Universität Ulm

Fakultät für Informatik

Prinzipien der Replikationskontrolle in verteilten Systemen

Thomas Beuter
Peter Dadam
Universität Ulm

Nr. 95-11

Ulmer Informatik-Berichte

Prinzipien der Replikationskontrolle in verteilten Systemen*

T. Beuter, P. Dadam
Universität Ulm

Zusammenfassung

Durch Datenreplikation können prinzipiell schnellere Zugriffszeiten und eine beliebig hohe Fehlertoleranz in einem verteilten System erreicht werden. Ein repliziertes verteiltes System muß allerdings zur Konsistenzsicherung zusätzliche Aufgaben erfüllen. Dazu wurden in der Literatur viele unterschiedliche *Replikationsverfahren* vorgeschlagen. Dieser Artikel beschreibt die zusätzlichen Aufgaben kurz, leitet daraus Kriterien zur Klassifikation der verschiedenen Prinzipien der Replikationskontrolle ab und stellt danach einige der Replikationsverfahren detaillierter vor.

1 Einleitung und Motivation

In den letzten Jahren kann ein immer stärkerer Trend zu verteilten Systemen beobachtet werden. Diese Tendenz läßt sich sowohl in klassischen Bereichen der Informationsverarbeitung (z.B.: verteilte Dateisysteme, verteilte Datenbank- und Informationssysteme) als auch in vielen neuen Anwendungsgebieten (z.B.: mobile Computing, Concurrent Engineering, Workflow-Management, Data-Warehousing) feststellen.

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 Systems gespeichert. Damit wird der Zugriff bei Ausfall datenhaltender Knoten erst ermöglicht sowie der Durchsatz erhöht, weil auf die lokal vorhandene bzw. auf eine schnell erreichbare Kopie zugegriffen werden kann. 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, weil 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 System wie ein nicht repliziertes 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)* [16] sind.
Diese Aufgabe wird durch das Auftreten von Rechner- und Netzausfällen erschwert. Besonders problematisch sind in diesem Zusammenhang *Partitionierungen*, bei denen ein verteiltes System in disjunkte Knotenmengen (*Partitionen*) aufgeteilt wird, wodurch 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 System in einen inkorrekten (= *inkonsistenten*) Zustand 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 (größe-

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

rer Änderungsaufwand, Gefahr von Dateninkonsistenzen) möglichst zu vermeiden. In der Abbildung 1 wird dieser Zielkonflikt und die daraus resultierenden Folgen für die Verwaltung von Datenreplikaten graphisch dargestellt.

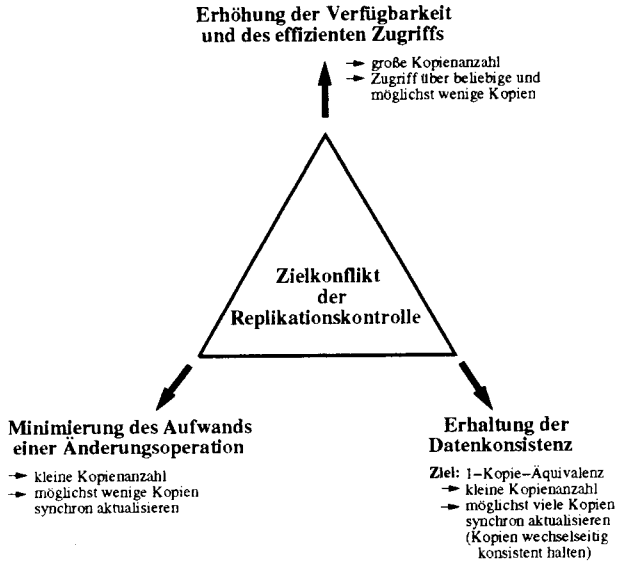


Abbildung 1: Zielkonflikt der Replikationskontrolle

Zur Lösung dieses Zielkonflikts werden in der Literatur viele verschiedene *Replikationsverfahren* vorgeschlagen. In diesem Übersichtartikel werden deshalb nach der Definition einiger grundlegender Begriffe[1] (Kapitel 2) die verschiedenen Aufgaben dieser Verfahren skizziert, um daraus Kriterien zur Klassifikation dieser Methoden zu erstellen (Kapitel 3). In den beiden folgenden Kapiteln werden einige dieser Verfahren detaillierter beschrieben sowie ihre jeweiligen Vor- und Nachteile diskutiert. Kapitel 6 bildet mit einigen Bemerkungen den Schluß dieses Papiers.

Im folgendem werden die verschiedenen Verfahren der Einfachheit halber im Kontext verteilter Datenbanksysteme beschrieben. Ihre Prinzipien lassen sich jedoch in der Regel leicht auf andere Anwendungsbereiche übertragen (z. B. verteilte Dateisysteme [6]).

[1] Diese Begriffe und Notationen, die in vielen Veröffentlichungen verwendet werden, sind deshalb sicherlich vielen Lesern bereits bekannt. Damit jedoch keine Mißverständnisse auftreten, was die Autoren darunter verstehen, werden sie in Kapitel 2 kurz zusammengefaßt.

2 Grundlegende Begriffe

In diesem Abschnitt werden einige grundlegende Definitionen und Notationen beschrieben, die in den folgenden Abschnitten verwendet werden.

Datenbanken, verteilte Datenbanken, replizierte Datenbanken:
Eine *Datenbank* besteht aus einer Menge von logischen *(Daten-) Objekten*, die vom Benutzer über *Transaktionen* manipuliert werden können. Eine *Transaktion* besteht dabei aus einer Folge von Lese- und Schreibaktionen und besitzt die ACID-Eigenschaften [31]. Eine Leseaktion einer Transaktion T auf das Objekt o wird mit $r_T(o)$, ein Schreibzugriff mit $w_T(o)$ bezeichnet.
Die Größe eines Datenobjekts ist im allgemeinen für das Verständnis des einzelnen Replikationsverfahrens unwesentlich. Ein Objekt kann beispielsweise ein oder mehrere Tupel, eine Tabelle oder eine Datei sein.[2] Die Verwaltung einer Datenbank wird von einem *Datenbankmanagementsystem (DBMS)* übernommen. Wird nicht zwischen Datenbank und DBMS unterschieden, so wird der Begriff *Datenbanksystem (DBS)* verwendet.

Bei einem *verteilten Datenbanksystem* werden einzelne Datenbanksysteme zu einem Datenbankverbund zusammengeschlossen. Die einzelnen Datenbanksysteme nennt man üblicherweise *lokale Datenbanksysteme (LDBS)*. Die Koordination und Verwaltung der verteilten Datenbank wird durch ein *globales DBMS* realisiert.

Eine verteilte Datenbank heißt *replizierte Datenbank*, wenn der Wert eines logischen Datenobjekts in mehreren *physikalischen Datenobjekten (Kopien)* gespeichert wird.

Schedule, Synchronisationsverfahren:
Die einzelnen Aktionen einer Transaktion werden aus Leistungsgründen nicht unmittelbar nacheinander („am Stück"), sondern in der Regel verschränkt mit den Aktionen anderer Transaktionen ausgeführt. Eine solche Folge von Lese- und Schreibaktionen verschiedener Transaktionen nennt man die *Schedule* dieser Transaktionen. Beliebige Verschränkungen der

[2] Die Granularität kann dabei allerdings die Leistung eines Verfahrens beeinflußen.

Aktionen können jedoch — z.B. aufgrund von *lost updates* — zu Dateninkonsistenzen führen. Die Aufgabe eines *Synchronisationsverfahren* ist es daher, nur diejenigen Schedule zuzulassen, die die Datenbankkonsistenz erhalten (*konsistente Schedules* [13]). Zur Entscheidung, ob eine Schedule konsistent ist oder nicht, wird meist das Kriterium der *Serialisierbarkeit* verwendet.

serielle, serialisierbare Schedule:
Eine Schedule S heißt *seriell*, wenn für je zwei Transaktionen T_i und T_j aus S gilt, daß alle Aktionen von T_i vor den Aktionen von T_j durchgeführt werden [4], d. h. alle Aktionen einer Transaktion werden unmittelbar nacheinander ausgeführt.
Eine Schedule S heißt *serialisierbar*, wenn es mindestens eine serielle Schedule mit denselben Transaktionen gibt, die den gleichen Datenbankzustand sowie die gleiche Ausgabe erzeugt wie S [4].

Ob eine Schedule serialisierbar ist, kann beispielsweise mit Hilfe eines *Abhängigkeitsgraphen* (*serialization graph* [4]) überprüft werden: Besitzt der Abhängigkeitsgraph einer Schedule S keine Zyklen, so ist eine hinreichende Bedingung dafür erfüllt, daß die Schedule S serialisierbar und damit konsistent ist.

Auch bei replizierten Datenbanken wird die Datenbankkonsistenz 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 die gleiche Ausgabe sowie den gleichen Datenbankzustand erzeugt wie S auf der replizierten Datenbank [4, 16].

Daneben wurden im Bereich der replizierten Datenbanken auch noch andere, im Vergleich zur 1–Kopie–Serialisierbarkeit leichter erfüllbare Serialisierbarkeitsbegriffe vorgeschlagen [58, 46, 26, 21, 35]. Als Beispiel sei hier die *Epsilon–Serialisierbarkeit* [58] genannt, bei der zugunsten höherer Verfügbarkeit/höherem Durchsatz die 1–Kopie–Äquivalenz für Lesetransaktionen aufgegeben wird. Dadurch müssen nicht alle Kopien gleichzeitig (*synchron*) aktualisiert werden. Zur Sicherstellung der Datenkonsistenz müssen aber die Kopien in gleicher serialisierbarer Reihenfolge (z.B. Zeitstempelreihenfolge) aktualisiert werden, so daß nach der Durchführung aller Transaktionen die Datenbank wieder zu einem konsistenten Zustand *konvergiert*.

Bis zum Erreichen dieser Konvergenz dürfen Lesetransaktion einen inkonsistenten Datenbankzustand sehen. Die Höhe der erlaubten Inkonsistenz wird entsprechend der Konsistenzanforderungen der jeweiligen Anwendungen durch den *Überlappungsparameter* ϵ individuell festgelegt. Als Maß der Überlappung wird dabei die (gewichtete) Anzahl der Konsistenzkonflikte herangezogen. Dies führt zu der folgenden formalen Definition der Epsilon–Serialisierbarkeit:

Epsilon–serialisierbare Schedule:
Eine Schedule S einer replizierten Datenbank heißt *epsilon–serialisierbar*, wenn gilt:

1. Die Schedule $S_u = S \backslash \{Lesetransaktionen\}$ ist serialisierbar.
2. Für jede Lesetransaktion gilt: Solange die gesehene Inkonsistenz kleiner als die Überlappungsgrenze ϵ ist, dürfen sich die Aktionen einer Lesetransaktion beliebig mit den Aktionen anderer mit dieser in Konflikt stehenden Transaktionen überschneiden.

Abbildung 1 zeigt die Kompatibilitätstabellen für die 1–Kopie–Serialisierbarkeit und die Epsilon–Serialisierbarkeit. Diese Tabellen legen fest, wann konkurrierende Zugriffe zweier Transaktionen verträglich sind. Durch die begrenzte Kompatibilität zwischen dem Zugriff einer Lesetransaktion (R_Q) und einer Schreibtransaktion (W_U) erlaubt die Epsilon–Serialisierbarkeit eine stärkere Verschränkung paralleler Transaktionen (größere Zahl an korrekten Schedules).

Beispiel 1:
Bei der Datenbank einer Bank ist das Konto k_1 auf die Knoten `A` und `B` repliziert. Auf Knoten `A` befindet sich zusätzlich noch das Konto k_2. Beide Konten enthalten jeweils 1000 DM. Auf dem Knoten `A` wird eine Änderungstransaktion T_U gestartet, die 200 DM von Konto k_1 auf das Konto k_2 überweist (Aktionen: $w^A_{T_U}(k_1, -200)$, $w^B_{T_U}(k_1, -200)$, $w^B_{T_U}(k_2, +200)$).

	R_U^i	W_U^i
R_U^j	kompatibel	inkompatibel
W_U^j	inkompatibel	inkompatibel

Tabelle 1.1

	R_Q^i	R_U^i	W_U^i
R_Q^j	kompatibel	kompatibel	begrenzt kompatibel
R_U^j	kompatibel	kompatibel	inkompatibel
W_U^j	begrenzt kompatibel	inkompatibel	inkompatibel

Tabelle 1.2

Tabelle 1: Kompatibilitätstabelle für die 1-Kopie-Serialisierbarkeit (Tabelle 1.1) und die Epsilon-Serialisierbarkeit (Tabelle 1.2)

R_U: Leseoperation einer Schreibtransaktion

W_U: Schreiboperation einer Schreibtransaktion

R_Q: Leseoperation einer Lesetransaktion

begrenzt kompatibel: bis zum Erreichen der ϵ-Grenze werden die Konflikte mit Schreibtransaktionen ignoriert

Gleichzeitig wird auf Knoten B die Lesetransaktion T_Q mit den Aktionen $r^B_{T_Q}(k_1)$, $r^A_{T_Q}(k_2)$ initiiert, die den Kontostand aller Konten (z. B. für statistische Auswertungen) bestimmt. Aufgrund von zeitlichen Verzögerungen wird die Transaktion T_U auf Knoten A vor der Transaktion T_Q ausgeführt, auf dem Knoten B dagegen erst nach T_Q (Schedules auf Knoten A bzw. B: $S_A = \langle w^A_{T_U}(k_1, -200),\ w^A_{T_U}(k_2, +200),\ r^A_{T_Q}(k_2)\rangle$, $S_B = \langle r^B_{T_Q}(k_1),\ w^B_{T_U}(k_1, -200)\rangle$). Die globale Schedule $S = (S_A, S_B)$ ist deshalb nicht 1-Kopie-serialisierbar, da T_Q die Zwischenzustände der Transaktion T_U (alten Kontostand von k_1 auf Knoten B, neuen Kontostand von k_2 auf Knoten A) und damit einen inkonsistenten Datenbankzustand sieht. Nach der Durchführung der Änderungen von T_U auch auf dem Knoten B sind die Kopien jedoch wieder wechselseitig konsistent (Kontostände bei allen Kopien 800 DM bzw. 1200 DM). Falls ein ϵ-Überlappungsparameter größer 1 gewählt wurde, ist deshalb diese globale Schedule eine korrekte epsilon-serialisierbare Schedule, da der (eine) Konflikt zwischen T_U und T_Q auf Knoten A toleriert werden kann.[3]

Das Korrektheitskriterium der Epsilon-Serialisierbarkeit wird beispielsweise in [51] für ein Replikationsverfahren für Echtzeit-Datenbanken eingesetzt, um Schreibtransaktionen weniger häufig durch Konflikte mit Lesetransaktionen zu verzögern. Damit ist ein höherer Durchsatz an Transaktionen in der zur Verfügung stehenden Zeit möglich. Weitere Anwendungsgebiete für die Epsilon-Serialisierbarkeit sind Auskunftssysteme oder Anwendungen für statistische Auswertungen, bei denen begrenzte Ungenauigkeiten für Lesetransaktionen tolerierbar sind.

