Informatik Forsch. Entw. (1996) 11: 203–212

Informatik
Forschung und Entwicklung

Prinzipien der Replikationskontrolle in verteilten Datenbanksystemen*

T. Beuter, P. Dadam

Abteilung Datenbanken und Informationssysteme, Fakultät für Informatik, Universität Ulm, D-89069 Ulm
(e-mail: {beuter,dadam}@informatik.uni-ulm.de, WWW: http://www.informatik.uni-ulm.de/abt/dbis)

Eingegangen am 6. Februar 1996 / Angenommen am 12. September 1996

Zusammenfassung. Durch Datenreplikation können prinzipiell schnellere Zugriffszeiten und eine beliebig hohe Fehlertoleranz in verteilten Datenbanksystemen erreicht werden. Anderseits erhöht Replikation die Gefahr von Inkonsistenzen und den Aufwand von Änderungsoperationen. Zur Lösung dieses Zielkonflikts wurden in der Literatur viele unterschiedliche *Replikationsverfahren* vorgeschlagen. Dieser Überblicksartikel beschreibt die den einzelnen Verfahren zugrundeliegenden Prinzipien zur Replikationskontrolle. Dazu werden die durch den Kopieneinsatz resultierenden Probleme erläutert, daraus Kriterien zur anschließenden Klassifikation abgeleitet und danach ausgewählte Replikationsverfahren näher vorgestellt.

Schlüsselwörter: verteiltes (Datenbank-) System, Replikation, Fehlertoleranz, Netzpartitionierung, 1-Kopie-Serialisierbarkeit, Datenkonsistenz

Abstract. The replication of data promises better performance and a high fault tolerance in distributed systems. However, replication increases the risk of data inconsistencies and the cost of updates. In order to find an appropriate solution for this trade-off many methods for the management of replicated data have been proposed in the literature. In this paper the underlying principles of these replication methods are critically surveyed. The problems caused by data replication are described, a classification schema is derived, and some important replication methods are critically discussed in more detail.

Key words: distributed (database) system, replication, fault tolerance, partitioned network, 1-copy serialisibility, data consistency

CR Subject Classification: C.4, C.2.4, D.4.5, H.2.4

1 Einleitung und Motivation

In den letzten Jahren kann ein immer stärkerer Trend zu verteilten Datenbanksystemen beobachtet werden. Diese Tendenz läßt sich sowohl bei klassischen Datenbankanwendungen (weg vom Mainframe, hin zu verteilten Client-Server-Applikationen) als auch in vielen neuen Anwendungsgebieten (z. B.: mobile Computing, Concurrent Engineering, Workflow-Management, Data Warehousing) feststellen, bei denen die Verteilung schon anwendungsbedingt vorgegeben ist.

Eine Voraussetzung für die bessere Fehlertoleranz und den höheren Durchsatz eines verteilten Systems stellt vielfach die Replikation der Daten dar: Der Wert eines (Daten-) Objekts wird hierbei auf mehrere (Rechner-) Knoten des verteilten Datenbanksystems gespeichert. Durch einen dann realisierbaren Zugriff auf die lokal vorhandene bzw. auf eine schnell erreichbare Kopie wird ein Weiterarbeiten auch bei Ausfall datenhaltender Knoten ermöglicht sowie der Durchsatz erhöht. Replikation ist deshalb besonders bei datenintensiven Anwendungen (z. B. Multi-Media) und bei verteilten Systemen notwendig, die über ein unsicheres Kommunikationsnetzwerk mit geringer Bandbreite verfügen (z. B. mobile Computing).

Allerdings führt Replikation bei der Veränderung eines Objekts zu höherem Kommunikations- und Verwaltungsaufwand, da das Objekt in der Regel auf mehreren Knoten aktualisiert werden muß. Dieser Aufwand erhöht sich noch, wenn gefordert wird, daß sich aus der Sicht der Anwendung ein repliziertes Datenbanksystem wie ein nichtrepliziertes System verhalten soll (sog. *1-Kopie-Äquivalenz (1-copy equivalence)* [4]). Das verteilte System muß dann darauf achten, daß die Kopien eines replizierten Objekts *wechselseitig konsistent (mutual consistent)* [14] sind. Diese Aufgabe wird durch das Auftreten von Rechner- und Netzausfällen erschwert. Besonders problematisch sind in diesem Zusammenhang *Partitionierungen*, bei denen eine verteilte Datenbank so in disjunkte Knotenmengen (*Partitionen*) aufgeteilt wird, daß eine Kommunikation über Partitionsgrenzen hinaus nicht mehr möglich ist. Dadurch kann aus dem „Nichterreichen“ eines Knotens nicht mehr auf dessen Ausfall und damit dessen Inaktivität geschlossen werden: Er kann sich auch funktionsfähig in einer anderen Partition befinden. Es besteht dann die Gefahr, daß Knoten aus verschiedenen Partitionen ihre erreichbaren Kopien unkoordiniert verändern und dadurch das verteilte Datenbanksystem in einen inkorrekten (= *inkonsistenten*) Zustand

* Diese Arbeit entstand im Rahmen eines von der Daimler-Benz-Forschung Ulm geförderten Forschungsprojekts

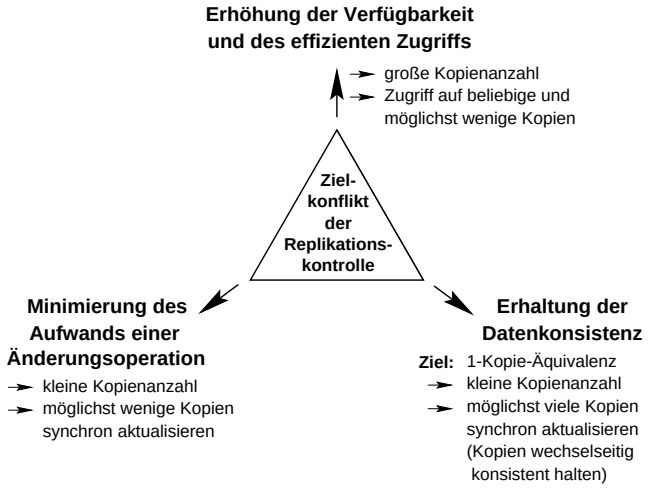


Abbildung 1. Zielkonflikt der Replikationskontrolle

überführen. Hier existiert ein Zielkonflikt zwischen maximaler Verfügbarkeit und der Korrektheit des Gesamtsystems.

Es muß deshalb ein geeigneter Kompromiß gefunden werden, der es ermöglicht, die Vorteile der Datenreplikation (höhere Verfügbarkeit, effizienterer Zugriff) zu nutzen, und dabei ihre Nachteile (größerer Änderungsaufwand, Gefahr von Dateninkonsistenzen) möglichst zu vermeiden. In der Abbildung 1 werden dieser Zielkonflikt und die daraus resultierenden Folgen für die Verwaltung von Datenreplikaten graphisch dargestellt.

Zur Lösung dieses Zielkonflikts werden in der Literatur viele verschiedene *Replikationsverfahren* vorgeschlagen. Ziel dieses Übersichtsartikels ist es, durch eine kritische Betrachtung der verschiedenen Grundprinzipien zur Replikationskontrolle, eine Klassifikation für Replikationsverfahren zu erstellen, um damit dem interessierten Leser den Einstieg in dieses Gebiet zu erleichtern.

Diese Grundprinzipien werden in Kapitel 2 anhand der verschiedenen Aufgaben der Replikationsverfahren erarbeitet, ihre jeweiligen Vor- und Nachteile kritisch beleuchtet und dabei Kriterien zur Klassifikation der Replikationsmethoden abgeleitet. Im darauffolgenden Kapitel werden einige ausgewählte Verfahren detaillierter beschrieben sowie ihre spezifischen Vor- und Nachteile diskutiert. Kapitel 4 bildet mit einigen Bemerkungen über verschiedene Kombinationsmöglichkeiten von Replikationsstrategien sowie mit Hinweisen auf kommerzielle Datenbanksysteme, die Replikationskomponenten enthalten, den Schluß dieses Papiers.

2 Klassifikation von Replikationsverfahren

Die Suche nach einem geeigneten Kompromiß für den in Abbildung 1 skizzierten Zielkonflikt führte in der Literatur zu recht unterschiedlichen Vorschlägen zur Replikationskontrolle. Diese Vorschläge lassen sich am besten anhand der folgenden Punkte klassifizieren:

1. Sicherung der Datenkonsistenz (Abschnitt 2.1)
2. Kopien-Update-Strategie (Abschnitt 2.2)
3. Behandlung von Netzpartitionierungen (Abschnitt 2.3)

2.1 Sicherung der Datenkonsistenz

Wie bei Datenbanksystemen üblich, wird auch bei replizierten Datenbanken die Datenkonsistenz häufig über die *Serialisierbarkeit* beschrieben. Dieser Serialisierbarkeitsbegriff muß zur Einhaltung der 1-Kopie-Äquivalenz erweitert werden:

1-Kopie-serialisierbare Schedule:
Eine Schedule S einer replizierten Datenbank heißt *1-Kopie-serialisierbar*, wenn es mindestens eine serielle Ausführung der Transaktionen dieser Schedule auf einer *nicht* replizierten Datenbank gibt, welche, angewandt auf denselben Anfangszustand, die gleiche Ausgabe sowie den gleichen Datenbankendzustand erzeugt wie S auf der replizierten Datenbank [4, 14].

