Einsatz von Replikation

Florian Munz
f_munz@informatik.uni-kl.de

16. Juli 2004

Inhaltsverzeichnis

1 Einleitung

Mit dem rapiden Wachstum des Internet und populären Websites wie Google, Amazon, Ebay, Yahoo etc. mit einem hohen Verkehrs- und Lastaufkommen steigen die Anforderungen an die Skalierbarkeit und Verfügbarkeit von webbasierten Informationssystemen. Neue Nutzungsszenarien für Informationssysteme und die zugrunde liegende Datenbanktechnologie ergeben sich auch aus Cluster- und Grid-/Blade-Konfigurationen sowie durch die rapide Verbreitung von autonomen mobilen Endgeräten wie Laptops und PDAs.

Die Replikation von Daten adressiert die Probleme der Verfügbarkeit und Skalierbarkeit sowie die Verbesserung der (lokalen) Antwortzeit. Replikation ist die bewusste Erzeugung von redundanten Daten (Kopien) und steht damit im Widerspruch zu dem sonst üblichen Ziel möglichst geringer Redundanz in Informationssystemen (z. B. im Umfeld der relationalen Normalformen). Im Unterschied zum Caching (Cache = Versteck, geheimes Lager), das in der Administration von Informationssystemen möglichst unsichtbar sein sollte und nur dynamische und flüchtige Datenkopien herstellt, entstehen durch die Replikation in der Regel neue Datenbankinstanzen mit Kopien von Daten, die bereits in anderen Datenbankinstanzen gespeichert werden. Je statischer die replizierten Daten sind, desto leichter erreicht man durch eine Replikation die folgenden Ziele:

- Skalierbarkeit von lesenden Zugriffen durch eine Lastverteilung auf mehrere völlig identische Datenbankinstanzen
- Erhöhung der Verfügbarkeit, weil der Ausfall einer redundanten Datenbankinstanz nur die Skalierung beeinträchtigt
- Reduktion der lokalen Antwortzeit, weil die Datenbankinstanz im selben LAN verfügbar ist.

In der Praxis sind die replizierten Daten natürlich nicht völlig statisch. Änderungen müssen deshalb an die anderen Datenbankinstanzen propagiert werden. Dies erfolgt nach [Oracle02] durch einen – in Abbildung 1 dargestellten – Synchronisationsprozess, der die Änderungen abgreift (Capture), sie zwischenspeichert (Staging) und dann am Zielsystem abliefert (Consumption).

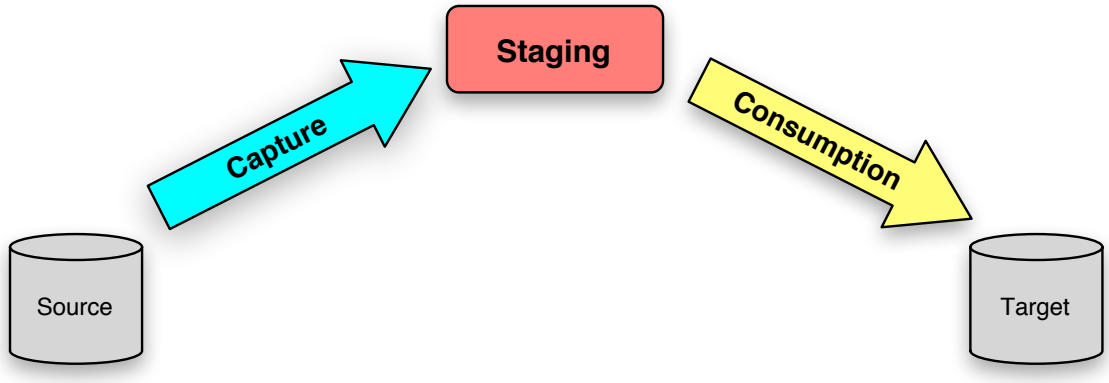

Abbildung 1: Synchronisationsprozess der Replikation

Diese Capture-Staging-Consumption-Methode ist das Prinzip aller Replikationsverfahren, wobei unterschiedliche Mittel zum Einsatz kommen. Das Ziel der Replikation ist weniger die transaktionale Konsistenz über alle durch die Replikation erzeugten Kopien, bei der Änderungen simultan in allen Kopien durchzuführen sind. Vielmehr wird die transaktionale Konsistenz durch das Ziel der Konvergenz ersetzt, d. h., man möchte, dass nach einer gewissen änderungsfreien Zeit alle Kopien einen identischen Zustand annehmen.

Für sehr dynamische Daten (z. B. in Hochleistungs-Transaktionssystemen) bereitet die Replikation mehr Probleme als sie löst und wird dort – mit Ausnahme der Standby-Konfigurationen zur Hochverfügbarkeit – auch nicht eingesetzt. Typische Fälle von replizierten Daten sind weitgehend statische Anwendungsdaten, sogenannte Stammdaten, wie z. B. Kundenadressen, Produktkataloge, Preislisten, die für viele Unternehmensteile und –anwendungen interessant sind, aber nur einem geringen Änderungsvolumen unterliegen.

Dieser Aufsatz stellt die verschiedenen Anwendungsmöglichkeiten der Replikation vor und gibt einen Überblick über die unterschiedlichen Replikationsverfahren für Anwendungsdaten in Informationssystemen und ihre Zielsetzungen.

2 Anwendungsszenarien

Um einen Überblick über das Themenfeld Replikation zu bekommen, folgt ein Überblick, welche Probleme damit in der Praxis gelöst werden. Im Prinzip können Anwendungen, bei denen eine Datenreplikation zum Einsatz kommt, in zwei Bereiche aufgeteilt werden: Zum einen wird Replikation benutzt, um die Daten „vor Ort“ zu bringen, d.h. sie regional oder geographisch zu verteilen, sei es auf die verschiedenen Unternehmenstandorte oder

-filialen oder auch auf die Laptops von (Außendienst-)Mitarbeitern. Zum anderen werden die Daten verteilt, um die Last von Anfragen (Datenbank-Queries) auf verschiedene Datenbankinstanzen mit identischem Inhalt zu verteilen und auch den Ausfall einer dieser Instanzen ohne größere Beeinträchtigung verkraften zu können.

Die Replikation von Daten kommt vor allem im Umfeld des Mobile Computing sowie bei der Skalierung und höheren Verfügbarkeit von Web-Anwendungen zum Einsatz. Im folgenden werden diese drei Bereiche weiter vertieft und anhand von Beispielen verdeutlicht.