[3] Statt der Anzahl der Konflikte kann auch die Höhe der Inkonsistenz (z. B. in DM) bei der Tolerierung von Konflikten herangezogen werden (gewichteter ϵ-Überlappungsparameter). Der Konflikt im Beispiel wird dann toleriert, wenn ein ϵ-Überlappungsparameter von mindestens 200 DM festgelegt wurde.

3 Klassifikation von Replikationsverfahren

Die Suche nach einem geeignetem Kompromiß für den in Abbildung 1 skizzierten Zielkonflikt führte in der Literatur zu recht unterschiedlichen Vorschlägen zur Replikationskontrolle. Diese Vorschläge werden nun klassifiziert, indem man anhand der durch die Kopien zusätzlich verursachten Aufgaben Klassifikationkriterien ableitet. Diese Aufgaben sind im einzelnen:

1. Kopienübergreifende Synchronisation der Transaktionszugriffe (Abschnitt 3.1)
2. Aktualisierung der Kopien beim Transaktions-Commit (Abschnitt 3.2)
3. Konsistenzsicherung im Fehlerfall (Abschnitt 3.3)

3.1 Aufgabe 1: kopienübergreifende Synchronisation der Transaktionszugriffe

Da ein Objekt durch mehrere verschiedene Kopien im System repräsentiert wird, genügt es nicht, auf eine korrekte Synchronisation *pro* Kopie zu achten. Stattdessen muß zur Gewährleistung der 1–Kopie–Serialisierbarkeit[4] zusätzlich eine *kopienübergreifende* Synchronisation erfolgen:

Beispiel 2: Flugbuchung
Für eine effizientere Buchung von Sitzplätzen in Flugzeugen sind die noch verfügbaren Plätze eines Flugs über zwei Knoten A und B repliziert. Die Sitzplatzreservierung erfolgt durch die folgende Transaktion:
$T_R(flug, zuBuchendePlaetze)$

```
BEGIN TRANSACTION
 localread(flug, freiePlaetze);
 IF freiePlaetze < zuBuchendePlaetze THEN
  print("nicht genuegend Plaetze frei");
 ELSE
  freiePlaetze := freiePlaetze – zuBuchendePlaetze;
  localwrite(flug, freiePlaetze);
 END
END TRANSACTION
```

Die Transaktion vergleicht also die Zahl der zu buchenden Sitzplätze mit den laut lokaler Kopie noch verfügbaren Plätzen eines Flugs und aktualisiert zuerst nur diese lokale Kopie. Danach wird die Zahl der neu gebuchten Sitzplätze (asynchron) an die anderen Kopien übermittelt. Ein von verschiedenen Knoten gleichzeitig initiiertes Reservieren von Sitzplätzen kann deshalb zu folgendem Verlauf der Transaktionen führen:

Knoten A		Knoten B	
Sitzplätze	Transaktion	Sitzplätze	Transaktion
3		3	
	$T_{R_1}(f,2)$		$T_{R_2}(f,1)$
			$T_{R_3}(f,1)$
1		1	
	$T_{R_2}(f,1)$		$T_{R_1}(f,2)$
	$T_{R_3}(f,1)$		
-1		-1	

Aus der Tabelle ist ersichtlich, daß trotz korrekter Transaktionssynchronisation pro Knoten durch die fehlende knotenübergreifende Synchronisation eine Überbuchung von Flugzeugen möglich ist (*lost-update-Anomalie*). Dadurch kann dieser Zustand nicht in einer zu dieser Datenbank 1–Kopien–äquivalenten Datenbank erreicht werden.

Eine Aufgabe eines Replikationsverfahrens ist es deshalb, für eine korrekte kopienübergreifende Synchronisation zu sorgen. Dazu wurden in der Literatur zwei Ansätze vorgeschlagen:

3.1.1 Absolutistische Verfahren

Beim absolutistischen Ansatz realisiert eine ausgezeichnete Stelle (*primary copy* [52] siehe Abschnitt 4.2.1.1, Besitzer des *tokens* [40], siehe Abschnitt 4.2.1.2, *exclusive writer* [9]) die kopienübergreifende Synchronisation. Will eine Transaktion auf ein Objekt zugreifen, so braucht sie die Zustimmung dieser Stelle.

Damit hängt die Verfügbarkeit eines logischen Objekts jedoch stark von der Verfügbarkeit der ausgezeichneten Stelle ab. Bei „Verlust" dieser Stelle kann diese durch einen Stellvertreter ersetzt werden. Dabei muß sichergestellt werden, daß immer nur eine ausgezeichnete Stelle für jedes Objekt existiert (besonderes, wenn nicht zwischen Knotenabstürzen und Netzwerkfehlern unterschieden werden kann).

3.1.2 Voting Verfahren

Einen „demokratischeren" Ansatz verfolgen die Voting Verfahren, bei denen die Synchronisation der Zugriffe durch *Abstimmung (voting)* erfolgt. Dazu erhält jede Kopie eine bestimmte Anzahl an Stimmen (häufig 1) zugeteilt. Der Zugriff einer Transaktion auf ein logisches Objekt wird nur dann erlaubt, wenn eine entscheidungsfähige Anzahl, ein sog. *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

[4] aber auch für andere Serialisierbarkeitsbegriffe für replizierte Datenbanken (siehe Kapitel 2)

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

Durch die Schreib/Schreib–Überschneidungsregel werden über die gemeinsame(n) Kopie(n) parallele Schreibzugriffe synchronisiert. Außerdem führt die notwendige Zustimmung von mehr als der Hälfte aller Stimmen dazu, daß Schreiben in immer nur einer Partition möglich ist (vgl. Aufgabe 3). Die Schreib/Lese–Überschneidungsregel stellt sicher, daß bei denen im Lesequorum enthaltenen Kopien mindestens eine den aktuellen Wert besitzt. Zur Bestimmung der Kopie mit dem aktuellsten Wert 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 den Zeitstempel erfolgt [32].

Die in der Literatur vorgeschlagenen Voting–Verfahren unterscheiden sich hauptsächlich in der Strukturierung des Quorums und in der Möglichkeit, die Größe des Quorums zu verändern. Die Verfahren lassen sich deshalb bzgl. zwei zueinander orthogonale 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.

3.1.2.1 Unstrukturiertes versus strukturiertes Voting

Bei den unstrukturierten Ansätzen 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 die Kopien in einer logischen Baum– oder Gitterstruktur angeordnet. Zur Erlangung eines Quorums müssen dann entlang dieser Struktur in l Ebenen jeweils s Stimmen[5] gesammelt werden. Die Überschneidungsregeln gelten dann sowohl für die Ebenen als auch für die Stimmen in einer Ebene. Der Vorteil dieser 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.

3.1.2.2 Dynamisches versus statisches Quorum

Wird eine replizierte Datenbank 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. 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 [32]. Damit kann z. B. bei einer zeitweilig hohen Update–Rate das Schreibquorum zu Lasten des Lesequorums verkleinert werden.

3.1.3 Read–One–Copy basierte Verfahren

Als Spezialfall der Voting Verfahren 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 Voting Verfahren 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 zuzugriffen, 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*[6] Methode (siehe z. B. [3]), bei

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

[6] Read One Write All

der alle Kopien synchron aktualisert werden müssen (write all). Dadurch wird der Vorteil des rein lokalen Lesen 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], [24, 25], siehe Abschnitte 4.2.3.1 bzw. 4.2.3.2) oder bei Nicht–Verfügbarkeit einzelner Kopien auf ein anderes Replikationsverfahren umschalten ([23], siehe Abschnitt 4.2.3.3).

Neben der Verfügbarkeit kann auch der Durchsatz des ROWA Ansatzes verbessert werden, in dem man die Lage und die Anzahl der Kopien dynamisch dem Zugriffsverhalten (Schreib/Leseoperationen pro Zeitraum) anpaßt ([33], siehe Abschnitt 4.2.3.5).

3.1.4 Festlegung des Korrektheitskriteriums: Syntaktische versus semantische Verfahren

Unabhängig von der Wahl der kopienübergreifenden Synchronisation (absolutistisch/demokratisch) muß das verwendete Korrektheitskriterium festgelegt werden. Diesbezüglich lassen sich die Replikationsverfahren in zwei Gruppen einteilen:

Bei den *syntaktischen Verfahren* wird die Korrektheit eines Zugriffs einzig über die Reihenfolge der zugreifenden Transaktionen bestimmt, d.h. die Korrektheit wird über die Serialisierbarkeit (z.B. 1–Kopie–Serialisierbarkeit, Epsilon–Serialisierbarkeit) definiert.

Dagegen nützen *semantische Verfahren* die Semantik der Transaktionen, wie z.B. Kommutativität der Operationen [5, 32], bzw. die Semantik der Datenbank [28] 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 ([28], siehe Abschnitt 5.1.2). Daneben existieren aber auch Verfahren, die semantisches Wissen zusätzlich zur Serialisierbarkeit verwenden, um die Verfügbarkeit oder den Parallelitätsgrad der Datenbank zu erhöhen ([5], [38], siehe Abschnitte 5.1.1 bzw. 5.2.1). Im allgemeinen sind die Einsatzmöglichkeiten semantischer Verfahren geringer als diejenigen syntaktischer Ansätze, da die semantischen Methoden die Anwendungssemantik mit berücksichtigen müssen.

3.2 Aufgabe 2: Aktualisierung der Kopien beim Transaktions–Commit

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

Die Zahl der synchronen Änderungen hängt nicht zuletzt von der Wahl des kopienübergreifenden Synchronisationsverfahrens ab. Bei den absolutistischen Verfahren genügt es, die Kopie der ausgezeichneten Stelle synchron zu aktualisieren. Dagegen müssen bei den Voting Verfahren mindestens die Kopien im Schreibquorum synchron aktualisiert werden.

3.3 Aufgabe 3: Konsistenzsicherung im Fehlerfall

3.3.1 Unterstützte Fehlersemantik

Wie in der Einleitung erwähnt, ist eine Erhöhung der Verfügbarkeit im Fehlerfall 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. Kann allerdings z.B. aufgrund der Netztopologie eine Partitionierung ausgeschlossen werden [45], so sind auch Replikati-

onsverfahren geeignet, die Datenkonsistenz nur bei Rechnerabstürzen garantieren.

3.3.2 Behandlung von Partitionen: Optimistische versus pessimistische Verfahren

Sind Partitionierungen möglich, so muß sichergestellt werden, daß eventuelle Inkonsistenzen nicht dauerhaft sind bzw. gar nicht erst auftreten können. Dazu werden in der Literatur zwei grundsätzliche Strategien verfolgt:

Optimistische Verfahren [15, 5, 27] gehen von der Annahme aus, daß Inkonsistenzen nur selten auftreten und daß diese beim Zusammenfügen erkannt und aufgelöst werden können. Deshalb besteht nicht die Notwendigkeit, im Partitionierungsfall den Datenzugriff einzuschränken. Optimistische Verfahren unterscheiden sich deshalb untereinander hauptsächlich in der Art und Weise, wie die aufgetretenen Inkonsistenzen behoben werden.

Pessimistische Strategien [52, 4, 45, 24, 30, 1, 38, 32] 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 zugunsten ihrer Konsistenz einzuschränken. Ungehindertes Arbeiten mit einem Objekt ist damit immer nur in einer Partition möglich, während in der anderen Partition allenfalls ein lesender Zugriff erlaubt wird. Pessimistische Verfahren unterscheiden sich deshalb 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 möglich waren. Diese müssen dann in den anderen Partitionen nachgezogen werden.

3.4 Gegenüberstellung der Klassifikationskriterien

Durch die verschiedenen Aufgaben der Replikationsverwaltung lassen sich die Replikationsverfahren bzgl. zwei zueinander orthogonaler Kriterien (syntaktisch/semantisch, Aufgabe 1, und optimistisch/pessimistisch, Aufgabe 3) klassifizieren und beliebig miteinander kombinieren: syntaktisch–optimistische, syntaktisch–pessimistische, semantisch–optimistische, semantisch–pessimistische Verfahren. Der Bereich der syntaktisch–pessimistischen Verfahren kann weiter in absolutistische, Read–One–Copy und Voting Verfahren aufgeteilt werden. Die Voting Verfahren lassen sich wiederum in zwei zueinander orthogonale Richtungen aufgliedern: unstrukturiert/strukturiert und statisch/dynamisch, woraus sich weitere vier Bereiche ergeben.

Die Abbildung 2 zeigt, in welchen Bereich die verschiedenen Vorschläge zur Replikationskontrolle fallen und gibt jeweils den Abschnitt an, in dem die detailliertere Vorstellung ausgewählter Replikationsverfahren erfolgt.

Da nicht jedes Replikationsmethode alle der hier skizzierten Aufgaben löst, wird bei der folgenden Beschreibung immer angegeben, welche der Einzelaufgaben durch dieses spezielle Verfahren realisiert wird (siehe auch tabellarische Zusammenfassung im Anhang).

4 Syntaktische Verfahren

Nachfolgend werden ausgewählte syntaktische Verfahren genauer vorgestellt.

4.1 Ein optimistisches Verfahren am Beispiel des Optimistischen Protokolls

Das von Davidson [15] vorgeschlagene Verfahren beschränkt sich allein auf die Wiederherstellung der Datenbankkonsistenz nach Partitionierungen (Aufgabe 3). Dazu wird in der Reintegrationsphase ein erweiterter Abhängigkeitsgraph erzeugt, der als Knoten alle Transaktionen enthält, die während der Partitionierung in den einzelnen Partitionen durchgeführt wurden. Die Erweiterung liegt in einer verfeinerten Semantik der Kanten: *Datenkanten* und *Vorrangskanten* drücken Schreib–Lesekonflikte bzw. Lese–Schreibkonflikte innerhalb einer Partition aus. Dagegen werden Lese–Schreibkonflikte zweier Transaktionen aus verschiedenen Partitionen über *Interferenzkanten* beschrieben.