Die Einhaltung des verwendeten Korrektheitskriteriums läßt sich auf zwei Arten realisieren:

Bei *syntaktischen Verfahren*, zu denen der Großteil der Replikationsverfahren gehört [30, 27, 33, 20, 24, 1, 3, 17], wird die Korrektheit eines Zugriffs einzig über die Reihenfolge der zugreifenden Transaktionen, also über die Folge ihrer Lese- und Schreiboperationen, bestimmt.

Dagegen nützen *semantische Verfahren* die Semantik der Transaktionen, wie z. B. Kommutativität der Operationen [6, 22, 26], bzw. die Semantik der Datenbank [18] aus, um die Anzahl zulässiger Ausführungsreihenfolgen (Schedules) zu erhöhen. Bei den semantischen Ansätzen gibt es Verfahren, bei denen die Korrektheit nur über semantische Integritätsbedingungen der Datenbank definiert wird ([18], siehe Abschnitt 3.2.1). Daneben existieren aber auch Verfahren, die semantisches Wissen zusätzlich zur Ausführungsreihenfolge verwenden, um die Verfügbarkeit oder den Parallelitätsgrad der Datenbank zu erhöhen ([6], [26] siehe Abschnitt 3.2.2).

Im allgemeinen sind semantische Verfahren nicht so universiell einsetzbar wie syntaktische Ansätze, da nicht jede Anwendung die für den Einsatz des jeweiligen semantischen Verfahrens notwendigen Voraussetzungen erfüllt (z. B. nur kommutative Datenmanipulationsoperationen). Anderseits sind semantische Verfahren gerade bei verteilten Systemen besonders attraktiv, da aufgrund der semantisch höheren Operationen der erforderliche Kommunikations- und Synchronisationsaufwand reduziert werden kann (siehe Abschnitt 2.2).

Der jeweils zugrundegelegte Korrektheitsbegriff muß sowohl beim *Kopien-Update* (Abschnitt 2.2) als auch im *Fehlerfall* (Abschnitt 2.3) eingehalten werden.

2.2 Kopien-Update-Strategie

Bei replizierten Datenbanksystemen muß festgelegt werden, welche und wieviele Kopien eines Objekts *synchron* mit dem Commit einer Änderungstransaktion und welche Kopien *asynchron* nach Beendigung der Transaktion aktualisiert werden. Aufgrund der geforderten Dauerhaftigkeit der Änderungen einer mit Commit beendeten Transaktion muß mindestens eine Kopie synchron aktualisiert werden. Je mehr Kopien synchron geändert werden, desto mehr Kopien sind wechselseitig konsistent, aber desto aufwendiger und feh-

leranfälliger wird ein Transaktions-Commit.[1] Asynchrone Änderungen blockieren dagegen das Commit einer Transaktion nicht. Dadurch können Knoten- und Netzausfälle wie asynchrone Änderungsoperationen mit besonders langer zeitlicher Verzögerung behandelt werden.

Je nach Update-Strategie variiert die Zahl der *synchronen* Änderungen zwischen nur einer Kopie (manche semantische bzw. absolutistische Verfahren) und (möglichst) allen Kopien (Read-One-Copy-basierte Verfahren).

Um Datenbank-Inkonsistenzen zu vermeiden, muß eine geeignete Kopien-Update-Strategie die folgenden Punkte sicherstellen:

Punkt 1. Konkurrierende Updates auf verschiedene Kopien desselben Objekts müssen koordiniert werden (*kopienübergreifende Synchronisation*). Die Art der kopienübergreifenden Synchronisation bestimmt im wesentlichen die Zahl der notwendigen synchronen Updates.
Punkt 2. Ein Update muß stets auf einer aktuellen Kopie basieren.

Bei der nun folgenden Beschreibung der verschiedenen Kopien-Update-Strategien wird auch immer auf die jeweilige Umsetzungen dieser Punkte eingegangen. Abschnitt 2.2.1 erläutert die grundsätzlichen Update-Strategien für syntaktische Verfahren. Abschnitt 2.2.2 zeigt, welche Vorteile semantisches Wissen bei der Kopien-Update-Strategie bringen kann.

2.2.1 Syntaktische Kopien-Update-Strategien

Bezüglich der Kopien-Update-Strategie wurden für syntaktische Verfahren die folgenden zwei grundsätzlichen Ansätze, *absolutistische* und *Abstimmungsverfahren* vorgeschlagen:

2.2.1.1 Absolutistische Verfahren. Beim absolutistischen Ansatz realisiert eine ausgezeichnete Kopie (*primary copy* [30], Besitzer des *tokens* [27]) die kopienübergreifende Synchronisation (Punkt 1). Will eine Transaktion ein Objekt ändern, so benötigt sie die Zustimmung dieser Kopie. In der Regel ist diese Kopie auch die einzige, die synchron aktualisiert wird. Da jedoch jeder Zugriff über diese ausgezeichnete Kopie erfolgt, basiert jedes Update auf dem aktuellen Objektwert (Punkt 2).

Lesetransaktionen dürfen, da sie den Datenbankzustand nicht verändern, dagegen meist ohne Zugriff auf die Primärkopie von einer beliebigen und idealerweise lokal vorhandenen Kopie lesen. Hierbei besteht jedoch die Gefahr, daß die Kopie, auf die zugegriffen wird, aufgrund der asynchronen Änderungen nicht den aktuellen Wert des Datenobjekts widerspiegelt (*running in the past* [3]). Greift eine Lesetransaktion auf mehrere verschiedene Objekte zu, so kann sie auch inkonsistente Daten sehen, wenn die Änderungen einer Schreibtransaktion auf nur einem Teil der gelesenen Objekte durchgeführt wurden. Absolutistische Ansätze garantieren in dieser Form also keine 1-Kopien-Äquivalenz für Lesetransaktionen. Ist dies für eine Lesetransaktion nicht tolerierbar, so muß auch zum Lesen auf die jeweilige ausgezeichnete Kopie zugegriffen werden (Lesetransaktion als Pseudo-Änderungstransaktion), wodurch der Vorteil des lokalen Lesens im wesentlichen verlorengeht.

Die Verfügbarkeit eines logischen Objekts hängt bei diesen Verfahren stark von der Verfügbarkeit der ausgezeichneten Kopie ab. Fällt diese aus, kann ein Stellvertreter die Synchronisation übernehmen. Dabei muß zur Vermeidung von Inkonsistenzen sichergestellt werden, daß zu jedem Zeitpunkt immer nur eine ausgezeichnete Kopie für jedes Objekt existiert. Kann dabei nicht zwischen Knotenausfällen und Netzwerkfehlern unterschieden werden, so ist die Bestimmung einer neuen ausgezeichneten Kopie aufwendig. Ein weiterer Nachteil dieser Ansätze besteht darin, daß jeder ändernde (und evtl. auch lesende) Zugriff über die ausgezeichnete Kopie des Objekts erfolgen muß. Bei vielen Zugriffen kann diese Kopie dann zum Engpaß werden.

Die Attraktivität absolutistischer Verfahren liegt vor allem in ihrer einfachen Realisierung. Im Prinzip kann jedes der bei nicht-replizierten Datenbanksystemen verwendete Synchronisationsverfahren (z. B. Sperrverfahren, optimistische Verfahren) eingesetzt werden.

