Synchronisationskonflikte beim mobilen Datenbankzugriff: Vermeidung, Erkennung und Behandlung

– Diplomarbeit –

zur Erlangung des akademischen Grades Diplom-Informatiker

eingereicht von

Matthias Liebisch

matthi@smigel.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

Februar 2003

Kurzfassung

Seit einigen Jahren zeichnet sich sowohl in der Arbeitswelt als auch im Freizeitbereich eine Veränderung im Umgang und der Arbeitsweise mit Informationssystemen ab, von einer bisher gewohnten starren und ortsgebundenen Form hin zu einer immer flexibler werdenden Art und Weise. Dabei wird von den Anwendern vorallem eine Kommunikationsmöglichkeit und der Zugriff auf beliebige Informationen an jedem Ort und zu jeder Zeit gewünscht, was beispielsweise eine Erweiterung des Angebots bestehender Dienstleistungsbranchen erfordert. Eine große Rolle spielt dabei die mobile Bereitstellung von relevanten Daten, die üblicherweise zentralisiert in einem Datenbanksystem (DBS) vorgehaltenen werden. Dafür müssen neue Konzepte in die Strukturen aktueller DBS integriert bzw. deren Funktionalitäten erweitert werden.

Ausgangslage für das hier zugrundegelegte Modell ist ein Client-Server-System, welches auch mobile, zeitweise unverbundene Clients, unterstützen soll. Für deren Arbeit ohne Verbindung zum Server ist einerseits die Replikation von Daten auf ein lokales DBS notwendig, andererseits müssen die unverbunden durchgeführten Änderungen mit dem Server synchronisiert werden. Arbeitet mehr als ein replizierender Client auf denselben Daten, kann es dabei zu Konflikten kommen. Diese müssen erkannt und gelöst bzw. sollten von Beginn an vermieden werden. Die Untersuchung der dafür einzusetzenden Konzepte ist Inhalt dieser Arbeit.

Neben der Klassifikation und Bewertung vorgestellter Synchronisationsverfahren gibt die Arbeit auch einen Überblick zur Realisierung in Produkten sowie neuen Ansätzen in Forschungsprojekten. Ein Schwerpunkt dieser Arbeit ist die Beschreibung der Integration von Konfliktvermeidungs- und Konfliktauflösungsmöglichkeiten in das Konzept der nutzerdefinierten Replikation.

Danksagung

An dieser Stelle möchte ich mich bei allen Beteiligten bedanken, die direkt oder indirekt zum Entstehen dieser Diplomarbeit beigetragen haben.

Für die hervorragende intensive Betreuung meiner Diplomarbeit möchte Christoph Gollmick und Prof. Dr. Klaus Küspert herzlichen Dank aussprechen. Insbesondere haben unzählige fachlich konstruktive Gespräche und Diskussionen mit Christoph Gollmick zu einer fortwährenden Entwicklung der Arbeit geführt. Weiterhin danke ich Prof. Dr. Klaus Küspert für sein persönliches Engagement, das es mir ermöglichte, an mehreren interessanten und erkenntnisreichen Veranstaltungen teilnehmen zu können.

Besonderer Dank für seelischen und moralischen Beistand geht an meine Familie und Freunde, vorallem an meine Eltern Gitta Liebisch und Karl-Heinz Liebisch, deren finanzielle Unterstützung mir das Studium erst ermöglicht hat, sowie an meine liebevolle Freundin Diana Saluski, die mir gerade in schwierigen Zeiten immer wieder motivierend zur Seite stand.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Einführung

Der Wunsch nach immer größerer Mobilität und Flexibilität hat zur Folge, daß der Nutzer möglichst unabhängig von seinem Aufenthaltsort über einen mobilen Client auf Daten eines Datenbankmanagementsystems (DBMS) zugreifen möchte, welches üblicherweise zentral auf einem stationären Server betrieben wird. Ist eine permanente Kommunikationsverbindung über ein Funkmedium zwischen mobilem Client und Server, beispielsweise aus Kosten-, Sicherheits- oder auch Abdeckungsgründen, nicht möglich, so bietet das hier zugrundegelegte Modell die Möglichkeit der zeitweise *unverbundenen Arbeit*. Hierfür müssen die lokal verwendeten Daten auf dem mobilen Gerät, wie beispielsweise einem PDA oder Laptop, *repliziert* werden. Dabei sind unter anderem auch Beschränkungen durch technische Spezifikationen der mobilen Geräte, wie beispielsweise geringe Speicherkapazitäten, zu beachten. Nach der Replikation können auf den lokalen Daten unverbundene Änderungen durchgeführt werden. Anschließend müssen diese lokalen Änderungen in den Datenbestand der Server-Datenbank reintegriert und konsolidiert werden. Daß es dadurch zu *Konflikten* kommen kann, zeigt folgendes Beispielszenario.

Betrachten wir zwei Vertreter, die sich aus einer gemeinsamen Datenbank gleiche Daten zu Produkten und deren Lagerbestand replizieren. Diese sollen im Laufe des Tages an verschiedene Kunden verkauft und Änderungen der Lagerbestandsdaten mobil, d.h. ohne Verbindung zur Datenbank auf dem Server, durchgeführt werden. Am Tagesende entsteht beim Synchronisieren der beiden Clients und ihrer Änderungen ein Konflikt, wenn in der Summe Produktmengen verkauft wurden, die nicht im Lager existieren, da sich jeder der beiden Vertreter im exklusiven Zugriffsrecht auf die replizierten Daten glaubte. Eine sofortige Auslieferung des betreffenden Produktes an alle Kunden ist dann nicht mehr möglich. Um solche Konflikte zu vermeiden, könnte jedem Verkäufer ein gewisser Anteil der Gesamtproduktmenge zur Verfügung gestellt werden, die er konfliktfrei während der unverbundenen Arbeit verkaufen kann.

Die vorliegende Arbeit beschäftigt sich mit der Untersuchung von Synchronisationskonflikten, verursacht durch unverbunden ausgeführte Änderungen auf replizierten Daten. Dazu werden Methoden untersucht, die zur Vermeidung, aber auch zur Erkennung und Auflösung solcher Konflikte geeignet sind. Desweiteren findet eine Beschreibung der Integration dieser Methoden zur Konfliktbehandlung in das Konzept der *nutzerdefinierten Replikation* [Gol02, Gol03] über eine an SQL angelehnte deskriptive Schnittstelle statt.

1.2 Aufbau der Arbeit

Die Arbeit ist wie folgt aufgebaut. Im anschließenden Kapitel 2 werden Annahmen und Voraussetzungen zu dem hier zugrundegelegten Modell des mobilen Datenbankzugriffs erläutert. Im darauffolgenden Kapitel 3 werden klassische Sperrverfahren und mobile Anpassungen zur Vermeidung von Konflikten auf ihre Einsatzmöglichkeiten untersucht. Kann oder will man Konflikte nicht vermeiden, so müssen alternativ optimistische Synchronisationsverfahren eingesetzt werden, deren Konzepte zur Erkennung und Behandlung von Konflikten im Kapitel 4 beschrieben sind. Einen Überblick, wie Produkte und andere Forschungsansätze den Anforderungen des mobilen Arbeitens, insbesondere der Konfliktbehandlung, gerecht werden, gibt Kapitel 5. In Kapitel 6 stellen wir einen deskriptiven Ansatz zur Integration von Strategien zur Konfliktbehandlung in das Konzept der nutzerdefinierten Replikation vor. Eine Bewertung und Kostenbetrachtung aller aufgezeigten Methoden und Verfahren zur Synchronisation mobiler Clients ist in Kapitel 7 zu finden. Abschließend wird im Kapitel 8 eine Zusammenfassung der Arbeit sowie ein Ausblick auf weiterführende Themen gegeben.

Kapitel 2

Grundlagen und Voraussetzungen

Das hier beschriebene Modell des in Abbildung 2.1 dargestellten Szenarios der mobilen Arbeit setzt eine *Client-Server*-Architektur [Dad96] voraus, in der Clients neben der klassischen festen Verbindung zum Server zeitweise unverbunden arbeiten können. Ein dritter möglicher, sogenannter schwacher, permanenter Verbindungszustand über ein Funkmedium wird aus den einleitend im Kapitel 1 dargestellten Kosten- oder Abdeckungsgründen hier nicht betrachtet. Desweiteren wird auf dem Server ein relationales DBMS zugrundegelegt und ein lokales DBS auf dem Client benötigt.

2.1 Arbeitsmodell

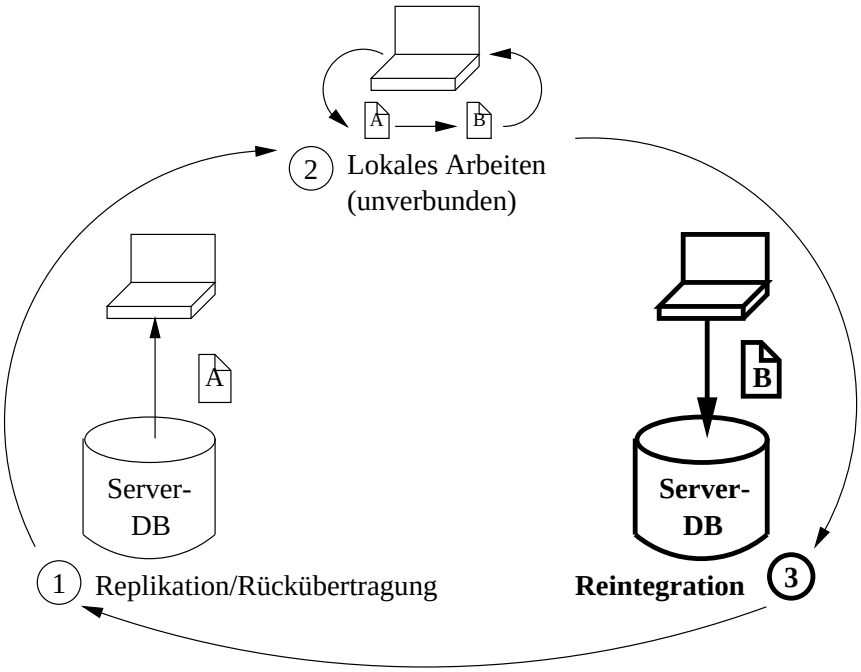


Abbildung 2.1: Mobiles Szenario aus der Sicht des Clients

Will ein mobiler Client zeitweise unverbunden arbeiten, sind zuvor die benötigten Daten zu kopieren. Dieser in Abbildung 2.1 mit *Replikation* bezeichnete Schritt erfolgt immer vom Server (Datenquelle) zum Client (Ziel), wobei nur transaktionskonsistente Daten, d.h. Ergebnisse von abgeschlossenen Transaktionen, repliziert und in dem lokalen DBS des Client abgelegt werden. Folgende Formen der Replikation können unterschieden werden:

- *Konfliktvermeidende Replikation*
  Bei der konfliktvermeidenden bzw. pessimistischen Replikation werden die Daten so repliziert, daß parallele Änderungen auf gleichen Daten nicht möglich sind, d.h. ein Client erhält entweder das exklusive Recht auf den replizierten Daten, üblicherweise realisiert durch Sperrverfahren, oder es wird die Semantik der Daten für Konzepte wie das im Abschnitt 3.2.1 beschriebene `KEY-POOL`-Verfahren genutzt. Diese im Kapitel 3 beschriebenen Verfahren gestatten eine einfache Konsolidierung der Daten ohne Konflikterkennung. Allerdings haben die replizierten Daten für den Zeitraum der Unverbundenheit des entsprechenden Client für andere Clients eine schlechte Verfügbarkeit.

