Fall muß jedoch darauf geachtet werden, das vollständige Join-Ergebnis zu erhalten. Eine Möglichkeit besteht darin, die Sätze der ersten Relation mit Join-Attributwert w unter mehreren Join-Rechnern zu partitionieren und die zugehörigen Sätze der zweiten Relation an den betreffenden Join-Rechnern zu replizieren (d.h. an alle Join-Rechner mit w-Sätzen der ersten Relation zu schicken). Der umgekehrte Ansatz ist, die w-Sätze der ersten Relation zu replizieren und die der zweiten Relation zu partitionieren.

Beispiel 18-2

Für die folgenden Relationen R und S soll der Join über Attribut B an zwei Join-Prozessoren berechnet werden.

R (A, B)	S (B, C)
(1, 3)	(1, 1)
(2, 3)	(2, 2)
(3, 3)	(3, 3)
(4, 4)	(4, 4)
	(5, 4)

Es wird entschieden, daß die Tupel im Wertebereich 1-3 für B dem ersten, und im Bereich 3-5 dem zweiten Join-Prozessor zugeordnet werden, so daß der Join-Attributwert 3 zwei Prozessoren zugeordnet wird. Eine Partitionierungsmöglichkeit von R hierzu ist, die R-Tupel (1, 3) und (2, 3) dem ersten und die Tupel (3, 3) und (4, 4) dem zweiten Join-Prozessor zuzuordnen. In diesem Fall ist das S-Tupel (3, 3) zu replizieren, also beiden Join-Prozessoren zuzuweisen, um das vollständige Join-Resultat zu erhalten.

Das Ausmaß von *Join-Produkt-Skew* ist durch die Werteverteilung des Join-Attributs beider Relationen bestimmt, wie sie sich im Join-Ergebnis niederschlägt. Eine Bereichspartitionierung der beiden Eingaberelationen in je p etwa gleich große Partitionen (p = Anzahl der Join-Prozessoren) reicht daher zur Vermeidung dieses Skew-Typs nicht aus, da der Umfang der Join-Ergebnisse für den einzelnen Partitionen stark schwanken kann. Die Lösung erfordert vielmehr eine Abschätzung der Gesamtgröße des Join-Ergebnisses sowie der dabei entstehenden Werteverteilung für das Join-Attribut, welche aus der Werteverteilung für die Eingaberelationen abgeleitet werden kann. Damit läßt sich dann eine Bereichspartitionierung festlegen, welche für jeden der p Join-Prozessoren ein etwa gleich großes Teilergebnis ergibt. Problematisch ist dabei vor allem wieder der Fall, wenn aufgrund stark ungleicher Werteverteilung ein einzelner Attributwert zur Überlastung eines Join-Prozessors führt. Solche Werte sind dann von mehreren Join-Prozessoren zu bearbeiten, wobei wieder für eine der Relationen eine Partitionierung, für die andere eine Replizierung der entsprechenden Sätze an die zuständigen

Join-Rechner notwendig wird. Zur Balancierung der Last ist es hierzu auch oft erforderlich, mehr Partitionen (Bereiche) als Join-Prozessoren zu bilden und jedem Prozessor mehrere Partitionen zuzuweisen [DNSS92].

Beispiel 18-3

Es soll ein Join zwischen zwei Relationen mit je 10.000 Sätzen bestimmt werden, wobei Join-Attributwert w in beiden Relationen 1000-mal vorkommen soll, alle übrigen Werte dagegen nur einmal. Für den Wert w ergeben sich somit 1 Million Resultats-Tupel, gegenüber höchstens 9000 Ergebnistupel für die anderen Werte. Bei 10 Join-Prozessoren ist einer somit hoffnungslos überlastet, wenn jeder Join-Attributwert nur einem Rechner zugeordnet wird (wie bei einer Hash-Funktion der Fall). Auch eine Bereichspartitionierung mit einer Aufteilung der Relationen in 10 Bereiche ergibt den gleichen Effekt, da die w-Tupel nur 1/10 der Sätze ausmachen und somit einer Partition (einem Join-Rechner) zugewiesen werden. Zur Lösung des Problems müssen die w-Tupel offenbar von allen 10 Join-Prozessoren bearbeitet werden, so daß für den Attributwert w bereits wenigstens 10 Partitionen vorzusehen sind. Für die Zuordnung der anderen Werte sind nochmals mindestens 10 Wertebereiche erforderlich, so daß also jedem Join-Prozessor wenigstens 2 Partitionen zugewiesen werden. Die 1000 w-Sätze der ersten Relation können somit unter den 10 Join-Prozessoren partitioniert werden, während die 1000 w-Sätze der zweiten Relation an jeden Join-Prozessor gehen. Damit umfaßt das Teilergebnis jedes Join-Prozessors 100.000 w-Tupel, wie zur Lastbalancierung bzw. Vermeidung von Join-Produkt-Skew erforderlich.

18.4.2 Unterstützung gemischter Arbeitslasten

Ein wesentliches Ziel paralleler DBS ist die effektive Unterstützung von Intra- sowie von Inter-Transaktionsparallelität. Insbesondere für Shared-Nothing-Systeme konnte anhand von Messungen [De90, EGKS90] bzw. Simulationen [MR92] nachgewiesen werden, daß die Leistungsziele "linearer Antwortzeit-Speedup" bzw. "linearer Durchsatz-Scaleup" unter günstigen Bedingungen erreicht werden können. Dies ist vor allem für homogene Anwendungslasten (1 Transaktionstyp) möglich, für die eine gleichmäßige Daten- und Lastverteilung über alle Rechner einstellbar ist. So kann für die in den DB-Benchmarks TPC-A und TPC-B [Gr93] zugrundeliegende Bankanwendung eine optimale Datenpartitionierung zur Unterstützung hoher Transaktionsraten (geringer Kommunikationsaufwand) und sehr guter Skalierbarkeit leicht bestimmt werden. Ein linearer Antwortzeit-Speedup läßt sich daneben für datenintensive (jedoch einfache) relationale DB-Operationen durch Einsatz von Intra-Operatorparallelität erzielen, vor allem im Einbenutzerbetrieb [MR92].

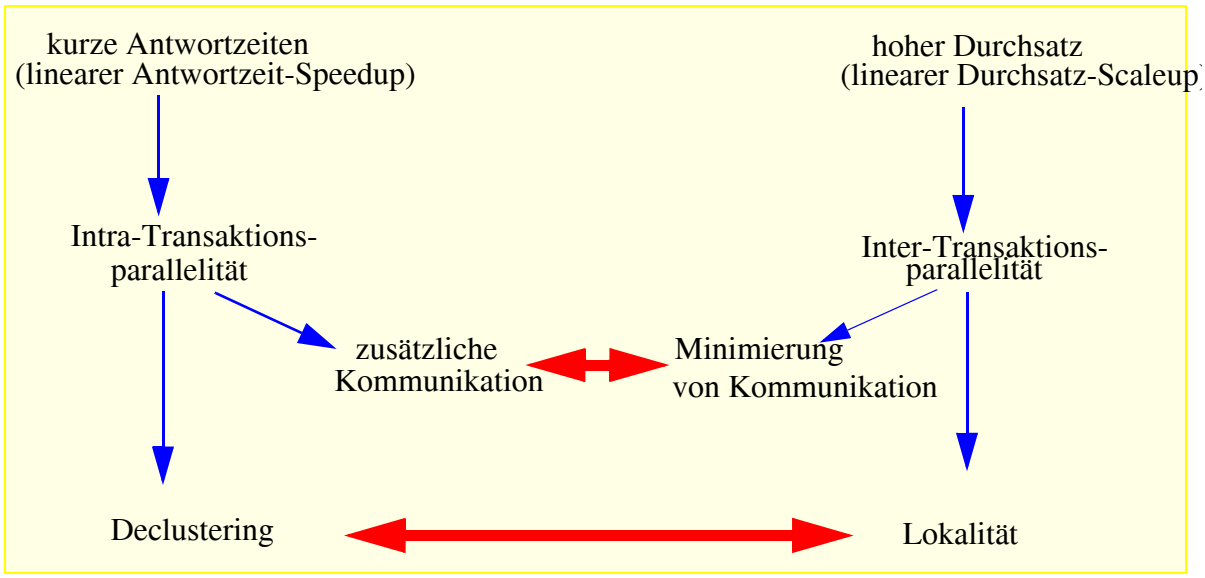
Ein noch weitgehend ungelöstes Problem ist jedoch die effektive Parallelisierung komplexer DB-Anfragen für heterogene bzw. gemischte Lasten, wenn parallel zu einer Anfrage andere Anfragen bzw. OLTP-Transaktionen im System auszuführen sind. Die Unterstützung solcher Arbeitslasten ist bereits für zentralisierte DBS ein großes Problem, da es zwischen den Lasttypen zu Interferenzen kommt, welche das Leistungsverhalten empfindlich beeinträchtigen können. Behinderungen zeigen sich sowohl für Datenobjekte im Rahmen der Synchronisation (*data contention*) als auch für allgemeine Betriebsmittel wie CPU, Platten, Hauptspeicher, etc.,

um die konkurriert wird (*resource contention*). So sind etwa zahlreiche Sperrkonflikte vorprogrammiert, wenn komplexe Anfragen lange Lesesperren halten, die parallel vorzunehmende Änderungen blockieren und somit den OLTP-Durchsatz reduzieren. Eine Abhilfemöglichkeit hierfür besteht in der Nutzung eines Mehrversionen-Sperrverfahrens, womit für lesende Transaktionen bzw. Anfragen Synchronisationskonflikte vermieden werden (Kap. 8.4).

Problematischer ist die Behandlung von Resource-Contention aufgrund des hohen Ressourcenbedarfs komplexer Anfragen, der zu starken Behinderungen gleichzeitig aktiver Transaktionen führen kann. Hier besteht im wesentlichen nur die Möglichkeit, durch geeignete Scheduling-Verfahren die Behinderungen zu kontrollieren. So kann OLTP-Transaktionen, deren zügige Bearbeitung in vielen Anwendungen oberstes Ziel sein muß, eine höhere Priorität als komplexen Anfragen eingeräumt werden, um die Behinderungen für sie einzuschränken. Allerdings werden solche Prioritäten in derzeitigen DBS noch kaum unterstützt. Ferner ist für die Wirksamkeit der Priorisierung wesentlich, daß Scheduling-Komponenten im DBS, Betriebssystem oder dem TP-Monitor die gleichen Prioritäten verwenden, was eine Kooperation zwischen diesen Subsystemen erfordert, die in derzeitigen Systemen nicht vorliegt [Ra93a].

Die Probleme mit gemischten Lasten verschärfen sich bei paralleler DB-Verarbeitung auf einem Mehrrechner-Datenbanksystem. Dies liegt daran, daß die gleichzeitige Unterstützung hoher Transaktionsraten für OLTP und kurzen Antwortzeiten für komplexe Anfragen hier weitere Zielkonflikte (Tradeoffs) verursacht, die zum Teil in Abb. 18-10 illustriert sind. So ist zur Erlangung kurzer Antwortzeiten für komplexe Anfragen Intra-Transaktionsparallelität erforderlich, die jedoch zwangsweise einen höheren Kommunikationsaufwand als eine sequentielle Bearbeitung hervorruft. Dies führt zu einer Verschärfung der Interferenzen mit gleichzeitig laufenden Transaktionen. Die Notwendigkeit dieser gleichzeitig aktiven Transaktionen ergibt sich aus der Forderung nach hohem Durchsatz, insbesondere für OLTP-Transaktionen. Denn dieser kann nur mit Inter-Transaktionsparallelität (Mehrbenutzerbetrieb) erzielt werden, da bei serieller Transaktionsausführung aufgrund von E/A- oder Kommunikationsunterbrechungen eine inakzeptable CPU-Auslastung entstünde. Die Erlangung hoher Transaktionsraten verlangt zudem die Minimierung des Kommunikations-Overheads, um die effektive CPU-Nutzung zu optimieren, was jedoch im Widerspruch zu der durch Intra-Transaktionsparallelität verursachten Zunahme des Kommunikations-Overhead steht.

Abb. 18-10: Performance-Tradeoffs zwischen Durchsatz- und Antwortzeitoptimierung



Schließlich ist es - vor allem bei Shared-Nothing - vielfach nicht möglich, eine Datenverteilung zu finden, die sowohl Intra-Transaktionsparallelität als auch Inter-Transaktionsparallelität (Minimierung von Kommunikationsvorgängen) gut unterstützt. Denn erstere Anforderung führt zu einem Declustering der Daten (Relationen), während letzteres Teilziel oft eine hohe Lokalität des DB-Zugriffs voraussetzt. Dieser Tradeoff wurde bereits in Kap. 17.1 diskutiert, wo festgestellt wurde, daß unterschiedliche Anfragetypen meist unterschiedliche Verteilgrade erfordern. Die Wahl einer Kompromißlösung - wie für Shared-Nothing erforderlich - ist relativ unbefriedigend, da damit ein suboptimales Leistungsverhalten einhergeht, das auch in Kauf zu nehmen ist, wenn zu einem Zeitpunkt jeweils nur ein bestimmter Anfragetyp aktiv ist. Shared-Disk und Shared-Everything bieten in dieser Hinsicht größere Freiheitsgrade, da der Verteilungsgrad der Daten nicht notwendigerweise den Parallelitätsgrad bestimmt. Vielmehr kann dieser je nach Anfragetyp gewählt werden. So können OLTP-Transaktionen stets sequentiell bearbeitet werden, um den Kommunikationsbedarf für sie minimal zu halten, während Intra-Transaktionsparallelität nur bei komplexen Anfragen eingesetzt wird.

Im verteilten Fall ist es zur Kontrolle von Resource-Contention im Mehrbenutzerbetrieb auch wesentlich, den Parallelitätsgrad und damit das Ausmaß an Behinderungen zwischen Transaktionen dynamisch an die aktuelle Lastsituation anzupassen. So kann für komplexe Anfragen bei geringer Systemauslastung ein hoher Parallelitätsgrad toleriert werden, während im Hochlastfall oft ein geringerer Grad an Intra-Transaktionsparallelität sinnvoll ist. Eine weitere Forderung dabei ist die Unterstützung einer dynamischen Lastbalancierung, um eine möglichst gleichmäßige Rechnerauslatung zu erreichen. Dazu sollten Transaktionen bzw.

einzelne Teilanfragen möglichst Rechnern mit geringer Auslastung zugewiesen werden.

Für beide Kontrollentscheidungen bieten Shared-Disk und Shared-Everything wiederum die höchsten Freiheitsgrade, da Parallelitätsgrad sowie ausführende Prozessoren durch die Datenverteilung nicht vorgegeben sind. Dies ist jedoch bei Shared-Nothing der Fall für Zugriffe auf Basisrelationen, so daß hierfür keine dynamischen Steuerungsmöglichkeiten bestehen. Shared-Nothing bietet diese lediglich für Operationen (z.B. Joins) auf abgeleiteten Daten, die dynamisch umverteilt werden, so daß die Anzahl der Zielrechner sowie deren Auswahl zur Laufzeit festgelegt werden können (Kap. 18.3.2).
Eine Leistungsbewertung verschiedener Verfahren zur dynamischen Parallelisierung und Lastbalancierung für Shared-Nothing findet sich in [RM93, RM95]. Die Untersuchungen konzentrieren sich auf die parallele Join-Verarbeitung und berücksichtigen die aktuelle CPU- und Hauptspeicherauslastung für dynamische Kontrollentscheidungen. In [RS95] wird die parallele Bearbeitung von Scan-Anfragen in Shared-Disk-Systemen analysiert. Es stellt sich u.a. heraus, daß durch eine dynamische Festlegung des Parallelitätsgrades Plattenengpässe reduziert werden können, was bei Shared-Nothing für Scan-Anfragen nicht möglich ist.

Übungsaufgaben

Aufgabe 18-1: Parallele Join-Berechnung

Relation R (1 Million Tupel) sei an 4 Rechnern (n=4), Relation S (100.000 Tupel) an 2 Rechnern (m=2) gespeichert. Wie hoch ist der Kommunikationsumfang (in MB) für die Datenumverteilung zur parallelen Join-Berechnung

- für dynamische Replikation
- für dynamische Partitionierung und 5 Join-Rechner (p=5)
- für dynamische Partitionierung, wenn für S das Verteil- mit dem Join-Attribut übereinstimmt, nicht jedoch für R ?

Die Tupelgröße betrage 100 B.

Aufgabe 18-2: Mindest-Hauptspeichergröße für partitionierten Hash-Join

Der GRACE-Hash-Join partitioniert S und R in q Partitionen, falls die kleinere Relation S nicht in den Hauptspeicher paßt. Zeigen Sie, daß bei einer Größe der S-Relation von b Seiten, eine Hauptspeichergröße von wenigstens $\sqrt{b} + 1$ Seiten erforderlich ist.

Aufgabe 18-3: TID-Hash-Join

Eine Reduzierung des Speicherbedarfs von Hash-Joins ergibt sich, wenn in der Hash-Tabelle anstelle der vollständigen Sätze nur die Schlüsselwerte für das Join-Attribut sowie die Verweise (tuple identifiers, TID) auf die Sätze gespeichert werden. Diskutieren Sie

Vor- und Nachteile eines solchen Ansatzes. Welche Auswirkungen ergeben sich für die parallele Join-Bearbeitung in Shared-Nothing-Systemen ?

Aufgabe 18-4: Paralleler Hash-Join mit Überlaufbehandlung an Datenknoten

Geben Sie analog zu Kap. 18.3.3 einen Algorithmus für einen parallelen Hash-Join in Shared-Nothing-Systemen an, bei dem die Überlaufbehandlung gemäß dem GRACE-Ansatz an den Datenknoten erfolgt.

Aufgabe 18-5: Paralleler Hash-Join bei Shared-Everything

Wie können die Algorithmen zur parallelen Hash-Join-Berechnung für Shared-Everything abgewandelt werden?

Aufgabe 18-6: Daten-Skew

Der Join zwischen R und S soll parallel auf 5 Join-Prozessoren berechnet werden. R umfasse 500, S 1000 Tupel, der Wertebereich für das Join-Attribut liege zwischen 1 und 1000. Der Wert 1 soll in R und S je 200-mal auftreten, der Wert 2 100-mal in R und 600-mal in S; alle übrigen Werte sollen höchstens einmal pro Relation vorkommen.

Welche Skew-Effekte sind zu erwarten, wenn eine Hash-Funktion zur Umverteilung der Tupel verwendet wird ? Welche Bereichspartitionierung vermeidet diese ?

Aufgabe 18-7: Parallele Änderungstransaktionen bei Shared-Disk

Intra-Query-Parallelität wird in derzeitigen Shared-Disk-Implementierungen nur für lesende DB-Operationen geboten. Welche Probleme bestehen bei der Parallelisierung von Update-Transaktionen bzw. -Operationen ? (Hinweis: Welche Auswirkungen ergeben sich für die Transaktionsverwaltung und Kohärenzkontrolle ?)

Teil VI
Systemüberblicke

19 Existierende Mehrrechner-Datenbanksysteme

Viele der in diesem Buch behandelten Konzepte zur verteilten und parallelen Transaktions- und Datenbankverarbeitung finden sich schon in kommerziell verfügbaren Produkten. Dies soll in diesem abschließenden Kapitel verdeutlicht werden, in dem einige existierende Mehrrechner-Datenbanksysteme überblicksartig vorgestellt werden. Die Berücksichtigung bzw. Nichtberücksichtigung verschiedener Systeme stellt keine Wertung dar, sondern geht primär auf die Verfügbarkeit von beschreibenden Systemunterlagen zurück. Da der Leistungsumfang der Systeme stetigem Wandel unterliegt, kann die Beschreibung zudem nur einen bestimmten Entwicklungsstand (1994) dokumentieren.

Unsere Beschreibung berücksichtigt die Produkte von zwölf Herstellern (IBM, Oracle, DEC, ASK/Ingres, Informix, Sybase, Tandem, NCR/Teradata, UniSQL, Computer Associates, Cincom und SNI). Abschließend werden einige Beobachtungen zusammengefaßt.

19.1 IBM

IBM bietet für jede seiner Hardware- und Betriebssystem-Plattformen Produkte zur Transaktions- und Datenbankverwaltung an. Der aus dem Mainframe-Bereich stammende TP-Monitor CICS wurde bzw. wird dabei auf alle Plattformen portiert (CICS/ESA, CICS/MVS, CICS/VSE, CICS/400, CICS/6000, CICS OS/2), ebenso auf Plattformen anderer Hersteller (HP). Weiterhin ist für jede Plattform ein relationales DBS vorhanden, nämlich DB2/MVS, DB2/6000, DB2/2, SQL/DS und SQL/400. DB2 wurde auch auf einige Unix-Systeme anderer Hersteller portiert (HP, Sun). Das nicht-relationale, hierachische IMS-Datenbanksystem ist auf Großrechner beschränkt und wird aufgrund seiner großen Verbreitung weiterhin

unterstützt. IMS ist ein integriertes DB/DC-System, bestehend aus dem DBS IMS DB und dem TP-Monitor IMS TM (Transaction Manager). IMS DB kann jedoch auch von CICS-Anwendungen genutzt werden.

Zur verteilten und parallelen Datenbankverarbeitung bietet IBM eine ganze Palette von Möglichkeiten:

- Eine *verteilte Transaktionsverarbeitung* wird bereits seit 1978 vom TP-Monitor CICS unterstützt [Wi89]. Die Realisierung eines globalen Commit-Protokolls auf Basis von LU6.2 erfolgte dabei um Jahre früher als in Produkten anderer Hersteller. CICS bietet alle in Kap. 11.2 behandelten Verteilformen, nämlich Transaktions-Routing, programmierte Verteilung (Distributed Transaction Processing) sowie - für IMS-Datenbanken - die Verteilung einzelner DB-Operationen (Function Request Shipping). Durch die Verfügbarkeit von CICS auf zahlreichen Plattformen wird eine Interoperabilität in heterogenen Umgebungen ermöglicht. Zudem kann durch die Verwendung des CICS-API eine hohe Portabilität der Anwendungen erreicht werden.
  Der TP-Monitor IMS TM erlaubt ein Transaktions-Routing zwischen verschiedenen IMS-Installationen im Rahmen des sogenannten Multiple Systems Coupling (MSC).
- Verteilte und heterogene DBS werden für die SQL-Datenbanksysteme von IBM im Rahmen von IBM *DRDA* (Distributed Relational Database Architecture) unterstützt, die bereits in Kap. 11.4.6 behandelt wurde.
- Die Produkte *DataPropagator* und *DataRefresher* gestatten die Definition und asynchrone Aktualisierung von DB-Schnappschüssen (Schnappschuß-Replikation, Kap. 9.4). Zur Extraktion und Aktualisierung der Daten bestehen umfassende und flexible Definitionsmöglichkeiten. So kann für relationale DBS der volle SQL-Sprachumfang zur Ableitung der Daten verwendet werden. Weiterhin besteht die Möglichkeit, Daten zwischen IMS- und DB2-Datenbanken auszutauschen, wobei Änderungen synchron in beide Richtungen propagiert werden können.
- Für IMS-Datenbanken wird mit der neuen Komponente *Remote Site Recovery* eine Katastrophen-Recovery unterstützt, wobei die Log-Daten vom Primärsystem asynchron an ein geographisch entferntes Backup-System übertragen werden (Kap. 9.5). Im Primärsystem kann dabei auch IMS Data Sharing (s.u.) eingesetzt werden.
- IMS verfolgte als erstes allgemeines DBS den Shared-Disk-Ansatz, und zwar seit 1981 mit *IMS Data Sharing*. Der Shared-Disk-Ansatz wird aufgrund der neuen Architekturen *Parallel Sysplex* und *Parallel Transaction Server* für IMS sowie DB2 künftig noch eine weit größere Rolle spielen (s.u.).
- Für DB2/MVS wird mit dem neuen *Parallel Query Server* Intra-Query-Parallelität auf Basis einer Shared-Disk-Architektur unterstützt. Für den Unix-Bereich (DB2/6000) ist eine parallele DB-Verarbeitung nach dem Shared-Nothing-Ansatz geplant.

Im folgenden beschreiben wir kurz die bisherige Shared-Disk-Unterstützung von IMS sowie DB2 und dem Spezialbetriebssystem TPF. Danach behandeln wir die im Frühjahr 1994 neu angekündigten Shared-Disk-Konfigurationen *Parallel Sysplex* und *Parallel Transaction Server,* die zunächst von IMS genutzt werden und auf Inter-Transaktionsparallelität beschränkt sind. Abschließend gehen wir auf parallele Versionen von DB2 ein, insbesondere den *Parallel Query Server*, bei denen die Unterstützung von Intra-Query-Parallelität im Vordergrund steht.

19.1.1 Bisherige Shared-Disk-Systeme von IBM

IMS Data Sharing [SUW82, Yu87] wurde 1981 eingeführt und kann auf zwei Arten verwendet werden. Beim *DB-Level-Sharing* bilden ganze Datenbanken (Dateien) die Einheit des Teilens, wobei jedoch nur ein lesender Zugriff von mehr als einem Knoten (IMS-System) möglich ist; Änderungen sind stets auf einen Rechner beschränkt*. In diesem, für den allgemeinen DB-Betrieb zu restriktiven Fall besteht keine Einschränkung hinsichtlich der Rechneranzahl. Die volle Unterstützung des Shared-Disk-Ansatzes wird beim *Block-Level-Sharing* möglich, wobei Zugriffe zwischen den Rechnern auf Blockebene (Seitenebene) synchronisiert werden. Die Synchronisation erfolgt über das in Kap. 14.5 skizzierte Pass-the-Buck-Protokoll und ist auf zwei Rechner beschränkt. Im Rahmen von Parallel Sysplex und Parallel Transaction Server erfolgt eine wesentlich effizientere, hardwareunterstützte Synchronisation, wobei bis zu 32 Rechner genutzt werden können (s.u.).

DB2/MVS unterstützt seit Version 2.3 ein beschränktes DB-Sharing, ähnlich dem IMS DB-Level Sharing. Dabei können bestimmte DB-Dateien (sogenannte Table Spaces) nur lesend von mehreren Knoten bearbeitet werden; Änderungen sind auf einen Rechner beschränkt. Eine weitergehende Unterstützung des Shared-Disk-Ansatzes erfolgt zunächst im Rahmen des Parallel Query Server, jedoch wird DB2/MVS künftig auch die Architekturen Parallel Sysplex und Parallel Transaction Server nutzen.

Ein Disk-Sharing von bis zu acht Großrechnern kann in dem auf Transaktionsverarbeitung spezialisierten Betriebssystem ***TPF*** (Transaction Processing System) verwendet werden. TPF wird - unter verschiedenen Bezeichnungen - bereits seit den sechziger Jahren zur Realisierung von Reservierungssystemen eingesetzt, insbesondere von Fluggesellschaften [Sc87]. Die Einfachheit dieses Systems erlaubt weit kürzere Pfadlängen für Systemfunktionen sowie darauf laufende Anwendungen, die zumeist in Assembler realisiert sind. Damit können auf einer bestimmten Hardware-Plattform weit höhere Transaktionsraten als mit einem allgemeinen Betriebssystem und DBS erzielt werden. Andererseits führt die geringe Funktionalität zu sehr hohen Entwicklungs- und Wartungskosten für Anwendungssysteme. TPF nutzt die Platten-Kontroller zur Realisierung eines einfachen Sperrprotokolls (Kap. 13.4.2) sowie zur globalen Pufferung von DB-Sätzen. Zur Umgehung von Plattenengpässen können die Sätze einer Datei gemäß einem Round-Robin-Verfahren unter mehrere Platten verteilt werden (Unterstützung von Auftragsparallelität für hohe E/A- und Transaktionsraten). TPF erlaubt ferner ein Transaktions-Routing, bei dem der Zielrechner für eine Transaktionsbearbeitung aus bestimmten Werten der Eingabenachricht bestimmt wird (z.B. Flug- oder Kontonummer).