2.2.1.2 Abstimmungsverfahren. Einen „demokratischeren“ Ansatz verfolgen die Abstimmungsverfahren [33, 20, 24, 1, 2], bei denen die kopienübergreifende Synchronisation der Zugriffe (Punkt 1) durch *Abstimmung (voting)* erfolgt. Dazu erhält jede Kopie eine bestimmte Anzahl an Stimmen (oft 1) zugeteilt. Der Zugriff einer Transaktion auf ein logisches Objekt wird nur dann erlaubt, wenn eine entscheidungsfähige Anzahl, ein sogenanntes *Quorum*, an Kopien zustimmt. Häufig wird dabei zwischen einem *Lesequorum* Q_R, das zum lesenden Zugriff erlangt werden muß, und einem *Schreibquorum* Q_W für den Schreibzugriff unterschieden. Werden bei der Wahl des Schreib- und Lesequorums die folgenden Überschneidungsregeln (vgl. z. B. [1]) eingehalten, so können Dateninkonsistenzen nicht mehr auftreten:

- Schreib/Schreib-Überschneidungsregel:
  $2 * Q_W > \sum$ über alle Stimmen
- Schreib/Lese-Überschneidungsregel:
  $Q_W + Q_R > \sum$ über alle Stimmen

Durch die Schreib/Schreib-Überschneidungsregel werden parallele Schreibzugriffe über die gemeinsame(n) Kopie(n) synchronisiert (Punkt 1). Werden außerdem mindestens Q_W Kopien beim Transaktions-Commit synchron aktualisiert, so ist sichergestellt, daß jedes Update auf dem *aktuellen Objektwert* basiert (Punkt 2). Zusammen mit der Schreib/Lese-Überschneidungsregel ist dann auch in jedem Lesequorum mindestens eine *aktuelle Kopie* enthalten.

Zur Bestimmung einer aktuellen Kopie werden entweder *Versionsnummern* oder *Zeitstempel* eingesetzt. Die Verwendung von Zeitstempeln hat den Vorteil, daß die Schreib/Schreib-Überschneidungsregel entfallen kann, da die Synchronisation der Schreibzugriffe über die Zeitstempelreihenfolge erfolgt [22].

Im Rahmen der Überschneidungsregeln ist die für eine Lese- bzw. Schreibquorum benötigte Stimmenzahl frei wählbar. Beim ursprünglichen Vorschlag, dem *Majority-Consensus-Verfahren* [33], wird für Lese- und Schreibquorum jeweils eine Stimmenmehrheit benötigt (Q_R = Q_W

[1] Beispielsweise zeigen Abschätzungen, daß die Deadlock-Wahrscheinlichkeit mit der vierten Potenz der Zahl an synchronen Updates wächst [21], Seite 429

$= N \div 2 + 1)$[2]. Wählt man dagegen eine asymmetrische Quorumseinteilung, so kann eine Zugriffsoperation (z. B. die in den meisten Anwendungen häufigere Leseoperation) auf Kosten der anderen optimiert werden.

Durch eine unterschiedliche Verteilung der Stimmen auf die Kopien (*weighted voting* [20]) können darüberhinaus auch einzelne Kopien bevorzugt werden. Damit kann bei gerader Kopienanzahl die Wahrscheinlichkeit, ein Quorum zu erreichen, erhöht werden: Wird beispielsweise ein Objekt auf vier Knoten repliziert, so benötigt man mindestens die Zustimmung von drei Knoten. Erhält jedoch ein (besonders ausfallsicherer) Knoten zwei Stimmen, so genügt schon die Zustimmung eines weiteren Knotens zur Erlangung des Quorums [7]. Kriterien für eine geeignete Wahl eines Quorums werden detailiert in [19] diskutiert.

Im Vergleich zu absolutistischen Verfahren sind Abstimmungsverfahren im fehlerfreien Fall aufwendiger, da zur Synchronisation mehr Nachrichten ausgetauscht und mehr Kopien synchron aktualisiert werden müssen. Dafür sind Abstimmungsverfahren nicht mehr von der Verfügbarkeit einzelner ausgezeichneter Kopien abhängig. Bei der Abstimmung kann der Ausfall einer Kopie jederzeit durch die Stimme(n) einer beliebigen anderen Kopie kompensiert werden, ohne daß zusätzliche Ersetzungsalgorithmen notwendig sind.[3]

Bei der Frage, wann welche Art der kopienübergreifenden Synchronisation vorzuziehen ist, muß sicherlich die Verfügbarkeit und Leistungsfähigkeit der einzelnen Komponenten des verteilten Systems berücksichtigt werden. Unterscheiden sich die einzelnen Komponenten in diesen Punkten stark (z. B. schneller Mainframe mit nahezu 100% Verfügbarkeit, kleinere Abteilungs- oder Arbeitsplatzrechner mit größeren Ausfallzeiten), so ist sicherlich ein absolutistischer Ansatz, bei dem der Rechner mit der höchsten Verfügbarkeit als ausgezeichnete Stelle fungiert, einem Abstimmungsverfahren vorzuziehen. Bei einer homogeneren Verteilung der Ausfallzeiten ist dagegen die Gefahr der Nichtverfügbarkeit der ausgezeichneten Kopie größer. Außerdem ist es wichtiger, daß alle Knoten des verteilten Systems möglichst gleichmäßig belastet werden, um Leistungsengpässen vorzubeugen. Deshalb ist bei einer solchen Systemkonfiguration ein Abstimmungsverfahren meist besser geeignet.

Die in der Literatur vorgeschlagenen Abstimmungsverfahren (für einen Überblick siehe auch [7]) unterscheiden sich hauptsächlich in der Strukturierung des Quorums und in der Möglichkeit, die Größe des Quorums zu verändern. Die Vorschläge lassen sich deshalb bzgl. zweier zueinander orthogonalen Dimensionen einteilen. Die eine Dimension legt den Aufbau (strukturiert, unstrukturiert) eines Quorums fest. In der anderen Dimension wird unterschieden, ob die Größe des Quorums fest vorgegeben ist oder ob sie sich dynamisch an die – aufgrund von Partitionierungen – veränderte Zahl an Stimmberechtigten anpaßt. Aus Platzgründen werden im folgenden die Unterschiede nur kurz skizziert. Eine ausführliche Beschreibung ist in [5] zu finden.

[2] N bezeichnet die Kopien- bzw. Stimmenanzahl; $\div$ entspricht der ganzzahligen Division

[3] Bei *strukturierten* Abstimmungsstrategien (siehe nächster Abschnitt) ist diese Fehlertoleranz eingeschränkt, da eine Kompensation ausgefallener Kopien nur entlang der bei diesen Verfahren vorgegebenen logischen Struktur erfolgen kann

Unstrukturiertes vs. strukturiertes Quorum: Bei den unstrukturierten Ansätzen [33, 20, 19, 24, 12, 32] wird ein Quorum gebildet, indem man Q_R bzw. Q_W Stimmen ansammelt, die von beliebigen Kopien stammen können. Dagegen werden bei den strukturierten Verfahren [1, 2, 25, 9, 8] die Kopien in einer logischen Baum- oder Gitterstruktur angeordnet. Zur Erlangung eines Quorums müssen dann entlang dieser Struktur in l Ebenen jeweils s Stimmen[4] gesammelt werden. Die Überschneidungsregeln gelten dann sowohl für die Ebenen als auch für die Stimmen in einer Ebene.

Der Vorteil der strukturierten Verfahren liegt bei den geringeren Zugriffskosten, da zum Erreichen eines Quorums weniger Stimmen als im unstrukturierten Fall benötigt werden, ohne daß die Ausfallsicherheit stark beeinträchtigt wird. Dieser Vorteil kommt besonders bei hohem Replikationsgrad zur Geltung.

Nachteilig dagegen ist neben dem höheren Verwaltungsaufwand, daß die Stimmen nicht mehr bei beliebigen Kopien, sondern nur noch entlang der vorgegebenen Struktur gesammelt werden können. Ähnlich zu den absolutistischen Ansätzen werden die Anfragen damit nicht mehr gleichmäßig über alle Knoten verteilt, sondern konzentrieren sich insbesondere bei baumartigen Strukturen auf wenige „strategische“ Knoten.

Das Prinzip eines auf einer logischen Baumstruktur basierenden Replikationsverfahrens wird in Abschnitt 3.1.1.2 näher beschrieben.

Dynamisches vs. statisches Quorum: Wird eine replizierte Datenbank infolge von Verbindungsunterbrechungen mehrfach partitioniert, so wird es immer schwieriger, die zu einem Quorum erforderliche Stimmenanzahl zu erreichen. Deshalb ist es aus Gründen der Verfügbarkeit wünschenswert, daß sich ein Quorum *dynamisch* an die gerade verfügbare Gesamtstimmenanzahl anpaßt [24, 12, 2]. Um Inkonsistenzen zu vermeiden, darf es dabei aber nicht vorkommen, daß in zwei getrennten Partitionen jeweils ein Schreibquorum erreicht werden kann.

Auch bei einer Veränderung des Zugriffsverhaltens ist eine andere Stimmverteilung wünschenswert [22]. Damit kann z. B. bei einer zeitweilig hohen Update-Rate das Schreibquorum zu Lasten des Lesequorums verkleinert werden.

Als typischer Vertreter dieser Art von Replikationsverfahren wird in Abschnitt 3.1.1.1 der Ansatz der dynamischen Abstimmung detaillierter beschrieben.