- *Konfliktbehaftete Replikation*
  Werden die Daten durch die konfliktbehaftete bzw. optimistische Replikation (siehe Kapitel 4) ohne weitere Kontrolle auf dem Client repliziert, gibt es keine durch Sperren erzeugte Verfügbarkeitseinbußen, da gleiche Daten auch mehrfach auf verschiedenen Clients repliziert werden können. Allerdings besteht dann die Gefahr von *Konflikten* durch inkompatible Änderungen verschiedener Clients auf denselben Daten. Diese müssen während der dann sehr aufwendigen Reintegration auf dem Server erkannt und gelöst werden.

Sind alle erforderlichen Daten repliziert, wird die Verbindung zwischen Client und Server getrennt und die Phase der lokalen Arbeit des Client beginnt. Der Zeitraum bis zur erneuten Verbindungsaufnahme mit dem Server kann als eine sogenannte *langlaufende Transaktion* [KS91] betrachtet werden.

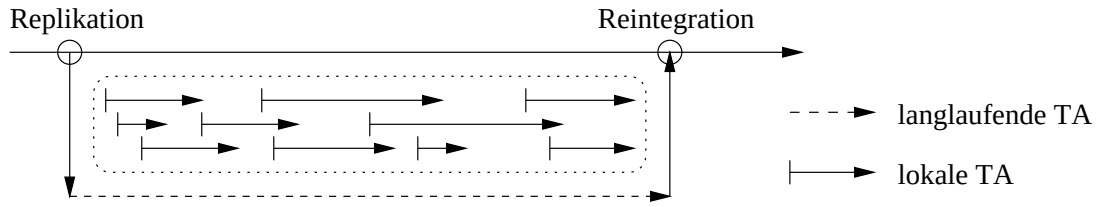


Abbildung 2.2: Langlaufende Transaktionen und lokale Transaktionen

Das ursprünglich als Lösung für systemübergreifende Anwendungen, wie beispielsweise komplexe Buchungssysteme, entwickelte Konzept erlaubt die Zerlegung einer langlaufenden Transaktion in einzelne *verschachtelte Transaktionen*. Nach erfolgreichem Abschluß der letzten geschachtelten Transaktion schließt die langlaufende Transaktion ebenfalls erfolgreich ab, eine Reintegration ist hier aufgrund von Sperranwendung nicht notwendig. Wurden verschachtelte Transaktionen abgebrochen, kann die langlaufende Transaktion dennoch unabhängig abgeschlossen oder ebenfalls abgebrochen werden. Die eigentliche unverbundene Arbeit eines mobilen Client besteht dagegen aus sogenannten *lokalen Transaktionen*, wie in Abbildung 2.2 dargestellt. Diese arbeiten auf den replizierten Daten und

können, aufgrund der fehlenden Verbindung zum Server, nur vorläufig abgeschlossen werden. Insbesondere ist bei optimistischer Replikation die Dauerhaftigkeit der Transaktionen inhärenten ACID-Eigenschaften [GR93] gefährdet.

Im letzten Schritt der *Synchronisation* werden die aktuellen Daten von Client und Server wechselseitig abgeglichen. Die *Reintegration* ist dabei für die Konsolidierung der lokalen Änderungen mobiler Clients im Server-DBS verantwortlich, während anschließend in der *Rückübertragung* aktuelle Daten zum Client propagiert werden. Eine Erkennung und Behandlung von Konflikten findet nur auf dem Server während der Reintegrationsphase statt, die abhängig von der Wahl der oben beschriebenen Replikationsform mehr oder weniger aufwendig ist. Die Synchronisation verläuft einerseits immer nur zwischen zwei Instanzen, wobei eine Instanz ein Client oder der Server sein kann,[1] andererseits beschränken wir uns hier auf den hierarchischen Fall, d.h. einer Synchronisation zwischen genau einem Client und dem Server.[2] Um zu gewährleisten, daß alle lokalen Transaktionen eines Client den gleichen aktuellen Zustand des Server zum Synchronisationszeitpunkt sehen, kann beispielsweise die Reintegration der mobilen Clients serialisiert werden. Dabei erhalten alle lokalen Transaktionen eines Client dessen Reintegrationszeitpunkt als Commit-Zeitstempel, allerdings bleibt durch Auswertung des Client-Log die Reihenfolge der lokalen Transaktionen untereinander erhalten. Um die replizierten und geänderten Daten verschiedener Clients sowie die Server-Daten bei der Reintegration auswerten und vergleichen zu können, werden folgende Bezeichnungen für eine Datenmenge D mit Bezug auf einen festen Synchronisationszeitpunkt eingeführt:

- *Before Image (BI_D^C/BI_D^S):*
  Inhalt der betrachteten Datenmenge D auf dem Client C bzw. Server S zum Zeitpunkt der Replikation bzw. nach Abschluß der vorangegangenen Synchronisation
- *Änderung des Before Image ($\Delta BI_D^C/\Delta BI_D^S$):*
  Menge der im BI_D geänderten Daten auf dem Client C bzw. Server S zum Zeitpunkt der Reintegration
- *After Image (AI_D^C):*
  aktueller Inhalt der betrachteten Datenmenge D auf dem Client C zum Zeitpunkt der Synchronisationsphase
- *Current Image (CI_D^S):*
  aktueller Inhalt der betrachteten Datenmenge D auf dem Server S zum Zeitpunkt der Synchronisationsphase
- *Read Set (RS_T^C/RS^C):*
  Menge aller gelesenen Daten einer Transaktion des Client C bzw. Menge aller gelesenen Daten des Client C zwischen letztem und aktuellem Synchronisationszeitpunkt
- *Write Set (WS_T^C/WS^C):*
  Menge aller geschriebenen Daten einer Transaktion des Client C bzw. Menge aller geschriebenen Daten des Client C zwischen letztem und aktuellem Synchronisationszeitpunkt

[1] Denkbar wäre auch die gleichzeitige Synchronisation von mehr als zwei Instanzen, beispielsweise von zwei Clients mit einem Server, allerdings nur unter enormen Komplexitätszuwachs.

[2] Die Synchronisation zwischen zwei Clients ist unter den hier getroffenen Voraussetzungen nicht möglich, da wichtige Informationen bezüglich der replizierten Daten nur auf dem Server gehalten werden.

Für den Prozeß der Reintegration von lokal geänderten Daten eines Client gibt es prinzipiell zwei Möglichkeiten:

- *Datenorientiert*
  Bei der datenorientierten Reintegration steht die Auswertung oben definierter Abbilder (*Images*) im Vordergrund. Diese kann auf zwei verschiedene Art und Weisen erfolgen, wie in Abbildung 2.3 dargestellt. Der eher theoretische Ansatz (1) prüft, ob das BI_D^C mit dem CI_D^S übereinstimmt. Ist dies der Fall, so wurden die replizierten Daten auf dem Server in der Abwesenheit des Client nicht verändert und dieser kann seine Änderungen AI_D^C durch Überschreiben der entsprechenden Daten CI_D^S einbringen. Implementiert wird dieses Vorgehen in der Praxis nach dem zweiten Ansatz (2), der nur die auf dem BI durchgeführten Änderungen von Server (ΔBI_D^S) und Client (ΔBI_D^C) betrachtet und miteinander vergleicht. Tupel, die nur in einer der beiden Mengen vorkommen, können konfliktfrei eingebracht bzw. propagiert werden. Ein Konflikt kann dagegen für Tupel entstehen, die in beiden Mengen existieren.

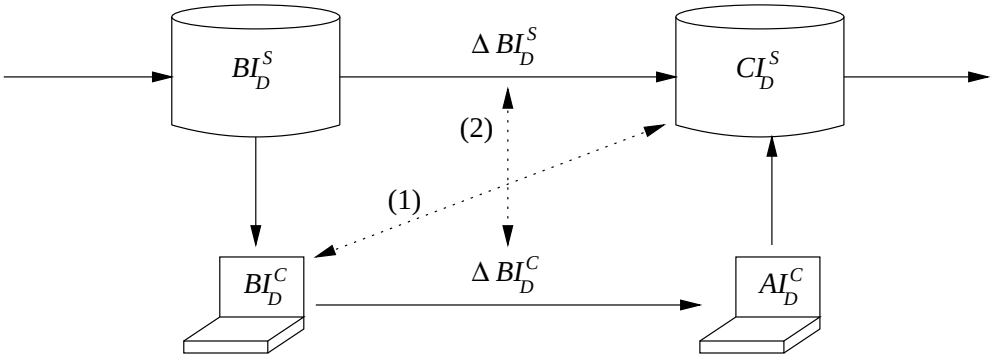


Abbildung 2.3: Entstehung und Vergleich der Datenabbilder

  Allerdings gibt es für eine Datenmenge D immer nur ein AI_D^C bzw. ein ΔBI_D^C, welches aber mitunter das Ergebnis mehrerer aufeinanderfolgender Transaktionen ist. Damit geht beim Reintegrieren der Rahmen sowie die Semantik einzelner Operationen bzw. Transaktionen verloren, wodurch einzelne konfligierende Operationen nicht mehr erkannt und behandelt werden können und gegebenenfalls die gesamten Änderungen für ein Tupel abgewiesen werden müssen. Außerdem fehlt die oft betriebswirtschaftlich wichtige Zuordnung zwischen Daten und den erzeugenden Transaktionen.

- *Operations- bzw. Transaktionsorientiert*
  Bei dieser Form der Reintegration werden sowohl lokal ausgeführte Operationen als auch Transaktionen nach dem *Two-Tier-Prinzip* [GHOS96] auf den aktuellen Daten des Server erneut ausgeführt. Dazu sind neben den hier ebenfalls verwendeten Abbildern (*Images*) zur Konflikterkennung die Übermittlung der protokollierten Änderungsoperationen des Client notwendig, mit deren Hilfe die auf BI_D^C durchgeführten Operationen bzw. Transaktionen auf dem aktuellen Datenbestand CI_D^S des Server nachgefahren werden. Dies geschieht in der Regel analog zum Nachfahren von Änderungen (*Redo*) im Recovery-Fall klassischer DBMS [GR93]. Die Reihenfolge der einzubringenden Transaktionen entspricht dabei der lokalen Ausführungsordnung. Ist zusätzlich die Semantik einer Operation bekannt, beispielsweise '`Inkrementiere Attribut Menge um 10`', ist bei der Konfliktbehandlung eine größere Flexibilität möglich.

Weiterführende Betrachtungen zum Vergleich zwischen daten- und transaktionsorientierter Reintegration finden sich in [Lie01].

Wurden Daten optimistisch repliziert, kann es durch Konflikte während der Reintegration zur Gefährdung der ACID-Eigenschaften [GR93] für lokale Transaktionen eines Client kommen, wie in Tabelle 2.1 dargestellt. Die ACID-Eigenschaften garantieren bei Gewährleistung für parallel ausgeführte Transaktionen ein Ergebnis, welches äquivalent zum Ergebnis einer seriellen Ausführung ist. Diese Eigenschaft wird unter dem sogenannten Serialisierbarkeitskriterium [GR93] zusammengefaßt. Eine Verletzung dieses Kriteriums kann zu Anomalien und Inkonsistenzen in der Datenbank führen, auf die im Kapitel 4 noch näher eingegangen wird.