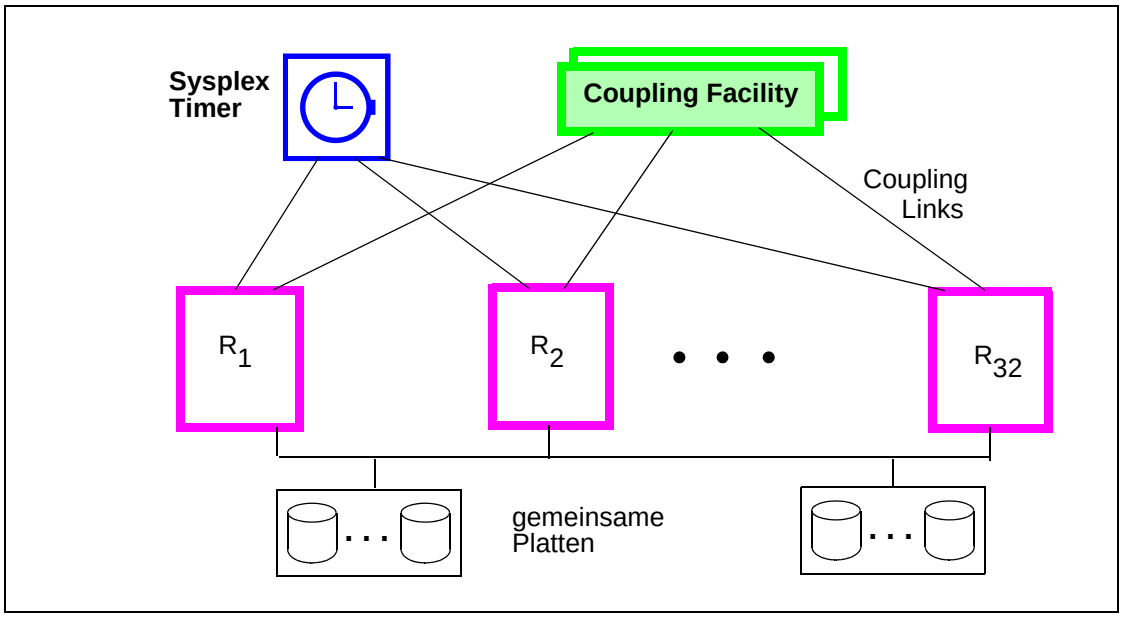
---

* Während des Änderungsbetriebes wird in den anderen Rechnern zudem kein konsistentes Lesen unterstützt.

19.1.2 Parallel Sysplex und Parallel Transaction Server

Parallel Sysplex ist eine nah gekoppelte Shared-Disk-Architektur, die 1994 von IBM für S/390-Umgebungen (MVS-Betriebssystem) eingeführt wurden. Sie verwendet als Knoten traditionelle IBM-S/390-Großrechner und stellt eine Erweiterung der bereits 1990 eingeführten Sysplex-Architektur (*Sys*tem Com*plex*) dar. Die nahe Kopplung erfolgt über eine spezielle *Coupling Facility*, auf die alle Rechner mit eigenen Maschinenbefehlen synchron zugreifen und welche u.a. zur globalen Sperrverwaltung verwendet wird (s.u.). *Parallel Transaction Server* basiert auf der Parallel-Sysplex-Architektur, verwendet jedoch neuere S/390-CMOS-Mikroprozessoren, welche voll kompatibel zu den Großrechnern sind, jedoch eine weit bessere Kosteneffektivität ermöglichen*.

Abb. 19-1: Aufbau des IBM Parallel Sysplex



Parallel Sysplex

Der Grobaufbau der Parallel-Sysplex-Architektur [Sy94] ist in Abb. 19-1 gezeigt. Sie umfaßt bis zu 32 S/390-Großrechner mit dem MVS-Betriebssystem, welche über Kanalverbindungen (channel to channel) gekoppelt sind. Dabei kann jeder Rechner ein Multiprozessor sein von derzeit mit bis zu 10 Prozessoren. Die Kommunikation erfolgt über Nachrichtenaustausch, wobei zur Reduzierung des Kom-

* Nach [IBM94] erfordern die in derzeitigen Großrechnern verwendeten ECL-Chips etwa 10-fach höhere Entwicklungskosten als für CMOS-Prozessoren, sind jedoch nur noch rund 2,5-mal schneller. Der Leistungsvorsprung nimmt zudem ständig ab, da für CMOS-Prozessoren höhere Leistungssteigerungen erzielt werden. Diese Prozessoren benötigen zudem keine Wasserkühlung bzw. Klimatisierung sowie erheblich weniger Stellfläche und Strom. Die Massenproduktion von CMOS-Chips verschafft zusätzliche Kostenvorteile. Aus diesen Gründen ist absehbar, daß künftig auch Großrechner komplett auf CMOS-Basis gefertigt werden.

munikationsaufwandes spezielle MVS-Dienste (Cross-system Extended Services, XES) genutzt werden. Die Anbindung der Platten an die Verarbeitungsrechner geschieht über Glasfaserverbindungen im Rahmen der ESCON-E/A-Architektur [Gr92]. Der Sysplex ist durch zwei spezielle Hardware-Einheiten gekennzeichnet, den Sysplex Timer sowie die Coupling Facility, die mit allen Rechnern über Glasfaserleitungen verbunden sind. Der *Sysplex Timer* stellt allen Rechnern die aktuelle Uhrzeit zur Verfügung, womit u.a. die Log-Daten der Rechner gekennzeichnet werden, um ihr korrektes Mischen zu einem globalen Log zu ermöglichen (Kap. 13.3.4). Die Coupling Facility dient zur schnellen Realisierung Shared-Disk-spezifischer Kontrollaufgaben. Aus Fehlertoleranzgründen können von beiden Komponenten mehrere Exemplare konfiguriert werden. Mehrere Coupling Facilities dienen auch zur Umgehung von Engpässen, da sie im Normalbetrieb gleichberechtigt nutzbar sind.

Die *Coupling Facility (CF)* entspricht einem gemeinsamen Halbleiterspeicher, der nicht-flüchtig ausgelegt sein kann und der von speziellen Mikroprozessoren verwaltet wird. Die Glasfaserverbindungen zwischen CF und Verarbeitungsrechner (coupling links) sind weit leistungsfähiger als ESCON-Verbindungen, um einen synchronen Speicherzugriff zu gestatten. Dabei werden Distanzen von bis zu 3 km zugelassen. Intern besteht die CF aus drei Bereichen, welche für unterschiedliche Aufgaben genutzt werden:

- Ein Sperrbereich (lock structure) dient zur Verwaltung globaler Sperrtabellen, mit denen eine systemweite Synchronisation auf gemeinsamen Ressourcen realisiert wird.
- Ein Pufferbereich (cache structure) ermöglicht die globale Speicherung von Datenseiten in einem schnellen Zwischenspeicher. Zudem werden Funktionen zur Kohärenzkontrolle angeboten.
- Ein Listenbereich (list structure) gestattet die Verwaltung globaler Listenstrukturen, insbesondere für gemeinsame Auftragswarteschlangen und globale Statusinformationen. Damit wird vor allem die Realisierung einer dynamischen Lastbalancierung erleichtert.

Die CF-Zugriffe erfolgen durch MVS, das seinerseits eine Reihe höherer Dienste Subsystemen wie dem DBS oder TP-Monitoren zur Verfügung stellt. Dies macht diese Hardware-Einheit für zahlreiche Zwecke nutzbar und erleichtert die CF-Nutzung, wenngleich auf Kosten erhöhter Zugriffszeiten. Von den Datenbanksystemen wird der Parallel-Sysplex zunächst von IMS Data Sharing genutzt; DB2 Data Sharing soll ab 1995 folgen. In beiden Fällen erfolgen Sperrbehandlung und Kohärenzkontrolle über die CF. IMS basiert auf einem Force-Ansatz zur Propagierung von Änderungen, wobei geänderte Seiten zur Beschleunigung des Schreibvorganges in die CF geschrieben werden können. DB2 wird dagegen einen Noforce-basierten Ansatz verfolgen. Nähere Informationen zur Realisierung der Kohärenzkontrolle mit Hilfe der CF liegen zur Zeit nicht vor.

Zur dynamischen Lastbalancierung wird von den TP-Monitoren CICS und IMS TM künftig ein entsprechendes Transaktions-Routing unter Nutzung der CF unterstützt. Dabei erfolgt eine Zusammenarbeit mit einer neuen MVS-Komponente, dem Workload Manager (WLM), um aktuelle Auslastungsinformationen sowie globale Leistungsvorgaben bei der Lastverteilung zu berücksichtigen. Ein affinitätsbasiertes Routing wird nicht verfolgt, da mit der CF der Synchronisationsaufwand unabhängig vom Ausmaß rechnerspezifischer Lokalität gehalten werden kann (jedoch nicht der E/A-Aufwand). Die dynamische Lastbalancierung wird zunächst von CICS durch das neue Produkt CICSPlex/SM unterstützt; IMS TM soll ab 1996 folgen.

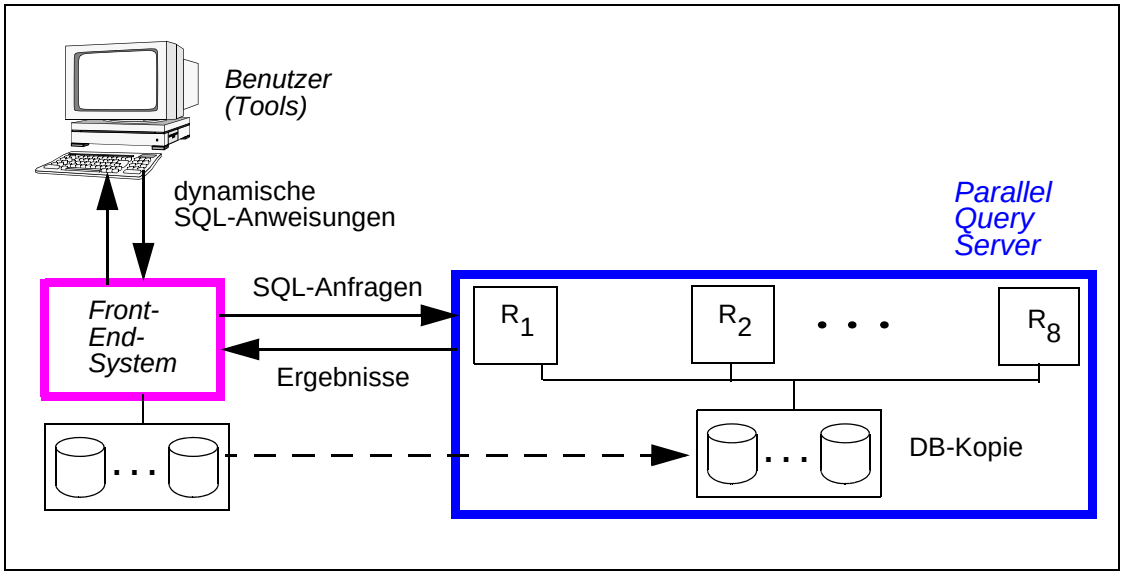
Parallel Transaction Server
Im Rahmen des S/390 Parallel Transaction Server können bis zu acht Rechnerknoten mit jeweils zwei bis sechs Mikroprozessoren eingesetzt werden. Die Shared-Disk-Unterstützung erfolgt wie für Parallel Sysplex über eine im Server-System integrierte Coupling Facility sowie einen (externen) Sysplex Timer. Das System kann als eigenständiger Server genutzt werden sowie als Knoten in einer Parallel-Sysplex-Konfiguration. In letzterem Fall wird somit der gemeinsame Einsatz von Großrechnern und Mikroprozessoren zur Transaktions- und DB-Verarbeitung möglich. Die DB-seitige Nutzung des Parallel Transaction Server erfolgt zunächst wiederum für IMS Data Sharing. Bei Verfügbarkeit von DB2 Data Sharing für den Parallel Sysplex ist es auch auf dem Parallel Transaction Server einsetzbar.

19.1.3 Paralleles DB2

Die Parallelisierung komplexer SQL-Anweisungen wird für DB2 zur Zeit nur in Form von E/A-Parallelität unterstützt (für DB2/MVS seit V3.1, für DB2/6000 und DB2/2 ab Version 2). Dazu können Relationen über eine Bereichspartitionierung auf einem bestimmten Attribut über mehrere Platten verteilt gespeichert werden, von denen dann die Daten parallel gelesen werden (Zugriffsparallelität). Verarbeitungsparallelität wird nicht genutzt, d.h. die Verarbeitung einer Query erfolgt nur durch einen Prozessor. Diese Beschränkung entfällt für den neuen Parallel Query Server für DB2/MVS sowie die parallele Version von DB2/6000.

Parallel Query Server
Der S/390 Parallel Query Server (PQS) ist ein Mikroprozessor-Cluster ähnlich dem Parallel Transaction Server, welches bis zu acht Rechnerknoten (48 Prozessoren) umfassen kann. Allerdings basiert PQS noch auf dem ursprünglichen Sysplex (ohne Coupling Facility). Weiterhin wird der Query-Server als dediziertes Back-End-System ausschließlich zur parallelen Anfrageverarbeitung für DB2/MVS eingesetzt (Abb. 19-2). Die Query-Verarbeitung erfolgt dabei auf einer Kopie der Datenbank, die asynchron aktualisiert wird und auf die alle Rechner des Back-End-Systems gemäß dem Shared-Disk-Prinzip Zugriff haben.

Abb. 19-2: Parallel Query Server



In der ersten Version des Parallel Query Servers können nur lesende, dynamische SQL-Anfragen parallelisiert werden. Damit erübrigen sich im Back-End-System Synchronisation sowie Kohärenzkontrolle, so daß keine Coupling Facility benötigt wird. SQL-Anfragen, die vom Benutzer gestellt bzw. von Tools erzeugt werden, gelangen über ein S/390-Front-End-System an den Query-Server; Anfrageergebnisse nehmen den umgekehrten Weg (Abb. 19-2). Das Front-End-System verwaltet die Primär-Datenbank unter Kontrolle von DB2/MVS. Es führt die Optimierung der SQL-Anfragen durch und entscheidet, ob eine parallele Bearbeitung im Backend-System sinnvoll ist. Im Back-End-System nimmt auf einem der Rechner eine spezielle Scheduling-Komponente die Zerlegung der Query in parallele Teilanfragen vor und kombiniert die Teilergebnisse.

Paralleles DB2/6000

Für DB2/6000 unter dem AIX-Betriebssystem wird künftig ebenfalls Intra-Query-Parallelität unterstützt, allerdings auf Basis eines Shared-Nothing-Ansatzes [Fe94]. Die Ausführung soll dabei auf dem Unix-Parallelrechner SP2 (Scalable POWERparallel System) von IBM erfolgen, in dem bis zu 128 RS/6000-Prozessoren eingesetzt werden können. Es wird dabei eine umfassende Parallelisierung lesender und ändernder SQL-Anweisungen sowie von Dienstprogrammen unterstützt. Daneben wird zur Erlangung hoher Transaktionsraten auch Inter-Transaktionsparallelität eingesetzt.

19.2 Oracle

Oracle wurde 1979 von der gleichnamigen Firma als erstes relationales SQL-Datenbanksystem überhaupt eingeführt. Es ist auf einer Vielzahl von Hardware- und Betriebsystemplattformen verfügbar, vom PC bis zum Mainframe. Marktführend

ist Oracle insbesondere im Unix-Bereich. Eine verteilte DB-Verarbeitung wird seit 1986 unterstützt. Seit 1990 wird zudem mit "Parallel Server" eine parallele DB-Verarbeitung nach dem Shared-Disk-Ansatz verfolgt.

19.2.1 Verteilte DB-Verarbeitung

Oracle gestattet, in einer Transaktion und auch innerhalb einer SQL-Operation auf mehrere unabhängige Datenbanken zuzugreifen (verteilte Join-Berechnung u.ä.). Die Übertragung von Operationen und Ergebnissen erfolgt über die Kommunikationskomponente *SQL*NET*, die eine Reihe unterschiedlicher LAN- und WAN-Kommunikationsprotokolle unterstützt (TCP/IP, SNA, DECnet etc.). SQL*NET ist dabei auf jedem Client- und Server-Rechner zu installieren. Die Übersetzung und Optimierung der SQL-Anweisungen erfolgt stets auf Server-Seite durch eines der Oracle-DBS, das mit anderen DBS (DB-Servern) zusammenarbeitet. Für diese Zusammenarbeit sind zuvor sogenannte DB-Links zu spezifizieren. Um systemweit eindeutige Objektnamen zu bekommen, werden sie mit dem jeweiligen Knotennamen erweitert. Durch Definition von Synonymen kann dem Benutzer gegenüber jedoch Ortstransparenz erreicht werden (Kap. 4.4).

Verteilte Änderungstransaktionen sind erst seit Einführung eines verteilten Zwei-Phasen-Commit-Protokolls in Oracle V7 (1992) möglich. Seit dieser Version wird auch eine Schnappschuß-Replikation unterstützt.

Für den Zugriff auf DBS anderer Hersteller bietet Oracle eine Reihe von Gateways im Rahmen der Produktfamilie SQL*CONNECT. *SQL*CONNECT* läuft dabei (zusätzlich zu SQL*NET) stets auf dem Server-Rechner des Fremd-DBS. Anwendungen können damit auf das Fremd-DBS quasi wie auf eine Oracle-Datenbank zugreifen. Beim Gateway-Einsatz sind drei Zugriffsformen zu unterscheiden:

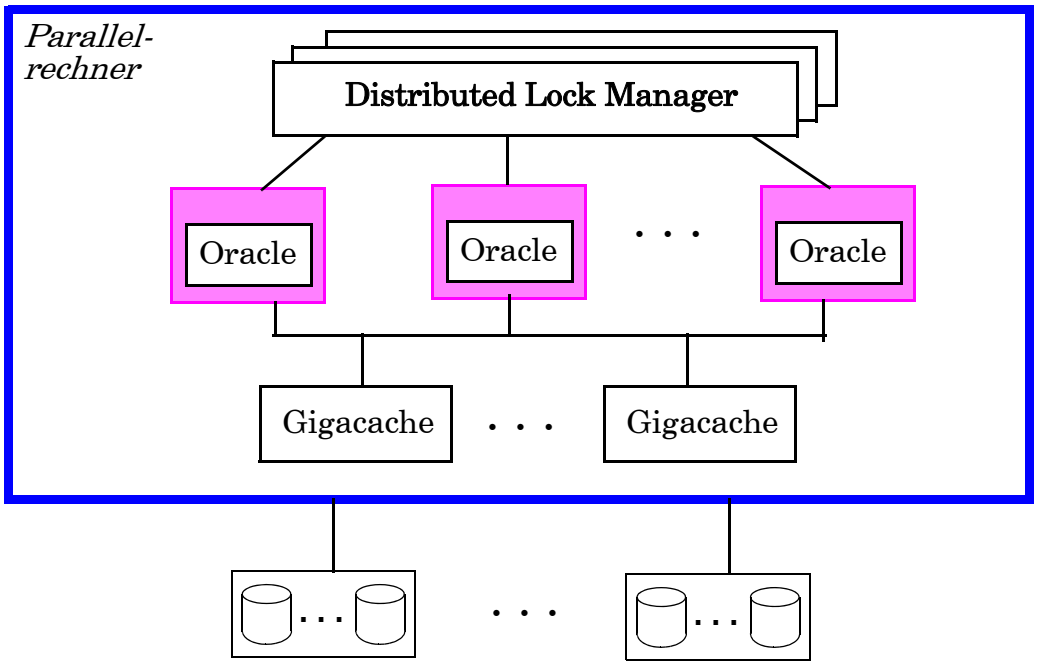
- Oracle-Tools können i.a. direkt über ein SQL*CONNECT-Gateway auf SQL-DBS anderer Hersteller zugreifen, also ohne Involvierung eines Oracle-DBS. SQL*CONNECT erzeugt dabei dynamische SQL-Anweisungen für das Fremd-DBS. Für DB2 und Tandem NonStop SQL sind neben lesenden auch ändernde Zugriffe möglich. Es handelt sich dabei jedoch um lokale Transaktionen, da jeweils nur ein DBS pro Transaktion involviert ist.
- Ein indirekter Zugriff liegt vor, wenn ein Oracle-Werkzeug mit einem Oracle-DBS zusammenarbeit, von dem aus über DB-Links auf externe Datenbanken zugegriffen wird. In diesem Fall sind lediglich lesende Zugriffe auf die Fremd-Datenbank möglich, jedoch können in einer SQL-Operation Daten mehrerer Datenbanken bearbeitet werden. Auch verteilte Sichten (Views) werden unterstützt. Mit dieser Einsatzform kann auch auf nicht-relationale DBS sowie Dateisysteme zugegriffen werden.
- Direkter Zugriff auf mehrere Datenbanken ist möglich über Anwendungsprogramme mit eingebettetem SQL-Anweisungen. Dabei sind von der Anwendung Verbindungen mit den einzelnen DB-Servern aufzubauen, über die dann die SQL-Anweisungen verschickt werden. Die Verteileinheiten sind somit ganze DB-Operationen.

Oracle-DBS können auch zur verteilten Transaktionsverarbeitung unter Kontrolle verschiedener TP-Monitore eingesetzt werden. Zudem unterstützt Oracle die XA-Schnittstelle von X/Open DTP (Kap. 11.4.1), so daß es in ein verteiltes Commit-Protokoll eingebunden werden kann. Ferner wird Microsofts ODBC-Schnittstelle (Kap. 11.4.5) über entsprechende Treiber abgedeckt, worüber Client-Anwendungen in standardisierter Weise auf Oracle-Datenbanken zugreifen können.

19.2.2 Oracle Parallel Server

Die Unterstützung von Shared-Disk-Konfigurationen durch *Oracle Parallel Server (OPS)* [Or93] begann in Version 6.2 von Oracle. OPS war zunächst auf Vax-Cluster-Systemen lauffähig, allerdings mit relativ geringer Leistungsfähigkeit. Mittlerweile wurde OPS auf eine ganze Reihe von Plattformen portiert, insbesondere mehrere Cluster-Architekturen (Sequent, Encore, Pyramid etc.) und Parallelrechner (nCUBE, Meiko, KSR, etc.). Großes Aufsehen erzielte 1991 ein TPC-B-Benchmark auf einer nCUBE2-Konfiguration mit 64 Rechnern, in dem über 1000 Transaktionen pro Sekunde zu sehr geringen Kosten erzielt wurden. Dies war zu diesem Zeitpunkt ein absolutes Spitzenergebnis, das erst zwei Jahre später übertroffen wurde. OPS erlaubte zunächst nur Inter-Transaktionsparallelität; seit Oracle V7.1 wird für lesende SQL-Anweisungen auch eine erste Unterstützung von Intra-Query-Parallelität geboten [Li93].

Abb. 19-3: Oracle Parallel Server auf einem Parallelrechner



Die prinzipielle Struktur von OPS auf einem Parallelrechner ist in Abb. 19-3 gezeigt. Das Oracle-DBVS läuft dabei auf einer Reihe von Rechnerknoten. Zur Synchronisation wird ein verteiltes Sperrprotokoll (Distributed Lock Manager) auf de-

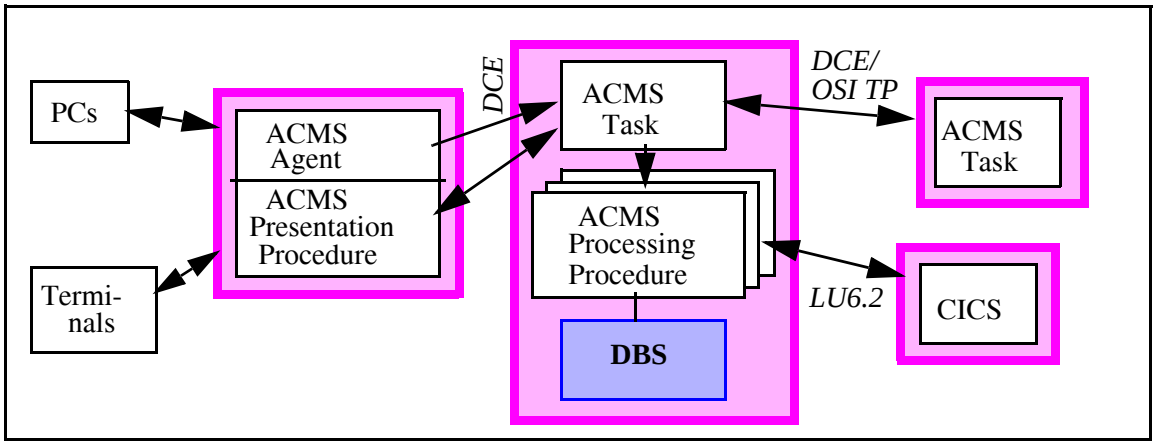
dizierten Rechnern verwendet (Kap. 14.1), wobei die Sperrzuständigkeit über eine Hash-Funktion unter den Knoten aufgeteilt ist. Der Hauptspeicher verbleibender Rechnerknoten kann optional als "Gigacache" genutzt werden, um DB-Seiten zwischenzupuffern. Die gepufferten Seiten können dann über das Verbindungsnetzwerk des Parallelrechners wesentlich schneller von diesem Cache als von Externspeicher angefordert werden. Die Kohärenzkontrolle bezüglich dieser Caches sowie der Puffer der Verarbeitungsrechner basiert auf dem Konzept der Haltesperren (Kap. 15.4). Zur Propagierung von Änderungen wird ein Noforce-Ansatz ("fast commit") mit Seitenaustausch über Externspeicher verfolgt. Ungeänderte Seiten werden jedoch auch direkt zwischen den Rechnern transferiert, um das E/A-Verhalten zu optimieren. Zur Reduzierung von Sperrkonflikten werden sowohl Satzsperren als auch ein Mehrversionen-Ansatz unterstützt, bei dem Lesetransaktionen auf veraltete Seiten zugreifen können.

19.3 DEC

DEC realisiert eine verteilte Transaktionsverarbeitung über seine beiden TP-Monitore *ACMS* und *DECintact* [SS91]. Insbesondere wird der Ansatz der programmierten Verteilung verfolgt, wobei durch Aufruf von Anwendungsfunktionen der Server-Rechner auf dort vorliegende Datenbanken zugegriffen wird. Die dabei verwendete Aufrufstruktur ist für ACMS in Abb. 19-4 verdeutlicht. Auf Client-Seite liegen Präsentationsdienste sowie ein ACMS-Agent vor, von dem aus sogenannte Tasks aufgerufen werden. Die ACMS-Tasks werden in einer eigenen Beschreibungssprache (task definition language) realisiert und spezifizieren den Ablauf einer verteilten Transaktion, insbesondere Transaktionsbeginn und -ende sowie welche Anwendungsfunktionen (processing procedures) auszuführen sind. Die Anwendungsfunktionen schließlich führen die DB-Zugriffe aus. Diese Modularisierung der Anwendungen gestattet hohe Freiheitsgrade für die Rechnerzuordnung der einzelnen Komponenten. ACMS kooperiert auch mit anderen TP-Monitoren (CICS, IMS TM, etc.), um auf deren Anwendungen und externe (heterogene) Datenbanken zuzugreifen. ACMS wird künftig stark auf bestehenden Standards aufbauen, insbesondere X/Open DTP (Kap. 11.4.1), MIA STDL (Kap. 11.4.2) und DCE (für Remote Procedure Calls).

