Evaluation von Transaktionsmodellen und transaktionsorientierten Reintegrationsstrategien für mobile Datenbanksysteme

– Studienarbeit –

Matthias Liebisch
Matthias.Liebisch@informatik.uni-jena.de

Betreuer
Dipl.-Inf. Christoph Gollmick
Prof. Dr. Klaus Küspert

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Datenbanken und Informationssysteme
Ernst-Abbe-Platz 1–4
07743 Jena

Dezember 2001

Kurzfassung

Die wachsenden Möglichkeiten in mobilen Anwendungen schließen sowohl lesende Zugriffe auf Daten als auch in stärker werdendem Maße ändernde Zugriffe auf Daten, die in einer Datenbank gehalten werden, ein. Deshalb müssen Konzepte aus Datenbankmanagementsystemen wie die Transaktionsverarbeitung auch für mobile Geräte verfügbar sein. Durch das teils verbindungslose Arbeiten von Clients ist eine Synchronisation zwischen Client und Server erforderlich, die unter anderem dazu dient, Änderungskonflikte durch konkurrierende Transaktionen zu erkennen und zu lösen. In dieser Arbeit wird ein Überblick über die wichtigsten Transaktionsmodelle und transaktionsorientierten Reintegrationsstrategien für mobile Datenbanksysteme gegeben. Dabei werden die Probleme und Herausforderungen im Umgang mit lokal, d.h. ohne Verbindung zum Server, auf replizierten Daten arbeitenden Clients aufgezeigt und die zur Verfügung stehenden Lösungsansätze miteinander verglichen. Neben der Vorstellung der Modelle werden diese nach Kriterien wie unterstützte Isolationslevel, Zusicherung von ACID-Eigenschaften, Voraussetzungen an Hardware und Kommunikationsumgebung sowie Aufwand für die Beschaffung zusätzlich notwendiger Informationen für die Konfliktauflösung untereinander bewertet. Zum Abschluß erfolgt ein Vergleich mit Realisierungen von Transaktionsmodellen bei Unterstützung mobiler Clients in ausgewählten Datenbank- und Dateisystemprodukten.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **1**

2.1 Mobile Datenbankmanagementsysteme 1
2.2 Transaktionsverarbeitung 2
2.3 Replikationsverfahren 3
2.4 Wiedereinbringstrategien 3
2.4.1 Datenorientiert 4
2.4.2 Transaktionsorientiert 5
2.5 Fehler- und Konfliktklassifikationen 6
2.5.1 Konfliktarten 6
2.5.2 Konflikterkennnung und -auflösung 6
2.5.3 Phänomene und Anomalien 7

3 Bewertungskriterien **10**

3.1 Architektur 10
3.1.1 Client 10
3.1.2 Netzwerkschicht 10
3.1.3 Server 11
3.2 Nutzer 11
3.3 Datenhaltung und -zugriff 12
3.3.1 Verwendung von Zusatzinformationen 12
3.3.2 Abschwächung der ACID-Eigenschaften 12
3.3.3 Konflikterkennung und Konfliktauflösung 12
3.4 Implementierung 13

4 Übersicht von Transaktionsmodellen **13**

4.1 Klassifikation 13
4.1.1 Verbundenheit 13
4.1.2 Schwache Verbundenheit 13
4.1.3 Unverbundenheit 14
4.1.4 Zielstellungen 14
4.2 Two-Tier und Bayou 14
4.2.1 Architektur 14
4.2.2 Nutzer 15

4.2.3 Datenhaltung und -zugriff . . . 16
4.2.4 Implementierung . . . 19
4.3 Multiversionsmodell . . . 19
4.3.1 Architektur . . . 19
4.3.2 Nutzer . . . 19
4.3.3 Datenhaltung und -zugriff . . . 20
4.3.4 Implementierung . . . 23
4.4 Semantische Transaktionen . . . 24
4.4.1 Architektur . . . 24
4.4.2 Nutzer . . . 25
4.4.3 Datenhaltung und -zugriff . . . 25
4.4.4 Implementierung . . . 27
4.5 Schwache/strikte Transaktionen . . . 28
4.5.1 Architektur . . . 28
4.5.2 Nutzer . . . 28
4.5.3 Datenhaltung und -zugriff . . . 28
4.5.4 Implementierung . . . 31
4.6 Isolation-Only Transactions . . . 31
4.6.1 Architektur . . . 31
4.6.2 Nutzer . . . 31
4.6.3 Datenhaltung und -zugriff . . . 32
4.6.4 Implementierung . . . 34
4.7 Offen geschachtelte Transaktionen . . . 34
4.7.1 Architektur . . . 34
4.7.2 Nutzer . . . 35
4.7.3 Datenhaltung und -zugriff . . . 35
4.7.4 Implementierung . . . 37
4.8 Kangaroo Transactions . . . 37
4.8.1 Architektur . . . 37
4.8.2 Nutzer . . . 38
4.8.3 Datenhaltung und -zugriff . . . 39
4.8.4 Implementierung . . . 40

5 Bewertung der Modelle **40**
5.1 Architektur 40
5.2 Nutzer 41
5.3 Datenhaltung und -zugriff 41
5.4 Implementierung 44

6 Exkurs Datenbank- und Dateisystemprodukte **44**
6.1 Abgrenzung 44
6.2 IBM DB2 DataPropagator 46
6.2.1 Architektur 46
6.2.2 Datenhaltung und -zugriff 46
6.3 Sybase Adaptive Server Anywhere 7.0.0 und SQL Remote 47
6.3.1 Architektur 47
6.3.2 Datenhaltung und -zugriff 48
6.4 Coda 49

7 Zusammenfassung und Ausblick **50**

Literatur **53**

Abbildungsverzeichnis **55**

Tabellenverzeichnis **55**

1 Einleitung

Mobilität von Nutzern prägt in wachsendem Maße das berufliche und alltägliche Leben. Damit werden auch und vorallem Datenbankmanagementsysteme (DBMS) konfrontiert. Grundlage jedes DBMS sind Transaktionen als Ausführungseinheit, denen die ACID-Eigenschaften (siehe Abschnitt 2) zugesichert werden. Die Hauptaufgabe der Transaktionsverarbeitung ist die Integritätssicherung der zugrundeliegenden Datenbank durch Synchronisation des Mehrbenutzerbetriebes. Diese Arbeit befaßt sich mit der Vorstellung und Bewertung von Modellen für das transaktionsorientierte Reintegrieren von vorher replizierten und lokal auf dem Client veränderten Daten, d.h. es wird auf eine Situation nach der Replikation von Daten aufgebaut. Der Schwerpunkt der Evaluation liegt dabei auf den Methoden zur Erkennung und Lösung von möglicherweise auftretenden Konflikten beim Wiedereinbringen der Änderungen.

Die Arbeit ist wie folgt gegliedert. In Abschnitt 2 werden die Grundlagen zum Verständnis mobiler Datenbankmanagementsysteme gegeben. Es folgt in Abschnitt 3 eine Vorstellung der zur Evaluation herangezogenen Bewertungskriterien. Schließlich werden in Abschnitt 4 eine Anzahl von Modellen vorgestellt. Anschließend werden im Abschnitt 5 die vorgestellten Modelle anhand der Kriterien bewertet und in einer Übersicht zusammengefasst. Mit dem Abschnitt 6 werden Transaktionsmodelle in Produkten kurz vorgestellt und den Forschungsmodellen gegenübergestellt. Abschließend werden die Ergebnisse der Arbeit in Abschnitt 7 zusammengefaßt und Ausblicke auf weiterführende Fragestellungen gegeben.

2 Grundlagen

2.1 Mobile Datenbankmanagementsysteme

Ein Mobiles DBMS wird hier weniger als eine Weiterentwicklung aus einem verteilten Datenbanksystem [Dad96] betrachtet. Vielmehr ist zur Unterstützung mobiler Clients eine Erweiterung des traditionellen Client/Server-Modells notwendig [Gol00], in welchem die Anfrageaktivitäten vom Client ausgehen. Dabei sind mobile Clients im Unterschied zu traditionellen festverbundenen Clients nur schwach über ein drahtloses Medium (z.B. Funk) oder periodisch über ein festes Netzwerk (z.B. LAN oder Modemeinwahl) mit dem Server verbunden. Aufgrund der hohen Kosten und geringeren Bandbreite drahtloser Kommunikationsmittel im Vergleich zu Festnetzverbindungen, ist die ständige Aufrechterhaltung einer Verbindung zum Server während der Arbeit auf dem mobilen Client im allgemeinen keine Lösung und teilweise unmögliche aufgrund von nicht abgedeckten Funkbereichen. Es ist deshalb für ein unabhängiges Arbeiten notwendig, Daten lokal auf mobilen Clients zu *replizieren*.

Dabei sind nicht nur Leseoperationen aus dem lokalen Cache des mobilen Gerätes sondern auch Änderungsoperationen auf den Replikaten zu ermöglichen. Um auch für mobilen Clients das Arbeiten unter dem Transaktionskonzept [GR93] zu gewährleisten, muß es möglich sein, Transaktionen lokal auf dem Client abschließen zu können (LOCAL COMMIT) im Gegensatz zum klassischen Client/Server-Modell, in dem es nur dem Server erlaubt ist, Änderungen abzuschließen. Andererseits ist klar, daß dadurch nicht die kompletten ACID-Eigenschaften [GR93] zugesichert werden können, da beim Wiedereinbringen und Konsolidieren der lokalen Änderungen auf dem Server Konflikte mit anderen konkurrierenden

Änderungen entstehen können. Insbesondere ist dabei die Dauerhaftigkeit der lokal abgeschlossenen Transaktionen nicht gewährleistet. Die bei der Reintegration entstehenden Konflikte müssen durch das DBMS erkannt und möglichst automatisch gelöst werden.

2.2 Transaktionsverarbeitung

Eine Transaktion ist nach [GR93] eine Zusammenfassung beliebiger Operationen, d.h. lesender (*read*) und schreibender (*write*) Zugriffe auf ein Objekt, mit den sogenannten *ACID*-Eigenschaften (atomicity, consistency, isolation, durability):

Atomarität : Die Änderungen einer Transaktion am Zustand der Datenbank sind atomar: entweder alle oder keine der enthaltenen Änderungen werden durchgeführt.

Konsistenz : Eine Transaktion überführt von einem korrekten (konsistenten) Zustand der Datenbank in einen nächsten, der sich nicht notwendigerweise vom vorherigen unterscheiden muß.

Isolation : Jede Transaktion sieht sich trotz physischem Mehrbenutzerbetrieb isoliert im logischen Einbenutzerbetrieb ausgeführt, d.h. entweder vor oder nach einer anderen Transaktion.

Dauerhaftigkeit : Sobald eine Transaktion erfolgreich mit COMMIT abgeschlossen wurde, kann der Benutzer sicher sein, daß seine abgeschlossenen Änderungen dauerhaft in der Datenbank eingebracht sind und Ausfälle überleben.

Eine Transaktion wird durch ein meist implizites BOT (*begin of transaction*) eingeleitet und wird bei erfolgreicher Abarbeitung mit einem expliziten COMMIT abgeschlossen. Treten Fehler während der Transaktion auf, so können alle Operationen mittels ABORT rückgängig gemacht werden. Die Absicherung obiger Eigenschaften vom Datenbankmanagementsystem geschieht durch verschiedene Protokolle und Mechanismen. Beispielsweise kann durch den Einsatz eines Sperrprotokolls die Isolationseigenschaft gewahrt werden, die Dauerhaftigkeit bei Externspeicherfehlern kann durch eine Recovery-Komponente gewährleistet werden.

Werden im Mehrbenutzerbetrieb ohne Koordination parallel laufenden Transaktionen ausgeführt, so gefährdet man die Konsistenz der Datenbank. Aus Performance-Gründen ist eine serielle Ausführungsfolge der Transaktionen nicht praktikabel, deswegen findet eine Verzahnung der einzelnen Transaktionen statt. Dabei muß gesichert werden, daß weiterhin die ACID-Eigenschaft für jede Transaktion gelten und das Kriterium der Serialisierbarkeit erhalten wird, d.h. daß es zu den verzahnt ausgeführten Transaktionen mindestens eine ergebnisäquivalente serielle Ausführungsfolge gibt. Um dies zu gewährleisten, werden im zentralen Fall, d.h. einer Datenbank auf einem Server, üblicherweise pessimistische Sperrverfahren wie das 2-Phasen-Sperrprotokoll (*2-phase locking, 2PL*) [GR93] verwendet, die zwar mehrere mögliche serielle Ausführungsfolgen ausschließen, dafür aber bei strikter Einhaltung Serialisierbarkeit parallel laufender Transaktionen garantieren. Allerdings werden aus Performance-Gründen verschiedene Stufen der Aufweichung solcher Protokolle verwendet, wodurch einige Phänomene und Anomalien des unkontrollierten Mehrbenutzerbetriebes (Abschnitt 2.5.3) während der Transaktionsverarbeitung zugelassen werden.

In mobilen Umgebungen bei unverbundenem Arbeiten und lokalem COMMIT können obige Sperrverfahren aufgrund der den Umgebungen inhärenten Eigenschaften nicht angewandt werden. Eine Lösungsmöglichkeit liegt in der Abschwächung der Serialisierbarkeitsforderung. Dadurch werden mehr Ausführungsfolgen möglich, die dann wiederrum Anomalien und Phänomenen des unkontrollierten Mehrbenutzerbetriebes aufweisen können (Abschnitt 2.5.3). Um trotzdem die Konsistenz der Datenbank zu erhalten, sollte das Kriterium der *1-Kopien-Serialisierbarkeit* nach [Dad96] gelten, angewandt für Transaktionen in mobilen Szenarien:

> Ein Schedule S von abgeschlossenen Transaktionen, die auf einem möglicherweise mehrfach repliziert gespeicherten Datenbestand ausgeführt wurden, heißt dann und nur dann *1-Kopien-serialisierbar*, wenn es mindestens eine serielle Ausführung der Transaktionen aus S auf einer Datenbank ohne Replikate gibt, welche, angewandt auf denselben Ausgangszustand, die gleiche Ausgabe sowie denselben Endzustand erzeugt.

2.3 Replikationsverfahren

Hier unterscheidet man zwischen optimistischen und pessimistischen Replikationsverfahren [PB00]. Letztere benutzen übliche Sperrverfahren um Konflikte zwischen konkurrierenden Zugriffen zu vermeiden, erfordern aber eine permanente Verbindung vom Client zum Server, da für jede Änderung von Daten eine Sperranforderung gebraucht wird. Wegen obiger Eigenschaften drahtloser Kommunikation ist diese Vorgehensweise für mobile Szenarien nur bedingt anwendbar und wird insbesondere nur von den Modellen mit einer schwachen Verbindung über ein drahtloses Medium implementiert. Um dieses Problem zu umgehen, könnten auch alle replizierten Daten nach Trennung des Clients auf dem Server nach dem Prinzip des *check-in/check-out* [LP83] gesperrt werden und damit eine permanente Verbindung überflüssig machen. Da diese Sperren möglicherweise erst Tage später beim Wiederanmelden des mobilen Nutzers freigegeben werden, stellt sich dadurch allerdings ein drastischer Rückgang der Verfügbarkeit ein.

Optimistische Replikationsverfahren gestatten es hier im Gegensatz zu pessimistischen Verfahren, lokale Änderungen und lokalen Transaktionsabschluß durchzuführen und damit ein unverbundenes Arbeiten zu ermöglichen. Allerdings werden durch den Verzicht von Sperren zugunsten der Verfügbarkeit neue Probleme in Form von Änderungskonflikten (siehe Abschnitt 2.5) beim Wiedereinbringen erzeugt. Diese entstehen aufgrund der nicht notwendigerweise disjunkten Replikation zwischen den mobilen Clients, wenn die lokalen Änderungen auf dem Server reintegriert und abgeglichen werden und müssen von der Wiedereinbringkomponente erkannt und gelöst werden.

2.4 Wiedereinbringstrategien

Das Wiedereinbringen lokaler Änderungen als Teil der Synchronisation zwischen Server und Client kann daten- oder transaktionsorientiert geschehen.

2.4.1 Datenorientiert

Diese Variante wird im allgemeinen von kommerziellen Produkten für mobile Umgebungen unterstützt, wie die Übersicht in [Fan00] zeigt. Hierbei werden als einzubringende Inhalte nur die *images* einzelner Datenelemente betrachtet. Lokal werden zum einen die Werte aller replizierten Daten nach der letzten Synchronisation (*before image*) und zum anderen die jeweils aktuellen Werte der Objekte gespeichert (*after image*), die durch eine Folge von Änderungen entstanden sind. Damit stehen zur Konflikterkennung und -auflösung nur diese zwei *images* des Clients sowie der aktuelle Zustand des Servers zur Verfügung. Um Konflikte zu lösen, werden Regeln benötigt, die auf bekannten Eigenschaften der zugrundeliegenden Operationen aufbauen.

Vorteile dieser Methode sind schnelle und billige Implementierbarkeit sowie die Tatsache, daß der Client nicht die möglicherweise vielen ausgeführten Operationen speichern muß, sondern nur die zwei *images* pro Objekt. Wesentliche Nachteile sind jedoch Unflexibilität bei sich ändernder oder unbekannter Transaktionssemantik und die Gefährdung der Atomarität einer Transaktion durch die Konfliktbehandlung pro Objekt und nicht pro Transaktion. Schließlich ist auch eine Zuordnung von eingebrachten Transaktionen zu Nutzern nicht mehr möglich, da im Zuge der Reintegration alle akzeptierten Änderungen mehrerer Transaktionen zu einer neuen Transaktion zusammengesetzt werden. Das folgende modifizierte Beispiel aus [PB00] veranschaulicht den Ablauf.

Wir betrachten ein Datenelement $x = 1$. Ein Client repliziert dieses Element x, trennt die Verbindung und läßt zwei Transaktionen $T_{1'} : x = x + 1$ und $T_{2'} : x = x + 2$ hintereinander laufen. Währenddessen ändert eine Transaktion $T_1 : x = 2$ auf dem Server das betrachtete Datenelement. Danach verbindet sich der Client wieder mit dem Server. Im folgenden bedeuten

- $r_i[x] = j$: Transaktion i liest x mit Wert j
- $w_i[x] = j$: Transaktion i ändert x auf den Wert j
- c_i : COMMIT der Transaktion i

und wir erhalten den Ablauf:

Server : $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1$

Client : repliziere $x = 1, r_{1'}[x] = 1, w_{1'}[x] = 2, c_{1'}, r_{2'}[x] = 2, w_{2'}[x] = 4, c_{2'}$, abgleichen

Der nun anlaufende Abgleich-Prozeß findet einen Konflikt, weil sich das *before image* von x des Clients, also 1, vom akutellen Wert von x auf dem Server, d.h. 2, unterscheidet. Zur Konfliktauflösung benutzt das System Regeln. Eine solche Regel könnte hier zum Beispiel die Kommutativität von Addition und Subtraktion nutzen:

$$x^s_{new} = x^s_{old} + x^c_{new} - x^c_{down}$$

Dabei stellt x^s_{new} den neuen Wert dar, der nach der Konfliktauflösung in die konsolidierte Datenbank auf dem Server geschrieben wird, x^s_{old} ist der Wert in der Datenbank zum Zeitpunkt vor der Konfliktauflösung, x^c_{new} ist der aktuelle Wert des Clients und x^c_{down} ist der Wert von x, den der Client repliziert hat. Mit Anwendung dieser Regel auf dem Server

$$x^s_{new} = 2 + 4 - 1 = 5$$

erhalten wir folgenden Ablauf nach der Konfliktauflösung:

Server : $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, w_c[x] = 5, c_c$

Mit T_c wurde eine neue Transaktion gebildet, die die Änderungen des Clients reflektiert und in der $T_{1'}$ und $T_{2'}$ aufgehen. Dies ist ein *blind write*, weil die Menge der gelesenen Objekte (*read set*) der ursprünglichen Client-Transaktionen bedeutungslos werden und nur noch das *before image* und *after image* zählen. Folglich geht der Transaktionskontext des Clients verloren und damit verlieren auch die durch das ACID-Konzept zugesicherten Eigenschaften ihre Gültigkeit, vorallem die Atomarität, da kein Rückschluß mehr auf die Originaltransaktionen möglich ist. Wegen obigen Gründen ist die datenorientierte Reintegration für das Einbringen lokal ausgeführter Transaktionen nicht geeignet und wir betrachten nur den im folgenden beschriebenen transaktionsorientierten Ansatz.

2.4.2 Transaktionsorientiert

Beim transaktionsorientierten Ansatz werden die einzelnen Operationen aufgezeichnet (z.B. im Log) und deren Wirkung unter Beachtung des Transaktionskontexts sukzessive eingebracht. Treten Konflikte bei enthaltenen Operationen auf, gibt es verschiedene Konfliktlösungsmöglichkeiten wie in Abschnitt 2.5.2 aufgezeigt wird. Analog müssen für abhängige Transaktionen (siehe Abschnitt 2.5.1) Auflösungsmechanismen gefunden werden. Der einfache Ansatz des Abbruchs einer konflikterzeugenden Transaktion ist bei einer hohen Konfliktwahrscheinlichkeit beispielsweise durch lange unverbundene Phasen nicht akzeptabel.

Unter Nutzung der Semantik von Transaktionen oder Operationen ist eine Entwicklung flexiblerer Einbringstrategien möglich, die z.T. auf Kosten der 1-Kopien-Serialisierbarkeit [Dad96] die Menge der eingebrachten Änderungen erhöht. Dem stehen die Nachteile gegenüber, daß mehr Aufwand auf der Clientseite für das Aufzeichnen kompletter Transaktionen vorgenommen werden muß und serverseitig einerseits die Entwicklung und andererseits die Ausführung komplexer Algorithmen für den Abgleich der Änderungen die Performance des Systems belasten. Um auch hier die Funktionsweise und Unterschiede zur datenorientierten Reintegration zu verdeutlichen, greifen wir wieder auf das oben eingeführte Beispiel zurück. Verwendet man die Semantik der betrachteten Transaktionen unter entsprechender Anpassung der gelesenen Werte erhalten wir folgenden Ablauf nach der Konfliktauflösung:

$$\textbf{Server :} \quad \begin{array}{l} w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, \\ r'_{1'}[x] = 2, w'_{1'}[x] = 3, c'_{1'}, \\ r'_{2'}[x] = 3, w'_{2'}[x] = 5, c'_{2'} \end{array}$$

Ist die Semantik der Transaktionen nicht bekannt und wird Serialisierbarkeit gefordert, so muß $T_{1'}$ abgewiesen werden, weil das *read set* von $T_{1'}$ auf dem Client $\{x = 1\}$ nicht mit dem *read set* auf dem Server $\{x = 2\}$ übereinstimmen würde. Allerdings kann immer noch Transaktion $T_{2'}$ eingebracht werden, da hier beide *read sets* übereinstimmen und man würde folgenden Ablauf erhalten:

$$\textbf{Server :} \quad \begin{array}{l} w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, \\ r'_2[x] = 2, w'_2[x] = 4, c'_2 \end{array}$$

2.5 Fehler- und Konfliktklassifikationen

2.5.1 Konfliktarten

Sollen lokale und unabhängige Änderungen ermöglicht werden, erfordert dies die Replikation von Ausschnitten der Datenbank wie in den letzten Abschnitten aufgezeigt. Dadurch kann es beim Einbringen zu folgenden Konflikten zwischen konkurrierenden Transaktionen und deren Operationen kommen. Die Erkennung und Auflösung dieser Konflikte wird im nächsten Abschnitt (2.5.2) betrachtet.

read-write: Eine Transaktion T_1 auf dem mobilen Client liest das replizierte Objekt x und führt aufgrund des gelesenen Wertes Änderungen auf anderen Objekten aus. Eine zweite Transaktion T_2 ändert währenddessen auf dem Server den Wert von x. Will nun Transaktion T_1 seine Änderungen wiedereinbringen, bauen diese auf einem veralteten Wert von x auf (*out-of-date*).

write-write: Zwei Transaktionen zweier Clients T_1 und T_2 ändern das Objekt x in unverträglicher Weise, d.h. für das Objekt x gibt es zwei verschiedene neue Werte, wobei aber nur ein Wert dem Objekt zugewiesen werden kann. Außnahme bildet hier das Multiversionsmodell, welches in Abschnitt 4.3 vorgestellt wird.

integrity violation: Dieser kaum beachtete Konflikt entsteht durch das Replizieren eines Ausschnitts A der Datenbank D. Besteht zwischen einem Element $x \in A$ und einem Element $y \in D \backslash A$ eine Integritätsbedingung, kann diese bei lokalen Änderungen von x auf dem Client nicht überprüft werden. Beim Wiedereinbringen kann durch Integritätsverletzung ein Konflikt entstehen.