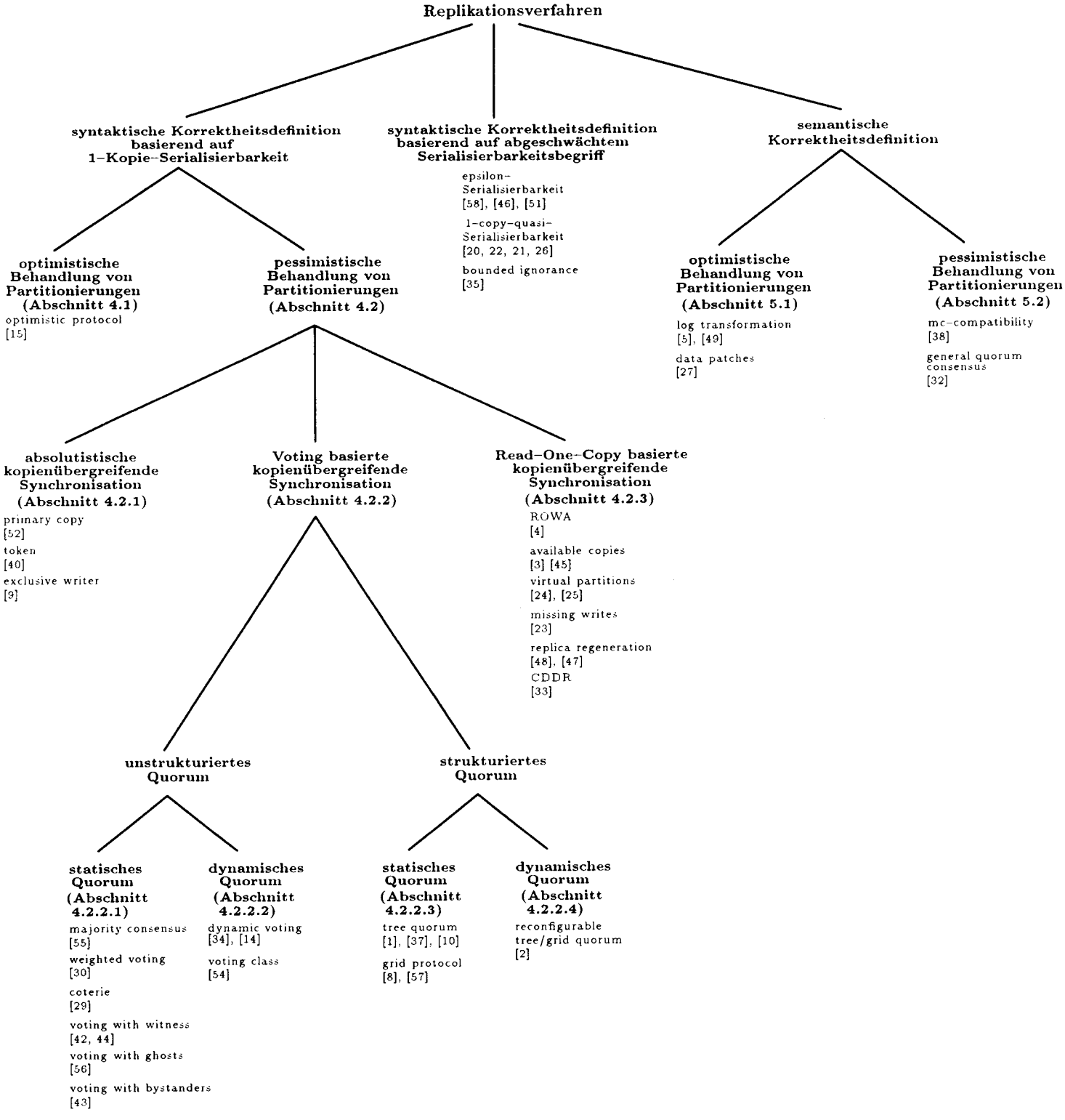


Abbildung 2: Klassifikation der Replikationsverfahren

Zyklen über Interferenzkanten zeigen an, daß sich die Datenbank in einem inkonsistenten Zustand befindet, da während der Partitionierung Transaktionen in nicht 1–Kopie–serialisierbarer Reihenfolge durchgeführt wurden. Zur Wiederherstellung der Datenbankkonsistenz werden die Zyklen durch Zurücksetzen einzelner Transaktionen aufgebrochen[7]. Wird dabei eine Transaktion T_i zurückgesetzt, so müssen auch alle Transaktionen zurückgesetzt werden, die mit T_i über Datenkanten verbunden sind (*kaskading rollback*). Dies ist notwendig, weil sie Datenobjekte gelesen haben, die von T_i verändert wurden.[8] Eine topologische Sortierung des zyklenfreien Graphen gibt dann die Serialisierungsreihenfolge vor, in der die Transaktionen in allen Partitionen durchgeführt werden müssen.

Nach Ansicht der Autorin ist das optimistische Protokoll gut geeignet, wenn Partitionierungen nur von kurzer Dauer sind und selten auftreten, die Anzahl der Änderungstransaktionen während einer Partitionierung gering ist bzw. Lesetransaktionen überwiegen sowie eine maximale Verfügbarkeit wichtiger als das gelegentliche Wiederholen einer bereits bestätigten Transaktion ist.

Zusätzlich kann die Anzahl der Transaktionen, die zurückgesetzt und wiederholt werden müssen, durch Ausnützen von semantischem Wissen (z. B. Kommutativität, siehe Abschnitt 5.1.1) reduziert werden.

4.2 Pessimistische Verfahren

4.2.1 Absolutistische Verfahren

4.2.1.1 Primärkopie Verfahren

Das *Primärkopie Verfahren* [52] beruht auf einem zentralen Sperralgorithmus. Jede Transaktion, die ein Objekt aktualisieren möchte, muß eine Sperranfrage an die *Primärkopie (primary copy)* des Objekts stellen. Dadurch wird auch die kopienübergreifende Transaktionssynchronisation (Aufgabe 1) zentral durch die Primärkopie geregelt. Da sich die Primärkopie nur in einer Partition befinden kann, sind Änderungen in nur dieser Partition möglich. Inkonsistenzen können dadurch nicht auftreten (vgl. Aufgabe 3).

Der Primärkopie Ansatz legt auch das Update–Verhalten (Aufgabe 2) fest. Die Aktualisierung eines Objekts erfolgt hier zweistufig: Beim Commit der Änderungstransaktion wird zuerst nur die Primärkopie aktualisiert. Die Änderung der anderen Kopien erfolgt asynchron, d.h. nach dem Commit, durch die Primärkopie.

Lesetransaktionen können dagegen ohne Zugriff auf die Primärkopie von jeder Kopie lesen. Das Lesen einer beliebigen Kopie beinhaltet allerdings die Gefahr, daß diese aufgrund der asynchronen Änderungen nicht den aktuellsten Wert des logischen Objekts 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 (vgl. Beispiel 1). Das Verfahren garantiert 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 Primärkopie zugegriffen werden (Lesetransaktion als Pseudo–Änderungstransaktion), wodurch der Vorteil des lokalen Lesens im wesentlichen verloren geht.

Der Vorteil des Primärkopie Ansatzes liegt in der einfachen Realisierung des zentralen Sperralgorithmus, der viele der in Kapitel 3 beschriebenen Aufgaben löst. Nachteilig ist jedoch, daß ein Zugriff stark von der Verfügbarkeit der Primärkopie abhängt. Bei Ausfall der Primärkopie sind solange keine Änderungen möglich, bis entweder die Primärkopie wieder verfügbar ist oder diese durch eine andere Kopie ersetzt wurde. Für eine eindeutige Primärkopieersetzung sind alle Kopien aufsteigend durchnumeriert. Aus der Menge der verfügbaren Kopien wird dazu die Kopie mit der kleinsten Nummer als neue Primärkopie ausgewählt. Der skizzierte Algorithmus kann aber nur dann eine neue eindeutige Primärkopie bestimmen, wenn zwischen Partitionierungen und Knotenausfällen unterschieden werden kann. Ist dies

[7] Das „Commit" einer Transaktion während einer Partitionierung ist deshalb nicht endgültig, da sie in der Reintegrationsphase zurückgesetzt werden kann.

[8] Das geeignete Aufbrechen der Zyklen ist ein NP–vollständiges Problem. In [15] wird eine schnellere, aber dafür nur suboptimale Lösung der Komplexität $O(N^{2.81})$ vorgeschlagen.

nicht der Fall, so wählt der Algorithmus für jede Partition getrennt eine neue (andere) Primärkopie aus, wodurch Inkonsistenzen ermöglicht werden.

4.2.1.2 Token–basierte Verfahren

Eine zum Primärkopie Ansatz ähnliche Methode ist das Token–basierte Verfahren von [40]. Statt einer statisch vorgegebenen Primärkopie wird die ausgezeichnete Stelle durch ein *Token* festgelegt. Eine Transaktion muß als Zugriffsberechtigung dieses Token der ausgezeichneten Stelle entziehen. Nach dem Zugriff kann die Transaktion das Token — im Gegensatz zum Primärkopie Ansatz — an eine beliebige andere Kopie weitergeben, die fortan als ausgezeichnete Stelle fungiert. Das (Schreib/Lese–) Token (*exclusive token*) kann in mehrere Lese–Token (*shared token*) umgewandelt werden (und umgekehrt). Durch die Umwandlung in mehrere Lese–Token wird der Lesezugriff erleichtert, schreibender Zugriff ist dann jedoch nicht mehr möglich. In [51] wird ein Token basiertes Verfahren für Echtzeit–Datenbanken verwendet, das als Korrektheitskriterium die Epsilon–Serialisierbarkeit verwendet.

4.2.2 Voting Verfahren

In Kapitel 3 wurden für die Voting Verfahren zwei orthogonale Kriterien (unstrukturiert/strukturiert bzw. statisch/dynamisch) beschrieben, mit denen sich die verschiedenen Verfahren in vier Bereiche einteilen lassen (siehe Abbildung 2). Aus jedem Bereich werden in diesem Abschnitt einzelne Verfahren genauer beschrieben.

4.2.2.1 Unstrukturierte, statische Verfahren

4.2.2.1.1 Majority Consensus Verfahren

Der ursprüngliche Vorschlag eines Voting–Verfahrens, der viele Folgevorschläge inspirierte, ist die *Majority Consensus* Methode von Thomas [55]. Im Gegensatz zu den absolutistischen Ansätzen sind beim Majority Consensus Verfahren alle Kopien gleichberechtigt. Jedes logische Objekt ist vollständig über alle Knoten der Datenbank repliziert, wobei jeder Knoten bei der Abstimmung genau eine Stimme besitzt. Ein Zugriff wird erlaubt, wenn mehr als die Hälfte aller Kopien (= majority) zustimmt. Sowohl für das Lesequorum Q_R als auch für das Schreibquorum Q_W aus Abschnitt 3.1.2 muß eine Mehrheit an Stimmen gesammelt werden.[9] Eine Abstimmung erfolgt entlang eines logischen Rings, in dem alle Kopien angeordnet sind. Die Abstimmungsentscheidung basiert dabei auf Zeitstempeln: Jeder Transaktion und jedem Objekt[10] wird dazu ein Zeitstempel zugeordnet.

Die Änderung einer Transaktion gliedert sich in zwei Phasen: In der ersten Phase liest eine Transaktion von der jeweils lokalen Kopie. Änderungen werden auch nur lokal (und für die anderen Transaktion nicht sichtbar) durchgeführt. In der anschließenden Voting–Phase sendet eine Transaktion eine Abstimmungsaufforderung mit ihrem und dem Zeitstempel der zugegriffen Objekte entlang des logischen Rings an den nächsten Knoten.

Jeder Knoten, der eine Abstimmungsaufforderung erhält, stimmt nach den folgenden vier Regeln ab:

1. Lehne ab, wenn einer der übermittelten Objektzeitstempel veraltet ist (Die Änderungen der Transaktion beruht auf veralteten Werten).
2. Stimme zu und markiere diesen Auftrag als *schwebend (pending)*, wenn alle Zeitstempel aktuell sind und keine Konflikte mit einem anderen schwebenden Auftrag existieren.

Besteht ein Konflikt mit einem anderen, schwebenden Auftrag mit Zeitstempel T, so:

3. Stimme mit Enthaltung, wenn der schwebende Auftrag der Ältere ist. (Die ältere Transaktion hat Vorrang.)
4. Andernfalls verzögere die Abstimmung. Erhält der schwebende Auftrag T das notwendige Quorum, so lehne ab, sonst stimme zu.

Der Knoten schickt seine Abstimmungsentscheidung zusammen mit den Entscheidungen der vorhergehenden Knoten an den nächsten Knoten im Ring weiter. Der Knoten, dessen Zustimmung zur Mehrheit führt, sendet ein globales Commit an alle anderen. Bei diesem Commit müssen dann mindestens die durch die

[9] Diese Kopienmehrheit bestimmt die *Mehrheitspartition*.
[10] Zeitstempel der letzten Änderung

Größe des Schreibquorums vorgegebene Anzahl an Kopien synchron aktualisiert werden, damit in einem späteren Lesequorum wenigstens eine aktuelle Kopie enthalten ist. Lehnt ein Knoten dagegen ab, so erzeugt er einen globalen Abbruch der Transaktion.

Analog zum Primärkopie Ansatz müssen Lesetransaktionen wie (Pseudo-) Schreibtransaktionen behandelt werden, um ein konsistentes Lesen zu gewährleisten.

Das Majority Consensus Verfahren kann als Erweiterung des optimistischen[11] Zeitstempelsynchronisationsverfahren [39] betrachtet werden. Bei beiden Verfahren basiert die Konfliktauflösung auf Zeitstempel-Vergleichen. Zur Einhaltung der 1-Kopie-Serialisierbarkeit müssen jedoch beim Majority Consensus Verfahren die Zeitstempel bei einer Mehrheit aller Kopien übereinstimmen und aktuell sein. Dadurch regelt das Majority Consensus Verfahren — wie alle Voting Verfahren — die transaktionsübergreifende Synchronisation (Aufgabe 1) und verhindert Inkonsistenzen im Fehlerfall (Aufgabe 3).

Die Kombination der Majority Consensus Verfahren mit einem pessimistischen Synchronisationsverfahren ist dagegen weniger geeignet, da dann für jeden Objektzugriff (und nicht für alle zugegriffenen Objekte gemeinsam) eine Voting-Phase gestartet werden müßte.

4.2.2.1.2 Das Weighted Voting Verfahren

Das Majority Consensus Verfahren besitzt zwei Nachteile:

1. Das Abstimmungsverfahren für einen konsistenten Lesezugriff ist gleich aufwendig wie für einen Schreibzugriff, da jeweils eine Mehrheit an Kopien zustimmen muß. Es kann damit nicht bzgl. einer Zugriffsart[12] optimiert werden.
2. Es bietet keine Möglichkeit, besonders sichere oder schnelle Knoten zu bevorzugen.