Eigenschaft	Beispiel für Gefährdung
Atomicity (Atomarität): entweder alle Operationen oder keine Operation einer Transaktion werden ausgeführt.	Konfligierende Transaktionen werden mit Hilfe alternativer Änderungen eingebracht.
Consistency (Konsistenz): Transaktionen überführen die Datenbank von einem konsistenten Zustand zum nächsten.	Bei einer Teilreplikation kann es zur Verletzung nicht replizierter Integritätsbedingungen kommen.
Isolation (Isolation): Transaktionen sehen sich trotz physischem Mehrbenutzerbetrieb im logischen Einbenutzerbetrieb ausgeführt.	Jede Art von Konflikt, der bei der Reintegration auftritt, verletzt die Isolationseigenschaft.
Durability (Dauerhaftigkeit): die Ergebnisse abgeschlossener Transaktion sind persistent in der Datenbank.	Eine lokal abgeschlossene Transaktion kann wegen Konflikten nicht reintegriert werden.

Tabelle 2.1: ACID-Verletzung für lokale Transaktionen

2.2 Identifizierung der Daten

Für die Identifizierung eines Datentupels stehen in der Regel folgende zwei Konzepte zur Verfügung:

- *RowID-Identifizierung*
  Zur eindeutigen Identifizierung eines Tupels innerhalb einer Tabelle wird diesem eine sogenannte *RowID* zugewiesen. Diese basiert auf dem TID-Konzept [HR01] (*Tupel-Identifikator*) und ist in der Regel wie folgt aufgebaut:

  TID = | Segmentnummer | Seitennummer | Slotnummer |

  Dabei legt die Kombination aus der Angabe des Segments, einer Seite in diesem Segment und eines konkreten Slots innerhalb dieser Seite die genaue physische Ablage eines Tupels im Speichersystem fest. Vorteilhaft ist hierbei der schnelle Zugriff durch Kenntnis der physischen Lage des Tupels innerhalb der Datenbank, Tupelverlagerungen auf andere Seiten müssen allerdings durch sogenannte *Forward*-Zeiger gelöst werden, wodurch sich die Anzahl der Speicherzugriffe von ursprünglich einem auf zwei Zugriffe verdoppelt.

In früheren Produktversionen wurde das RowID-Konzept beispielsweise auch von Oracle verwendet, ist aber mittlerweile bis auf die Identifizierung eines Tupels im physischem Speichersystem über den TID in keinem relationalen DBMS mehr zu finden. Für die Replikation ist dieses physische Konzept nicht geeignet, da unter anderem eine Transformation der *RowID* eines Datentupels zwischen Client und Server aufgrund unterschiedlicher Speichersysteme notwendig wäre. Nachteilig wirkt sich auch der große Aufwand zur Änderung und Anpassung der RID nach einer Reorganisation der Datenbank aus.

- *PK-Identifizierung*
  Im Gegensatz zur physisch orientierten Identifizierung über die RowID bietet das Konzept der Identifizierung über den *Primärschlüssel (Primary Key, PK)* eine Entkopplung bzw. nicht erkennbare Zuordnung zwischen dem logischen Tupelgranulat und seiner physischen Speicherung. Die Werte bzw. Kombinationen der Werte eines solchen aus dem relationalen Modell hergeleiteten Primärschlüssels dürfen innerhalb einer Tabelle nur einmal auftreten. Dabei kann einerseits der Primärschlüssel über ein oder mehrere Attribute eines Tupels definiert werden und andererseits reale Werte der abgebildeten Miniwelt (*natürlicher Schlüssel*) oder automatisch generierte Werte (*künstlicher Schlüssel*) zugewiesen bekommen. Anders als die RowID ist der Primärschlüssel vom verwendeten DBMS relativ unabhängig und erlaubt eine einfache Migration. Speziell für die Replikation und einige Verfahren zur Vermeidung bzw. Auflösung von Konflikten ist der Primärschlüssel notwendige Voraussetzung, sodaß in dieser Arbeit das Konzept des Primärschlüssels zur Tupelidentifikation zugrunde gelegt wird.

2.3 Anwendungsszenarien

2.3.1 Szenario 1: Verkäufer im Außendienst

Sei eine Lagerverwaltung eines Herstellers betrachtet, dessen Außendienstmitarbeiter die Produkte im mobilen Verkaufsgeschäft an Großhändler vermitteln. Dazu replizieren sie sich am Tagesbeginn neben den Daten von zu besuchenden Kunden auch gewisse Mengen zu verkaufender Produkteinheiten. Desweiteren ist ihre Aufgabe neue innovative Produkte zu finden und in Eigenverantwortung in den Produktkatalog zu übernehmen. Am Tagesende werden Änderungen an diesen Daten (Vertragsänderungen, Änderungen persönlicher Daten, Verkaufsmenge) wieder mit dem Server synchronisiert.

In diesem Umfeld ist oft eine sehr starke regionale Aufteilung festzustellen, d.h. ein Verkäufer hat beispielsweise im Umkreis seines Wohnortes eine feste Kundschaft und umgekehrt ist für einen gewissen Bereich auch nur genau ein Mitarbeiter zuständig, sodaß eine nichtüberlappende Zuordnung zwischen Kunden und Verkäufern entsteht. Allerdings können sehr wohl mehrere Verkäufer die gleichen Produkte verschiedenen Kunden verkaufen, sodaß eine vorherige Reservierung von bestimmten Kontigenten je Verkäufer vorgenommen werden muß, um Konflikte durch den Vertrieb nicht vorhandener Mengen zu vermeiden. Schließlich sind auch Konflikte bezüglich neuer Einfügungen im Produktkatalog möglich und sollten vermieden werden.

Die Eigenschaften dieses Szenarios sind eine anwendungsgegebene disjunkte Verteilung und Bearbeitung von Daten unter den einzelnen mobilen Mitarbeitern, wodurch Konflikte

automatisch durch eine entsprechende Schemadefinition vermieden werden können. Einige wenige Konflikte, wie der parallele Zugriff auf eine gemeinsame Produktmenge oder die Einfügung neuer Produkte, können über entsprechende Konzepte, wie beispielsweise das `ESCROW`-Verfahren oder das `KEY-POOL`-Verfahren (Kapitel 3), vermieden werden.

2.3.2 Szenario 2: Reiseinformationssystem HERMES

Wie in [Bau03] vorgeschlagen, sind mobile Nutzer am Aufbau und der Pflege des interaktiven HERMES-Systems zur Verwaltung von Reiseinformationen beteiligt. Anwender können auf bereits bestehende Daten zugreifen, diese lesen und verändern sowie neue Daten einfügen. Dabei werden geographische und demographische Daten zu Städten bereitgestellt, aber auch Angaben zu Sehenswürdigkeiten, Gastronomie und kulturellen Einrichtungen. Vervollständigt wird das System durch die Möglichkeit der Nutzer, persönliche Bewertungen beispielsweise zu einem besuchten Restaurant abgeben zu können. Um an einem Reiseziel derartige Aktionen durchführen zu können, müssen vorher Daten repliziert werden, aber weiterhin für andere Nutzer verfügbar bleiben. Trotz gewisser Zugriffsrechte und Regeln für Nutzer, kommt es hier zu einer breiten Konfliktvielfalt, wenn beispielsweise mehrere Nutzer verschiedene historische Informationen zu einer Sehenswürdigkeit eintragen wollen.

Charakteristisch für solche Szenarien ist die nichtdisjunkte Replikation von Daten auf verschiedene Clients, da es keine strikte Aufgabenteilung gibt und somit prinzipiell alle Daten für jeden Client relevant sein können. Dadurch enstehen verstärkt Konflikte aufgrund inkompatibler Änderungen auf gleichen Daten. Dementsprechend sind zusätzliche Möglichkeiten zur Vermeidung und Behandlung von Konflikten notwendig.

Kapitel 3

Konfliktvermeidende Synchronisation

Geht man von einer Situation mit vielen unabhängig voneinander arbeitenden und replizierenden Clients aus, so entstehen dementsprechend viele parallele langlaufende Transaktionen, welche die unverbundene Arbeitsphase der mobilen Clients repräsentieren. Der einfachste Ansatz zur Synchronisation mehrerer langlaufenden Transaktionen ist die Anwendung bekannter klassischer Sperrverfahren, welche im Abschnitt 3.1 behandelt werden. Allerdings ergeben sich bei Anwendung auf das mobile Szenario erhebliche Verfügbarkeitseinbußen, weswegen im Abschnitt 3.2 und Abschnitt 3.3 alternative und für den mobilen Einsatz modifizierte konfliktvermeidende Verfahren betrachtet werden.

3.1 Konfliktvermeidung durch Sperren

3.1.1 Grundlagen

Seien A_i die Menge der Aktionen von Transaktion T_i und A_k die Menge der Aktionen von Transaktion T_k, wobei $T_i \neq T_k$. Zwei Aktionen $a \in A_i$ und $b \in A_k$ stehen in **Konflikt** zueinander, wenn beide auf dasselbe Datenbankobjekt zugreifen und mindestens eine von beiden eine Schreibaktion ist.

Abbildung 3.1: Allgemeine Konfliktdefinition

Will man parallel laufende Transaktionen unter Vermeidung der in Abbildung 3.1 dargestellten allgemeinen Konfliktdefinition [Dad96] synchronisieren, werden üblicherweise Sperren verwendet. Sperrverfahren haben allgemein die Eigenschaft, daß eine Transaktion vor dem Zugriff auf ein Objekt, abhängig von der beabsichtigen Operation, eine Sperre für dieses anfordern muß. Zur Gewährleistung der ACID-Eigenschaften und damit des Serialisierbarkeitskriteriums müssen dabei nach dem *Fundamentalsatz des Sperrens* [HR01] folgende Bedingungen erfüllt sein:

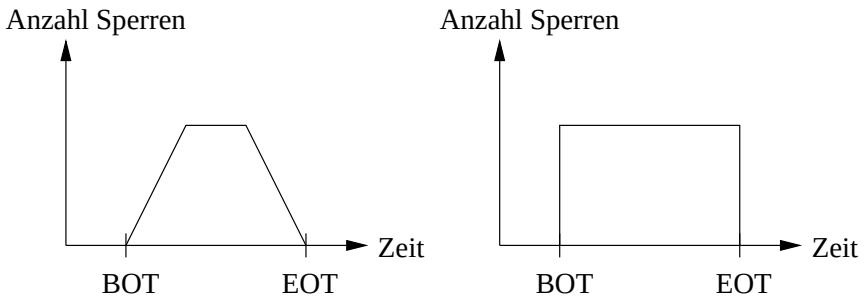
- Jedes Objekt muß vor dem Zugriff mit einer Sperre belegt werden.

- Sperren anderer Transaktionen sind zu beachten, d.h. bei unverträglicher Sperranforderung muß auf die Freigabe gewartet werden.
- Transaktionen fordern keine schon in ihrem Besitz befindlichen Sperren an.
- Es gibt zwei sich nicht überschneidende Phasen, eine Sperranforderungsphase und eine Sperrfreigabephase.
- Spätestens bei Transaktionsende werden alle Sperren einer Transaktion freigegeben.

Allerdings ist ein gewisser Overhead für die Protokollierung der Sperrverwaltung nötig, desweiteren sind pessimistische Synchronisationsverfahren oft zu restriktiv und verringern dadurch die Verfügbarkeit von Daten.

3.1.2 Zwei-Phasen-Sperrprotokoll

Das *Zwei-Phasen-Sperrprotokoll* (2PL, Two-Phase Locking) stellt die direkte Umsetzung des vorgestellten Fundamentalsatzes dar. Wir betrachten hier die in Abbildung 3.2 dargestellten zwei Varianten.