Transaktionsabhängigkeit: Wird Transaktion T_1 lokal und damit *vorläufig* abgeschlossen und baut eine nachfolgende Transaktion T_2 des Clients auf Ergebnissen von T_1 auf, dann entsteht eine Abhängigkeit zwischen diesen beiden Transaktionen. Wird anschließend T_1 aufgrund eines Konfliktes nicht eingebracht, verliert sie ihre Dauerhaftigkeit. Damit hat aber T_2 möglicherweise Ergebnisse von T_1 gesehen und verarbeitet, die auf der konsolidierten Datenbank nie existierten.

2.5.2 Konflikterkennnung und -auflösung

Konflikte können auf verschiedene Arten erkannt werden, die unterschiedlich aufwendig sind und mehr oder wenig viele zusätzliche Informationen benötigen:

- Abhängigkeitsgraph
  Alle Transaktionen werden formal in einem Graphen dargestellt und aufwendig auf Abhängigkeit getestet. Enthält er Kreise, ist ein Konflikt entdeckt.

- Akzeptanzkriterium
  Dies stellt ein vom Benutzer zu definierendes Kriterium dar, daß die Ergebnisse einer Transaktion auf die semantische Gültigkeit innerhalb eines Betrachtungsrahmens prüft. Beispielsweise könnte eine solche Bedingung lauten: „Es gibt keine negativen Gesamtzahlen gebuchter Flugzeugplätze.“

- Snapshotsuche
  Diese Möglichkeit für Multiversionsmodelle sucht den zum *read set* der Transaktion konsistenten Snapshot in der Datenbank. Bei wachsender Größe einer Datenbank und der damit steigenden Anzahl der Snapshots ist dies eine sehr aufwendige Variante.
- Vergleich des *before/after image*
  Hier wird der Wert eines Objektes zum Zeitpunkt der Replikation mit dem Wert zum Zeitpunkt des Einbringens der Transaktion verglichen. Unterscheiden sich diese, liegt ein Konflikt vor.

Ausgehend vom einfachen Abbruch einer Transaktion, stehen sehr viel umfassendere und komplexere Konfliktauflösungen zur Verfügung:

- Abbruch der Transaktion
  Diese einfachste Lösung verletzt gleichzeitig aber offensichtlich die Dauerhaftigkeit der für Transaktionen zugesicherten ACID-Eigenschaften. Dadurch kann es bei Transaktionsabhängigkeiten zum fortgesetzten Abbruch (*cascading abort*) kommen.
- Einschränkung der Serialisierbarkeitsforderungen
  Werden Transaktionen unter schwächeren Isolationsleveln ausgeführt, können mehr Folgen von Transaktionen zugelassen werden. Allerdings sind auch mehr Phänomene und Anomalien als unter strikteren Isolationsleveln möglich.
- Abschwächung der ACID-Eigenschaften
  Hierunter fällt vorallem die Möglichkeit, die Atomarität einer Transaktion abzuschwächen bzw. aufzuheben, um mehr Transaktionen im Konfliktfall einbringen zu können. Beispielsweise könnten alle Operationen ohne Konflikte einer eingebrachten Client-Transaktion übernommen werden und für die restlichen Operationen benutzerdefinierte Konfliktauflösungen aufgerufen werden. Damit wird allerdings die Originaltransaktion verändert.
- Regelbasierte Auflösung
  Hierunter fallen einfache Regeln wie *first committer wins*, d.h. diejenige Transaktion, die zuerst eine Änderung an einem Objekt vorgenommen hat, wird beibehalten, alle anderen werden zurückgewiesen. Eine andere Regel ist *server wins*, d.h. bei konkurrierenden Änderungen zwischen einer Server- und Clienttransaktion, wird die Servertransaktion übernommen.
- Manueller Eingriff
  Diese Möglichkeit sollte immer zur Verfügung stehen, ist allerdings für viele Transaktionen bei entsprechender Nutzeranzahl und Konfliktwahrscheinlichkeit nicht praktikabel. Hinzu kommt, daß ein Nutzer schon wieder unverbunden sein kann, wenn eine seiner Transaktionen als Konflikt erkannt wird.

2.5.3 Phänomene und Anomalien

Im klassischen Client/Server-Modell müssen die Transaktionen von parallel arbeitenden Nutzern durch die Transaktionsverarbeitung synchronisiert werden, um Konfliktmöglichkeiten gering zu halten. Dies kann auf unterschiedlichen Stufen der Nebenläufigkeit geschehen, den sogenannten *Isolationsleveln.* Wie in [BBG+95] beschrieben, dienen unterschiedliche Isolationslevel zum Abwägen zwischen hoher Parallelität von Transaktionen und

hoher Konsistenz von Daten. Niedrigere Isolationslevel erlauben dabei mehr Nebenläufigkeit mit der Gefahr, daß Transaktionen inkonsistente Datenbankzustände sehen und damit ungültige Daten erzeugen können. In traditionellen Datenbankmanagementsystemen wird die klassische Serialisierbarkeit als Isolationslevel SERIALIZABLE durch das 2-Phasen Sperrprotokoll erreicht. Damit verhindert man das Auftreten der klassischen Anomalien und Phänomene LOST UPDATE, DIRTY READ, FUZZY READ und PHANTOM.

Aufgrund der Eigenschaften mobiler Umgebungen kann hier ein strikter Isolationslevel wie SERIALIZABLE unter dem Aspekt der Performance und Verfügbarkeit nicht verwendet werden, da hierzu üblicherweise Sperrprotokolle im Einsatz sind. Als Lösung bieten sich schwächere Isolationslevel mit gewissen Anomalien oder die Abschwächung anderer Eigenschaften von ACID an. Ziel ist es deshalb einen praktikablen Isolationslevel zu verwenden, der zwischen Verfügbarkeit und Vermeidung möglichst vieler Anomalien zur Konsistenzerhaltung einen Kompromiß findet. Diese sollen im folgenden nach einer Notation von [ALO00, PB00] erläutert werden. Es bedeuten $w_i[x]$ eine Schreiboperation von Transaktion i auf dem Objekt x, $r_i[x]$ eine Leseoperation von Transaktion i auf dem Objekt x, c_i ein COMMIT und a_i ein ABORT der Transaktion i.

Dirty Read P1: Transaktion T_1 ändert das Objekt x. Transaktion T_2 liest dieses aktuell geänderte Objekt. Wenn jetzt entweder T_1 ein ABORT vollzieht oder Objekt x ein weiteres Mal ändert und danach mit COMMIT abschließt, hat Transaktion T_2 eine nicht existierende bzw. inkonsistente Version von Objekt x gelesen, formal:

$$\textbf{P1:} \quad w_1[x] \ldots r_2[x] \ldots (a_1 \wedge c_2)$$

oder

$$w_1[x] \ldots r_2[x] \ldots w_1[x] \ldots (c_1 \wedge c_2)$$

Nonrepeatable/Fuzzy Read P2: Transaktion T_1 liest Objekt x. Transaktion T_2 ändert oder löscht danach Objekt x und schließt mit COMMIT ab. Wenn Transaktion T_1 nun versucht, Objekt x nochmals zu lesen, erhält sie einen veränderten Wert oder muß feststellen, daß Objekt x nicht mehr existiert, formal:

$$\textbf{P2:} \quad r_1[x] \ldots w_2[x] \ldots c_2 \ldots r_1[x] \ldots c_1$$

Phantom P3: Transaktion T_1 liest eine Menge von Objekten, die einem Suchprädikat P genügen. Transaktion T_2 erzeugt mehrere Objekte, die ebenfalls das Prädikat P erfüllen und teilweise in der bereits von T_1 gelesenen Menge liegen, und schließt mit COMMIT ab, noch bevor T_1 am Ende der Leseaktion ist. Wenn T_1 mit dem Lesen fertig ist, hat es eine inkonsistente Anzahl von Objekten gelesen (z.B. eine ungerade Anzahl von verheirateten Personen), da einige „im Rücken" von T_1 eingefügt wurden, formal:

$$\textbf{P3:} \quad r_1[P] \ldots w_2[x \in P] \ldots c_2 \ldots r_1[P] \ldots c_1$$

Lost Update P4: Transaktion T_1 und T_2 lesen Objekt x und T_1 ändert Objekt x aufgrund des gelesenen Wertes. Danach ändert T_2 das Objekt x und schließt mit COMMIT ab. Folglich wird die Änderung von Transaktion T_1 überschrieben, ganz egal, ob T_1 mit COMMIT oder ABORT beendet wird, formal:

$$\textbf{P4:} \quad r_2[x] \ldots r_1[x] \ldots w_1[x] \ldots w_2[x] \ldots c_2$$

Isolationslevel	P1	P2	P3	P4	P0	A5A	A5B
READ UNCOMMITTED	✓	✓	✓	✓	–	✓	✓
READ COMMITTED	–	✓	✓	✓	–	✓	✓
REPEATABLE READ	–	–	✓	–	–	–	–
SNAPSHOT ISOLATION	–	–	✓	–	–	–	✓
SERIALIZABLE	–	–	–	–	–	–	–

✓. . . möglich –. . . nicht möglich

Tabelle 1: Isolationslevel und Phänomene

Nach einer Spezifikation von ANSI/ISO SQL92 [ANS92] werden vier Isolationslevel definiert, READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ und SERIALIZABLE, die sich wie im ersten Teil von Tabelle 1 gezeigt anhand der jeweils erlaubten Phänomene **P1** bis **P4** charakterisieren lassen. Wie zu erkennen ist, werden erst durch den strengsten Isolationslevel SERIALIZABLE alle Phänomene verboten, sodaß die Konsistenz der Datenbank nicht gefährdet wird.

Weitere Anomalien, über die Definition von ANSI/ISO SQL92 hinausgehend und teilweise spezielle Ausprägungen vorangegangener Phänomene, werden in [BBG+95] beschrieben. Diese erweiterten Phänomene lassen sich zur Beschreibung feinerer Isolationslevel verwenden, die von einigen Transaktionsmodellen zur Reintegration lokal abgeschlossener Transaktionen verwendet werden. Beispielsweise arbeitet das Multiversionsmodell [PB00] auf dem SNAPSHOT ISOLATION-Level, welcher sich zwischen SERIALIZABLE und READ COMMITTED einordnet und nicht direkt vergleichbar ist mit REPEATABLE READ. Die erweiterten Isoaltionslevel und Phänomene sind im zweiten Teil der Tabelle 1 zu sehen.

Dirty Write P0: Transaktion T_1 ändert das Objekt x. Transaktion T_2 ändert danach ebenfalls das Objekt x, bevor T_1 ein COMMIT oder ABORT vollzieht. Falls dann T_1 oder T_2 mit ABORT abschließt, ist unklar, welches der korrekte Wert von Objekt x ist. Da die Änderung durch T_2 auf einem *blind write* basiert, d.h. einer Änderung unter der Annahme, daß das Objekt x gewisse Bedingungen erfüllt, kann dadurch die Datenbank in einen inkonsistenten Zustand versetzt werden, weil eben genau die Bedingungen nicht mehr gültig waren, formal:

$$\textbf{P0:} \quad w_1[x] \ldots w_2[x] \ldots (a_1 \vee a_2)$$

Read Skew A5A: Transaktion T_1 liest Objekt x und im Anschluß ändert Transaktion T_2 die beiden Objekte x und y, zwischen denen eine Bedingung C in der Datenbank existiert, und schließt mit COMMIT ab. Falls nun T_1 das Objekt y liest, sieht sie möglicherweise eine Verletzung der Bedingung C und erzeugt ein inkonsistentes Ergebnis, formal:

$$\textbf{A5A:} \quad r_1[x] \ldots w_2[x] \ldots w_2[y] \ldots r_1[y] \ldots (c_1 \vee a_1)$$

Write Skew A5B: Transaktion T_1 liest die Objekte x und y, welche die Bedingung C erfüllen. Eine weitere Transaktion T_2 liest ebenfalls x und y, ändert x und schließt mit COMMIT ab. Dann ändert T_1 das Objekt y und schließt ebenfalls mit COMMIT ab, womit möglicherweise die Bedingung C verletzt ist, formal:

$$\textbf{A5B:} \quad r_1[x] \ldots r_1[y] \ldots r_2[x] \ldots r_2[y] \ldots w_2[x] \ldots w_1[y] \ldots (c_1 \wedge c_2)$$

Einige dieser Phänomene des unkontrollierten Mehrbenutzerbetriebes treten auch in mobilen Szenarien mit optimistischer Replikation beim Wiedereinbringen als Ergebnis einer Konfliktauflösung auf. Beispielsweise kann die Änderung eines Clients durch eine gleichzeitige Änderung desselben Objektes auf dem Server beim Einbringen verloren gehen, wenn eine Regel der Art *server wins* vorliegt. Dann entspricht dies dem LOST UPDATE-Phänomen. Ähnlich verhält es sich mit einer lokal abgeschlossenen Transaktionen T, deren Ergebnisse von nachfolgenden Transaktionen des Client gelesen werden können. Wird beim Einbringen diese Transaktion T aufgrund eines Konfliktes abgebrochen, haben nachfolgende Transaktionen Daten gelesen, die eigentlich nicht existieren dürften (DIRTY READ).

3 Bewertungskriterien

Zur Bewertung und Vorstellung der einzelnen Modelle werden vier große Kriterienkomplexe herangezogen. In einem ersten Teil sollen Fragen zu den Voraussetzungen an die Architekturmerkmale der gesamten Umgebung bestehend aus mobilem Client, dem Server und der verbindenden Netzwerkschicht geklärt werden. Im Anschluß wird das vom Modell geforderte Verhalten des mobilen Nutzers selbst genauer untersucht. Im dritten und größten Teil sollen in einem Fragenkatalog Kriterien für die Datenhaltung bzw. den Zugriff auf Daten für die einzelnen Modelle betrachtet werden. Abschließend findet eine Bewertung zu einer möglichen Implementierung statt. Aufgrund unterschiedlich gesetzter Schwerpunkte der einzelnen Modelle kann ein Kriterium im Umfang der Betrachtung für verschiedene Modelle unterschiedlich ausfallen.

3.1 Architektur

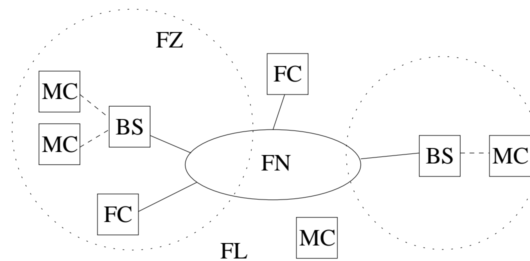
3.1.1 Client

Welche Anforderungen werden an die Hardware, insbesondere an Speicher, Energieversorgung und Rechenleistung (CPU) eines mobilen Gerätes gestellt? Benötigt beispielsweise ein Modell eine schwache Verbindung, d.h. eine Verbindung über ein drahtloses Medium, zwischen Client und Server zum Abschließen von Transaktionen, dann impliziert dies entsprechend hohe Anforderungen an die Energieversorgung des Clients aufgrund der energieaufwendigen Funkverbindung.

3.1.2 Netzwerkschicht

- *Welche Verbindung zwischen Client und Server wird vom Modell für welche Operation vorausgesetzt?* Hier steht neben der geforderten Verbindungsart zum Abschließen lokaler Transaktionen vorallem eine detaillierte Betrachtung der Verbindungsanforderung für andere Operationen, wie eine normale Schreiboperation auf Daten, zur Diskussion.

- *Welche Netztopologie wird vorausgesetzt bzw. zugrundegelegt, um ein Modell zum Einsatz bringen zu können?* Beispielsweise kann das in [WC95] erläuterte und allgemein anerkannte Zellsystem für schwach verbundene Clients als Grundlage dienen. Dazu sei auf Abbildung 1 verwiesen. In diesem Szenarium sind traditionelle Clients



`FC` . . . verbundener Client `MC` . . . mobiler Client `BS` . . . Basisstation
`FN` . . . festes Netzwerk `FZ` . . . Funkzelle `FL` . . . Funkloch

Abbildung 1: Mobile Umgebung für schwach verbundene Clients

und Server (Basisstationen) untereinander festverbunden. Mobile Clients halten innerhalb des abgedeckten Funkbereichs den Kontakt zur Basisstation und damit dem festverbundenen Netzwerk über eine schwache Verbindung. Befindet sich ein mobiler Client außerhalb der Reichweite aller Basissationen, hat er keinerlei Verbindungsmöglichkeiten zum Netzwerk.

- *Für welche Aufgaben ist eine eventuell benötigte Middleware zuständig und aus welchen Komponenten besteht sie?* Im Szenarium der schwach verbundenen Clients in einem Zellsystem könnten die zuständigen Basisstationen neben dem Aufrechterhalten einer Verbindung zum Client auch für dessen Datenanfragen verantwortlich sein.

3.1.3 Server

- *Welche Architekturvoraussetzungen werden an das DBMS des Servers gestellt?* Hierbei wird beispielsweise geklärt, ob ein Modell auf dem allgemeinen Fall eines zentralen DBMS aufbaut oder ob es einen anderen Ansatz verwendet. Desweiteren fällt hierunter die Erläuterung besonderer Merkmale, wie z.B. die Verwendung eines Multiversionsmodells.
- *Welche Aufgaben übernimmt das DBMS auf dem Server?* Hier wird unter anderem geklärt, ob beispielsweise die Konflikterkennung auf dem Client, auf dem Server oder auf beiden zu gewissen Teilen durchgeführt wird.

3.2 Nutzer

- *In welchem Maße wird der mobile Aspekt vor dem Nutzer verborgen, d.h. welche Änderungen im gewohnten Arbeitsverhalten müssen in Kauf genommen werden?* Die Betrachtung dieser Frage zielt vorallem auf das Maß der Involvierung des Nutzers bei der Lösung von Problemen ab, die durch das unverbundene oder auch schwach verbundene Arbeiten entstehen, wie z.B. der Zugriff auf nichtverfügbare Daten, eine manuelle Konfliktauflösung oder der Zellwechsel des Nutzers.
- *Gibt es Einschränkungen in der Mobilität des Nutzers?* Hierbei wird beispielsweise geklärt, wie häufig ein unverbundener Nutzer mit dem System synchronisiert werden

muß oder ob ein Verlassen von Funkzellen schwach verbundener Clients unterstützt wird.

3.3 Datenhaltung und -zugriff

3.3.1 Verwendung von Zusatzinformationen

- *Welche Eigenschaften von Daten, Operationen und Transaktionen werden zur Unterstützung eines Modells gefordert?* Die Untersuchung dieses Kriteriums trägt entscheidend zur Beantwortung der Frage nach der Implementierbarkeit und Praxistauglichkeit eines Modells bei. Beispielsweise sind Forderungen nach der Fragmentierbarkeit von Objekten, Kommutativität von Operationen oder der Existenz von Kompensationstransaktionen nicht trivial und lassen sich mitunter nur schwer realisieren.

- *Welche Zusatzinformationen werden benötigt?* Ein weiterer wichtiger Punkt in der Entscheidung, ob ein Modell praxistauglich ist oder nicht. Dabei wird geklärt, welche zusätzlichen Informationen, wie z.B. die Menge der gelesenen Daten (*read set*) oder die Semantik von Transaktionen, benötigt werden.

- *Mit welchem Aufwand sind die zusätzlich benötigten Informationen zu beschaffen und wofür werden sie verwendet?* Hierbei ist zu beachten, daß sich der Aufwand sowohl in erhöhtem Speicherbedarf als auch in Performance-Verlusten ausdrücken kann. Beispielsweise müßten zur Beschaffung des *read sets* einer Transaktion selbst Leseoperationen im Log protokolliert werden, was zur Folge hat, daß ein lesender Zugriff formal einen schreibenden Zugriff verursacht. Dies widerspricht dem Bestreben, die Anzahl der I/O-Operationen möglichst klein zu halten, da diese im allgemeinen die größten Kosten, d.h. Performance-Verluste, nach sich ziehen.

3.3.2 Abschwächung der ACID-Eigenschaften

- *In welcher Weise werden die ACID-Eigenschaften abgeschwächt?* Beispielsweise wird durch das unverbundene Arbeiten von Clients die Dauerhaftigkeit der lokal abgeschlossenen Änderungen gefährdet, wenn es beim Wiedereinbringen zu Konflikten kommt.

- *Unter welchem Isolationslevel arbeitet das Modell und welche Methoden werden zur Konsistenzsicherung eingesetzt?* Wie auch schon in traditionellen Transaktionsmodellen im verbundenen Client/Server-Betrieb werden unterschiedliche Abschwächungen von SERIALIZABLE als Isolationslevel zur Erhöhung der Performance und der Nebenläufigkeit eingesetzt, aber auch um die Anzahl einbringbarer Änderungen zu steigern.

3.3.3 Konflikterkennung und Konfliktauflösung

- *Wie werden vom Modell Konflikte erkannt?* Ein wichtiger Punkt beim Wiedereinbringen ist die Erkennung von Konflikten wie sie beispielsweise durch Leseabhängigkeiten verursacht werden können. Diese Frage klärt, welche Verfahren dazu vom

Modell unter Verwendung der oben aufgeführten zusätzlichen Informationen angewandt werden.

- *Welche Verfahren werden zur Konfliktlösung herangezogen und welche Informationen werden dazu benötigt?* Da eine manuelle Lösung von Konflikten nicht immer möglich ist, beispielsweise wenn der entsprechende Nutzer bei Erkennung eines Konfliktes nicht mehr erreichbar ist, sollen hier vorallem die vom Modell angebotenen Alternativen der automatischen Konfliktauflösungen betrachtet werden, wie z.B. die Übernahme der zuerst eingebrachten Änderung bei konkurrierenden Änderungen auf einem Objekt.