Die Realisierung eines verteilten zwei-Phasen-Commit-Protokolls erfolgt durch eine spezielle Komponente des VMS-Betriebssystems *DECdtm* (Digital distributed transaction manager). Damit steht dieser Dienst sowohl den TP-Monitoren als auch den DBS zur Verfügung.

Abb. 19-4: Verteilte Transaktionsverarbeitung mit DEC ACMS [Tr93]



DEC bietet zwei DBS an, ein CODASYL-DBS namens *VAX DBMS* sowie das relationale System *Rdb**. Rdb kann in eine verteilte DB-Verarbeitung mit DBS und Dateisystemen anderer Hersteller (DB2, Oracle, Sybase, RMS, VSAM) eingebunden werden, auf die über Gateways der Produktfamilie *DB Integrator (DBI)* zugegriffen wird. Rdb selbst sowie die DBI-Gateways unterstützen über spezielle Treiber Microsofts ODBC-Protokoll (Kap. 11.4.5) für Lese- und Schreibzugriffe. Eine Schnappschuß-Replikation wird mit dem Rdb-Zusatzprodukt *DEC Data Distributor* realisiert. Die Zieldatenbanken, welche Kopien einer gesamten DB oder Untermengen davon sein können, werden von Rdb wie reguläre Datenbanken verwaltet, können jedoch nur lesend bearbeitet werden. Als Quelldatenbanken sind auch Fremd-Datenbanken möglich, auf die über Gateways zugegriffen wird.

DECs Vax-Cluster, welche die wohl am weitesten verbreitete Shared-Disk-Hardware-Architektur repräsentieren, werden softwaremäßig von VAX DBMS und Rdb unterstützt. Beide Systeme verwenden einen gemeinsamen DBS-Kern namens KODA, der auch zur Lösung der Shared-Disk-spezifischer Aufgaben dient (Pufferverwaltung, Kohärenzkontrolle etc.). Zur Synchronisation wird dabei auf den Distributed Lock Manager des VMS-Betriebssystems zurückgegriffen, der ein verteiltes Sperrprotokoll mit dynamischer GLA-Zuordnung realisiert (Kap. 14.4). Pufferinvalidierungen werden mit einer On-Request-Invalidierung über Versionsnummern (Kap. 15.3.1) erkannt. Die Update-Propagierung erfolgte zunächst gemäß einem Force-Ansatz, wurde jedoch auf Noforce umgestellt [LARS92].

19.4 ASK/Ingres

In den siebziger Jahren entstand an der Univ. Berkeley unter M. Stonebraker mit "University Ingres" ein erster Prototyp eines relationalen DBS (neben System R von IBM). Eine kommerzielle Ingres-Version für den Unix-Bereich wurde von der

* Rdb wird mittlerweile von Oracle vertrieben.

1980 gegründeten Firma Relational Technology Inc. vertrieben, die 1989 in Ingres Corp. umfirmierte. Seit Oktober 1990 gehört Ingres zur ASK-Gruppe.

Ingres unterstützte als eines der ersten kommerziellen DBS eine verteilte DB-Verarbeitung. Mit der bereits 1983 eingeführten Kommunikationskomponente *Ingres/Net* können Anwendungen auf Client-Rechnern SQL-Operationen an entfernte Server-Datenbanken stellen. Das föderative Mehrrechner-DBS *Ingres/Star* wurde 1986 eingeführt. Der Zugriff auf Fremd-DBS kann über eine Reihe von Gateways erfolgen, welche unter der Bezeichnung *Enterprise Access* zusammengefaßt werden. Daneben gibt es seit kurzem eine Komponente (*Replicator*) zur Realisierung einer Schnappschuß-Replikation. Das neuerdings als OpenIngres bezeichnete DBS unterstützt die XA-Schnittstelle und arbeitet darüber mit mehreren TP-Monitoren zur verteilten Transaktionsverarbeitung zusammen. Schließlich bietet Ingres eine Unterstützung des Shared-Disk-Ansatzes für Vax-Cluster.

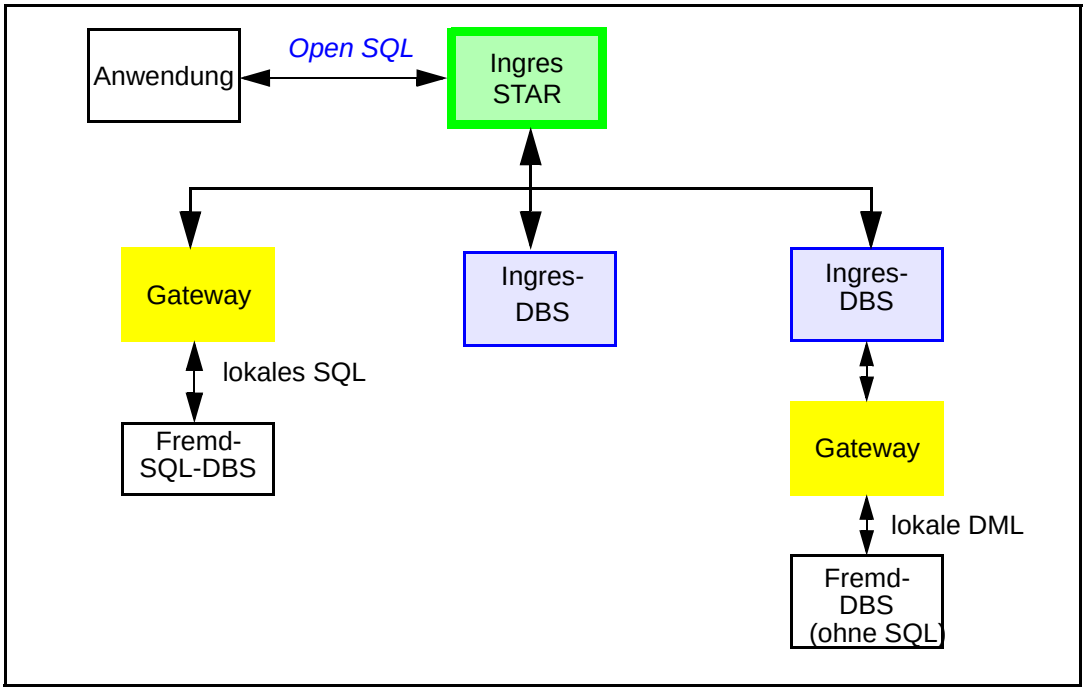
Ingres/Net ermöglicht den Zugriff auf Ingres-Datenbanken sowie - über Gateways - auf Fremd-DBS und Dateisysteme an entfernten Rechnern. Seit Ingres-Version 6.3 kann dabei in einer Transaktion auf mehrere Datenbanken zugegriffen werden. Das dann benötigte verteilte Zwei-Phasen-Commit-Protokoll mußte zunächst in der Anwendung ausprogrammiert werden, wodurch die Konsistenz der beteiligten Datenbanken von der Korrektheit der Anwendungsprogramme abhing! Dieses Manko wurde zwischenzeitlich durch ein "transparentes" Commit-Protokoll behoben.

Für den Zugriff auf heterogene Datenbanken bietet Ingres die Anfragesprache “Open SQL” an, welche aus einer Untermenge der gebräuchlichsten SQL-Dialekte gebildet wurde. Weiterhin wurde eine Menge generischer Fehler-Codes festgelegt, um den Anwendungen eine einheitliche Fehlerbehandlung auch beim Zugriff auf heterogene Datenbanken zu ermöglichen. Diese Festlegungen werden vom Ingres-DBS sowie den Ingres-Gateways beachtet. Gateways existieren nicht nur für SQL-DBS (z.B. DB2, Rdb), sondern auch für nicht-relationale DBS (IMS, HP Allbase) sowie Dateisysteme (RMS). In letzterem Fall sind die SQL-Operationen durch ein vorgesetztes Ingres-DBS zu verarbeiten, welches das Fremd-DBS nur für elementare Satzoperationen verwendet.

Ingres/Star ist eine zentralisierte Systemkomponente, über die SQL-Anweisungen an mehrere unabhängige DBS gestellt werden können (Abb. 19-5). Die Datenbanken werden in einer Föderation zusammengefaßt, wobei dem Benutzer mit einem gemeinsamen föderativen Schema eine weitgehende Verteilungstransparenz geboten wird. Darüber ist es auch möglich, in einer SQL-Operation auf mehrere Datenbanken zuzugreifen (z.B. verteilte Joins). Die Optimierung globaler Anfragen erfolgt durch Ingres/Star, ebenso die Zerlegung in lokal ausführbare Teilanfragen und das Mischen und Nachbearbeiten der Teilergebnisse. Weiterhin ist Ingres/Star für die Erkennung globaler Deadlocks sowie die Durchführung des verteilten Commit-Protokolls verantwortlich. Wie in Abb. 19-5 skizziert, kann auch Ing-

res/Star über Gateways auf relationale und nicht-relationale Fremd-DBS bzw. Dateisysteme zugreifen. Nicht gezeigt sind in der Abbildung die Ingres/Net-Komponenten, die in jedem der beteiligten Rechner zur Kommunikationsunterstützung vorliegen müssen.

Abb. 19-5: Einsatz von Ingres/Star



Ingres/Star erlaubt die Definition mehrerer DB-Föderationen mit jeweils einem eigenen föderativen Schema. Das föderative Schema legt im wesentlichen nur für jede Datenbank die Relationen fest, die an der Föderation beteiligt sein sollen. Eine Schemaintegration zur Auflösung semantischer Konflikte erfolgt jedoch nicht. Nachteilig ist daneben die Zentralisierung von Ingres/Star, wodurch Verfügbarkeit und Knotenautonomie stark eingeschränkt werden. Ferner wird diese Komponente leicht zum Flaschenhals des Systems, da sie sämtliche globalen Anfragen bearbeiten muß.

Die Zusatzkomponente *Ingres/Replicator* dient zur Erstellung und Wartung von Schnappschuß-Replikaten, die aus Ingres-Datenbanken oder aus über Gateways erreichbaren Fremd-Datenbanken abgeleitet werden [Zi93]. Bei der Aktualisierung der Kopien werden nur die seit der letzten Aktualisierung angefallenen Änderungen erfolgreich beendeter Transaktionen propagiert, so daß die Kopie stets transaktionskonsistent bleibt. Neben dem Datenaustausch zwischen zwei Datenbanken werden auch Mehrfach-Replikation (mehrere, verteilte Kopien einer Quelle) sowie kaskadierende Replikationen unterstützt. Als problematisch ist die Möglichkeit wechselseitiger Replikationen einzustufen, bei der Änderungen in der Quell- und Zieldatenbank möglich sind. Denn da die Replikate außerhalb der DBS

verwaltet werden, können Änderungskonflikte (wenn also dasselbe Objekt unkoordiniert in beiden Kopien geändert wird) zwar erkannt, müssen jedoch manuell behoben werden!

19.5 Informix

Das relationale DBS Informix wurde in der SQL-Version 1985 eingeführt und hat vor allem im Unix-Bereich weite Verbreitung gefunden [Pe93]. Eine verteilte DB-Verarbeitung wird seit 1990 mit *Informix-Star* unterstützt; Kommunikationsunterstützung bietet die bereits früher eingeführte Komponente *Informix-Net*. Informix-Star gestattet die Verarbeitung verteilter DB-Operationen auf unabhängigen Datenbanken ähnlich wir Ingres-Star oder Oracle. Ortstransparenz für den Benutzer wird wiederum durch die Verwendung von Synonymen erreicht. Seit Informix V5 ist über ein (transparentes) verteiltes Zwei-Phasen-Commit auch die Bearbeitung verteilter Änderungstransaktionen möglich. Globale Deadlocks werden über einen Timeout-Ansatz aufgelöst.

Der Zugriff auf Fremd-Datenbanken erfolgt wiederum über Gateways. Mit DB2 und anderen IBM-DBS ist durch die Unterstützung von DRDA eine direkte Zusammenarbeit möglich. Daneben unterstützt Informix die XA-Schnittstelle zur verteilten Transaktionsverarbeitung sowie ODBC.

Der seit Informix V6 verfügbare *Data Replication Server* gestattet das Führen einer DB-Kopie, mit der nach einem Systemausfall die Verarbeitung fortgesetzt werden kann (Katastrophen-Recovery, Kap. 9.5). Hierzu werden die Log-Daten zwischen Primär- und Backup-System ausgetauscht und auf der DB-Kopie angewendet. Die Aktualisierung der Kopie erfolgt wahlweise synchron oder asynchron. Im Normalbetrieb können Lesezugriffe zur Entlastung des Primärsystems auf der DB-Kopie abgewickelt werden.

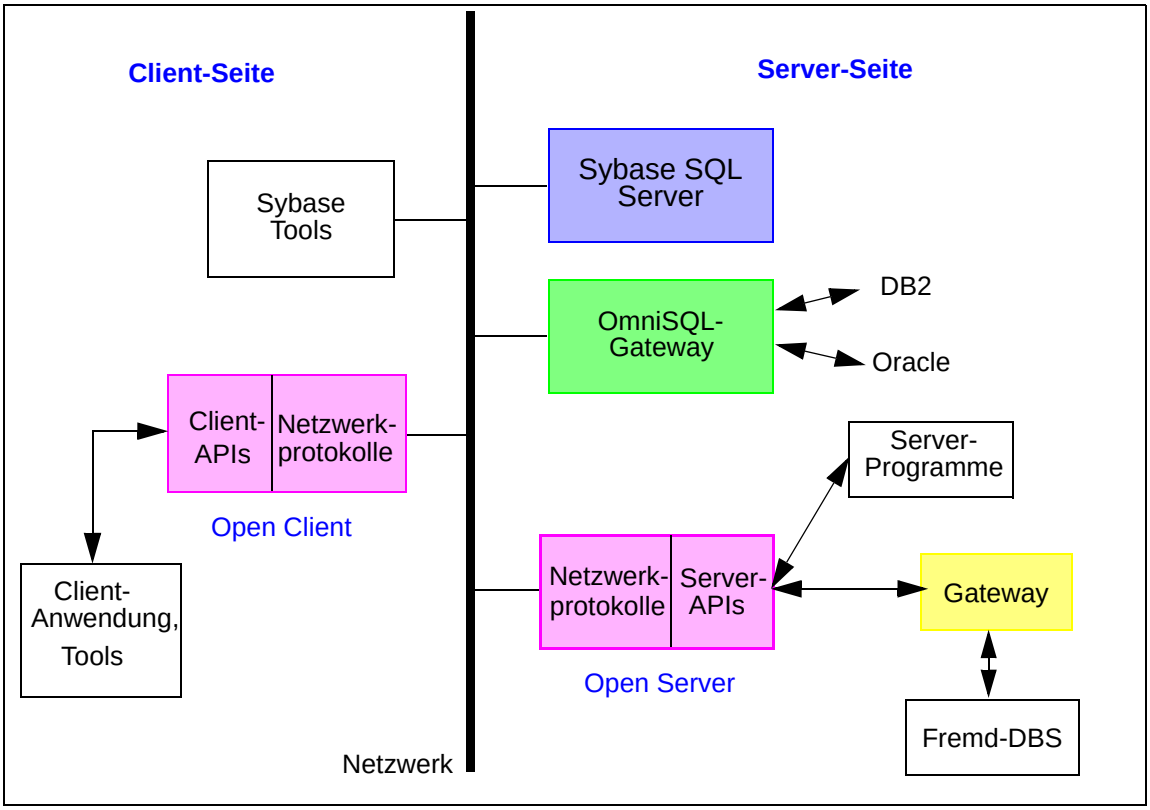
Seit Version 6 können Multiprozessoren (Shared Everything) zur parallelen Indexgenerierung genutzt werden [Dav92]. Eine weitergehende Intra-Query-Parallelisierung soll in V7 folgen.

19.6 Sybase

Sybase wurde 1986 gegründet und konnte sich als relationales DBS v.a. im Unix-Markt etablieren. Von Beginn an wurde konsequent eine Client/Server-Transaktionsverarbeitung unterstützt. Mit dem 1993 eingeführten *System 10* wird eine Ausweitung des Client/Server-Ansatzes auf Unternehmensebene (enterprise client/server computing) angestrebt [Co93]. Damit soll nicht nur im lokalen Umfeld, sondern auch in geographisch verteilten Umgebungen eine Client/Server-Transaktionsverarbeitung erfolgen. Weiterhin müssen Anwendungen und Datenbanken anderer Unternehmensbereiche bzw. von zentralen Unternehmens-Servern (meist

Großrechner) eingebunden werden. Zudem muß eine sehr hohe Leistungsfähigkeit, Verfügbarkeit und Erweiterbarkeit unterstützt werden. Dies soll durch System 10 mit einer Reihe von neuen Komponenten erreicht werden, insbesondere dem *OmniSQL-Gateway*, dem *Replication Server* sowie dem *Navigation Server*.

Abb. 19-6: Einsatzumgebung von Sybase



Die beim Einsatz von Sybase beteiligten Systemkomponenten sind in Abb. 19-6 gezeigt. Dabei können Anwendungen und Tools auf Client-Seite auf DB-Server und Anwendungen auf Server-Seite zugreifen. Sybase-Tools können quasi direkt mit dem Sybase-DBS, *SQL Server* genannt, zusammenarbeiten, da sie bereits entsprechende Kommunikationsfunktionen enthalten. Ansonsten erfolgt die Interoperabilität über von Sybase unterstützte Client/Server-Interfaces, mit denen die Zusammenarbeit in heterogenen Umgebungen ermöglicht wird. Zu den Client/Server-Schnittstellen zählen:

- *Open Client*
  Diese Komponente bieten Anwendungen und Tools mehrere APIs für den transparenten Zugriff auf Server-Dienste, insbesondere SQL, entfernte Prozeduraufrufe sowie zur Transaktionsverwaltung. Neben der Sybase-Anfragesprache TransactSQL wird dabei auch ODBC unterstützt. Die Kommunikationskomponente von Open Client unterstützt mehrere Netzwerkprotokolle (TCP/IP, DECnet, NetWare etc.).
- *Open Server*
  Auf Server-Seite erlaubt diese Komponente die Erstellung eigener Server-Dienste,

auf die Client-Anwendungen über ihre APIs zugreifen können. Dazu bietet Open Server neben der Kommunikationsunterstützung spezielle Server-APIs zur Entgegennahme von DB-Operationen und Prozeduraufrufen bzw. zur Rückgabe von Ergebnissen. Eine spezielle Version des Opern Servers ermöglicht die Zusammenarbeit mit CICS und damit den Zugriff auf Mainframe-Anwendungen.

- *Open Gateways*
  Für den Zugriff auf Fremd-DBS werden - zusätzlich zum Open Server - noch Gateways benötigt, welche eine Abbildung zwischen TransactSQL und der Anfragesprache des jeweiligen Systems vornehmen. Solche Gateways bestehen für DB2, Oracle, Ingres, Informix, Rdb und das Dateisystem RMS. Mit dem neuen OmniSQL-Gateway kann daneben in integrierter Weise auf mehrere unabhängige Datenbanken zugegriffen werden (s.u.).

Sybase bot als erstes System das Konzept der gespeicherten Prozeduren (stored procedures) an, um Anwendungsfunktionen im DBS zu verwalten. Diese Prozeduren werden vollständig in der Sprache TransactSQL erstellt, welche neben SQL-Befehlen auch lokale Programmvariablen und allgemeine Kontrollstrukturen (IF, WHILE etc.) unterstützt. Der Aufruf einer solchen Prozedur erfolgt in einer Anweisung, einem sogenannten Datenbank-RPC. Innerhalb der Prozeduren ist es möglich, weitere "stored procedures" aufzurufen, die zu einem anderen SQL-Server gehören können. Damit wird das Konzept der programmierten Verteilung (Kap. 11.2) nicht nur zwischen Client und Server, sondern auch zwischen Server-Rechnern unterstützt.

Nachteilig ist, daß Sybase noch kein transparentes Zwei-Phasen-Commit anbietet, sondern die verteilte Commit-Bearbeitung weitgehend von den Anwendungsprogrammen zu kontrollieren ist. Insbesondere fungiert die Anwendung als Commit-Koordinator und muß beide Commit-Phasen durch explizite Aufrufe an die an der Transaktion beteiligten Server initiieren. Wird einer der Server bei der Commit-Behandlung "vergessen" oder ihm ein falsches Commit-Ergebnis mitgeteilt, ergeben sich inkonsistente Datenbankzustände! Ein weiteres Problem ist, daß die Ausführung der Commit-Koordinierung auf unsicheren Client-Rechnern (Workstations, PCs) stattfindet. Um zumindest den aktuellen Stand des Commit-Protokolls auf einer “sicheren” Seite zu vermerken, unterstützt Sybase die Einrichtung eines Commit-Services auf einem der Server. Allerdings liegt es wieder in der Verantwortung des Programmierers, diesen Service einzurichten und ihm das Commit-Ergebnis mitzuteilen (der Commit-Service protokolliert daraufhin die Commit-Entscheidung). Nur im Fehlerfall (Server-Ausfall, Timeout) wenden sich die involvierten Datenbank-Server an den Commit-Service, um den Zustand der Transaktion zu erfragen.
Das skizzierte Commit-Protokoll ist zudem auf Sybase-DBS beschränkt. Da Sybase SQL-Server die XA-Schnittstelle unterstützt, kann jedoch durch Nutzung eines XA-konformen TP-Monitors ein für die Anwendungen transparentes Commit-Protokoll erreicht werden.

Das neue Sybase *OmniSQL-Gateway* vereint die Funktionalität von Open Server sowie mehreren Gateways (zu DB2, Oracle, Rdb etc.). Damit wird es möglich, in einer SQL-Anweisung auf Daten mehrerer unabhängiger Datenbanken zuzugreifen, insbesondere um verteilte Join-Berechnungen vorzunehmen. Im OmniSQL-Gateway wird dazu ein globaler Katalog verwaltet, der den Benutzern ein hohes Maß an Verteilungstransparenz bietet. Mit dem globalen Katalog wird nicht nur die Datenlokation der einzelnen Objekte verborgen, sondern auch eine Angleichung unterschiedlicher Namenskonventionen und Datentypen vorgenommen. Damit wird die Funktionalität eines föderativen DBS erreicht. Eine Beschränkung besteht darin, daß der globale Katalog nicht mit den lokalen Kataloge der einzelnen DBS synchronisiert ist, so daß der Administrator durch Anwendung eines Tools lokale Schemaänderungen in den globalen Katalog bringen muß. Zudem sind Änderungen auf eine Datenbank beschränkt. Da das OmniSQL-Gateway an seiner Anwenderschnittstelle TransactSQL anbietet, werden auch globale gespeicherte Prozeduren unterstützt, in denen auf mehrere Datenbanken zugegriffen werden kann.

Eine weitere Neuerung in System 10 stellt der *Replication Server* dar, mit dem eine Kopie der Datenbank (bzw. Teilen davon) an einem entfernten Ort geführt werden kann. Die Aktualisierung der Kopie erfolgt asynchron durch Übertragung der Log-Daten. Dies ermöglicht eine schnelle Katastrophen-Recovery, indem beim Ausfall des Primärsystems die Verarbeitung mit der DB-Kopie fortgeführt wird. Im Normalbetrieb können lesende Anfragen, die nicht den absolut neuesten DB-Zustand benötigen, auf der Kopie arbeiten. Somit lassen sich komplexe Anfragen ohne Beeinträchtigung von OLTP-Anwendungen ausführen.

Der *Navigation Server* schließlich erlaubt eine parallele DB-Verarbeitung auf einem eng oder lose gekoppelten Parallelrechner. Dabei wird bei loser Kopplung ein Shared-Nothing-Ansatz verfolgt. Der Navigation Server ist eine dedizierte Komponente, die komplexe Anfragen in Teilanfragen zerlegt, die von mehreren Instanzen des Sybase SQL Servers parallel bearbeitet werden. Neben Anfragen können jedoch auch OLTP-Anwendungen auf derselben Datenbank bearbeitet werden. Die Festlegung der Datenpartitionierung erfolgt durch ein spezielles Tool (Configurator), das u.a. Angaben zum erwarteten Lastprofil als Eingabe verlangt. Es kann jedoch auch die aktuelle DB-Verarbeitung überwachen und Änderungen in der DB-Verteilung empfehlen. Der Navigation Server wurde zusammen mit NCR entwickelt und ist daher zunächst auf NCR-Parallelrechnern lauffähig.

19.7 Tandem

Tandem bietet bereits seit Mitte der siebziger Jahre fehlertolerante Computersysteme und ist Marktführer auf diesem Gebiet. Durch redundante Auslegung aller wesentlichen Hardware- und Software-Komponenten können sämtliche Einfach-

Fehler automatisch behandelt werden, so daß sich keine Beeinträchtigung der Verfügbarkeit ergibt. Die Mehrrechner-Systeme von Tandem sind inkrementell von 2 bis auf über 4000 Prozessoren erweiterbar. Auch eine geographische Verteilung der Systeme wird unterstützt, insbesondere zur Katastrophen-Recovery.

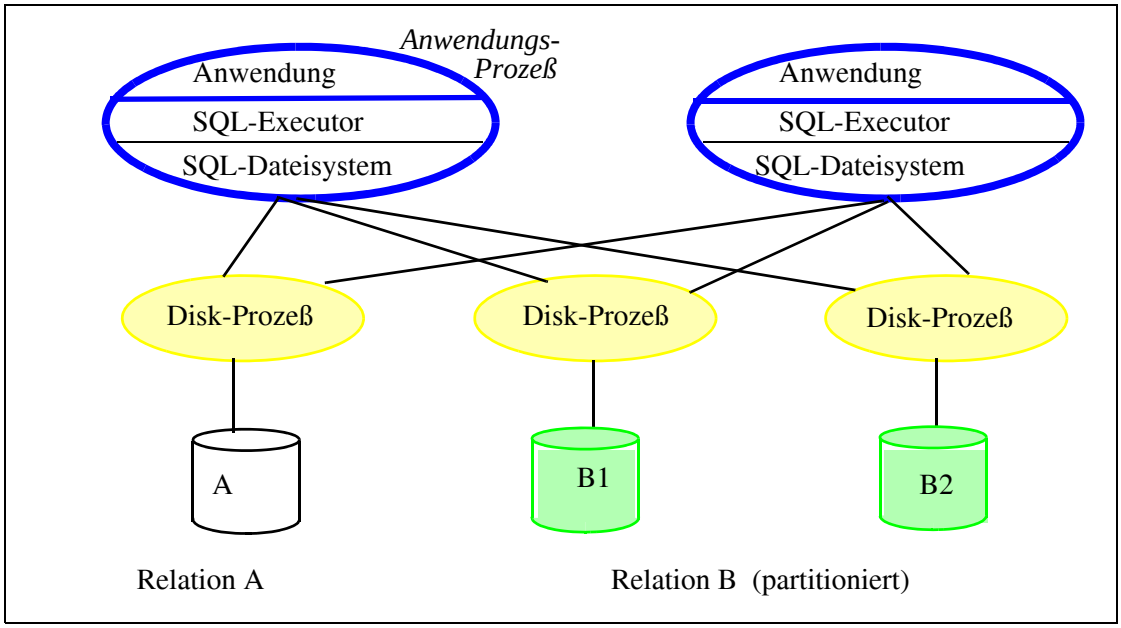
Tandem-Rechner verwenden traditionellerweise das *Guardian*-Betriebssystem, das u.a. sehr effiziente Kommunikationsmechanismen bereitstellt. Daneben wurde eine fehlertolerante Unix-Version realisiert. Beide Betriebssysteme basieren auf einem gemeinsamen Kern, dessen Komponente *TMF* (Transaction Management Facility) Dienste der Transaktionsverwaltung TP-Monitoren und DBS anbietet. Dazu zählt vor allem die Presumed-Abort-Variante eines verteiltes Zwei-Phasen-Commit-Protokolls. Die Auflösung globaler Deadlocks erfolgt über einen Timeout-Ansatz.