(a) Normales 2PL (b) Striktes 2PL

Abbildung 3.2: Varianten des Zwei-Phasen-Sperrprotokolls

- *Normales 2PL*
  In der Sperranforderungsphase werden sukzessiv nur für die aktuelle Transaktionsverarbeitung benötigte Sperren reserviert. Analog werden nicht mehr benötigte Sperren so früh wie möglich und mitunter vor dem Abschluß der Transaktion (EOT, *End Of Transaction*) freigegeben. Dadurch werden Änderungen einer Transaktion T_1 für andere Transaktionen sichtbar, die im Falle eines Rücksetzens von T_1 zu Anomalien und Nebeneffekten wie dem kaskadierenden Rücksetzen (*Cascading Abort*) führen können. In dem in Abbildung 3.3 dargestellten Szenario liest beispielsweise Transaktion T_2 das Datum a, welches vorher von der noch nicht abgeschlossenen Transaktion T_1 geschrieben wurde, die schließlich mit Abort abbricht und somit Transaktion T_2 ebenfalls zum Abbruch zwingt. Eine genauere Betrachtung dieses auch im mobilen Szenario relevanten Konfliktfalls findet sich im Kapitel 4.

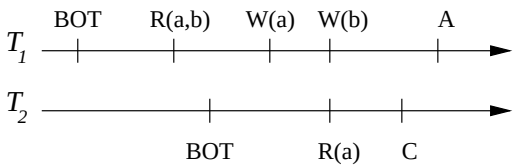


Abbildung 3.3: Probleme beim normalen Zwei-Phasen-Sperrprotokoll

- *Striktes 2PL*
  Unter dem strikten 2PL wird hier sowohl eine vollständige Sperranforderung zum Beginn der Transaktion (BOT, *Begin Of Transaction*), das sogenannte *Preclaiming*, als auch eine vollständige Sperrfreigabe zum Zeitpunkt des Commit verstanden. Durch dieses Vorgehen kann eine Transaktion vollkommen isoliert von parallel laufenden Transaktionen ausgeführt oder zurückgesetzt werden. Allerdings verringert sich hierbei wesentlich die Verfügbarkeit der Daten, weswegen diese Form des Sperrens in realen Datenbankumgebungen mit einer Vielzahl von Nutzern bis auf Ausnahmen im Design- und Entwicklungsbereich komplexer Objekte kaum Verwendung findet. Desweiteren ist es schwierig, ohne eine genaue Anwendungsanalyse die zu sperrende Datenmenge a-priori zu bestimmen. Dieses Problem wird im mobilen Szenario dadurch gelöst, daß in der Replikationsphase zumindest eine Obermenge der verwendeten Daten bekanntgegeben wird.

Anwendung kann das Zwei-Phasen-Sperrprotokoll in verschiedenen Aspekten des mobilen Szenarios finden, wie beispielsweise während der eigentlichen Replikations- und Reintegrationsphase, um konkurrierende Änderungen anderer Nutzer auf den betroffenen Daten zu vermeiden. Für das in dieser Arbeit zugrundegelegte mobile Szenario ist aufgrund der Unverbundenheit nur das strikte 2PL für langlaufende Transaktionen verschiedener Clients verwendbar, da nach der Verbindungstrennung keine weiteren Sperren vom Client angefordert werden können. Ebenso lassen sich freigegebene Sperren nicht vor der Reintegration zum Server übermitteln. Effektiv werden also die benötigten Daten durch die Replikation auf dem Server gesperrt (*Check-out*) und im Zuge der Reintegration ohne weitere Konfliktbehandlung wieder freigegeben (*Check-in*), da Konflikte auf den gesperrten Daten nicht entstehen können. Dieses Vorgehen entspricht dem in Abschnitt 3.1.3 vorgestellten *Check-out/Check-in*.

3.1.3 Check-out/Check-in

Eine übliche Methode für den Zugriff auf große Objekte in Entwicklungsumgebungen mit mehreren parallel arbeitenden Teammitgliedern ist die Verwendung des *Check-out/Check-in*-Verfahrens [HSD+94, SS99]. Bei Zugriffswunsch wird für den Nutzer eine Kopie der zu bearbeitenden Datenobjekte mit *Check-out* angelegt. Nach seinen Änderungen auf der replizierten Objektkopie bringt er diese mit *Check-in* wieder ein. Häufig wird durch diesen Vorgang eine neue Version erzeugt, wie im Multiversionsmodell [GB94, HR01] erläutert, und ist nicht gleichzusetzen mit einer üblichen Schreiboperation [KS91]. Während der Bearbeitung der mit *Check-out* angeforderten Objekte auf dem Client, verwaltet der Server für diese Objekte eine dauerhafte Sperre, analog zum strikten 2PL wie in Abbildung 3.2(b) dargestellt. Eine Erhöhung der Verfügbarkeit trotz langer Sperren für solche Objekte kann

durch eine geeignete Versionsverwaltung unter Nutzung älterer Kopien stattfinden [HR01]. Weiterhin muß dafür gesorgt werden, daß die Änderungen an den ausgelagerten Objekten konsistenterhaltend sind und beim Einbringen alle Integritätsbedingungen mit im Zusammenhang stehenden Objekten erfüllen. Genauere Ausführungen zu diesem Thema finden sich in [LP83].

Die Nutzung von *Check-out/Check-in* für langlaufende Transaktionen beim mobilen Datenbankzugriff ist nur in einigen Ausnahmefällen sinnvoll, da die Verfügbarkeit der Daten stark eingeschränkt wird, insbesondere wenn die exklusiv angeforderte Replikatmenge sehr groß ist bzw. häufig angefragte Daten, sogenannte *Hot Spots*, enthält. Ist die Häufigkeit paralleler Zugriffe auf eine Datenmenge dagegen gering, beispielsweise bei einem relativ unabhängigen disjunkten Replikationsschema der Nutzer, dann zieht eine Sperrung von replizierten Daten über einen längeren Zeitraum keine allzugroßen Einschränkungen für andere Nutzer nach sich. Diese Annahme kann allerdings nur für das bereits im Abschnitt 2.3.1 vorgestellte Verkäufer-Szenario aufrecht erhalten werden. Für Anwendungen wie das HERMES-Szenario ist eine Synchronisation mit *Check-out/Check-in* jedoch nicht generell für alle Nutzer einsetzbar.

3.2 Konfliktvermeidung durch Reservierungen

Da eine unveränderte Verwendung klassischer pessimistischer Synchronisationsverfahren für den mobilen Datenbankzugriff im hier zugrundegelegten Arbeitsmodell aus den im Abschnitt 3.1 dargelegten Gründen nur in wenigen Fällen geeignet ist, wird die Nutzung anderer Methoden zur Konfliktvermeidung notwendig. Eine solche Methode ist die Erzeugung *künstlicher Unabhängigkeit* zwischen replizierten Daten. Im folgenden werden dazu das `KEY-POOL`-Verfahren (Abschnitt 3.2.1), das `ESCROW`-Verfahren (Abschnitt 3.2.2) sowie das `SLOT`-Verfahren (Abschnitt 3.2.3) vorgestellt. Diese Konzepte verwenden gewisse Eigenschaften von Datenstrukturen und Attributen, um einen parallelen, aber dennoch konfliktfreien, Zugriff auf diese zu ermöglichen. Hierfür ist allerdings die Integration von Semantik der Daten in die Verfahren zur Konfliktbehandlung nötig.

3.2.1 `KEY-POOL`-Verfahren

Das unter anderem von Sybase im *Adaptive Server Anywhere/SQL Anywhere Studio* [Syb02a] eingesetzte `KEY-POOL`-Verfahren dient der Gewährleistung unabhängiger Einfügeoperationen verschiedener mobiler Clients mit dem Ziel, die Schlüsseleigenschaft von unabhängig eingefügten Tupeln zu wahren. Grundlage ist hierbei die Tupelidentifizierung über den Primärschlüssel, wobei dieser einattributig und automatisch generierbar sein muß. Eine Modifikation des `KEY-POOL`-Konzeptes für einen mehrattributigen natürlich Primärschlüssel ist nicht sinnvoll, da dieser auf einen einattributigen künstlichen Primärschlüssel reduziert werden kann. Für mehrattributige künstliche Primärschlüssel, d.h. im Zusammenhang mit Fremdschlüsselbeziehungen, ist dieses Verfahren nicht ohne Modifikation anwendbar. Dadurch sind beispielsweise in Sybase ausschließlich künstliche und zahlwertige einattributige Primärschlüssel für das `KEY-POOL`-Verfahren zugelassen.

Beim Replizieren von Daten wird nun individuell jedem Nutzer eine gewisse Anzahl von eindeutigen Primärschlüsseln zur alleinigen Verwendung bereitgestellt, wie dies Abbildung 3.4 veranschaulicht. Unverbundene Einfügungen von Tupeln mit diesen reservier-

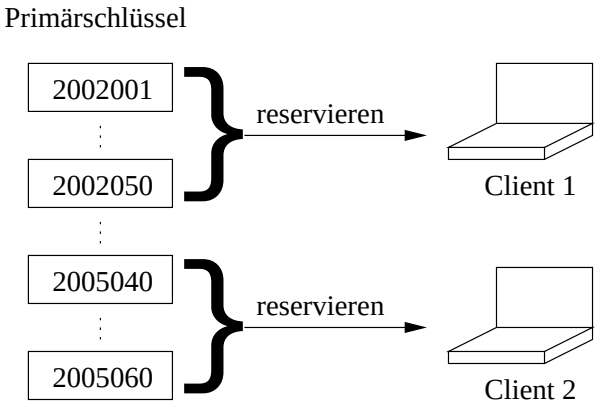


Abbildung 3.4: Beispiel für die `KEY-POOL`-Technik

ten Primärschlüsseln können garantiert konfliktfrei eingebracht werden, da Mechanismen im Server-DBMS darauf achten, reservierte Primärschlüssel nicht zu verwenden. Bei der Reintegration kann automatisch die Auffüllung des privaten `KEY-POOL`-Bereiches vorgenommen werden. Sowohl die Vergabe als auch der Abgleich von `KEY-POOL`-Mengen einzelner Clients findet bei Sybase mit Hilfe einer Prozedur, der sogenannten *Replenish-Pool*-Prozedur [Syb02b], statt. Diese muß in regelmäßigen Abständen, auch automatisiert möglich, auf der konsolidierten Serverdatenbank gestartet werden und füllt eine entsprechende `KeyPool`-Tabelle mit neuen Primärschlüsseln bis zu einer für alle `KEY-POOL`-Nutzer gleichen maximalen Anzahl auf. Die `KeyPool`-Tabelle enthält für alle mit dem `KEY-POOL`-Prinzip arbeitenden Tabellen die jeweils reservierten Primärschlüssel.

3.2.2 `ESCROW`-Verfahren

Ziel des in [O'N86] vorgestellten `ESCROW`-Verfahrens ist die Unterstützung von Änderungen langlaufender Transaktionen auf häufig frequentierten Attributen (*Hot Spots*), ohne den Zugriff paralleler Transaktionen auf diese Attribute zu serialisieren. Allerdings ist diese Verfahren nur für spezielle im folgenden beschriebene Attributentypen anwendbar. Ursprünglich für das Umfeld verteilter Datenbanken und langlaufender Transaktionen entwickelt, findet das `ESCROW`-Verfahren gerade in mobilen Datenbankszenarien eine neue Anwendung. Der zugrundeliegende Ansatz ist das weniger bekannte IMS/VS Fast Path [GK85] aus dem Jahr 1974 mit einigen Erweiterungen [Reu82]. Dabei werden aggregierte und häufig zugegriffene Attribute betrachtet, die Angaben zu Mengen bzw. Anzahlen beinhalten. Ein solche Attribut ist beispielsweise `Bestand` in Abbildung 3.5, welches den Lagerbestand eines Produktes darstellt.