3.4 Implementierung

- *Inwieweit ist eine prototypische Implementierung praktikabel?* Neben der Vorstellung eines möglicherweise existierenden Prototypen soll aufgrund vorangegangener Bewertungen kurz festgestellt werden, inwiefern das betrachtete Modell für mobile Datenbanken geeignet ist. Eine ausführlichere Auswertung aller betrachteten Kriterien findet sich im Abschnitt 5.

4 Übersicht von Transaktionsmodellen

4.1 Klassifikation

In diesem Abschnitt sollen sowohl Teile von Transaktionsmodellen (Reintegrationskomponenten) als auch komplette Modelle vorgestellt und bewertet sowie anhand der für die Arbeit des Modells zugrundegelegten Verbindungsart zwischen Client und Server klassifiziert werden. Dabei können drei verschiedene Zustände unterschieden werden.

4.1.1 Verbundenheit

Dieser Fall entspricht dem klassischen Client/Server-Ansatz [Dad96] mit einer festen Verbindung (z.B. LAN) zwischen Client und Server und wird hier nicht betrachtet.

4.1.2 Schwache Verbundenheit

Hier wird über ein drahtloses Medium eine Verbindung zum Server entweder für jede Operation oder nur zum Transaktionsabschluß (COMMIT) benötigt. Dieses Medium hat typischerweise eine geringere Bandbreite gegenüber dem Festnetz, ist fehleranfälliger sowie kosten- und energieintensiv im Betrieb für die mobile Einheit. Insbesondere ist aufgrund des großen Protokoll-Overhead der Einsatz von 2-Phase-Commit (2PC) [GR93] für derartige Verbindungen mit geringer Übertragungskapazität nicht praktikabel. Dennoch bietet sich hier eine Form der pessimistischen Replikation an, wobei ein Abschluß der Transaktion zum COMMIT-Zeitpunkt über die schwache Verbindung stattfindet. Die Modelle dieser Kategorie legen die in Abbildung 1 gezeigte mobile Umgebung zugrunde.

4.1.3 Unverbundenheit

Der Client arbeitet völlig unabhängig auf replizierten Daten und es müssen eventuell entstehende Änderungskonflikte oder auch Konflikte bezüglich Integritätsverletzungen beim Wiedereinbringen erkannt und gelöst werden, wie dies einleitend im Abschnitt 2 dargestellt wurde. Dieses Konzept wird von der Mehrheit der in dieser Arbeit untersuchten Modelle verwendet.

4.1.4 Zielstellungen

Aus der Kategorie der Modelle mit Unterstützung von Unverbundenheit werden folgende Modelle vorgestellt:

- Two-Tier und Bayou (Abschnitt 4.2)
- Multiversionsmodell (Abschnitt 4.3)
- Semantische Transaktionen (Abschnitt 4.4)
- Schwache/strikte Transaktionen (Abschnitt 4.5)
- Isolation-Only Transactions (Abschnitt 4.6)

Diese versuchen mehr Transaktionen einzubringen als im beschriebenen klassischen Fall (Abschnitt 2) durch beispielsweise geschickte Konfliktauflösungsalgorithmen oder die Aufzeichnung und Verwendung von zusätzlichen Informationen.

Eine schwache Verbindung zwischen Client und Server legen folgende Modelle zugrunde:

- Offen geschachtelte Transaktionen (Abschnitt 4.7)
- Kangaroo Transactions (Abschnitt 4.8)

Sie haben gegenüber Modellen mit Unterstützung von Unverbundenheit vorallem die Ziele, einen Zellwechsel (*handoff*) mit entsprechender Kommunikation der Basisstationen zu unterstützen, die Datenverfügbarkeit durch sofortigen Abschluß über die schwache Verbindung zu erhöhen und die vorhandenen Ressourcen optimal zu nutzen, indem beispielsweise unnötiger Datentransfer über die geringe Bandbreite vermieden wird.

4.2 Two-Tier und Bayou

4.2.1 Architektur

Das in Bayou [DPS+94b] verwendete *Two-Tier*-Modell [GHOS96] ist eine Modifizierung des sogenannten *Master*-Replikationsschemas. Dabei wird jedem Datenobjekt ein Server als Besitzer (*Master*) zugewiesen, der das alleinige Recht hat, das entsprechende Objekt zu ändern. Hierbei ist es erlaubt, daß verschiedene Master auf unterschiedlichen Servern plaziert sind. Auf untereinander verbundenen Servern werden jeweils vollständige Kopien einer Datenbank vorgehalten. Clients können sich Daten lokal replizieren und darauf im

unverbundenen Zustand arbeiten, wobei vorläufig abgeschlossene Transaktionen (*tentative transactions*) vorläufige Versionen (*tentative versions*) der lokalen Daten erzeugen. Außer den dafür notwendigen Ressourcen zur Speicherung und Verarbeitung von Daten werden vom Modell keinerlei Anforderungen an die Clients gestellt. Allerdings findet unter Bayou keine explizite Trennung zwischen Client und Server statt, d.h. sobald ein Client eine vollständige Kopie der Datenbank vorhält, wird er von anderen Maschinen als Server betrachtet. Durch die Unterstützung des unverbundenen Arbeitens werden keine Verbindungen zwischen Client und Server zur Transaktionsverarbeitung benötigt. Durch diese fehlenden Anforderungen ist der Einsatz auch nicht durch eine vorliegende Netztoplogie beschränkt.

Als Middleware können die sogenannten Sekundärserver (*secondary server*), die für das vorläufige Akzeptieren und Propagieren der Änderungen des Clients verantwortlich sind, betrachtet werden. Diese halten zwar alle die vollständige Datenbank vor, aber nur auf einem ausgezeichneten Server je replizierter Datensammlung, dem Primärserver (*primary server*), werden alle vorläufigen Änderungen auf den gespeicherten Masterkopien endgültig festgeschrieben. Eine Kommunikation der Server zum Änderungsaustausch untereinander findet über eine Variante des zeitstempelbasierten Anti-Entropy-Verfahrens (*time stamped anti-entropy, TSAE)* von Golding [Gol92] statt und implementiert ein *peer-to-peer*-Protokoll. Somit ist das Gesamtsystem eine Variante eines verteilten DBMS, die Datenbank jedes einzelnen Servers kann dagegen von einem beliebigen System verwaltet werden, wie z.B. einem Dateisystem oder relationalem DBMS.

Das Aufgabenspektrum eines Sekundär-/Primärservers ist sehr groß und beinhaltet zum einen die Wiederausführung von vorläufigen Client-Transaktionen als Basistransaktionen (*base transactions*) auf den Basisversionen des Servers (*base versions*). Aufgrund der Wiederausführung auf aktuellen Daten kann eine lokale Transaktion unter Umständen ein anderes Ergebnis erzeugen. Für erfolgreiches Einbringen muß das neue Ergebnis einen Test durchlaufen, das sogenannte *Akzeptanzkriterium*, wie z.B. `Kontostand darf nicht negativ sein`. Wird das Akzeptanzkriterium verletzt, ist der Server für eine semantikbasierende Konflikterkennung und -auflösung und eventuelle Benachrichtigung des mobilen Clients sowie für die Weitergabe von neu aufgenommenen Änderungen an andere Server verantwortlich. Aufgrund der Gleichberechtigung von Client und Server wird ein zusätzlicher Kontext, der sogenannte Gültigkeitsbereich einer Transaktion (*transaction scope*), benötigt. Dieser garantiert, daß lokale Transaktionen nur diejenigen Daten ändern, deren Besitzer ein festverbundener Server oder der mobile Client selbst ist. Wäre dies nicht gesichert, müssten alle anderen mobilen Clients beim Synchronisieren eines mobilen Nutzers ebenfalls zum Netzwerk verbunden sein. Zusätzlich muß der Primärserver vorläufige Änderungen endgültig abschließen und somit eine stabile Version der Datenbank erzeugen, d.h. neueintreffende Änderungen können die gefundene Reihenfolge von endgültig abgeschlossenen Änderungen nicht verändern.

4.2.2 Nutzer

Ein Ziel von Bayou ist das Verbergen des mobilen Aspektes im unverbundenen Zustand, d.h. der Nutzer soll möglichst keinen Unterschied zum Arbeiten mit einem zentralen, hochverfügbaren DBMS bemerken. Darunter fällt die Konflikterkennung vom System und die anwendungsspezifische Konfliktauflösung auf dem Server. Allerdings sollten Anwendungen wissen, daß sie schwach konsistente Daten lesen und durchgeführte Änderungen zu einem

```
Bayou_Write(
  update = {
    insert, Meetings, 12/18/95, 10:00am, 60min, "Meeting Kevin"
  }
  dependency_check = {
    query = "SELECT key FROM Meetings WHERE day = 12/18/95
             AND start < 11:00am AND end > 10:00am",
    expected_result = EMPTY
  },
  mergeproc = {
    alternates = {12/18/95, 12:00pm};
    newupdate = {};
    FOREACH a IN alternates {
      # check if there would be a conflict
      IF (NOT EMPTY(
        SELECT key FROM Meetings WHERE day = a.date
        AND start < a.time + 60min AND END > a.time))
        THEN CONTINUE;
      # no conflict, can schedule meeting at that time
      newupdate = {insert, Meetings, a.date, a.time, 60min, "Meeting Kevin"};
      BREAK;
    }
    IF (newupdate = {})   # no alternate is acceptable
      THEN newupdate = {insert, ErrorLog, 12/18/95, 10:00am, 60min, "Meeting Kevin"};
    RETURN newupdate;
  }
)
```

Abbildung 2: Eine Bayou-Schreiboperation am Beispiel des Gruppenkalenders

späteren Zeitpunkt Konflikte erzeugen können, die mitunter ein manuelles Eingreifen zur Auflösung erfordern. Weiterhin benötigt Bayou eine nicht näher spezifizierte Anzahl von Anwendungen, die im Konfliktfall mit kommutativen Änderungen arbeiten oder Konflikte angemessen lösen können. Dies schränkt das Arbeitsfeld des Nutzers entsprechend ein. Dagegen kann sich der Nutzer ungebunden bewegen. Zum einen wird das vollständig unverbundene Arbeiten unterstützt und damit ist der Nutzer nicht an eventuelle Funkzellen gebunden, zum anderen erlaubt das sogenannte *read-any/write-any*-Prinzip von Bayou, daß Daten von einer Datenbankkopie auf einem beliebigen Server lokal repliziert werden und die durchgeführten Änderungen ebenfalls auf jeder Datenbankkopie eines beliebigen Servers eingebracht werden können. Die Lösungen von Problemen der somit schwach konsistenten replizierten Datenbanken werden im folgenden Abschnitt betrachtet.

4.2.3 Datenhaltung und -zugriff

Besondere Eigenschaften der verwendeten Daten und Transaktionen werden weder vom Modell noch von Bayou gefordert. Damit wird eine große Flexibilität erreicht, allerdings können Transaktionen und Daten während des Reintegrationsprozesses durch die Konfliktauflösungsprozedur verändert werden. Für Schreiboperationen werden als zusätzliche Informationen Zeitstempel, die Abhängigkeitsmenge (*dependency set*) und die Konfliktauflösungsprozedur (*mergeproc*) benötigt. Abbildung 2 zeigt eine Schreiboperation in Bayou am Beispiel eines Gruppenterminkalenders [EMP+97]. Weiterhin werden *read-set* und *write-set* einer Transaktion benötigt, wenn die von Bayou angebotenen Sitzungsgaran-

tien angefordert werden. Diese werden später im Text noch genauer erläutert. Zeitstempel werden verwendet, um die richtige Reihenfolge der an verschiedenen Servern eingebrachten Änderungen zu ermitteln. Dabei wird jeder Schreiboperation ein Zeitstempel durch den Server zugewiesen, der als erster die betreffende Änderung vom mobilen Client erhalten hat. Die Abhängigkeitsmenge (*dependency set*) besteht aus einer Sammlung von anwendungsgegebenen Anfragen und deren erwartetem Ergebnis und stellt die Implementierung des Akzeptanzkriteriums zur Konflikterkennung dar. Ein Konflikt ist dann erkannt, wenn die gegen einen Server mit seiner aktuellen Kopie gestellten Anfragen nicht die erwarteten Ergebnisse liefern. Dies ist am Beispiel des Gruppenkalenders in Abbildung 2 dann der Fall, wenn eine Terminüberlappung vorliegt und somit nicht die erwartete leere Überlappungsmenge bei Anfrage der Abhängigkeitsmenge erzeugt wird.

Wie auch schon die Abhängigkeitsmenge verwendet die Konfliktlösungsprozedur (*mergeproc*) die Semantik von Transaktionen bzw. Anwendungen. Wird ein Konflikt erkannt, d.h. der Abhängigkeitstest schlägt fehl, so erlaubt Bayou die Angabe einer automatischen Konfliktauflösungsprozedur. Diese arbeiten unter der Voraussetzung, daß es eine nicht genauer spezifizierte Anzahl von Anwendungen gibt, für die die Reihenfolge konkurrierender Schreiboperationen kein Problem ist oder die den Konflikt angemessen lösen können. Jede Schreiboperation in Bayou enthält diese anwendungsspezifische Konfliktauflösungsprozedur (*mergeproc*), die bei einem Schreibkonflikt automatisch auf dem Server aufgerufen wird und diesen Konflikt löst, indem sie eine alternative Menge von Änderungsoperationen passend zum aktuellen Datenbankinhalt erzeugt. Zusätzlich muß für jedes Datenelement ein Versionsvektor mitgeführt werden, um den gezielten Änderungsaustausch (*anti-entropy*) zwischen zwei Servern zu ermöglichen. Der Aufwand für Versionsvektor und Zeitstempel fällt sowohl in Berechnung als auch Speicherung gering aus. Die Konfliktauflösungsprozedur und die Abhängigkeitsmenge sind anwendungsspezifisch und müssen pro Anwendung nur einmal entwickelt werden, damit sind allerdings keine ad-hoc Anfragen möglich. Einzig die Speicherung der konkreten Ausprägungen der Konfliktauflösungsprozedur und Abhängigkeitsmenge pro Änderungsoperation erzeugt einen gewissen Mehraufwand wie in [DPS+95] beschrieben. Schließlich erfordert das Aufzeichnen des *read-set* das Protokollieren von Leseoperationen, was zu Lasten der Performance geht.

Aufgrund der Möglichkeit des lokalen Abschlusses von Transaktionen im unverbundenen Zustand ist im Falle eines nichtauflösbaren Konfliktes beim Wiedereinbringen die Dauerhaftigkeit dieser Änderungen nicht gegeben. Zu einer möglichen Einschränkung kann es auch bei der Isolation kommen, wenn Client-Transaktionen auf Daten aufbauen, die durch spätere Konfliktlösung verändert werden. In diesem Zusammenhang ist auch die Atomarität nicht gesichert, falls bei einem Konflikt die Auflösung nur die Ausführung eines Teils der Transaktion vorsieht. Da das *read-any/write-any*-Prinzip verwendet wird, ist die Konsistenz der gesamten Datenbank gefährdet, dies betrifft aber nicht die Konsistenzeigenschaft von ACID. Beispielsweise könnte eine Schreiboperation die gewünschte Änderung auf einem Sekundärserver erzeugen und auf einem anderen als Konflikt erkannt werden und dadurch ein unterschiedliches Ergebnis der ausgeführten Konfliktauflösung (*mergeproc*) erzeugen. Um dennoch die Konsistenz der Datenbankkopien untereinander nach einer gewissen Zeit zu erreichen, tauschen die Server ihre Änderungen mittels *peer-to-peer*-Protokoll in der sogenannten *anti-entropy*-Session aus. Hiermit wird abgesichert, daß alle Kopien einer Datenbank auf einen gemeinsamen Zustand zustreben [Gol92]. Um dies zu erreichen, müssen die Server nicht nur alle Änderungen erhalten, sondern sie auch in

eine konsistente Reihenfolge bringen. Die richtige Reihenfolge wird mit Hilfe der erwähnten Zeitstempel ermittelt.

Um aktuelle Änderungen eines Servers weiterzuleiten, wählt dieser periodisch einen anderen Server aus. Die Auswahl hängt dabei von der Verfügbarkeit des Servers, den Kosten und dem Nutzen, d.h. der zu erwartenden Anzahl neuer Änderungen, eines Verbindungsaufbaus zu einem anderen Server ab. Nachdem der Austausch der Änderungen vollzogen ist, haben beide Server identische Kopien der Datenbank, d.h. es wurden die gleichen Schreiboperationen tatsächlich in der gleichen Reihenfolge ausgeführt. Hierbei sichert die in [PST+97] näher erläuterte Präfixeigenschaft (*prefix property*) ab, daß ein Server R mit einer Änderung W_i, die ursprünglich von einem anderen Server S als erstem entgegengenommen wurde, auch alle von Server S empfangenen Änderungen vor W_i erhält. Allerdings können neueintreffende Änderungen diese Ordnung wieder verändern. Um eine möglichst schnelle Stabilität und Dauerhaftigkeit der Daten zu erreichen, wird ein neuer Zustand für Änderungen eingeführt (COMMITTED). Sobald eine Änderung mit COMMITTED markiert wird, darf keine andere vorläufige Änderung davor eingereiht werden. Dies geschieht auf einem ausgezeichneten Server, dem sogenannten Primärserver (*primary server*), wobei es verschiedene Masterserver für verschiedene Datenelemente geben kann, wie eingangs erwähnt. Alle anderen Sekundärserver (*secondary server*) akzeptieren vorläufig die einzubringenden Änderungen und geben diese an den Primärserver mittels *anti-entropy* weiter.

Weiterhin können folgende Garantien (*session guarantees*) für eine Sitzung angefordert werden, die in [DPS+94a] genauer beschrieben werden. Dabei wird hier unter einer Sitzung eine Folge von Lese- und Schreiboperationen auf einem Datenbestand während der Ausführung einer Anwendung verstanden. Somit ist die Anwendung für eine sinnvolle Transaktionsklammerung verantwortlich.

Read Your Writes : Eine Leseoperation spiegelt die Auswirkungen von vorangegangenen Schreiboperationen innerhalb einer Sitzung wider, d.h. Leseoperationen sind auf Kopien der Datenbank (Server) beschränkt, die alle vorangegangenen Änderungen der betroffenen Sitzung enthalten.

Monotonic Read : Der Nutzer sieht eine Datenbank, deren Aktualität sich nicht verringert, d.h. Leseoperationen werden nur auf den Servern ausgeführt, die alle Änderungen enthalten, deren Auswirkungen schon von vorangegangenen Leseoperationen innerhalb der Sitzung gesehen wurden.

Writes Follow Reads : Traditionelle Lese-Schreib-Abhängigkeiten werden in der Umordnung aller Schreiboperationen auf den Servern erhalten, d.h. Änderungen einer Sitzung werden nach den Schreiboperationen geordnet, deren Auswirkungen schon von vorangegangenen Leseoperationen der betroffenen Sitzung gesehen wurden. Diese Garantie unterscheidet sich von den vorangegangenen beiden darin, daß sie auch Nutzer außerhalb der Sitzung betrifft, die dann die gleiche Reihenfolge der Änderungen sehen, auch wenn sie keine Sitzungsgarantien angefordert haben.

Monotonic Writes : Änderungen müssen vorangegangenen Änderungen einer Sitzung folgen, d.h. eine Schreiboperation wird in eine Datenbankkopie auf einem Server aufgenommen, wenn diese alle vorangegangenen Schreiboperationen der Sitzung enthält.

Allerdings hat das Anfordern von gewissen Garantien eine potentielle Verringerung der Verfügbarkeit zur Folge, da die Menge der Server, die die angeforderte Garantie erfüllen,

möglicherweise kleiner ist als die Gesamtmenge. Weiterhin ist nicht zu erkennen, wie eine Kennzeichnung der Server, welche eine bestimmte Garantie erfüllen können, stattfindet. Problematisch ist hierbei auch die Tatsache, daß unter Bayou keine explizite Trennung zwischen Client und Server stattfindet und somit keine Verwaltungshierarchie entsteht. Das Konzept der Sitzungsgarantien steht in Analogie zu der Konsistenzsicherung über die in Abschnitt 2.5.3 einleitend angesprochenen Isolationslevel, eine Zuordnung von Sitzungsgarantien und Isolationslevel wird allerdings nicht vorgenommen.

4.2.4 Implementierung

Bayou stellt eine Implementierung des *Two-Tier*-Modells dar und wurde als Projekt von 1993 bis 1997 am Xerox PARC entwickelt [DPS+94b]. Eine Realisierung der Server und Clients wurde für SunOS und Linux basierend auf POSIX sowie basierend auf Java für kommerzielle Datenbanken mittels JDBC vollzogen. Es stellt eine Plattform für replizierte, hochverfügbare und variabel konsistente Datenbanken dar. Anwendungsbereiche für mobile Nutzer finden sich in der Verwaltung von Terminkalendern, Bibliothekseinträgen, Newsmitteilungen und gemeinsamer Dokumententwicklung. Eine wirkliche Verwendung außerhalb des Forschungsparks von Xerox ist nicht bekannt.

4.3 Multiversionsmodell

4.3.1 Architektur

Das in [PB00] vorgestellte Modell baut auf einer erweiterten Client/Server-Architektur [Gol00] auf und betrachtet verstärkt die Reintegrationskomponente mit Konflikterkennung und Konfliktauflösung. Es wird auch hier ein völlig unverbundenes Arbeiten des Clients durch den lokalen Abschluß unterstützt, daher sind keine besonderen Voraussetzungen an die Hardware des Clients nötig. Unverbundene Änderungen des Clients auf replizierten Daten, den sogenannten *Snapshots*, werden lokal abgeschlossen (*local COMMIT*). Zum Wiedereinbringen muß der Client eine Verbindung zum Server aufnehmen, es läuft dann der im folgenden näher erläuterte Abgleichalgorithmus ab und im Erfolgsfall werden lokale Änderungen auf den Primärdaten des Servers global abgeschlossen (*global COMMIT*). Damit implementiert dieses Modell auch ein zweistufiges Replikationsschema. Als Netztopologie kann jedes beliebige Szenario zum Einsatz kommen, solange eine Unterstützung des Client/Server-Betriebs gesichert ist. Eine Middleware wird nicht vorausgesetzt bzw. nicht betrachtet. An das DBMS des Servers werden keine konkreten Voraussetzungen formuliert, ein zentralisierter Ansatz scheint hier jedoch am besten geeignet. Als Eigenschaft des Multiversionsmodells können zu einem Datenelement verschiedene Ausprägungen (Varianten) gespeichert sein. Im Abschnitt 4.3.3 wird auf die genaue Definition eingegangen. Schließlich ist das DBMS auf dem Server vollständig zur Konflikterkennung und -auflösung zuständig, da nur auf dem Server alle Kombinationen von Datenversionen (Snapshots) vorliegen, die zur Konfliktlösung herangezogen werden.