Tandem unterstützt bereits seit Anfang der achtziger Jahre verteilte Datenbanken, zunächst für das satzorientierte System *Encompass* [Bo81] und seit 1987 für das relationale *NonStopSQL* [Ta89]. Bei lokaler Verteilung repräsentiert NonStopSQL ein paralleles DBS vom Typ Shared-Nothing. Relationen können horizontal über bis zu 200 Knoten partitioniert werden; für Indexstrukturen (B*-Bäume) erfolgt eine analoge Partitionierung. Zunächst lag der Haupteinsatz von NonStopSQL im Bereich von OLTP-Anwendungen. Durch die in den letzten Versionen erfolgte umfassende Unterstützung von Intra-Query-Parallelität lassen sich jetzt jedoch auch komplexe DB-Anfragen - z.B. zur Entscheidungsunterstützung (decision support) - effizient bearbeiten. Die parallele Join-Verarbeitung kann sowohl über eine dynamische Replikation (Kap. 18.3.1) als auch über eine dynamische Partitionierung (Kap. 18.3.2) der Eingaberelationen erfolgen. Als lokale Join-Methode werden neben Nested-Loop- und Sort-Merge- auch Hash-Joins unterstützt. Hierzu wurde ein speicheradaptives Hash-Join-Verfahren (Kap. 18.3.3) realisiert, das die gleichzeitige Bearbeitung von OLTP-Transaktionen zuläßt [ZG90]. Für Mehr-Wege-Joins werden in NonStopSQL zur Begrenzung des Suchraums ausschließlich rechts-tiefe Operatorbäume (Kap. 18.3.4) berücksichtigt.

Abb. 19-7 zeigt die bei der Ausführung von SQL-Operationen involvierten Systemkomponenten, wenn keine Intra-Query-Parallelität verwendet wird. Dabei werden der SQL-Executor und das SQL-Dateisystem im Prozeß des Anwendungsprogrammes ausgeführt. Daneben besteht für jede DB-Platte ein eigener Disk-Prozeß, der sämtliche Datenzugriffe auf die Platte durchführt sowie für die Puffer- und Sperrverwaltung der betreffenden Daten verantwortlich ist. Der SQL-Executor kontrolliert die Ausführung der vom SQL-Compiler erzeugten Ausführungspläne. Das Dateisystem verwaltet die Dateipartitionen und zerlegt die Operationen einer Relation in Teilanfragen, die auf jeweils einer Partition (Platte) vom zuständigen Disk-Prozeß bearbeitet werden können. Die Disk-Prozesse sind i.a. über verschiedene Rechner verteilt und können relationale Operatoren auf ihren Partitionen auszuführen. Dies ist wesentlich zur Reduzierung des Kommunikationsaufwan-

des, da durch Ausführung von Selektionen und Projektionen in den Disk-Prozessen nur die relevanten Daten an den Executor zu übertragen sind. Die parallele Query-Verarbeitung erfolgt in mehreren, parallel laufenden Hilfsprozessen (Executor Server Processes), die vom Executor koordiniert werden und ihrerseits mit den Disk-Prozessen unterschiedlicher Partitionen zusammenarbeiten [Ze90, Le91].

Abb. 19-7: Laufzeitkomponenten von NonStop SQL



Tandem war der erste Hersteller, der für sein DBS eine Katastrophen-Recovery (Kap. 9.5) unterstützte, und zwar durch die Komponente *RDF* (Remote Data Facility) [Ly90]. Die Behandlung von Platten- und Rechnerausfällen erfolgt über Spiegelplatten, die stets mit zwei Rechnern verbunden sind (Kap. 17.4). Zur Interoperabilität mit unterschiedlichen Client-Systemen bietet Tandem für NonStopSQL ein ODBC-Gateway an. Eine verteilte Transaktionsverarbeitung ist möglich über den TP-Monitor *PATHWAY* von Tandem. Künftig soll auch X/Open DTP (Kap. 11.4.1) unterstützt werden.

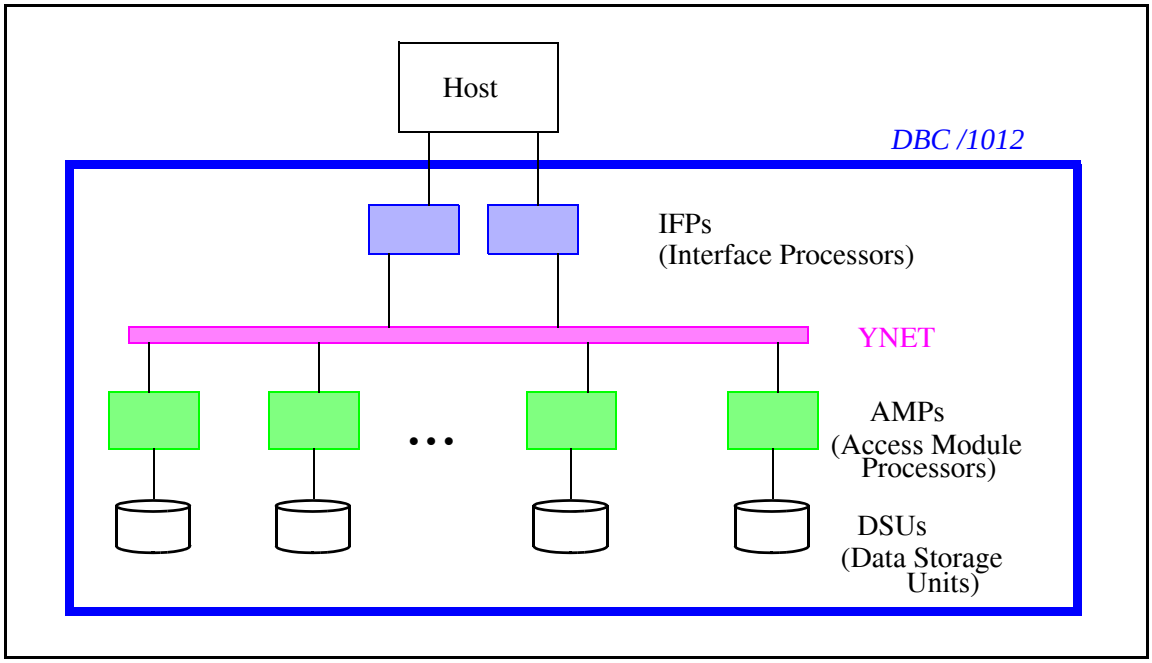
19.8 NCR/Teradata

Teradata gelang es als erstem Unternehmen, mit einer DB-Maschine zur parallelen DB-Verarbeitung kommerziell erfolgreich zu sein. Dies geschah mit dem Back-End-System DBC/1012 [Te83, Ne86], das 1983 auf den Markt kam. 1991 wurde Teradata von NCR übernommen.

Der Grobaufbau von Teradatas DB-Maschine ist in Abb. 19-8 gezeigt. Es handelt sich dabei um ein Shared-Nothing-System mit bis zu 1024 Prozessoren. Die Datenverarbeitung erfolgt durch sogenannte AMP-Rechner (Access Module Processors), die jeweils eine Partition bearbeiten. Der Anschluß der DB-Maschine an Host-

Rechner bzw. Workstations erfolgt durch IFP-Rechner (Interface Processors), über die SQL-Befehle ins Back-End-System gelangen und die Ergebnisse zurückgeliefert werden. Das Herz der Architektur ist jedoch das aktive Kommunikationsnetz YNET. Es besteht aus speziellen Kommunikationsprozessoren, die neben Kommunikationsfunktionen auch Sortier- und Mischvorgänge realisieren. Als Prozessoren kommen in den IFPs und AMPs Standard-Intel-CPUs zum Einsatz. Es bestehen Installationen mit ca. 300 Prozessoren.

Abb. 19-8: Aufbau der Teradata-DB-Maschine DBC /1012



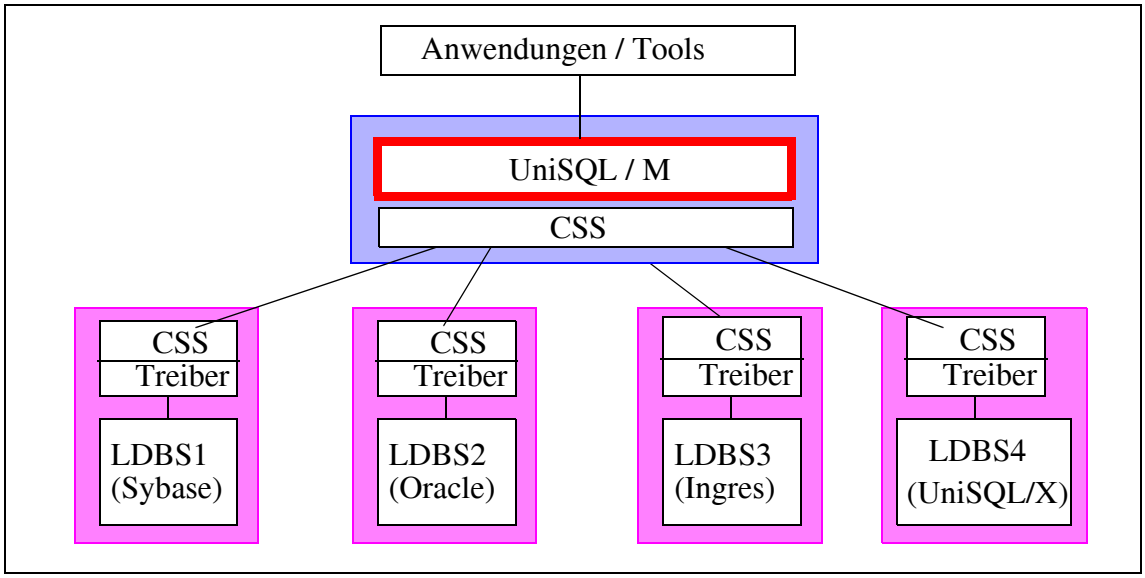
Die Datenverteilung basiert auf einer horizontalen Partitionierung aller Relationen über alle Rechner (!) und ist durch eine Hash-Funktion auf dem Primärschlüssel definiert. Dies macht zwar die Bestimmung der Datenverteilung sehr einfach und gestattet eine breite Parallelisierung. Dafür müssen jedoch alle Anfragen, die sich nicht auf den Primärschlüssel beziehen, auf allen Rechnern bearbeitet werden, was einen hohen Kommunikationsaufwand verursacht. Dies dürfte auch einer der Gründe dafür sein, daß Teradata bislang v.a. zur Parallelisierung datenintensiver Anfragen erfolgreich war, jedoch weniger für OLTP-Anwendungen. Zur schnellen Behandlung von Rechner- und Plattenausfällen unterstützt Teradata eine Datenreplikation in Form des Interleaved Declustering (Kap. 17.4).

Nach der Übernahme durch NCR wurde die Teradata-Software auf den Unix-Parallelrechner NCR 3700 portiert [WCK93]. Dabei können bis zu 4096 Rechner eingesetzt werden, die jeweils aus 4 Intel-Prozessoren bestehen. Die Rechnerkopplung erfolgt über ein mehrstufiges Banyan-Netzwerk namens BYNET, das wesentlich leistungsfähiger und skalierbarer als das ursprüngliche YNET ist.

19.9 UniSQL

Die Fa. UniSQL Inc. wurde 1990 von W. Kim gegründet und bietet ein gleichnamiges objekt-orientiertes DBS an. Als Anfragesprache dient eine objekt-orientierte Erweiterung von SQL, so daß eine Kompatibilität mit relationalen DBS erreicht wird. Durch das Produkt *UniSQL/M* [Ki93], welches ein eng gekoppeltes föderatives Mehrrechner-DBS (Kap. 12) repräsentiert, erfolgt die Unterstützung heterogener DBS. Wie Abb. 19-9 zeigt, ist UniSQL/M eine zentralisierte Komponente, an die sämtliche globalen DB-Operationen zu richten sind. UniSQL/M transformiert die Anfragen in mehrere lokal ausführbare Teiloperationen, die den lokalen DBS (LDBS) zur Bearbeitung übergeben werden. Zur Anpassung der Teiloperationen an die lokalen DB-Schnittstellen wird ein Treiber am Ort der LDBS benötigt. Zur Zeit existieren solche Treiber für Ingres, Oracle und Sybase; daneben wird ein Tool-Kit zur Realisierung eigener Treiber angeboten. Weiterhin ist der Zugriff auf das objekt-orientierte DBS UniSQL/X möglich. Zur Kommunikation ist an jedem Rechner die CSS-Komponente (communication subsystem) erforderlich, welche z.Zt. auf TCP/IP beschränkt ist.

Abb. 19-9: Einsatzumgebung von UniSQL/M



Das Besondere an UniSQL/M ist, daß es eine Schemaintegration (Kap. 12.3) zur Überbrückung von semantischen Unterschieden in den lokalen Datenbanken unterstützt. Die Erstellung des globalen Schemas sowie seine Wartung muß zwar manuell erfolgen, jedoch wird durch ein Tool eine weitreichende Unterstützung zur Angleichung der wichtigsten Problemfälle geboten (Umbenennungen, Formattransformationen, etc.). Da UniSQL/M ein objekt-orientiertes Datenmodell verwendet, können auch strukturelle Unterschiede vielfach behoben werden.
UniSQL/M erlaubt verteilte Änderungstransaktionen, wobei jedoch unterstellt wird, daß jedes LDBS ein striktes Zwei-Phasen-Sperrprotokoll sowie ein Zwei-

Phasen-Commit befolgt. In diesem Fall ist die globale Transaktionsverwaltung bekanntlich sehr einfach (Kap. 11.3.1, Kap. 12.5).

19.10 Computer Associates

Computer Associates (CA) ist eines der weltweit größten Software-Unternehmen und vertreibt eine reichhaltige Palette von über 200 Produkten. Datenbanksysteme bietet es für zahlreiche Plattformen an, insbesondere die Systeme *CA-DATACOM* und *CA-IDMS*. Eine verteilte DB-Verarbeitung wird vor allem für das relationale CA-DATACOM unterstützt, welches schwerpunktmäßig auf IBM-Großrechnern mit den Betriebssystemen MVS und VSE im Einsatz ist (daneben existieren jedoch auch Versionen für VM, PC-DOS und Unix). Die Verwaltung verteilter Datenbanken ist Aufgabe der Komponente *CA-DB:STAR* [CA91], die an jedem der beteiligten Knoten vorliegt. Relationen können horizontal über eine Bereichsfragmentierung partitioniert werden, wobei volle Fragmentierungstransparenz geboten wird. Ein verteiltes Zwei-Phasen-Commit-Protokoll erlaubt die Ausführung globaler Änderungstransaktionen.

Ferner werden replizierte Datenbanken in ihrer allgemeinen Form unterstützt, wobei die Aktualisierung der Kopien synchron gemäß einem Write-All-Ansatz (Kap. 9.1) erfolgt. Von Nachteil dabei ist, daß eine Änderungstransaktion abgebrochen wird, sobald eine der Kopien nicht aktualisiert werden kann. Zudem werden danach keine Änderungen mehr zugelassen bis wieder alle Kopien verfügbar sind. Bei längeren Ausfällen ist es möglich, einzelne Kopien manuell aus dem System herauszunehmen, um den Änderungsbetrieb nicht weiter zu blockieren. Bei Hinzunahme eines reparierten Knotens ist dann zunächst eine Aktualisierung der Kopien zu veranlassen.

CA-DB:STAR realisiert Verteilte DBS in ihrer ursprünglichen Zielstellung und unterstellt daher auch eine weitgehend homogene Umgebung. Zur Integration von Dateien bzw. Datenbanken anderer Systeme müssen diese in CA-DATACOM-Datenbanken konvertiert werden. Bestehende Anwendungen können dabei weiterhin genutzt werden, da ihre Datenzugriffe über Konverter-Funktionen (Transparency Migrations-Software) auf DB-Anweisungen für CA-DATACOM abgebildet werden.

19.11 Cincom

Die Fa. Cincom vertreibt ein verteiltes SQL-DBS namens *Supra*, das ursprünglich vor allem im Mainframe-Bereich eingesetzt wurde. Mittlerweile werden jedoch zahlreiche Plattformen (Unix, VM, OS/2, etc.) unterstützt, so daß DB-Anwendungen ohne Änderungen auf unterschiedliche Umgebungen übertragbar sind. Bei der verteilten DB-Verarbeitung läuft an jedem der Knoten eine Instanz des *Supra Distributed Server* [Su92]. Verteilte Anfragen und verteilte Sichten werden unterstützt, Fragmentierung und Replikation von Relationen jedoch nicht. Katalogda-

ten werden an jedem Knoten geführt. Dabei sind benutzerspezifische Angaben an allen Knoten repliziert, während relationenspezifische Metadaten wahlweise partitioniert oder repliziert gehalten werden. Zur Wartung replizierter Katalogdaten wird ein Write-All-Ansatz (Kap. 9.1) verfolgt, bei dem jeder Knoten verfügbar sein muß.

Die realisierte Variante des verteilten Zwei-Phasen-Commit-Protokolls ("Transaction Partnering" genannt) reduziert die Blockierungswahrscheinlichkeit durch einen Koordinatorausfall, indem jedem beteiligten Rechner in der ersten Commit-Phase mitgeteilt wird, welche Rechner ansonsten noch am Commit teilnehmen. Damit kann nach einem Koordinatorausfall überprüft werden, ob einem dieser Rechner das Commit-Ergebnis noch mitgeteilt wurde (Kap. 7.2.1). Ist auch dies nicht der Fall, kann der Systemverwalter zur Aufhebung der Blockierung für die betroffene Transaktion auf Abort oder Commit entscheiden (!), womit natürlich inkonsistente DB-Zustände möglich sind.

Zur Interoperabilität ist geplant, IBM DRDA sowie ODBC zu unterstützen.

19.12 SNI

SNI (Siemens Nixdorf Informationssysteme) entwickelt und vertreibt für seine BS2000-Umgebungen das relationale DBS *Sesam* sowie das CODASYL-DBS *UDS*. Für beide DBS besteht der Zugang über eine einheitliche SQL-Schnittstelle. Die Komponenten *Sesam-DCN* bzw. *UDS-D* [Gra88] unterstützen die verteilte DB-Verarbeitung in einem homogenen BS2000-Umfeld. In beiden Fällen ist es möglich, in einer Transaktion auf mehrere, unabhängige Sesam- bzw. UDS-Datenbanken zuzugreifen, wobei jedoch jede Operation auf einen Knoten (eine DB) beschränkt ist. Ein verteiltes Zwei-Phasen-Commit-Protokoll gestattet die Ausführung verteilter Änderungstransaktionen. Globale Deadlocks werden explizit erkannt und aufgelöst.

Eine verteilte Transaktionsverarbeitung mit programmierter Verteilung ist für beide DBS über den TP-Monitor *UTM* von SNI möglich [Go90]. Dieser TP-Monitor ist neben BS2000 auch auf verschiedenen Unix-Plattformen lauffähig und kann mit CICS zusammenarbeiten. Zudem unterstützt er die XA-Schnittstelle, so daß eine Vielzahl von DBS in die verteilte Transaktionsverarbeitung eingebunden werden kann.

19.13 Abschließende Bemerkungen

Die (unvollständige) Übersicht existierender Produkte zeigt, daß es zur parallelen und verteilten DB-Verarbeitung bereits eine ganze Reihe leistungsfähiger Implementierungen gibt. Zusammenfassend lassen sich dabei folgende Beobachtungen treffen:

- Parallele DBS sind auf dem Vormarsch. Implementierungen bestehen für beide Hauptansätze, nämlich Shared-Disk (IBM Parallel Sysplex, Parallel Transaction Server und Parallel Query Server, Oracle Parallel Server, DEC, Ingres) sowie Shared-Nothing (Teradata, Tandem NonStop SQL, Sybase Navigation Server, DB2/6000). Daneben werden auch Multiprozessoren von vielen DBS zur parallelen DB-Verarbeitung genutzt.
- Eine Client/Server-Verarbeitung, bei der DB-Anwendungen und Tools von Client-Rechnern (PCs, Workstations, etc.) auf Server-Datenbanken zugreifen, erlauben fast alle Systeme. Für den Zugriff auf mehrere, heterogene Server-Datenbanken in einer Transaktion bzw. in einer DB-Operation werden die drei wichtigsten Alternativen unterstützt: programmierte Verteilung, Verteilung von DB-Operationen sowie föderative DBS.
- Verteilte DBS im eigentlichen Sinn, bei denen eine logische Datenbank unter mehreren, geographisch verteilten Knoten aufgeteilt wird, unterstützen relativ wenige Hersteller (Computer Associates, Cincom). Diese Systeme verlangen i.a. eine homogene Umgebung (identische DBVS-Instanzen) und beschränken die Autonomie der beteiligten Knoten.
- Die Ansätze zur verteilten DB-Verarbeitung in Ingres/Star, Oracle, Informix-Star, Sybase OmniSQL etc. entsprechen dagegen lose gekoppelten föderativen DBS. Es liegen mehrere, unabhängige Datenbanken vor, auf denen mit einer einheitlichen Anfragesprache (SQL) verteilte Operationen formuliert werden. Ortstransparenz wird i.a. über die Verwendung von Synonymen erreicht. Das eng gekoppelte föderative DBS UniSQL/M bietet eine weitergehende Verteilungstransparenz, indem eine Schemaintegration der lokalen Datenbanken vorgenommen wird.
- Eine für den Benutzer transparente Fragmentierung von Relationen wird außer in parallelen Shared-Nothing-DBS meist nicht unterstützt, ebensowenig replizierte Datenbanken in ihrer allgemeinen Form (Änderungsmöglichkeit aller Kopien, vollständige Aktualität, volle Replikationstransparenz). Eine Ausnahme stellt nur CA-DB:STAR von Computer Associates dar. Neben dem hohen Realisierungsaufwand dürfte der Grund dafür sein, daß die Erfüllung dieser Forderungen zwangsweise eine enge Zusammenarbeit der beteiligten Rechner und damit eine Reduzierung der Knotenautonomie erfordert.
- Weite Verbreitung finden jedoch die einfacheren Replikationsansätze der Schnappschuß-Replikation bzw. zur Katastrophen-Recovery. Ein (transparentes) verteiltes 2PC wird mittlerweile von den meisten Systemen unterstützt, zum Teil bereits im Betriebssystem (VMS, Guardian).
- Die Zusammenarbeit zwischen Produkten verschiedener Hersteller erfolgt meist über Gateways, deren Funktionalität jedoch oft eingeschränkt ist (vielfach nur lesende Zugriffe bzw. keine verteilten Transaktionen). Standardisierungen wie DRDA und ODBC gewinnen jedoch zunehmend an Bedeutung.
- Eine verteilte Transaktionsverarbeitung mit programmierter Verteilung wird von zahlreichen TP-Monitoren unterstützt. Damit wird der Zugriff auf mehrere, heterogene DBS unter Wahrung einer hohen Knotenautonomie erreicht. Neben LU6.2 setzt sich zunehmend die verteilte Transaktionsverarbeitung nach X/Open durch, da immer mehr TP-Monitore und DBS die XA-Schnittstelle unterstützen.

Teil VII
Anhang

Lösungen zu den Übungsaufgaben

Aufgabe 3-1: Shared-Everything

Datenbank- und Log-E/A während eines kritischen Abschnittes muß vermieden werden; Vermeidung von Paging-E/A sowie Zeitscheibenentzug im kritischen Abschnitt würde jedoch Kooperation mit dem Betriebssystem erfordern. Generell kann Konfliktgefahr durch Verwendung kleiner Sperrgranulate (vieler Semaphore) reduziert werden; z.B. nicht nur 1 Semaphor für die gesamte Sperrtabelle, sondern ein Semaphor pro Hash-Klasse.

Aufgabe 3-2: Shared-Nothing vs. Shared-Disk

Leistungsfähigkeit: größere Möglichkeiten zur Lastbalancierung für Shared-Disk, da jede DB-Operation von jedem DBVS abarbeitbar ist; mehr Einsatzmöglichkeiten einer nahen Rechnerkopplung

Erweiterbarkeit: Verknüpfung sehr vieler Rechner mit allen Platten aufwendig für Shared-Disk; erfordert nachrichten-basierte E/A mit sehr leistungsfähigem, skalierbarem Kommunikationsnetzwerk. Bei Shared-Nothing erfordert Hinzunahme von Rechnern aufwendige physische Neuverteilung der Datenbank

Verfügbarkeit: Datenbank nach Rechnerausfall bei Shared-Disk weiterhin erreichbar, bei Shared-Nothing nur, falls Replikation vorliegt bzw. Daten noch umverteilt werden können. Komplexe Recovery-Protokolle für Shared-Disk (globaler Log); aufwendiges Commit-Protokoll für Shared-Nothing

Administration: Festlegung und Anpassung der physischen Datenbankverteilung bei Shared-Nothing i.a. vom Systemverwalter zu veranlassen

Aufgabe 4-1: Katalogverwaltung

Neben partitionierten Katalogen eignet sich vor allem eine vollständige Replikation, da hiermit lesende Katalogzugriffe stets lokal abgewickelt werden können. Selbst die Replizierung von Angaben des internen Schemas ist zu empfehlen, um an jedem Knoten eine schnelle Übersetzung und Optimierung von Anfragen zu erreichen. Da Änderungen von Katalogdaten eher selten sind, ist der Aufwand zur Wartung der Replikation vertretbar.

Aufgabe 4-2: Namensauflösung in R*

a) Erst die Kombination von <Benutzername> und <Benutzerknoten> ergibt globale Eindeutigkeit. Somit können auch Benutzernamen lokal vergeben werden. Ohne <Benutzerknoten> wäre es weiterhin nicht möglich, daß zwei Benutzer mit demselben lokalen Namen (<Benutzername>) an einem bestimmten Knoten verschiedene Objekte mit gleichem <Objektname> erzeugen.

b)

	Aufruf durch Benutzer	
	RAHM	sonstige Benutzer
in L	BEISPIEL@F	RAHM.BEISPIEL@F
in B	@L.BEISPIEL@F	RAHM@L.BEISPIEL@F
in F	@L.BEISPIEL	RAHM@L.BEISPIEL

Aufgabe 4-3: Namensverwaltung (SQL92)

Globale Eindeutigkeit ließe sich erreichen, wenn Katalognamen global eindeutig sind. Allerdings ist auch dann eine lokale Überprüfung von Objektnamen nur möglich, wenn alle Datenbanken eines (SQL92-)Katalogs auf einen Rechner begrenzt werden (d.h., Kataloge als Verteilungsgranulate dienen). Dies ist jedoch viel zu restriktiv, da selbst einzelne Datenbanken und Relationen über mehrere Knoten zu verteilen sind.
Durch Erweiterung von Objektnamen um den Knotennamen kann in ähnlicher Weise wie in Kap. 4.4 diskutiert globale Eindeutigkeit erreicht werden.