2.2.1.3 Read-One-Copy-basierte Verfahren. Als Spezialfall der Abstimmungsverfahren können die *Read-One-Copy*-basierten Verfahren angesehen werden, bei denen im fehlerfreien Fall zum Lesen die Zustimmung einer beliebigen Kopie genügt (Q_R = 1). Dadurch kann durch den alleinigen Zugriff auf die lokale Kopie (falls vorhanden) der in den meisten Anwendungen sehr viel häufigere Lesezugriff optimiert werden.

Bei den anderen Abstimmungsverfahren müssen dagegen im allgemeinen Fall für den aktuellen Objektwert mehrere Kopien konsultiert werden. Bei den absolutistischen Verfahren genügt es zwar, zum Lesen auf nur eine Kopie zuzugrei-

[4] bzw. alle Stimmen auf Ebene k, falls Ebene k weniger als s Stimmen besitzt

fen, diese ist für konsistentes Lesen jedoch fest vorgeschrieben (Kopie der ausgezeichneten Stelle), so daß in der Regel ein entfernter Zugriff notwendig ist (Verlust des lokalen Lesens).

Die Basis aller Read-One-Copy-basierten Verfahren bildet die *ROWA*-Methode[5] (siehe z. B. [3]), bei der alle Kopien synchron aktualisiert werden müssen (*write all*). Dadurch wird der Vorteil des rein lokalen Lesens mit dem Verlust der Schreibverfügbarkeit bei Knotenausfällen/Partitionierungen erkauft.

Weitergehende Ansätze versuchen diese schlechte Schreibverfügbarkeit im Fehlerfall zu verbessern, indem sie beim Schreiben nur die erreichbaren Kopien aktualisieren ([3, 16, 17], siehe Abschnitt 3.1.2) oder bei Nicht-Verfügbarkeit einzelner Kopien auf ein anderes Replikationsverfahren umschalten [15].

Neben der Verfügbarkeit kann auch der Durchsatz des ROWA-Ansatzes verbessert werden, indem man die Lage und die Anzahl der Kopien dynamisch dem Zugriffsverhalten (Schreib/Leseoperationen pro Zeitraum) anpaßt [23].

Als Repräsentant dieser Klasse von Verfahren wird in Abschnitt 3.1.2 das Available-Copies-Verfahren näher beschrieben.

Der Einsatz Read-One-Copy-basierter Verfahren eignet sich immer dann, wenn der Anteil an Lesetransaktionen hoch und Fehlerfälle selten sind, da die für die Realisierung des „1-Kopien-Lesens“ zusätzlichen Verwaltungsinformationen bei jedem Fehlerfall durch aufwendige Rekonfigurationsalgorithmen aktualisiert werden müssen (siehe Abschnitt 3.1.2).

2.2.2 Semantische Kopien-Update-Strategien

Bei den bisher beschriebenen (syntaktischen) Kopien-Update-Strategien müssen entweder mehrere Kopien (Abstimmungs- und Read-One-Copy-basierte Verfahren) oder die im allgemeinen entfernte ausgezeichnete Datenkopie (absolutistische Ansätze) synchron aktualisiert werden. Besitzt das Replikationsverfahren jedoch semantische Kenntnisse über die durchzuführende Update-Operation (z. B. Operationen sind kommutativ), so kann im Idealfall eine Update-Strategie gewählt werden, bei der nur die *jeweils lokale* Kopie mit dem Transaktions-Commit aktualisiert werden muß. Das semantische Wissen kann damit die sonst notwendige kopienübergreifende Synchronisation (Punkt 1 in Abschnitt 2.2) ersetzen. In Abschnitt 3.2.2 wird eine semantische Update-Strategie vorgestellt, bei der es aufgrund der Kenntnis über die Kommutativität von Transaktionen genügt, die lokale Kopie synchron zu aktualisieren.

Neben der schon erwähnten begrenzten Einsatzfähigkeit semantischer Verfahren kann, wegen der großen Anzahl asynchroner Updates, das Lesen eines konsistenten Datenbankzustands erschwert oder sogar unmöglich werden (siehe Abschnitt 3.2.2).

Abbildung 2 zeigt als Zusammenfassung für die jeweilige Update-Strategie die Zahl und Lage der *synchron* zu aktualisierenden Kopien und ordnet die in Abschnitt 3 konkreter beschriebenen Replikationsverfahren entsprechend der hier erarbeiteten Klassifikationskriterien ein.

[5] Read-One-Write-All

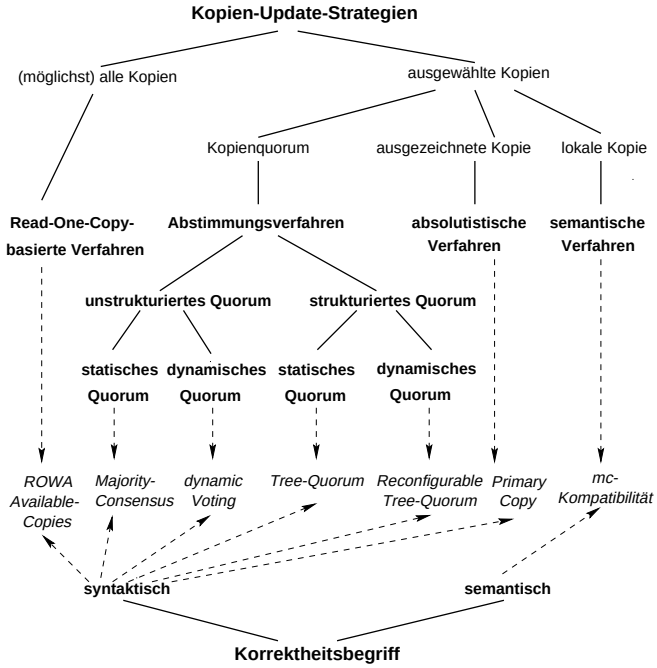


Abbildung 2. Die Kopien-Update-Strategien im Überblick: Die Abbildung zeigt die Zahl der synchron zu aktualisierenden Kopien, die Kriterien zur Klassifikation der verschiedenen Replikationsmethoden sowie die Einordnung konkreter Replikationsverfahren, die im Rahmen dieses Artikels näher beschrieben werden. Außerdem wird das jeweilige Korrektheitskriterium angegeben, auf dem die Verfahrensklasse basiert

2.3 Behandlung von Netzpartitionierungen: Optimistische vs. pessimistische Verfahren

Die Gewährleistung bzw. Erhöhung der Verfügbarkeit im Fehlerfall ist ein wichtiger Grund für den Einsatz von Kopien. Deshalb sollte ein Zugriff sowohl bei Knotenausfällen als auch bei Partitionierungen weiterhin möglich sein, ohne dabei die Datenbankkonsistenz zu gefährden. Knotenausfälle stellen dabei den weit einfacheren Fehlerfall dar, da hier ein zeitweise ausgefallener Rechnerknoten durch einfaches „Nachziehen“ der in der Zwischenzeit stattgefundenen Updates reintegriert werden kann. Lassen sich deshalb z. B. aufgrund der Netztopologie Partitionierungen ausschließen [29], so können auch einfachere Replikationsverfahren eingesetzt werden, welche die Datenkonsistenz nur bei Rechnerausfällen garantieren. Diese Voraussetzung trifft jedoch in den meisten Fällen nicht zu. Die überwiegende Mehrheit der Replikationsverfahren ist deshalb so konzipiert, daß sie auch Netzpartitionierungen tolerieren können. Die dazu in der Literatur zu findenden Vorschläge gehen von einer der beiden folgenden Annahmen aus:

Optimistische Verfahren [13, 6, 18] gehen von der Annahme aus, daß Inkonsistenzen nur selten auftreten und daß diese bei Aufhebung der Partitionierung erkannt und aufgelöst werden können. Deshalb besteht nicht die Notwendigkeit, im Partitionierungsfall den Datenzugriff einzuschränken. Optimistische Ansätze unterscheiden sich untereinander hauptsächlich in der Art und Weise, wie die aufgetretenen Inkonsistenzen behoben werden. Als Beispiel für ein optimistisches Verfahren wird in Kapitel 3.2.1 das Prinzip des Data-Patch-Verfahrens näher beschreiben, bei dem

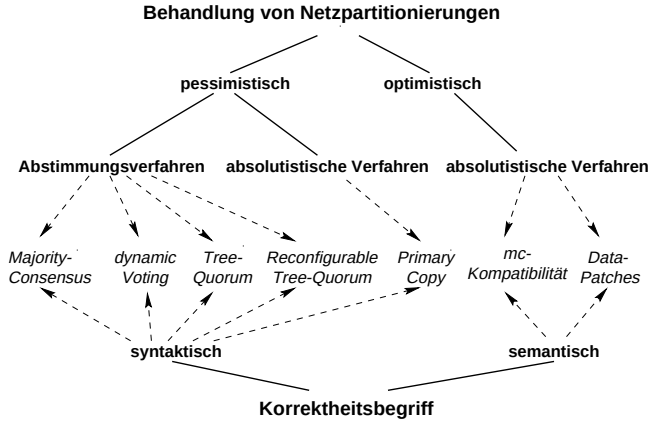


Abbildung 3. Behandlung von Netzpartitionierungen im Überblick: Die Abbildung zeigt die verschiedenen Prinzipien zur Behandlung von Netzpartitionen. Dabei wurden die in Abbildung 2 schon verwendeten Klassifikationskriterien sowie die konkreten Verfahren wieder aufgegriffen und entsprechend eingeordnet

schon beim Datenbankentwurf Regeln zur Konfliktauflösung definiert werden.