4.3.2 Nutzer

Aufgrund der Möglichkeit des Transaktionsabschlusses bei Unverbundenheit zum Server, kann es beim Wiedereinbringen trotz größtmöglicher automatischer Konfliktlösung zu einer

nicht abgleichbaren Änderungen des Clients kommen. Die grundlegende Idee des Multiversionsmodells ist nun, daß Änderungsaktionen Datenelemente nicht überschreiben, sondern jeweils neue Versionen erzeugen. Jedoch ist zumindest für die Entwicklung einer effektiven Konfliktauflösung ein manuelles Eingreifen erforderlich. Damit kann auch hier der mobile Aspekt nicht vollständig verborgen werden. Allerdings kann durch die Verwendung von Versionen eine deutlich höhere Anzahl von Client-Transaktionen reintegriert werden, da Lese-Aktionen einen erhöhten Freiheitsgrad beim Suchen einer konsistenten Version haben. Damit wird effektiv ein größerer Nutzen für den mobilen Nutzer erzeugt, weil sich dieser mit größerer Wahrscheinlichkeit auf die Dauerhaftigkeit seiner lokalen Änderungen verlassen kann. Die Bewegungsfreiheit des Nutzer dürfte aufgrund der Möglichkeit des unverbundenen Arbeitens kaum eingeschränkt sein.

4.3.3 Datenhaltung und -zugriff

Das oben eingeführte Multiversionsmodell wird wie folgt formalisiert. Der Datenbankzustand auf dem Server besteht aus:

- einer Menge von Elementen $D \subset X \times \mathbb{Z}^+$ (endliche Menge von Versionsnummern von Datenelementen aus X),
- einer *Werte*-Funktion, die von der Version eines Datenelementes auf den aktuellen Wert des Elementes abbildet:

$$value : D \to \bigcup_{x \in X} domain(x)$$

Es gibt zwei Arten von Snapshots, *Version*-Snapshots und *Werte*-Snapshots, wobei ein Snapshot hier einen Datenbankzustand zu einem bestimmten Zeitpunkt definiert, d.h. jedem Element wird durch Angabe einer Version, hier eines ganzheitlichen Zeitstempels, eindeutig ein Wert zugeordnet. Version-Snapshots bestehen aus Datenelementen und ihren Versionen, sowie indirekt aus ihren Werten über die *Werte*-Funktion. Die Version eines Datenelements ist der COMMIT-Zeitstempel der Transaktion, die diese Version geschrieben hat. Die *Version*-Snapshotfunktion $S : \mathbb{Z}^+ \to D$ liefert zu jedem Zeitstempel eine Menge von Objektversionen mit folgenden Eigenschaften:

1. Jeder Snapshot ist eine Teilmenge des Datenbankzustands.
2. Für jeden Snapshot $S(v)$ gilt : wurde das Objekt x durch eine Transaktion mit Zeitstempel v geschrieben, dann ist die Version v dieses Objektes in $S(v)$ enthalten.
3. Kein Objekt in $S(v)$ kann eine größere Versionsnummer haben als v.
4. Für jedes Objekt x kann nur eine Version in $S(v)$ enthalten sein.
5. Ein Snapshot $S(v)$ enthält von den Objekten die letzten, d.h. aktuellesten Versionen, die kleiner oder gleich v sind.

Anders ausgedrückt ist $S(v)$ der Snapshot einer Datenbank, den eine nur-lesende Transaktion mit Startzeitstempel v sehen würde. Entsprechend zu jedem Versionsnapshot $S(v)$ gibt es einen *Werte*-Snapshot $S^v(v)$, der aus Datenelementen mir ihren Werte (statt Versionen)

Datenbankzustand :	$D = \{x_0, x_1, x_2, y_2, z_0, z_1\}$
Werte-Funktion :	$value = \{[x_0, 1], [x_1, 2], [x_2, 3], [y_2, 10], [z_0, 2], [z_1, 3]\}$
Zeitstempel 1 :	$S(1) = \{x_1, z_1\}$
	$S^v(1) = \{[x, 2], [z, 3]\}$
Zeitstempel 2 :	$S(2) = \{x_2, y_2, z_1\}$
	$S^v(2) = \{[x, 3], [y, 10], [z, 3]\}$

Abbildung 3: Beispiel für einen Datenbankzustand und entsprechende Snapshots

besteht und wie folgt definiert ist. Zur besseren Unterscheidung wird der Werte-Snapshot mit hochgestelltem v (für *value*) markiert.

$$S^v(v) = \{[x, i] \quad | \quad [x, v'] \in S(v) \wedge i = value(x, v')\}$$

Ein Beispiel für Version- und Werte-Snapshots zu verschiedenen Zeitstempeln ist in Abbildung 3 zu sehen. Ein Snapshot enthält dabei nicht immer alle Datenelemente der Datenbank, da nicht zu jedem Datenelement entsprechende Versionen vorliegen. Dies ist beispielsweise in Abbildung 3 für das Element y und Zeitstempel 1 der Fall, da y erst zum Zeitpunkt mit Zeitstempel 2 existiert. Die mit Multiversionsmodellen gut arbeitende Serialisierungsgarantie ist SNAPSHOT ISOLATION [BBG+95], welche nur solche Abfolgen von Transaktionen erlaubt, in denen Transaktionen von einem Snapshot der Datenbank lesen und konkurrierende Transaktionen verschiedene Datenelemente schreiben, d.h. falls ein Element von gleichzeitig zwei Transaktionen verwendet wird, darf nur eine Transaktion dieses Element ändern. Unter SERIALIZABLE dürfte dagegen überhaupt nur eine Transaktion Änderungen durchführen, d.h. andere Transaktionen dürften nur lesen. Damit können potentiell mehr Ausführungsfolgen unter Verwendung des SNAPSHOT ISOLATION-Levels zugelassen werden. Dies wird unter anderem auch dadurch erreicht, daß eine Transaktion auf verschiedenen, eventuell älteren, Snapshots der Datenbank abgeglichen werden kann. Dies sei an folgendem Beispiel gezeigt, nachdem x_0 und y_0 jeweils mit dem Wert 1 initialisiert wurden:

Client : repliziere $x = 1, y = 1$; $r_{1'}[x] = 1, r_{1'}[y] = 1, w_{1'}[y] = 3, c_{1'}$, abgleichen von $T_{1'}$
Server : $r_1[x_0] = 1, r_1[y_0] = 1, w_1[x_1] = 2, c_1$

Zum Zeitpunkt der Reintegration der Änderungen des mobilen Clients liegen zwei unterschiedliche Snapshots in der Datenbank vor, $\{x_0 = 1, y_0 = 1\}$ und $\{x_1 = 2, y_0 = 1\}$. Der erste ist konsistent zum *read-set* von $T_{1'}$, damit kann $T_{1'}$ auf diesem Snapshot serialisiert werden und man erhält folgenden Ablauf auf dem Server:

Server : $r_{1'}[x_0] = 1, r_{1'}[y_0] = 1, r_1[x_0] = 1, w_{1'}[y'_1] = 3, c_{1'}, w_1[x_1] = 2, c_1$

Diese Auflösung funktioniert aber nur deshalb, weil Transaktion T_1 auf dem Server nicht schreibend auf das Objekt y zugreift und damit jede Transaktion disjunkt einen Teil eines Snapshots ändert, sodaß alle Änderungen schließlich wieder zu einem Zustand zusammengefügt werden können. Ist diese Eigenschaft für zwei Transaktionen nicht gegeben, schreiben beide neue Snapshots. Weitere Eigenschaften an Daten oder Transaktionen werden nicht gefordert. Zur Umsetzung der SNAPSHOT ISOLATION werden folgende zusätzliche Informationen benötigt:

- die Menge der gelesenen (*read-set*) und geschriebenen Daten (*write-set*), d.h. zu einer Version v sind für eine Transaktion T folgende Mengen definiert:
  - $RSET^v(T)$ enthält alle Objekte, die von T gelesen, aber nicht geschrieben wurden
  - $RWSET^v(T)$ enthält alle Objekte, die von T sowohl gelesen als auch geschrieben wurden
  - $WSET^v(T)$ enthält alle Objekte, die von T geschrieben wurden, bevor sie gelesen wurden (*blind write*)
  - $READSET^v(T) = RSET^v(T) \cup RWSET^v(T)$
  - $WRITESET^v(T) = RWET^v(T) \cup WSET^v(T)$
- Zeitstempel zur Versionsverwaltung
- optional eine vom Benutzer angepaßte Konfliktauflösungsprozedur

Allerdings führt gerade die Aufzeichnung von Leseoperationen zu erhöhter I/O-Belastung und damit zu verminderter Performance und möglicherweise erhöhtem Speicherbedarf. Der für die Versionsverwaltung benötigte global eindeutige Zeitstempel ist aufgrund nie vollkommen synchroner Uhren nicht realisierbar, worauf das Modell aber nicht weiter eingeht und dadurch die Frage der Praxisrelevanz ungeklärt läßt. Bei den ACID-Eigenschaften werden vom Modell Atomarität und Konsistenz aufrechterhalten. Durch die Möglichkeit des lokalen Abschlusses ist auch eine potentielle Abschwächung der Dauerhaftigkeit unvermeidbar. Die Isolationseigenschaft ist nicht vollständig gewährleistet, da unter einer Abschwächung des Isolationslevels SNAPSHOT ISOLATION gearbeitet wird, der sich zwischen SERIALIZABLE und READ COMMITTED einordnet [BBG+95] und sowohl hohe Performance als auch einen hohen Konsistenzgrad erzeugt. Die abgeschwächte Form der SNAPSHOT ISOLATION kommt zustande, da die Anomalien A5B (Write Skew) und P3 (Phantom) nicht vermeidbar sind (vgl. Abschnitt 2.5.3). Für Anomalie P3 soll dies an folgendem Szenario gezeigt werden.

Eine Transaktion T_c auf dem Client liest Daten, die ein Prädikat P erfüllen. Dies seien die Elemente x, y und z. Damit ist $READSET(T_c) = \{x, y, z\}$. Sei nur die Standard-Konfliktlösungsprozedur aus Abbildung 4 gegeben. Der Abgleich-Algorithmus in Abbildung 5 bringt als Ergebnis eine Serialisierung der Transaktion T_c gegen einen Snapshot mit identischen Werten von x, y und z, aber mit einem weiteren Element a, welches ebenso das Prädikat P erfüllt. Dieses Element a würde dann niemals im *read-set* von T_c erscheinen, wodurch Phänomen P3 erlaubt wird.

Konflikte werden automatisch erkannt, indem geprüft wird, ob die von der lokalen Transaktion gelesenen Werte (*read-set*) eines Snaphsots mit den aktuellen auf dem Server vorliegenden Werten übereinstimmen oder nicht. Im letzten Fall besteht ein Konflikt und es muß ein anderer Snapshot mit Hilfe des in Abbildung 5 dargestellten Abgleich-Algorithmus gesucht werden, auf den die Änderungen angewendet werden können. Die Grundidee ist, für jede Client-Transaktion den Abgleich-Algorithmus zu durchlaufen und einen Snapshot zu suchen, der konsistent mit dem *read-set* der Client-Transaktion ist und die geringsten Arbeitskosten erzeugt, die mit Hilfe der Kostenfunktion C_{T_c} aus Abbildung 4 berechnet werden können. Für jede Version der Menge V, die aus allen zum Zeitpunkt des Client-Abgleichs in der Datenbank verwendeten Versionen besteht, werden in aufsteigender Folge

$$C_{T_c}(READSET^v(T_c), WRITESET^v(T_c), S^v_{in}) = \begin{cases} 0 & \text{falls } READSET^v(T_c) \subseteq S^v_{in} \\ \infty & \text{sonst} \end{cases}$$

$$CR_{T_c}(READSET^v(T_c), WRITESET^v(T_c), S^v_{in}) = \begin{cases} WRITESET^v(T_c) & \text{falls } READSET^v(T_c) \subseteq S^v_{in} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Abbildung 4: Standardkosten- und Standardkonfliktauflösungsfunktion

entsprechende Snapshots über Werte- und Version-Snapshotfunktion berechnt. Ein Konflikt ist gelöst, wenn die neu zu schreibenden Daten (*write-set*) mit keinem schon vorhandenen Snapshot kollidieren. Dies wird dadurch erreicht, daß alle Schreiboperationen im neuen *write-set* immer serialisierbar entweder nach der letzten Version eines Objektes oder davor als *blind write* auf dem Objekt sind. Die im Abgleich-Algorithmus verwendeten zwei Funktionen $S^{\downarrow}(k)$ und $S^{\uparrow}(k)$ sind folgendermaßen definiert:

normale Snapshotfunktion :

$$S^{\downarrow}(k) = \{[x, v'] \quad | \quad [x, v'] \in D \wedge v' = \max\{v'' | [x, v''] \in D \wedge v'' \leq k\}\}$$

umgekehrte Snapshotfunktion :

$$S^{\uparrow}(k) = \{[x, v'] \quad | \quad [x, v'] \in D \wedge v' = \min\{v'' | [x, v''] \in D \wedge v'' \geq k\}\}$$

$S^{\uparrow}(k)$ kann dabei als zeitliche Umkehrung von $S^{\downarrow}(k)$ angesehen werden. Da hier nichtnegative vollständige Zeitstempel als Versionen verwendet werden, sind $S^{\downarrow}(k)$ und $S(k)$ außerdem identisch. Schließlich ist V die Versionenmenge, die aus allen in der Datenbank verwendeten Versionen zum Zeitpunkt des Abgleichs besteht. Der Algorithmus verwendet die in Abbildung 4 dargestellten Standardfunktionen zur Konfliktauflösung (CR_{T_c}) und Kostenfunktionen (C_{T_c}), die allerdings vom Benutzer angepaßt werden können. Die vorgestellten Standardfunktionen erlauben eine Serialisierung einer Transaktion genau dann, wenn das von der Transaktion gesehene *read-set* des gelesenen Snapshots und der Snapshot S^v_{in}, gegen den serialisiert werden soll, identisch sind. Die Standard-Konfliktauflösungsfunktion stellt effektiv eine Wiederausführung der Client-Transaktion T_c auf dem neuen Snapshot S^v_{in} dar. Kaskadierendes Zurücksetzen von aufeinander aufbauenden Transaktionen bei Abbruch einer Clienttransaktionen ist nicht nötig wie in [PB00] gezeigt wird.

4.3.4 Implementierung

Einen im Einsatz befindlichen Prototypen gibt es für das bewertete Modell nicht, allerdings einige Beispielanwendungen und der aufgeführte, wenn auch mit sehr hohem Aufwand durchzuführende, Pseudocode in Abbildung 5 aus [PB00]. Aufgrund der Mächtigkeit und großen Kombinationsvielfalt des Multiversionsmodells besteht eine größere Chance, mehr konflikterzeugende Änderungen einzubringen. Andererseits erfordert die Verwaltung von Datenversionen und das Finden eines passenden Snapshots einen höheren Aufwand

INPUT: Datenbankzustand D, Transaktion T_c, Funktionen CR_{T_c}, C_{T_c}
INIT: $opt = -1, cost = \infty$, static $iter = 1$
FOR ALL $v \in V \cup \{1 + \max(V)\}$ in aufsteigender Ordnung **DO**
    $S^{\downarrow} = S^{\downarrow}(v - \frac{\triangle}{iter})$
    $S^{\uparrow} = S^{\uparrow}(v - \frac{\triangle}{iter})$
    // nun herausfinden, ob dieser Snapshot optimale Kosten hat und verwendbar ist
    IF $C_{T_c}(READSET^v(T_c), WRITESET^v(T_c), value(S^{\downarrow})) < cost$ **THEN**
        $NEWWRITESET^v = CR_{T_c}(READSET^v(T_c), WRITESET^v(T_c), value(S^{\downarrow}))$
        // $NEWWRITESET$ darf mit keinem existierenden Snapshot in D kollidieren
        FOR ALL $[x, i] \in NEWWRITESET^v$ **DO**
            // ist das Element aus $S^{\uparrow}$?
            // wenn ja, wurde es durch ein blind write erzeugt?
            IF $\exists v' : [x, v'] \in S^{\uparrow}$ **THEN**
                Sei T die Transaktion mit Zeitstempel v'
                IF $\exists i : [x, i] \in WSET^v(T)$ **THEN**
                    // $[x, v']$ ist das Ergebnis eines blind write
                    Setze innere **FOR**-Schleife fort
                ELSE
                    // dieser Snapshot kann nicht verwendet werden
                    Setze äußere **FOR**-Schleife fort
                END IF
            END IF
        END FOR
        $cost = C_{T_c}(READSET^v(T_c), WRITESET^v(T_c), value(S^{\downarrow}))$
        $opt = v$
    END IF
END FOR
IF $cost = \infty$ **THEN**
    // Änderungen der Transaktion T_c können nicht abgeglichen werden
    ABORT T_c
ELSE
    // Abgleich von T_c erfolgreich
    Serialisiere T_c mit Zeitstempel $opt - \frac{\triangle}{iter}$
    $iter = iter + 1$
END IF

Abbildung 5: Multiversions-Abgleichalgorithmus

und könnte zu Performance-Problemen führen. Am schwerwiegendsten ist allerdings die Aufzeichnung des *read-sets* zur Konflikterkennung und -auflösung. Das Protokollieren von Leseoperationen würde sich in enormen Performanceverlusten niederschlagen. Somit ist eine Realisierung auf diesem Wege wenig geeignet.

4.4 Semantische Transaktionen

4.4.1 Architektur

Das in [WC95] vorgestellte Modell soll vorallem eine Unterstützung von beabsichtigten oder unbeabsichtigten Phasen der Unverbundenheit unterstützen, obwohl es auch für Szenarien mit einer schwachen Verbindung geeignet ist. Es ist ein Ziel dieses Modells, den in der Einleitung erläuterten nötigen Speicher- und Kommunikationsbedarf für das unver-

bundene Arbeiten so gering wie möglich zu halten. Die dazu verwendeten Konzepte werden in den folgenden Abschnitten vorgestellt. Da auch unverbundenes Arbeiten vom Modell unterstützt wird, sind für Lese- und Schreiboperationen keine speziellen Verbindungsanforderungen nötig. Im Fall der Unverbundenheit wird autonome Transaktionsverarbeitung mit lokalem Abschluß auf dem Client gesichert, der zweite Fall der schwachen Verbundenheit wird vom Modell zwar unterstützt, aber nicht weiter betrachtet. Als Netztopologie wird das bereits erläuterte Zellsystem aus Abbildung 1 zugrundegelegt. Eine Middleware wird vom Modell nicht explizit betrachtet, allerdings wird sie von den Basisstationen gebildet, zu deren Aufgabe unter anderem die Aufrechterhaltung der Verbindung zum Client gehört. Auf weitere Aufgaben, vorallem im Bereich der Transaktionsverarbeitung, wird im Modell nicht weiter eingegangen. Die Daten werden auf einer nicht näher spezifizierten Anzahl von Servern im festverbundenen stationären Netz gespeichert, d.h. sowohl ein zentralisiertes als auch verteiltes DBMS kann als Grundlage dienen. Der Server ist für die alleinige Konflikterkennung und -auflösung zuständig. Zusätzlich muß er das Konzept der Fragmentierung von Objekten realisieren und Objektgranulate je nach Parameter des Clients bereitstellen können. Schließlich hat er auch die Aufgabe, die von Clients geänderten Fragmente eines Objektes wieder korrekt zusammenzusetzen.

4.4.2 Nutzer

Der unverbundene Fall wird durch lokales Cachen von Daten abgesichert, allerdings ist die Dauerhaftigkeit lokaler Änderungen nicht garantiert, wodurch sich für den Nutzer keine vollständige Transparenz bietet. Denkbar ist, wie bei anderen Modellen auch, eine größtenteils automatische Konfliktauflösung mit manuellem Eingriff als letzte Möglichkeit. Dagegen ist die Bewegungsfreiheit des Nutzers überhaupt nicht einschränkt, da sowohl im schwach verbundenen Zustand ein Zellwechsel als auch der vollständig unverbundene Zustand unterstützt werden. Durch zwei neue semantische Konzepte, *Fragmentierbarkeit* und *Umsortierbarkeit*, wird versucht, nur relevante Datenteile von Objekten auf dem Client zu replizieren und mehr konkurrierende Transaktionen einbringen zu können, indem Konfliktmöglichkeiten schon während der Replikation beseitigt werden.

4.4.3 Datenhaltung und -zugriff

Um einen hohen Grad an Nebenläufigkeit, große Datenverfügbarkeit und eine Vereinfachung der Recovery zu unterstützen, verwendet auch dieses Transaktionsmodell einiges semantisches Wissen. Dabei spielen folgende Merkmale eine Rolle:

1. *Semantik der Operationen*, die in Beziehung zu den Effekten einer Operation auf einem Zustand eines Objektes steht.

2. *Eingabe-/Ausgabewerte der Operation*, welche Bezug auf die Richtung des Informationsflußes zu und vom Objekt sowie der Interpretation dieser Werte nimmt.

3. *Aufbau eines Objektes*, welcher auf die abstrakte Organisation eines Objektes, im Gegensatz zu seiner physikalischen Implementierung, verweist.

4. *Nutzung eines Objektes*, welche beschreibt, wie ein Objekt genutzt wird und wie die extrahierten Informationen verwendet werden.

Da im vorliegende Modell die semantikbasierte Transaktionsverarbeitung im Vordergrund steht, werden entsprechend einige semantische Anforderungen an Daten und Operationen gestellt. Die verwendeten Daten (Objekte im allgemeineren Sinne) müssen fragmentierbar und wieder zusammensetzbar sein. Die grundlegende Idee dabei ist, große und komplexe Objekte in kleinere Fragmente mit gleicher Struktur zu zerlegen unter Ausnutzung des Objektaufbaus. Beispielsweise läßt sich eine Menge in disjunkte Teilmengen zerlegen, wobei allgemeine Eigenschaften sowohl für die Gesamtmenge als auch für die einzelnen Teilmengen gelten und somit die Struktur erhalten bleibt. Dadurch braucht ein mobiler Client nur die Objektfragmente replizieren, welche für bevorstehende Aufgaben auch benötigt werden und kann dadurch Speicheranforderungen optimieren. Die zweite Idee ist, diese Fragmente als Grundeinheit des Abgleichs von Änderungen, d.h. als Konsistenzeinheit, zu betrachten. Um mehr Flexibilität als auch eine Behandlung von Fällen zu ermöglichen, in denen Fragmentierung unter strikter Konsistenzerhaltung für ein Objekt nicht möglich ist, können Anwendungen explizit ihre zu erfüllenden Konsistenzbeschränkungen definieren, auf die jedoch im Modell nicht näher eingegangen wird und nur noch eine Fragmentierung von Objekten betrachtet wird, für die strikte Konsistenzerhaltung garantiert werden kann. *Fragmentierbare* Objekte sind Objekte, die um folgende zwei im Objekt gekapselten typspezifischen Operationen erweitert sind:

split : Durch diese Operation wird ein Objekt auf dem Server in Fragmente zerlegt, deren Granularität sich nach den folgenden anzugebenden Parametern richtet. Diese Fragmente kann der mobile Client dann replizieren.