2.1 Mobile Computing

Verbunden mit der weiten Verbreitetung von mobilen Endgeräten und der rasanten Entwicklung von drahtlosen mobilen Kommunikationssystemen gewinnen unternehmensweite Informationssysteme mit mobilen Endgeräten an Bedeutung. Das Ziel von solchen Informationssystemen ist, den Zugriff auf die unternehmensweiten Informationen zu jedem beliebigen Zeitpunkt und von jedem beliebigen Ort zu ermöglichen. Typische Szenarien sind Unternehmen, die ihre Außendienstmitarbeiter (Vertrieb, Kundendienst) mit aktuellen Informationen versorgen (Kundenadressen, Produktbeschreibungen, Montagepläne, Serviceanleitungen, Preislisten, Einsatzpläne) und ihnen diese Daten in Form einer lokalen Kopie zur Verfügung stellen, die z. B. täglich aktualisiert wird. Die Außendienstmitarbeiter haben dann einen autonomen Zugriff auf die für sie relevanten Informationen, der unabhängig von Funklöchern oder Faradayschen Käfigen in Stahlbetongebäuden ist. Die Kundendaten und Einsatzpläne werden in Regel spezifisch für ein Vertriebs- oder Servicegebiet verteilt, während allgemein interessierende Informationen auf allen Laptops zur Verfügung gestellt werden. In der Firmenzentrale steht dann meist eine konsolidierte Datenbank, in der alle Informationen für alle Außendienstmitarbeiter aufgebaut und verwaltet werden, während einzelne Außendienstmitarbeiter nur den für sie relevanten Ausschnitt erhalten.

Auf dem Laptop werden dann Offline-Anwendungen bereitgestellt, die mit dem lokalen Datenbestand arbeiten. Bei Vertriebsmitarbeitern können so z. B. Kundenbestellungen erfasst oder geändert werden. Die Übertragung dieser Daten in die Informationssysteme des Unternehmens erfolgt erst, wenn sich der Vertriebsmitarbeiter in seinem Vertriebsbüro oder auch Home Office wieder an das unternehmensweise Informationssystem anschließt und damit eine Aktualisierung der in seinem Laptop gespeicherten Daten vornimmt. Je

nach der Richtung der eingesetzten Replikation spricht man von einer unidirektionalen Replikation (z. B. von der Unternehmenszentrale zu dem Laptop im Außendienst) oder von einer bidirektionalen Replikation, wenn auch der Rückweg unterstützt wird. Bei unidirektionaler Replikation erhält der Adressat nur eine Leseberechtigung bzw. seine Änderungen bleiben lokal. Bei einer bidirektionalen Replikation dürfen beide Replikationsteilnehmer ändern und die Änderungen werden in beide Richtungen propagiert. Bei der Synchronisation solcher bidirektionaler Replikationen kann es zu Konflikten kommen, für deren Behandlung es verschiedene Strategien gibt.

Für mobile Endgeräte gibt es neben den autonomen Offline-Anwendungen auch Online-Anwendungen, die von einer permanenten Erreichbarkeit des Endgeräts über Funk ausgehen. Diese Anwendungen und ihr Betrieb unterscheiden sich sehr stark von Offline-Anwendungen und ihrem Bedarf an Datenreplikation. Online-Anwendungen verhalten sich prinzipiell wie Client-/Server-Andwendungen und operieren in der Regel auf einer zentralen Datenbankinstanz. Dies ist zwar einfacher, mobile Online-Anwendungen sind aber nicht so autonom zu betreiben wie Offline-Anwendungen. Die ständige Verfügbarkeit des zentralen Servers und der Funkverbindung werden vorausgesetzt. Beim Abruf von großen Datenmengen (Dokumente, Grafik) über die immer noch schmalbandigen Funknetze kommt es zudem zu langen Wartezeiten. Im Kontext der Replikation werden die Aspekte mobiler Online-Anwendungen deshalb nicht weiter vertieft.

2.2 Skalierbarkeit von Leselast

Populäre Websites müssen täglich Hundertausende, manche auch mehrere Millionen Besucher verkraften. Dabei dominiert der Informationsabruf, d. h. die Leselast gegenüber den Änderungen bzw. Buchungen oder Bestellungen dieser Besucher. Auch die größten Serverkonfigurationen sind heute nicht in der Lage, die erforderliche Leselast mit einer einzigen Serverinstanz zu bewältigen. Ein naheliegender Ausweg ist es deshalb, für die relativ statischen Daten des Produkt- bzw. Informationsangebots von Ebay, Amazon, Google einen Lastverbund von Datenbankinstanzen zu schaffen, der aus identischen Kopien des Produkt- bzw. Informationskatalogs besteht und auf den die Lesezugriffe der Besucher verteilt werden. Die Lastverteilung erfolgt z. B. durch ein Multiplexing auf IP-Ebene oder auf der DNS-Ebene. Die Kopien in den beteiligten Datenbankinstanzen werden typischerweise durch eine Master/Slave-Replikation aktualisiert, wobei nur die Master-Kopie aktualisiert wird und die Replikation dann dafür sorgt, dass die Slave-Kopien denselben Änderungszustand erhalten. Bei größeren Websites findet man heute durchaus 20 – 50 Datenbankinstanzen für den Betrieb der Slave-Kopien. Die zu replizierenden Daten ändern sich in diesen

Nutzungsszenarien nur selten, so dass die Aktualisierung der Kopien gegenüber der Abwicklung der Leselast in den Hintergrund tritt. Auch eine zeitverzögerte Synchronisation der Slave-Kopien stört deshalb kaum.

Bei typischen e-Commerce Websites sind nach meiner Erfahrung 97-99% aller Besucher rein lesend unterwegs, nur 1-3% der Besucher führen schließlich auch eine Buchung oder eine Bestelllung durch. Die eigentlichen Bestelltransaktionen (Warenkorb) werden üblicherweise von anderen Datenbankinstanzen und manchmal auch von anderen Datenbankprodukten abgewickelt als die lesenden Katalogzugriffe.

2.3 Höhere Verfügbarkeit

Redundanz ist ein übliches Mittel, um die Verfügbarkeit von Systemen zu erhöhen. Die bereits beschriebene Datenreplikation nach dem Master/Slave-Prinzip mit vielen Slave-Kopien hat als positiven Nebeneffekt, dass der Ausfall der Master-Kopie oder einer Slave-Kopie nur zu einer geringen Beeinträchtigung im Leistungsverhalten des Lastverbundes führt. Dieses Replikationsprinzip ist allerdings nur für sehr statische Datenbestände mit einer hohen Leselast einsetzbar.