Pessimistische Verfahren [30, 29, 16, 20, 1, 26, 22] gehen von einer *worst-case*-Annahme aus: Falls es in den Partitionen zu Inkonsistenzen kommen kann, dann treten diese auch ein. Deshalb ist es besser, die Verfügbarkeit der Daten im Partitionierunsfall zugunsten ihrer Konsistenz einzuschränken. Ungehindertes Arbeiten mit einem Objekt ist damit immer nur in einer Partition möglich, während in den anderen Partitionen allenfalls ein lesender Zugriff erlaubt wird. Pessimistische Verfahren unterscheiden sich untereinander hauptsächlich darin, wann und wie die Einschränkung erfolgt. Das Zusammenfügen von Partitionen bei der Reintegration ist dadurch natürlich problemlos möglich, da die Änderungen nur in jeweils einer Partition erlaubt waren. Diese müssen dann in den anderen Partitionen nachgezogen werden.

Abbildung 3 beschreibt die Strategien der verschiedenen (Klassen von) Replikationsverfahren im Partitionierungsfall.

3 Ausgewählte Verfahren

Einige der zuvor allgemein beschriebenen Prinzipien der Replikationskontrolle sollen nun anhand ausgewählter Verfahren exemplarisch veranschaulicht werden. Aus Platzgründen konzentrieren wir uns hier jedoch in der Regel auf bestimmte Teilaspekte des jeweiligen Verfahrens. Eine detailliertere und umfangreichere Beschreibung verschiedener Replikationsmethoden ist in [5] zu finden.

3.1 Syntaktische Verfahren

3.1.1 Abstimmungsverfahren

In Kapitel 2 wurden für die Abstimmungsverfahren zwei orthogonale Kriterien (unstrukturiert/strukturiert bzw. statisch/dynamisch) beschrieben, mit denen sich die verschiedenen Verfahren in vier Bereiche einteilen lassen (siehe Abbildung 2). Abgesehen vom Bereich der unstrukturierten, statischen Abstimmungsverfahren, deren Grundprinzip bereits in Abschnitt 2.2.1 vorgestellt wurde, wird nun aus jedem dieser Bereiche ein Verfahren detaillierter beschrieben.

3.1.1.1 Die Dynamic-Voting-Methode: ein unstrukturiertes, dynamisches Verfahren. Fortgesetzte und eventuell langdauernde Partitionierungen führen dazu, daß eine immer größere Zahl an Knoten nicht erreichbar ist (und damit ihre potentielle Zustimmung wegfällt), wodurch die Bildung eines Quorums erschwert oder sogar unmöglich wird.
Beispiel 1: Probleme bei der Quorumsbildung mit dem Majority-Consensus-Verfahren

Zerfällt ein nach dem Majority-Consensus-Verfahren [33] arbeitendes Datenbanksystem mit Replikationsgrad 20 zunächst in zwei Partitionen mit 9 bzw. 11 Knoten und anschließend die „11er-Partition“ weiter in 5 und 6 Knoten, so kann in keiner der drei resultierenden Partitionen die für das Quorum notwendige Stimmenanzahl mehr erreicht werden.

Der Grund dafür ist, daß die Größe eines Quorums *statisch* durch die ursprüngliche Gesamtzahl der Stimmen festgelegt wird und deshalb nicht an die jeweils aktuell verfügbare Zahl der Knoten angepaßt werden kann.

Einen flexibleren Ansatz in diesem Zusammenhang bietet das *Dynamic-Voting-Verfahren* [24][6], bei dem die Gesamtzahl der stimmberechtigten Knoten bei jeder Abstimmung neu festgelegt wird: Nur die Knoten, die bei der letzten Änderung teilgenommen haben, besitzen bei der nächsten Abstimmung Stimmrecht.
Beispiel 2: Quorumsanpassung beim Dynamic-Voting-Verfahren

Haben in Beispiel 1 vor der ersten Partitionierung 11 Kopien einem Update zugestimmt, so sind nur noch diese 11 Kopien stimmberechtigt, d. h. bei weiteren Updates müssen nur noch $11 \div 2 + 1 = 6$ Kopien zustimmen. Dadurch kann im Gegensatz zum statischen Majority-Consensus-Verfahren die benötigte Stimmenanzahl auch noch nach der zweiten Partitionierung erreicht werden.

Das Beispiel zeigt allerdings auch ein Problem auf. Durch fortlaufende Partitionierungen der Mehrheitspartition[7] sind immer weniger Knoten stimmberechtigt. Im Extremfall kann dies zu einer „2-Kopien-Mehrheitspartition“ führen. Kommt es zu einer Reintegration der restlichen 18 Kopien, so ist mit diesen kein Arbeiten möglich, da sie solange kein Stimmrecht mehr besitzen, bis sie an einer Änderung mit den letzten beiden stimmberechtigten Kopien teilgenommen haben [24, 7]. Diese Reduzierung der Verfügbarkeit kann gemäß des Vorschlags von Tang [32] dadurch begrenzt werden, daß man eine untere Grenze für die minimale Zahl der Stimmberechtigten festlegt.

3.1.1.2 Das Tree-Quorum-Verfahren: ein strukturiertes, statisches Verfahren. Ein hoher Replikationsgrad führt bei einer Quorumsbestimmung zu großem Kommunikationsaufwand. Beispielsweise müssen bei 100 Kopien mindestens 51 Kopien befragt werden, um mit dem Majority-Consensus-Verfahren ein Quorum zu erhalten. Ordnet man dagegen die Kopien über eine logische Baumstruktur an [25, 1, 9] und sammelt man die für das (strukturierte) Quorum notwendi-

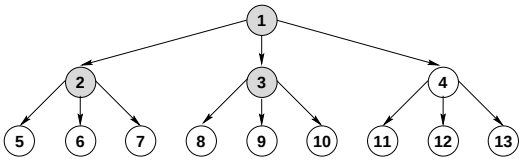
[6] Ein sehr ähnlicher Ansatz wurde von [12] vorgeschlagen

[7] Mit Mehrheitspartition wird die Partition bezeichnet, in der Updates möglich sind

gen Stimmen entlang dieser Struktur, so müssen bedeutend weniger Knoten angefragt werden (siehe Beispiel 3).

Die logische Baumstruktur wird über zwei Parameter beschrieben: die Höhe h und den Verzweigungsgrad d. Zur Vereinfachung der Darstellung wird von vollständigen Bäumen ausgegangen. In [1] werden die erforderlichen Modifikationen für unvollständige Bäume diskutiert.

Der von [1] vorgeschlagene (depth-first)-Algorithmus konstruiert ein Quorum der Dimension $\langle l, s\rangle$, indem er bei der Wurzel beginnt und in jeder Ebene des Baumes s beliebige Knoten s' zur Stimmabgabe auffordert. Kommt ein Knoten aus s' nicht innerhalb eines bestimmten Zeitraums (timeout) der Abstimmungsaufforderung nach, so werden stattdessen s seiner Kinder zur Stimmabgabe aufgefordert. Das Quorum $\langle l, s\rangle$ konnte erfolgreich gebildet werden, wenn auf insgesamt l Ebenen jeweils s Knoten zugestimmt haben. *Beispiel 3:* minimales Quorum der Dimension $\langle 2, 2\rangle$



Ein Quorum der Dimension $\langle 2, 2\rangle$ kann im Beispiel 3 im optimalen Fall durch die Wurzel 1 und zwei ihrer Kinder, z. B. 2, 3, gebildet werden (schraffierte Knoten). Ist Knoten 2 nicht erreichbar, so kann seine Stimme durch die Stimmen zweier seiner Kinder, also z. B. 5 und 6, ersetzt werden. Die meisten Knoten müssen befragt werden, wenn die Wurzel nicht verfügbar ist. Dann benötigt man stattdessen zwei Stimmen ihrer Kinder (z. B. 3 und 4) sowie noch jeweils zwei von deren Kindern (z. B. 8, 9 und 11, 13), um das notwendige Quorum zu erhalten.