- Über das *Auswahlkriterium* kann das zu replizierende Objekt und die vom Client benötigte Größe des Objektfragmentes spezifiziert werden. Wenn der angeforderte Objektteil auf dem mobilen Client zwischengespeichert ist, wird er logisch aus der Hauptversion (*master copy*) des Servers entfernt (*check-out*) und ist nur für Transaktionen des mobilen Clients zugänglich. Der übrige Teil des Objektes ist davon nicht betroffen und kann von Transaktionen konkurrierender Clients verwendet werden.
- Die *Konsistenzbedingungen* geben die Beschränkungen auf den Fragmenten an, welche zur Erhaltung der Konsistenz des gesamten Objekts erfüllt werden müssen. Darunter können sowohl erlaubte Operationen und Bedingungen an deren Eingabewerte als auch Bedingungen an den Zustand des Objektes fallen. Beispielsweise könnten einige Operationen verboten werden, um zu garantieren, daß die Fragmente eines Objektes wieder richtig zusammenfügt werden können. In anderen Fällen können diese Beschränkungen notwendig sein um sicherzustellen, daß einige Fragmente zusammengefügt werden können und gleichzeitig die Änderungen an Fragmenten anderer schon abgeschlossener Transaktionen erhalten bleiben.

Als Ergebnis werden entsprechende Fragmente der *master copy* des Objektes vom Server übertragen.

merge : Mit Hilfe dieser Operation werden nach der individuellen Bearbeitung der Fragmente diese wieder in das Originalobjekt *master copy* eingegliedert. Dabei ist zu beachten, daß Fragmente eine *logische* oder *physische* Zerlegung eines Objektes sein können. Fragmente der ersten Kategorie werden durch eine logische oder arithmetische Operation reintegriert, während Fragmente der zweiten Variante dementsprechend physisch zusammengesetzt werden müssen.

Zur Erhöhung der Verfügbarkeit, d.h. daß ein Fragment von mehreren Clients repliziert und bearbeitet werden kann, ist als weitere Eigenschaft die *Kommutativität* von Operationen nutzbar. Durch Verwendung kommutativer Operationen werden Abhängigkeiten von Änderungsoperationen der Clients untereinander vermieden. Operationen sind *kommutativ*, wenn ihre Auswirkungen auf den Zustand eines Objektes und ihre Rückgabewerte unabhängig von ihrer Ausführungsfolge die gleichen sind, d.h. sie können in einer beliebigen Ausführung angeordnet werden. Fragmentierbare Objekte, auf die mit kommutativen Operationen zugegriffen werden kann, heißen *umsortierbar*. Allerdings kann diese Eigenschaft nur von wenigen üblichen Operationen, wie z.B. der numerischen Addition, zugesichert werden. Einige Operationen sind auch nur bedingt kommutativ, wie Subtraktion und Addition im Zusammenwirken im Bereich der natürlichen Zahlen. Oft wird auch die Reihenfolge der Rekombination durch die Struktur des gesamten Objektes oder der Operationen auf den Fragmenten vorgegeben. Um unabhängiges Abschließen von Transaktionen verschiedener mobiler Clients unterstützen zu können, müssen dann die Auswirkungen der Operationen beim Zusammenfügen erhalten werden. Kann jedoch uneingeschränkte Kommutativität garantiert werden, so wird dadurch kaskadierendes ABORT vermieden, da die Recovery im Fehlerfall auf Transaktionsgrenzen beschränkt wird. Sind alle Operationen auf einem Objekt in jedem Zustand kommutativ, dann kann dieses Objekt ohne Koordination mit dem Server auf mobilen Clients repliziert und manipuliert werden, wobei lokal abgeschlossene Änderungen in Abständen zum Server weitergegegeben werden müssen.

Neben den vom Client zu spezifizierenden Parametern zur Granularität und Fragmentierung des gewünschten zu replizierenden Objektes, werden vorallem Zusatzinformationen in Bezug auf die Möglichkeit der Fragmentierbarkeit und Umsortierbarkeit sowie der Semantik der verwendeten Daten bzw. Transaktionen benötigt. Im Bereich der ACID-Eigenschaften kann es aufgrund der Fragmentierung nicht zu konkurrierendem Zugriff mehrerer Transaktionen auf ein Objektfragment kommen, da auf Fragmenten gesperrt wird. Wird diese pessimistische Replikation aufgehoben und der konkurrierende ändernde Zugriff auf Fragmente erlaubt, müssen diese Änderungen umsortierbar (kommutativ) sein. Somit ist die Dauerhaftigkeit in beiden Fällen gesichert. Alle weiteren drei Eigenschaften sind ebenfalls nicht in ihrer Gültigkeit betroffen. Allerdings gibt es hierbei Einschränkungen für nichtfragmentierbare Objekte über anwendungsspezifische Konsistenzbedingungen. Das Konzept der Korrektheitsforderungen von *strikter* Konsistenz bis zu *letztendlicher* Konsistenz steht auch hier in Analogie zu den in Abschnitt 2.5.3 aufgeführten Isolationsleveln, wobei strikte Konsistenzbedingungen an ein Fragment zu Serialisierbarkeit führen und letztendliche Konsistenz kurzzeitige Abweichungen erlaubt. Auf Verfahren zur Konflikterkennung und Konfliktauflösung wird nicht näher eingegangen, da es bei strikter Umsetzung des vorgestellten Konzeptes nicht zu Konflikten kommen kann.

4.4.4 Implementierung

Eine Realisierung dieses Modells liegt zwar nicht vor, allerdings hat das vorgestellte Konzept interessante Aspekte zu bieten. Die Verwendung der semantischen Eigenschaft Fragmentierbarkeit ist zwar eher auf in realen DBMS selten verwendete Datentypen wie Aggregate, Mengen, Stacks und Warteschlangen beschränkt, bietet dafür aber eine Möglichkeit, Konflikte zwischen gleichzeitig auf einem Objekt arbeitenden Clients von vornherein zu vermeiden. Weiterhin trifft für sehr wenige Operationen die Eigenschaft der Umsortierbarkeit zu. Für das Modell spricht auch, daß bis auf die semantischen Forderungen an Daten

und Operationen keine weiteren möglicherweise aufwendig zu beschaffenden Zusatzinformationen benötigt werden.

4.5 Schwache/strikte Transaktionen

4.5.1 Architektur

Das Modell von [PB95] stellt keine speziellen Voraussetzungen an die Ausstattung der Clients, denen das unverbundene Arbeiten ermöglicht wird. Daten sind im Sinne einer Verteilten Datenbank auf feste Knoten und mobile Clients verteilt. In Transaktionen können sowohl auf anderen Knoten liegende als auch lokal auf dem Client gehaltene Daten involviert sein. Die Elemente einer Datenbank sind in sogenannten *Clustern* zusammengefaßt. Cluster sind hier die Grundeinheiten der Konsistenz unter der Forderung, daß alle Daten innerhalb eines Clusters untereinander strikt konsistent sind. Dies bedeutet, daß alle Nebenbedingungen innerhalb eines Clusters (*intra-cluster constraints*) gelten. Dagegen können zwischen Clustern gewisse Inkonsistenzen auftreten und somit nicht alle Nebenbedingungen zwischen Daten verschiedener Cluster (*inter-cluster contraints*) erfüllt sein. Der Grad der Konsistenz kann dabei von der verfügbaren Netzverbindung zwischen den Clustern abhängen. Ein Cluster könnte beispielsweise die Daten aller untereinander verbundenen Clients enthalten. Diese Clusterbildung ist dynamisch und paßt sich somit der wechselnden Verbundenheit von Clients an.

Je nach Verbindungsart werden unterschiedliche Transaktionsarten (*strikte* und *schwache* Transaktionen) angeboten, die Ergebnisse unter entsprechenden Konsistenzgraden liefern und erzeugen. Auf deren Eigenschaften wird im folgenden näher eingegangen. Eine bestimmte Netztopologie wird nicht zugrundegelegt, damit läßt sich dieses Modell sowohl in Zellsystemen als auch festen Netzwerkstrukturen implementieren. Aufgrund der Modellannahme eines verteilten DBMS, ist keine Unterscheidung zwischen Servern und Clients möglich, abgesehen von einer erhöhten Unverbundenheit der mobilen Clients. Somit liegt die Verantwortung über die Verbreitung von Änderungen, Konflikterkennung und -auflösung sowohl auf Seiten der Server als auch der Clients.

4.5.2 Nutzer

Das Ziel des Modells ist hier vorallem eine bessere Konsistenzsicherung der Daten, die mobile Nutzer über lokale Transaktionen ändern. Der mobile Aspekt wird dabei durch schwache Transaktionen im unverbundenen Zustand zum größten Teil verborgen, nur gewisse Einschränkungen in der zur Verfügung stehenden Auswahl von Anwendungen müssen vom Nutzer toleriert werden

4.5.3 Datenhaltung und -zugriff

Abgesehen vom Aspekt der Einteilung von Daten in Cluster, sind keinerlei Anforderungen von den Daten zu erfüllen. Um lokale Bearbeitung zu unterstützen und Netzwerkzugriffe zu minimieren, wird dem Nutzer erlaubt, mit lokal verfügbaren *m-konsistenten* Daten innerhalb eines Clusters Cl_i zu agieren. Ein Datenbankzustand ist hierbei m-konsistent genau dann, wenn alle Clusterzustände konsistent und alle Integritätsbedingungen zwischen den Clustern (*inter-cluster constraints*) $m - gradig$ konsistent sind. Die m-gradige

Konsistenz einer Replikationsbedingung R auf einer Menge von Replikaten $\{x_{i_1}, \ldots, x_{i_k}\}$ eines Datenelementes x_i ist dabei wie folgt definiert:

$$\begin{aligned} R(x_{i_1}, \ldots, x_{i_k}) \quad = \quad & \exists j, l \in [1 \ldots k] \wedge (x_{i_j} \in Cl_m, x_{i_l} \in Cl_n) : \\ & (m = n \Rightarrow x_{i_j} = x_{i_l}) \wedge \\ & (m \neq n \Rightarrow m\text{-Grad}(x_{i_j}, x_{i_l})) \end{aligned}$$

Befinden sich also Kopien eines Datenelementes im gleichen Cluster (mehrere Knoten können einen Cluster bilden und jeweils einen Ausschnitt der Datenbank replizieren), besitzen sie den gleichen Wert. Werden Kopien eines Datenelementes in verschiedenen Clustern betrachtet, so wird deren Wert durch die passend zu definierende m-Grad-Relation bestimmt. Für replizierte Daten kann diese Relation die Werte-Abweichung zwischen den einzelnen Kopien ausdrücken. In diesem Fall könnte der Grad der Abweichung durch folgende Kriterien begrenzt werden:

- maximale Abweichung einer Kopie vom Original
- Anzahl der Transaktionen, die auf inkonsistenten Daten operieren
- Anzahl der erlaubten Versionen
- Anzahl der voneinander abweichenden Datenelemente
- Anzahl der voneinander abweichenden Kopien je Datenelement

Allerdings muß der Mangel an strikter Konsistenz durch die Semantik der Anwendung toleriert werden. Dafür werden zwei neue Operationen eingeführt, das schwache Lesen (*weak read, wr*) und schwache Schreiben (*weak write, ww*). Die üblichen Operationen im verbundenen Zustand heißen striktes Lesen (*strict read, sr*) und striktes Schreiben (*strict write, sw*). Ein $wr[x]$ liest eine lokal verfügbare Kopie von x, d.h. den Wert, der von der letzten schwachen oder strikten Schreiboperation in diesem Cluster geschrieben wurde. Ein $ww[x]$ schreibt eine lokale Kopie und ist vorerst nicht dauerhaft. Eine $sr[x]$ liest den Wert von x, der von der letzten strikten Schreiboperation geändert wurde. Schließlich schreibt ein $sw[x]$ eine oder mehr Kopien von x, die in den verschiedenen Clustern existieren. Aufbauend auf den Operationstypen werden, wie schon erwähnt, Transaktionen des mobilen Clients in folgender Weise unterschieden:

- Transaktionen, die nur aus schwachen Lese- und Schreiboperationen bestehen, werden schwache Transaktionen (*weak transaction*) genannt. Diese greifen nur auf lokal, d.h. innerhalb eines Clusters, konsistente Daten zu. Schwache Transaktionen haben zwei COMMIT-Punkte, einen lokalen im zugehörigen Cluster und einen globalen, nachdem der Cluster mit dem restlichen (festverbundenen) Netzwerk nach der Verbindungsaufnahme zum Server zusammengeführt wurde. Änderungen von lokal abgeschlossenen schwachen Transaktionen sind nur für andere schwache Transaktionen des gleichen Clusters verfügbar. Erst bei globalem Abschluß werden diese Änderungen permanent und damit auch für strikte Transaktionen sichtbar.
- Transaktionen, die nur aus strikten Lese- und Schreiboperationen bestehen, werden strikte Transaktionen (*strict transaction*) genannt. Sie benötigen eine Verbindung zum Netzwerk, um Daten auf anderen Clustern lesen bzw. erzeugen zu können, d.h. sie greifen nicht auf lokal replizierte und möglicherweise inkonsistente Daten zu.

Da die Möglichkeit eines Konfliktes beim Einbringen lokal abgeschlossener schwacher Transaktionen besteht, fordert das Modell für deren Verwendung, daß entsprechende *Kompensationstransaktionen* zur Verfügung stehen, die die Wirkung von zugehörigen Transaktionen semantisch zurücksetzen. Gibt es keine solchen Kompensationstransaktionen und ist die Wahrscheinlichkeit für Konflikte sehr gering, dürfen dennoch schwache Transaktionen verwendet werden. Zum Beispiel können schwache Transaktionen private Daten und strikte Transaktionen gemeinsam verwendete Daten ändern.

Jede übermittelte Transaktion wird in eine Anzahl von schwachen und strikten Sub-Transaktionen entsprechend dem von der Anwendung geforderten Konsistenzgrad zerlegt. Zur Umsetzung dieses Konzeptes werden zwei Versionen, x_k^s und x_k^w, für jede Kopie eines Datenelements x vom lokalen Transaktionsmanager in Cluster Cl_k verwaltet. Die Version x_k^s wird nur von strikten Transaktion gelesen und geändert und heißt deswegen strikte Version (*strict version*). Dagegen wird x_k^w sowohl von strikten als auch schwachen Transaktionen geändert, aber nur von schwachen Transaktionen gelesen, und heißt schwache Version (*weak version*). Um eine Transaktion zu verarbeiten, übersetzt das DBMS die enthaltenen Operationen auf Datenelemente in Operationen auf replizierte Kopien dieser Datenelemente mit Hilfe einer *Übersetzungsfunktion h*. Wird diese beispielsweise so definiert, daß jede strikte Schreiboperation nur eine Kopie eines Datenelementes ändert, ist dies mit dem Master-Replikationsschema von Bayou aus Abschnitt 4.2 äquivalent und nur eine Version pro Datenelement wird benötigt. Allgemein wird ein $wr[x]$ in eine Leseoperation $r[x_k^w]$ übersetzt, ein $ww[x]$ in eine Schreiboperation $w[x_k^w]$ in einem Cluster Cl_k auf eine lokal verfügbaren Kopie x_k^w. Ein ABORT oder COMMIT einer schwachen Transaktion wird in die entsprechenden lokalen Varianten der Transaktionsabschlüsse übersetzt. Dagegen wird in einem Cluster Cl_k ein $sr[x]$ in eine Leseoperation $r[x_i^s]$ und ein $sw[x]$ in eine Folge von Schreiboperationen $w[x_{j_1}^s], \ldots, w[x_{j_k}^s]$ und $w[x_{j_1}^w], \ldots, w[x_{j_k}^w]$ für jede Kopie $x_{j_1}, \ldots, x_{j_k}$ von x, d.h. eine strikte Schreiboperation ändert alle Replikate eines Objektes. ABORT und COMMIT werden in die globalen Varianten übersetzt.

Eine *Projektion* eines Schedules S auf einen Cluster Cl_k ist der Ablauf, den man durch entfernen aller Operationen aus S enthält, die nicht auf den Cluster Cl_k zugreifen. Entsprechend erhält man eine Projektion auf strikte Transaktion durch entfernen aller schwachen Transaktionen eines Schedules S. Um für strikte Transaktionen die Tatsache der Replikation zu verbergen, d.h. daß unter um Umständen durch eine Änderung mehrere replizierte Objekte in verschiedenen Clustern quasi gleichzeitig geändert werden müssen, muß gefordert werden, daß strikte Transaktionen so operieren, als wäre nur eine Kopie für jedes Objekt im System vorhanden, d.h. sie müssen den Forderungen der 1-Kopien-Serialisierbarkeit (1SR) genügen.

Für das Wiedereinbringen von lokalen (schwachen) Änderungen und der damit verbundenen Übersetzung in Operationen auf replizierte Datenelemente werden einige Informationen zur Versionsverwaltung benötigt. Entsprechender Aufwand ist auch für die Bereitstellung von Kompensationstransaktionen zu verzeichnen. Eine Abschwächung der ACID-Eigenschaften ist in der Gefährdung der Dauerhaftigkeit von lokal abgeschlossenen Transaktionen zu finden. Die Konsistenzgarantien für lokale Daten eines Clusters bezüglich anderer Cluster können von strikt für strikte Transaktionen im verbundenen Zustand bis zu einem gewissen Grad für schwache Transaktionen abgeschwächt sein. Diese Abschwächung läßt sich durch die oben definierte m-gradige Relation ausdrücken. Zum Testen der Korrektheit, d.h. der Erkennung von Konflikten in Schedules eines Clusters, wird ein modifizierter graphentheoretischer Ansatz gewählt. Unter Verwendung der Korrektheitskriterien

ist dies ein rein syntaktischer Ansatz, der anwendungsunabhängig versucht, so viele schwache Schreiboperationen wie möglich zu akzeptieren, ohne die 1SR-Forderung von strikten Transaktionen zu verletzen. Um Konflikte von *inter-cluster-Schedules*, d.h. zwischen Transaktionsfolgen verschiedener Cluster, zu lösen, werden diejenigen Transaktionen zurückgerollt, deren schwache Schreiboperationen mit strikten Transaktionen kollidieren. Durch die Clusterbildung wird dabei erreicht, daß nur schwache Transaktionen des gleichen Clusters durch kaskadierendes ABORT betroffen sind, wenn eine schwache Transaktion zurückgerollt werden muß. Dieser Konsistenzansatz mit strikten und schwachen Transaktionen verfolgt ein *hybrides* Modell, da pessimistische (für strikte Transaktionen) und optimistische (für schwache Tansaktionen) Nebenläufigkeits-Kontrollmechanismen zum Einsatz kommen.

4.5.4 Implementierung

Für eine Implementierung fehlen diesem Modell zwar einige wichtige architekturiellen Beschreibungen und weitere Ansätze zur Konfliktauflösung, die keine Kompensationstransaktionen verwenden. Allerdings bietet es eine sehr fundierte Grundlage, die Konsistenz bei Replikation im mobilen Szenarium zu erhalten. Dadurch könnte es als Erweiterung in anderen Modellen wie dem Two-Tier-Modell im Abschnitt 4.2 Einsatz finden.

4.6 Isolation-Only Transactions

4.6.1 Architektur

Die im Modell von [LS94] vorausgesetzten Clients müssen eine vollständige Transaktionsverarbeitung auf lokal replizierten Daten leisten können, da auch hier der unverbundene Zustand unterstützt wird. Für das unverbundene Arbeiten stehen die sogenannten *secondclass* Transaktionen zur Verfügung, dagegen führen *firstclass* Transaktionen keinen Zugriff auf lokal replizierte Daten aus. Dieses Konzept folgt der Idee der zweistufigen Transaktionsverabeitung, wie es im Two-Tier-Modell im Abschnitt 4.2 vorgestellt wurde. Eine spezielle Netztoplogie und Middleware wird dabei nicht vorausgesetzt bzw. nicht näher erläutert. Zur Architektur des DBMS auf dem Server werden keine Aussagen gemacht, da allerdings dieses Modell im Rahmen von Coda (Abschnitt 6.4) seinen Einsatz findet [LS95], wird ein zentrales DBMS gegenüber anderen Architekturen in der Verwendung überwiegen. Die Aufgaben des Servers sind vorallem in der Konflikterkennung und -auflösung der lokal auf dem Client ausgeführten *secondclass*-Transaktionen zu finden, um gewisse Konsistenzgarantien zu gewährleisten. Der Client ist zudem für die Serialisierbarkeit von auf ihm lokal ausgeführten Transaktionen verantwortlich. Ziel des Modells sind hohe Effizienz und einfache Operationen, was hier dazu führt, daß nur die Isolationseigenschaft von ACID für Transaktionen zugesichert wird. Auf die Erläuterung dieser Problematik wird im Abschnitt 4.6.3 eingegangen.

4.6.2 Nutzer

Der Nutzer soll durch die Verwendung des Modells vor Inkonsistenzen gesichert werden, die durch das mobile Arbeiten im unverbundenen Zustand entstehen können. Die dafür definierten Kriterien verlangen allerdings vom Nutzer bzw. der Anwendung Kenntnisse

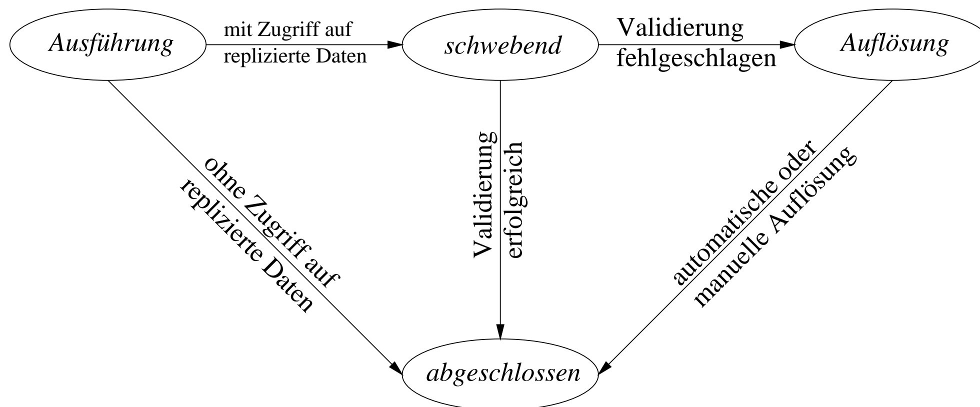


Abbildung 6: Zustandübergangsdiagramm für Isolation-Only Transactions

über den mobilen Aspekt und schwächen so das Bild einer unveränderten Arbeitsumgebung im getrennten Zustand. Einschränkungen in der Bewegungsfreiheit des Nutzer gibt es keine, wie auch bei anderen Modellen, die den unverbundenen Zustand unterstützen, werden hier keine Forderungen an die Synchronisationszeitpunkte gestellt.

4.6.3 Datenhaltung und -zugriff

Abhängig von der Verbindungsqualität werden *Isolation-only Transactions* (IOT) in zwei Klassen eingeteilt:

firstclass : Die Ausführung einer Transaktion in diesem Modus hat keinen Zugriff auf lokal replizierte Daten, d.h. der Client hält für jeden Datenzugriff eine Verbindung zum Server aufrecht. Damit kann die Transaktion auf dem Server mit COMMIT beendet werden und die Ergebnisse werden sofort für andere Transaktionen auf dem Server sichtbar.

secondclass : In diesem Fall enthält die Ausführung einer Transaktion einen Zugriff auf replizierte Daten, d.h. die Transaktion greift auf lokale Daten zu und braucht dafür keine Verbindung zum Server aufnehmen. Bei lokalem Abschluß der Transaktion kommt diese in den sogenannten *schwebenden* Zustand und wartet auf spätere Validierung. Die Ergebnisse dieser Transaktion sind nur für nachfolgende Transaktionen auf dem selben Client sichtbar. Die Validierung bei Wiederverbinden zum Server erfolgt entsprechend den Konsistenzkriterien, die im folgenden noch vorgestellt werden. Ist dieser Vorgang erfolgreich, wird die betreffende Transaktion sofort auf dem Server integriert und mit COMMIT abgeschlossen. Ansonsten verursacht die Transaktion einen Konflikt, wechselt in den *aufzulösenden* Zustand und kann manuell oder automatisch zum COMMIT geführt werden.