Je mehr Änderungen im Lastprofil vorhanden sind, desto mehr ist eine Reduktion des Master/Slave-Prinzips auf genau einen Master und genau einen Slave sinnvoll. Um die Änderungen des Masters im Slave nachzuvollziehen, wird dann das Redo-Log des Masters auf den Slave übertragen um damit alle Datenbankänderungen nachzufahren. Die Übertragung des Redo-Logs kann synchron und damit transaktionsgenau erfolgen, was aber zur Konsequenz hat, dass der Slave den Transaktionsdurchsatz des Masters limitiert. Typischerweise werden deshalb die Änderungen auf dem Slave asynchron nachvollzogen, d. h., der Slave läuft mit seinem Änderungsstand etwas hinter dem Master her. Beim Ausfall des Masters übernimmt der Slave die Funktion des Masters: die Datenbank-Sessions werden auf den Slave umgelenkt (IP-Switch), der die letzten abgeschlossenen Datenbanktransaktionen nachvollzieht und dann die Rolle des Masters übernimmt. Für die gerade aktiven Anwendungen bricht die aktuelle Transaktion ab, können dann aber in ihrer Session fortfahren. Das Slave-System wird nur für den Ausfall des Master-Systems bereitgehalten, Änderungen sind nur auf dem Master möglich. Der Slave wird meist nicht für Datenbank-Sessions verfügbar gemacht, in einigen Systemen werden aber auch Anfragen auf dem Slave zugelassen. Solche Hot-Standby-Konfigurationen sind ein Extremfall der Replikation, weil hier Datenbankinhalte mit einem hohen Änderungsvolumen propagiert werden, um eine Ausfallsicherheit auf Kosten eines redundanten Servers zu erreichen.

3 Replikationsverfahren

Das durch eine Replikation von Daten erzeugte Grundproblem ist es, die verschiedenen Kopien konsistent zu halten (Multi-Kopien-Problem). Hierzu gibt es eine größere Menge von Algorithmen, die zum großen Teil im Umfeld verteilter Datenbanksysteme entstanden sind. Im Unterschied zum Anspruch verteilter Datenbanksysteme begnügen sich Replikationsverfahren mit einer geringeren Abstraktion, weil eine vollständige Transparenz der Datenablage nicht mehr angestrebt wird. Vielmehr werden die Daten explizit und redundant auf verschiedene, voneinander völlig unabhängige Datenbankinstanzen verteilt. Auch die strikte Transaktionskonsistenz zentraler Datenbanksysteme wird für replizierte Daten in der Regel aufgegeben und durch den schwächeren Konsistenzbegriff der Konvergenz ersetzt. Dieser schwächere Konsistenzbegriff ist mit den Zielen der Replikation wie lokale Verfügbarkeit von Daten, Skalierbarkeit der Leselast und höherer Verfügbarkeit verträglicher als die übliche transaktionale Konsistenz.

3.1 Serialisierbarkeit und Konsistenz

In nicht-replizierten Datenbanken arbeitet man mit dem Begriff der Serialisierbarkeit.

Definition der Serialisierbarkeit bei einer replikatfreien Datenbank

> Ein Schedule S heißt serialisierbar, wenn es mindestens einen seriellen Schedule mit denselben TA gibt, der den gleichen DB-Zustand sowie die gleichen Ausgaben erzeugt wie S.

Übertragen auf replizierte Datenbanken führt das zu der Definition des strikten Konsistenzbegriffs:

Definition der 1-Kopien-Serialisierbarkeit

> Ein Schedule S einer replizierten DB heißt 1-Kopien-serialisierbar, wenn es mindestens eine serielle Ausführung der TA dieses Schedule auf einer replikatfreien DB gibt, der die gleichen Ausgaben sowie den gleichen DB-Zustand erzeugt wie S auf der replizierten DB.

Das heißt, obwohl mehrere physische Kopien existieren, gibt es ihre Zusammenfassung zu einer einzigen logischen Kopie und die Ausführung der nebenläufigen Transaktionen wird so koordiniert, das sie äquivalent zu einer Ausführung auf der logischen Kopie ist. Dies ist das strikteste Konsistenz-Kriterium. Es bedeutet, dass alle Kopien synchron aktualisiert werden.

Da man schon bei nicht replizierten Datenbanken den Begriff der Serialisierbarkeit aufgeweicht hat, um in der Praxis bessere Performance zu erreichen (siehe Konsistenzebenen), verwundert es nicht, wenn bei der Replikation der gleiche Weg eingeschlagen wurde. In der Praxis hat sich ein schwächeres Konsistenzkriterium durchgesetzt, um losere Kopplung der replizierten Datenbanken zu erreichen. Dabei werden nicht alle Kopien innerhalb der aktuellen Transaktion auf denselben Stand gebracht, sondern man hat nur noch das Ziel, dass alle Kopien im Gesamtsystem nach einer gewissen Zeit den selben Endzustand annehmen (Konvergenz).

3.2 Klassifikation der Replikationsverfahren

Die unterschiedlichen Kriterien der Konsistenz lassen sich nach [GHOS96] mit zwei Parametern beschreiben, nach denen man die Replikations-Protokolle klassifizieren kann (siehe auch Abbildung 2):

1. **Wann** werden Aktualisierungen zwischen den Kopien propagiert und
2. **Wo** werden die Aktualisierungen ausgeführt, also in welcher bzw. welchen Kopien.

Findet die Propagierung der Änderungen innerhalb der Transaktiongrenze statt, spricht man von einem *eager* bzw. synchronen Verfahren. Bei einem asynchronen – auch *lazy* genannten – Verfahren, werden die Änderungen erst nach dem Commit an die anderen Kopien weitergegeben.

Propagation vs. Ownership	**Lazy**	**Eager**
Group	N transactions N object owners	one transaction N object owners
Master	N transactions one object owner	one transaction one object owner

Abbildung 2: Klassifikation nach [GHOS96]

Die Aktualisierungen können zudem nur auf einem ausgezeichneten System, der *Primary Copy*, zugelassen werden oder auf potentiell jedem Knoten des Replikationsverbunds (*Update Anywhere*). Im ersten Fall spricht man auch von einer unidirektionalen Replikation (*Update Master*), im zweiten Fall von einer bidirektionalen Replikation, wie in Abbildung 3 dargestellt.