Mit der von Gifford [30] vorgeschlagenen gewichteten Abstimmung (*weighted voting*) werden beide Nachteile aufgehoben. Bei seinem Verfahren sind die Stimmen für das Schreib- bzw. Lesequorum im Rahmen der Überschneidungsregeln frei wählbar. Soll z.B. das Lesen kostengünstiger gestaltet werden, so kann das Lesequorum Q_R auf Kosten des Schreibquorums Q_W verkleinert werden. Dies kann im Extremfall bis zum ROWA Verfahren (siehe Abschnitt 3.1.3) führen, bei dem zum Lesen nur die Zustimmung *einer* Kopie, zum Schreiben jedoch die Zustimmung (und damit die Verfügbarkeit) *aller* Kopien benötigt wird. Außerdem können beim Verfahren von Gifford durch eine unterschiedliche Verteilung der Stimmen auf die Kopien (*weighted voting*) einzelne Kopien bevorzugt werden. So kann bei Vergabe aller Stimmen auf eine Kopie das weighted voting Verfahren zum Primärkopien Verfahren (siehe Abschnitt 4.2.1.1) degenerieren. Das Weighted Voting Verfahren erhöht außerdem bei gerader Kopienanzahl die Wahrscheinlichkeit, ein Quorum zu erreichen: Wird beispielsweise ein Objekt auf vier Knoten repliziert, so benötigt man mindestens die Zustimmung von drei Knoten. Erhält jedoch ein (besonders ausfallsicherer) Knoten 2 Stimmen, so genügt schon die Zustimmung eines weiteren Knotens zur Erlangung des Quorums [6]. Kriterien für eine geeignete Wahl eines Quorums werden detailliert in [29] beschrieben.

4.2.2.1.3 Abstimmung mit Zeugen, Geistern oder Zuschauern

Da bei einer Leseoperation nur auf eine (aktuelle) Kopie aus dem Lesequorum zugegriffen wird, muß nicht jede Kopie den eigentlichen Datenwert eines logischen Objekts speichern. Dadurch kann insbesondere bei großen Objekten Speicherplatz eingespart werden, ohne die Verfügbarkeit stark einzuschränken. Paris (*Zeugen* [42, 44], *Zuschauer* [43]) sowie von van Renesse und Tanenbaum (*Geister* [56]) schlagen deshalb vor, zwischen Kopienbesitz und Stimmberechtigung zu unterscheiden. Knoten, die als Zeugen, Geister oder Zuschauer fungieren, speichern deshalb nicht eine vollständige Kopie eines logischen Objekts, sondern nur die (Meta-) Informationen, die zur Stimmabgabe notwendig sind. Der Zugriff auf den eigentlichen Datenwert eines Objekts kann dann natürlich nur über einen „vollständigen" Knoten erfolgen. Während Zeugen immer vorhanden sind, wer-

[11] „optimistisch" im Sinne der Synchronisationsverfahren, d.h. die Konfliktauflösung erfolgt erst am Ende der Transaktion und nicht jeweils beim Zugriff auf ein Objekt wie bei pessimistischen Synchronisationsverfahren

[12] z. B. für den in der Praxis viel häufigeren Lesezugriff

den Geister und Zuschauer nur bei Ausfall vollständiger Knoten erzeugt und bei deren Wiederherstellung wieder vernichtet. Einige Nachteile dieser Verfahren werden in [6] kurz dargestellt.

4.2.2.2 Unstrukturierte, dynamische Verfahren am Beispiel des Dynamic Voting Verfahrens

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 3: Probleme bei der Quorumsbildung mit dem Majority Consensus Verfahren
Zerfällt ein nach dem Majority Consensus Verfahren [55] arbeitendes Datenbanksystem mit Replikationsgrad 100 zunächst in zwei Partitionen mit 49 bzw. 51 Knoten und anschließend die „51er–Partition" weiter in 25 und 26 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* [34][13], 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. Die stimmberechtigten Knoten lassen sich über eine *logische Versionsnummer* LN bestimmen, die bei jeder Änderung inkrementiert wird.

Im folgenden wird der Dynamic Voting Algorithmus zur Vereinfachung anhand des Majority Consensus Verfahrens skizziert:

Ein Knoten S sendet eine Abstimmungsaufforderung an alle Knoten, die eine Kopie des zu aktualisierenden Objekts besitzen. Falls ein zur Abstimmung aufgeforderter Knoten S_i zustimmt, so sendet er seine logische Versionsnummer LN_i und die dazu korrespondierende SC_i[14] an den Knoten S zurück. Dieser sammelt bis zum Erreichen einer Zeitschranke die eintreffenden Zustimmungen in der Menge $Voted$ ein und bestimmt dann daraus die Menge der stimmberechtigten Knoten $RightToVote = \{S_i | S_i \in Voted \wedge LN_i = LN_{max}\}$ mit $LN_{max} = max\{LN_i | S_i \in Voted\}$ und die entsprechende Gesamtstimmenzahl SC_{max} ($SC_{max} = SC_i$ für ein $S_i \in RightToVote$). $RightToVote$ umfaßt damit alle Knoten, die an der letzten Änderung teilgenommen haben. Sind mehr als $SC_{max}/2$ Knoten in $RightToVote$ enthalten, so wird die Änderung akzeptiert. Es wird dazu eine neue logische Versionsnummer erzeugt ($LN_{neu} = LN_{max} + 1$) sowie SC_{neu} berechnet ($SC_{neu} = card(Voted)$)[15]. Diese Informationen werden zusätzlich zur eigentlichen Änderung beim globalen Commit an alle Knoten versandt.

Beispiel 4: Quorumsanpassung beim dynamic Voting Verfahren
Erfolgt bei den im Beispiel 3 skizzierten Partitionierungen nach der ersten Partitionierung eine Änderung des Objekts, an dem 26 der 51 stimmberechtigten Knoten zustimmen, so kann auch nach der zweiten Partitionierung erfolgreich ein Quorum gesammelt werden, weil durch die dynamische Anpassung nur noch $26/2 = 13$ Knoten zustimmen müssen.

Das Beispiel zeigt allerdings auch ein Problem auf. Durch fortlaufende Partitionierungen der Mehrheitspartition sind immer weniger Knoten stimmberechtigt. Im Extremfall kann dies zu einer „1–Kopien Mehrheitspartition" führen (weitere Zerlegung der Mehrheitspartition in 7, 4, 2, 1 Kopien). Kommt es zu einer Reintegration der restlichen 99 Kopien, so ist mit diesen kein Arbeiten möglich, da sie solange kein Stimmrecht mehr besitzen, bis sie an einer Änderung mit der letzten stimmberechtigten Kopie teilgenommen haben [34, 6]. Diese Reduzierung der Verfügbarkeit kann gemäß des Vorschlags von Tang [54] dadurch begrenzt werden, daß man eine untere Grenze für die minimale Zahl der Stimmberechtigten festlegt.

[13] Ein sehr ähnlicher Ansatz wurde von [14] vorgeschlagen.

[14] Die Variable SC legt die Gesamtzahl der gerade stimmberechtigten Knoten fest. Sie muß nach jeder Änderung neu bestimmt werden.

[15] Bei gerader Gesamtstimmenanzahl bestimmt ein ausgezeichneter Knoten *distinguished site* die Mehrheitspartition.

4.2.2.3 Strukturierte, statische Verfahren

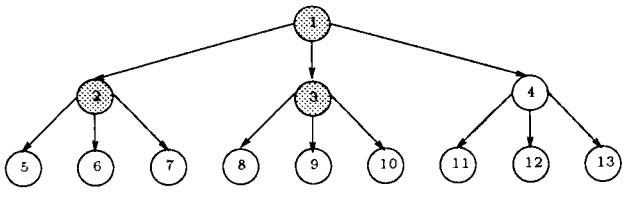
4.2.2.3.1 Das Tree Quorum Verfahren

Ein hoher Replikationsgrad (ab etwa 100 Kopien) 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 [37, 36, 1, 10], so müssen zur Erlangung eines (strukturierten) Quorums bedeutend weniger Knoten angefragt werden (siehe Beispiel 5).

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 5: minimales Quorum der Dimension $\langle 2, 2\rangle$



Ein Quorum der Dimension $\langle 2, 2\rangle$ kann im Beispiel 5 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 3.1.2 für beide Dimensionen gelten:[16]

- 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 5 erfüllt diese Bedingungen.

Im Vergleich zum unstrukturierten Majority Consensus Verfahren müssen jedoch weniger Kopien zur Stimmabgabe aufgefordert werden (minimal 3, maximal 6 anstatt 7), ohne daß die Verfügbarkeit stark reduziert wird [1, 36].

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* ($Q_R = \langle 1, d/2 + 1\rangle$, $Q_W = \langle h, d/2 + 1\rangle$) 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 [10] 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.

Nachteilig bei allen strukturierten Verfahren ist, daß die Knoten, die zur Abstimmung aufgefordert werden, strikt durch die logische Struktur festgelegt sind. Fällt z. B. ein Knoten X aus, so kann seine Stimme nur durch Knoten aus seinem Unterbaum, nicht aber durch Geschwisterknoten (bzw. deren Nachkommen) ersetzt werden. Der Vorteil des geringeren Kommunikationsaufwands wird außerdem durch höheren Ver-

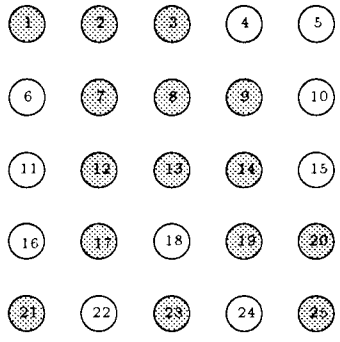
[16] bei jeweils 1 Stimme pro Knoten

waltungsaufwand (logische Struktur) erkauft, so daß sich diese Verfahren nur bei hohem Replikationsgrad lohnen.

4.2.2.3.2 Das Gitter Verfahren

Der „Flaschenhals" Wurzel läßt sich auch vermeiden, wenn statt der Baumstruktur eine Gitterstruktur verwendet wird [8, 2].

Beispiel 6: Gitterstruktur für ein logisches Objekt mit Replikationsgrad 25



Die Überschneidungsregeln müssen wiederum sowohl für die Zeilen als auch für die Spalten gelten:[17]

- Schreib/Schreib–Überschneidungsregel:
  $2 * Q_W.z > \sqrt{n}$ und $2 * Q_W.s > \sqrt{n}$
- Schreib/Lese–Überschneidungsregel:
  $Q_R.z + Q_W.z > \sqrt{n}$ und $Q_W.s + Q_R.s > \sqrt{n}$

Im $\sqrt{read}$–Protokoll [2] wurde für $Q_R = \langle Zeile, Spalte\rangle = \langle 1, \sqrt{n}/2 + 1\rangle$ und $Q_W = \langle \sqrt{n}, \sqrt{n}/2 + 1\rangle$ gewählt (bzgl. der Zeilen wird das ROWA–Protokoll, bzgl. der Spalten das Majority Consensus Verfahren eingesetzt). Damit kann ein Schreibquorum erlangt werden, solange in jeder Spalte über die Hälfte aller Kopien verfügbar sind. Diese Einteilung ermöglicht ein zum ROWA Verfahren vergleichbar günstiges Lesen (3 Kopien im Beispiel 6), ohne daß das Schreiben von der Verfügbarkeit einer einzelnen Kopie abhängt. Damit die Schreibverfügbarkeit ähnlich gut wie beim Majority Consensus Verfahren [2]. Im Beispiel 6 wurde ein gültiges Schreibquorum für das $\sqrt{read}$–Protokoll markiert.

[17] für ein Gitter mit jeweils n Zeilen und Spalten sowie 1 Stimme pro Knoten

4.2.2.4 Strukturierte, dynamische Verfahren am Beispiel des Tree Quorum Verfahrens mit rekonfigurierbarer Baumstruktur

Die bisherigen Vorschläge zur Verbesserung der Schreibverfügbarkeit beim *Read Root* Verfahren (andere Struktur: Gitter, Wald von Bäumen, andere Quorumsverteilung, siehe Abschnitte 4.2.2.3.1 und 4.2.2.3.2) verteuern unnötigerweise den Lesezugriff im fehlerfreien Fall. Deshalb wird in [2] 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 ($Q_{reconfig} = \langle h/2 + 1, d/2 + 1\rangle$, Abschnitt 4.2.2.3.1) 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. $Q_r = \langle 2, d/2 + 1\rangle$, $Q_w = \langle h - 1, d/2 + 1\rangle$) arbeitet.

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

4.2.3 Read–One–Copy basierte Verfahren

Die Basis aller Read–One–Copy Verfahren, die ROWA Methode, wurde schon in Abschnitt 3.1.3 skizziert. Hier werden nun einige Folgevorschläge beschrieben.

4.2.3.1 Das Available Copies 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*, Aufgabe 2). Welche Kopien verfügbar sind, wird in replizierten *Verzeichnissen (directories)* gespeichert. Vor jedem Lese/Schreibzugriff wird das lokale Verzeichnis konsultiert, um die gegenwärtig verfügbaren Kopien 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, 45]. Außerdem toleriert dieser Ansatz nur Knoten- und keine Netzausfälle (vgl. Aufgabe 3), so daß es nur in Systemen eingesetzt werden kann, bei denen Partitionierungen nicht möglich sind.[18] In [45] werden durch Hinzunahme von Zeugen (siehe Abschnitt 4.2.2.1.3) und unter Verwendung einer geeigneten Netztopologie (z.B. Ethernet–Vernetzung mit Gateway–Rechnern) die Kopien in partitionsfreie Segmente eingeteilt, so daß im Hauptsegment das Available Copies Verfahren eingesetzt werden kann.

[18] Netzpartionierungen können aufgrund unterschiedlicher Verzeichniseinträge zu Inkonsistenzen führen [50].

4.2.3.2 Virtual Partitions

Das *Virtual Partition Verfahren* [24] kann als Erweiterung des *Available Copies Ansatzes* [3] betrachtet werden. Jeder Knoten verwaltet die Menge der aus seiner Sicht erreichbaren Knoten in seiner *View*. Damit kann innerhalb einer View wiederum die ROWA Methode angewandt werden. Durch die zusätzliche Einschränkung, daß ein Zugriff eines Knotens nur erlaubt wird, wenn seine View die Mehrheit aller Knoten enthält, kann das Verfahren jedoch auch Partitionen tolerieren (vgl. Aufgabe 3). Dazu müssen alle Knoten die gleiche View besitzen, weil unterschiedliche Views zu Inkonsistenzen führen können (siehe Abbildung 3).