Das gesamte Zustandübergangsdiagramm für beide Transaktionsklassen ist in Abbildung 6 dargestellt. Ein Abbruch der Transaktion ist nicht explizit vorgesehen, könnte aber auch unter manueller Auflösung verstanden werden. Dabei werden vom Modell keine zusätzlichen Anforderungen an Daten oder Transaktionen gestellt. Über die Notwendigkeit von Zusatzinformation und deren Beschaffung werden vom Modell keine Aussagen gemacht,

jedoch werden zumindest für das Testen der globalen Serialisierbarkeit von unverbunden abgeschlossenen Transaktionen (*secondclass*) gewisse Informationen, wie zum Beispiel Zeitstempel, über die Ausführungsreihenfolge der einzelnen Transaktionen benötigt.

Eine Abschwächung der ACID-Eigenschaften findet zumindest im Konfliktfall für die Dauerhaftigkeit der *secondclass*-Transaktionen und für die Atomarität statt. Um eine unvollständig vollzogene Transaktion im möglichen Fehlerfall zurücksetzen zu können, bedarf es einiger Verwaltungsmechanismen wie Loggen der ausgeführten Operationen. Dieser Prozess findet dabei auf dem Client statt und belastet die ohnehin schon knappen Ressourcen noch mehr. Um Ressourcen zu sparen und weil diese sogenannte *alles-oder-nichts*-Eigenschaft nicht immer wünschenswert ist, wird die Atomarität nicht gewährleistet. Einzig die Isolationseigenschaft bleibt unverändert. Mit den im folgenden aufgelisteten Konsistenzgarantien sollen IOTs als eine zusätzliche Option für bessere Konsistenzsicherung im mobilen Umfeld sorgen. Gleichzeitig werden die vom IOT-Modell angebotenen Konfliktauflösungmöglichkeiten dargestellt:

Serialisierbarkeit (SR) : Die Ausführung einer *firstclass*-Transaktion ist garantiert serialisierbar mit allen anderen durch COMMIT abgeschlossenen Transaktionen. Dies ist klar, da *firstclass*-Transaktionen wie Transaktionen im klassischen Transaktionsmodell mit fest verbundenen Clients arbeiten und dadurch Serialisierbarkeit mittels Sperrprotokollen gesichert werden kann.

Lokale Serialisierbarkeit (LSR) : Für die Ausführung einer *secondclass*-Transaktion wird Serialisierbarkeit mit anderen *secondclass*-Transaktionen auf dem gleichen Client garantiert. Auch dies ist klar, da die Ergebnisse von *secondclass*-Transaktionen für nachfolgende Transaktionen auf dem Client verfügbar sind und lokal auf dem Client traditionelle Transaktionsmodelle zum Einsatz kommen können.

Globale Serialisierbarkeit (GSR) : Eines der Konsistenzkriterien zur Validierung einer *schwebenden* Transaktion ist, daß diese global serialisierbar mit allen durch COMMIT abgeschlossenen Transaktionen ist. Dabei unterscheidet sich GSR von LSR oder SR in der Weise, daß GSR nicht zum Zeitpunkt der Transaktionsausführung erzwungen, sondern nur beim Reintegrieren getestet werden kann. Schlägt dieser Test fehl, gibt es folgende automatische Auflösungsoptionen zur Wiederherstellung der Konsistenz:

- Nochmaliges Ausführen der Transaktion (*re-execution*) unter Nutzung der aktuellen Daten auf dem Server. Dies ist der Standardwert und folgt dem Konzept des Two-Tier-Modells aus Abschnitt 4.2.
- Verwendung eines anwendungsspezifischen Auflösungsskriptes (*ASR*). Das IOT-Modell erlaubt dem Nutzer ein ASR-Skript einer Transaktion zuzuordnen, welches dann im Konfliktfall automatisch vom System aufgerufen wird.
- Abbruch der Transaktion durch ABORT.
- Benachrichtigung des Nutzer und markieren der entsprechenden Transaktion für eine manuelle Auflösung.

Globale Zertifikationsfolge (GCO) : In manchen Situation ist GSR allein nicht ausreichend für unverbundene Clients, wenn zum Beispiel *read-write*-Konflikte dadurch

umgangen werden, daß die Transaktion, deren Ergebnisse auf veralteten Daten aufbauen, vor der Transaktion neu ausgeführt wird, die diese gelesenen Daten überschreibt und auf den aktuellen Stand bringt. Damit sind die von der vorangegangenen Transaktion erzeugten Resultate möglicherweise nicht mehr gültig. Um dieses Problem zu lösen, fordert GCO, daß eine *schwebende* Transaktion nicht nur mit, sondern auch nach allen anderen mit COMMIT abgeschlossenen Transaktionen serialisierbar ist. Für GCO gibt es die gleichen Auflösungsoptionen wie für GSR.

Die Zusicherung obiger Garantien erfordert allerdings zusätzliche Performance-Einbußen. Die Erkennung von Konflikten, wie beispielsweise eine Verletzung der globalen Serialisierbarkeitsforderung (GSR) , wird über Kreisdetektion in einem Abhängigkeitsgraphen aller *secondclass*-Transaktionen realisiert.

4.6.4 Implementierung

Eine Implementierung in bestehende Coda-Systeme ist vollzogen und wird in [LS95] beschrieben. Allerdings sind Parallelen zum Two-Tier-Modell und Bayou nicht zu übersehen, da auch hier die zweistufige Transaktionsbehandlung mit lokalem Abschluß und späterer Reintegration durch vorrangig erneutes Ausführen verwendet wird. Problematisch bezüglicher der Performance ist jedoch das Testen der globalen Serialisierbarkeit (GSR) mittels Kreisdetektion in einem Abhängigkeitsgraphen, welche in Bayou mit ebenfalls nicht trivialen Zeitstempeln garantiert wurde.

4.7 Offen geschachtelte Transaktionen

4.7.1 Architektur

Das in [Chr93] vorgestellte Modell setzt eine schwache Verbindung für das Arbeiten auf dem mobilen Client voraus und unterstützt demzufolge kein unverbundenes Arbeiten. Das Prinzip der Offen geschachtelten Transaktionen [GR93, WS92] wird für die Verwendung im mobilen Szenario etwas modifziert, um folgende beiden Ziele zu erreichen. Zum einen wird versucht, Teile des Zustands und der Durchführung von Transaktionen des Clients aufgrund der in Abschnitt 2 beschriebenen Beschränkungen mobiler Geräte auf Server im festen Netzwerk zu verlagern. Zum zweiten soll die Verfügbarkeit von Daten durch Abschluß von Teiltransaktionen erhöht werden. Transaktionen zur Erfüllung der gestellten Ziele können somit nicht die Eigenschaft der Atomarität besitzen. Zudem muß auch die Isolation von Transaktionen aufgehoben werden, da diese eine Transaktion vor dem Freigeben von Zwischenergebnisse bzw. Zuständen bewahrt.

Als Netztopologie wird hier ein Zellsystem zugrundegelegt, deren Basisstationen die Middleware bilden und für die teilweise Ausführung von Client-Transaktionen verantwortlich sind. Desweiteren haben sie die Aufgabe, die Bewegung des mobilen Clients von einer Funkzelle zur nächsten mit Hilfe bestimmter Teiltransaktionen (*berichtende* und *mitwirkende* Transaktionen) zu koordinieren. An das zugrundeliegenden Datenbanksystem werden vom Modell keine einschränkenden Forderungen gestellt, vorstellbare wäre also ein zentrales DBMS auf einem Server oder auch ein über die versorgenden Basisstationen verteiltes DBMS. Die Aufgabenverteilung auf Server und Client ist nicht klar aus der Modellbeschreibung zu erkennen, da auf eine Konflikterkennung und Auflösung nicht näher

eingegangen wird. Eine solche Komponente sollte jedoch auf der Basisstation oder einem zentral gelagerten Server liegen, um die parallel arbeitenden Clients zu koordinieren.

4.7.2 Nutzer

Für den Nutzer gibt es kaum Einschränkungen gegenüber dem Arbeitsverhalten an einem festverbundenen Client, da seine lokalen Änderungen über die schwache Verbindung abgeschlossen und validiert werden und sofort sichtbar sind. Da das Modell nicht explizit eine optimistische Replikation verwendet und damit Daten sehr wahrscheinlich pessimistisch repliziert werden, sind Konflikte zwischen konkurrierenden Änderungen nicht zu erwarten. Die Bewegungsfreiheit des Nutzers ist durch das Aufrechterhalten einer schwachen Verbindung auf den netzabgedeckten Bereich eingeschränkt, ein Zellwechsel wird allerdings durch das Modell unterstützt.

4.7.3 Datenhaltung und -zugriff

Eine mobile Transaktion wird in [Chr93] als eine Menge von relativ unabhängigen Teiltransaktionen beschrieben. Eine Teiltransaktion kann nach dem Prinzip der offen geschachtelten Transaktionen [GR93] in weitere Teiltransaktionen bzw. Komponenten aufgeteilt sein. Dadurch kann eine beliebige Schachtelung erreicht werden. Hier wird angenommen, daß t eine mobile Transaktion mit einer 2-Level-Schachtelung ist und n Transaktionskomponenten $t_1, \ldots, t_n$ hat. Einige dieser Komponenten sind kompensierbar, die entsprechende Kompensationstransaktion wird mit $comp_{t_i}$ bezeichnet. Eine Kompensationstransaktion macht semantisch gesehen die Effekte von t_i ungeschehen, wobei die Datenbank nicht notwendigerweise in den Zustand vor der Ausführung von t_i versetzt wird.

Transaktionskomponenten können unabhängig (unilateral) abschließen, d.h. ohne auf den Abschluß anderer Komponenten t_i oder der Transaktion t zu warten. Wird allerdings t mit ABORT abgebrochen, so werden auch alle noch nicht mit COMMIT abgeschlossenen Transaktionen abgebrochen. Voraussetzung für die erfolgreiche Umsetzung obiger Annahmen ist die gute Zerlegbarkeit einer mobilen Transaktion in eine relativ unabhängige Menge von Transaktionskomponenten, d.h. es gibt keine zusammenhängende langlebende Folge von Operationen. Vom Modell werden folgende Typen von Transaktionskomponenten unterschieden:

Atomare Transaktionen : Diese besitzen die klassischen Eigenschaften von COMMIT und ABORT. Atomare Transaktionen gliedern sich in kompensierbare und kompensierende Transaktionen mit entsprechenden Abhängigkeiten. Kompensationstransaktionen müssen aus Konsistenzgründen in umgekehrter Reihenfolge zur Abschlußfolge ihrer entsprechenden Transaktionskomponenten ausgeführt und abgeschlossen werden. Bricht t nach dem COMMIT einer kompensierbaren Komponente t_i ab, so muß die Kompensationstransaktion $comp_{t_i}$ ebenfalls noch mit COMMIT abgeschlossen werden.

Nichtkompensierbare Transaktionen : Diese Transaktionskomponenten haben keine assoziierte Kompensationstransaktion. Sie können auch jederzeit mit COMMIT abschließen, dürfen aber ihre Resultate auf Daten nicht festschreiben, da sie nicht kompensierbar sind. Zum COMMIT-Zeitpunkt werden alle enthaltenen Operationen an t

(die mobile Transaktion) delegiert. *Delegation* ist die Möglichkeit einer Transaktion t_a, die Verantwortung über COMMIT oder ABORT einer Operationen an eine andere Transaktion t_b zu übergeben und wird mit $Delegate_{t_a}[t_b, op]$ bezeichnet. Sobald diese Übergabe stattgefunden hat, liegt die Situation vor, als wäre die Operation op von t_b und nicht von t_a durchgeführt worden. Delegation kann zur Kontrolle der Sichtbarkeit von Objekten eines Transaktionsmodells verwendet werden. Soll beispielsweise eine Teilmenge von Resultaten einer Transaktion beim ABORT weder ungeschehen noch permanent gemacht werden, dann können diese Resultate zu einer entsprechenden Transaktion delegiert werden. Dadurch gehen die Auswirkungen der delegierten Operationen nicht verloren, selbst wenn die delegierende Transaktion abbricht.

Berichtende Transaktionen : Diese Komponenten können ihre Teilergebnisse mit t teilen, indem eine berichtenden Komponente t_i während der Ausführung zu einem beliebigen Zeitpunkt einige Resultate an t delegiert. Abhängig davon, ob t_i kompensierbar ist oder nicht, werden zum COMMIT-Zeitpunkt von t_i alle vorher noch nicht berichteten Ergebnisse an t delegiert oder nicht. Im Gegensatz zu den mitwirkenden Transaktionen können diese parallel ausgeführt werden.

Mitwirkende Transaktionen : Diese Transaktionskomponenten sind eine Variante der berichtenden Transaktionen, bei der zum Zeitpunkt der Delegation von Teilergebnissen die Kontrolle ebenfalls mitübergeben wird, d.h. die mitwirkende Transaktion wird ausgesetzt und nimmt die Ausführung zum Zeitpunkt der Kontrollrückgabe an gleicher Stelle wieder auf. Damit bewahren diese Transaktionen im Gegensatz zu nichtkompensierbaren Transaktionen ihren Zustand über Ausführungen hinweg.

Sowohl berichtende Transaktionen (*reporting transaction*) als auch mitwirkende Transaktionen (*co-transaction*) sind in [CR94] genauer beschrieben. Beide stehen in einer engen Wechselbeziehung zueinander und sind in der Lage, viele Ergebnisse auszutauschen. Dadurch können sie ihre Ausführung von einer Basisstation zur nächsten verlagern und so ihre Kommunikationskosten minimieren. Beispielsweise kann eine mitwirkende Transaktion auf der Basisstation ausgeführt werden und ihre entsprechende berichtende Transaktion auf dem mobilen Client. Damit kann die mitwirkende Transaktion über die untereinander verbundenen Basisstationen der Bewegung des mobilen Client folgen, auf dem die berichtenden Transaktion ausgeführt wird.

Aus den in [Chr93] aufgeführten Axiomen zur genauen Definition von berichtenden und mitwirkenden Transaktionen geht hervor, daß durch die Delegation von Operationen zwischen der berichtenden und der mitwirkenden Transaktion eine *Abbruch-Abhängigkeit* entsteht, d.h. daß die berichtende Transaktion ebenfalls abbrechen muß, wenn die mitwirkende Transaktion abbricht. Damit werden die Resultate der berichtenden Transaktion erst dauerhaft in der Datenbank gemacht, wenn die mitwirkende Transaktion mit COMMIT abschließt. Weiterhin wird einer Transaktion durch ein Axiom verboten, daß sie zu mehr als einer Transaktion Operationen und Resultate delegiert. Für mitwirkende Transaktionen gibt es zusätzlich noch eine *zwingende COMMIT-Abhängigkeit*, die die berichtende Transaktion ebenfalls zum COMMIT veranlaßt, sobald die mitwirkende Transaktion mit COMMIT abschließt. Durch diese beiden Abhängigkeiten werden entweder beide, d.h. die berichtende und mitwirkende Transaktion, oder keine mit COMMIT abgeschlossen.

Abgesehen von der nichttrivialen Forderung, daß einige der Komponenten kompensierbar sind, werden keine weiteren Voraussetzungen an Transaktionen und die verwendeten

Daten gestellt, somit könnte das Modell grundsätzlich auf jedem existierenden Datenbanksystem ohne größere Umstrukturierungen implementiert werden. Zusätzliche Informationen werden nicht benötigt bzw. nicht im Modell erwähnt. Allerdings ist der Aufwand für die Beschaffung und Durchführung von Kompensationstransaktionen entsprechend hoch. Geht man davon aus, daß das Zurücksetzen von Transaktionen mittels Kompensationstransaktionen keine Gefährdung der Dauerhaftigkeit im engeren Sinne ist, so werden nur folgende zwei ACID-Eigenschaften abgeschwächt:

- Die Atomarität verlangt nicht, daß die aufrufende Transaktion von Operationen die Transaktion sein muß, die diese Operationen mit COMMIT oder ABORT abschließt. Hierbei muß beachtet werden, daß die Auswirkungen der Operationen auf ein Objekt nicht zum Ausführungszeitpunkt der Operation permanent werden, sondern erst durch ein explizites Abschließen. Dadurch können delegierte Operationen durch diejenige Transaktion mit COMMIT oder ABORT abgeschlossen werden, die die Kontrolle darüber erlangt hat. Andere nicht delegierte Operationen werden durch die aufrufende Transaktion zum Abschluß gebracht. Dies kennzeichnet eine Transaktion als *quasi fehleratomar*, d.h. wenn alle Operationen, für die sie verantwortlich ist, abgeschlossen werden oder keine.
- Eine Verletzung der Isolation ist dadurch möglich, daß auch die Ergebnisse von vorzeitig abgeschlossenen Komponenten für andere Clients sichtbar werden und möglicherweise durch Kompensationstransaktionen rückgängig gemacht werden. Dann müsste allerdings jede andere Transaktion, die auf dann ungültigen Daten aufgebaut hat, ebenfalls zurückgesetzt werden, was nicht sicherzustellen ist.

Hiermit werden nun auch mehr Phänomene zugelassen, wie beispielsweise Fuzzy Read, indem Änderungen einer abgeschlossenen Transaktionskomponente vor Abschluß der Gesamttransaktion und damit endgültiger Dauerhaftigkeit sichtbar sind. Zur Konflikterkennung und -auflösung werden vom Modell keine Angaben gemacht, da diese aufgrund der schwachen aber doch häufigen Verbindung zwischen Client und Server kaum ein Rolle spielen.

4.7.4 Implementierung

Ein auf dem beschriebenen Modell basierender Prototyp existiert nicht. Der Hauptgrund hierfür liegt unter anderem auch in der Forderung nach Kompensationstransaktionen für einen sinnvollen Einsatz.

4.8 Kangaroo Transactions

4.8.1 Architektur

Dieses Modell von [DHB97] baut auf dem im Abschnitt 4.7 vorgestellten Modell der Offen geschachtelten Transaktionen [WS92], den globalen Transaktionen in einem Multi-Datenbanksystem [BGMS92, WS92] und den Split-Transaktionen [PKH88] auf und beachtet vorallem das Ausführungsverhalten einer mobilen Transaktion. Dabei erfordert die Bewegung des mobilen Nutzers in einem Zellsystem (Abbildung 1), daß Transaktionen möglicherweise zellübergreifend ausgeführt werden und somit Startpunkt und Endpunkt

nicht identisch sein müssen. Einem mobilen Client muß es laut Modell möglich sein, eine drahtlose Verbindung zum festen Netzwerk zu jedem Zeitpunkt aufrechthalten zu können, da unverbundenes Arbeiten nicht unterstützt wird. Transaktionsbeginn, Lese- und Schreiboperationen sowie Transaktionsabschluß benötigen die schwache Verbindung, da die jeweilige Transaktionsverarbeitung auf den Basisstationen des zugrundegelegten Zellsystems stattfindet. Das Referenzmodell besteht aus den folgenden drei Ebenen:

Grundsystem: Dieses befindet sich auf allen drei Komponenten der Zellsystemarchitektur, d.h. auf den festverbundenen Servern, den Basisstationen und den mobilen Clients. Das Grundsystem bietet dem mobilen Nutzer gewisse Informationsdienste wie beispielsweise Wetterdienst und Aktienkurse an und kann ein beliebiges existierendes stationäres System mit einem Client/Server- oder *peer-to-peer*-Design sein.

Datenzugriffsagent: Er koordiniert den Zugriff mobiler Transaktionen auf die Daten im Grundsystem (vorallem beim Zellwechsel des Nutzers) und unterstützt die Recovery. Der Datenzugriffsagent, *Data Access Agent (DAA)*, befindet sich ausschließlich jeweils auf den Basisstationen.

Mobile Transaktion: Diese werden sowohl auf mobilen Clients als auch auf Basisstationen ausgeführt und gruppieren die für eine Anfrage des mobilen Nutzers nötigen Operationen.

Alle drei Ebenen bilden gemeinsam die Middleware und sind für die Durchführung von mobilen Transaktionen (*Kangaroo Transactions*) verantwortlich. Bekommt der DAA eine Transaktionsanfrage eines mobilen Nutzers, leitet er sie zur entsprechenden Basisstation oder einem anderen festverbundenen Knoten weiter, der die benötigten Daten vorhält. Wird der mobile Nutzer durch seine Eigenbewegung aus der aktuellen Funkzelle in eine andere übergeben, dann erhält der DAA in der neuen Basisstation alle Transaktionsinformationen der alten Basisstation. Damit kann eine mobile Transaktion eine Menge von Knoten (Basisstationen) während ihrer Verarbeitung durchlaufen, wobei diese Menge erst am Ende der Transaktionsausführung vollständig bekannt ist. Eine wichtige Komponente des DAA ist der *Mobile Transaktionsmanager (MTM)*, der sich ebenfalls auf jeder Basisstation befindet und für das Verfolgen des Ausführungsstatus aller gleichzeitig laufenden mobilen Transaktionen einer Basisstation verantwortlich ist.

Grundlegendes Prinzip des Modells ist es, die Eigenschaften der mobilen Umgebung vor dem zugrundeliegenden DBMS zu kapseln und umgekehrt. Um einen Benachrichtigungs-Overhead, resultierend aus der Bewegung des mobilen Nutzers von einer Zelle zur anderen, zu vermeiden, wird angenommen, daß sich der MTM mit der mobilen Einheit bewegt. Aufgrund der ständigen, wenn auch schwachen, Verbundenheit des Clients mit dem Server entstehen die für das mobile Umfeld typischen Konflikte beim Wiedereinbringen lokaler unverbundener Änderungen hier nicht, daher beschränkt sich der Verantwortungsbereich des Servers auf die in traditionellen Datenbankmanagementsystemen entstehenden Aufgaben.

4.8.2 Nutzer

Das Modell unterstützt im großen und ganzen das traditionelle Arbeiten, wobei die sonst feste Verbindung zwischen Client und Server über eine drahtlose Verbindung realisiert

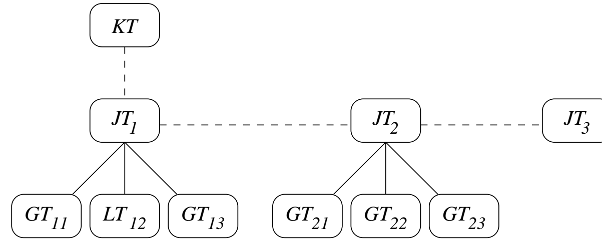


Abbildung 7: Eine mögliche Konfiguration für Kangaroo Transactions

wird. Allein die damit verbundenen Probleme wie Netzausfall in Funklöchern oder auch Einschränkungen in der Übertragungskapazität lassen den mobilen Aspekt sichtbar werden. Werden mobile Transaktionen im später erläuterten *Compensating Mode* ausgeführt, werden vom Nutzer zusätzlich Informationen zu Kompensationstransaktionen benötigt. Dagegen dürften zurückgewiesene Änderungen des Clients aufgrund von Konflikten so gut wie nicht ins Gewicht fallen, weil für deren Entstehung die Grundlage der unverbundenen Änderung fehlt. Allerdings sollte sich der Nutzer nur in dem vom Zellsystem abgedeckten Bereich aufhalten und sich auch nur mit einer durch technische Grenzen festgesetzten Geschwindigkeit fortbewegen.