Aufgabe 4-4: Herausnahme von Knoten aus dem System

Sämtliche Daten, die an dem betroffenen Knoten erzeugt wurden, sind davon betroffen. Es genügt prinzipiell, die Katalogdaten an einen anderen (oder mehrere) Knoten zu übertragen und in den Synonymtabellen die Knotennamen anzupassen. Sämtliche Daten und Anwendungen des abzuschaltenden Knotens sind natürlich auch auf andere Rechner zu migrieren.

Aufgabe 4-5: Synonyme

Im Extremfall sind für alle (an mehreren Knoten benötigten) DB-Objekte Synonyme zu definieren. Weiterhin muß die Replikation von Synonymangaben gewartet werden. Generell ergibt sich eine Verkomplizierung der Systemadministration.

Aufgabe 4-6: Nutzung standardisierter Directory-Dienste

Zur Nutzung von Directory-Diensten wie X.500 wären globale Namen auf die X.500-Konventionen abzubilden. Allerdings würde damit nur ein Bruchteil der zur Katalogverwaltung notwendigen Funktionalität abgedeckt. Insbesondere ist es notwendig, die Katalogdaten transaktionsbasiert zu verarbeiten und auf ihnen selbst DB-Anfragen zu verarbeiten. Weiterhin ist keine ausreichende Flexibilität gegeben zur Spezifizierung komplexer Verteilungskriterien, wie im Falle fragmentierter Relationen benötigt.

Aufgabe 5-1: Korrektheit der abgeleiteten horizontalen Fragmentierung

S-Tupel mit Nullwerten als Fremdschlüssel finden keinen Verbundpartner in R. Um die Vollständigkeit der Fragmentierung zu gewährleisten, sind diese S-Tupel einem (oder mehreren) eigenen Fragment(en) zuzuordnen. Für die Join-Berechnung spielen diese S-Tupel keine Rolle, außer wenn ein äußerer Verbund (outer join) zu berechnen ist, bei dem sämtliche Tupel der beteiligten Relationen im Ergebnis zu berücksichtigen sind.

Aufgabe 5-2: Abgeleitete horizontale Fragmentierung

Horizontale Fragmentierung von ABT:

$$ABT_1 = \sigma_{AORT = "KL"}(ABT)$$
$$ABT_2 = \sigma_{AORT = "F"}(ABT)$$
$$ABT_3 = \sigma_{AORT = "L"}(ABT).$$

Abgeleitete horizontale Fragmentierung von PERSONAL (für den Fremdschlüssel ANR seien keine Nullwerte zulässig):

$$PERSONAL_1 = PERSONAL \ltimes ABT_1$$
$$PERSONAL_2 = PERSONAL \ltimes ABT_2$$
$$PERSONAL_3 = PERSONAL \ltimes ABT_3.$$

Abgeleitete horizontale Fragmentierung von PMITARBEIT:

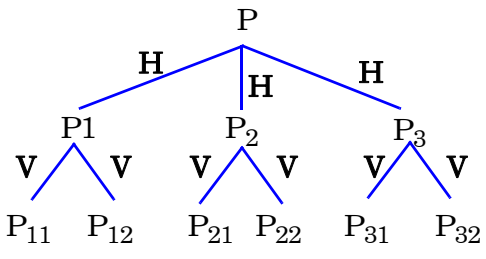
$$PMITARBEIT_1 = PMITARBEIT \ltimes PERSONAL_1$$
$$PMITARBEIT_2 = PMITARBEIT \ltimes PERSONAL_2$$
$$PMITARBEIT_3 = PMITARBEIT \ltimes PERSONAL_3.$$

Im Falle einer horizontalen Fragmentierung von PROJEKT könnte für PMITARBEIT eine abgeleitete horizontale Fragmentierung auch bezüglich PROJEKT erfolgen. Dies wäre vorteilhafter als die auf PERSONAL ausgerichtete Fragmentierung, wenn häufiger eine Join-Berechnung mit PROJEKT als mit PERSONAL vorzunehmen ist.

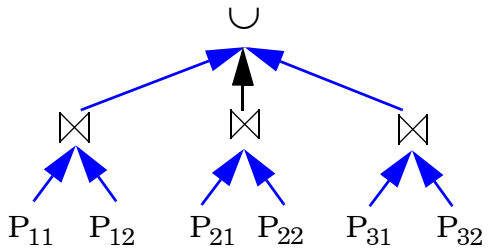
Aufgabe 5-3: Hybride Fragmentierung

Vertikale Fragmentierung von $PERSONAL_1$ ($PERSONAL_2$ und $PERSONAL_3$ analog):

$$PERSONAL_{11} = \pi_{PNR, PNAME, ANR, BERUF}(PERSONAL_1)$$
$$PERSONAL_{12} = \pi_{PNR, GEHALT}(PERSONAL_1)$$



Fragmentierungsbaum



Operatorbaum zur Rekonstruktion

Aufgabe 5-4: Bestimmung der Datenallokation

Allokation: F1 -> R2, F4 -> R3, F3 -> R1, F2 -> R1 (da R2 und R3 ansonsten überlastet)

CU_1=21,925 MIPS (73 %); CU_2=22,85 MIPS (76 %); CU_3=19,925 MIPS (66 %)

Anteil lokaler Zugriffe: 53 %; Kommunikations-Overhead: 12,3 MIPS

Aufgabe 6-1: Normalisierung

Konjunktive Normalform:

(PNAME LIKE "M%" ∨ PNR > 550 ∨ BERUF="Programmierer") ∧
(PNAME LIKE "M%" ∨ PNR > 550 ∨ GEHALT < 80000) ∧
(PNAME LIKE "M%" ∨ BERUF="Programmierer" ∨ GEHALT < 80000) ∧
(PNAME LIKE "M%" ∨ GEHALT < 80000) ∧
(BERUF = "Techniker" ∨ PNR > 550 ∨ BERUF="Programmierer") ∧
(BERUF = "Techniker" ∨ PNR > 550 ∨ GEHALT < 80000) ∧
(BERUF = "Techniker" ∨ BERUF="Programmierer" ∨ GEHALT < 80000) ∧
(BERUF = "Techniker" ∨ GEHALT < 80000).

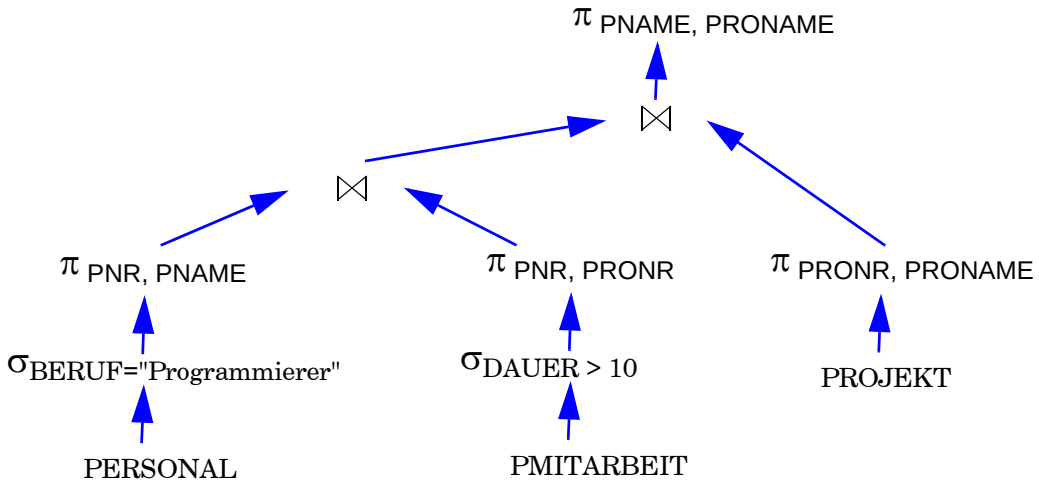
Disjunktive Normalform:

(PNAME LIKE "M%" ∧ BERUF = "Techniker") ∨
(PNR > 550 ∧ GEHALT < 80000) ∨
(BERUF="Programmierer" ∧ GEHALT < 80000).

Aufgabe 6-2: Vereinfachung

Es ergibt sich die leere Ergebnismenge wegen den gegensätzlichen Prädikaten bzgl. GEHALT

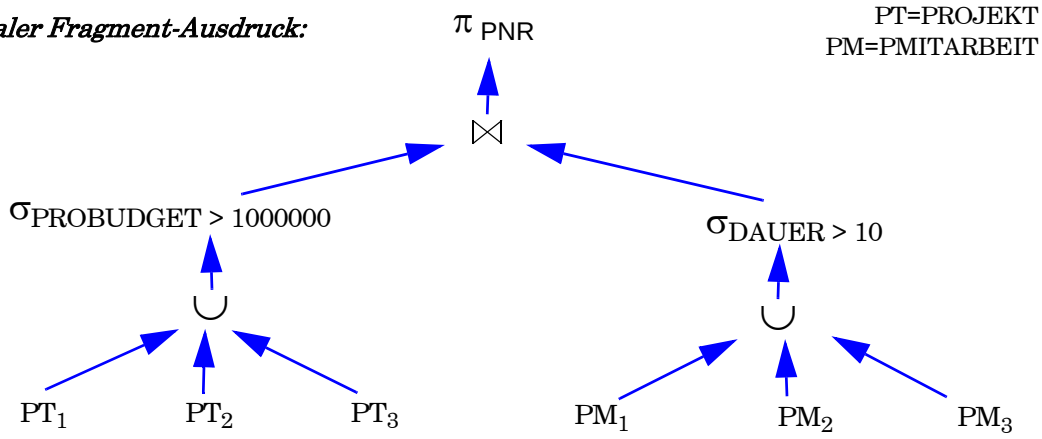
Aufgabe 6-3: Operatorbaum und Rekonstruktion



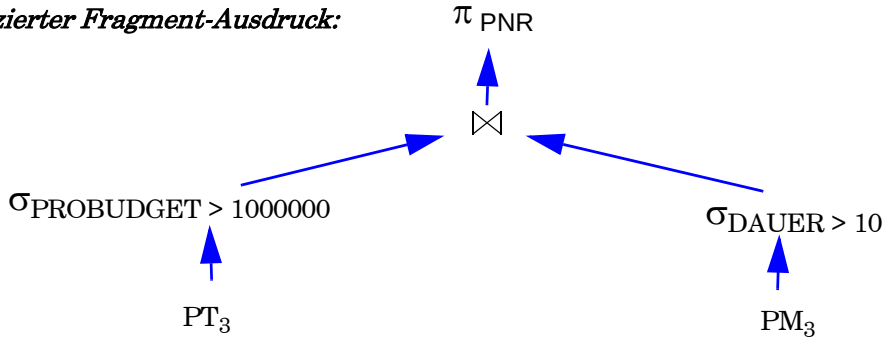
Neben Selektionen wurden auch Projektionen vorgezogen. Es ist dabei darauf zu achten, daß die für die Join-Bearbeitung sowie für die Ausgabe benötigten Attribute erhalten bleiben. Die Reihenfolge der Join-Operationen wurde nicht optimiert, da über die Kardinalitäten keine Aussagen gemacht wurden.

Aufgabe 6-4: Daten-Lokalisierung (abgeleitete Fragmentierung)

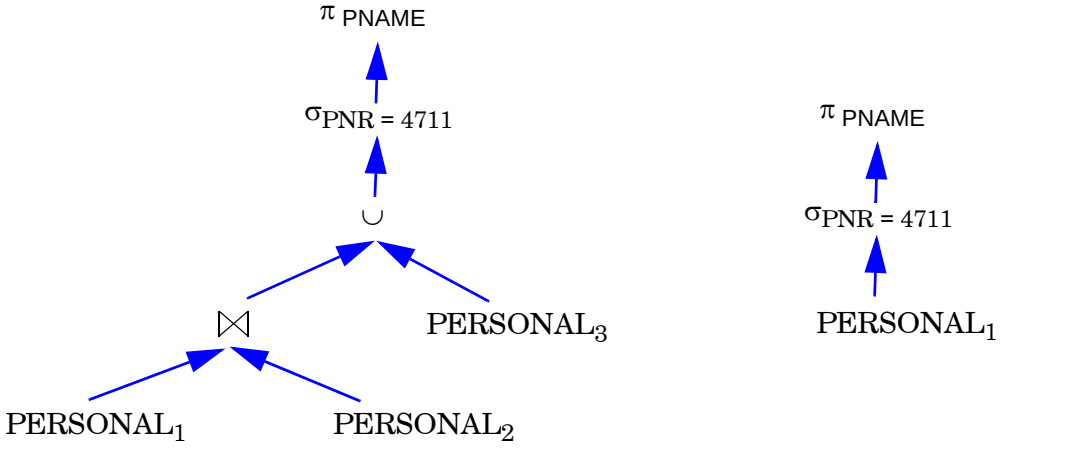
a) Initialer Fragment-Ausdruck:



b) Reduzierter Fragment-Ausdruck:



Aufgabe 6-5: Daten-Lokalisierung (hybride Fragmentierung)



a) Initialer Fragment-Ausdruck *b) Reduzierter Fragment-Ausdruck*

Aufgabe 6-6: Einfache Join-Strategien

S1 (Ship Whole, Join-Berechnung am R-Knoten):
#Nachrichten = 2; #AW = 1000*5 = 5 000.

S2 (Ship Whole, Join-Berechnung am S-Knoten):
#Nachrichten = 2; #AW = 10 000 * 5 =50 000.

F1 (Fetch As Needed; Join-Berechnung am R-Knoten):
#Nachrichten = 10 000*2=20 000; #AW = 10 000 * 0.001 * 1000 * 5 = 50 000.

F2 (Fetch As Needed; Join-Berechnung am S-Knoten):
#Nachrichten = 1000*2= 2 000; #AW = 1000 * 0.001 * 10000 * 5=50 000.

=> S1 ist optimal.

Aufgabe 6-7: Ship-Whole vs. Semi-Join vs. Bitvektor-Join

Ship-Whole; Join-Berechnung an Knoten $K_{PMITARBEIT}$

- K -> $K_{PMITARBEIT}$: Nachricht zum Query-Start
- $K_{PMITARBEIT}$ -> $K_{PERSONAL}$: Anfordern der PERSONAL-Daten
- $K_{PERSONAL}$ -> $K_{PMITARBEIT}$: 1000*0.2*3 = 600 AW
- Join-Berechnung an $K_{PMITARBEIT}$.
- $K_{PMITARBEIT}$ -> K: Join-Ergebnis (200*0.75*2*5=1500 AW)

=> #Nachrichten=4; #AW=2100.

Ship-Whole; Join-Berechnung an Knoten K

- K -> $K_{PMITARBEIT}$: Anfordern der PMITARBEIT-Daten
- K -> $K_{PERSONAL}$: Anfordern der PERSONAL-Daten
- $K_{PMITARBEIT}$ -> K: 1500*3=4500 AW
- $K_{PERSONAL}$ -> K : 1000*0.2*3 = 600 AW
- Join-Berechnung an K

=> #Nachrichten=4; #AW=5100.

Semi-Join; Join-Berechnung an Knoten $K_{PERSONAL}$

- K -> $K_{PERSONAL}$: Nachricht zum Query-Start
- $K_{PERSONAL}$ -> $K_{PMITARBEIT}$: 1000*0.2 = 200 AW Verbundattributwerte
- $K_{PMITARBEIT}$ -> $K_{PERSONAL}$: 200*0.75*2*3=900 AW (Verbundpartner)
- Join-Berechnung an $K_{PERSONAL}$.
- $K_{PERSONAL}$ -> K: Join-Ergebnis (200*0.75*2*5=1500 AW)

=> #Nachrichten=4; #AW=2600.

Semi-Join; Join-Berechnung an Knoten K

- K -> $K_{PMITARBEIT}$: Anfordern der reduzierten Daten
- K -> $K_{PERSONAL}$: Anfordern der reduzierten Daten
- $K_{PMITARBEIT}$ -> $K_{PERSONAL}$: 750 AW Verbundattributwerte
- $K_{PERSONAL}$ -> K : 750*0.2*3=450 AW (Verbundpartner)
- $K_{PERSONAL}$ -> $K_{PMITARBEIT}$: 1000*0.2 = 200 AW Verbundattributwerte
- $K_{PMITARBEIT}$ -> K: 200*0.75*2*3=900 AW (Verbundpartner)
- Join-Berechnung an K

=> #Nachrichten=6; #AW=2300.

Bitvektor-Join; Join-Berechnung an Knoten K

wie vor, jedoch wird in Schritten 3 und 5 jeweils der Bitvektor übertragen (-> 10 AW); in Schritt 4 (6) erhöht sich der Übetragungsumfang auf ca. 475 (945) AW.

=> #Nachrichten=6; #AW=1430.

Aufgabe 6-8: Semi-Join-Berechnung an drittem Knoten K

a) 1. In K_R: Bestimmung der Verbundattributwerte $\pi_V(R)$ und Übertragung an K_S

2. In K_S: Bestimmung von Z1 = (S $\ltimes$ $\pi_V(R)$) = (S $\ltimes$ R) und Übertragung an K

3. In K_S: Bestimmung der Verbundattributwerte $\pi_V(Z1)$ und Übertragung an K_R

4. In K_R: Bestimmung von Z2 = (R $\ltimes$ $\pi_V(Z1)$) = (R $\ltimes$ S) und Übertragung an K

5. In K: Bestimmung von Z1 $\bowtie$ Z2 = R $\bowtie$ S.

b) #AW=17 (statt 19), da zwischen K_S und K_R in Schritt 3 nur noch zwei Attributwerte (1, 5) statt vier übertragen werden.

Aufgabe 6-9: Mehr-Wege-Join

Zu berechnen ist folgende gekettete Join-Anfrage

$$\sigma_{BERUF = "Programmierer"}(PERSONAL) \bowtie PMITARBEIT \bowtie PROJEKT.$$

Wir bertrachten zwei Ship-Whole-Strategien sowie die Semi-Join-Strategie mit vollständiger Reduzierung:

S1: Join-Berechnung an $K_{PERSONAL}$; Anfordern und Transfer von PMITARBEIT (1500*3=4500AW) und PROJEKT (200*3=600 AW).
=> #Nachrichten=4; #AW=5100.

S2: Start der Join-Berechnung an $K_{PMITARBEIT}$; Anfordern und Transfer von PROJEKT (200*3=600 AW) und $\sigma_{BERUF = "Programmierer"}$ (PERSONAL) (100*4=400 AW); Join-Ergebnis (100*0.75*2*10=1500 AW) an $K_{PERSONAL}$
=> #Nachrichten=6; #AW=2500.

SJ: $K_{PERSONAL}$ -> $K_{PMITARBEIT}$: 1000*0.1 = 100 AW Verbundattributwerte.
$K_{PMITARBEIT}$ -> $K_{PROJEKT}$: 100*0.75*2=150 AW Verbundattributwerte
$K_{PROJEKT}$ -> $K_{PMITARBEIT}$: 150 AW Verbundattributwerte
$K_{PMITARBEIT}$ -> $K_{PERSONAL}$: 75 AW Verbundattributwerte
$K_{PROJEKT}$ -> $K_{PERSONAL}$: Transfer der reduzierten Relation (150*3=450 AW)
$K_{PMITARBEIT}$ -> $K_{PERSONAL}$: Transfer der reduzierten Relation (150*3=450 AW)
Join-Berechnung an $K_{PERSONAL}$
=> #Nachrichten=6; #AW=1375.

Aufgabe 7-1: Presumed-Abort- vs. Standard-2PC

	a)	b)	c)
Presumed-Abort	20 N + 13 L	16 N + 11 L	14 N + 9 L
Standard-2PC	20 N + 12 L	16 N + 10 L	18 N + 11 L

N = Nachrichten, L = Log-Vorgänge

Presumed-Abort schneidet im Aufwand für gescheiterte Transaktionen (Fall c) besser ab, da die ACK-Nachrichten und Ende-Sätze eingespart werden. Im Erfolgsfall verursacht der Zwischenknoten in der Hierarchie für Presumed-Abort einen zusätzlichen Log-Vorgang für den Ende-Satz.

Aufgabe 7-2: Nicht-blockierendes 2PC durch Prozeß-Paare

Für ein zentralisiertes 2PC sind drei Nachrichten vom Koordinator zum Backup-TM zu schicken: vor Absenden der PREPARE-Nachrichten, vor Schreiben des Commit-Ergebnisses und vor Schreiben des Ende-Satzes. Der Erhalt dieser Nachrichten ist vom Backup-TM jeweils zu quittieren, so daß insgesamt 6 zusätzliche Nachrichten anfallen. Dies verursacht für N<3 einen höheren Aufwand als 3PC, für N>3 ist der Aufwand geringer. Da der Koordinator jeweils synchron auf die Quittung warten muß, ergibt sich jedoch eine signifikante Antwortzeiterhöhung, vor allem wenn die Kommunikation mit dem Backup über ein Weitverkehrsnetz erfolgen muß.

Aufgabe 8-1: Verbesserung von BOCC

Es wird ein Transaktionszähler TNC geführt, dessen Wert Änderungstransaktionen als Commit-Zeitstempel zugewiesen und anschließend inkrementiert wird. Objekte besitzen einen Schreibzeitstempel WTS, der dem Commit-Zeitstempel der ändernden Transaktion entspricht. Für jedes gelesene Objekt x von Transaktion T_j wird der Schreibzeitstempel der gelesenen Version $ts_j(x)$ im Read-Set vermerkt. Bei der Validierung ist lediglich zu prüfen, ob die gesehenen Objektversionen noch aktuell sind.
Insgesamt ergibt sich folgende Commit-Behandlung für Transaktion T_j:

```
VALID := true;
    for (alle x in RS (Tj)) do
        if  tsj (x) < WTS (x)  then VALID := false;
    end;
if  not VALID then Rollback (Tj);
else if  WS (Tj)  ≠  ∅  then do { Änderungstransaktion }
    cts (Tj) := TNC;     { Commit-Zeitstempel }
    TNC := TNC + 1;
    for (alle x  in WS (Tj)) do
        WTS (x)  := cts (Tj);
    end;
    Schreibphase für Tj ;
end;
```

Für den Zugriff auf die aktuellen WTS-Werte sind nicht die Objekte selbst zu referenzieren, vielmehr können diese Werte in einer Tabelle vermerkt werden. Darin sind die WTS-Werte nur für diejenigen Änderungstransaktionen zu führen, die seit dem Start der ältesten (laufenden) Transaktion validiert haben.

Aufgabe 8-2: FOCC

Für jede Transaktion wird die Anzahl bereits erfolgter Rücksetzungen mitgeführt. Im Konfliktfall wird stets die Transaktion mit der geringeren Anzahl von Rücksetzungen abgebrochen. Damit ist der Transaktion mit der höchsten Anzahl von Rücksetzungen das Durchkommen sicher. Das Verhungern einer bereits (mehrfach) zurückgesetzten Transaktion wird verhindert, da sie irgendwann die meisten Rücksetzungen von allen unbeendeten Transaktionen aufweist.

Aufgabe 8-3: Schmutzige vs. unsichere Änderungen

Schmutzige Änderungen stammen von laufenden Transaktionen, deren Commit daher noch ungewiß ist. Unsichere Änderungen sind eine Variante schmutziger Änderungen, die im verteilten Fall relevant wird. Sie bezeichnen Änderungen von Transaktionen, die lokal bereits erfolgreich zu Ende gekommen sind, deren globales Commi jedoch noch ungewiß ist.

Bei OCC werden Änderungen innerhalb privater Transaktions-Puffer vorgenommen, so daß schmutzige Änderungen nicht sichtbar sind.

Aufgabe 8-4: Vergleich von Synchronisationsverfahren

a) Rücksetzung von T1; T2 < T3

b) Rücksetzung von T2; T3 < T1

c) Rücksetzung von T1 oder T2; T3 < T2 oder T3 < T1

d) Wait (T1), Die (T2); T3 < T1

e) Wound (T3), Wait (T2); T1 < T2

f) Wait (T1), Wait (T2); T3 < T1 < T2

g) keine Blockierung/Rücksetzung; T2 < T3 < T1

Aufgabe 8-5: Deadlock-Erkennung

Lokale Wartegraphen (EXT = EXTERNAL):

R1: EXT -> T2 -> T1 -> EXT
R2: EXT -> T1 -> T3 -> EXT
R3: EXT -> T3 -> T2 -> EXT

R2 sendet keine Nachricht, da *ts (T1) < ts (T3)*

Nachricht von R1 nach R2: EXT -> T2 -> T1 -> EXT
Vervollständigung in R2 zu EXT -> T2 -> T1 -> T3 -> EXT;
keine Folgeaktion, da *ts (T2) < ts (T3)*

Nachricht von R3 nach R1: EXT -> T3 -> T2 -> EXT
Vervollständigung in R1 zu EXT -> T3 -> T2 -> T1 -> EXT

Nachricht von R1 an R2,
Vervollständigung ergibt T3 -> T2 -> T1 -> T3

Erkennung des Deadlocks in R2 (=> Rücksetzung der lokalen Transaktion T2)

Deadlock wurde mit 3 Nachrichten aufgelöst
(2 Nachrichten, falls erste Nachricht von R1 nach R2 nicht gesendet wird)

Aufgabe 9-1: Netzwerk-Partitionierungen

ROWA und Write-All-Available erlauben keine weiteren Änderungen nach einer Netzwerk-Partitionierung, Lesezugriffe sind jedoch auf allen Kopien weiterhin möglich. Beim Primary-Copy-Verfahren kann die Verarbeitung in der Partition mit dem Primärkopien-Rechner fortgeführt werden. Bei Voting-Verfahren können in höchstens einer Partition Änderungen fortgeführt werden, sofern noch ein ausreichendes Schreib-Quorum erreichbar ist. Ein Lesen ist möglich in allen Partitionen, in denen noch ein Lese-Quorum gebildet werden kann (falls noch ein Schreiben möglich ist, kann in keiner anderen Partition ein Lese-Quorum erreicht werden).

Aufgabe 9-2: Datenbankverteilung

Mit dem ROWA-Protokoll kostet ein Lesezugriff auf eine lokale Kopie keine Nachrichten, für einen nicht-lokalen Lesezugriff fallen 4 Nachrichten an (2 für den eigentlichen Zugriff mit Sperrerwerb, 2 zur Sperrfreigabe). Jede Änderung erfordert pro externer Kopie 6 Nachrichten (2 zur Sperranforderung, 4 zur Commit-Behandlung).
Im Beispiel sollte Objekt A an R2 gespeichert werden, da dort die meisten Zugriffe erfolgen. An Rechner R4 ist eine Speicherung von A nicht sinnvoll, da damit nur 80 Nachr./s eingespart würden; die insgesamt 30 Änderungen/s würden jedoch 180 Nachr./s zum

Sperren/Aktualisieren der Kopie erfordern. Diese Änderungskosten entstehen auch bei einer Speicherung von A an Rechner R3 an. Dort betragen jedoch die Einsparungen für Lesezugriffe 200 Nachr./s, so daß A an R3 gespeichert werden sollte. Eine Kopie an Rechner R1 würde 120 Nachr./s aufgrund der 20 externen Änderungen/s verursachen. Die Einsparungen für Lesezugriffe an R1 betragen ebenfalls 120 Nachr./s, so daß eine Speicherung von A an R1 vetretbar ist.

Aufgabe 9-3: Kommunikationsaufwand (ROWA, Primary-Copy)

Nachrichtenbedarf ROWA

Die Lesezugriffe sind alle lokal bis auf Rechner R4. Setzt man pro Lesezugriff 4 Nachrichten an (2 für Objektzugriff, 2 für Sperrfreigabe), so entstehen dort 80 Nachr./s.