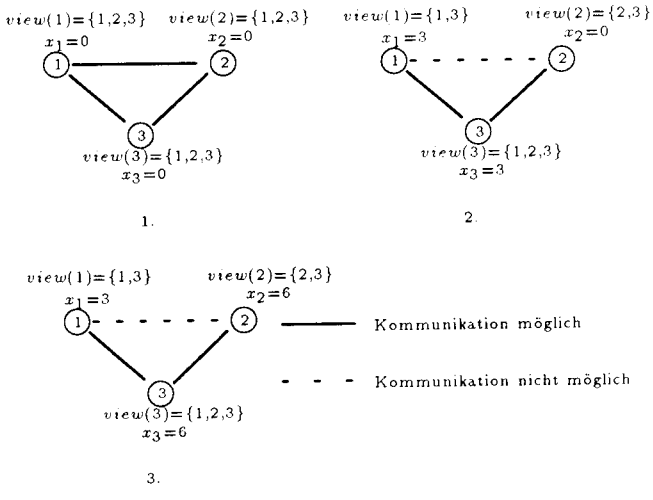


Abbildung 3: Das Objekt x sei über die drei Knoten 1, 2 und 3 repliziert. Da alle Knoten miteinander kommunizieren können, besteht bei allen dreien die View aus den Knoten 1, 2 und 3 (siehe 1.). Aufgrund einer Verbindungsunterbrechung können die Knoten 1 und 2 nicht mehr miteinander kommunizieren; sie passen ihre Views entsprechend an (siehe 2.). Eine durch den Knoten 1 initiierte Kopienänderung aktualisiert alle Knoten in dessen View ($x_1 = x_3 = 3$, siehe 2.). Eine danach auf Knoten 2 initiierte Schreibtransaktion ($x := x + 6$) liest von der veralteten Kopie x_2 und aktualisiert basierend auf diesem Wert die Knoten 2 und 3 ($x_2 = x_3 = 6$). Damit geht die vorherige Änderung verloren (siehe 3.).

Da sich durch reale Partitionen unterschiedliche, sich widersprechende Views nicht vermeiden lassen, werden als Abstraktion *virtuelle Partitionen* eingeführt

und auf dieser Ebene das ROWA Verfahren angewandt. Eine virtuelle Partition besteht aus einer Menge von Knoten, die sich alle auf eine gleiche View geeinigt haben. Im Gegensatz zu realen Partitionen entstehen virtuelle Partitionen nicht „willkürlich", sondern werden nach einem von [24] vorgeschlagenen Protokoll erzeugt, in dessen Verlauf auch die gemeinsame View festgelegt wird:

Das virtual Partition Protokoll

Idealerweise spiegelt eine virtuelle Partition die reale Partitionierungssituation wieder. Erkennt ein Knoten, daß seine View nicht mehr den realen Partitionen entspricht,[19] so initiiert er eine neue virtuelle Partition, in dem er eine neue virtuelle Partitionsnummer vP_1 erzeugt. Er sendet dann an alle Knoten der verteilten Datenbank die Aufforderung, in diese virtuelle Partition vP_1 einzutreten. Anhand der erhaltenen Antworten bestimmt er die View der neuen virtuellen Partition vP_1 und sendet diese an alle beigetretenen Knoten, die diese übernehmen.

Durch Ablehnung von Änderungsaufforderungen aus anderen virtuellen Partitionen verhindert dieses Verfahren auch durch Partitionierungen verursachte Inkonsistenzen und löst damit Aufgabe 3 aus Kapitel 3. Die in Abbildung 3 gezeigten Inkonsistenzen sind dadurch nicht mehr möglich (siehe Abbildung 4).

Kritisch bei diesem Verfahren ist, daß viele kleine nicht mehrheitsfähige virtuelle Partitionen entstehen können, wodurch in keiner dieser virtuellen Partitionen ein Zugriff auf die Kopien mehr möglich ist. Auch verhindert dieses Verfahren nicht, daß Transaktionen aufgrund nicht mehr aktueller Views veraltete Werte lesen können (running in the past, [3]). Views können jedoch von Zeit zu Zeit durch *probe messages* aktualisiert werden.

In einem Folgepapier [25] wird das Verfahren erweitert, so daß in einer virtuellen Partition statt der ROWA Methode ein beliebiges (und in jeder Partition ein anderes) Voting Verfahren verwendet werden kann, wodurch höhere Flexibilität möglich wird.

[19] z. B. weil eine Schreibaufforderung an einen Knoten fehlschlägt

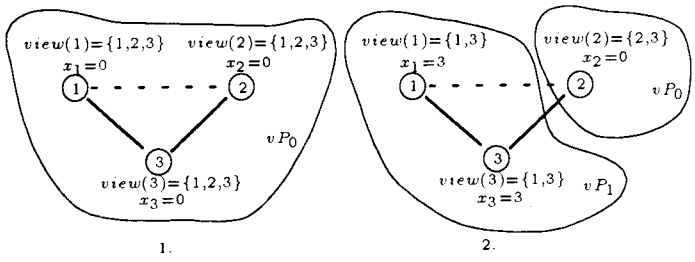


Abbildung 4: Zu Beginn der in Abbildung 3 gezeigten Verbindungsunterbrechung existiert nur die virtuelle Partition vP_0 (siehe 1.). Erkennt beispielsweise Knoten 1, daß diese nicht mehr den realen Partitionen entspricht, erzeugt er eine neue virtuelle Partition vP_1 und fordert die anderen Knoten zum Eintreten auf. Knoten 3 tritt bei und erhält zur Bestätigung die View $view(1) = \{1, 3\}$ zugeteilt (siehe 2.). Die spätere Änderungsaufforderung des Knotens 2 lehnt er ab, da diese aus einer anderen virtuellen Partition (vP_0) stammt.

4.2.3.3 Das Missing Update Verfahren

Eine andere Möglichkeit zur Verbesserung der Schreibverfügbarkeit beim ROWA Ansatz bietet das *Missing Update Verfahren* [23]. Dort wird nur im fehlerfreien Fall (*Normalmodus*) das ROWA Verfahren verwendet. Treten Fehler auf (vgl. Aufgabe 3), so wird in den *Fehlermodus* umgeschaltet, wo ein fehlertoleranteres Voting Verfahren[20] zum Einsatz kommt.

Eine Transaktion startet bei diesem Vorschlag im Normalmodus. Kann sie dort eine Kopie nicht aktualisieren (*missing update*) oder erkennt sie an einer *missing update Markierung* (siehe unten), daß eine Transaktion in ihrer Vergangenheit[21] eine Kopie nicht aktualisieren konnte, wird sie abgebrochen und im Fehlermodus neu gestartet. Dort kann sie dann eventuell mittels der Voting Methode beendet werden. Die Missing Updates führen jedoch zu Kopien unterschiedlichen Aktualitätsgrads. Um dabei Änderungsoperationen, die auf veralteten Kopien basieren, zu verhindern, müssen auch alle „Folgetransaktionen" von T über die missing Updates informiert werden, damit auch sie im Fehlermodus arbeiten.[22]

[20] Majority Consensus Verfahren, siehe Abschnitt 4.2.2.1.1

[21] genauer: eine Transaktion, die im Abhängigkeitsgraphen vor T steht

[22] Nur das Lesen aller Kopien im Quorum stellt sicher, daß auch die aktuellste Kopie gefunden wird.

Da man im voraus nicht weiß, welche Transaktionen einer Transaktion T nachfolgen werden, erfolgt das Informieren über eine Indirektionsstufe: Können bei einem Objekt nicht alle Kopien geändert werden, so wird bei den aktualisierten Kopien, eine *missing update Markierung* hinterlassen. Greift eine im Normalmodus laufende Transaktion auf eine so markierte Kopie zu, wechselt auch sie in den Fehlermodus und markiert ihrerseits bei erfolgreicher Beendigung alle zugegriffenen Objekte mit einer missing update Markierung. Eine solche Markierung bleibt solange erhalten, bis die missing Updates auf allen Kopien nachgezogen wurden.

Der Vorteil dieses Verfahrens besteht darin, daß im fehlerfreien Fall kein zusätzlicher Verwaltungsaufwand benötigt wird [16]. Zusätzlicher Aufwand entsteht nur im Fehlerfall durch die Verwaltung der missing Update Informationen, wobei diese schnell anwachsen können [23, 16]. Außerdem kann sich durch die Propagierung der missing Updates das System „aufschaukeln", so daß fast alle Transaktionen im aufwendigeren Fehlermodus ablaufen müssen [50].

4.2.3.4 Replikationsregeneration

Um beim ROWA Ansatz ein Blockieren zu verhindern, wird bei diesem Verfahren bei einer Änderung — ähnlich zur *Available Copies Methode* (siehe Abschnitt 4.2.3.1) — eine nicht erreichbare Kopie ignoriert. Im Gegensatz zum Available Copies Ansatz werden die fehlenden Änderungen jedoch später nicht nachgezogen, sondern der nicht verfügbare Knoten wird aus der *Menge der aktuellen Knoten* gestrichen [48].[23] Unterschreitet die Zahl der aktuellen Kopien eine untere Grenze und gefährdet damit die Verfügbarkeit des Objekts, so wird ein „freier" Knoten ausgewählt und ihm eine neue aktuelle Kopie zugeteilt (*Regeneration*, [48, 47]), um die Kopie auf dem nicht erreichbaren Knoten zu ersetzen.

Analog zum Available Copies Verfahren können Partitionierungen zu Inkonsistenzen führen [7]. Die Verhinderung von Inkonsistenzen im Fehlerfall (Aufgabe 3) kann dieses Verfahren in dieser Form nur für partionsfreie Systeme erfüllen.

4.2.3.5 Das CDDR Verfahren

Die geeignete Zahl an Kopien für ein Objekt läßt sich *statisch* nur ungenügend festlegen. Für häufige Leseanfragen sind viele Kopien sinnvoll, da dann meist auf die lokale Kopie zugegriffen werden kann. Überwiegt dagegen die Zahl der Schreibzugriffe, so verursachen viele Kopien einen unnötig hohen Änderungsaufwand (vgl. Abbildung 1). Zur Steigerung des Durchsatzes kann deshalb beim auf der ROWA Methode basierenden *Competitive Dynamic Data Replication (CDDR) Verfahren* [33] die Zahl der Kopien *dynamisch* dem Zugriffsverhalten angepaßt werden.

Dazu werden Knoten der replizierten Datenbank in drei Kategorien eingeteilt:

- kopienhaltenden Knoten (data sites)
- kopienlose Knoten (non-data sites)
- Primärknoten, an den die non-data Knoten ihre Schreib/Leseanforderungen stellen und der ihr Zugriffsverhalten überwacht.

Die Verwaltung erfolgt für jedes Objekt über zwei Zähler (read-Zähler = Zahl der von diesem Knoten initiierten Leseanfragen, r-write-Zähler = Anzahl der Schreibaufforderung von anderen Knoten).

Initiiert ein non-data Knoten zu viele Leseanforderungen (sein read-Zähler überschreitet eine obere Grenze), so wird er vom Primärknoten aufgefordert, zum lokalen Lesen selbst eine Kopie vor Ort zu halten. Der non-data Knoten wird dadurch zu einem data Knoten. Erhält ein data Knoten zu viele Änderungsaufforderungen von anderen Knoten, ohne seine lokale Kopie für Lesezugriffe zu verwenden, so gibt er die lokale Kopie auf, um Schreibzugriffe kostengünstiger zu gestalten.

Durch viele Schreibaufforderungen[24] kann sich die Zahl der Kopien bis auf eine reduzieren. Zur Aufrechterhaltung der Verfügbarkeit kann eine Minimalzahl an kopienhaltenden Knoten angegeben werden (vgl. Abschnitt 4.2.3.4). Wird diese unterschritten, so ist es einem data Knoten nicht mehr erlaubt, seine Kopie aufzugeben.

[23] Ein ausgefallener Knoten löscht deshalb während der Recovery alle seine Kopien und trägt sich in die *Menge der freien Knoten* ein.

[24] insbesondere, wenn diese nur von einem Knoten stammen

Zur Transaktionssynchronisation (Aufgabe 1) wird in diesem Papier ein modizifiertes Available Copy Verfahren ([3], siehe auch Abschnitt 4.2.3.1) verwendet, bei dem die kopienlosen Knoten — im Gegensatz zu den kopienhaltenden Knoten — die Zustimmung des Primärknotens benötigen.

Mit dieser Synchronisationsmethode kann das Verfahren nur Knotenfehler und keine Partitionierungen (Aufgabe 3) tolerieren. Die Idee der dynamischen Anpassung der Kopien an sich veränderte Zugriffsprofile ist jedoch auch mit anderen Synchronisationsmethoden kombinierbar (vgl. Abschnitt 5.2.2).

5 Semantische Verfahren

Die in diesem Kapitel beschriebenen Verfahren nützen semantisches Wissen aus, um die in Kapitel 3 angesprochenen Aufgaben lösen zu können.

5.1 Optimistische Verfahren

5.1.1 Log Transformation

Der von Blaustein, Kaufman und Sarin [49, 5] vorgeschlagene Ansatz ist vergleichbar mit dem optimistischen Protokoll von [15] (siehe Abschnitt 4.1). Auch hier ist während einer Partitionierung uneingeschränktes Arbeiten in allen Partitionen möglich. Die Auflösung der während einer Partitionierung möglichen Inkonsistenzen (Aufgabe 3) wird bei beiden Verfahren durch Zurücksetzen und Wiederholen von Transaktionen realisiert (*merge log*). Die beiden Verfahren unterscheiden sich darin, daß bei [49, 5] versucht wird, dieses merge log durch Ausnutzen von semantischen Eigenschaften der Transaktionen zu verkleinern (*log transformation*).

Zur Transformation des merge logs schlagen die Autoren unter anderem die folgenden semantischen Regeln vor:

1. Vertauschen benachbarter Transaktionen, wenn diese kommutativ sind.
2. Löschen einer Transaktion, wenn die von ihr ausgelösten Änderungen komplett von der folgenden Transaktion überschrieben werden.
3. Löschen zweier benachbarter Transaktionen, falls die zweite die Rücksetzung der ersten bewirkt.

Das Vertauschen mittels der Regel 1 dient dazu, merge logs herzustellen, die sich durch Anwendung der Regeln 2 und 3 verkürzen lassen.

Die folgenden Schritte skizzieren das Vorgehen des Verfahrens bei der log Transformation:

1. Eine Transaktion aktualisiert zuerst nur die lokale Kopie. Das Ändern der restlichen Kopien erfolgt dagegen asynchron (Aufgabe 2). Als Synchronisationskriterium wird dabei der Zeitstempel der Transaktion verwendet.
2. Trifft eine (asynchrone) Änderungstransaktion verspätet auf einem Knoten ein, so wird zuerst ein *initial merge log* erstellt, das aus den Rücksetzungen aller Änderungstransaktionen mit höheren Zeitstempeln und der Durchführung der Transaktionen in Zeitstempel–Reihenfolge besteht.
3. Dieses initial merge log wird mit Hilfe der obigen semantischen Regeln optimiert. Man erhält ein *final merge log.*
4. Das hoffentlich kürzere final merge log wird auf die Kopie des Knotens angewendet.

Das Finden der geeigneten Transformationen ist im allgemeinen schwierig. In [5] wird dafür eine Methode vorgeschlagen, bei der die richtigen Transformationen mit Hilfe von Graphen bestimmt werden.