Zur Vorbereitung der unverbundenen Arbeit reserviert und repliziert sich beispielsweise ein mobiler Verkäufer die benötigte Menge von Produkteinheiten, die er im Laufe der unverbundenen Arbeit umsetzen will. Zum Zeitpunkt der Synchronisation bringt er seine Änderungen im Rahmen der reservierten Mengen konfliktfrei ein. Die von Transaktionen auf solchen Attributen durchgeführten Operationen beschränken sich auf kommutative Inkrementationen bzw. Dekrementationen wie in [O'N86] beschrieben. Zur Nutzung der `ESCROW`-Technik muß dem DBMS die Semantik der zugrundeliegenden Anwendung bzw.

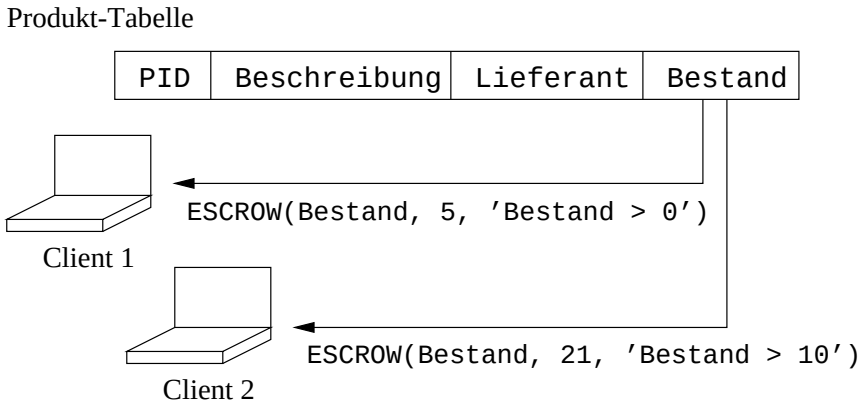


Abbildung 3.5: Beispiel für das ESCROW-Verfahren

des betreffenden Attributes bekannt sein. Dabei steht zur Diskussion, welche Auswirkungen die durchgeführten Reservierungen haben:

- **Minimum:**
  Im Minium-Fall stellt der Wert des ESCROW-Attributes nur die tatsächlich vorgenommenen Änderungen dar. Für existierende Reservierungen muß jedoch weiterhin garantiert werden, daß deren Änderungen konfliktfrei reintegriert werden können. Dadurch stellen Fälle, in denen weitere Reservierungen auf dem sichtbaren Wert eines Attributes möglich wären, bei Beachtung gültiger und nicht sichtbarer Reservierungen vom System jedoch abgewiesen werden, eine gewisse Inkonsistenz für den Nutzer bzw. die Anwendung dar.

- **Maximum:**
  Im Maximum-Fall wird dagegen durch angeforderte Reservierungen von Teilmengen das betroffene Attribut (virtuell) aktualisiert und stellt somit die noch maximal verfügbare Menge des Attributes dar. Diese pessimistisch vorgenommenen Änderungen sind allerdings nicht endgültig, da nicht benötigte Kapazitäten, beispielsweise von zu verkaufenden Produktmengen, während der Synchronisation in den Bestand rücküberführt werden können. Aufgrund der einfacheren Realisierung wird in dieser Arbeit, insbesondere im Kapitel 6, diese Semantik von Reservierungen zugrundegelegt.

In [O'N86] werden für den Zugriff auf ESCROW-Attribute die zwei Operationen ESCROW und USE eingeführt. Wie in Abbildung 3.6 zu sehen, erwartet die ESCROW-Funktion als Parameter das Attribut (field), auf welches zugegriffen werden soll, die zu reservierende Menge (quantity) und eine durch die Anwendung vorgegebene Konsistenzbedingung (test) für dieses Attribut. Ist die so spezifizierte Reservierungsanfrage erfolgreich, wird für den anfragenden Client die angeforderte Menge C1 auf dem Bestand-Attribut reserviert und im sogenannten *ESCROW-Pool* hinterlegt. Dabei muß die angegebene test-Bedingung erfüllt sein, *nachdem* vom Attribut Bestand die Menge C1 entfernt wurde. Diese Bedingung kann zentral durch eine zugrundegelegte Anwendung definiert sein, beispielsweise 'Der Lagerbestand darf nicht unter 20 Einheiten fallen', aber auch durch verschiedene mobile Anwendungen mit Zugriff auf diesselben Daten unterschiedliche Ausprägungen besitzen.

Für die Einhaltung bestehender Testbedingungen im Fall von neuen Reservierungsanfragen ist das DBMS auf dem Server zuständig.

```
IF ESCROW (field=Bestand, quantity=C1, test=(Bestand > 0))
   THEN USE (field=Bestand, quantity=C2)
        ...
        weiterer Programmverlauf
        ...
   ELSE ABORT
```

Abbildung 3.6: Eine typische `ESCROW`-Anforderung

Eine wichtige Eigenschaft ist die Garantie, daß jede einmal durchgeführte Reservierung von Teilmengen ihre Gültigkeit bewahrt, im ursprünglichen Szenario also bis zum Ende der anfordernden langlaufenden Transaktion. Im mobilen Szenario entspricht dies der Reservierung einer Teilmenge mit Hilfe der `ESCROW`-Funktion zum Zeitpunkt der Replikation und der konfliktfreien Verwendung im unverbundenen Zustand, verdeutlicht durch die Funktion `USE`. Diese entnimmt die Menge `C2` aus dem zum Attribut `Bestand` gehörenden `ESCROW`-Pool. Dabei darf eine Transaktion maximal die für sich reservierte Menge verwenden (`C2` $\leq$ `C1`), d.h. die Anwendung auf dem mobilen Client muß absichern, daß Änderungen innerhalb des reservierten Bereichs bleiben. Nicht genutzte Mengen werden nach Transaktionsende aus dem `ESCROW`-Pool wieder an das `ESCROW`-Attribut zurückgegeben. Zudem ist es einer Transaktion erlaubt, mehrere `ESCROW`-Anforderungen zu stellen, was allerdings aufgrund der fehlenden Verbindung im hier zugrundgelegten mobilen Szenario nicht möglich ist.

3.2.3 `SLOT`-Verfahren

Das `SLOT`-Verfahren, wie es auch in MobiSnap [PBM+99, PBM+00] Verwendung findet, dient zur Vermeidung von Einfügekonflikten, allerdings ist es im Gegensatz zum `KEY-POOL`-Verfahren auch für Tabellen geeignet, in denen Tupel über einen natürlichen Primärschlüssel identifiziert werden. Will ein mobiler Client unter Anwendung des `SLOT`-Verfahrens konfliktfrei einfügen, muß er im Zuge der Replikation die entsprechenden Tupel über ein sogenanntes `PRE-INSERT` bereits einfügen. Neben den zum Schlüssel gehörenden Attributen sind dabei auch durch `NOT NULL` gekennzeichnete Attribute mit den späteren und hier bereits notwendigerweise bekannten Werten zu belegen. Dies ist nicht automatisch möglich, wie beispielsweise die Vergabe von Primärschlüsseln beim `KEY-POOL`-Verfahren, und erfordert das genaue Wissen über die zukünftige unverbundene Arbeit eines Nutzers. Das Ergebnis ist die Reservierung des Einfügerechtes auf die spezifizierten Tupel, konkurrierende Einfügungen von Tupeln mit gleichem Schlüssel sind nicht mehr möglich, da die reservierten Tupel in der Datenbank des Server eingefügt und mit einer Sperre versehen werden. Die unverbunden durchgeführten Einfügungen stellen dann zur Synchronisation effektiv ein `UPDATE` der vorerst mit `NULL` belegten Tupelwerte dar.

Als Beispiel betrachten wir die Tabelle `Meetings` in Abbildung 3.7, deren Tupel geplante Treffen darstellen, wobei der Primärschlüssel durch die Kombination aus `Raum`, `Datum` und `Zeit` gebildet wird. Weitere optionale Daten sind die Benennung eines Leiters, der die Koordination des geplanten Treffens übernimmt, sowie eine Inhaltsbeschreibung des

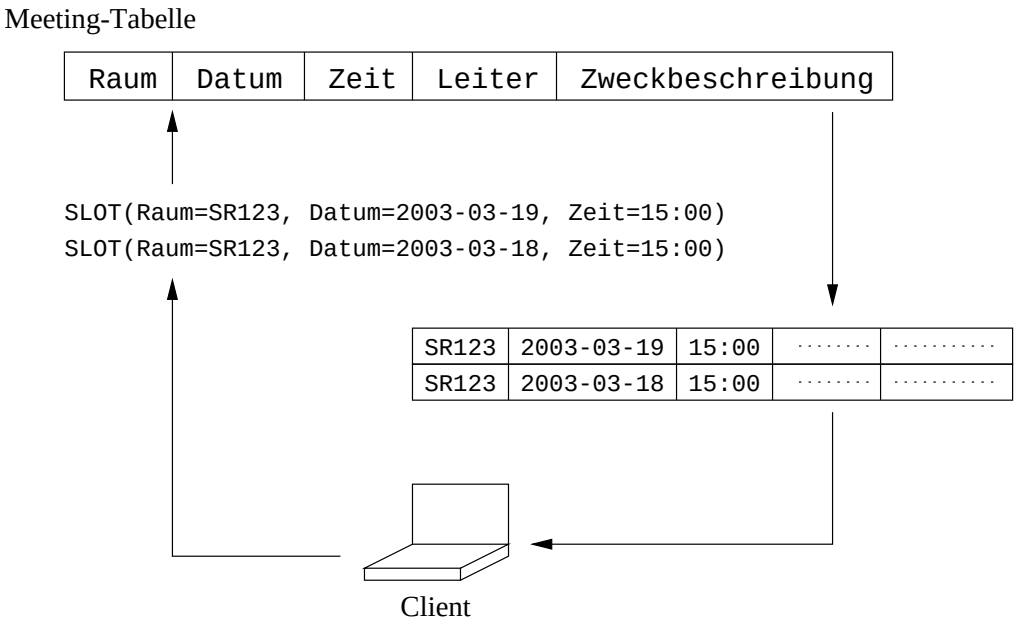


Abbildung 3.7: Beispiel für eine `SLOT`-Anforderung

Treffens. Will ein mobiler Client neue Treffen einfügen, muß er während der Replikation den Raum sowie die zeitlichen Daten festlegen, wie im Beispiel gezeigt, und erhält das Einfügerecht für die spezifizierten Tupel. Ergänzende Daten zu den Attributen `Leiter` und `Zweckbeschreibung` können nun mobil eingefügt und später als `UPDATE` auf die bereits existierenden Tupel reintegriert werden.

3.3 Weitere Konfliktvermeidungsverfahren

3.3.1 Begrenzung von Sperren

Eine Möglichkeit, unverbunden durchgeführte Änderungen konfliktfrei synchronisieren zu können, ist die exklusive Sperrung der betroffenen Daten für die Zeit der Unverbundenheit, wie im Abschnitt 3.1.3 erläutert. Allerdings ergibt sich dadurch das Problem der geringen Verfügbarkeit von gesperrten Daten vorallem für lange unverbundene Phasen. Erschwerend kommt hinzu, daß eine Entscheidungsgrundlage fehlt, ob ein Client nur eine lange Zeit vom Server getrennt arbeitet oder durch einen Systemausfall ohne Wiederherstellungsmöglichkeiten, wie z.B. auf einem PDA, die zu erwartende Reintegration nicht durchführen kann.