Für jede Änderung in R1 bzw. R2 sind die Schreibsperren bei jeweils zwei anderen Rechnern anzufordern (je 2 Nachr.), die auch ins Commit-Protokoll einzubeziehen sind (je 4 Nachr.). Dies erfordert insgesamt (10 + 20) * (2*2 + 2*4) = 360 Nachr./s.

Der Gesamtaufwand für ROWA beträgt somit 440 Nachr./s.

Primary-Copy

Rechner R2 sei der Primärkopien-Rechner. Kommunikation entsteht somit nur für die Zugriffe in den anderen Rechnern.

Für die Änderungszugriffe in R1 fallen 10*(2+4) = 60 Nachr./s für die Kommunikation mit R2 an. Für die Lesezugriffe in R4 entstehen wieder 80 Nachr./s; die Lesezugriffe in R1 und R3 sind lokal, wenn auf Konsistenz verzichtet wird, anderenfalls ist eine Interaktion mit dem Primary-Copy-Rechner R2 erforderlich (Sperrerwerb bzw. Objektzugriff). Dafür fallen (30 +50) * (2 + 2) = 320 Nachr./s an.
Daneben verursacht noch das asynchrone Ändern der Kopien durch den Primary-Copy-Rechner Kommunikationsaufwand. Die Änderungen von R1 sind zusätzlich an R3 zu propagieren, die von R2 an R1 und R3. Setzt man pro Änderung jeweils 2 Nachrichten an, ergeben sich 10*2 + 20*2*2 = 100 Nachr./s.

Der Gesamtaufwand für Primary-Copy-Verfahren beträgt somit 560 Nachr./s (bzw. 240 Nachr./s bei inkonsistentem Lesen), wobei 100 Nachr./s asynchron abgewickelt werden.

Aufgabe 9-4: Voting-Verfahren

T1: P1; T2: P2; T3: -

T4 könnte für die gegebene Partitionierung weiterhin bearbeitet werden, indem man R1 oder R3 (welche Kopien von A und B halten) eine für Änderungen ausreichende Stimmenzahl zuordnet. Eine Möglichkeit wäre die für A vorliegenden Stimmen (<2, -, 1> und Quoren (r=2, w=2) beizubehalten und für B die Stimmenverteilung <3, 1, 1> sowie w=3, r=3 zu wählen. In diesem Fall könnte T4 lokal auf R1 (also in Partition P1) ausgeführt werden.

Aufgabe 9-5: Katastrophen-Recovery

Systemkonfigurationen zur Katastrophen-Recovery: Beschränkung auf zwei Knoten (Rechenzentren); meist geringe Nutzung der Backup-Kopie im Normalbetrieb; keine Unterstützung für Netzwerk-Partitionierungen; effizientere Propagierung von Änderungen; keine Synchronisation auf Backup-Kopie; neben Daten sind im Backup-System auch Anwendungen sowie Netzwerkverbindungen repliziert.

Unterschiede zur Schnappschuß-Replikation: Hauptziel hohe Verfügbarkeit; meist Führen einer vollständigen DB-Kopie anstatt benutzerspezifischer Teilmengen; Backup-Kopie meist nahezu aktuell, während bei Schnappschüssen benutzerdefinierte Aktualisierungsintervalle bestehen

Aufgabe 11-1: Beschränkung globaler Änderungstransaktionen

a) Ohne globales Commit-Protokoll erfolgt am Ende jeder lokalen Sub-Transaktion ein Commit mit Sperrfreigabe. Damit ist z.B. folgender, nicht serialisierbare Ablauf für zwei globale Transaktionen GT_1 und GT_2 möglich (c_i = lokales Commit einer Sub-Transaktion von GT_i):

LDBS1: $r_1(x), c_1, w_2(x), c_2$ $(GT_1 < GT_2)$
LDBS2: $r_2(y), c_2, w_1(y), c_1$ $(GT_2 < GT_1)$

b) ja

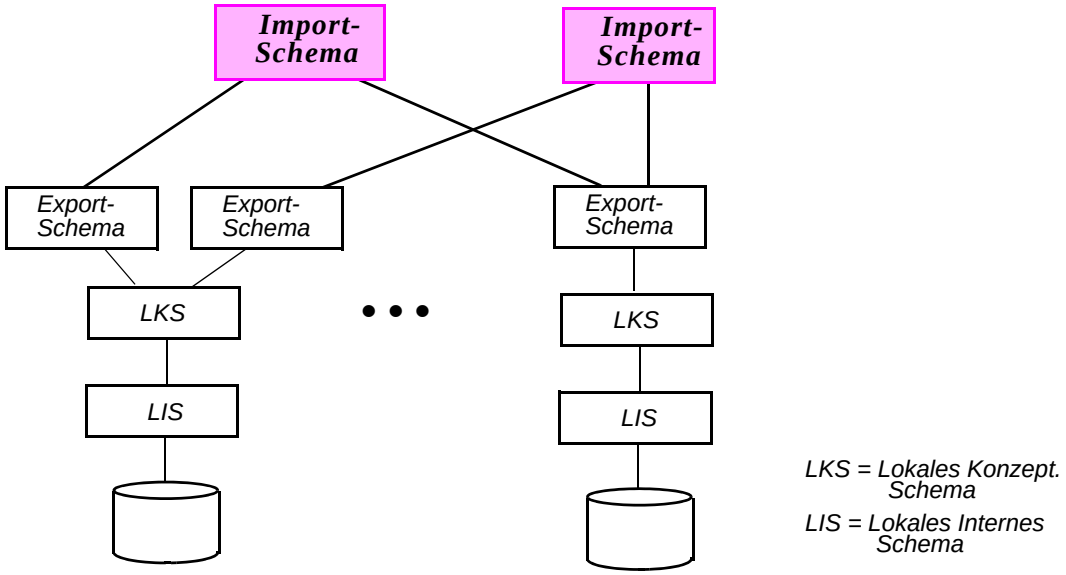
Aufgabe 11-2: Gateway-Anzahl

a) 10*5 = 50, b) 10+5 = 15, c) 5

Aufgabe 11-3: Abruf großer Ergebnismengen

Der satzweise Transport der Ergebnismenge kann durch gebündelte Übertragung und Pufferung mehrerer Sätze bzw. des gesamten Ergebnisses auf Client-Seite vermieden werden. Dies setzt dort jedoch zumindest eine begrenzte DBVS-Funktionalität zur Pufferung und Cursor-Verwaltung voraus, insbesondere um einzelne Satzanforderungen (FETCH-Aufrufe) zu bearbeiten. Dies läßt sich in homogenen Umgebungen (Tools und DBVS eines Herstellers) am einfachsten realisieren und wird auch von verschiedenen Produkten bereits unterstützt ("Block Fetch"). Im heterogenen Fall könnten DB-Gateways eine ähnliche Funktionalität unterstützen, sofern sie auf Client-Seite ausgeführt werden. Eine Alternative wäre eine neue Systemkomponente auf Client-Seite für Pufferung und Cursor-Verwaltung anzubieten, die dann mit den einzelnen DBVS bzw. Gateways auf Server-Seite kommuniziert.

Aufgabe 12-1: Schemaarchitektur eines lose gekoppelten FDBS



Aufgabe 12-2: Semantische Heterogenität

Namenskonflikt: Synonyme PNAME und BNAME

Struktureller Konflikt: unterschiedliche Repräsentation von Adressen (einfaches Attribut ANSCHRIFT vs. drei Attributen)

Integritätsbedingungen: unterschiedliche Primärschlüssel (PNR bzw. BNR), die unabhängig voneinander vergeben werden; unterschiedliche Datentypen für PNAME und BNAME (40 vs. 50 Zeichen maximale Länge) sowie für Adressattribute

Aufgabe 12-3: Schemaintegration

Relation PERSON (*PNR* *INTEGER, {Personalnummer}*
BNR *INTEGER, {Benutzernummer, optional}*
NAME *CHAR (50),*
WOHNORT *CHAR (30),*
PLZ *INTEGER,*
STRASSE *CHAR (30), ...)*

Für Bibliotheksbenutzer ohne Personalnummer muß diese durch eine eigene Funktion erzeugt werden. Daneben werden Transformationsfunktionen benötigt, um aus dem einfachen Attribut ANSCHRIFT die drei Einzelkomponenten zu extrahieren. Gegebenenfalls ist auch eine Transformation zur Angleichung der Datenrepräsentation für Namen erforderlich.

Aufgabe 12-4: Query-Optimierung in FDBS

Zur Optimierung stehen prinzipiell nur die Angaben der Export-Schemata zur Verfügung. Ohne Kenntnis des vollständigen DB-Aufbaus, von Angaben der internen Schemata sowie der von den LDBS unterstützten Algorithmen (z.B. zur Join-Berechnung) kann jedoch keine ausreichende Kostenabschätzung verschiedener Ausführungspläne vorgenommen werden. Die Bereitstellung dieser Informationen beeinträchtigt jedoch die Autonomie der LDBS.

Aufgabe 12-5: Quasi-Serialisierbarkeit

Es liegt keine Serialisierbarkeit vor, da gilt:

LDBS1: $GT_1 < LT_3 < GT_2$
LDBS2: $GT_2 < LT_4 < GT_1$.

Der Schedule ist jedoch quasi-serialisierbar, da die Konflikte in LDBS1 irrelevant sind und die Sub-Transaktion von GT_2 vor der von GT_1 ausgeführt wird. Somit besteht in beiden Knoten die quasi-serielle Ausführungsreihenfolge GT_2 vor GT_1.

Aufgabe 13-1: Abhängigkeiten zwischen Systemfunktionen

Synchronisation <-> Lastverteilung: Nutzung von Lokalität zur Einsparung von Synchronisationsnachrichten sowie globaler Konflikte; Lokalität durch Lastverteilung zu unterstützen (affinätsbasiertes Routing)

Synchronisation <-> Crash-Recovery: korrekte Fortführung des Synchronisationsprotokolls im Fehlerfall (Rekonstruktion globaler Sperrtabellen u.ä.); Änderungen während Recovery mit sonstiger Transaktionsverarbeitung zu synchronisieren

Lastverteilung <-> Crash-Recovery: Umstellung der Routing-Strategie (ausgefallenem Rechner können keine Transaktionen mehr zugewiesen werden; Neustart betroffener Transaktionen auf überlebendem Rechner)

Aufgabe 13-2: GEH-Synchronisation

Wenigstens je zwei GEH-Zugriffe zum Setzen bzw. Freigeben einer Sperre: Lesen des Sperreintrages; Modifikation im Hauptspeicher, Zurückschreiben in GEH mit Compare&Swap (um sicherzustellen, daß nicht anderer Rechner gleichzeitig denselben Eintrag modifiziert). Bei Verwaltung von Sperreinträgen innerhalb Hash-Tabelle sind ggf. weitere Lesezugriffe aufgrund von Namenskollisionen erforderlich. Doppeltes Führen der Sperrtabelle erfordert nur zweifache Ausführung der Schreibzugriffe (GEH-Zugriffshäufigkeit steigt um maximal 50%)

Aufgabe 14-1: Zentrale Sperrprotokolle

Aufgrund lokaler Verteilung ist keine Knotenautonomie mehr zu unterstützen; Kommunikation zur Synchronisation für Shared-Disk unabdingbar (bei loser Kopplung), während sie in Verteilten DBS mit verteiltem Sperrprotokoll vermeidbar ist

Aufgabe 14-2: GLM-Ausfall

Im Extremfall sind alle laufenden Transaktionen abzubrechen, und die Verarbeitung muß mit "leerer" globaler Sperrtabelle auf einem neuen GLM-Rechner beginnen. Das Abbrechen laufender Transaktionen kann umgangen werden, wenn die globale Sperrtabelle aus den Sperrangaben der lokalen Sperrtabellen der Verarbeitungsrechner rekonstruiert wird, was i.a. möglich ist.

Aufgabe 14-3: Dedizierte Sperrverfahren

größeres Potential zur Nachrichtenbündelung; weniger Nachrichten zur Sperrfreigabe; komplette Sicht auf globale Sperrsituation erleichtert globale Deadlock-Erkennung

Aufgabe 14-4: Lese- und Schreibautorisierungen

Sperranforderungen:

- mit Schreibautorisierungen: 22 Nachrichten (Entzug der Schreibautorisierung mit jeweils 4 Nachrichten zu den Zeitpunkten t4, t5, t6, t7 und t8)
- mit Leseautorisierungen: 18 Nachrichten (RA-Entzug zu Zeitpunkt t9: 6 Nachrichten)
- mit Lese- und Schreibautorisierungen: 14 Nachrichten (t4: WA-Entzug mit Konvertierung nach RA für R1; t9: RA-Entzug mit WA-Gewährung für R3)

Nutzung der Schreibautorisierungen allein ungünstig wegen geringer rechnerspezifischer Lokalität;Wirksamkeit der Leseautorisierungen durch X-Sperren eingeschränkt

Sperrfreigabe:

- Ohne Autorisierungen: 9 globale Sperrfreigaben (9 Nachrichten)
- Bei Verwendung von Schreibautorisierungen sowie von Lese- und Schreibautorisierungen werden in dem Beispiel keine zusätzlichen Nachrichten zur Sperrfreigabe erforderlich, da Sperren zusammen mit WA-/RA-Entzug freigegeben werden !
- Bei alleiniger Nutzung von Leseautorisierungen sind 3 X-Sperren explizit zurückzugeben (t1, t3, t9)

Gesamtnachrichtenanzahl:

- ohne Autorisierungen:	27 (18 + 9)
- Schreibautorisierungen:	22 (22 + 0)

- Leseautorisierungen: 21 (18 + 3)
- Lese-/Schreibautorisierungen: 14 (14 + 0)

Aufgabe 14-5: Primary-Copy-Sperrverfahren

ohne Leseautorisierung: 15 (10+5) Nachrichten
5 globale Sperranforderungen
=> 10 Nachrichten zur Sperranforderung, 5 zur Freigabe

mit Leseautorisierung: 10 (8+2) Nachrichten
t4, t5: RA-Zuweisung an R2 und R3
t9: RA-Entzug (4 Nachrichten)
globale Sperrfreigabe für X-Sperre von R3 (t9)

Aufgabe 14-6: Dynamische GLA-Zuordnung

Beibehaltung der GLA in Rechner R1: 21 Nachrichten
t1: GLA-Übernahme (2 Nachrichten)
t2, t3, t6: lokal
t4, t7: je 4+1 Nachrichten für Sperranforderung/freigabe (10 Nachrichten)
t5,t8,t9: je 2+1 Nachrichten (9 Nachrichten)

Schreibautorisierungen im Beispiel nicht sinnvoll, da gute GLA-Nutzung.
Leseautorisierungen sinnvoll (17 statt 21 Nachrichten): lokale Synchronisation für t7 und t8 (Einsparung von 8 Nachrichten); Entzug für t9 kostet 4 zusätzliche Nachrichten

GLA-Migration im Beispiel weniger sinnvoll, da GLA-Änderung zu den Zeitpunkten t1, t4, t5, t6, t7 und t8 anfällt (Anpassung der Directory-Information, dafür stets lokale Sperrfreigabe). Gesamtaufwand: 20 Nachrichten

Aufgabe 15-1: Seitentransfers

schnellere Bereitstellung der Seite als bei Seitenaustausch über Platte, da Einlesen im Normalfall entfällt; langsamer als direkter Seitentransfer, dafür einfachere Recovery wie für Seitenaustausch über Platte

Aufgabe 15-2: On-Request-Invalidierung und Autorisierungen

Die Konzepte sind verträglich, da lokale Autorisierung garantiert, daß seit der Zuweisung durch den GLM die Seite nicht an einem anderen Rechner geändert wurde. Folglich kann keine Pufferinvalidierung für die Seite vorliegen.

Aufgabe 15-3: On-Request-Invalidierung

a) dediziertes Sperrprotokoll

1. X-Anforderung durch R1 (2 Nachrichten);
   Plattenzugriff zum Einlesen von B;
   X-Freigabe (1 Nachr.): I=0111, PO=R1 (Page-Owner)
2. R-Anforderung durch R4 (2 Nachr.)
   Seitentransfer R1 -> R4 (2 Nachr.), I = 0110, PO=R1
   R-Freigabe (1 Nachr.)
3. X-Anforderung durch R2 (2 Nachr.)
   Seitentransfer R1 -> R2, I=0010 (2 Nachr.)
   X-Freigabe (1 Nachr.): I=1011, PO=R2

=> 13 Nachrichten, davon 2 Seitentransfers; 1 Lese-E/A

b) Primary-Copy-Sperrverfahren

1. X-Anforderung durch R1 an GLA-Rechner R2 (2 Nachr.);
   Seite B kann mit Sperrgewährung an R1 übertragen werden;
   X-Freigabe mit Übergabe der neuen Seitenversion an R2 (1 Nachr.): I=0011
2. R-Anforderung durch R4 (2 Nachr.)
   Seitentransfer R1 -> R4 kann mit Sperrgewährung kombiniert werden, I = 0010
   R-Freigabe (1 Nachr.)
3. X-Anforderung durch R2 (lokal)
   X-Freigabe (lokal): I=1011

=> 6 Nachrichten, davon 3 Seitentransfers; keine Lese-E/A

Aufgabe 15-4: Haltesperren

1. X-Anforderung durch R1 an GLM;
   RA-Entzug für R2 und R3 mit Eliminierung von Seite B;
   Sperrgewährung an R1 (6 Nachrichten); R1: WA, PO=R1
   Einlesen von Seite B von Platte
2. R-Anforderung durch R4 an GLM;
   WA-Entzug für R1 (Konversion nach RA); Seitentransfer R1 -> R4;
   RA-Gewährung an R4 (5 Nachrichten); R1, R4: RA, PO=R1
3. X-Anforderung durch R2, RA-Entzug für R1 und R4;
   Seitentransfer R1 -> R2; WA-Gewährung an R2 (7 Nachrichten); R2: WA; PO=R2

=> 18 Nachrichten, davon 2 Seitentransfers; 1 Lese-E/A

sehr hohe Nachrichtenanzahl durch RA/WA-Entzug; Haltesperren erlauben im Beispiel keine lokale Sperrvergabe

Aufgabe 15-5: Kohärenzkontrolle bei dynamischer GLA-Zuordnung

Nach Kap. 15.6 empfiehlt sich eine On-Request-Invalidierung mit dynamischer Page-Owner-Zuordnung. Kommunikationseinsparungen sind möglich, wenn der GLM-Rechner einer Seite auch die Page-Owner-Funktion wahrnimmt. Dies erfordert eine Migration der GLA an den Rechner, wo eine Seite geändert wird. Damit können Seitenanforderungen stets mit Sperranforderungen kombiniert werden, und Seiten werden mit der Sperrgewährung bereitgestellt. Dafür sind die Angaben zur GLM-Lokalisierung häufig anzupassen.

Aufgabe 15-6: Kohärenzkontrolle in Workstation/Server-DBS

Der Server-Rechner fungiert als GLM-Rechner sowie zentraler Page-Owner für die gesamte Datenbank, so daß Sperr- und Seitenanforderungen stets kombiniert werden. Zudem erfolgt die Übertragung von Änderungen zum Server zusammen mit der Sperrfreigabe. Zur Behandlung von Pufferinvalidierungen kommt sowohl eine On-Request-Invalidierung als auch der Einsatz von Haltesperren in Betracht. Damit sind folgende Verfahrenskombinationen als sinnvoll zu erachten: S1 + P3/P4 + U2.

Lese/Schreibautorisierungen erlauben Kommunikationseinsparungen, da sie eine lokale Sperrvergabe in den Workstations gestatten und die Gültigkeit lokaler Seitenkopien garantieren (auch bei On-Request-Invalidierung). Damit wird auch die Anzahl von Seitentransfers zum Server reduziert, da erst beim Entzug einer Schreibautorisierung eine Übertragung notwendig ist. Bei entsprechender Lokalität können somit viele Änderungen vor einer Übertragung akkumuliert werden. Der Seitentransfer erfolgt zudem asynchron

bezüglich der ändernden Transaktionen (die Log-Daten sind jedoch aus Sicherheitsgründen beim Commit zum Server zu transferieren).

Hierarchische Sperren erlauben weitergehende Kommunikationseinsparungen, vor allem wenn beim Sperrerwerb zugleich große Datenmengen in die Workstation gebracht werden.

Aufgabe 15-7: Kohärenzkontrolle bei der Katalogverwaltung

SDD-1: Multicast-Invalidierung; R*: On-Request-Invalidierung

Aufgabe 16-1: Antwortzeit-Speedup

100-MIPS-Prozessor: AZ = 10 (CPU) + 50 (E/A) = 60 ms; Speedup = 150/60 = 2.5
E/A verhindert linearen Speedup (Amdahls Gesetz) !

10*10 MIPS: AZ = 10 (CPU) + 5 (E/A) + 0.15 (Komm.) = 15,15 ms; Speedup = 9.9
(linearer Speedup)

100*1 MIPS: AZ = 10 + 0.5 + 1.05 = 11.55 ms; Speedup = 13
(besser als 10*10 MIPS, da weniger E/A pro Rechner)

Aufgabe 16-2: Parallelisierung von Transaktionsprogrammen

Zerlegung in vier parallele Teilprogramme mit Zugriffen auf ACCOUNT (2 Operationen), TELLER, BRANCH und HISTORY.

Aufgabe 17-1: Bestimmung des Verteilgrades

Es gilt p_{opt} = SQRT (c * K / b)

Relationen-Scan: K = 10^7
p_{opt} = SQRT ($5*10^{-5} * 10^7 / 5*10^{-3}$)= SQRT (10^5) = 316 > N = 100
=> p_{opt} = 100

Index-Scan (1%): K = 10^5
p_{opt} = SQRT ($5*10^{-5} * 10^5 / 5*10^{-3}$)= SQRT (10^3) = 31,6

Gewichtetes Mittel: D = 0,3*100 + 0,7*31,6 = 52

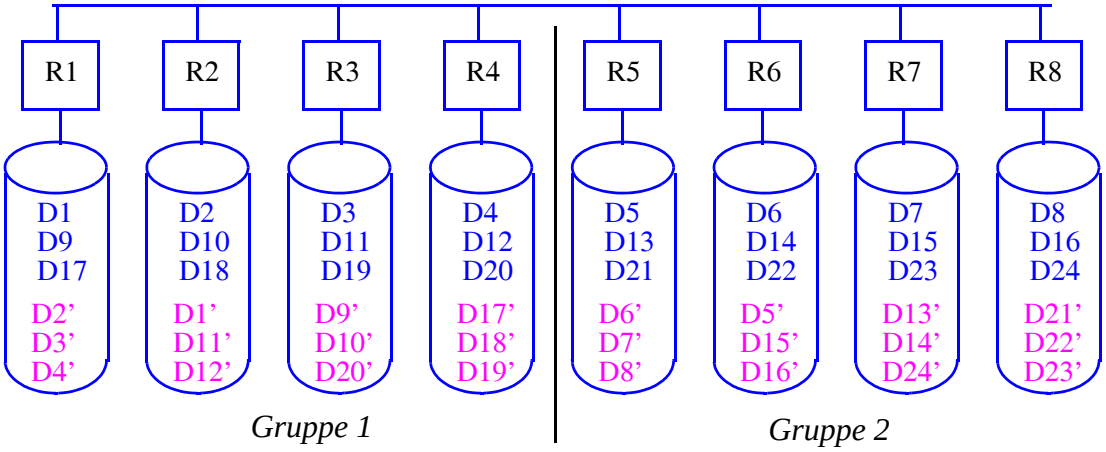
Aufgabe 17-2: Verteilattribut

hohe Zugriffshäufigkeit; meist hoher Anteil von Exact-Match-Anfragen, die dann mit minimalem Kommunikationsaufwand (1 Rechner) bearbeitbar sind

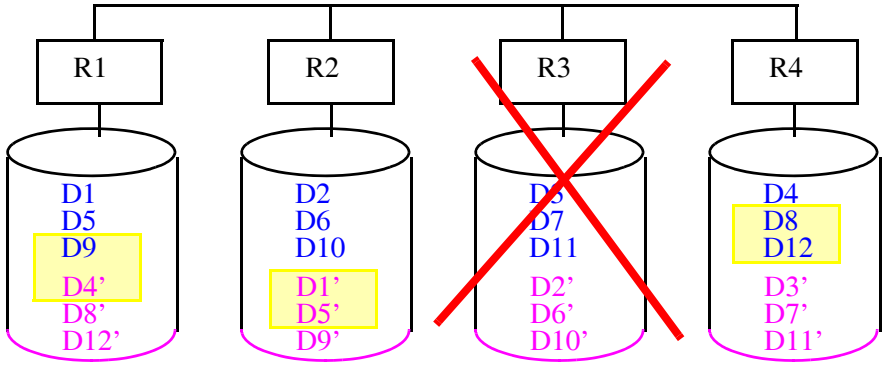
Aufgabe 17-3: Verteilinformationen in Indexstrukturen

Ein Problem liegt in der Speicherung dieser erweiterten Indexstruktur selbst. Bei vollkommener Replikation ensteht ein sehr hoher Änderungsaufwand, da jede Einfüge- und Löschoperation auf der Relation sowie Änderungen bezüglich des indizierten Attributs auf allen Kopien anzuwenden sind. Bei zentralisierter oder partitionierter Speicherung fallen zusätzliche Kommunikationsverzögerungen für den Indexzugriff an. Generell ist daneben mit zusätzlichen E/A-Operationen zu rechnen, da der Index im Gegensatz zur Fragmentierungsinformation i.a. nicht im Hauptspeicher gehalten werden kann. Für Sekundärschlüssel, bei denen einzelne Attributwerte mehrfach vorkommen, können zudem einzelne Attributwerte über viele Knoten verteilt gespeichert sein, zumal die Datenverteilung hinsichtlich der Werteverteilung eines anderen Attributs erfolgte. Somit muß die Anfrage ggf. dennoch auf einem Großteil der Rechner verarbeitet werden.

Aufgabe 17-4: Verstreute Replikation



Aufgabe 17-5: Verkettete Replikation



Die unterlegten Bereiche, werden nach Ausfall von R3 nicht genutzt..Jeder Rechner verwaltet damit 4 Bereiche.

Aufgabe 18-1: Parallele Join-Berechnung

R: 1 Mill. · 100 B = 100 MB; S: 100.000 · 100 B = 10 MB

Dynamische Replikation (Verschicken der S-Relation an alle R-Knoten):

10 MB · 4 = 40 MB

Dynamische Partitionierung (Umverteilung von R und S):

100 + 10 MB = 110 MB

Dynamische Partitionierung ohne Umverteilung von S: 100 MB

Die dynamische Replikation verursacht in diesem Beispiel den geringsten Kommunikationsumfang !

Aufgabe 18-2: Mindest-Hauptspeichergröße für partitionierten Hash-Join

Für q Partitionen werden zur Partitionierung wenigstens q+1 Seiten im Hauptspeicher benötigt (1 Eingabepuffer, q Ausgabepuffer). Für die Hash-Tabelle jeder S-Partition werden wenigstens b/q Seiten notwendig. Setzt man q=b/q, ergibt sich q= $\sqrt{b}$, so daß der minimale Hauptspeicherumfang $\sqrt{b}$ + 1 Seiten beträgt.