Das Verfahren kann prinzipiell auch die Aufgabe der Transaktionssynchronisation im fehlerfreien Fall übernehmen (Aufgabe 1). Allerdings spiegelt eine Kopie auch im fehlerfreien Fall aufgrund der asynchronen Aktualisierung nicht zu jedem Zeitpunkt alle Änderungen wieder. Dadurch kann eine Änderungstransaktion auf veralteten Kopien basieren, so daß sie beim Eintreffen der noch fehlenden Änderungen zurückgesetzt und wiederholt werden muß, falls sie sich nach Anwendung der Transformationsregeln im final merge log befindet.

Ein weiterer Nachteil ist, daß das Erzeugen eines final merge logs für jede zu spät eintreffende Transaktion aufwendig und damit in der Praxis nicht unbedingt für Synchronisation im fehlerfreien Fall geeignet ist.

5.1.2 Das Data–Patch Verfahren

Das Aufgabengebiet des *Data–Patch Verfahrens* [27] beschränkt sich auf die Wiederherstellung eines konsistenten Datenbankzustands nach Partitionierungen (Aufgabe 3). Die Konsistenzfindung erfolgt hier aber nicht über die Analyse der durchgeführten Transaktionen (siehe Abschnitte 4.1, 5.1.1). Stattdessen 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, dann 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 7: Konto–Relation bei einer Bank
Die Schemadefinition bei der Konto–Relation einer Bank besteht vereinfachend aus:

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

Wird also 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 aber die Transaktionsverarbeitung natürlich auch während Partitionierungen Auswirkungen auf die reale Welt hat (siehe folgendes Beispiel einer Geldauszahlung an einen Bankkunden), ist das Erreichen dieses serialisierbaren Datenbankzustands nicht von allergrößter Wichtigkeit.

Beispiel 8: Auszahlungstransaktion mit Integritätsregel
Eine Auszahlungstransaktion T_A darf das Konto eines Kunden nicht überziehen:

```
T_A(konto, auszahlung)
BEGIN TRANSACTION
  read(konto, Kontostand);
  IF Kontostand < auszahlung THEN
    print("nicht ausreichend Geld auf Konto");
  ELSE
    Kontostand := Kontostand – auszahlung;
    write(konto, Kontostand);
    <Zahle Betrag auszahlung aus>
  END
END TRANSACTION
```

Wird in zwei verschiedenen Partitionen jeweils das gesamte Guthaben abgehoben, so führt dies aufgrund fehlender Informationen aus der anderen Partition zu einen negativen Kontostand und damit zur Verletzung der Integritätsbedingung. Die Wiederherstellung dieser Integritätsbedingung (Kontostand ≥ 0 DM) und damit eines serialisierbaren Da-

tenbankzustands entspricht aber nicht mehr der Realität, da tatsächlich zweimal das gesamte Guthaben an den Kunden ausgezahlt wurde.

5.2 Pessimistische Verfahren

5.2.1 Die Multi–Copy Kompatibilität

Ein Ziel bei der Replikation von Objekten besteht darin, den Zugriff auf diese Objekte zu beschleunigen. Dabei muß ein guter Kompromiß zwischen lesenden und schreibenden Zugriffen gefunden werden. Häufig wird jedoch die eine Zugriffsart auf Kosten der anderen optimiert: Beispielsweise wird beim ROWA Verfahren (siehe Abschnitt 3.1.3) das billige Lesen durch teures und ausfallgefährdendes Schreiben erkauft. Das Majority Consensus Verfahren (siehe Abschnitt 4.2.2.1.1) dagegen verteuert die Leseoperationen, um das Schreiben zu erleichtern.

Die von Kumar und Stonebraker [38] vorgeschlagene Methode der *Multi–Copy Kompatibilität (mc–Kompatibilität, mc–compatibility)* nützt die Kommutativität einzelner Transaktionen aus, um billiges Lesen *und* Schreiben zu ermöglichen. Dazu werden die Transaktionen in eine kommutative (*c-type*) und eine nicht kommutative (*nc-type*) Klasse eingeteilt. Transaktionen aus der kommutativen Klasse können dadurch in unterschiedlicher Reihenfolge auf den einzelnen Kopien durchgeführt werden, ohne die Datenbankkonsistenz zu gefährden. Eine kopienübergreifende Synchronisation (Aufgabe 1) ist deshalb für diese Klasse nicht notwendig.

Bei den nicht kommutativen Transaktionen wird angenommen, daß sie nach der Berechnung des neuen Objektwerts auf einer Kopie durch eine kommutative Transaktion ersetzt werden kann, die dann die anderen Kopien auf den aktuellen Stand bringt (*mc–Kompatibilität*).

Beispiel 9: Beispiel für eine mc–kompatiblen Transaktion
Bei einer Bank sind alle Konten über die Knoten A, B und C repliziert. Zur Berechnung des Zinses Z wird die folgende nicht kommutative Transaktion $T_Z(k,p) = k + pk$ verwendet, die den Zins $Z = pk$ berechnet und auf dem Konto gutschreibt. Statt diese Transaktion auf allen Kopien durchzuführen, wird sie beispielsweise nur auf der Kopie A angewendet und dort der Zins berechnet. Auf den anderen Kopien wird sie dagegen durch die zu ihr *mc–kompatiblen* Einzahlungstransaktion $T_E(k,Z) = k + Z$ ersetzt und $T_E(k,Z)$ ausgeführt. Diese Transaktion ist kommutativ und kann deshalb auf den Kopien B und C in der Reihenfolge beliebig mit anderen kommutativen Transaktionen vertauscht werden.

Da die Serialisierungsreihenfolge nicht kommutativer Transaktionen auf den verschiedenen Kopien eingehalten werden muß, werden diese durch das Majority Consensus Verfahren (siehe Abschnitt 4.2.2.1.1)[25] synchronisiert. Daraus ergibt sich die folgende Arbeitsweise des Verfahrens.

Auf jedem Knoten wird ein Zustandsvektor (nc_i, c_i) verwaltet, wobei nc_i und c_i die Anzahl der nicht kommutativen bzw. kommutativen Transaktionen bezeichnet, die auf dem Knoten i beendet wurden. Der Algorithmus unterscheidet sich für kommutative und nicht kommutative Transaktionen.

Algorithmus für kommutative Transaktionen

1. Führe Änderung auf der lokalen Kopie inklusive Commit durch.
2. Aktualisiere c_i
3. Übergebe nach dem Transaktions–Commit die Transaktion zur Weiterleitung und asynchronen Durchführung auf den anderen Knoten an das lokale Spoolprogramm (Änderungsauftrag).

Algorithmus für nicht kommutative Transaktionen

1. Sammle ein Quorum durch Sperrung einer Mehrheit von Kopien.
2. Wähle die Kopie mit dem höchsten nc–Wert. Dort wurden die meisten nicht kommutativen Änderungen durchgeführt.
3. Führe auf dieser Kopie die Änderung mit der nicht kommutativen Transaktion durch.
4. Bestimme die kommutative Transaktion T_{mc}, die zu dieser nicht kommutativen Transaktion mc–kompatibel ist.
5. Führe T_{mc} auf allen Kopien im Quorum durch.

[25] Dabei ist auch jedes andere Verfahren, das den Zugriff auf unterschiedliche Kopien synchronisiert möglich.

6. Aktualisiere die nc–Werte aller Kopien im Quorum.
7. Gib die Sperren frei.
8. Übergib T_{mc} an das Spoolprogramm zum asynchronen Aktualisieren der restlichen Kopien.

Beispiel 10 veranschaulicht die Algorithmen.

Beispiel 10: (Fortsetzung von Beispiel 9)
Beim Konto $k4$ der Familie Maier wird der Zins berechnet. Dafür wird auf Knoten B die nicht kommutative Zinsberechnungstransaktion $T_Z(k,p)$ gestartet. Kurz davor hat die Frau Maier über den Knoten C 1000 DM auf das Konto $k4$ eingezahlt. Dies führte zu der kommutativen Transaktion $T_E(k,e)$. Tabelle 2 zeigt einen möglichen zeitlichen Verlauf.

Das Beispiel zeigt, daß aufgrund der asynchronen Änderungen der kommutativen Transaktionen — wie beim Log Transformation Verfahren (siehe Abschnitt 5.1.1) — zeitweise unterschiedliche Versionen von einem Datenobjekt existieren. Deshalb wurde der im Beispiel von Frau Maier über den Knoten C eingezahlte Betrag bei der Zinsberechnung nicht mehr berücksichtigt.
Zur Abschwächung von Problemen dieser Art erhalten die kommutativen Transaktionen des Spoolprogramms eine hohe Priorität, damit sie möglichst bald ausgeführt werden. Außerdem werden periodisch Synchronisationsphasen eingebaut, in der alle noch ausstehenden asynchronen Änderungen durchgeführt werden. Während einer Synchronisationsphase werden allerdings keine weiteren Transaktionen angenommen (Parallelitätsverlust!). Nach einer Synchronisationsphase sind damit alle Kopien wieder wechselseitig konsistent. Die Häufigkeit der Synchronisationsphasen ist dabei abhängig von den Konsistenzanforderungen der Anwendungen.

Wegen der Kommutativität und der mc–Kompatibilität müssen bei kommutativen Transaktionen nur die lokale Kopie und bei nicht kommutativen Transaktionen nur die Kopien im Quorum synchron aktualisiert werden. Alle anderen Kopien werden erst nach dem Commit einer Transaktion asynchron aktualisiert (vgl. Aufgabe 2). Dadurch können kommutative Transaktionen unabhängig von Knotenausfällen oder Partitionierungen durchgeführt werden. Lesender Zugriff erfolgt immer rein lokal. Der Durchsatzgewinn des Verfahrens ist natürlich abhängig vom Anteil der kommutativen Transaktionen am Gesamtvolumen der Transaktionen. Bei Anwendungen, die einen hohen Anteil an kommutativen Transaktionen besitzen und bei denen die zeitweisen Inkonsistenzen tolerierbar sind, erscheint dieses Verfahren durchaus attraktiv.

5.2.2 Das General Quorum Consensus Verfahren

Wie schon in Abschnitt 3.1.2 erwähnt, kann durch die Verwendung von Zeitstempeln die Schreib/Schreib–Überschneidungsregel entfallen. Dadurch ist ein Schreibquorum mit weniger als der Hälfte aller Stimmen möglich ($Q_W \cap Q_W = \emptyset$), wodurch sich bei n Kopien die Zahl der Möglichkeiten zur Quorumseinteilung von $n/2$ auf n erhöht (Extremfälle: ROWA für optimale Leseverfügbarkeit und RAWO[26] für optimale Schreibverfügbarkeit). Aufgrund der immer noch notwendigen Schreib/Lese–Überschneidungsregel führt dies jedoch bei realistischen Anwendungen nicht zu einer Erhöhung der Verfügbarkeit von Änderungsoperationen, da in der Regel vor jeder Objektänderung ein lesender Zugriff auf das Objekt erfolgt. (Die höchste Verfügbarkeit von realisistischen Änderungsoperationen erhält man bei einer Majority–Verteilung des Lese– und Schreibquorums.)

Beim *General Quorum Consensus* Verfahren [32] kann für jede Änderungsoperation individuell und ihrer jeweiligen Semantik angepaßt das Lesequorum, genannt *Init Quorum (IQ)*, sowie das Schreibquorum, genannt *Final Quorum (FQ)*, festgelegt werden. Herlihy [32] hat dazu Giffords Voting Verfahren ([30], siehe Abschnitt 4.2.2.1.2) auf abstrakte Datentypen (ADT) erweitert, indem er die Lese– und Schreiboperation im Verfahren von Gifford durch die (Änderungs–) Operationen des ADTs ersetzt und für jede ADT–Operation ein geeignetes Init Quorum bzw. Final Quorum angibt.

Ein ADT `Queue` besitzt beispielsweise die Änderungsoperationen `Enq(x)` und `x=Deq()` zum Anhängen bzw. Entfernen von Elementen, wobei aufgrund ihrer Semantik die Änderungsoperation `Enq` durch-

[26] Read All Write One

Knoten A			Knoten B			Knoten C	
Konto		Transaktion	Konto		Transaktion	Konto	Transaktion
100			100			100	
							$T_E^C(k4, 1000)$
						1100	
X	100		X	100	$T_Z^B(k4, 10\%)$		
X			X	110			
X		$T_E^B(k4, 10)$	X				
X	110		X	110			
		$T_E^C(k4, 1000)$			$T_E^C(k4, 1000)$		
1110			1110				
							$T_E^B(k4, 10)$
						1110	

Tabelle 2: zeitlicher Verlauf der Durchführung einer kommutativen und einer nicht kommutativen Transaktion beim Multi-Copy Kompatibilitätsverfahren (Der hochgestellte Index bei den Transaktionen gibt an, auf welchem Knoten die Transaktion initiiert wurde.)

Da die Einzahlungstransaktion $T_E^C(k4, 1000)$ kommutativ ist, wird sie zuerst rein lokal auf dem Knoten C durchgeführt. Sie liest den gegenwärtigen Kontostand der Kopie, addiert 1000 DM und legt den neuen Wert wieder in der lokalen Kopie ab. Nach dem Commit wird an das Spoolprogramm auf C der Änderungsauftrag gegeben, diese Einzahlungstransaktion auch an die anderen Knoten weiterzuleiten. Bevor das Spoolprogramm auf Knoten B diesen Änderungsauftrag erhält, wird dort die Zinsberechnungstransaktion $T_Z^B(k4, 10\%)$ initiiert. Diese nicht kommutative Transaktion sammelt ein Quorum durch Sperren der Kopien A und B (gekennzeichnet durch X). Dann wird auf Knoten B der Zins berechnet und auf der lokalen Kopie gutgeschrieben. Die zu T_Z^B mc-kompatible Transaktion $T_E^B(k4, 10)$ wird an die übrigen Knoten im Quorum (hier: A) übermittelt und dort durchgeführt. Nach Freigabe der Sperren wird diese Transaktion durch das Spoolprogramm auch an die restlichen Kopien (hier: C) gesendet. Durch die zeitlichen Verzögerungen im Netz kommt es dazu, daß auf Knoten A und C die Einzahlungstransaktionen T_E^C und T_E^B in unterschiedlicher Reihenfolge durchgeführt werden. Wegen der Kommutativität der Einzahlungstransaktion wird aber trotzdem ein konsistenter Datenbankzustand (Kontostand: 1110 DM) erreicht. Bei der Zinsberechnung auf Knoten B wurde allerdings die zuvor getätigte Einzahlung (Knoten C) nicht mehr berücksichtigt.

geführt werden kann, ohne den aktuellen Zustand der Queue zuvor lesen zu müssen. (Die Position eines Elements in der Queue wird einzig über den Zeitstempel der **Enq**-Operation bestimmt.) Deshalb ist das Init Quorum der **Enq**-Operation leer. Dagegen hängt das Ergebnis einer **Deq**-Operation von anderen parallelen **Enq**- bzw. **Deq**-Operationen ab, weshalb sich das Init Quorum der **Deq**-Operation mit den jeweiligem Final Quorum der beiden anderen Operationen überschneiden muß. Bei n Kopien[27] führt dies zu folgender Quorumsverteilung:

[27] mit jeweils einer Stimme

$$\texttt{Deq} = (\text{IQ}, \text{FQ}) = (m, n - m + 1)$$
$$\texttt{Enq} = (0, n - m + 1) \qquad \text{mit } 0 < m \leq n$$

Die Quorumsverteilung für die Schreib/Leseoperationen beim Verfahren nach Gifford kann damit folgendermassen beschrieben werden:

$$\texttt{Write} = (\text{IQ}, \text{FQ}) = (m, n - m + 1)$$
$$\texttt{Read} = (m, 0) \qquad \text{mit } 0 < m \leq n$$

Wie aus diesen Beispielen ersichtlich, müssen zur kopienübergreifenden Synchronisation (Aufgabe 1) und zur Erhaltung der Datenbankkonsistenz im Fehlerfall (Aufgabe 3) die verschiedenen Operationen eines ADTs natürlich wie bei allen Voting Verfahren

bestimmte Überschneidungsregeln erfüllen. Die dafür notwendigen allgemeinen Bedingungen werden in [32] formuliert.