4.8.3 Datenhaltung und -zugriff

Besondere Eigenschaften an die Daten und Operationen werden vom Modell nicht vorausgesetzt, da es auf einem existierenden DBMS aufbaut, ohne dieses zu verändern. Das zugrundeliegende Prinzip der Offen geschachtelten Transaktionen [WS92] zeigt sich im Aufbau von mobilen Transaktionen in diesem Modell, dabei sei auf Abbildung 7 als eine beispielhafte Konfiguration verwiesen. Wird vom Benutzer eine Transaktionsanfrage gestartet, erzeugt der DAA auf der zum mobilen Nutzer zugehörigen Basisstation die sogenannte *Kangaroo Tranaction (KT)*. Diese umfasst alle weiteren Transaktionen des Nutzers und kennzeichnet in diesem Modell eine mobile Transaktion. In dieser Transaktion befinden sich eine Stufe tiefer geschachtelt die sogenannten *Joey Transaction (JT)*. Jede JT ist dabei für die Transaktionsverarbeitung innerhalb einer bestimmten Zelle zuständig und wird vom DAA auf der Basisstation koordiniert. Beim Zellwechsel des Nutzers (*handoff*) wird als Folge der Kontrollweitergabe über die KT an den DAA der nächsten Basisstation sofort eine neue JT innerhalb der KT erzeugt, wie dies in Abbildung 7 zu sehen ist. Durch dieses Splitten wird das Bewegungsverhalten des Nutzers explizit in Untertransaktionen abgebildet. Schließlich kann eine JT aus beliebig vielen weiteren Untertransaktionen (*lokale/globale Transaktionen*) bestehen, die allerdings hier auf dem Grundsystem ausgeführt werden. Diese repräsentieren die eigentlichen Anfragen als eine Folge von Lese- und Schreiboperationen mit COMMIT oder ABORT, wobei globale Transaktionen erneut geschachtelt sein können. Zusätzlich werden alle lokalen und globalen Transaktion, die zu einer Kangaroo Transaktion gehören, als Beutel (*Pouch*) bezeichnet. Zwei Kangaroo Transaktionen sind äquivalent, falls sie den gleichen Beutel haben.

Joey-Transaktionen können voneinander unabhängig mit COMMIT abschließen. Trotzdem kann ein Fehler einer JT zur Ungültigkeit einer ganzen KT führen. Für die Ausführung von KT's gibt es zwei Modi:

Compensating : Ein Fehler einer JT veranlaßt die Annullierung mittels Kompensationstransaktion der aktuellen JT und aller vorangegangenen oder folgenden JT's, d.h. der gesamten KT. Dazu werden Informationen vom Nutzer oder vom Grundsystem benötigt, um kompensierende Transaktionen zu erzeugen. Dieser Modus garantiert Atomarität und Serialisierbarkeit innerhalb von JT's, nicht aber die Isolation von Transaktionen.

Split : Dies ist die Grundeinstellung. Falls eine JT zu einem Fehler kommt, wird keine neue globale oder lokale Transaktion als Teil der aktuellen KT angefordert. Jedoch wird die Entscheidung, ob die aktuell ausgeführten Transaktionen mit COMMIT oder ABORT beendet werden, dem DBMS überlassen. Transaktionen, die vor dem Auftreten des Fehlers abgeschlossen wurden, werden nicht kompensiert.

Um also die Autonomität des zugrundeliegenden DBMS zu sichern, sollte eine KT im ersten Modus ausgeführt werden, wofür Kompensationstransaktionen vorliegen müssen.

Zusätzliche Informationen zur Konfliktlösung werden nicht benötigt, allerdings ist für Verwaltung und Koordination der einzelnen Subtransaktionen (JT) auf den für die Bewegung des Nutzer verantwortlichen Basisstationen eine Menge von Identifikatoren nötig. Dies übernimmt aber der DAA auf der Basisstation, somit ist der Aufwand für deren Verwaltung nicht vom Client zu tragen. Je nach Ausführungsmodus für KT findet möglicherweise eine Abschwächung der ACID-Eigenschaften entweder für die Isolation im *Compensating*-Modus oder für die Atomarität im *Split*-Modus statt, weil dann im letzten Fall Teile einer KT akzeptiert würden und andere nicht. Weder der *Compensating*- noch der *Split*-Modus garantieren Serialisierbarkeit einer KT, d.h. also aller JT untereinander. Es wird dabei angenommen, daß die lokalen Transaktionen einer JT nacheinander ausgeführt werden, d.h. der Nutzer fordert die nächste Untertransaktion erst nach Beendigung der aktuellen an. Dies garantiert, daß eine JT_i in serialisierbarer Reihenfolge der nächsten JT_{i+1} vorangeht. Allerdings können lokale Transaktionen der alten JT noch ausgeführt werden, während eine neue JT beim Zellwechsel dynamisch erzeugt wird.

4.8.4 Implementierung

Ein Prototyp existiert für das vorgestellte Modell noch nicht, dafür aber einige Beispielszenarien, die von diesem Modell hervorragend unterstützt werden.

5 Bewertung der Modelle

In diesem Abschnitt findet eine zusammenfassende Bewertung aller vorgestellten Modelle anhand der Kriterien aus Abschnitt 3 statt. Dabei sollen Gemeinsamkeiten, einzelne Unterschiede und Besonderheiten aufgezeigt werden. In einer abschließenden Übersicht (Tabelle 2) sind die wichtigsten Merkmale für jedes Modell und für die im Abschnitt 6 vorgestellten Produkte dargestellt.

5.1 Architektur

Die vorangegangene Untersuchung zeigt, daß sich fünf der sieben Modelle (Two-Tier und Bayou, Multiversionsmodell, Semantische Transaktionen, Schwache/strikte Transaktionen

und Isolation-Only Transactions) für den unverbundenen Einsatz im mobilen Umfeld eignen. Die anderen beiden Modelle (Offen geschachtelte Transaktionen und Kangaroo Transactions) benötigen eine schwache Verbindung zum Transaktionsabschluß und für alle anderen Schreibzugriffe, so daß eine permanent schwache Verbindung zwischen Client und Server vorausgesetzt wird. Dies ist aufgrund der hohen Kosten und geringen Bandbreite einer drahtlosen Verbindung eine relativ große Einschränkung für den Einsatz in der Praxis. Allerdings wird zumindest die verfügbare Bandbreite und die Netzabdeckung in Zukunft weiter ausgebaut werden und somit als begrenzender Faktor an Gewicht verlieren.

5.2 Nutzer

Mit der Wahl der Netztopologie eines Modells steht auch die Bewegungsfreiheit des Nutzers im engen Zusammenhang. Diese wird natürlich durch die Bindung an eine schwache Verbindung innerhalb eines Zellsystems eingeschränkt, weil dann der Nutzer den abgedeckten Funkbereich nicht verlassen darf, will er Datenänderungen vornehmen. Dagegen haben die Modelle mit einer schwachen Verbindung die Möglichkeit, Konflikte konkurrierender Änderungen durch pessimistische Replikation zu vermeiden. Dadurch entfällt der Aufwand zur Entwicklung komplexer Konfliktauflösungsalgorithmen, was sich in der fehlenden Beschreibung dieser Problematik in den beiden Modellen dieser Kategorie (Offen geschachtelte Transaktionen und Kangaroo Transactions) zeigt. Jedoch müssen neue Probleme, die „verteilte“ Transaktionsausführung über mehrere Basisstationen hinweg betreffend, gelöst werden.

Entscheidend in dieser Diskussion ist auch, daß sich die Zielstellungen von Modellen auf Grundlage des Zellsystem von denen der Modelle mit Unterstützung von Unverbundenheit wie am Beginn von Abschnitt 4 dargestellt unterscheiden. Hierbei wird der Nutzer in keiner Weise in seiner Mobilität eingeschränkt, wenn das Modell unverbundenes Arbeiten ermöglicht, da eine Netzbindung für das Arbeiten bzw. Ändern nicht erforderlich ist. Allerdings können dann beim Synchronisieren Konflikte zwischen Änderungen auftreten, die mitunter einen manuellen Eingriff zur Auflösung erfordern (Abschnitt 2).

Bei zwei Modellen, die das unverbundene Arbeiten unterstützen, gibt es zusätzliche Einschränkungen bezüglich der einsetzbaren Anwendungen. Das Wissen über die Semantik dieser Anwendungen wird für die Konfliktauflösung verwendet. So fordert Bayou, daß es eine gewisse Anzahl von Anwendungen gibt, die mit kommutativen Operationen arbeiten bzw. Konflikte in jedem Fall angemessen lösen können. Dazu zählen beispielsweise Terminkalender, News und gemeinsame Dokumententwicklungen. Im Fall der Semantischen Transaktionen werden vom Modell nur spezielle Datentypen unterstützt, die eine gute Zerlegbarkeit aufweisen bzw. wird die Kommutativität von Operationen gefordert. Beide Anforderungen werden allerdings von Daten bzw. Operationen in üblichen DBMS nicht erfüllt.

5.3 Datenhaltung und -zugriff

Replikation

Optimistische Replikation wird von allen Modellen verwendet, die das unverbundene Arbeiten unterstützen. Für die beiden Modelle mit einer permanent schwachen Verbindung

werden in den entsprechenden Modellbeschreibungen keine Aussagen darüber gemacht, sodaß auch eine pessimistische Replikation denkbar wäre. Für das Modell der Semantischen Transaktionen wird allerdings bei Verwendung der Zerlegbarkeitseigenschaft für Objekte explizit ein Sperren der zur Bearbeitung ausgegliederten Fragmente durch *check-in/check-out* [LP83] verlangt, wobei jedoch durch möglichst feine Sperrgranulate versucht wird, weiterhin eine hohe Datenverfügbarkeit zu erzielen. Optimistische Replikation ist nur bei Verwendung der Umsortierbarkeitseigenschaft, d.h. bei Kommutativität von Operationen, möglich.

Konflikterkennung und Konfliktauflösung

Modelle mit Unterstützung von Unverbundenheit bieten teilweise konkrete Konflikterkennungs- und Konfliktauflösungsalgorithmen an. Eine Konflikterkennung findet dabei unter der Verwendung von zusätzlichen Informationen wie *before/after image*, *read set* oder Zeitstempeln statt.

- Bayou:
  Hier wird mit der *mergeproc* eine Schnittstelle zur Definition einer vom Nutzer entwickelten anwendungsspezifischen Konfliktauflösung angeboten. Diese erzeugt im Kontext passende alternative Änderungen zu einer ursprünglichen Schreiboperation eines Nutzers.

- Multiversionsmodell:
  Hier wird ein optimaler Snapshot gesucht, der konsistent mit dem *read-set* der Transaktion ist. Dabei können vom Nutzer angepaßte Funktionen verwendet werden, um einen alternativen Snapshot im Konfliktfall bzw. dessen Verwendungskosten zu bestimmten. Wird allerdings ein solcher Snapshot nicht gefunden, sieht der Algorithmus den Abbruch der Transaktion vor.

- Semantische Transaktionen:
  Dieses Modell vermeidet Änderungskonflikte durch das Sperren von Fragmenten gut zerlegbarer Objekte bzw. durch die Forderung nach kommutativen Operationen. Einerseits wird dadurch zwar Aufwand zur Entwicklung eines Konfliktauflösungsalgorithmus vermieden, andererseits sind die Einsatzmöglichkeiten in der Praxis auf entsprechende Daten- und Operationstypen eingeschränkt.

- Isolation-Only Transactions:
  Hier werden vier Optionen im Falle eines entdeckten Konfliktes als Lösung angeboten:

  - Wiederausführung auf aktuellen Daten analog zum Two-Tier-Konzept
  - anwendungsabhängige Auflösungs (ASR) analog zur *mergeproc* von Bayou
  - einfacher Abbruch der Transaktion wie im klassischen Fall
  - manueller Eingriff, ist im mobilen Umfeld jedoch eher ungeeignet, da Konflikte zwischen eingebrachten Änderungen möglicherweise erst nach Abkopplung des betroffenen Nutzers entdeckt werden

Transaktionsfehler in Modellen mit einer schwachen Verbindung, beispielsweise der Abbruch einer geschachtelten Transaktion durch Verlassen der Netzabdeckung (keine Änderungskonflikte), werden von den beiden Modellen dieser Kategorie (Offen geschachtelte Transaktionen und Kangaroo Transactions) durch Kompensation von schon abgeschlossenen Teiltransaktionen gelöst, soweit diese kompensierbar sind.

Verwendung von Zusatzinformationen

Die Verwendung von zusätzlichen Informationen zumeist für die Konflikterkennung und Konfliktauflösung variiert zwischen den Modellen in Bezug auf Menge und Aufwand, der für die Beschaffung entsteht. Die beiden Modelle Two-Tier und Multiversionsmodell benötigen jeweils *read-set* und Zeitstempel. Beide Zusatzinformationen sind nicht einfach zu erhalten. Das *read-set* einer Transaktion erfordert das Aufzeichnen aller gelesenen Daten für eine Transaktion, was zu erhöhtem E/A-Aufwand und Speicherplatzbedarf führt. Zeitstempel sind aufgrund technischer Grenzen nicht vollständig synchron realisierbar, da hierfür exakt gleichgetakte Uhren auf allen involvierten Geräten zur Verfügung stehen müssten.

Andere Modelle benötigen entweder weniger Informationen zur Konflikterkennung und Konfliktauflösung oder werden diesbezüglich nicht genau genug vorgestellt und beschrieben. Ebenso werden in kaum einem der Modelle, die Kompensationstransaktionen verwenden (Schwache/strikte Transaktionen, Offen geschachtelte Transaktionen und Kangaroo Transactions), die Schwierigkeiten in Hinblick auf die Bestimmung der in der Kompensationstransaktion enthaltenen kompensierenden Operationen aufgezeigt. Weiterhin ist der Nutzer für die Bereitstellung von Informationen für Kompensationstransaktionen verantwortlich. In anderen Modellen (z.B. Semantische Transaktionen) werden zusätzliche Informationen als Forderungen an Daten oder Operationen formuliert.

Abschwächung der ACID-Eigenschaften

Ein weiterer Schwerpunkt der Bewertung der Modelle liegt in der Abschwächung der für traditionelle Transaktionen zugesicherten ACID-Eigenschaften. Hierbei ist grundsätzlich die Art der unterstützten Verbindung zu unterscheiden.

unverbunden: Hier werden Transaktionen lokal abgeschlossen, folgende Transaktionen auf dem Client bauen unter Umständen auf den vorläufig abgeschlossenen Daten auf. Beim Synchronisieren und Einbringen der lokalen Änderungen kann es zu Konflikten kommen. Deshalb unterscheiden beispielsweise die Transaktionsmodelle Schwache/strikte Transaktionen und Isolation-Only Transactions explizit zwischen Transaktionen mit Zugriff (schwache Transaktionen) und ohne Zugriff (strikte Transaktionen) auf vorläufige Daten. Durch das unter Umständen notwendige Rücksetzen der Transaktion ist die Eigenschaft der Dauerhaftigkeit gefährdet und damit auch die Isolation, wenn Client-Transaktionen Daten einer später konflikterzeugenden Transaktion gesehen haben. Zur Erhaltung der Isolation ist fortgeführtes Zurücksetzen abhängiger Transaktionen (*cascading abort*) nötig.

Eine Ausnahme bildet in dieser Kategorie das Modell der Semantischen Transaktionen, da es aufgrund von Sperren zu keinen Konflikten kommen kann. Für das Two-Tier-Modell kann es eventuell zu einer Abschwächung der Atomarität durch

die Konfliktauflösung kommen, wenn Teile einer Transaktion mit einem anderen (alternativen) Ergebnis eingebracht werden.

schwach verbunden: Das von den hier betrachteten Modellen verwendete Prinzip der Schachtelung von Transaktionen in gewisse Komponenten verletzt die Eigenschaft der Atomarität, wenn man davon ausgeht, daß die ursprünglich mobile Transaktion als eine Einheit betrachtet wird. Betrachtet man dagegen einzelne Teiltransaktionen, ist für diese die Atomarität gewährleistet, wenn kompensierende Transaktionen existieren. Da auch hier die Ergebnisse der Sub-Transaktionen nach Abschluß dieser für andere Transaktionen eines mobilen Clients sichtbar werden, ist wiederrum die Isolationseigenschaft verletzt, sollte es zum Abbruch der umfassenden Transaktion kommen. Auch hier müssen zur Gewährleistung der Isolation schon abgeschlossene Komponenten rückgängig gemacht werden, was zum fortgesetzten Abbruch führen kann.

5.4 Implementierung

Anhand der Zahl der Modelle mit existierenden Prototypen (Bayou und Isolation-Only Transactions) läßt sich erkennen, daß eine umfassende Transaktionsunterstützung für mobile Datenbanksysteme noch nicht existiert, sondern vorerst nur Lösungen für Einzelprobleme entwickelt wurden. Die vorgestellten Modelle präsentieren teilweise verschiedene, teilweise aufeinander aufbauende Ansätze, um die einleitend diskutierten Probleme mehr oder weniger gut zu lösen. Jedoch verhindert gerade die Forderung nach zusätzlichen Informationen wie das *read-set* einer Transaktion oder global eindeutiger Zeitstempelvergabe eine Implementierung, da sich die Umsetzung dieser Forderungen meist nicht effizient realisieren läßt.

6 Exkurs Datenbank- und Dateisystemprodukte

6.1 Abgrenzung

Die folgende Übersicht zeigt die derzeit existierenden Produktlösungen für Datenbanken im mobilen Umfeld, die in der Arbeit von [Fan00] ausführlich vorgestellt werden. Hiervon werden im Anschluß nur diejenigen Produkte näher betrachtet, die eine transaktionsorientierte Arbeitsweise für die Replikations- und Reintegrationskomponente aufweisen. Dies sind die Produkte *IBM DB2 DataPropagator* [IBM99a] und *Sybase Adaptive Server Anywhere 7.0.0* [Syb00a].

- Oracle 9*i* Lite (Oracle 8*i* Lite)
- IBM DB2 DataPropagator
- IBM DB2 Everyplace Version 7.2.1 (IBM DB2 Everywhere Version 1.2)
- Sybase Adaptive Server Anywhere 7.0.0
- Sybase UltraLite Version 7.0

	Two-Tier und Bayou	Multiversionsmodell	Semantische Transaktionen	Schwache/strikte Transaktionen	Isolation-Only Transactions	Offen geschachtelte Transaktionen	Kangaroo Transactions	IBM DB2 DataPropagator	Sybase Adaptive Server Anywhere 7.0.0
Architektur und Nutzer									
○ Zellsystem wird zugrundegelegt	–	–	✓	–	–	✓	✓	–	–
○ Unterstützung von Unverbundenheit	✓	✓	✓	✓	✓	–	–	✓	✓
○ Client/Server-Architektur	–	✓	✓	–	✓	✓	✓	✓	✓
○ verteilte DBMS-Architektur	✓	–	–	✓	–	–	–	–	–
○ Einschränkung durch mögliche Änderungskonflikte	✓	✓	–	✓	✓	–	✓	✓	✓
○ Einschränkung in der Anwendungsfunktionalität	✓	–	–	–	✓	–	–	✓	–
○ Prototyp/Produkt existiert	✓	–	–	–	✓	–	–	✓	✓
Datenhaltung und -zugriff									
○ Optimistische Replikation	✓	✓	✓[1]	✓	✓	–	–	✓	✓
○ Zusatzinformationen/Eigenschaften nötig									
· read-set	✓	✓	–	–	–	–	–	–	–
· Transaktionssemantik/Kontext	✓	–	–	–	–	–	–	–	–
· Zeitstempel	✓	✓	–	–	✓	–	–	–	–
· Fragmentierbarkeit von Objekten	–	–	✓[2]	–	–	–	–	–	–
· Kommutativität von Transaktionen	–	–	✓[2]	–	–	–	–	–	–
· Kompensationstransaktionen	–	–	–	✓	–	✓	✓	–	✓
○ Abschwächung von ACID									
· Atomarität	✓	–	–	–	✓	✓[3]	✓[3]	–	✓[4]
· Konsistenz	–	–	–	✓[5]	–	–	–	–	–
· Isolation	✓	✓	–	–	–	✓	✓	–	–
· Dauerhaftigkeit	✓	✓	–	✓	✓	–	–	✓	✓
○ Konflikterkennung	✓	✓	n/a	✓	✓	n/a	n/a	✓	✓
· Abhängigkeitsgraph	–	–	–	✓	✓	–	–	–	–
· Akzeptanzkriterium	✓	–	–	–	–	–	–	–	✓
· Testen aller Snapshots	✓	✓	–	–	–	–	–	–	–
· *before/after image*-Vergleich	–	–	–	–	–	–	–	✓	✓
○ automatische Konfliktauflösung	✓	✓	n/a	✓	✓	n/a	n/a	✓	✓
· Benutzerdefinierte Konfliktauflösungsprozedur	✓	✓	–	–	✓	–	–	✓	✓
· Snapshotsuche/Multiversionsmodell	–	✓	–	–	–	–	–	–	–
· Transaktionsabbruch	✓	✓	–	✓	✓	–	–	✓	✓

✓. . . ja –. . . nein n/a. . . nicht anwendbar

Tabelle 2: Übersicht der Transaktionsmodelle und Produkte

[1] sowohl optimistische (wenn Fragmentierbarkeit erfüllt ist) als auch pessimistische (wenn Kommutativität erfüllt ist) Replikation anwendbar (Abschnitt 4.4)
[2] wenigstens eine der beiden Eigenschaften muß erfüllt sein (Abschnitt 4.4)
[3] Transaktion in Subtransaktionen geschachtelt (Abschnitte 4.7 und 4.8)
[4] durch attributbezogene Konfliktauflösung über RESOLVE UPDATE-Trigger (Abschnitt 6.3)
[5] konsistente Änderungen nur innerhalb eines Konsistenz-Clusters möglich (Abschnitt 4.5)

Auch in Dateisystemen wurden Möglichkeiten zur Unterstützung unverbundener Nutzer geschaffen. Sie werden für einfache mobile Anwendungen vorallem auf Kleinstgeräten, wie Handhelds, eingesetzt. Obwohl sie keine Implementierung des Transaktionskonzeptes aufweisen sowie einfachere Datenstrukturen verwalten müssen und daher kein Ersatz für mobile DBMS sind, lassen sich doch einige Konzepte von mobilen Dateisystemen in Modellen und Produkten für mobile DBMS wiederverwenden. Die Vorstellung von Coda, dem wichtigsten Vertreter mobiler Dateisysteme, orientiert sich an dem entsprechenden Abschnitt in der Arbeit von [Zuk98].