Eine Lösung für derartige Probleme bietet das Konzept von begrenzten Reservierungen für gewisse Operationen auf den replizierten Daten, welches auch im MobiSnap-Projekt [PBM+99, PBM+00] Anwendung findet. Dabei wird in dieser Arbeit nur eine Begrenzung des Lösch- und Änderungsrechtes auf bereits existierenden Daten betrachtet. Für die Sicherung des Einfügerechtes müßte entsprechend die gesamte Tabelle gesperrt werden. Für geänderte oder gelöschte Daten wird, ebenso wie bei der klassischen Sperrung, eine konfliktfreie Reintegration dieser Daten garantiert, wenn die Synchronisation innerhalb einer zu spezifizierenden Gültigkeitsdauer erfolgt. Andernfalls werden die gesperrten

Daten, möglicherweise in verschiedenen Abstufungen, auf dem Server freigegeben, wie in Abbildung 3.8 dargestellt. Dadurch erhält man einen Wechsel von pessimistischer zu optimistischer Synchronisation (siehe Kapitel 4) für den betroffenen Client, d.h. seine Änderungen können bei der Reintegration möglicherweise Konflikte erzeugen, die erkannt und gelöst werden müssen. Dieses Vorgehen muß vom lokalen DBS eines Client untersützt werden.

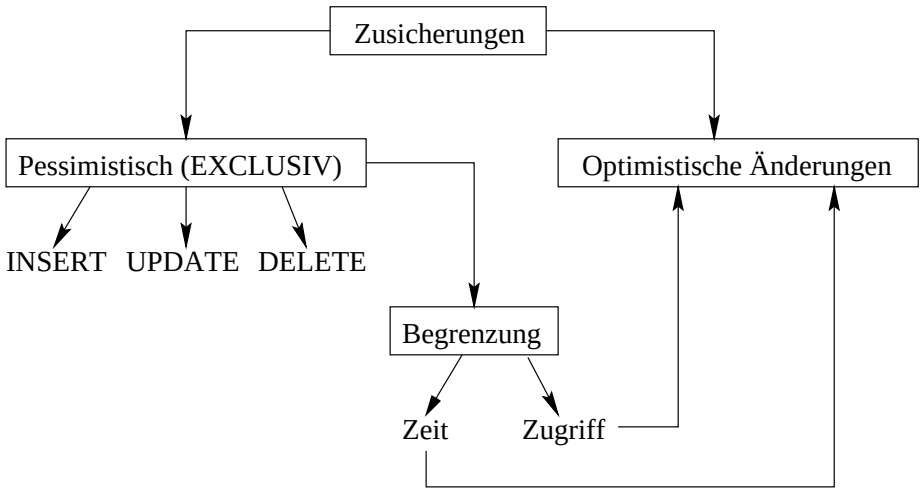


Abbildung 3.8: Klassifikation von Zusicherungen

Für das zu spezifizierende Gültigkeitskriterium sind verschiedene Ansätze denkbar, hier sollen zwei auch im späteren Teil der Arbeit (siehe Kapitel 6) verwendete Möglichkeiten vorgestellt werden:

- *Zeitliche Begrenzung*
  Durch anwendungs- oder systemgesteuerte Festlegung läßt sich eine Zeitspanne für die Gültigkeit gewisser Rechte auf replizierten Daten definieren. Beispielsweise könnte ein Client das exklusive Änderungsrecht auf einer Tabelle für zwei Stunden anfordern. Inwiefern diese Anforderung unter den aktuellen Bedingungen erfüllt werden kann, muß von der Replikationskomponente des Server-DBMS entschieden werden. Wird dieses Recht bewilligt, ist vom DBMS des Server eine Schreibsperre für die angeforderten Tupel zu vergeben, allerdings mit einer zeitlichen Beschränkung versehen. Vorteilhaft ist hier die relativ genaue Informationsmöglichkeit des Client über die verbleibende Gültigkeitsdauer von zeitlich befristeten Sperren auch ohne Kontakt zum Server, d.h. diese kann aus der aktuellen Systemzeit sowie dem Zeitpunkt der Gewährung einer befristeten Zusicherung berechnet werden.

- *Begrenzung durch maximale Zugriffe*
  Will oder kann man keine zeitliche Begrenzung für vergebene Sperren festlegen, läßt sich eine exklusive Operationszusicherung auch limitieren, indem durch Festlegung eines *Schwellwertes* nur eine maximale Anzahl konkurrierender Zugriffsanforderungen für die reservierten Daten erlaubt wird. Dabei können wiederrum durch die Anwendung oder das System genaue Werte für maximale Zugriffe je Operation definiert werden. Beispielsweise könnte das System eine Zusicherung für die konfliktfreie

Löschung von Daten auf 10 Änderungswünsche paralleler Clients beschränken. Fordert nun ein Client C_1 das so begrenzte Recht zur Löschung an, werden die ersten 10 Replikatanforderungen anderer Clients mit konfligierenden Reservierungen auf den gleichen Daten abgewiesen. Danach wird die Löschreservierung von Client C_1 ungültig, d.h. ab diesem Zeitpunkt durchgeführte Löschungen könnten beim Reintegrieren zu Konflikten führen.

In einem noch viel größerem Maße als bei der zeitlichen Begrenzung spielen hierbei die Auswertungen statistischer Informationen über Zugriffsprofile eine enorme Rolle, um die nötigen Parameter sinnvoll zu wählen. D.h. diese Werte sollten, abgesehen von Ausnahmen, durch das System generiert werden. Nachteilig wirkt sich hierbei auch die ungenaue Kenntnis des Clients über den Gültigkeitsstand seiner Reservierungen auf seine unverbundene Arbeit aus, eine Lösung hierzu würden asynchrone Kommunikationsmedien bieten, auf die im Ausblick in Kapitel 8 eingegangen wird.

3.3.2 Einschränkung verfügbarer Operationen

Wird in klassischen Datenbanksystemen eine Schreibsperre für ein Datenobjekt angefordert, so sind in der Regel alle weiteren schreibenden und lesenden Zugriffe paralleler Transaktionen auf diese Daten nicht möglich, da eine exklusive Schreibsperre mit jedem weiteren Sperrtyp unverträglich ist. Dagegen betrachten wir hier Zusicherungen, die getrennt für eine der Operationen `READ`, `INSERT`, `UPDATE` oder `DELETE` den exklusiven Zugriff erlauben. Da diese einen konfliktfreien unverbundenen Zugriff ermöglichen sollen, müssen die für konkurrierende Replikatanforderungen erlaubten Operationen auf diejenigen beschränkt werden, die bei Zugriff auf die gleichen Daten keinen Konflikt erzeugen können. Durch diese Differenzierung der Änderungsoperationen ist eine Verbesserung der Verfügbarkeit gegenüber klassischen Sperrkonzepten möglich.

	Client 2			
Client 1	`READ`	`INSERT`	`DELETE`	`UPDATE`
`READ`	✓	✓	✓	✓
`INSERT`	✓	✓	✓	✓
`DELETE`	✓	✓	✓	–
`UPDATE`	✓	✓	–	–

✓ : kompatibel – : nicht kompatibel

Tabelle 3.1: Verträglichkeitsmatrix von Operationen

Die in Tabelle 3.1 dargestellte Verträglichkeitsmatrix zeigt mögliche Kompatibilitäten zwischen jeweils zwei Operationen, wobei die Anforderung der Operation von Client 2 nach der Reservierung von Client 1 erfolgt. Auf die einzelnen Konfliktfälle wird im Abschnitt 4.3 näher eingegangen. Anwendbar ist dieses Vorgehen nur bei einer operationsorientierten Wiedereinbringstrategie sowie der Möglichkeit, spezielle Operationsrechte auf replizierte Daten anfordern zu können. Beide Voraussetzungen werden von unserem in Kapitel 6 vorgestellten Ansatz erfüllt.

Kapitel 4

Konfliktbehaftete Synchronisation

4.1 Grundlagen

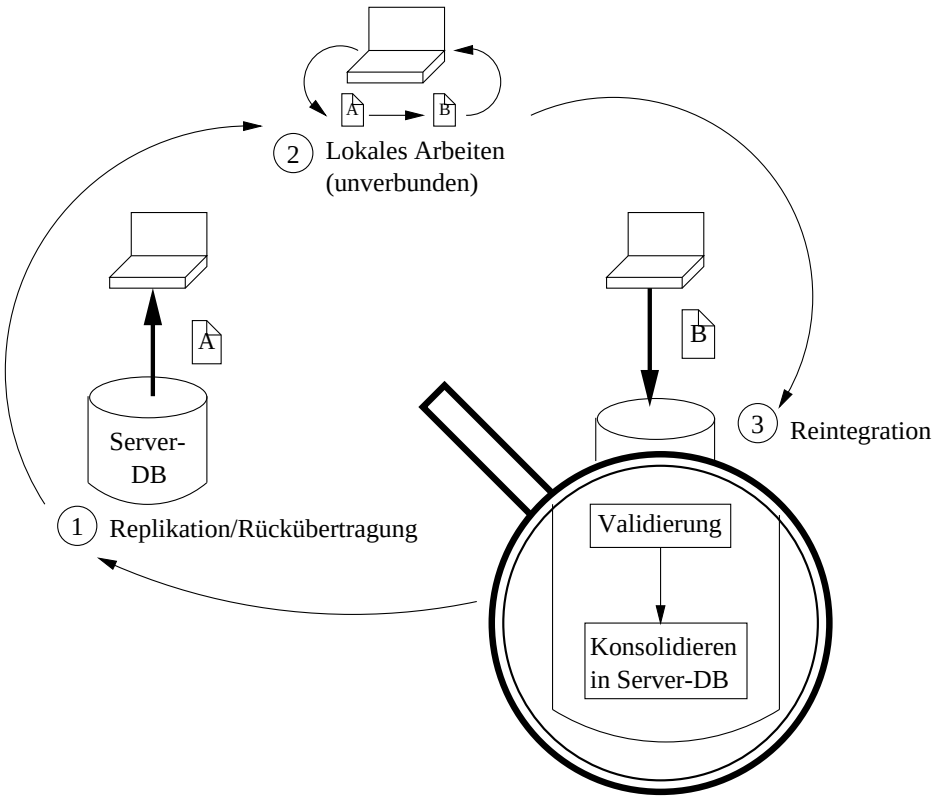


Abbildung 4.1: Konfliktbehaftete Synchronisation

Können, z.B. aus Verfügbarkeitsgründen, konfliktvermeidende Verfahren nicht verwendet werden oder ist deren Einsatz aufgrund eines geringen Konfliktpotentials zu aufwendig, muß auf optimistische Synchronisationsverfahren zurückgegriffen werden. In Abbildung 4.1 ist hierzu das Grundprinzip dargestellt. Ausgangslage ist auch hier die Replikation der benötigten Daten auf dem Client, allerdings bleiben diese Daten auf dem Server frei

verfügbar, d.h. verschiedene mobile Clients können in relativ unkontrollierter Art und Weise gleiche oder überlappende Ausschnitte der Server-Datenbank replizieren. Folglich bleibt eine hohe Verfügbarkeit auch bei vielen Replikatanfragen gewährleistet.

Die unverbunden ausgeführten Änderungen müssen nun bei der Wiederaufnahme der Verbindung zum Server unter erheblich mehr Aufwand reintegriert werden. Die einfache Phase der Reintegration des zugrundegelegten und im Kapitel 2 beschriebenen Modells besteht nun aus zwei Teilphasen. In der *Validierung* wird geprüft, inwiefern lokale Änderungen des Client konfliktfrei bezüglich der aktuell vorliegenden Daten des Servers eingebracht werden können. Dies wird durch ein entsprechendes Verfahren zur *Konflikterkennung* gewährleistet. Anschließend wird in der Phase der *Konsolidierung* die dauerhafte Reintegration der lokalen Änderungen in den Datenbestand des Server vorgenommen. Wurden während der Validierung Konflikte erkannt, sind diese zuvor mit einem geeigneten, möglichst automatischen, Verfahren zur *Konfliktauflösung* zu behandeln.