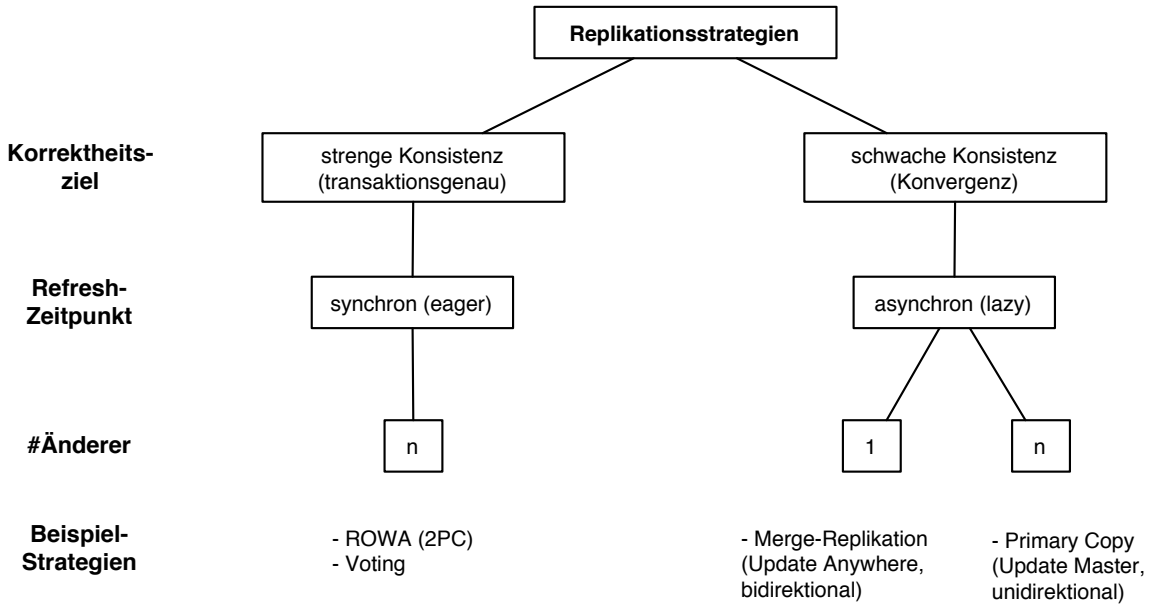

Abbildung 3: Klassifikation der Replikationsstrategien

3.2.1 Synchrone Verfahren

In synchronen Replikations-Verfahren werden Konflikte vor dem Commit der ändernden Transaktion erkannt. Dadurch können die ACID-Eigenschaften auf direkte Weise garantiert werden. Der Hauptnachteil ist der Overhead der Kommunikation zwischen den Knoten, der vor allem die Antwortzeit merklich erhöht.

Bei synchroner Replikation werden alle Replikate aktualisiert, wenn eine Transaktion eine beliebige Instanz eines Objekts ändert. Es gibt keine Inkonsistenzen und Anomalien und es muss kein weiterer Kopienabgleich gemacht werden. Durch das Setzen von Sperren auf jedem Replikat werden Konflikte entdeckt und führen je nach Sperrstrategie zu Wartesituationen oder auch Deadlocks.

Wenn alle Replikate erreichbar sind, bekommt man bei Lesezugriffen immer die aktuellste Version. Sind Teile des Replikationsverbunds nicht erreichbar, können prinzipiell veraltete Kopien entstehen; meistens sind in so einem Fall aber Aktualisierungen nicht erlaubt. Nicht erreichbare Knoten werden dann später mit aktuellen Daten versorgt, wenn sie wieder online sind.

Auch wenn alle Replikate immer online sind, können Änderungstransaktionen aufgrund von Deadlocks scheitern oder durch Sperrkonflikte lange verzögert werden.

Dies führt nach [GHOS96] zu schwerwiegenden Problemen, wenn man eine Replikation mit sehr vielen Knoten und vielen Änderungstransaktionen betreibt. In einem Netzwerk mit N Knoten, von denen jeder ein Replikat aller Objekte hat und eine feste Anzahl Transaktionen pro Sekunde initiiert, muss jeder Knoten bei einer Änderungstransaktion die Änderungen zu den anderen $N - 1$ Knoten senden. Das führt zu einer Erhöhung der Transaktionsgröße um den Faktor N und die Anzahl der Aktualisierungen der Knoten wächst um den Faktor N^2. Dieses nichtlineare Wachstum zeigt deutlich, dass sich synchrone Replikationsverfahren für die weiter oben beschriebenen Anwendungsszenarien nicht eignen:

1. Mobile Knoten können, wenn sie offline sind, keine synchrone Replikation verwenden
2. Die Wahrscheinlichkeit von Deadlocks und damit von Transaktionsabbrüchen steigt mit einer größeren Anzahl von Replikaten sehr stark an.

3.2.2 Asynchrone Verfahren

Asynchrone Replikation führt zu guten Antwortzeiten für die Änderungstransaktion, weil die Verbreitung der Änderungen erst nach dem Ende der Transaktion, also nach dem Commit, stattfindet.

Beim *Primary-Copy*-Ansatz sind Änderungen nur auf dem jeweiligen Master-System des Replikationsverbunds möglich. Von dort aus werden sie an alle Sekundärkopien oder Slave-Systeme verteilt. Für das Transaktionsmanagement und die Sperrverwaltung gelten deshalb dieselben Regeln wie in einem zentralen DBMS, was die Situation erheblich vereinfacht. Für das Szenario der Skalierbarkeit von Leselast durch mehrere Datenbankinstanzen ist dieses Verfahren sehr gut geeignet, solange die Änderungshäufigkeit im Master-System nicht allzu groß ist. Durch das Master-System wird allerdings ein Single-Point-of-Failure eingeführt. Dieser Nachteil kann abgeschwächt werden, wenn beim Ausfall des Masters einer der Slaves die Master-Rolle übernimmt oder zugewiesen bekommt.

Bei dezentralen Updates – dem *Update-Everywhere*-Ansatz – kann im Prinzip jede Kopie auf jedem Knoten aktualisiert werden. Dabei werden nur lokale Sperren gesetzt und nur lokale Transaktionen durchgeführt. Die Weitergabe der Änderungen muss dann synchronisiert werden und kann zu Konflikten führen, z. B. wenn zwei Knoten unabhängig voneinander dasselbe Objekt ändern. Bei mobilen Einsatzszenarien der Replikation wird in der Regel die Unterstützung der *Update-Everywhere*-Eigenschaft erwartet. Andernfalls hätte man lokal nur Read-Only-Datenbestände verfügbar. Will man z. B. Kundendaten oder Bestellinformationen auch im mobilen Endgerät ändern können, so erfordert dies die Unterstützung einer bidirektionalen Replikation. Hierbei treten Änderungskonflikte auf, die erkannt und

behandelt werden müssen, damit keine Änderungen verloren gehen (*Lost Update*). Hierfür gibt es mehrere Strategien.