Aufgabe 18-3: TID-Hash-Join

Der TID-Hash-Join zahlt sich vor allem aus, wenn damit die E/A-Vorgänge zur Überlaufbehandlung vollkommen umgangen werden. Die Einsparungen sind bei kurzen Join-Attributen bzw. langen Sätzen am höchsten (Bsp.: Komprimierung der Hash-Tabelle um Faktor 10 für Satzlänge 100 B gegenüber Schlüssel+TID von 10 B). Nachteilig ist, daß eine Nachbearbeitung notwendig wird, um die im Verbundergebnis enthaltenen Tupel der kleineren Relation über die TIDs zu rekonstruieren. Der Aufwand für die damit verbundenen Externspeicherzugriffe ist umso höher, je mehr Tupel sich qualifizieren. Im verteilten Fall (Shared-Nothing) ist der Ansatz weniger attraktiv, da bei einer Join-Berechnung außerhalb der Datenknoten keine direkte Auflösung von TID-Verweisen möglich ist (-> Kommunikation).

Aufgabe 18-4: Paralleler Hash-Join/Überlaufbehandlung an Datenknoten

1. *Paralleles Lesen und Partitionieren von S*
   an jedem S-Knoten j (j=1 .. m) führe parallel durch:
   lies lokale S-Partition S_j und partitioniere sie in q Sub-Partitionen zur Überlaufbehandlung S_{j1} bis S_{jq} durch Anwendung einer Partitionierungsfunktion g auf dem Join-Attribut
2. *Paralleles Lesen und Partitionieren von R*
   an jedem R-Knoten i (i=1..n) führe parallel durch:
   lies lokale R-Partition R_i und partitioniere sie in q Sub-Partitionen R_{i1} bis R_{iq} (Partitionierungsfunktion g)
3. *Parallele Umverteilung und Join-Berechnung*
   für *l* = 1, 2 bis q führe nacheinander aus:
   - an jedem S-Knoten j (j=1 .. m) lies Sub-Partition S_{jl} und sende jedes S-Tupel an zuständigen Join-Rechner (Verteilungsfunktion f)
   - in jedem Join-Knoten k (k=1...p) füge eingehende S-Tupel in Hash-Tabelle ein
   - an jedem R-Knoten i (i=1 .. m) lies Sub-Partition R_{il} und sende jedes R-Tupel an zuständigen Join-Rechner (Verteilungsfunktion f)
   - in jedem Join-Knoten k (k=1...p) überprüfe Hash-Tabelle für eingehende R-Tupel und bilde das Join-Ergebnis

Aufgabe 18-5: Paralleler Hash-Join bei Shared-Everything

Voraussetzung für die Nutzung von Datenparallelität ist auch bei SE die verteilte Speicherung der Eingaberelationen sowie ggf. temporärer Dateien auf mehreren Platten. Weiterhin muß eine Aufteilung des gemeinsamen Hauptspeichers in mehrere Bereiche erfolgen, in denen parallel die Join-Berechnung durch unterschiedliche Join-Prozesse erfolgen kann. Das Einlesen der ersten Relation erfolgt parallel durch mehrere Leseprozesse, wobei jeder Prozeß auf disjunkte Plattenmengen zugreift. Die eingelesenen Tupel werden dann - ggf. durch separate Prozesse - in die zugehörige Hash-Tabelle eingetragen, wobei eine Synchronisation (über Semaphore) notwendig wird. Das Lesen der zweiten Relation erfolgt analog, wobei die Join-Berechnung nur Lesezugriffe auf die Hash-Tabellen und somit keine Synchronisation erfordert. Die Join-Ergebnisse werden durch spezielle Schreibprozesse (asynchron) geschrieben. Der Datenaustausch zwischen Lese-, Join- und Schreibprozessen erfordert synchronisierte Zugriffe auf gemeinsame Pufferbereiche; dafür entfällt der Kommunikations-Overhead zur Datenumverteilung von SN-Systemen.

Aufgabe 18-6: Daten-Skew

Hash-Funktion führt zu Umverteilungs- und Join-Produkt-Skew. Das Join-Ergebnis von rund 100.000 Tupeln ist gleichmäßig von 5 Prozessoren zu generieren. Hierzu kann eine überlappende Bereichsfragmentierung gewählt werden, mit 2 bzw. 3 Partitionen für Wert 1 bzw. 2, wobei jede der 5 Partitionen einem anderen Prozessor zugewiesen wird. Die restlichen Werte können daneben gleichmäßig 5 weiteren Bereichen zugeteilt werden (z.B., 3-202, 203-402, 403-601, 602-801, 802-1000), die von jeweils einem der Join-Prozessoren zu bearbeiten sind.

Aufgabe 18-7: Parallele Änderungstransaktionen bei Shared-Disk

Parallele Teiltransaktionen verschiedener Rechner können dieselben Daten referenzieren und ändern, so daß eine Synchronisation zwischen Teiltransaktionen erforderlich ist (z.B. im Rahmen eines geschachtelten Transaktionskonzepts). Ebenso ist die Kohärenzkontrolle zu erweitern, um Änderungen zwischen Teiltransaktionen auszutauschen. Die verteilte Transaktionsausführung mit Änderungen an verschiedenen Rechnern verlangt zudem ein verteiltes Commit-Protokoll.

Index

Literatur

ABG90 Alonso, R., Barbara, D., Garcia-Molina, H.: *Data Caching Issues in an Information Retrieval System*. **ACM Trans. on Database Systems** 15 (3), 359-384, 1990

ACD83 Agrawal, R., Carey, M.J., DeWitt, D.J.: *Deadlock Detection is Cheap*. **ACM Sigmod Record** 13 (2), 19-34, 1983

ACL87 Agrawal, R., Carey, M.J., Livny, M.*: Concurrency Control Performance Modeling: Alternatives and Implications.* **ACM Trans. on Database Systems** 12 (4), 609-654, 1987

ACM87 Agrawal, R., Carey, M.J., McVoy, L.W.: *The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems*. **IEEE Trans. on Software Engineering** 13 (12), 1348-1363, 1987

AD76 Alsberg, P.A., Day, J.D.: *A Principle of Resilient Sharing of Distributed Resources.* **Proc. 2nd Int. Conf. on Software Engineering**, 562-570, 1976

Ah91 Ahmed, R. et al.: *The Pegasus Heterogeneous Multidatabase System.* **IEEE Computer** 24 (12), 19-27, Dec. 1991

AL80 Abida, M., Lindsay, B.: *Database Snapshots.* **Proc. 6th Int. Conf. on Very Large Data Bases**, 86-91, 1980

Ap88 Apers, P.M.G.: *Data Allocation in Distributed Database Systems*. **ACM Trans. on Database Systems**, 13 (3), 263-304, 1988

Ap92 Apers, P.M.G. et al.: *PRISMA/DB: A Parallel, Main-Memory Relational DBMS*. **IEEE Trans. on Knowledge and Data Engineering** 4, 1992

Ar91 Arnold, D. et al.: *SQL Access: An Implementation of the ISO Remote Database Access Standard*. **IEEE Computer** 24 (12), 74-78, Dec. 1991

AS89 Agrawal, D., Sengupta, S.: *Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control*. **Proc. ACM SIGMOD Conf**., 408-417, 1989

Ba91 Balboni, J.H.: *SQL Access Overview.* **Proc. IEEE Spring CompCon Conf**., 114-119, 1991

BC92 Bober, P.M., Carey, M.J.: *On Mixing Queries and Transactions via Multiversion Locking*. **Proc. 8th IEEE Data Engineering Conf**., 535-545, 1992

BDS79 Behman, S.B, Denatale, T.A., Shomler, R.W.: *Limited Lock Facility in a DASD Control Unit*. Technical report TR 02.859, IBM General Products Division, San Jose, 1979

Be81 Bernstein, P.A. et al.: *Query Processing in a System for Distributed Databases (SDD-1).* **ACM Trans. on Database Systems**, 6 (4), 602-625, 1981

Be90 Bernstein, P.A.: *Transaction Processing Monitors*. **Comm. ACM** 33 (11), 75-86, 1990

Be93a Bernstein, P.A.: *Middleware - An Architecture for Distributed System Service*. DEC Cambridge Research Lab., TR 93/6, March 1993

Be93b Bernstein, P.A. *STDL - A Portable Language for Transaction Processing*. **Proc. 19th Int. Conf. on Very Large Data Bases**, 218-229, 1993

BEHR82 Bayer, R., Elhardt, K., Heller, H., Reiser, A.: *Dynamic Timestamp Allocation for Transactions in Database Systems.* **Distributed Data Bases**, North-Holland, 9-20, 1982

BEKK84 Bayer, R., Elhardt, K., Kießling, K., Killar, D.: *Verteilte Datenbanksysteme - Eine Übersicht über den heutigen Entwicklungsstand*. **Informatik-Spektrum** 7 (1), 1-19, 1984

BG81 Bernstein, P.A., Goodman, N.: *The Poweer of Natural Semijoins.* **SIAM J. Computing** 10 (4), 751-771, 1981

BG83 Bernstein, P.A., Goodman, N.: *Multiversion Concurrency Control - Theory and Algorithms*. **ACM Trans. on Database Systems**, 8 (4), 465-483, 1983

BG92 Bell, D., Grimson, J.: **Distributed Database Systems**. Addison Wesley, 1992

Bh88 Bhide, A.: *An Analysis of Three Transaction Processing Architectures*. **Proc. 14th Int. Conf. on Very Large Data Bases**, 339-350, 1988

BHG87 Bernstein, P.A., Hadzilacos, V., Goodman, N.*:* **Concurrency Control and Recovery in Database Systems***.* Addison Wesley, 1987

BHM90 Bernstein, P.A., Hsu, M., Mann, B.: *Implementing Recoverable Requests Using Queues*. **Proc. ACM SIGMOD Conf**., 112-122, 1990

BHP92 Bright, M.W., Hurson, A.R., Pakzad, S.H.: *A Taxonomy and Current Issues in Multidatabase Systems.* **IEEE Computer** 25 (3), 50-60, March 1992

Bi89 Bitton, D.: *Arm Scheduling in Shadowed Disks*. **Proc. IEEE Spring CompCon Conf.**, 132-136, 1989

Bl70 Bloom, B.H.: *Space/Time Tradeoffs in Hash Coding with Allowable Errors*. **Comm. ACM** 13 (7), 422-426, 1970

BLN86 Batini, C., Lenzirini, M., Navathe, S.B.: *A Comparative Analysis of Methodologies for Database Schema Integration.* **ACM Computing Surveys** 18 (4), 323-364, 1986

Bo81 Borr, A.: *Transaction Monitoring in ENCOMPASS*. **Proc. 7th Int. Conf. on Very Large Data Bases**,1981

Bo90 Boral, H. et al.: *Prototyping Bubba: A Highly Parallel Database System*. **IEEE Trans. on Knowledge and Data Engineering** *2,* 1, 4-24, 1990

Bo91 Borghoff, U.M.: *Fehlertoleranz in verteilten Dateisystemen: Eine Übersicht über den heutigen Entwicklungsstand bei den Votierungsverfahren*. **Informatik-Spektrum** 14 (1), 15-27, 1991

BR92 Butsch, K., Rahm, E.: *Architekturansätze zur Unterstützung heterogener Datenbanken*. **Proc. 12. GI/ITG-Fachtagung "Architektur von Rechensystemen",** Springer-Verlag, Informatik aktuell, 106-118, 1992

BSR80 Bernstein, P.A., Shipman, D.W., Rothnie, J.B.: *Concurrency Control in a System for Distributed Databases (SDD-1)*. **ACM Trans. on Database Systems** 5 (1), 18-51, 1980

BST90 Breitbart, Y., Silberschatz, A., Thompson, G.R.: *Reliable Transaction Management in a Multidatabase System.* **Proc. ACM SIGMOD Conf**., 1990

BT90 Burkes, D.L., Treiber, R.K.: *Design Approaches for Real-Time Transaction Processing Remote Site Recovery*. **Proc. IEEE Spring CompCon Conf**., 568-572, 1990

Bü91 Bültzingsloewen, G. v.: **SQL-Anfragen.** FZI-Berichte Informatik, Springer-Verlag 1991

CA91 *CA-DB:STAR - Distributed Database Processing for the MVS and VSE Environments*. Computer Associates 1991

CABK88 Copeland, G., Alexander, W., Boughter, E., Keller, T.: *Data Placement in Bubba.* **Proc. ACM SIGMOD Conf**., 99-108, 1988

CFLS91 Carey, M.J., Franklin, M.J., Livny, M., Shekita, E.J.: *Data Caching Tradeoffs in Client-Server DBMS Architectures*. **Proc. ACM SIGMOD Conf.**, 357-366, 1991

Ch82 Chan, A. et al.: *The Implementation of an Integrated Concurrency Control and Recovery Scheme*. **Proc. ACM SIGMOD Conf***.,* 184-191, 1982

CK89 Copeland, G., Keller, T.: *A Comparison of High-Availability Media Recovery Techniques.* **Proc. ACM SIGMOD Conf**., 98-109, 1989

CL88 Carey, M.J., Livny, M.: *Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication.* **Proc. 14th Int. Conf. on Very Large Data Bases**, 13-25, 1988

Cl92 Claybrook, B.: **OLTP Online Transaction Processing Systems**. John Wiley & Sons, 1992

CLYY92 Chen, M., Lo, M., Yu, P.S., Young, H.C.: *Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins*. **Proc. 18th Int. Conf. on Very Large Data Bases**, 15-26, 1992

CM86 Carey, M.J., Muhanna, W.A.: *The Performance of Multiversion Concurrency Control Algorithms*. **ACM Trans. on Computer Systems** 4 (4), 338-378, 1986

Co93 Colton, M.: *SYBASE System 10*. Sybase Technical Paper Series, 1993

CP84 Ceri, S., Pelagatti, G*.:* **Distributed Databases. Principles and Systems***.* Mc Graw-Hill, 1984

Cr89 Cristian, F.: *Probabilistic Clock Synchronization*. **Distributed Computing** 3 (3), 146-158, 1989

CS91 Choy, D.M., Selinger, P.G.: *A Distributed Catalog for Heterogeneous Distributed Database Resources.* **Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS-91)**, 236-244, 1991

Cy91 Cypser, R.J.: **Communications for Cooperative Systems**. Addison-Wesley, 1991

CYW92 Chen, M., Yu, P.S., Wu, K.: *Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries*. **Proc. 8th IEEE Data Engineering Conf**., 58-67, 1992

Da86 Dadam, P.: **Verteilte Datenbanken**. Kursunterlagen, Fernuniversität Hagen, 1986

Da90 Date, C.J.: **An Introduction to Database Systems**. 5th edition. Addison Wesley,1990

Da92 Date, C.J.: *Distributed Database: A Closer Look*. In: **Relational Database Writings** (1989-1991). Addison-Wesley, 1992

Da93 Date, C.J.: **A Guide to DB2**. 4th edition. Addison Wesley,1993

Dav92 Davison, W.: *Parallel Index Building in Informix OnLine 6.0.* **Proc. ACM SIGMOD Conf.**, S. 103, 1992

DD92 Date, C.J., Darwen, H.: **A Guide to the SQL Standard**. 3rd edition, Addison Wesley, 1992

DDY91 Dan, A., Dias, D.M., Yu, P.S.: *Analytical Modelling a Hierarchical Buffer for the Data Sharing Environment.* **Proc. ACM SIGMETRICS**, 156-167, 1991

De84 DeWitt, D. et al*.: Implementation Techniques for Main Memory Database Systems.* **Proc. ACM SIGMOD Conf.**, 1-8, 1984

DE89 Du, W., Elmagarmid, A.K.: *Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase.* **Proc. 15th Int. Conf. on Very Large Data Bases**, 347-355, 1989

De90 DeWitt, D.J. et al. 1990. *The Gamma Database Machine Project*. **IEEE Trans. on Knowledge and Data Engineering** 2 ,1, 44-62, 1990

DEK91 Du, W., Elmagarmid, A.K., Kim, W.: *Maintaining Quasi Serializability in Multidatabase Systems.* **Proc. 7th IEEE Data Engineering Conf**., 360-367, 1991

DF82 Dowdy, L., Foster, D.: *Comparative Models of the File Assignment Problem.* **ACM Computing Surveys** 14 (2), 287-313, 1982

DFMV90 DeWitt, D.J., Futtersack, P., Maier, D., Velez, F.: *A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems*. **Proc. 16th Int. Conf. on Very Large Data Bases**, 107-121, 1990

DG92 DeWitt, D.J., Gray, J.: *Parallel Database Systems: The Future of High Performance Database Systems.* **Comm. ACM** 35 (6), 85-98, 1992

DIRY89 Dias, D.M., Iyer, B.R., Robinson, J.T., Yu, P.S.*: Integrated Concurrency-Coherency Controls for Multisystem Data Sharing.* **IEEE Trans. on Software Engineering** 15 (4), 437-448, 1989

DNSS92 DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: *Practical Skew Handling in Parallel Joins*. **Proc. of the 18th Int. Conf. on Very Large Data Bases**, 1992

DY92 Dan, A., Yu, P.S.: *Performance Analysis of Coherency Control Policies through Lock Retention.* **Proc. ACM SIGMOD Conf.**, 114-123, 1992

DY93 Dan, A., Yu, P.S.: *Performance Analysis of Buffer Coherency Policies in a Multi-system Data Sharing Environment*. **IEEE Trans. on Parallel and Distributed Systems** 4 (3), 289-305, 1993

DYJ94 Dan, A., Yu, P.S., Jhingran, A.: *Recovery Analysis of Data Sharing Systems under Deferred Dirty Page Propagation Policies.* IBM Research Report, Yorktown Heights, 1994

EGKS90 Englert, S., Gray, J., Kocher, T., Shath, P.: *A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scale-Up on Large Databases*. **Proc. ACM SIGMETRICS Conf.**, 245-246, 1990

EGLT76 Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: *The notions of consistency and predicate locks in a database system*. **Comm. ACM** 19 (11), 624-633, 1976.

EH84 Effelsberg, W., Härder, T.*: Principles of Database Buffer Management.* **ACM Trans. on Database Systems** 9 (4), 560-595, 1984.

El86 Elmagarmid, A.K.: *A Survey of Distributed Deadlock Detection Algorithms.* **ACM SIGMOD Record** 15 (3), 37-45, 1986

Fe93 Felt, E.P.: *Distributed Transaction Processing in the TUXEDO System.* **Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS-93)**, 266-267, 1993

Fe94 Fecteau, G.: *DATABASE 2 AIX/6000 Parallel Technology.* IBM Software Solutions Lab., 1994

Fi85 Fink, T.: *Design of a Distributed CODASYL Database System*. **Information Systems** 10 (4), 425-440, 1985

FL93 Frohn, J., Lausen, G.: *Integration heterogener relationaler Datenbankschemata mittels eines objektorientierten Datenmodells.* **Proc. 5. GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft"**, Springer-Verlag, Informatik aktuell, 285-295, 1993

GA87 Gray, J., Anderton, M.: *Distributed Computer Systems: Four Case Studies*. **Proc. of the IEEE**, 75 (5), 719-726, 1987

GB85 Garcia-Molina, H., Barbara, D.: *How to Assign Votes in a Distributed System*. **Journal of the ACM** 32, 841-860, 1985

GD90 Ghandeharizadeh, S., DeWitt, D.J.: *Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines.* **Proc. 16th Int. Conf. on Very Large Data Bases**, 481-492, 1990

GDQ92 Ghandeharizadeh, S., DeWitt, D.J., Qureshi, W.: *A Performance Analysis of Alternative Multi-Attribute Declustering Strategies.* **Proc. ACM SIGMOD Conf.**, 29-38, 1992

GHOK81 Gray, J., Homan, P., Obermarck, R., Korth, H.: *A Straw Man Analysis of Probability of Waiting and Deadlock.* IBM Research Report RJ 3066, San Jose, 1981

GHW90 Gray, J., Horst, B., Walker, M.: *Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput*. **Proc. 16th Int. Conf. on Very Large Data Bases**, 148-161, 1990

Gi79 Gifford, D.K.: *Weighted Voting for Replicated Data.* **Proc. 7th ACM Symp. on Operating System Principles (SOSP)**, 150-162, 1979

GK88 Garcia-Molina, H., Kogan, B.: *Node Autonomy in Distributed Systems*. **Proc. Int. Symp. on Databases in Parallel and Distributed Systems**, 158-166, 1988

Go90 Gonschorek, J.: *UTM - Ein universeller Transaktionsmonitor.* In: Das Mainframe-Betriebssystem BS2000, Hrsg.: H. Görling, **State of the Art** 8, Oldenbourg-Verlag, 1990

GPH90 Garcia-Molina, H., Polyzois, C.A., Hagmann, R.: *Two Epoch Algorithms for Disaster Recovery*. **Proc. 16th Int. Conf. on Very Large Data Bases**, 222-230, 1990

Gr78 Gray, J.*: Notes on Data Base Operating Systems.* In: **"Operating Systems - An Advanced Course**", Springer-Verlag, Lecture Notes in Computer Science 60, 393-481, 1978

Gr81 Gray, J.: *The Transaction Concept: Virtues and Limitations*. **Proc. 7th Int. Conf. on Very Large Data Bases**, 144-154, 1981

Gr83 Gray, J.P. et al.: *Advanced Program-to-Program Communication in SNA*. **IBM Systems Journal** 22 (4), 298-318, 1983

Gr86 Gray, J.N.: *An Approach to Decentralized Computer Systems*. **IEEE Trans. on Software Engineering** 12 (6), 684-692, 1986

Gr87 Gray, J.: *Transparency in its Place - The Case Against Transparent Access to Geographically Distributed Data*. **Unix Review** 5 (2), 42-50, 1987; ebenso in: M. Stonebraker (Hrsg.): Readings in Database Systems, 2nd ed., Morgan Kaufmann, 592-602, 1994

Gr92 Grossman, C.P.*: Role of the DASD Storage Control in an Enterprise Systems Connection Environment.* **IBM Systems Journal** 31 (1), 123-146, 1992

Gr93 Gray, J.: *Why TP-Lite Will Dominate the TP Market.* **Proc. 5th Int. Workshop on High Performance Transaction Systems**, Asilomar, Sep. 1993

Gr93b Gray, J. (Hrsg.): **The Benchmark Handbook for Database and Transaction Processing Systems**. (2nd ed.), Morgan Kaufmann Publishers, 1993

GR93 Gray, J., Reuter, A.: **Transaction Processing: Concepts and Techniques**. Morgan Kaufmann Publishers, 1993

Gra88 Graf, A.: *UDS-D: Die verteilte Datenhaltung für UDS*. **Proc. 18. GI-Jahrestagung**, Springer-Verlag, Informatik-Fachberichte 188, 665-674, 1988

Gra93 Graefe, G.: *Query Evaluation Techniques for Large Databases*. **ACM Computing Surveys** 25 (2), 73-170, 1993

Gra94 Graefe, G.: *Volcano - an Extensible and Parallel Query Evaluation System*. **IEEE Trans. on Knowledge and Data Engineering**, 6, 120-135, 1994

GS91 Gray, J., Siewiorek, D.P.: *High-Availability Computer Systems*. **IEEE Computer** 24 (9), 39-48, Sep. 1991

Ha90 Haderle, D.: *Parallelism with IBM's Relational Database2 (DB2).* **Proc. IEEE Spring CompCon Conf.**, 488-489, 1990

Ha92 Hansen, H.R.: **Wirtschaftsinformatik**, 6. Auflage, Uni-Taschenbücher UTB 802, Fischer-Verlag, 1992

Hä79 Härder, T.: *Die Einbettung eines Datenbanksystems in eine Betriebssystem-Umgebung.* In: **Datenbanktechnologie**, Teubner-Verlag, 9-24, 1979

Hä84 Härder, T.: *Observations on Optimistic Concurrency Control*. **Information Systems** 9 (2), 111-120, 1984

Hä87 Härder, T.: *Realisierung von operationalen Schnittstellen*. In [LS87], 163-335

Hä88a Härder, T.: *Transaktionskonzept und Fehlertoleranz in DB/DC-Systemen*. **Handbuch der modernen Datenverarbeitung**, Heft 144, Forkel-Verlag, Nov. 1988

Hä88b Härder, T.: *Handling Hot Spot Data in DB-Sharing Systems*. **Information Systems** 13 (2), 155-166, 1988

HD90 Hsiao, H., DeWitt, D.J.: *Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines*. **Proc. 6th IEEE Data Engineering Conf**., 1990

HD92 Härtig, M., Dittrich, K.: *An Object-Oriented Integration Framework for Building Heterogeneous Database Systems.* **Proc. IFIP Conf. on Semantics of Interoperable Database Systems**, 1992

HK89 Hsiao, D.K., Kamel, M.N.: *Heterogeneous Databases: Proliferations, Issues, and Solutions.* **IEEE Trans. on Knowledge and Data Engineering** 1 ,1, 45-62, 1989

HM86 Härder, T., Meyer-Wegener, K.: *Transaktionssysteme und TP-Monitore - Eine Systematik ihrer Aufgabenstellung und Implementierung*. **Informatik Forschung und Entwicklung** 1, 3-25, 1986

HM90 Härder, T., Meyer-Wegener, K.: *Transaktionssysteme in Workstation/Server-Umgebungen*. **Informatik Forschung und Entwicklung** 5, 127-143, 1990

HMNR95 Härder, T., Mitschang, B., Nink, U., Ritter, N.: *Workstation/Server-Architekturen für datenbankbasierte Ingenieuranwendungen*. **Informatik Forschung und Entwicklung**, 10, 1995

Ho92 Hong, W.: *Exploiting Inter-Operation Parallelism in XPRS*. **Proc. ACM SIGMOD Conf.**, 19-28,1992

HP87 Härder, T., Petry, E.: *Evaluation of Multiple Version Scheme for Concurrency Control.* **Information Systems** 12 (1), 83-98, 1987

HP90 Hennessy, J.L., Patterson, D.A.: **Computer Architecture: A Quantitative Approach**. Morgan Kaufmann Publishers, 1990

HR83 Härder, T., Reuter, A.*: Principles of Transaction-Oriented Database Recovery.* **ACM Computing Surveys** 15 (4), 287-317, 1983

HR85 Härder, T., Rahm, E.: *Quantitative Analyse eines Synchronisationsprotokolls für DB-Sharing.* **Proc. 3. GI/NTG-Tagung Messung, Modellierung und Bewertung von Rechensystemen***,* Informatik-Fachberichte 110, Springer-Verlag , 186-201, 1985

HR86 Härder, T., Rahm, E.: *Mehrrechner-Datenbanksysteme für Transaktionssysteme hoher Leistungsfähigkeit*. **Informationstechnik** *28* (4), 214-225, 1986

HR88 Hevner, A.R., Rao, A.: *Distributed Data Allocation Strategies*. **Advances in Computers** 27, 121-155, 1988

HR92 Härder, T., Rahm, E.: *Zugriffspad-Unterstützung zur Sicherung der Relationalen Invarianten*. ZRI-Bericht 2/92, Univ. Kaiserslautern, FB Informatik, 1992

HR93 Härder, T., Rothermel, K.: *Concurrency Control Issues in Nested Transactions.* **VLDB-Journal**, 1993

IBM94 *Offen, parallel und wirtschaftlich*. **IBM Nachrichten** 44, Heft 317, 1994

IK91 Ioannidis, Y., Kang, Y.: *Left-deep vs. Bushy Trees: an Analysis of Strategy Spaces and its Implications for Query Optimization.* **Proc. ACM SIGMOD Conf.**, 1991

Je93 Jenz, D.E.: *Datenbanken in Netzwerken - die Komplexität beherrschen.* **Handbuch der modernen Datenverarbeitung**, Heft 174, Forkel-Verlag, 42-57, 1993