Für die Einhaltung der 1-Kopie-Serialisierbarkeit müssen die Überschneidungsregeln aus Abschnitt 2.2.1 für beide Dimensionen gelten:[8]

- Schreib/Schreib-Überschneidungsregel:
  $2 * Q_W.l > h$ und $2 * Q_W.s > d$
- Schreib/Lese-Überschneidungsregel:
  $Q_W.l + Q_R.l > h$ und $Q_R.s + Q_W.s > d$

Das Quorum $\langle 2, 2\rangle$ aus Beispiel 3 erfüllt diese Bedingungen.

Im Vergleich zu unstrukturierten Abstimmungsverfahren müssen weniger Kopien zur Stimmabgabe aufgefordert werden (Größenordnung $\log N$ anstatt $N \div 2$ beim Majority-Tree-Verfahren), ohne daß die Verfügbarkeit stark reduziert wird [1].

Neben der im Beispiel verwendeten Mehrheitsverteilung im jeweiligen Quorum (*Majority-Tree-Verfahren*) werden auch andere Verteilungen vorgeschlagen. Beispielsweise genügt im fehlerfreien Fall beim *Read-Root-Verfahren* ($Q_R = \langle 1, d \div 2+1\rangle$, $Q_W = \langle h, d \div 2+1\rangle$) zum Lesen die Zustimmung der Wurzel, während zum Schreiben in jeder Ebene des Baumes jeweils eine Mehrheit der Kopien zustimmen muß [1]. Ein Lesezugriff ist damit ähnlich günstig wie beim ROWA-Protokoll. Die Schreibverfügbarkeit ist dabei jedoch mit dem Majority-Consensus-Verfahren vergleichbar [1]. Sie hängt allerdings völlig von der Verfügbarkeit der Wurzel ab. Zur Abschwächung dieser Abhängigkeit von der Verfügbarkeit der Wurzel wird in [9] vorgeschlagen, die Wurzel aufzusplitten: Aus einem Baum wird dann ein „Wald“ von Bäumen. Zum Lesen und zum Schreiben muß dann jeweils die Mehrheit der Wurzeln bzw. deren Kinder zustimmen.

3.1.1.3 Das Tree-Quorum-Verfahren mit rekonfigurierbarer Baumstruktur: ein strukturiertes, dynamisches Verfahren. Die im vorherigen Abschnitt skizzierten Vorschläge zur Verbesserung der Schreibverfügbarkeit beim *Read-Root*-Verfahren verteuern unnötigerweise den Lesezugriff im fehlerfreien Fall. In [2] wird ein anderer Weg vorgeschlagen, der ohne Erhöhung der Lesekosten die Schreibverfügbarkeit verbessert: *Rekonfiguration*. Ist aufgrund von Fehlern die Bildung eines Schreibquorums in der gegenwärtigen logischen Struktur nicht mehr möglich, so wird diese so „umgebaut“ (rekonfiguriert), daß in der neuen Struktur ein Schreibquorum wieder erreicht werden kann.

Zur Durchführung dieser Rekonfiguration muß ein (leichter erreichbares) *Rekonfigurationsquorum* gesammelt werden, das zur Konsistenzerhaltung die folgenden Überschneidungsregeln einhalten muß:

1. Rekonfiguration/Schreib-Überschneidungsregel:
   Damit in der rekonfigurierten Struktur auch ein Knoten mit aktueller Kopie enthalten ist, muß sich ein Rekonfigurationsquorum mit dem Schreibquorum überschneiden.
2. Rekonfiguration/Rekonfiguration-Überschneidungsregel:
   Diese Regel stellt sicher, daß zu jedem Zeitpunkt nur eine logische Struktur gültig ist.

Die Quorumsverteilung des fehlertoleranten Majority-Tree-Verfahrens ($Q_{reconfig} = \langle h \div 2+1, d \div 2+1\rangle$, Abschnitt 3.1.1.2) erfüllt beispielsweise diese Überschneidungsregeln für das Read-Root-Verfahren.

Sind Fehler relativ selten, so kann damit ein leseoptimiertes Verfahren beibehalten und trotzdem die Schreibverfügbarkeit verbessert werden. Bei höherer Fehlerwahrscheinlichkeit sind jedoch die Kosten für die häufige Rekonfiguration höher als ein Verfahren, das auch im fehlerfreien Fall mit einem nur „suboptimalen“ Lesequorum (z. B. $Q_r = \langle 2, d \div 2 + 1\rangle$, $Q_w = \langle h - 1, d \div 2 + 1\rangle$) arbeitet.

Die Idee der Rekonfiguration zur Erhöhung der Schreibverfügbarkeit wurde von den Autoren auch für logische Gitterstrukturen angewandt [2].

3.1.2 Die Available-Copies-Methode: ein Read-One-Copy-basiertes Verfahren

Das Verfahren von Bernstein und Goodman [3] verbessert die Schreibverfügbarkeit des ROWA-Ansatzes im Fehlerfall, indem bei einer Änderungsoperation statt aller nur die *verfügbaren* Kopien synchron aktualisiert werden müssen (*Read-One-Write-All-Available*). Welche Kopien verfügbar sind, wird in replizierten *Verzeichnissen (directories)* gespeichert. Vor jedem Lese/Schreibzugriff wird das lokale Verzeichnis konsultiert, um die gegenwärtig verfügbaren Kopien

[8] bei jeweils 1 Stimme pro Knoten

zu erhalten. Zum Lesen wird dann auf eine dieser verfügbaren Kopien zugegriffen. Beim Schreiben werden nur die laut Verzeichnis verfügbaren Kopien aktualisiert. Sind die Informationen in den Verzeichnissen aktuell, so kann damit das Schreiben nicht blockieren. Stellt sich z. B. durch das Mißlingen einer Änderungsoperation heraus, daß die in den Verzeichnissen gehaltenen Informationen nicht mehr der gegenwärtigen Verfügbarkeitssituation entsprechen, so werden diese dynamisch über Systemtransaktionen (status transactions) angepaßt. Die Aktualisierung der replizierten Verzeichnisse ist jedoch aufwendig, so daß das Verfahren nur geeignet ist, wenn Knotenausfälle selten sind [3, 29].

In dieser Grundform toleriert dieser Ansatz nur Knotenausfälle (vgl. Abschnitt 2.3). Zur Tolerierung von Netzpartitionierunen sind weitere Restriktionen notwendig. Ein interessanter Vorschlag hierzu wurde von [16] mit dem *Virtual-Partition-Protokoll* gemacht. Hier wird der Zugriff eines Knotens nur dann erlaubt, wenn in seinem Verzeichnis eine Mehrheit aller Knoten enthalten ist. Für eine korrekte Arbeitsweise müssen dazu jedoch alle Kopien einer Partition identische Verzeichniseinträge besitzen. Dafür müssen Änderungen im Verzeichnis über ein aufwendiges Protokoll [16, 5] ausgetauscht werden.

3.2 Semantische Verfahren

Die in diesem Kapitel beschriebenen Verfahren nützen semantisches Wissen aus, um die in Kapitel 2 angesprochenen Punkte bzgl. Datenkonsistenz, Kopien-Update-Verhalten und Fehlerbehandlung realisieren zu können.

3.2.1 Das Data-Patch-Verfahren: ein optimistisches Verfahren

Das Aufgabengebiet des *Data-Patch-Verfahrens* [18] beschränkt sich auf die Wiederherstellung eines konsistenten Datenbankzustands nach Partitionierungen (Punkt 3). Dazu werden beim Entwurf der Datenbank für jede Relation zwei Regeln spezifiziert, wie aus den unterschiedlichen Kopienwerten ein neuer einheitlicher, konsistenter Wert erzeugt wird: Im folgenden wird zur einfacheren Beschreibung von zwei Partitionen, P_1 und P_2, ausgegangen.

Tupel-Einfügungsregel: Eine dieser Regeln wird angewandt, wenn ein Tupel in nur einer Partition eingefügt wurde.

Keep-Regel: Wenn ein Tupel in nur einer Partition eingefügt wurde, füge es auch in der anderen Partition ein.
Remove-Regel: Wenn ein Tupel in nur einer Partition gefunden wurde, dann lösche es wieder.
Program-Regel: Beauftrage das angegebene Programm mit der Konfliktauflösung.
Notify-Regel: Benachrichtige den Datenbankadministrator. Dieser muß den Konflikt dann manuell auflösen.

Tupel-Integrationsregel: Eine dieser Regeln wird verwendet, wenn ein Tupel mit gleichen Schlüssel sowohl in der Partition P_1 als auch in der Partition P_2 eingefügt oder verändert wurde.