Dazu wird das Änderungsobjekt (meist eine Tabellenzeile) mit einer Versionskennung versehen (Versionsnummer oder auch Timestamp). Bei der Propagation von Änderungen werden nicht nur der neue Zustand des Änderungsobjekts übertragen, d. h. die aktualisierte Tabellenzeile, sondern auch die alte und neue Versionskennung. Änderungskonflikte können dann dadurch entdeckt werden, dass im Zielsystem das Änderungsobjekt bereits mit einer neueren Versionskennung vorliegt. In diesem Fall darf die Änderung nicht übernommen werden, es muss eine Konfliktbehandlung erfolgen. Hierzu werden unterschiedliche Verfahren verwendet:

Neueste Änderung gewinnt

Bei Timestamp-basierten Versionskennungen überschreibt die Version mit einem neueren Zeitstempel ältere Versionen. Neben dem generellen Problem eines synchronisierten Zeittakts in verteilten Systemen kann diese Strategie zu Lost Updates führen, sie erfüllt aber das Konvergenz-Kriterium.

Master-System gewinnt

Auch bei bidirektionaler Replikation kann man im Falle von Änderungskonflikten ein Master/Slave-Prinzip ausprägen und z. B. in mobilen Szenarien den Änderungen der Firmenzentrale den Vorrang vor Änderungen auf dem mobilen Endgerät geben. (Dieses Master/Slave-Verhalten wird dann lediglich zur Konfliktauflösung angewendet und darf nicht mit der unidirektionalen Master/Slave-Replikation verwechselt werden, die nur eine Änderungsrichtung zulässt.) Auch bei dieser Strategie sind Lost Updates möglich, die Konvergenz der Kopien ist aber sichergestellt.

Benachrichtigung des Benutzers

Bei einem Änderungskonflikt wird zunächst die „Master-System gewinnt"- oder „Neueste Änderung gewinnt"-Strategie angewandt. Um den Lost-Update-Effekt zu vermeiden oder abzumildern, werden dann beide Versionen des Änderungsobjekts an den „Verlierer" zurückgeschickt. Benutzer mobiler Endgeräte hätten dann z. B. die Chance, ihre Änderung noch einmal abzuschicken oder die Änderung der Zentrale zu übernehmen oder auch beide Änderungen abzumischen, wenn sie unterschiedliche Spalten einer Tabellenzeile betreffen.

3.3 Klassische Verfahren

Die klassischen synchronen Verfahren der Replikation, die sich aus dem Gebiet der verteilten Datenbanken für die Replikation entwickelt haben, werden hier nur kurz beschrieben, da sie für ein heutigen Einsatz der Replikation keine große Bedeutung haben.

3.3.1 Read-One-Write-All-Verfahren (*ROWA*)

Der einfachste Weg, alle Kopien stets auf dem gleichen Stand zu halten, ist es, bei Änderungen alle Kopien innerhalb derselben Transaktion zu aktualisieren. Da alle Kopien gleichzeitig gesperrt sind, können auch keine inkonsistenten oder veralteten Zustände von Lesetransaktionen gelesen werden. Die Lesetransaktion kann sich eine beliebige Kopie auswählen. Die Dauer einer Änderungstransaktion, insbesondere die Durchführung des 2PC-Protokolls, ist allerdings durch die langsamste Kopie bestimmt.

Um die 1-Kopien-Serialisierbarkeit zu gewährleisten, kann jedes konventionelle Synchronisations-Verfahren auf den Knoten eingesetzt werden, da die Kopien, die ja innerhalb der Transaktion geändert werden, im Prinzip normale Datenelemente sind.

Das ROWA-Verfahren ist dadurch sehr *extrem*, dass keinen Kompromiss versucht; es ist für Lesetransaktionen ideal und für Schreibtransaktionen sehr schlecht. Außerdem müssen alle Kopien erreichbar sein, damit ein Update möglich ist. Die Verfügbarkeit des Systems sinkt also mit jeder zusätzlichen Kopie. Damit ist das ROWA-Verfahren in der reinen Form für den praktischen Einsatz nicht geeignet.

3.3.2 Auswahl der Update-Kopie

Im Prinzip – ROWA ausgenommen – werden bei der Durchführung der Aktualisierung nicht alle Kopien benötigt. Die Verfahren unterscheiden sich darin, nach welcher Strategie die benötigten Kopien ausgewählt und ihre Anzahl bestimmt wird.

Verfahren mit vorbestimmter Kopie Eine bestimmte Kopie wird als Primärkopie (auch Master genannt) festgelegt, sie ist die Originalversion, alle anderen sekundären Kopien (Slaves) sind nur davon abgeleitet. Ein Update erfolgt zweistufig: Die Slaves sperren und ändern nur den Master, der dann die Änderungen nach erfolgreichem Commit an die Slaves verteilt. Problematisch ist hier das aktuelle lokale Lesen der Slaves und der Ausfall des Masters.

Da es aus Sicht der Aktualisierungstransaktion nur eine Kopie gibt, ist die Gewährleistung der 1-Kopien-Serialisierbarkeit relativ einfach durch konventionelle Synchronisationsverfahren auf dem Master machbar.

Bekanntester und ältester Vertreter dieser Kategorie der Verfahren ist das Primary-Copy-Verfahren [Ston79]:

Das Primary-Copy-Verfahren ist ein asymmetrisches Verfahren mit synchronen Änderungen einer Primärkopie. Pro Primärkopie existieren n Sekundärkopien. Ein Ziel des Verfahren ist es, möglichst viele Lesezugriffe auf den lokalen Kopien direkt durchzuführen, um den häufigen Lesezugriff zu optimieren.

Die primäre Kopie verwaltet die Schreib- und Lesesperren und ist die Referenzkopie, was die Aktualität der Daten angeht. Die Änderungen einer erfolgreichen Schreibtransaktion auf dem Primärkopie-Rechner werden nach dem Commit (und somit asynchron) an alle Sekundärkopien verteilt. Beim Lesen teilt der Primärkopie-Rechner die aktuelle Versionsnummer der angeforderten Kopie bei Gewährung der Lesesperre dem lesenden Sekundärkopie-Rechner mit, der daraufhin entweder die aktuelle lokale Kopie liest oder eine aktuellere Version vom Primär-System.