4.1.1 Szenario und Voraussetzungen

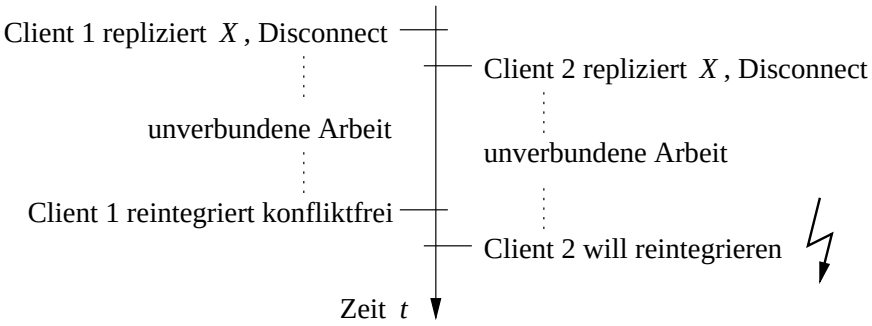


Abbildung 4.2: Konfliktszenario

Ausgangslage für das in Abbildung 4.2 dargestellte Szenario sind zwei mobile Clients, Client 1 und Client 2, die einen **gemeinsamen** Datenbestand, verdeutlicht durch das Tupel X, replizieren. Nach der parallelen unverbundenen Arbeit wird ohne Beschränkung der Allgemeinheit Client 1 als erster synchronisiert, unter der Annahme, daß es dabei zu keinem Konflikt kommt. Dadurch entsteht ein neuer konsolidierter Datenbankzustand, auf dem anschließend Client 2 ebenfalls synchronisiert wird, wobei es nun zu unterschiedlichen Konflikten kommen kann. Für eine erfolgreiche Konflikterkennung basierend auf dem Vergleich von Primärschlüsseln wird eine Unveränderbarkeit dieser vorausgesetzt.

4.1.2 Klassifikation der Konfliktbehandlung

Im folgenden sollen verschiedene Möglichkeiten zur Konflikterkennung mit entsprechenden Varianten zur Konfliktauflösung untersucht werden. Das Klassifikationsmerkmal ist hierbei die Art der Reintegration. Die einfachste und von den meisten Produkten verwendete Variante ist die im Abschnitt 4.2 beschriebene datenorientierte Konfliktbehandlung, d.h. die Reintegration der Änderung eines einfachen Datentupels. Protokolliert man auf dem mobilen Client zusätzlich die Operationen, welche diese Änderungen verursacht haben,

können dadurch genauere Verfahren zur sogenannten operationsorientierten Konfliktbehandlung, wie im Abschnitt 4.3 dargestellt, angewendet werden. Schließlich kann unter Beachtung des Transaktionskontextes mit der im Abschnitt 4.4 beschriebenen transaktionsorientierten Reintegration zumindest die Atomarität einzubringender Transaktionen gewährleistet werden.

Sowohl für die Konflikterkennung als auch für die automatische Konfliktauflösung auf den unterschiedlichen Granulatebenen sind verschiedene zusätzliche Informationen nötig, wie beispielsweise das RS_T und WS_T einer Transaktion T. Die Bereitstellung dieser Informationen wird im Kapitel 7 genauer untersucht und bewertet. Neben den jeweils aufgezeigten automatischen Konfliktauflösungen besteht natürlich auch immer die Möglichkeit, eine manuelle Konfliktauflösung durch den Nutzer oder Administrator durchzuführen. Dabei sind eine entsprechende Benachrichtigung sowie genaue Informationen zu konfligierenden Daten vom System bereitzustellen, um den Aufwand für die manuelle Konfliktbehandlung so gering wie möglich zu halten.

4.2 Datenorientierte Konfliktbehandlung

4.2.1 Konfliktdefinition

Bei der datenorientierten Reintegration stehen außer dem Primärschlüssel und den in Kapitel 2 definierten Abbildern BI_X^C, AI_X^C und CI_X^S eines Tupels X keine weiteren Informationen zur Verfügung. Ein Konflikt bezüglich eines Tupels X entsteht dann, wenn das Tupel X sowohl auf dem Client als auch auf dem Server geändert wurde, d.h. es gelten folgende Bedingungen:

$$\begin{array}{ccc} BI_X^C \neq AI_X^C & \text{und} & BI_X^C \neq CI_X^S \\ & \text{bzw.} & \\ X \in \Delta BI^C & \text{und} & X \in \Delta BI^S \end{array}$$

4.2.2 Konflikterkennung

Zur Konflikterkennung wird jedes geänderte Tupel eines Client mit dem korrespondierenden Tupel des Server, d.h. dem Tupel mit gleichem Primärschlüssel, verglichen. Tritt dabei ein Konflikt nach obiger Definition auf, muß dieser entsprechend gelöst werden. Existiert ein solches Tupel nicht, d.h. es wurde auf dem Client neu erzeugt, wird es im Zuge der Reintegration auf dem Server ebenfalls eingefügt. Hat ein Tupel X sowohl im BI_X^C als auch im CI_X^S gleiche Werte, ist aber nicht mehr im AI_X^C enthalten, so wurde Tupel X durch den Client gelöscht und muß entsprechend auf dem Server ebenfalls gelöscht werden.

4.2.3 Konfliktauflösung

Für die automatische Konfliktauflösung stehen folgende Möglichkeiten zur Auswahl:

Auswahl eines Tupels: Stehen zwei Versionen für ein Tupel zur Verfügung, wird über einfache Regeln eine der beiden Tupelversionen ausgewählt und konsolidiert. Solche Regeln können beinhalten, daß beispielsweise immer die Version des Server gewählt

wird (`server wins`) oder die Version des Client übernommen wird (`client wins`). Diese Art der starren Regelanwendung kann durch Auswertung von, an Nutzer oder Clients vergebene, Prioritäten flexibler gestaltet werden, d.h. das Tupel mit höherer Priorität wird dann übernommen. Beispielsweise könnten in dem in Abschnitt 2.3.2 vorgestellten Reiseinformationssystem HERMES Änderungen an Daten durch deren Besitzer und Erzeuger eine höhere Priorität haben als von anderen Nutzern.

Kombination beider Tupel: Durch Anwendung einer Funktion über die im Konflikt stehenden Tupelversionen kann ein neues Tupel erzeugt werden. Allgemein könnte man hierfür jede beliebige nutzerdefinierte Funktion (*UDF, User Defined Function*) zulassen. Für Konflikte bezüglich numerischer Attributewerte zweier Tupel können aber auch im DMBS integrierte Funktionen wie `sum` oder `average` verwendet werden, solange beispielsweise die Durchschnittsbildung zweier konfligierender Zahlwerte als Konfliktlösung bezüglich der Anwendungssemantik sinnvoll ist.

Umbenennung eines Tupels: Wurde ein Eindeutigkeitskonflikt entdeckt, d.h. es soll ein Tupel mit schon existierendem Primärschlüssel in eine Tabelle eingefügt werden, so kann dieser Konflikt durch eine Umbenennung des Primärschlüssels aufgelöst werden. Allerdings ist diese Variante nur für künstliche Schlüssel geeignet, die vom eigentlichen Inhalt des Tupels unabhängig sind.

4.3 Operationsorientierte Konfliktbehandlung

4.3.1 Konfliktdefinition

Grundlage für Operationskonflikte ist die bereits im Kapitel 3 in Abbildung 3.1 vorgestellte allgemeine Konfliktdefinition. Allerdings werden nur Operationen des Client bzw. Server ohne Beachtung des Transaktionskontextes unterschieden. Um zu erkennen, ob *dasselbe Datenbankobjekt* geändert wird, können verschiedene Informationen verwendet werden. Eine direkte Umsetzung der allgemeinen Konfliktdefinition ist die Verwendung von Zeitstempeln, wie in Abschnitt 4.3.2 beschrieben. Eine für das mobile Umfeld besser nutzbare Methode besteht in der Auswertung von Tupel-Abbildern. Allerdings ist hier gegenüber der datenorientierten Reintegration durch Kenntnis der zugrundeliegenden Operation eine differenziertere Konfliktbehandlung möglich. Eine genaue Definition einzelner Operationskonflikte wird im Abschnitt 4.3.3 gegeben.

4.3.2 Konflikterkennung mit Zeitstempeln

Für den Einsatz dieses in [EN01, RG00] beschriebenen Verfahrens wird jeder Transaktion T eine eindeutige Kennung $TS(T)$ vom DBMS zugeordnet. Dieser Zeitstempel (*Time Stamp*) kennzeichnet den Start-Zeitpunkt der Transaktion T, wobei man üblicherweise nur den Commit-Zeitstempel zur Verfügung hat. In ähnlicher Weise müssen für ein Datenelement X ebenfalls zwei Zeitstempel auf dem Server verwaltet werden. Der Lesezeitstempel $RTS(X)$ (*Read Time Stamp*) kennzeichnet den Zeitstempel der jüngsten Transaktion, die X gelesen hat. Der Schreibzeitstempel $WTS(X)$ (*Write Time Stamp*) ist analog der Zeitstempel der jüngsten Transaktion, die X geschrieben hat. Betrachtungen zu Aufwand und Generierung solcher Zeitstempel finden sich im Kapitel 7.

```
IF TS(T) < RTS(X)
   THEN Abweisung von T
   ELSE IF TS(T) < WTS(X)
            THEN Schreiboperation ignorieren, T fortsetzen
            ELSE Schreibe Objekt X
```

Abbildung 4.3: Konflikterkennung mit Zeitstempeln für eine Schreiboperation

Anhand von einfachen Regeln, die auf Vergleich der Zeitstempel basieren, können Konflikte während der Reintegration erkannt werden. Will beispielsweise eine Operation der Transaktion T das Objekt X schreiben, sind die in Abbildung 4.3 dargestellten Vergleiche notwendig. Die Abweisung von T nach dem ersten Vergleich wird notwendig, wenn eine jüngere, d.h. nach T gestartete, Transaktion den Wert von X gelesen hat. Das Ignorieren der Schreiboperation nach dem zweiten Zeitstempelvergleich unter fortgesetzter Ausführung von Transaktion T wird auch als *Thomas Write Rule* bezeichnet und erlaubt weniger Abweisungen von Transaktionen. Eine genauere Beschreibung dieser sowie weiterer Regeln und Eigenschaften des Zeitstempelverfahrens finden sich in [RG00].