Latest-Regel: Das zuletzt eingefügte Tupel ist das gültige.
Primary-Regel: Das Tupel auf dem Knoten k ist das korrekte.
Arithmetic-Regel: Der Tupelwert berechnet sich nach der folgenden Formel: $neuer\,Wert = Wert(P_1) + Wert(P_2) - alter\,Wert$.
Program-Regel, Notify-Regel: siehe oben

Dabei ist es natürlich für die Anwendung der Tupel-Integrationsregel notwendig, daß jedes neu eingefügte Tupel auch während einer Partitionierung immer einen eindeutigen und unveränderbaren Schlüssel erhält.

Beispiel 4: Konto-Relation bei einer Bank

Die Schemadefinition bei der Konto-Relation einer Bank besteht vereinfachend aus:

Konto#	Kunden-Name	Konto-stand	Einf.-regel	Integ.-regel
Key	String	Real	Keep	Arithmetic

Wird bei einer Partitionierung in irgendeiner Kopie ein neues Tupel (= neues Konto) eingefügt, so wird dieses Tupel aufgrund der Keep-Regel auch in den anderen Kopien ergänzt. Wird dagegen ein Tupel in mehreren Kopien verändert, so erfolgt die Konfliktauflösung mittels der Arithmetic-Regel.

Der Zustand, in dem sich die Datenbank nach Anwendung aller Regeln befindet, ist im allgemeinen nicht mit dem serialisierbaren Zustand identisch, der erreicht worden wäre, wenn die Partitionierung nicht stattgefunden hätte. Da sich Datenänderungen häufig auch außerhalb der Datenbank auswirken (siehe folgendes Beispiel einer Geldauszahlung an einen Bankkunden), ist das Erreichen dieses serialisierbaren Datenbankzustands nicht von allergrößter Wichtigkeit.

Beispiel 5: Transaktion mit Integritätsbedingung

Dürfen Kunden ihr Konto nicht überziehen, so wird eine Auszahlungstransaktion über eine Integritätsbedingung den auszuzahlenden Betrag mit dem momentanen Kontostand vergleichen und bei einer Kontoüberziehung die Auszahlung ablehnen. Wird in zwei verschiedenen Partitionen jeweils das gesamte Guthaben abgehoben, so führt dies aufgrund fehlender Informationen aus der anderen Partition zu einem negativen Kontostand und somit zur Verletzung der Integritätsbedingung. Die Wiederherstellung der Integritätsbedingung (Kontostand $\geq 0\,\mathrm{DM}$) und damit eines serialisierbaren Datenbankzustands entspricht aber nicht mehr der Realität, da tatsächlich zweimal das gesamte Guthaben ausgezahlt wurde.

3.2.2 Die Multi-Copy-Kompatibilität

Wie in Abschnitt 2 erwähnt, kann durch Einbeziehung von Anwendungssemantik der Durchsatz und die Verfügbarkeit erhöht werden. Ein gutes Beispiel dafür ist die von Kumar und Stonebraker [26] vorgeschlagene Methode der *Multi-Copy-Kompatibilität (mc-Kompatibilität, mc-compatibility)*, bei der die Kommutativität einzelner Transaktionen ausgenützt wird. Kommutative Transaktionen ermöglichen es, Updates in unterschiedlicher Reihenfolge auf den einzelnen Kopien durchzuführen, ohne die Datenbankkonsistenz zu gefährden.

Da jedoch die wenigsten Anwendungen aus nur kommutativen Transaktionen bestehen, werden bei diesem Ver-

fahren[9] die Transaktionen in eine kommutative (*C-TA*) und eine nicht-kommutative (*NC-TA*) Klasse eingeteilt. Bei einer Bankanwendung sind z. B. Ein- bzw. Auszahlungstransaktionen $T_{EA}(k, ea) = k + ea$ kommutativ, Zinsberechnungstransaktionen $T_Z(k, p) = k + pk$ dagegen nicht.

Für jede Klasse von Transaktionen wird nun eine eigene Update-Strategie angewandt:

Aufgrund ihrer Kommutativität kann bei den *C-TAs* eine rein lokale Update-Strategie realisiert werden, bei der ohne kopienübergreifenden Synchronisation nur die lokale Kopie synchron aktualisiert werden muß. Die restlichen Kopien werden asynchron über ein *Spoolprogramm* [26] aktualisiert.

Für *NC-TAs* muß dagegen ein restriktiveres *Abstimmungsverfahren* zum Kopien-Update eingesetzt werden, d. h. es müssen alle Kopien im Schreibquorum *synchron* aktualisiert werden. Um wenigstens das Update der restlichen Kopien asynchron und konsistent durchführen zu können, muß die NC-TA durch eine zur ihr „äquivalente“ C-TA ersetzt werden. Beispielsweise kann die obige Zinsberechnungstransaktion T_Z nach Berechnung des Zinses $Z = pk$ durch die äquivalente Transaktion $T_{EA}(k, Z) = k + Z$ ersetzt werden. Diese C-TA nennt man dann *mc-kompatibel* zu der entsprechenden NC-TA.

Durch Ausnutzung der Kommutativität kann also im Vergleich zu einem rein syntaktischen Replikationsverfahren ein höherer Transaktionsdurchsatz erzielt werden. Da außerdem die Update-Reihenfolge von kommutativen Transaktionen unerheblich ist, sind auch während Partitionierungen Updates in allen Partitionen möglich. Diese *optimistische* Partitionierungsbehandlung erhöht damit auch die Verfügbarkeit. Der Durchsatz- und Verfügbarkeitsgewinn ist natürlich abhängig vom Anteil der kommutativen Transaktionen am Gesamtvolumen der Transaktionen.

Das Verfahren besitzt jedoch auch einen schwerwiegenden Nachteil, der bei vielen Anwendungen einen Praxiseinsatz fraglich erscheinen läßt: Aufgrund der *asynchronen und verteilten* Ausführung der C-TAs ist im laufenden Betrieb nie sichergestellt, daß eine Kopie tatsächlich alle Updates und damit einen aktuellen Datenbankzustand widerspiegelt. Damit ist auch nicht sicher, daß das Quorum einer NC-TA eine Kopie enthält, die den aktuellen Objektwert besitzt. Es kann deshalb beispielsweise vorkommen, daß die Einzahlung (C-TA) eines Kunden bei einer wenig später stattfindenden Zinsberechnung (NC-TA) noch nicht berücksichtigt wird. Die hieraus resultierenden Inkonsistenzen werden wohl nur für wenige Anwendungen tolerierbar sein.

4 Abschließende Bemerkungen

Erst durch die Replikation von Daten ist ein verteiltes System bzgl. der Verfügbarkeit einem zentralen System überlegen. Daneben profitiert meist auch der Datendurchsatz von der Existenz von Kopien. Andererseits benötigen replizierte Systeme zusätzliche Synchronisationsmechanismen – realisiert durch sogenannte *Replikationsverfahren* – zur Sicherstellung der Datenkonsistenz.

In diesem Papier wurden zuerst die verschiedenen Punkte diskutiert, die ein Replikationsverfahren zu realisieren hat. Anhand dieser Punkte wurden die verschiedenen prinzipiellen Lösungsvorschläge klassifiziert sowie ihre Vor- und Nachteile kritisch beleuchtet. Anschließend erfolgte eine Beschreibung einiger der klassifizierten Verfahren.

Der inhärente Zielkonflikt zwischen Verfügbarkeit/ Durchsatz einerseits sowie Datenkonsistenz andererseits wird dabei von den verschiedenen Verfahren sehr unterschiedlich gelöst. Die Unterschiede basieren zum einen an dem den einzelnen Verfahren zugrundeliegenden Korrektheitskriterium. Dieses kann rein syntaktisch definiert (z. B. 1-Kopie-Serialisierbarkeit), unter Zuhilfenahme von semantischen Wissen (z. B. Kommutativität) oder allein über die Anwendungssemantik bestimmt werden.

Da die Replikationsverfahren nicht alle in Kapitel 2 beschriebenen Aufgabenfelder abdecken, sind hier auch Kombinationen von Verfahren mit unterschiedlichen Korrektheitsbegriffen denkbar. Beispielsweise kann die syntaktische Available-Copies-Methode (siehe Abschnitt 3.1.2) mit dem semantischen Data-Patch-Verfahren (siehe Abschnitt 3.2.1) kombiniert werden, um damit auch Partitionierungen tolerieren zu können: Das Sperren aller verfügbaren Kopien beim Schreiben garantiert dann innerhalb einer Partition die 1-Kopie-Serialisierbarkeit. Die durch Partitionierungen möglichen Inkonsistenzen werden dann auf der Basis der semantischen Regeln des Data-Patch-Verfahrens aufgelöst. Damit wird dann ein anwendungssemantisch korrekter, aber nicht notwendigerweise 1-Kopie-serialisierbarer Datenbankzustand erreicht.