Abstimmungsverfahren Ein Update auf einer Kopie wird nur dann durchgeführt, wenn die Transaktion eine Mehrheit der Stimmen der Kopien gewinnen kann (also z. B. geeignet zu sperren). Die Kopien können alle gleichbehandelt (ungewichtete Stimmen) oder teilweise bevorzugt[1] werden (gewichtete Stimmen). Die Verfahren unterscheiden sich außerdem nach der Bestimmung der zur Mehrheit (Quorum) nötigen Stimmen. Entweder wird das Quorum statisch – also vorher fest vorgegeben – oder dynamisch mit einer Bestimmung zur Laufzeit festgelegt.

Um die 1-Kopien-Serialisierbarkeit zu gewährleisten, muss man – da ja ein Teil der Kopien asynchron aktualisiert wird – fordern, dass das Update stets auf einer aktuellen Kopie basiert und dass durch Updates, die in falscher Reihenfolge eintreffen, keine Lost Updates auftreten. Praktisch alle Abstimmungsverfahren bedienen sich hierzu transaktions- und objektbezogener Zeitstempel oder Versionsnummern, um auch den Aspekt Zeit zu berücksichtigen.

Das Majority-Consensus-Verfahren [Thom79] war das erste Replikationsverfahren, bei dem ein Abstimmungsverfahren eingesetzt wurde. Jedes Replikat kann gleichzeitig von mehre-

[1] Beispielsweise können besonders robusten Stationen oder Stationen, auf denen viele der relevanten Transaktionen ablaufen, mehrere Stimmen erhalten.

ren Transaktionen gelesen, jedoch nur von einer Transaktion geändert werden. Die Bestimmung des aktuellen Objektzustandes geschieht über eine Versionsnummer. Im Fehlerfall ist die Verarbeitung weiter möglich, solange die Mehrheit der Replikate noch erreichbar ist.

Das Protokoll arbeitet nun wie folgt: Jede Kopie erhält eine bestimmte Anzahl von Stimmen und das Lesen und Schreiben erfordert eine bestimmte Anzahl von Stimmen. Die Änderung eines Objekts wird nur dann durchgeführt, wenn die entsprechende Transaktion in der Lage ist, eine Mehrheit von Kopien hierfür zu gewinnen, d. h. geeignet zu sperren.

Dafür ist jede Transaktion mit einem global eindeutigen Zeitstempel versehen und jedes Datenobjekt ist mit einem Zeitstempel versehen, der die zuletzt durchgeführte Änderung wiedergibt. Die Knoten sind durch einen logischen Ring verbunden, entlang dessen die Entscheidung vorangetrieben wird. So ist jeder Knoten über die Entscheidung des Vorgängers informiert.

Jede Update-Transaktion führt die Änderungen zunächst lokal durch und schickt dann eine Liste aller geänderten Objekte mit jeweiligen Zeitstempeln und einem Transaktionszeitstempel entlang des Rings an alle Knoten. Die Änderung wird dann permanent gemacht, wenn die Mehrheit der Knoten zustimmt. Die anderen Knoten stimmen nach einem Vergleich der Zeitstempel mit „Ok“ oder „Abgelehnt“. Der Knoten, der dem Antrag die Mehrheit verschafft hat, verschickt eine globale Commit- bzw. Abort-Meldung mittels Broadcast an alle Knoten.

Man sieht leicht, dass dieser Ansatz hohe Kommunikationskosten zwischen den Replikaten sowohl für Lese- als auch Schreibtransaktionen erfordert. Dafür wird ein Ausgleich der Leistungsfähigkeit zwischen Lese- und Änderungstransaktionen erreicht, d. h., ein Teil des Overheads wird von den Änderungstransaktionen auf die Lesetransaktionen verlagert.

3.4 Zusammenfassung

Generell gibt es zwei Klassen von Replikationsverfahren, synchrone und asynchrone. Synchrone Replikationsverfahren sind im wesentlichen im Umfeld verteilter Datenbanksysteme entstanden und erprobt worden. In der Praxis kommen sie wegen ihrer schlechten Skalierbarkeit und engen Kopplung der beteiligten Systeme kaum zur Anwendung. Asynchrone Replikationsverfahren dominieren in den typischen Anwendungsszenarien für replizierte Datenbestände. Die losere Kopplung der replizierten Systeme wird möglich, wenn man den strikten Transaktionsbegriff für die Replikate aufgibt und ihn durch den schwächeren Konsistenzbegriff der Konvergenz ersetzt.

4 Implementierungen von Replikationsverfahren

Alle Replikationsverfahren unterstützen die Phasen *Capture*, *Staging* und *Consumption*, wie sie in Abbildung 1 illustriert werden.

Je nach den Ausgangsvoraussetzungen und Zielsetzungen der Replikation kommen dafür unterschiedliche Implementierungen zum Einsatz, die zusammenfassend für die drei Szenarien in der Tabelle 1 dargestellt werden. Die Replikation kann auf unterschiedlich große Granulate angewendet werden. In den Anwendungsszenarien mit hoher Verfügbarkeit und zur Skalierung von Leselast wird typischerweise die ganze Datenbankinstanz, also alle Tabellen einer Datenbank, repliziert. In den mobilen Szenarien mit asynchroner Replikation sind es eher einzelne Tabellen oder auch Projektionen bzw. Selektionen von Tabellen, die als Replikat in ein mobiles Endgerät propagiert werden.

Szenario 1: Hohe Änderungslast, Datenbank als Replikationsgranulat, unidirektionale Replikation	
Capture	Redo-Log
Staging	Medium des Redo-Logs (Dateisystem, Platte)
Consumption	Standby-System
Szenario 2: Geringe Änderungslast, Datenbank als Replikationsgranulat, unidirektionale Replikation	
Capture	Kommando-Log
Staging	Medium des Kommando-Logs (Dateisystem, Platte)
Consumption	Weitere DB-Instanz zur Skalierung von Leselast
Szenario 3: Geringe Änderungslast, Tabelle als Replikationsgranulat, bidirektionale Replikation	
Capture	Insert-, Update-, Delete-Trigger mit Schattentabelle
Staging	Dedizierte Datenbankinstanz (Replication Server) - Queue-Prinzip - Publish/Subscribe-Prinzip
Consumption	Mobile Endgeräte