Statt dabei den aktuellen Wert eines ADT-Objekts zu speichern, wird auf jeder Kopie des Objekts eine möglichst vollständige zeitliche Reihenfolge der auf diesem Objekt durchgeführten Operationen in einer *History* (= Operation plus Zeitstempel) verwaltet. Dadurch kann — im Gegensatz zu „wertbasierten" Ansätzen (z. B.: [55, 30]) — der aktuelle Objektwert auch aus unvollständigen Knoten-Histories berechnet werden, in dem aus diesen durch zeitliche Sortierung und Elimination von Mehrfacheinträgen eine vollständige History erzeugt wird. Das folgende Beispiel verdeutlicht diese Vorgehensweise.

Beispiel 11: Der ADT `Queue` mit den Operationen `Enq` und `Deq`
Bei einem Replikationsgrad $n = 4$ ist die folgende Quorumsverteilung für die `Deq`- und `Enq`-Operation möglich: `Deq` = (3,2) `Enq` = (0,2).

Es wird nun die Operationsfolge `Enq(a)`, `Enq(b)`, `Deq()` auf der über die Knoten `A`, `B`, `C`, `D` replizierten Queue durchgeführt. Zu Beginn besitzt jeder Knoten eine leere History.

Die Operation `Enq(a)` muß aufgrund ihres leeren Initial Quorums keine Histories zur Bestimmung des aktuellen Zustands der Queue anfordern. Die History des Final Quorums besteht deshalb nur aus der neuen Operation mit Zeitstempel 2 (Tabelle 3.1).

Operation: `Enq(a)`	
History Initial Quorum	History Final Quorum
–	2 `Enq(a)`

Tabelle 3.1

Diese History wird an FQ Knoten (`A`, `B`) verschickt (Tabelle 3.2).

History Knoten `A`	History Knoten `B`	History Knoten `C`	History Knoten `D`
2 `Enq(a)`	2 `Enq(a)`	–	–

Tabelle 3.2

Analog wird mit der Operation `Enq(b)` verfahren (Tabellen 3.3–3.4).

Operation: `Enq(b)`	
History Initial Quorum	History Final Quorum
–	3 `Enq(b)`

Tabelle 3.3

History Knoten `A`	History Knoten `B`	History Knoten `C`	History Knoten `D`
2 `Enq(a)`	2 `Enq(a)`	–	–
–	–	3 `Enq(b)`	3 `Enq(b)`

Tabelle 3.4

Nach diesen `Enq`-Operationen speichert keiner der Knoten eine History, die alle Operationen widerspiegelt. Die für die `Deq`-Operation notwendige vollständige History kann aber aus drei beliebigen Knoten-Histories erzeugt werden (Initial-Quorum für `Deq` ist 3), um den Rückgabewert (hier: `a`) dieser Operation bestimmen zu können (Tabelle 3.5).

Operation: `a = Deq()`	
History Initial Quorum	History Final Quorum
2 `Enq(a)`	2 `Enq(a)`
3 `Enq(b)`	3 `Enq(b)`
	4 `Deq()`

Tabelle 3.5

Die Final History wird an zwei beliebige Knoten verschickt und dort in die lokalen Histories eingereiht (Tabelle 3.6).

History Knoten `A`	History Knoten `B`	History Knoten `C`	History Knoten `D`
2 `Enq(a)`	2 `Enq(a)`	2 `Enq(a)`	–
3 `Enq(b)`	–	3 `Enq(b)`	3 `Enq(b)`
4 `Deq()`	–	4 `Deq()`	–

Tabelle 3.6

Wie aus dem Beispiel ersichtlich, wachsen die Histories der einzelnen Knoten immer mehr an, wodurch die Berechnung des aktuellen Objektwerts immer aufwendiger wird. Deshalb wird in [32] vorgeschlagen, in bestimmten Zeitabständen die Histories durch den aktuellen Objektwert zu ersetzen (*log compaction*). Diese neue *Objektversion* dient dann als Bezugspunkt für weitere Histories.

Herlihy schlägt neben den Operationen zur Manipulation der ADT-Objekte auch Operationen für die dynamische Veränderung der Quorumsverteilung vor, um diese an andere Zugriffsprofile (vgl. dazu Abschnitt 4.2.3.5) anpassen zu können [32].

Der Vorteil der individuellen Quorumsverteilung wird insbesondere bei einer großen Zahl verschiedener Zugriffsoperationen deutlich, da es dann immer schwieriger wird, ein für alle Operationen geeignetes festes

Schreib/Lesequorum zu finden. Anderseits ist im allgemeinen Fall das Finden einer individuellen und korrekten Quorumsverteilung bei vielen Zugriffsoperationen aufwendig.

6 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. Anderseits benötigen replizierte Systeme zusätzliche Synchronisationsmechanismen — realisiert durch sogenannte *Replikationsverfahren* — zur Sicherstellung der Datenkonsistenz.

In diesem Papier wurden zuerst die verschiedenen Aufgaben eines Replikationsverfahrens beschrieben und daran die Methoden klassifiziert, wie sie diese Aufgaben lösen. Anschließend erfolgte eine Beschreibung einiger der klassifizierten Verfahren. Ein tabellarischer Überblick über die hier beschriebenen Lösungsvorschläge ist im Anhang zu finden.

Der inhärente Zielkonflikt zwischen Verfügbarkeit/Durchsatz und Datenkonsistenz 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, Epsilon–Serialisierbarkeit), unter zu Hilfenahme von semantischen Wissen (z.B. Kommutativität) oder alleinig über die Anwendungssemantik bestimmt werden. Dieser Korrektheitsbegriff wird dabei je nach Aufgabenbereich für unterschiedliche Zwecke verwandt. Beispielsweise definiert das optimistische Protokoll (siehe Abschnitt 4.1) eine korrekte Reintegration von Partitionen über die 1–Kopie–Serialisierbarkeit (Aufgabe 3), während die Voting Verfahren diesen Korrektheitsbegriff für die kopienübergreifende Transaktionssynchronisation einsetzen (Aufgabe 1).

Da die Replikationsverfahren nicht alle in Kapitel 3 beschriebenen Aufgabenfelder abdecken, sind hier auch Kombinationen von Verfahren mit unterschiedlichen Korrektheitsbegriffen denkbar. Beispielsweise kann die syntaktische Available Copy Methode (siehe Abschnitt 4.2.3.1) mit dem semantischen Data Patch Verfahren (siehe Abschnitt 5.1.2) 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 Copy Ansatz zeigt auch, daß sich die Replikationsverfahren bzgl. der unterstützten Fehlersemantiken [12] unterscheiden. Während einige wenige Verfahren nur Knotenabstürze tolerieren, erhalten andere die Datenbankkonsistenz auch im Falle von Partitionierungen.

Inzwischen bieten einige kommerzielle Datenbanksysteme Replikationsmöglichkeiten an: Sybase Replication Server [11], Oracle [53], Ingres Replicator, Adabas Entire SQL, Informix, DB2. Häufig stehen diese Kopien jedoch in einer *Master–Slave*–Beziehung [41]: Eine Änderung ist nur über die Master–Kopie möglich. Die meist asynchron über Trigger aktualisierten Slave–Kopien stehen nur für einen Lesezugriff zur Verfügung (vgl. Abschnitte 4.2.1.1, 4.2.1.2).

Ingres und Oracle bieten dagegen auch eine *Peer–to–Peer* Replikationsmethode [41] an, bei der Änderungen über jede Kopie möglich sind. Da auch hier die Änderungenpropagierung asynchron über Trigger erfolgt (optimistische Ansätze) sind Inkonsistenzen nicht ausgeschlossen, die dann über vom Anwender zu definierende Regeln aufgelöst werden (vgl. Abschnitt 5.1.2).
Ein kurzer Überblick über den derzeitigen Stand an kommerziellen Systemen mit Replikationsmöglichkeiten (Stand Ende 94) ist in [41] zu finden.

Danksagung

Wir danken Daniela Beuter, Christian Heinlein sowie Manfred Reichert für ihre wertvollen Hinweise und Ratschläge beim Entstehen dieser Ausarbeitung.

ANHANG

Verfahren	kopienübergreifende Synchronisation der Transaktionszugriffe		Aktualisierung der Kopien beim Transaktions–Commit	Konsistenzsicherung im Fehlerfall	
	Lesetransaktion	Änderungstransaktion		unterstützte Fehlersemantik	Behandlung von Partitionen
optimistic protocol	o	o	o	Partitionierungen	Zyklenanalyse und Aufbrechen von Zyklen durch Transaktionsrücksetzungen und -wiederholungen
primary copy, token	konsistentes Lesen: Zustimmung der Primärkopie bzw. Token wenn inkonsistentes Lesen tolerierbar: Zustimmung der lokalen Kopie	Zustimmung der Primärkopie bzw. Token	synchrone Änderung der Primärkopie bzw. der "Token"–Kopie	Knotenausfälle, Partitionierungen problematisch	Nachziehen der fehlenden Änderungen
majority consensus, weighted voting, voting with witness, ghosts, bystanders, tree quorum, grid quorum	Zustimmung eines Lesequorums	Zustimmung eines Schreibquorums	synchrones Aktualisieren aller Kopien im Schreibquorum	Knotenausfälle, Partitionierungen	Nachziehen der fehlenden Änderungen in den Nicht–Mehrheitspartitionen
ROWA	Zustimmung einer Kopie, allerdings "running in the past" möglich	Zustimmung aller Kopien	synchrones Aktualisieren aller Kopien	Knotenausfälle, Partitionierungen	während Partitionierungen keine Änderung möglich
available copies	Zustimmung einer Kopie, allerdings "running in the past" möglich	Zustimmung der verfügbaren Kopien	synchrones Aktualisieren aller erreichbaren Kopien	nur Knotenausfälle	Nachziehen der fehlenden Änderungen beim Wiederanlauf eines Knotens
virtual partitions	Zustimmung einer Kopie aus Verzeichnis, allerdings "running in the past" möglich	Zustimmung aller Kopien aus Verzeichnis	synchrones Aktualisieren aller Kopien aus Verzeichnis	Knotenausfälle, Partitionierungen	Nachziehen der fehlenden Änderungen beim Eintritt in eine neue virtuelle Partition
missing Updates	Normalmodus: Zustimmung einer Kopie, "running in the past" möglich Fehlermodus: Zustimmung eines Lesequorums	Normalmodus: Zustimmung aller Kopien Fehlermodus: Zustimmung eines Schreibquorums	Normalmodus: synchrones Aktualisieren aller Kopien Fehlermodus: synchrones Aktualisieren aller Kopien im Schreibquorum	Knotenausfälle, Partitionierungen	Nachziehen der fehlenden Änderungen

Tabelle 4: Die verschiedenen Replikationsverfahren im Überblick, Teil 1

o: Diese Aufgabe wird durch dieses Verfahren nicht behandelt

Verfahren	kopienübergreifende Synchronisation der Transaktionszugriffe		Aktualisierung der Kopien beim Transaktions-Commit	Konsistenzsicherung im Fehlerfall	
	Lese-transaktion	Änderungs-transaktion		unterstützte Fehler-semantik	Behandlung von Partitionen
Replication regeneration	Zustimmung einer Kopie	Zustimmung der verfügbaren Kopien	synchrones Aktualisieren aller erreichbaren Kopien	nur Knotenausfälle	Ersetzung eines nichterreichbaren Knotens durch einen neuen
CDDR	kopienhaltenden Knoten: Zustimmung der lokalen Kopie kopienlose Knoten: Zustimmung des Primärknotens	Zustimmung aller gerade kopienhaltenden Knoten	synchrones Aktualisieren aller erreichbaren Kopien	nur Knotenausfälle	wiederanlaufender Knoten wird zum kopienlosen Knoten
log transformation	Zustimmung einer Kopie, jedoch evtl. durch verspätete Transaktionen nach log Transformation Transaktionsrücksetzungen und -wiederholungen		synchrone Änderung der lokalen Kopie	Knotenausfälle, Partitionierungen	nach log Transformation evtl. Transaktionsrücksetzungen und -wiederholungen
data patch	o	o	o	Knotenausfälle, Partitionierungen	Festlegung des aktuellen Objektwerts durch Anwendung von bei der Schemadefinition spezifizierten Regeln
mc-compatibility	kommutative Transaktion: Zustimmung einer Kopie, "running in the past" möglich nicht kommutative Transaktion: Zustimmung eines Lesequorums	kommutative Transaktion: Zustimmung einer Kopie nicht kommutative Transaktion: Zustimmung eines Schreibquorums	kommutative Transaktion: synchrone Änderung der lokalen Kopie nicht kommutative Transaktion: synchrones Aktualisieren des Schreibquorums	Knotenausfälle, Partitionierungen	kommuntative Transaktion: Nachziehen der Änderungen in beliebiger Reihenfolge möglich nicht kommutative Transaktion: Nachziehen der fehlenden Änderungen in den Nicht-Mehrheitspartitionen
general quorum consensus	Zustimmung des Initquorums der ADT-Operation	Zustimmung des Finalquorums der ADT-Operation	synchrones Aktualisieren der Kopien im Finalquorum	Knotenausfälle, Partitionierungen	nur bei log compaction notwendig

Tabelle 4: Die verschiedenen Replikationsverfahren im Überblick, Teil 2

o: Diese Aufgabe wird durch dieses Verfahren nicht behandelt

Literatur

[1] D. Agrawal and A. El Abbadi. The generalized tree quorum protocol: An efficient approach for managing replicated data. *ACM Trans. on Database Systems*, 17(4):689–717, Dec. 1992.