Ji88 Jiang, B.: *Deadlock Detection is Really Cheap*. **ACM Sigmod Record** 17 (2), 2-13, 1988

JK84 Jarke, M., Koch, J.: *Query Optimization in Database Systems.* **ACM Computing Surveys** 16 (2), 111-152, 1984

Jo91 Joshi, A.M.: *Adaptive Locking Strategies in a Multi-node Data Sharing Environment*. **Proc. 17th Int. Conf. on Very Large Data Bases**, 181-191, 1991

JTK89 Jenq, B.P., Twichell, B., Keller, T.: Locking Performance in a Shared Nothing Parallel Database Machine. **IEEE Trans. on Knowledge and Data Engineering** 1 (4), 1989

Ka78 Katzman, J.A.: *A Fault-Tolerant Computing System.* **Proc. Hawaii Int. Conf. of System Sciences**, 1978

KGP89 Katz, R.H., Gibson, G.A., Patterson, D.A.*: Disk System Architectures for High Performance Computing.* **Proc. of the IEEE** 77 (12), 1842-1858, 1989

KHGP91 King, R., Halim, N., Garcia-Molina, H., Polyzois, C.A.: *Management of a Remote Backup Copy for Disaster Recovery*. **ACM Trans. on Database Systems**, 16 (2), 338-368, 1991

Ki83 Kitsuregawa, M. et al.: *Application of Hash to Data Base Machine and its Architecture.* **New Generation Computing**, No. 1, 62-74, 1983

Ki84 Kim, W.: *Highly Available Systems for Database Applications.* **ACM Computing Surveys** 16 (1), 71–98, 1984

Ki93 Kim, W.*: Object-Oriented Database Systems: Promises, Reality and Future.* **Proc. 19th Int. Conf. on Very Large Data Bases**, 676-687, 1993

KLS86 Kronenberg, N.P., Levy, H.M., and Strecker, W.D.: *VAX Clusters: a Closely Coupled Distributed System*. **ACM Trans. on Computer Syst.** 4 (2), 130-146, 1986

Kn87 Knapp, E.: *Deadlock Detection in Distributed Databases.* **ACM Computing Surveys** 19 (4), 303-328, 1987

KP85 Kiessling, W., Pfeiffer, H.: *A Comprehensive Analysis of Concurrency Control Performance for Centralized Databases.* **Proc. 4th Int. Workshop on Database Machines**, Springer-Verlag, 1985

KP92 Keim, D.A., Prawirohardjo, E.S.: **Datenbankmaschinen**. BI-Wissenschaftsverlag, Reihe Informatik Band 86, 1992

KR81 Kung, H.T., Robinson, J.T.: *On Optimistic Methods for Concurrency Control.* **ACM Trans. on Database Systems** 6 (2), 213-226, 1981

KS91 Kim, W., Seo, J.: *Classifying Schematic and Data Heterogeneity in Multidatabase Systems.* **IEEE Computer** 24 (12), Dec. 1991, 12-18

KTN92 Kitsuregawa, M., Tsudaka, S., Nakano, M.: *Parallel GRACE Hash Join on Shared-Everything Multiprocessor: Implementation and Performance Evaluation on Symmetry S81.* **Proc. 8th IEEE Data Engineering Conf**., 256-264,1992

La78 Lamport, L.: *Time, Clocks, and the Ordering of Events in a Distributed System.* **Comm. ACM** 21 (7), 558-565, 1978

La91 Lamersdorf, W.: *Remote Database Access: Kommunikationsunterstützung für Fernzugriff auf Datenbanken*. **Informatik-Spektrum (Das aktuelle Schlagwort)** 14 (3), 161-162, 1991

LARS92 Lomet, D., Anderson, R., Rengarajan, T.K., Spiro, P.: *How the Rdb/VMS Data Sharing System Became Fast*. DEC Technical Report CRL 92/4, 1992

Le91 Leslie, H.: *Optimizing Parallel Query Plans and Execution*. **Proc. IEEE Spring CompCon Conf.**, 105-109, 1991

Li81 Lindsay, B.: *Object Naming and Catalog Management for a Distributed Database Manager.* **Proc. 2nd IEEE Int. Conf. on Distributed Computing Systems**, 31-40, 1981

Li86 Lindsay, B. et al.: *A Snapshot Differential Refresh Algorithm*. **Proc. ACM SIGMOD Conf.**, 53-60, 1986

Li93 Linder, B.: *Oracle Parallel RDBMS on Massively Parallel Systems*. **Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS-93)**, 67-68, 1993

LKK93 Lockemann, P.C., Krüger, G., Krumm, H.: **Telekommunikation und Datenhaltung**. Hanser-Verlag, 1993

LMR90 Litwin, W., Mark, L., Roussopoulos, N.: *Interoperability of Multiple Autonomous Databases*. **ACM Computing Surveys** 22 (3), 267-293, 1990

LNS90 Lipton, R.J., Naughton, J.F., Schneider, D.A.: *Practical Selectivity Estimation through Adaptive Sampling*. **Proc. ACM SIGMOD Conf.**, 1-11, 1990

Lo90 Lomet, D.: *Recovery for Shared Disk Systems using Multiple Redo Logs*. Technical Report CRL 90/4, DEC Cambridge Research Lab., Cambridge, MA, 1990.

LS87 Lockemann, P.C., Schmidt, J.W. (Hrsg.): **Datenbank-Handbuch**. Springer-Verlag, 1987

LST91 Lu, H., Shan, M., Tan, K.: *Optimization of Multi-Way Join Queries for Parallel Execution*. **Proc. 17th Int. Conf. on Very Large Data Bases**, 549-560,1991

LVZ93 Lanzelotte, R., Valduriez, P., Zait, M.: *On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces.* **Proc. 19th Int. Conf. on Very Large Data Bases**, 493-504, 1993

LWL89 Lai, M., Wilkinson, K., Lanin, V.: *On Distributing JASMIN's Optimistic Multiversioning Page Manager.* **IEEE Trans. on Software Engineering** 15 (6), 696-704, 1989

Ly90 Lyon, J.: *Tandem's Remote Data Facility*. **Proc. IEEE Spring CompCon Conf.**, 562-567, 1990

LY89 Lorie, R.A., Young, H.C.: *A Low Communication Sort Algorithm for a Parallel Database Machine*. **Proc. 15th Int. Conf. on Very Large Data Bases**, 125-134, 1989

Ma93 Marek, R.: *Ein Kostenmodell der parallelen Anfragebearbeitung in Shared-Nothing-Datenbanksystemen*. ZRI-Bericht 3/93, Univ. Kaiserslautern, 1993

Me88 Meyer-Wegener, K.: **Transaktionssysteme**. Teubner-Verlag, 1988

ME92 Mishra, P., Eich, M.H.: *Join Processing in Relational Databases*. **ACM Computing Surveys** 24 (1), 63-113, 1992

ML86 Mackert, L.F., Lohman, G.M.: *R* Optimizer Validation and Performance Evaluation for Distributed Queries.* **Proc. 12th Int. Conf. on Very Large Data Bases**, 149-159,1986

MLO86 Mohan, C., Lindsay, B., Obermarck, R.: *Transaction Management in the R* Distributed Database Management System*. **ACM Trans. on Database Systems** 11 (4), 378-396, 1986

Mi94 Mitschang, B.: *Anfrageverarbeitung in Datenbanksystemen: Entwurfs- und Implementierungskonzepte.* Habilitationsschrift, FB Informatik, Univ. Kaiserslautern, 1994

MN91 Mohan, C., Narang, I.: *Recovery and Coherency-control Protocols for Fast Intersystem Page Transfer and Fine-granularity Locking in a Shared Disks Transaction Environment.* **Proc. 17th Int. Conf. on Very Large Data Bases**,193-207, 1991

MN92b Mohan, C., Narang, I.: *Data Base Recovery in Shared Disks and Client-Server Architectures.* **Proc. 12th Int. Conf. on Distributed Computing Systems**, IEEE Computer Society Press, 1992

MNS91 Mohan, C., Narang, I., Silen, S.: *Solutions to Hot Spot Problems in a Shared Disks Transaction Environment*. **Proc. 4th Int. Workshop on High Performance Transaction Systems,** Asilomar, CA, 1991

Mo84 Mohan, C.: *Current and Future Trends in Distributed Database Management.* **Proc. NYU Symp. "New Directions for Database Systems"**, 1984 (ebenso: IBM Research Report RJ 4240)

Mo85 Moss, J.E.B.: **Nested Transactions: An Approach to Reliable Distributed Computing.** MIT Press, 1985

Mo92a Mohan, C. et al.: *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*. **ACM Trans. on Database Systems** 17 (1), 94-162, 1992

Mo92b Mohan, C.: *Less Optimism about Optimistic Concurrency Control*. **Proc. 2nd Workshop on Research Issues on Data Engineering (RIDE-2)**, Tempe, AZ, IEEE Computer Society Press, 199-204, 1992

MR92 Marek, R., Rahm, E.: *Performance Evaluation of Parallel Transaction Processing in Shared Nothing Database Systems*. **Proc. 4th Int. PARLE Conf.** (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science 605, 295-310, 1992.

MS92 Melton, J., Simon, A.R.: **Understanding the New SQL: A Complete Guide**. Morgan Kaufmann, 1992

MTO93 Mohan, C., Treiber, K., Obermarck, R.: *Algorithms for the Management of Remote Backup Data Bases for Disaster Recovery*. **Proc. 9th IEEE Data Engineering Conf**., 511-518, 1993

Ne86 Neches, P.M.: *The Anatomy of a Database Computer - Revisited*. **Proc. IEEE Spring CompCon Conf.**, 374-377, 1986

NW87 Noe, J.D., Wagner, D.B.: *Measured Performance of Time Interval Concurrency Control Techniques*. **Proc. 13th Int. Conf. on Very Large Data Bases**, 359-367, 1987

Ob82 Obermarck, R.: *Deadlock Detection for All Resource Classes.* **ACM Trans. on Database Systems** 7 (2), 187-208, 1982

ON86 O'Neil, P.E.: *The Escrow Transactional Method*. **ACM Trans. Database Systems** 11 (4), 405-430, 1986

Or93 *Oracle for Massively Parallel Systems - Technology Overview*. Oracle Corporation, part number A10061, 1993

ÖV91 Özsu, M.T., Valduriez, P.: **Principles of Distributed Database Systems**. Prentice Hall, 1991

Pa91 Pappe, S.: **Datenbankzugriff in offenen Rechnernetzen**. Springer-Verlag, Reihe "Informationstechnik und Datenverarbeitung", 1991

PCL93 Pang, H., Carey, M.J., Livny, M.: *Partially Preemptible Hash Joins.* **Proc. ACM SIGMOD Conf.**, 59-68, 1993

Pe87 Peinl, P.: **Synchronisation in zentralisierten Datenbanksystemen**. Springer-Verlag, Informatik-Fachberichte 161, 1987

Pe93 Petkovic, D.: **INFORMIX 4.0/5.0**. Addison-Wesley, 1993

PGK88 Patterson, D.A., Gibson, G., Katz, R.H.*: A Case for Redundant Arrays of Inexpensive Disks (RAID).* **Proc. ACM SIGMOD Conf.,** 109-116, 1988

Pi90 Pirahesh, H. et al.: *Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches*. **Proc. 2nd Int.Symposium on Databases in Parallel and Distributed Systems**, IEEE Computer Society Press, 1990

Ra86a Rahm, E.: *Nah gekoppelte Rechnerarchitekturen für ein DB-Sharing-System*. **Proc. 9. NTG/GI-Fachtagung "Architektur und Betrieb von Rechensystemen**", Stuttgart, VDE-Verlag, NTG-Fachberichte 92, 166-180, 1986

Ra86b Rahm, E.: *Primary Copy Synchronization for DB-Sharing*. **Information Systems** 11 (4), 275-286, 1986

Ra87 Rahm, E.: *Design of Optimistic Methods for Concurrency Control in Database Sharing Systems*. **Proc. 7th Int. Conf. on Distributed Computing Systems**, IEEE Computer Society Press, 154-161, 1987

Ra88a Rahm, E.: *Optimistische Synchronisationskonzepte in zentralisierten und verteilten Datenbanksystemen*. **Informationstechnik** 30 (1), 28-47, 1988

Ra88b Rahm, E.: **Synchronisation in Mehrrechner-Datenbanksystemen***.* Springer-Verlag, Informatik-Fachberichte 186, 1988

Ra89 Rahm, E.: *Der Database-Sharing-Ansatz zur Realisierung von Hochleistungs-Transaktionssystemen*. **Informatik-Spektrum** 12 (2), 65-81, 1989

Ra91a Rahm, E.: *Recovery Concepts for Data Sharing Systems.* **Proc. 21st Int. Symp. on Fault-Tolerant Computing**, IEEE Computer Society Press, 368-375, 1991.

Ra91b Rahm, E.: *Use of Global Extended Memory for Distributed Transaction Processing.* **Proc. 4th Int. Workshop on High Performance Transaction Systems**, Asilomar, Sep. 1991.

Ra92a Rahm, E.: *Performance Evaluation of Extended Storage Architectures for Transaction Processing.* **Proc. ACM SIGMOD Conf.**, 308-317, 1992.

Ra92b Rahm, E.*: A Framework for Workload Allocation in Distributed Transaction Systems.* **The Journal of Systems and Software** 18 (3), 171-190, 1992.

Ra93a Rahm, E.: **Hochleistungs-Transaktionssysteme**. Vieweg-Verlag (Reihe "Datenbanksysteme"), 1993.

Ra93b Rahm, E.: *Kohärenzkontrolle in verteilten Systemen*. **GI-Workshop "Datenbanken und Betriebssysteme: Architekturen, Schnittstellen, Interoperabilität"**, Heidelberg, April 1993, in: Datenbank-Rundbrief Nr. 11 der GI-Fachgruppe Datenbanken, Mai 1993.

Ra93c Rahm, E.: *Evaluation of Closely Coupled Systems for High Performance Database Processing.* **Proc. 13th Int. Conf. on Distributed Computing Systems**, IEEE Computer Society Press, 301-310, 1993.

Ra93d Rahm, E.*: Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems.* **ACM Trans. on Database Systems** 12 (2), 333-377, 1993.

Ra93e Rahm, E.: *Parallel Query Processing in Shared Disk Database Systems.* **Proc. 5th Int. Workshop on High Performance Transaction Systems**, Asilomar, Sep. 1993 (Kurzfassung in: ACM SIGMOD Record 22 (4), 1993).

Raz92 Raz, Y.: *The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment.* **Proc. 18th Int. Conf. on Very Large Data Bases**, 292-312, 1992

Raz93 Raz, Y.: *Extended Commitment Ordering, or Guaranteeing Global Serializability by Applying Commitment Order Selectively to Global Transactions.* **Proc 12th ACM Symp. on Principles of Database Systems (PODS),**1993

Re81 Reuter, A.: **Fehlerbehandlung in Datenbanksystemen**. Carl Hanser, 1981

Re86 Reuter, A.: *Load Control and Load Balancing in a Shared Database Management System*. **Proc. 2nd IEEE Data Engineering Conf**., 188-197,1986

Re87 Reuter, A.: *Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen*. In [LS87], 337-479

Re92 Reuter, A.: *Grenzen der Parallelität*. **Informationstechnik** 34 (1), 62-74, 1992

RM93 Rahm, E., Marek, R.: *Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems*, **Proc. 19th Int. Conf. on Very Large Data Bases,** 182-193, 1993.

RM95 Rahm, E., Marek, R.: *Dynamic Multi-Resource Load Balancing in Parallel Database Systems.* **Proc. 21th Int. Conf. on Very Large Data Bases,** 395-406, 1995.

RMW93 Reiner, D., Miller, J., Wheat, D.: *The Kendall Square Query Decomposer.* **Proc. IEEE Spring Comp Con Conf.**, 300-302, 1993

Ro85 Robinson, J.T.: *A Fast General-Purpose Hardware Synchronization Mechanism*. **Proc. ACM SIGMOD Conf.,** 122-130, 1985

RS95 Rahm, E., Stöhr, T.: *Analysis of Parallel Scan Processing in Shared Disk Database Systems.* **Proc. EURO-PAR 95**, Springer-Verlag, Lecture Notes in Computer Science 966, 485-500, 1995.

RSL78 Rosenkrantz,D.J., Stearns,R., Lewis,P.: *System Level Concurrency Control for Distributed Database Systems*. **ACM Trans. on Database Systems** 3 (2), 1978

RSW89 Rengarajan, T.K., Spiro, P.M., Wright, W.A.: *High Availability Mechanisms of VAX DBMS Software*. **Digital Technical Journal**, No. 8, 88-98, Feb. 1989

Sa90 Salzberg, B. et al.: *FastSort: A Distributed Single-Input Single-Output External Sort*. **Proc. ACM SIGMOD Conf**., 94-101, 1990

SB90 Simonsen, D., Benningfield, D.: *INGRES Gateways: Transparent Heterogeneous SQL Access*. **IEEE Data Engineering** 13 (2), 40-45, 1990

SBCM93 Samaras, G., Britton, K., Citron, A., Mohan, C.: *Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment*. **Proc. 9th IEEE Data Engineering Conf**., 520-529, 1993

Sc87 Scrutchin Jr., T.W.*: TPF: Performance, Capacity, Availabilty.* **Proc. IEEE Spring CompCon Conf.**, 158-160, 1987

Sc92 Schill, A.: *Remote Procedure Call: Fortgeschrittene Konzepte und Systeme - ein Überblick*, **Informatik-Spektrum** 15, 79-87 u. 145-155, 1992

Sc93 Schill, A.: *Das OSF Distributed Computing Environment.* Springer-Verlag, 1993

SD89 Schneider, D.A., DeWitt, D.J.: *A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment*. **Proc. ACM SIGMOD Conf**., 110-121, 1989

SD90 Schneider, D.A., DeWitt, D.J.: *Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines.* **Proc. 16th Int. Conf. on Very Large Data Bases**, 469-480, 1990

Se79 Selinger, P. et al.: *Access Path Selection in a Relational Database System.* **Proc. ACM SIGMOD Conf**., 1979

Se84 Sekino, A. et al.: The *DCS - a New Approach to Multisystem Data Sharing*. **Proc. National Comp. Conf**., 59-68, 1984

Se93 Seifert, M.: *Konkurrenz der Architektur-Standards für Offene Systeme.* **Handbuch der modernen Datenverarbeitung**, Heft 172, Forkel-Verlag, 35-50, 1993

Sh86 Shoens, K.: *Data Sharing vs. Partitioning for Capacity and Availability*. **IEEE Database Engineering** 9 (1), 10-16,1986

Sh93 Sherman, M.: *Distributed Transaction Processing with Encina.* **Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS-93)**, 268-269, 1993

Sk81 Skeen, D.: *Non-blocking Commit Protocols*. **Proc. ACM SIGMOD Conf**., 133-142, 1981

Sk92 Skelton, C.J. et al.: *EDS: A Parallel Computer System for Advanced Information Processing*. **Proc. 4th Int. PARLE Conf.** (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science 605, 3-18, 1992

SKPO88 Stonebraker, M., Katz, R., Patterson, D., Ousterhout, J.*: The Design of XPRS.* **Proc. 14th Int. Conf. on Very Large Data Bases**, 318-330, 1988

SL76 Severance, D.G., Lohman, G.M.: Differential Files: Their Application to the Maintenance of Large databases. **ACM Trans. on Database Systems** 1 (3), 256-267, 1976

SL90 Sheth, A.P., Larson, J.A.: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*. **ACM Computing Surveys** 22 (3), 183-236, 1990

Sm93 Smerik, R.: *TOP END: Distributed Transaction Processing for Open Systems.* **Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS-93)**, 270-271, 1993

SS91 Speer, T.G., Storm, M.W.: *Digital's Transaction Processing Monitors*. **Digital Technical Journal** 3 (1), Winter 1991, 18-32

St79 Stonebraker, M.: *Concurrency Control and Consistency of Multiple Copies in Distributed Ingres*. **IEEE Trans. on Software Engineering** 5 (3), 188-194, 1979

St80 Stonebraker, M.*: The Argument Against CODASYL.* In **Distributed Data Bases** (Eds.: I.W. Draffan, F. Poole), Cambridge Univ. Press, 361-370, 1980

St86 Stonebraker, M.*: The Case for Shared Nothing.* **IEEE Database Engineering** 9 (1), 4-9, 1986

St89 Stonebraker, M.*: The Distributed Database Decade*. **Datamation**, 38-39, Sep. 15, 1989

St90 Stenström, P.: *A Survey of Cache Coherence Schemes for Multiprocessors*. **IEEE Computer** 23 (6), 12-24, June 1990

ST87 Snaman Jr., W.E., Thiel, D.W.: *The VAX/VMS Distributed Lock Manager*. **Digital Technical Journal**, No. 5, 29-44, Sep. 1987

STY93 Shekita, E.J., Tan, K., Young, H.C.: *Multi-Join Optimization for Symmetric Multiprocessors*. **Proc. 19th Int. Conf. on Very Large Data Bases,** 479-492, 1993

Su88 Su, S.Y.W.: **Database Computers**. McGraw-Hill, 1988

Su92 *Supra Server*. Broschüre der Firma Cincom, 1992

SUW82 Strickland, J., Uhrowczik, P., Watts, V.: *IMS/VS: an Evolving System*. **IBM Systems Journal** 21 (4), 490-510, 1982

SW85 Steinbauer, D., Wedekind, H.: *Integritätsaspekte in Datenbanksystemen.* **Informatik-Spektrum** 8 (2), 60-68, 1985

SW91 Schek, H.-J., Weikum, G.: *Erweiterbarkeit, Kooperation, Föderation von Datenbanksystemen*. **Proc. 4. GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft"**, Springer-Verlag, Informatik-Fachberichte 270, 38-71, 1991

Sy94 *Sysplex Overview.* IBM Manual GC28-1208, 1994

Ta88 Tanenbaum, A.: **Computer Networks** (2nd ed.), Prentice-Hall, 1988

Ta89 The Tandem Database Group: *NonStop SQL, a Distributed, High-Performance, High-Availability Implementation of SQL*. **Proc. 2nd Int. Workshop on High Performance Transaction Systems**, Springer-Verlag, Lecture Notes in Computer Science 359, 60-103, 1989

Te83 *DBC/1012 Data Base Computer Concepts and Facilities.* Teradata, April 1983

TGGL82 Traiger, I.L., Gray, J., Galtieri, C.A., Lindsay, B.G.: *Transactions and Consistency in Distributed Database Systems.* **ACM Trans. on Database Systems** 7 (3), 323-342, 1982

Th79 Thomas, R.H.: *A Majority Consensus Approach to Concurrency Control for Multiple Copies Data Bases.* **ACM Trans. on Database Systems** 4 (2), 180-209, 1979

Th90 Thomas, G. et al.: *Heterogeneous Distributed Database Systems for Production Use*. **ACM Computing Surveys** 22 (3), 237-266, 1990

TR90 Thomasian, A., Rahm, E.: *A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking*, **Proc. 10th IEEE Int. Conf. on Distributed Computing Systems**, 294-301, 1990

Tr93 Trehan, V.: *ACMS: Digital's Open, Distributed TP Monitor.* **Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS-93)**, 264-265, 1993

Ul88 Ullman, J.D.: **Principles of Database and Knowledge-Base Systems**. Vol. I und II, Computer Science Press, 1988/1989

Va93a Valduriez, P.: *Parallel Database Systems: Open Problems and New Issues*. **Distr. and Parallel Databases** 1 (2), 137-165, 1993

Va93b Valduriez, P.: *Parallel Database Systems: The Case for Shared-Something*. **Proc. 9th Int. Conf. on Data Engineering**, 460-465, 1993

VG93 Vossen, G., Gross-Hardt, M.: **Grundlagen der Transaktionsverarbeitung**. Addison-Wesley, 1993

Wa83 Walter, B.: *Using Redundancy for Implementing Low-Cost Read-Only Transactions in a Distributed Database System*. **Proc. IEEE Infocom Conf.**, 153-159, 1983

Wa84 Wah, B.: *File Placement on Distributed Computer Systems*. **IEEE Computer** 17 (1), 23-32, Jan. 1984

WCK93 Witkowski, A., Carino, F., Kostamaa, P.: *NCR 3700 - The Next-Generation Industrial Database Computer.* **Proc. 19th Int. Conf. on Very Large Data Bases**, 230-243, 1993

WDJ91 Walton, C.B., Dale, A.G., Jenevein, R.M.: *A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins.* **Proc. 17th Int. Conf. on Very Large Data Bases**, 537-548, 1991

We87 Weihl, W.E.: *Distributed Version Management for Read-Only Actions.* **IEEE Trans. on Software Engineering** 13 (1), 55-64, 1987

We88 Weikum, G.: **Transaktionen in Datenbanksystemen**. Addison-Wesley, Bonn, 1988

WFA93 Wilschut, A.; Flokstra, J.; Apers, P.: *Parallelism in a Main-Memory DBMS: The Performance of PRISMA/DB*. **Proc. 18th Int. Conf. on Very Large Data Bases**, 521-532, 1992

Wi89 Wipfler, A.J.: **Distributed Processing in the CICS Environment**. McGraw Hill, 1989

Wi93 Wittmann, I.: *Die Transaktionssysteme Encina und CICS/6000.* **unix/mail** 11 (2), 102-107, 1993

WIH83 West, J.C., Isman, M.A., Hannaford, S.G.: *PERPOS Fault-tolerant Transaction Processing*. **Proc. 3rd Symposium on Reliability in Distributed Software and Database Systems**, IEEE Computer Society Press,189-194, 1983

WT91 Watson, P., Townsend, P.: *The EDS Parallel Relational Database System*. In: **Parallel Database Systems**, Springer-Verlag, Lecture Notes in Computer Science 503, 149-168, 1991

WZ93 Weikum, G., Zabback, P.: *I/O-Parallelität und Fehlertoleranz in Disk-Arrays.* **Informatik-Spektrum** 16, 133-142 und 206-214, 1993

YCDI87 Yu, P.S., Cornell, D.W., Dias, D.M., Iyer, B.R.: *Analysis of Affinity based Routing in Multi-system Data Sharing*. **Performance Evaluation** 7 (2), 87-109, 1987

YD94 Yu, P.S., Dan, A.: *Performance Evaluation of Transaction Processing Coupling Architectures for Handling System Dynamics.* **IEEE Trans. on Parallel and Distributed Systems** 5 (2), 139-153, 1994

Yu87 Yu , P.S. et al.: *On Coupling Multi-Systems through Data Sharing*. **Proc. of the IEEE**, 75 (5), 573-587, 1987

Ze89 Zeller, H.: *Parallelisierung von Anfragen auf komplexen Objekten durch Hash Joins*. **Proc. 3. GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft"**, Springer-Verlag, Informatik-Fachberichte 204, 361-367, 1989

Ze90 Zeller, H.: *Parallel Query Execution in NonStop SQL*. **Proc. IEEE Spring CompCon Conf.**, 484-487, 1990

ZG90 Zeller, H., Gray, J.: *An Adaptive Hash Join Algorithm for Multiuser Environments*. **Proc. 16th Int. Conf. on Very Large Data Bases**, 186-197, 1990

Zi93 Ziekursch, W.: *Original oder Kopie?* **Relationen** (ASK/Ingres-Magazin) 4/93, 7-10, 1993

ZZB93 Ziane, M., Zait, M., Borla-Salamet, P.: *Parallel Query Processing in DBS3.* **Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS-93)**, 93-102, 1993