Der Available-Copies-Ansatz zeigt auch, daß sich die Replikationsverfahren bzgl. der unterstützten Fehlersemantiken [11] unterscheiden. Während einige wenige Verfahren nur Knotenausfälle tolerieren, erhalten andere die Datenbankkonsistenz auch im Falle von Partitionierungen.

Inzwischen bieten einige kommerzielle Datenbanksysteme Replikationsmöglichkeiten an: Sybase Replication Server [10], Oracle [31], Ingres Replicator, Adabas Entire SQL, Informix, DB2. Häufig basieren diese Verfahren auf dem im Abschnitt 2.2.1.1 vorgestellten Primary-Copy-Prinzip, da die Kopien in einer *Master-Slave*-Beziehung [28] zueinander stehen: Eine Änderung ist nur über die Master-Kopie möglich. Die meist asynchron über Trigger aktualisierten Slave-Kopien stehen nur für einen Lesezugriff zur Verfügung.

Ingres und Oracle bieten dagegen auch eine *Peer-to-Peer* Replikationsmethode [28] an, bei der Änderungen über jede Kopie möglich sind. Da auch hier die Änderungspropagierung asynchron über Trigger erfolgt (optimistische Ansätze) sind Inkonsistenzen nicht ausgeschlossen, die dann über vom Anwender zu definierende Regeln aufgelöst werden (vgl. Abschnitt 3.2.1).
Ein kurzer Überblick über den derzeitigen Stand an kommerziellen Systemen mit Replikationsmöglichkeiten ist in [28] zu finden.

Danksagung. Wir danken Daniela Beuter, Christian Heinlein sowie Manfred Reichert für ihre wertvollen Hinweise und Ratschläge beim Entstehen dieser Ausarbeitung. Außerdem gebührt unser Dank den Gutachtern für ihre sehr hilfreichen Anregungen zur Verbesserung des Beitrags.

[9] im Gegensatz zu vielen anderen „kommutativen“ Replikationsverfahren

References

1. Agrawal, D., El Abbadi, A.: The generalized tree quorum protocol: An efficient approach for managing replicated data. ACM Trans. on Database Systems **17**, 689–717 (1992)
2. Agrawal, D., El Abbadi, A.: Resilient logical structures for efficient management of replicated data. In: Proc. 18th Int'l Conf. on Very Large Data Bases (VLDB) pp. 151–162, Vancouver (1992)
3. Bernstein, P. A., Goodman, N.: An algorithm for concurrency control and recovery in replicated distributed databases. ACM Trans. on Database Systems **9**, 596–615 (1984)
4. Bernstein, P. A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. New York: Addison-Wesley, 1987
5. Beuter, T., Dadam, P.: Prinzipien der Replikationskontrolle in verteilten Systemen. Ulmer Informatik Berichte 95–11, Universität Ulm (1995)
6. Blaustein, B. T., Kaufman, C. W.: Updating replicated data during communications failures. In: Proc. 11th Int'l Conf. on Very Large Data Bases (VLDB), pp. 49–58, Stockholm (1985)
7. Borghoff, U. M.: Fehlertoleranz in verteilten Dateisystemen: Eine Übersicht über den heutigen Entwicklungsstand bei den Votierungsverfahren. GI Informatik Spektrum **14**, 15–27, (1991)
8. Cheung, S. Y., Ahamad, M., Ammar, M. H.: Multi–dimensional voting: A general method for implementing sychronization in distributed systems. In: Proc. 10th Int'l Conf. on Distributed Computing Systems, pp. 362–369, Paris (1990)
9. Chung, S. M.: Enhanced tree quorum algorithm for replica control in distributed database systems. Data & Knowledge Engineering **12**, 63–81, (1994)
10. Colton, M.: Replicated data in a distributed environment. In: Proc. ACM SIGMOD Int'l Conf. on Management of Data, pp. 464–466, Washington, DC (1993)
11. Cristian, F.: Understanding fault-tolerant distributed systems. Communications of the ACM **34**, 56–78 (1991)
12. Davcev, D.: A dynamic voting scheme in distributed systems. IEEE Trans. on Software Engineering **15**, 93–97, (1989)
13. Davidson, S. B.: Optimism and consistency im partitioned distributed database systems. ACM Trans. on Database Systems **9**, 456–481 (1984)
14. Davidson, S. B., Garcia-Molina, H., Skeen, D.: Consistency in partitioned networks. ACM Computing Surveys **17**, 341–370 (1985)
15. Eager, D. L., Sevcik, K. C.: Achieving robustness in distributed database systems. ACM Trans. on Database Systems **8**, 354–381 (1983)
16. El Abbadi, A., Skeen, D., Cristian, F.: An efficient, fault–tolerant protocol for replicated data management. In: Readings in Database Systems, Stonebraker, M. (ed.), pp. 259–273. San Mateo: Morgan Kaufmann 1988
17. El Abbadi, A., Toueg, S.: Maintaining availability in partitioned replicated databases. ACM Trans. on Database Systems **14**, 264–290 (1989)
18. Garcia-Molina, H.: Data–patch: Integrating inconsistent copies of a database after a partition. In: Proc. 3rd Symp. on Reliable Distributed Systems, pp. 38–48, New York (1983)
19. Garcia-Molina, H., Barbará, D.: How to assign votes in a distributed system. Journal of the ACM **32**, 841–860 (1985)
20. Gifford, D. K.: Weighted voting for replicated data. In: Proc. 7th ACM Symp. on Operating Systems Principles (SIGOPS), pp. 150–159, Pacific Grove, California (1979)
21. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. San Mateo: Morgan Kaufmann 1993
22. Herlihy, M.: A quorum–consensus replication method for abstract data types. ACM Trans. on Computer Systems **4**, 32–53 (1986)
23. Huang, Y., Wolfson, O.: A competitive dynamic data replication algorithm. In: Proc. 9th Int'l Conf. on Data Engineering, pp. 310–317, Vienna, (1993)
24. Jajodia, S., Mutchler, D.: Dynamic voting algorithms for maintaining the consistency of a replicated database. ACM Trans. on Database Systems **15**, 230–280 (1990)
25. Kumar, A.: Hierarchical quorum consensus: A new algorithm for managing replicated data. IEEE Trans. on Computers **C-40**, 996–1004 (1991)
26. Kumar, A., Stonebraker, M.: Semantics based transaction management techniques for replicated data. In: Proc. ACM SIGMOD Int'l Conf. on Management of Data, pp. 117–125, Chicago (1988)
27. Minoura, T., Wiederhold, G.: Resilient extended true–copy token schema for a distributed database system. IEEE Trans. on Software Engineering **SE–8**, 173–188 (1982)
28. Ofer, U.: Was Sie schon immer über Replication Server wissen wollten. Datenbank Fokus **6**, 31–36 (1994)
29. Paris, J.-F.: A highly available replication control protocol using volatile witnesses. In: Proc. 14th Int'l Conf. on Distributed Computing Systems, pp. 536–543, Pittsburgh, Pennsylvania (1994)
30. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Trans. on Software Engineering **SE–5**, 188–194 (1979)
31. Stürmer, G.: Oracle 7 A User's and Developer's Guide. Including Release 7.1. London: Thomson 1995
32. Tang, J.: Voting class – an approach to achieving high availability for replicated data. In: Proc. 2nd Int'l Conf. on Databases in Parallel and Distributed Systems, pp. 146–156, Dublin, Ireland (1990)
33. Thomas, R. H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Systems **4**, 180–209 (1979)

Thomas Beuter seit Oktober 1992 wiss. Mitarbeiter an der Universität Ulm, Abteilung Datenbanken und Informationssysteme. Diplom-Informatiker (Technische Universität München, 1992). Seit 1995 Mitarbeiter im Forschungsprojekt *Concurrent Engineering* (in Kooperation mit der Daimler–Benz–Forschung Ulm). *Aktuelle Arbeitsgebiete*: Architekturen für Workflow–Management–Systeme, Replikationsstrategien, erweiterte Transaktionskonzepte

Peter Dadam seit 1990 Professor für Informatik an der Universität Ulm und Leiter der Abteilung Datenbanken und Informationssysteme. Diplom-Wirtschaftsingenieur, Fachrichtung Informatik/Operations Research (Universität Karlsruhe,1978), wiss. Mitarbeiter an der Universität Dortmund und der FernUniversität Hagen, Promotion zum Dr. rer. nat. (FernUniversität Hagen, 1982). Ab 1982 Mitarbeiter am Wissenschaftlichen Zentrum der IBM in Heidelberg, ab 1985 Leiter der Abteilung *Advanced Information Management*.
Aktuelle Arbeitsgebiete: Verteilte, kooperative Informationssysteme, Workflow–Management, Datenbanktechnologie für und Datenbankanwendungen in medizinischen und technisch/wissenschaftlichen Anwendungsgebieten.

This article was processed by the author using the LaTeX style file *gpljour2* from Springer-Verlag.