[2] D. Agrawal and A. El Abbadi. Resilient logical structures for effient management of replicated data. In *Proc. 18th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 151–162, Vancouver, Canada, Aug. 1992.

[3] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Systems*, 9(4):596–615, 1984.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[5] B. T. Blaustein and C. W. Kaufman. Updating replicated data during communications failures. In *Proc. 11th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 49–58, Stockholm, Aug. 1985. Morgan Kaufmann.

[6] U. M. Borghoff. Fehlertoleranz in verteilten Dateisystemen
Eine Übersicht über den heutigen Entwicklungsstand bei den Votierungsverfahren. *GI Informatik Spektrum*, 14(1):15–27, Feb. 1991.

[7] S. Y. Cheung, M. Ahamad, and M. H. Ammar. Multi-dimensional voting: A general method for implementing sychronization in distributed systems. In DCS [17], pages 362–369.

[8] S. Y. Cheung, M. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. In DE [18], pages 438–445.

[9] W. W. Chu and J. Hellerstein. The exclusive-writer approach to updating replicated files in distributed processing systems. *IEEE Trans. on Computers*, C-34(6):489–500, May 1985.

[10] S. M. Chung. Enhanced tree quorum algorithm for replica control in distributed database systems. *Data & Knowledge Engineering*, 12(1):63–81, Feb. 1994.

[11] M. Colton. Replicated data in a distributed environment. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 464–466, Washington, DC, May 1993.

[12] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb. 1991.

[13] P. Dadam. Synchronisation in verteilten Datenbanken: Ein Überblick, Teil 1. *GI Informatik Spektrum*, 4:175–184, 1981.

[14] D. Davcev. A dynamic voting scheme in distributed systems. *IEEE Trans. on Software Engineering*, 15(1):93–97, Jan. 1989.

[15] S. B. Davidson. Optimism and consistency im partitioned distributed database systems. *ACM Trans. on Database Systems*, 9(3):456–481, Sept. 1984.

[16] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, Sept. 1985.

[17] *Proc. 10th International Conference on Distributed Computing Systems*, Paris, France, May/June 1990.

[18] *Proceedings of the 6th International Conference on Data Engineering*, Los Angeles, California, USA, Feb. 1990.

[19] *Proceedings of the 7th International Conference on Data Engineering*, Kobe, Japan, Apr. 1991.

[20] W. Du and A. K. Elmagarmid. Quasi-serializability: A correctness criterion on global concurrency control in InterBase. In *Proc. 15th Int'l Conf. on Very Large Data Bases (VLDB)*, Amsterdam, The Netherlands, Aug. 1989. Morgan Kaufmann.

[21] W. Du, A. K. Elmagarmid, W. Kim, and O. Bukhres. Supporting consistent updates in replicated multidatabase systems. *IBM Systems Journal*, 2(2):215–241, Apr. 1993.

[22] W. Du, A. K. Elmargarmid, and W. Kim. Maintaining quasi-serializability in multidatabase systems. In DE [19], pages 360–367.

[23] D. L. Eager and K. C. Sevcik. Achieving robustness in distributed database systems. *ACM Trans. on Database Systems*, 8(3):354–381, Sept. 1983.

[24] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In M. Stonebraker, editor, *Reading in Database Systems*, pages 259–273. Morgan Kaufmann, San Mateo, California, 1988.

[25] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. on Database Systems*, 14(2):264–290, June 1989.

[26] A. K. Elmargarmid and W. Du. Maintaining quasi-serializability in multidatabase systems. In DE [18], pages 37–46.

[27] H. Garcia-Molina. Data-patch: Integrating inconsistent copies of a database after a partition. In *IEEE Proc. 3rd Symposium on Reliable Distributed Systems*, pages 38–48, New York, Oct. 1983.

[28] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(2):186–213, June 1983.

[29] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, Oct. 1985.

[30] D. K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symposium on Operating Systems Principles (SIGOPS)*, pages 150–159, Pacific Grove, California, Dec. 1979.

[31] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[32] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. on Computer Systems*, 4(1):32–53, 1986.

[33] Y. Huang and O. Wolfson. A competitive dynamic data replication algorithm. In *Proceedings of the 9th International Conference on Data Engineering*, pages 310–317, Vienna, Austria, Apr. 1993.

[34] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Systems*, 15(2):230–280, June 1990.

[35] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. on Database Systems*, 19(4):586–625, Dec. 1994.

[36] A. Kumar. Performancs analysis of a hierarchical quorum consensus algorithm for replicated objects. In DCS [17], pages 378–385.

[37] A. Kumar. Hierarchical quorum consensus: A new algorithmen for managing replicated data. *IEEE Trans. on Computers*, C-40(9):996–1004, Sept. 1991.

[38] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 117–125, Chicago, Illinois, USA, June 1988.

[39] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2):213–226, 1981.

[40] T. Minoura and G. Wiederhold. Resilient extended true-copy token schema for a distributed database system. *IEEE Trans. on Software Engineering*, SE-8(3):173–188, May 1982.

[41] U. Ofer. Was Sie schon immer über Replication Server wissen wollen. *Datenbank Fokus*, 6:31–36, 1994.

[42] J.-F. Paris. Efficient management of replicated data. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, volume 426, pages 397–409. Springer, 1988.

[43] J.-F. Paris. Voting with bystanders. In *Proc. 9th International Conference on Distributed Computing Systems*, pages 394–401, Newport Beach, CA, 1989.

[44] J.-F. Paris. Efficient voting protocols with witnesses. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, volume 470, pages 305–316. Springer, 1990.

[45] J.-F. Paris. A highly available replication control protocol using volatile witnesses. In *Proc. 14th International Conference on Distributed Computing Systems*, pages 536–543, Pittsburgh, Pennsylvania, May 1994.

[46] C. Pu and A. Leff. Replica control in distributed systems: An asynchonous approach. *ACM SIGMOD Record*, 20(2):377–386, June 1991.

[47] C. Pu, J. D. Noe, and A. Proudfoot. Regeneration of replicated objects: A technique and its Eden implementation. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 175–187, Los Angeles, California, USA, Feb. 1986.

[48] M. Rusinkiewicz, D. Georgakopoulos, and R. Thomas. RDS: A primitive for the mainenance of replicated data objects. In *IEEE Proc. 2nd Symposium on Parallel and Distributed Processing*, pages 658–667, Dallas, Dec. 1990.

[49] S. K. Sarin, B. T. Blaustein, and C. W. Kaufman. System architecture for partition-tolerant distributed databases. *IEEE Trans. on Computers*, C-34(12):1158–1163, Dec. 1985.

[50] S. H. Son. Replicated data management in distributed database systems. *ACM SIGMOD Record*, 17(4):62–69, Dec. 1988.

[51] S. H. Son and S. Kouloumbis. Replication control for distributed real-time database systems. In *Proc.*

12th International Conference on Distributed Computing Systems, pages 144–151, Yokohama, Japan, June 1992.

[52] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. on Software Engineering*, SE–5(3):188–194, May 1979.

[53] G. Stürmer. *Oracle 7 A User's and Developer's Guide. Including Release 7.1.* Thomson, 1995.

[54] J. Tang. Voting class — an approach to achieving high availability for replicated data. In *Proceedings of Second International Symposium on Databases in Parallel and Distributed Systems*, pages 146–156, Trinity College, Dublin, Ireland, July 1990.

[55] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.

[56] R. van Renesse and A. S. Tanenbaum. Voting with ghosts. In *Proc. 8th International Conference on Distributed Computing Systems*, pages 456–462, Washington, DC, 1988.

[57] C. Wu and G. G. Belford. The triangular lattice protocol: A highly fault tolerant and highly efficient protocol for replicated data. In *IEEE Proc. 11th Symposium on Reliable Distributed Systems*, pages 66–73, Houston, Texas, Oct. 1992.

[58] K.-L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In DE [19], pages 506–515.

Liste der bisher erschienenen Ulmer Informatik-Berichte

Einige davon sind per FTP von `ftp.informatik.uni-ulm.de` erhältlich
Die mit * markierten Berichte sind vergriffen

List of technical reports published by the University of Ulm

Some of them are available by FTP from `ftp.informatik.uni-ulm.de`
Reports marked with * are out of print

91-01 *Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe*
Instance Complexity

91-02* *K. Gladitz, H. Fassbender, H. Vogler*
Compiler-Based Implementation of Syntax-Directed Functional Programming

91-03* *Alfons Geser*
Relative Termination

91-04* *J. Köbler, U. Schöning, J. Toran*
Graph Isomorphism is low for PP

91-05 *Johannes Köbler, Thomas Thierauf*
Complexity Restricted Advice Functions

91-06* *Uwe Schöning*
Recent Highlights in Structural Complexity Theory

91-07* *F. Green, J. Köbler, J. Toran*
The Power of Middle Bit

91-08* *V.Arvind, Y. Han, L. Hamachandra, J. Köbler, A. Lozano, M. Mundhenk, A. Ogiwara, U. Schöning, R. Silvestri, T. Thierauf*
Reductions for Sets of Low Information Content

92-01* *Vikraman Arvind, Johannes Köbler, Martin Mundhenk*
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets

92-02* *Thomas Noll, Heiko Vogler*
Top-down Parsing with Simulataneous Evaluationof Noncircular Attribute Grammars

92-03 *Fakultät für Informatik*
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen

92-04* *V. Arvind, J. Köbler, M. Mundhenk*
Lowness and the Complexity of Sparse and Tally Descriptions

92-05* *Johannes Köbler*
Locating P/poly Optimally in the Extended Low Hierarchy

92-06* *Armin Kühnemann, Heiko Vogler*
Synthesized and inherited functions -a new computational model for syntax-directed semantics

92-07* *Heinz Fassbender, Heiko Vogler*
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing

92-08* *Uwe Schöning*
On Random Reductions from Sparse Sets to Tally Sets

92-09* *Hermann von Hasseln, Laura Martignon*
Consistency in Stochastic Network

92-10 *Michael Schmitt*
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent
Any Linearly Separable Boolean Function

92-11 *Johannes Köbler, Seinosuke Toda*
On the Power of Generalized MOD-Classes

92-12 *V. Arvind, J. Köbler, M. Mundhenk*
Reliable Reductions, High Sets and Low Sets

92-13 *Alfons Geser*
On a monotonic semantic path ordering

92-14* *Joost Engelfriet, Heiko Vogler*
The Translation Power of Top-Down Tree-To-Graph Transducers

93-01 *Alfred Lupper, Konrad Froitzheim*
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications
Manager

93-02 *M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch*
The COCOON Object Model

93-03 *Thomas Thierauf, Seinosuke Toda, Osamu Watanabe*
On Sets Bounded Truth-Table Reducible to P-selective Sets

93-04 *Jin-Yi Cai, Frederic Green, Thomas Thierauf*
On the Correlation of Symmetric Functions

93-05 *K.Kuhn, M.Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam*
A Conceptual Approach to an Open Hospital Information System

93-06 *Klaus Gaßner*
Rechnerunterstützung für die konzeptuelle Modellierung

93-07 *Ullrich Keßler, Peter Dadam*
Towards Customizable, Flexible Storage Structures for Complex Objects

94-01 *Michael Schmitt*
On the Complexity of Consistency Problems for Neurons with Binary Weights

94-02 *Armin Kühnemann, Heiko Vogler*
A Pumping Lemma for Output Languages of Attributed Tree Transducers

94-03 *Harry Buhrman, Jim Kadin, Thomas Thierauf*
On Functions Computable with Nonadaptive Queries to NP

94-04 *Heinz Faßbender, Heiko Vogler, Andrea Wedel*
Implementation of a Deterministic
Partial E-Unification Algorithm for Macro Tree Transducers

94-05 *V. Arvind, J. Köbler, R. Schuler*
On Helping and Interactive Proof Systems

94-06 *Christian Kalus, Peter Dadam*
Incorporating record subtyping into a relational data model

94-07 *Markus Tresch, Marc H. Scholl*
A Classification of Multi-Database Languages

94-08 *Friedrich von Henke, Harald Rueß*
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge

94-09 *F.W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker*
Construction and Deduction Methods for the Formal Development of Software

94-10 *Axel Dold*
Formalisierung schematischer Algorithmen

94-11 *Johannes Köbler, Osamu Watanabe*
New Collapse Consequences of NP Having Small Circuits

94-12 *Rainer Schuler*
On Average Polynomial Time

94-13 *Rainer Schuler, Osamu Watanabe*
Towards Average-Case Complexity Analysis of NP Optimization Problems

94-14 *Wolfram Schulte, Ton Vullinghs*
Linking Reactive Software to the X-Window System

94-15 *Alfred Lupper*
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen

94-16 *Robert Regn*
Verteilte Unix-Betriebssysteme

94-17 *Helmuth Partsch*
Again on Recognition and Parsing of Context-Free Grammars: Two Exercises in Transformational Programming

94-18 *Helmuth Partsch*
Transformational Development of Data-Parallel Algorithms: an Example

95-01 *Oleg Verbitsky*
On the Largest Common Subgraph Problem

95-02 *Uwe Schöning*
Complexity of Presburger Arithmetic with Fixed Quantifier Dimension

95-03 *Harry Buhrman, Thomas Thierauf*
The Complexity of Generating and Checking Proofs of Membership

95-04 *Rainer Schuler, Tomoyuki Yamakami*
Structural Average Case Complexity

95-05 *Klaus Achatz, Wolfram Schulte*
Architecture Indepentent Massive Parallelization of Divide-And-Conquer Algorithms

95-06 *Christoph Karg, Rainer Schuler*
Structure in Average Case Complexity

95-07 *P. Dadam, K. Kuhn, M. Reichert, T. Beuter, M. Nathe*
ADEPT: Ein integrierender Ansatz zur Entwicklung flexibler, zuverlässiger kooperierender Assistenzsysteme in klinischen Anwendungsumgebungen

95-08 *Jürgen Kehrer, Peter Schulthess*
Aufbereitung von gescannten Röntgenbildern zur filmlosen Diagnostik

95-09 *Hans-Jörg Burtschick, Wolfgang Lindner*
On Sets Turing Reducible to P-Selective Sets

95-10 *Boris Hartmann*
Berücksichtigung lokaler Randbedingung bei globaler Zieloptimierung mit neuronalen Netzen am Beispiel Truck Backer-Upper

95-11 *Thomas Beuter, Peter Dadam*
Prinzipien der Replikationskontrolle in verteilten Systemen

Ulmer
Informatik-Berichte

ISSN 0939-5091

Herausgeber: Fakultät für Informatik

Universität Ulm, Oberer Eselsberg, D-89069 Ulm