6.2 IBM DB2 DataPropagator

6.2.1 Architektur

Ursprünglich als Komponente für die Replikation in verteilten Datenbankszenarien konzipiert, läßt sich der *IBM DB2 DataPropagator* auch für die Replikation in sogenannten satellitenartigen Umgebungen einsetzen. Diese besteht aus mehreren Satellitendatenbanken, die jeweils als mobiles oder stationäres System realisierbar sind. Der Einsatz für mobile Szenarien ist durch die nur gelegentliche Verbindung der Satelliten zum Kommunikationsnetzwerk gekennzeichnet.

Die Datenreplikation erfolgt über ein *Publish & Subscribe*-Verfahren. Hierzu können Benutzertabellen als Replikationsquelle deklariert (*publish*) und Inhalte dieser Tabellen von anderen Satelliten als Replikatdaten für lokale Zieltabellen angefordert werden (*subscribe*). Weiterhin werden änderbare, sogenannte Replikattabellen, und nicht änderbare Zieltabellen, im einfachsten Fall eine Kopie der Quelltabelle, unterschieden. Aufgabe der Replikationskomponente ist es nun, an der Replikationsquelle bzw. Replikattabelle vorgenommene Änderungen mit Hilfe des Programms *Capture* festzustellen und diese auf allen Replikationszielen bzw. der Replikationsquelle mittels *Apply* nachzuvollziehen. Die Instanzen der beiden Programme können in der Regel auf verschiedenen Satelliten gleichzeitig und unabhängig voneinander arbeiten.

6.2.2 Datenhaltung und -zugriff

Anwendungsprogramme und Transaktionen, die auf den Satelliten ausgeführt werden, unterliegen den gleichen Transaktionsmechanismen wie in der Standarddatenbank *DB2 Universal Database* [IBM99b] auch. Für Transaktionen mit Änderungen auf Replikattabellen kann speziell die Dauerhaftigkeit der ACID-Eigenschaften bis zur ersten Datensynchronisation nach Transaktionsende nicht garantiert werden. Transaktionen mit Zugriffen auf replizierte Daten werden protokolliert. Diese Protokolldaten werden im Synchronisationsvorgang von *Capture* ausgelesen und in die sogenannte *Änderungstabelle* geschrieben. Aus dieser Tabelle liest *Apply* und führt die entsprechenden Änderungen an Quell- und Zieltabellen durch. Dies geschieht dezentral und asynchron. Eine solche Synchronisation der Daten kann nur für geänderte Daten (*differential refresh*) oder für alle Daten einer Tabelle (*full refresh*) erfolgen.

Eine Konflikterkennung und -behandlung übernimmt *Apply*. Konflikte werden von *Apply* dann erkannt, wenn bei der Einbringung einer Operation das durch den Schlüssel des *before image* identifizierte Tupel in der Quelltabelle nicht die selben Attributwerte wie das *before image* in der Zieltabelle aufweist, d.h. wenn seit der letzten Synchronisation des Satelliten

eine Änderung am entsprechenden Tupel in der Originaltabelle vorgenommen wurde. Der Zustand der Replikationsquelle wird hierbei immer als konsistent angenommen und dient als Primärkopie in dem Master-Replikationsschema. Die Standard-Konfliktauflösung führt dementsprechend ein Rollback der verursachenden Transaktion T auf dem Quellsystem durch und beläßt die entsprechende Transaktion in der sogenannten UOW-Tabelle (*unit of work*). Damit würde beim nächsten *Apply*-Zyklus versucht werden, die Transaktion erneut einzubringen. Eine endgültige Lösung wird dem Nutzer über einen manuellen Eingriff überlassen, beispielsweise indem er auf dem Satellitensystem eine kompensierende Transaktion für die konflikterzeugende Transaktion findet bzw. durch eine benutzerdefinierte Behandlung zurückgewiesener Transaktionen, die automatisch nach jeder Beendigung eines *Apply*-Zyklus startet. Allerdings werden Abhängigkeiten nachfolgender Transaktion von einer zurückgewiesenen nicht automatisch erkannt.

Damit wird hier ein ähnlicher Ansatz wie im Two-Tier-Modell (Abschnitt 4.2) verfolgt. Auf dem Satelliten ist ein lokaler Transaktionsabschluß möglich, für die enthaltenen Änderungen kann allerdings keine Dauerhaftigkeit gewährleistet werden. Das Synchronisieren wird effektiv durch ein Wiederausführen auf aktuellen Daten realisiert mit den bekannten im Abschnitt 4.2 aufgezeigten Problemen und Konflikten. Die simple Konfliktlösung vom *IBM DB2 DataPropagator* zeigt die Schwierigkeiten in diesem Zusammenhang.

6.3 Sybase Adaptive Server Anywhere 7.0.0 und SQL Remote

6.3.1 Architektur

Neben dem Einsatz auf stationären Systemen ist dieses Datenbanksystem auch für den Einsatz im mobilen Umfeld ausgelegt. Dabei werden neben Systemplattformen wie Windows CE für PDAs auch andere zahlreiche Betriebssysteme für unterschiedliche Geräteklassen unterstützt, wie beispielsweise Windows 98, Windows NT, Windows 2000 und verschiedenen Unix-Varianten wie Linux, Sun Solaris und IBM AIX.

Grundlage in der verwendeten Client/Server-Architektur ist eine sogenannte *konsolidierte Datenbank*, welche zentral auf dem Server die Originale aller zu replizierenden Daten enthält. Die auf den Clients laufende Datenbank (*remote database*) enthält dann Replikate der konsolidierten Datenbank. Die Replikation erfolgt wie im zuvor beschriebenen *IBM DB2 DataPropagator* über ein *Publish & Subscribe*-Verfahren, wobei für jeden Client Publikationen auf dem Server erstellt werden, die dann von den Clients mit Hilfe einer SUBSCRIBE BY-Klausel abonniert werden können. Nach Definition der Publikationen hat der Client nahezu keinerlei Einflußmöglichkeiten mehr auf den Umfang der zu replizierenden Daten. Die Kommunikation und Synchronisation der mobilen Clients mit dem zentralen Server findet über die Komponente *SQL Remote* [Syb00b] statt. Diese erlaubt einen asynchronen, nachrichtenbasierten Abgleich von Änderungen in beide Richtungen. Dazu werden Änderungstransaktionen, die publizierte Daten auf dem Server oder abonnierte Daten auf dem Client betreffen, protokolliert und die Operationen als Nachrichten, z.B. per E-Mail, an die sogenannte Inbox der Gegenstelle verschickt. Dort werden eingetroffene Änderungen als Transaktionen auf dem aktuellen Datenbestand erneut ausgeführt.

6.3.2 Datenhaltung und -zugriff

Operationen können im *Sybase Adaptive Server Anywhere 7.0.0* in zwei verschiedenen Varianten abgearbeitet werden. Zum einen erfolgt bei der *unverketteten* Abarbeitung nach jeder Anweisung ein Abschluß durch implizites COMMIT. Dies ist die Standardeinstellung. Besteht eine Transaktion aus mehreren aufeinanderfolgenden Operationen, müssen diese mit BEGIN TRANSACTION und explizitem COMMIT oder ROLLBACK abgeschlossen werden. Im Gegensatz dazu werden während der *verketteten* Abarbeitung von SQL-Anweisungen Transaktionen implizit begonnen und müssen explizit abgeschlossen werden, was der traditionellen Arbeit mit Transaktionen entspricht.

Im produkteigenen SQL-Dialekt werden geschachtelte Transaktionen unterstützt, die Dauerhaftigkeit einer geschachtelten Transaktion wird durch COMMIT des äußeren explizit markierten Blockes erreicht. Für Client-Transaktionen werden bis auf die globale Dauerhaftigkeit auf der konsolidierten Datenbank alle ACID-Eigenschaften zugesichert. Die Dauerhaftigkeit kann durch Konfliktmöglichkeiten beim Synchronisieren nicht garantiert werden, da Client-Transaktionen in diesem Fall (vorerst) zurückgewiesen werden. Auf der konsolidierten Datenbank zurückgewiesene Transaktionen müssen durch Kompensationstransaktionen auf dem Client, auf dem sie durchgeführt wurden, rückgängig gemacht werden. Dies erfordert entsprechend hohen Aufwand bei der Bereitstellung der dazu nötigen Kenntnisse.

Die bei der Replikation auftretenden Konfliktsituationen, Synchronisationskonflikte und -fehler, werden vom *Message Agent*, einer Komponente von *SQL Remote*, serverseitig erkannt. Diese Konflikte sind wie folgt charakterisiert:

- Synchronisationskonflikte stellen behebbare Fehlersituationen dar, hierunter fallen ausschließlich Änderungskonflikte. Ein Konflikt ist erkannt, wenn die in einer VERIFY-Klausel angegebenen Attributwerte beim Vergleich mit den in der konsolidierten Datenbank vorhandenen Werten nicht übereinstimmen. Die Identifzierung eines Tupels erfolgt hierbei anhand des Primärschlüssels. Die automatische Auflösung findet durch Ausführung aller benutzerdefinierten, sogenannter RESOLVE UPDATE-Trigger statt, die bezüglich der konfliktbehafteten Spalten definiert sind. Dies kann eine Verletzung der Atomaritätseigenschaft zur Folge haben, wenn Teile einer konflikterzeugenden Transaktion durch diese Trigger-Behandlung mit einem anderen Ergebnis abgeschlossen werden.

- Synchronisationsfehler besitzen keine aktiven Behandlungsmechanismen. Diese Art von Konflikten tritt bei Lösch- und Eindeutigkeitskonflikten (Abschnitt 2) sowie bei Verletzungen der referentiellen Integrität auf. Der Fehler wird nur erkannt und die entsprechende Transaktion von der konsolidierten Datenbank zurückgewiesen. Der *Message Agent* muß dies protokollieren und kann optional eine Benachrichtigung des Nutzers beispielsweise per E-Mail über eine *stored procedure* durchführen, der für die weitere Konfliktbehandlung zuständig ist.

Das potentielle Auftreten von Synchronisationsfehlern sollte schon durch eine geeignete Gestaltung der Replikationsbeziehungen vermieden werden. Eindeutigkeitskonflikte, die beim Einfügen von Tupeln in replizierte Tabellen auftreten können, lassen sich beispielsweise durch den Einsatz von *primary-key pools* [Syb00a] verhindern.

6.4 Coda

Das in [SKK+90] vorgestellte und an der Carnegie Mellon University (CMU) entwickelte Softwaresystem stellt erstmalig eine Erweiterung eines Dateisystem zur expliziten Unterstützung mobiler Nutzer dar. Ziel von Coda ist es, ein beliebiges Dateisystem auch im unverbundenen Zustand von Clients nutzbar zu machen, d.h. dem Nutzer einen transparenten Dateizugriff zu ermöglichen. Um diese Transparenz zu gewährleisten, werden Teile des Dateisystem lokal auf dem Client repliziert. Die Verwaltung dieser Replikate übernimmt eine in den Cache-Manager integrierte optimistische Replikationskomponente mit folgenden Aufgaben:

- Replizieren (*Hoarding*) von Dateien vor der Verbindungstrennung zwischen Client und Server. Die Auswahl der zu replizierenden Daten wird aufgrund von Zugriffsheuristiken und benutzerdefinierten Listen getroffen. Granulate sind hierbei Verzeichnisse und Dateien.

- Nachbildung einer Verbindung bei Zugriff auf replizierte Dateien im unverbundenen Zustand. Hierbei werden Modifikation protokolliert.

- Reintegration der replizierten und veränderten Daten bei Neuverbindung unter Verwendung der Protokolldaten. Dabei führt jeder Server die Änderungen innerhalb einer Synchronisation als eine Aktion aus. Da das Transaktionskonzept nicht verwendet wird und Atomarität außerhalb eines einzelnen Systemaufrufs nicht existiert, sind nur *write-write*-Konflikte (siehe Abschnitt 2) von Interesse. Jedes Datenobjekt besitzt ein zusätzliches Feld (*storeid*), welches eindeutig die letzte Änderung vermerkt. Ein Konflikt ist erkannt, wenn der Vergleich auf Gleichheit dieses Feldes zwischen dem Serverobjekt und den Protokolldaten fehlschlägt [KS91]. Die Konfliktlösung hängt dann von der Art des replizierten Objektes ab (Datei, Verzeichnis, etc.) und besteht im allgemeinen aus dem Zurückweisen der lokalen Änderung.

Einzig die Manipulation der auf dem Server gehaltenen Meta-Daten, bestehend aus Informationen über replizierte Verzeichnisse, Inhalte symbolischer Verweise, Statusinformationen zu replizierten Daten und Protokolldaten zur Reintegration, unterliegen den vollständigen Transaktionseigenschaften bezüglich ACID und Wiederherstellung bei Systemausfall durch Protokollierung.

Das beobachtete Zugriffsverhalten der Nutzer im Coda-Projekt [SKM+93] zeigte, daß replizierte Daten nur selten gemeinsam genutzt werden und daher bei der Reintegration kaum Konflikte entstehen. Trotzdem wurden in weiteren Versionen von Coda verbesserte Konfliktauflösungen mittels der aus Abschnitt 4.6 bekannten anwendungsspezifischen Konfliktauflösung (ASR) implementiert [KS95]. Weiterhin wurde mit den „Isolation-Only Transactions“ (Abschnitt 4.6) ein eingeschränktes Transaktionsmodell eingeführt. Schließlich wurde Coda um einen dritten, den schwach verbundenen Zustand, erweitert [MES95]. Dieser brachte neue Möglichkeiten und Probleme beispielsweise für das Replizieren mit sich, da nun zusätzlich ein Replizieren von Dateien im Bedarfsfall über eine schwache, aber schmalbändrige Verbindung möglich war und ein Abwägen zwischen beiden Varianten des Replizierens erfordert, wenn auf größere Datenmengen zugegriffen werden soll.

7 Zusammenfassung und Ausblick

Die Arbeit hat einen Überblick über die wichtigsten Konzepte und Probleme der Transaktionsverarbeitung und Reintegration von Änderungen auf replizierten Daten in mobilen Datenbanksystemen gegeben. Unter den derzeit existierenden Modellen für eine Lösung wurden die wichtigsten vorgestellt und bewertet. Dabei lag der Schwerpunkt auf der transaktionsorientierten Reintegration und optimistischen Replikation. Wie sich gezeigt hat, gibt es aufgrund der Anforderung an die Art der Verbindung zwischen Client und Server während der Arbeit auf dem mobilen Client zwei Kategorien, Modelle mit Unterstützung der Unverbundenheit und Modelle mit Einschränkung auf die schwache Verbundenheit. Allerdings verfolgen die Modelle der beiden Kategorien jeweils unterschiedliche Zielstellungen wie im Abschnitt 4 aufgezeigt wurde. Innerhalb jeder Kategorie bauen die meisten Modelle auf einem Grundprinzip auf. Für die Modelle der ersten Kategorie ist dies das Prinzip der Zweistufigkeit, d.h. die Unterscheidung in lokalen und globalen Transaktionsabschluß, Modelle der zweiten Kategorie legen das Prinzip der Transaktionsschachtelung zugrunde.

Allerdings bringt die Frage nach der Implementierung eines Modells in der Praxis unterschiedlich schwerwiegende Probleme hervor. Dabei spielen meist die Beschaffung von zusätzlich benötigten Informationen für die Konfliktbehandlung (z.B. *read-set*) und einschränkende Forderungen an Daten und Transaktionen (z.B. kommutative Operationen) eine große Rolle. Dies äußert sich schließlich darin, daß nur zwei der vorgestellten sieben Modelle wenigstens als Projekt realisiert wurden und der Exkurs in die vorhandenen DBMS-Produkte für mobile Anwendungen zeigte, daß auch hier noch mitunter große Einschränkungen in der Benutzung akzeptiert werden müssen, vorallem im Bereich der Konfliktauflösung und bei den zur Verfügung stehenden Operationen auf replizierten Daten.

Aufbauend auf dieser Arbeit könnten weitere Modelle, neue Produktversionen sowie akutelle Forschungsergebnisse betrachtet und miteinander verglichen werden. Vorallem im Bereich der Produktlösungen könnte hierbei der Schwerpunkt auf vorhandenen Defiziten liegen, die einen umfassenden praktischen Einsatz bisher verhindern. Desweiteren sollte die Frage nach den prinzipiellen Grenzen und Möglichkeiten geklärt werden, die eine hier betrachtete mobile Umgebung mit sich bringt, d.h. es sind eine minimale Menge von Forderungen zu bestimmen, die für den praktischen Einsatz in einem Szenario unabdingbar sind und die Möglichkeiten abzugrenzen, die man in einem Szenario prinzipiell hat. Eine weitere interessante Thematik könnte die Kombination der einzelnen Komponenten verschiedener Modelle für eine optimale Lösung zu einem konkreten Szenario sein. Beispielsweise kann das Two-Tier-Modell als Grundlage für die Replikation dienen und die Konfliktauflösung basierend auf der Snapshot-Suche des Multiversionsmodells durchgeführt werden. Schließlich ließen sich auch einige der vorgestellten Modelle näher betrachten und erweitern bzw. neue Modelle entwickeln, basierend auf den Betrachtungen in dieser Arbeit.

Literatur

[ALO00] A. Adya, B. Liskov und P. O'Neil. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering*, Seiten 67–78, San Diego, CA, USA, März 2000.

[ANS92] ANSI. X3.135-1992. In *American National Standard for Information Systems - Database Language - SQL*, November 1992.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil und P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seiten 1–10, San Jose, CA, Mai 1995.

[BGMS92] Y. Breitbart, H. Garcia-Molina und A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal: Very Large Data Bases*, 1(2):181–293, 1992.

[Chr93] P. K. Chrysanthis. Transaction Processing in Mobile Computing Environment. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, Seiten 77–83, Princeton, New Jersey, Oktober 1993.

[CR94] P. K. Chrysanthis und K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[Dad96] P. Dadam. *Verteilte Datenbanken und Client/Server-Systeme*. Springer-Verlag, Berlin, Heidelberg, 1996.

[DHB97] M. H. Dunham, A. Helal und S. Balakrishnan. A Mobile Transaction Model that Captures both the Data and Movement Behavior. *ACM Baltzer Journal on Mobile Networks and Applications (MONET)*, 2(2), Oktober 1997.

[DPS+94a] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer und B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, Seiten 140–140, Austin, Texas, September 1994.

[DPS+94b] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer und B. Welch. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems & Applications*, Seiten 2–7, Santa Cruz, CA, USA, Dezember 1994.

[DPS+95] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer und C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, Seiten 172–183, Copper Mountain Resort, Colorado, USA, Dezember 1995.

[EMP+97] W. K. Edwards, E. D. Mynatt, K. Petersen, M. Spreitzer, D. B. Terry und M. Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proceedings of the 10th Annual ACM Symposium*

on User Interface Software and Technology, Seiten 119–128, Banff, Alberta, Canada, Oktober 1997.

[Fan00] T. Fanghänel. Vergleich und Bewertung kommerzieller mobiler Datenbanksysteme. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, September 2000.

[GHOS96] J. Gray, P. Helland, P. O'Neil und D. Shasha. The Dangers of Replication and a Solution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seiten 173–182, Montreal, Quebec, Canada, Juni 1996.

[Gol92] Richard A. Golding. *Weak-Consistency Group Communication and Membership.* Dissertation, University of California at Santa Cruz, 1992.

[Gol00] C. Gollmick. Anwendungsklassen und Architektur Mobiler Datenbanksysteme. In *Tagungsband des 12. Workshop „Grundlagen von Datenbanken", Plön*, Bericht Nr. 2005, Seiten 36–40, Christian-Albrechts-Universität Kiel, Juni 2000.

[GR93] J. Gray und A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[IBM99a] IBM, Corp. *IBM DB2 – Replication Guide and Reference*, Juni 1999.

[IBM99b] IBM, Corp. *IBM DB2 Universal Database – Administration Guide: Design and Implementation (Version 6)*, 1999.

[KS91] J. J. Kistler und M. Satyanarayanan. Disconnected Operation in the Coda File System. *13th ACM Symposium on Operating Systems Principles*, 25(5):213–225, 1991.

[KS95] P. Kumar und M. Satyanarayanan. Flexible and Safe Resolution of File Conflicts. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, Seiten 95–106, New Orleans, Louisiana, USA, Januar 1995.

[LP83] R. Lorie und W. Plouffe. Complex Objects and their Use in Design Transactions. In *Proceedings of the ACM SIGMOD Database Week – Enginieering Design Applications*, Seiten 115–121, Silver Spring, MD, Mai 1983.

[LS94] Q. Lu und M. Satyanaranyanan. Isolation-Only Transactions for Mobile Computing. *Operating Systems Review*, 28(2), 1994.

[LS95] Q. Lu und M. Satyanaranyanan. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Isaln, WA, USA, Mai 1995.

[MES95] L. B. Mummert, M. Ebling und M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, Seiten 143–155, Copper Mountain Resort, Colorado, USA, Dezember 1995.

[PB95] E. Pitoura und B. Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, Seiten 404–413, British Columbia, Canada, Mai 1995.

[PB00] S. Phatak und B. Badrinath. Transaction-centric Reconciliation in Disconnected Client Server Databases. Technical report, Department of Computer Science, Rutgers University, New Jersey, 2000.

[PKH88] C. Pu, G. Kaiser und N. Hutchinson. Split-Transactions for Open-Ended Activities. In *Proceedings of the 14th International Conference on Very Large Databases*, Seiten 26–37, Los Angeles, California, USA, 1988. Morgan Kaufmann.

[PST+97] K. Petersen, M. Spreitzer, D. B. Terry, M. Theimer und A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, Seiten 288–301, Saint Malo, France, Oktober 1997.

[SKK+90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel und D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SKM+93] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar und Q. Lu. Experience with Disconnected Operation in a Mobile Computing Environment. In *Proceedings of the USENIX Symposium on Mobile & Location-Independent Computing*, Seiten 11–28, August 1993.

[Syb00a] Sybase, Inc. *Adaptive Server Anywhere User's Guide (Version 7.0.0)*, März 2000.

[Syb00b] Sybase, Inc. *Replication and Synchronization Guide (Version 7.0.0)*, März 2000.

[WC95] G. D. Walborn und P. K. Chrysanthis. Supporting Semantics-Based Transaction Processing in Mobile Database Applications. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, Seiten 31–40, Bad Neuenahr, Germany, September 1995.

[WS92] G. Weikum und H.-J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models for Advanced Applications*, Seiten 515–553. Morgan Kaufmann, 1992.

[Zuk98] O. Zukunft. *Integration mobiler und aktiver Datenbankmechanismen als Basis für die ortsungebundene Vorgangsbearbeitung.* Logos Verlag Berlin, 1998.

Abbildungsverzeichnis

1 Mobile Umgebung für schwach verbundene Clients 11
2 Eine Bayou-Schreiboperation am Beispiel des Gruppenkalenders 16
3 Beispiel für einen Datenbankzustand und entsprechende Snapshots 21
4 Standardkosten- und Standardkonfliktauflösungsfunktion 23
5 Multiversions-Abgleichalgorithmus 24
6 Zustandübergangsdiagramm für Isolation-Only Transactions 32
7 Eine mögliche Konfiguration für Kangaroo Transactions 39

Tabellenverzeichnis

1 Isolationslevel und Phänomene 9
2 Übersicht der Transaktionsmodelle und Produkte 45