Tabelle 1: Varianten der Implementierung

4.1 Szenario 1: Hochverfügbarkeit

In Hochverfügbarkeitsszenarien, in denen meistens eine hohe Änderungslast verkraftet werden muss, erfolgt der Abgriff der Änderungen über das Redo-Log, das das Quellsystem (*Primary Copy*) ohnehin schreibt. Das Zielsystem (*Secondary Copy*, Standby-System) liest die Log-Einträge mit und zieht die dort aufgeführten Änderungen und Transaktionen nach. Das Mitlesen der Log-Einträge erfolgt typischerweise asynchron, um den Transaktionsdurchsatz im Quellsystem nicht zu bremsen. Dies hat zur Folge, dass das Zielsystem je nach der aktuellen Lastsituation in der Regel einige Transaktionen hinterherläuft. Beim Ausfall des Quellsystems muss das Zielsystem daher erst noch die ausstehenden abgeschlossenen Transaktionen nacharbeiten und die letzten abgebrochenen Transaktionen zurückrollen, bevor es als neues Primärsystem einspringt. Diese Übernahme erfolgt üblicherweise im Sekundenbereich. Voraussetzung hierfür sind eine Cluster-Konfiguration des zugrunde liegenden Betriebssystems, die durch eine „Heartbeat"-Überwachung den Ausfall des Primärsystems erkennt, diesen als Transaktionsabbruch an die Anwendungs-Sessions des Datenbanksystems signalisiert und durch einen IP-Switch die Anwendungs-Sessions transparent zum Sekundärsystem umlenkt. Das Redo-Log muss sowohl vom Primär- wie auch vom Sekundärsystem aus erreichbar sein, was z. B. durch Dual-Ported Disks bewerkstelligt wird. Zur Initialisierung einer Standby-Konfiguration zieht das Sekundärsystem eine Kopie der Datenbank, z. B. durch das Einspielen eines Datenbank-Backups.

Derartige Hochverfügbarkeits-Konfigurationen werden von nahezu allen kommerziellen Datenbanksystemen unterstützt. Einige von ihnen erlauben auch Lesezugriffe auf das Sekundärsystem für Decisions-Support- und Data-Warehouse-Anwendungen.

4.2 Szenario 2: Skalierung der Leselast

Zur Verteilung von Leselast auf eine größere Anzahl von identischen Datenbankinstanzen (und damit nur geringer Änderungslast) ist das technisch einfachste Verfahren das Schreiben eines Kommando-Logs, der alle Änderungen in der Form von SQL-Statements mit Literalen enthält. MySQL verwendet z. B. diese Methode, um ausgehend von einer Masterkopie mehrere Slavekopien zu aktualisieren[2]. Das Kommando-Log enthält durchnummerierte Einträge, die von den Slave-Systemen durch einen Remote-Filesystem-Zugriff gelesen und verarbeitet werden. Dabei erleichtert die Nummerierung das Wiederaufsetzen im Fehlerfall. Die Slave-Systeme werden mit einer Backup-Kopie des Masters initialisiert

[2] Da MySQL (mit MyISAM) kein Transaktionskonzept unterstützt, ist dieses Kommando-Log auch das für Recovery-Zwecke verwendete Redo-Log.

und erhalten dann eine Delta-Übertragung der Änderungen über das Kommando-Log. Fällt ein einzelner Slave aus, wird die Skalierbarkeit des Systemverbunds nur geringfügig beeinflusst. Je nach dem Typ des Ausfalls setzt er die Delta-Übertragung fort oder der Slave wird bei einem Medienverlust neu initialisiert. Fällt der Master aus, kann man einem Slave die Rolle des Masters zuweisen und der ausgefallene Master übernimmt später eine Slave-Rolle.

Für die Skalierung der Leselast eignen sich auch die nachfolgend beschriebenen Replikationsverfahren für mobile Endgeräte. Sie sind flexibler, aber auch mit mehr Administrations- und Systemaufwand verbunden.

4.3 Szenario 3: Mobile Computing

Um mehr oder weniger statische Daten in größeren Unternehmen „vor Ort“ zu bringen, d. h. Replikate in Filialen, Außenstellen oder auch auf mobilen Endgeräten zu unterstützen, hat Sybase als einer der ersten Anbieter das Konzept eines *Replication Server* eingeführt. Dies ist eine eigene Datenbankinstanz zum Zwischenspeichern (Staging) und Vermitteln der aufgelaufenen Änderungen wie in Abbildung 4 dargestellt. Ein solcher Replication Server entkoppelt die Quell- und Zielsysteme maximal; Änderungen können auch propagiert werden, wenn das Quellsystem gerade nicht verfügbar ist. Viele Sybase-Kunden haben den Replication Server auch dazu genutzt, um z. B. Daten aus einer Oracle-Instanz in SQL-Server-Instanzen zu propagieren. Der Sybase Replication Server hat dazu auf der Capture- und Consumption-Seite Adaptoren für die marktüblichen DBMS-Produkte. Inzwischen unterstützen fast alle DBMS-Anbieter derartige Replication-Server-Konzepte, z. B. bei IBM DB2 den Data Propagator und beim MS SQL Server den Synchronization Manager sowie den Oracle Replication Manager.

Ein solcher Replication Server erlaubt es, die Replikation auf Tabellen-Niveau zu definieren. Dazu werden im Zielsystem Insert-, Update- und Delete-Trigger angelegt, die Änderungen in einer lokalen Schattentabelle protokollieren. Diese Capture-Technik eignet sich gut, wenn nur wenige Tabellen repliziert werden sollen und die Änderungslast insgesamt gering ist. Für eine komplette Datenbankinstanz ist ein triggerbasierter Ansatz aufwändiger als ein logbasierter Abgriff der Änderungen, aber immer noch praktikabel, wenn die Änderungslast gering ist. Die Trigger-Programmierung ermöglicht es auch, die Replikation auf Ausschnitte (Projektion, Selektion) der beteiligten Tabelle zu reduzieren, was vor allem für regionale Daten in mobilen Endgeräten, z. B. für Vertriebsregionen Sinn macht.

Neben der üblichen Übertragung von Änderungen von einem Quellobjekt zu einem Zielobjekt, die dem Prinzip einer Warteschlange folgt, erlaubt ein Replication Server durch das Staging auch Übertragungen nach dem Publish/Subscribe-Prinzip. Hierbei werden Quelle und Ziel nicht mehr durch eine Übertragungs-Queue gekoppelt, sondern ein Publisher stellt die Änderungsdaten bereit, damit sie von beliebigen Subscribern zu einem beliebigen Zeitpunkt abgerufen werden können. Der Publisher aktualisiert von Zeit zu Zeit sein Datenangebot, das unter verschiedenen Themengebieten (Topics, Channels) gruppiert wird. Der Subscriber prüft bei jedem Abruf, ob er noch die neuesten Daten der ihn interessierenden Themengebiete hat und erhält unter Umständen eine Aktualisierung durch eine Übertragung der Deltas. Dazu wird jede Datenbankänderung in eine Message verwandelt und im Replication Server zwischengespeichert. Jeder Publishing-Vorgang aktualisiert eine Versionsnummer des zugehörigen Topics. Ein Subscriber ruft dann die für ihn neuen Messages aus den ihn interessierenden Topics ab und verarbeitet sie als Datenbankänderung. Auch die Initialisierung des Datenbestands eines mobilen Endgeräts kann durch eine solche Übertragung von Messages erfolgen.

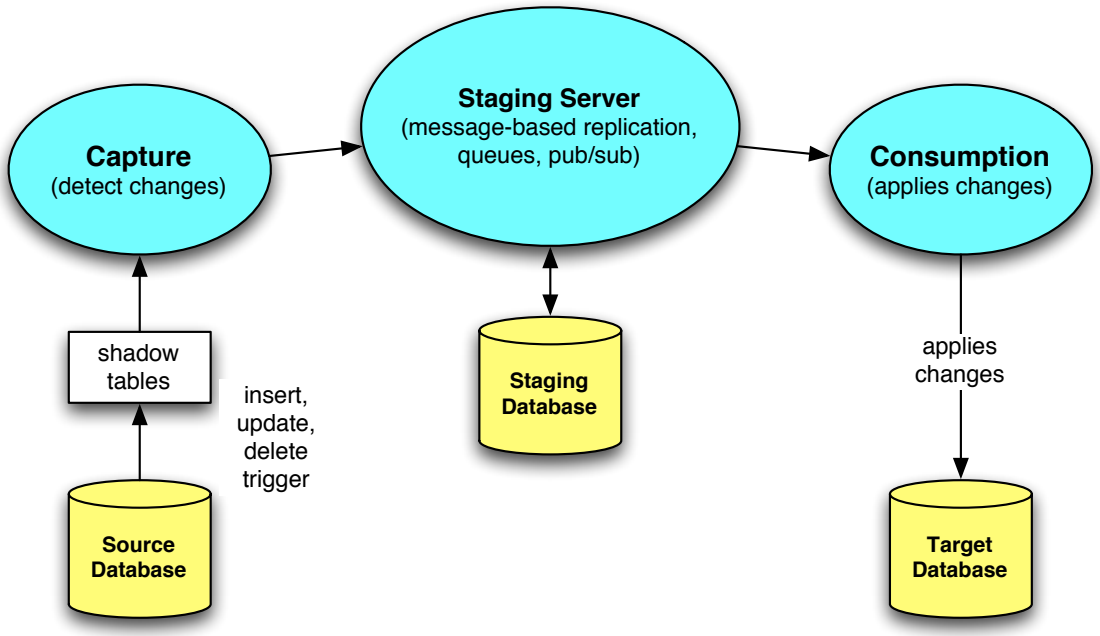

Abbildung 4: Prinzip des dedizierten *Replication Server*

5 Zusammenfassung

Die Unterstützung von Replikation ist aus dem heutigen IT-Alltag nicht mehr wegzudenken. Sie findet sowohl anwendungsbasiert, betriebssystembasiert (File Transfer) wie auch datenbankbasiert statt. Für die datenbankbasierte Replikation finden sich Anwendungsszenarien im Umfeld von Hochverfügbarkeitssystemen, der Skalierung von Leselast und der Unterstützung mobiler Endgeräte.

In all diesen Szenarien dominiert die asynchrone Replikation, die den in DBMS üblichen Transaktionsbegriff nicht auf die Replikate ausdehnt, sondern sich aus Skalierungsgründen mit dem schwächeren Konsistenzbegriff der Konvergenz zufrieden gibt. Dadurch sind die Systeme in einem Replikationsverbund nur noch lose miteinander gekoppelt, was gegenüber der engen Kopplung verteilter Datenbanksysteme ein Vorteil ist und ihren praktischen Betrieb erleichtert. Die verwendeten Replikationsverfahren profitieren von der Forschung auf dem Gebiet verteilter Datenbanksysteme, auch wenn ihre Einsatzszenarien deutlich einfacher sind. Ihre praktische Bedeutung ist dafür größer. Die Unterstützung der Replikation durch dedizierte Replikations-Server stellt die neueste Entwicklung dar und adressiert in erster Linie Replikate auf mobilen Endgeräten.

Literatur

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha, „The Dangers of Replication and a Solution", in: *Proc. of the SIGMOD Conf*, 1996, S. 173-182.

[ABKW98] T. A. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, „Replication, Consistency, and Practicality: Are these Mutually Exclusive?", in: *Proc. of the SIGMOD Conf*, 1998, S. 484-495.

[Thom79] R. H. Thomas, „A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", in: *ACM Transactions on Database Systems, Vol. 4*, 1979, S. 180-209.

[Ston79] M. Stonebraker, „Concurrency Control and Consistency of Mutliple Copy Databases in Distributed INGRES", in: *IEEE Trans. Software Eng., Vol. 5*, 1979, S. 188-194.

[BKRSS99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz, „Update Propagation Protocols for Replicated Databases", in: *Proc. of the SIGMOD Conf*, 1999, S. 97-108.

[Härder02] Theo Härder, *Vorlesung: Verteilte und parallele DBS*, Vorlesungsskript 2002 URL: http://wwwdvs.informatik.uni-kl.de/courses/VDBS/.

[Dadam96] Peter Dadam, *Verteilte Datenbanken und Client/Server-Systeme*, Springer 1996.

[KE04] A. Kemper, A. Eickler, *Datenbanksysteme - Eine Einführung*, Oldenbourg 2004.

[DB2] IBM, *DB2 DataPropagator - Product Overview*,
URL: http://www-306.ibm.com/software/data/dpropr/. Stand 04.07.04.

[MSSQL] Microsoft, *SQL Server: Technical Resources*,
URL: http://www.microsoft.com/sql/techinfo/default.asp. Stand 04.07.04.

[Oracle02] Oracle White Paper, *Replication with Oracle Streams*, 2002, URL: http://otn.oracle.com/products/dataint/pdf/streams_replication_twp.pdf.

[Sybase] Sybase, *Replication Server*,
URL: http://www.sybase.com/products/middleware/replicationserver. Stand 04.07.04.

[SyWhite] Sybase White Paper, *Replication Strategies: Data Migration, Distribution and Synchronization*, URL: http://www.sybase.com/content/1028711/6143_whitepaper_v2.pdf.

[MySQL] MySQL Manual, *Replication in MySQL*, URL: http://dev.mysql.com/doc/mysql/en/Replication.html. Stand 04.07.04.