Allerdings ist das Zeitstempelverfahren, ursprünglich als optimistisches Synchronisationsverfahren für den klassischen Datenbankbetrieb entwickelt, ohne Modifikation für den Einsatz im mobilen Szenario nicht anwendbar, wie das Beispiel in Abbildung 4.4 zeigt. In diesem Fall wird das replizierte Datenobjekt X von der unverbunden ausgeführten Transaktion T_2 gelesen und verändert (`UPDATE(X)`). Parallel dazu wird auf dem Server X ebenfalls durch Transaktion T_1 gelesen und geschrieben. Dadurch dürfte Transaktion T_2 seine Änderungen an X nicht konfliktfrei reintegrieren, die Veränderung der entsprechenden Lese- und Schreibzeitstempel wird jedoch durch den Algorithmus zur Konflikterkennung in Abbildung 4.3 nicht als Konflikt erfaßt, da

$$\begin{aligned} TS(T_2) = \quad & 5 > 3 \quad = RTS(X) \\ & \text{und} \\ TS(T_2) = \quad & 5 > 4 \quad = WTS(X). \end{aligned}$$

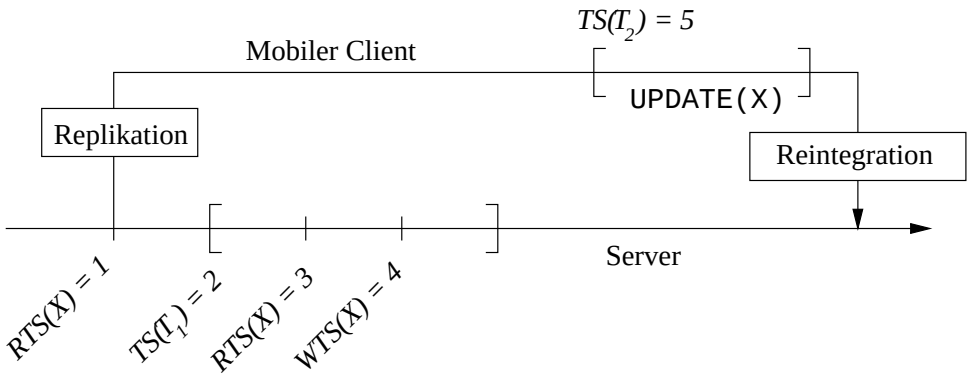


Abbildung 4.4: Probleme des Zeitstempelverfahrens

Eine korrekte Umsetzung des Zeitstempelverfahrens für das hier zugrundegelegte mobile Szenario würde bedeuten, daß eine erfolgreiche Reintegration nur möglich ist, wenn die Zeitstempel von replizierten Daten auf dem Server seit dem Replikationszeitpunkt nicht verändert wurden. Dies führt jedoch zu der in Abschnitt 4.4 beschriebenen Konflikterkennung, die für Transaktionen jeweils die Mengen der gelesenen (RS) bzw. geschriebenen (WS) Daten miteinander vergleicht.

4.3.3 Konflikterkennung mit Abbildern

Wird zusätzlich zu den Abbild-Informationen BI, AI und CI eines Tupels X die Operation protokolliert, die zu einer Veränderung (`INSERT`, `UPDATE`, `DELETE`) des Tupels X geführt hat, so kann vorallem die Konfliktauflösung differenzierter gestaltet werden. Tabelle 4.1 zeigt eine Übersicht der möglichen Konfliktklassen, die auf Grundlage des in Abbildung 4.2 skizzierten Verhaltens zweier Clients möglich sind. Hat Client 1 ein gemeinsam repliziertes Tupel nur gelesen, so kann Client 2 anschließend mit jeder beliebigen Operation auf Tupel X konfliktfrei synchronisiert werden, soweit es keine weiteren Bedingungen oder Abhängigkeiten gibt. Die restlichen Konfliktklassen orientieren sich an der Operation des zuerst eingebrachten Client 1. Beispielweise sind alle Konflikte, die durch eine Lösch-Operation von Client 1 verursacht werden können, im Abschnitt 4.3.3.3 erläutert.

<table>
<tr><td rowspan="2">Client 1</td><td colspan="4">Client 2</td></tr>
<tr><td>Lesen</td><td>Einfügen</td><td>Löschen</td><td>Ändern</td></tr>
<tr><td>Lesen</td><td colspan="4">–</td></tr>
<tr><td>Einfügen</td><td rowspan="3">Konflikte durch Verwendung veralteter Daten (4.3.3.1)</td><td colspan="3">Einfügekonflikte (4.3.3.2)</td></tr>
<tr><td>Löschen</td><td colspan="3">Löschkonflikte (4.3.3.3)</td></tr>
<tr><td>Ändern</td><td colspan="3">Änderungskonflikte (4.3.3.4)</td></tr>
</table>

Tabelle 4.1: Konfliktklassen

In den folgenden Abschnitten werden die einzelnen Konfliktklassen näher untersucht. Die Konflikterkennung beruht dabei im wesentlichen auf den Prinzipien der datenorientierten Konflikterkennung aus Abschnitt 4.2, d.h. auf dem Vergleich der Primärschlüssel und Tupel-Abbilder BI und CI. Die beispielhaft angegebenen Möglichkeiten zur Konfliktauflösung werden am Ende dieses Abschnitts nochmal zusammengefaßt.

4.3.3.1 Konflikte durch Verwendung veralteter Daten

Durch die Möglichkeit, gleiche Daten auf verschiedenen Clients zu replizieren und zu ändern, kann nur der erste reintegrierende mobile Client seine Änderungen bezüglich einer gemeinsamen Datenmenge X konfliktfrei einbringen. Für nachfolgende Clients hat sich durch die eingebrachten Änderungen des ersten Client der aktuelle Datenbestand CI_X^S des Server verändert und die Reintegration erzeugt einen Konflikt, da die unverbunden durchgeführten Änderungen auf dem replizierten Abbild der Daten (BI_X^C) und somit auf mittlerweile veralteten Daten beruhen.

Dies betrifft insbesondere alle Fälle des zugrundegelegten Reintegrationsszenarios zweier Clients in Abbildung 4.2, bei denen Client 1 eine Schreiboperation (`INSERT`, `UPDATE`,

`DELETE`) auf eine Datenmenge X durchgeführt hat und Client 2 zur Vermeidung eines *Blind Write* lokal mit einem lesenden Zugriff vor weiteren Operationen beginnt. Eine Veränderung der Tupelmenge, durch Einfügungen oder Löschungen von Client 1, kann als Resultat einer Anfrage dabei ebenso das Ergebnis einer Transaktion beeinflussen wie der Wert eines konkreten veränderten Tupels. Ergeben sich beispielsweise je nach Wert eines Attributes Veränderungen im Anwendungs- bzw. Nutzerverhalten durch Verzweigung der Ablaufstruktur mittels Konstrukten wie `IF-THEN-ELSE`, so kann die Reintegration solcher Nutzeraktionen auf veränderten Daten ein völlig anderes Ergebnis liefern als auf dem mobilen Client. Problematisch hierbei ist jedoch, daß die Anwendungen des Client nicht vollständig erneut ausgeführt werden knnen. Die triviale Auflösung solcher Konflikte durch Verwerfen konfligierender Änderungen wurde bereits vorgestellt, andere Varianten durch Nutzung der Transaktionssemantik werden im Abschnitt 4.4 aufgezeigt.

4.3.3.2 Einfügekonflikte

	Einfügen von X durch Client 1		
	Einfügen von X'	Löschen	Ändern
Konflikt	Tupel X' verletzt die Schlüsseleigenschaft bzw. erzeugt doppelten Inhalt	n.a.	n.a.
Erkennung	Primärschlüsselvergleich ($\texttt{key}_X \stackrel{?}{=} \texttt{key}_{X'}$), Tupelinhaltsvergleich ($\texttt{cont}_X \stackrel{?}{=} \texttt{cont}_{X'}$)	n.a.	n.a.
Behandlung	z.B. `discard, rename key`	n.a.	n.a.
Beispiel	Zwei Nutzer des HERMES-Systems fügen unterschiedliche Daten zu denselben Sehenswürdigkeiten ein	n.a.	n.a.

Tabelle 4.2: Einfügekonflikte

Grundlage von Einfügekonflikten ist ein von Client 1 bereits erfolgreich eingebrachtes neues Tupel X (Tabelle 4.2). Darauf aufbauend werden nun alle Möglichkeiten betrachtet, in denen Client 2 nicht konfliktfrei synchronisieren kann. Hierbei sind Änderungen und Löschungen auf Tupel X nicht anwendbar (n.a.), da nach Voraussetzung des Konfliktszenarios beide Clients den gleichen Datenbestand repliziert haben. Somit konnte Tupel X auf Client 2 während der unverbundenen Arbeit nicht vorhanden sein, während es gleichzeitig auf Client 1 erfolgreich eingefügt wurde.

Client 1	Client 2
`INSERT INTO produkte (id, preis)` `VALUES (0815, 5.50);`	`INSERT INTO produkte (id, preis)` `VALUES (0815, 4.00);`

Tabelle 4.3: Beispiel für einen Einfügekonflikt

Dagegen kommt es zu einem Konflikt, wenn Client 2 ein Tupel X' mit $\texttt{key}_X = \texttt{key}_{X'}$ eingefügt hat und diese Änderung nun reintegrieren will. Tabelle 4.3 zeigt beispielhaft für das mobile Verkaufsszenario das Einfügen eines neuen Artikels mit dem Primärschlüssel

`id=0815` durch zwei verschiedene Clients mit jeweils unterschiedlichen Preisen. Hierbei lassen sich nun verschiedene Fälle unterscheiden:

- $\texttt{key}_X = \texttt{key}_{X'}, \texttt{cont}_X = \texttt{cont}_{X'}$
  Beide Tupel sind vollkommen identisch, d.h. sie besitzen den gleichen Primärschlüssel und den gleichen Inhalt. Dann genügt es zur Konfliktlösung die Einfügung von Client 2 zu ignorieren (`discard`).

- $\texttt{key}_X = \texttt{key}_{X'}, \texttt{cont}_X \neq \texttt{cont}_{X'}$
  Tupel X und Tupel X' besitzen den gleichen Primärschlüssel, haben allerdings einen unterschiedlichen Inhalt. Stellt der Primärschlüssel einen künstlichen Schlüssel dar, so kann zur Auflösung des Konfliktes eine Umbenennung des Primärschlüssels von Tupel X' vorgenommen werden, wobei eventuell existierende Referenzen auf diesen Schlüssel aktualisiert werden müssen. Eine weitere Möglichkeit besteht darin, die zweite konfligierende `INSERT`-Anweisung in ein `UPDATE` auf das bestehende Tupel zu überführen.

- $\texttt{key}_X \neq \texttt{key}_{X'}, \texttt{cont}_X = \texttt{cont}_{X'}$
  Der Vergleich über den Primärschlüssel ergibt zwar zwei unterschiedliche Tupel, allerdings ist deren Inhalt identisch. Werden solche Situation ebenfalls als Konflikt angesehen, ist für deren Erkennung ein vollständiger Vergleich des einzufügenden Tupels X' mit allen in der Zieltabelle vorhandenen Tupeln nötig. Um diesen enormen Aufwand zu reduzieren, kann beispielsweise eine Prüfsumme (Hash-Funktion) eingesetzt oder die `UNIQUE`-Eigenschaft über entsprechende Attribute definiert werden. Hat man einen solchen Konflikt erkannt, kann auch hier als Lösung beispielsweise das Tupel X' mit Anpassung der Abhängigkeiten verworfen werden (`discard`).

4.3.3.3 Löschkonflikte

	Löschen von X durch Client 1		
	Löschen	Ändern	Einfügen
Konflikt	Zu löschendes Tupel wurde bereits gelöscht	Zu änderndes Tupel wurde bereits gelöscht	n.a.
Erkennung	Primärschlüssel von X existiert im CI_X^S nicht	Primärschlüssel von X existiert im CI_X^S nicht	n.a.
Behandlung	z.B. `discard`	z.B. `new insert`	n.a.
Beispiel	Zwei Nutzer im HERMES-System bringen die Schließung eines Restaurants als Löschung ein	Ein Nutzer im HERMES-System bringt die Schließung eines Restaurants als Löschung ein, während ein zweiter Nutzer den neuen Besitzer als Änderung einbringt	n.a.

Tabelle 4.4: Löschkonflikte

Hat Client 1 das Tupel X gelöscht und diese Änderung erfolgreich in der Datenbank konsolidiert, so entstehen Konflikte in allen Fällen, in denen Client 2 auf das bereits gelöschte Tupel zugreift, wobei hier nur der ändernde Zugriff betrachtet wird, wie in Tabelle 4.4 dargestellt.

- Löschen-Löschen-Konflikt
  Ein Konflikt bezüglich einer Löschoperation auf Tupel X durch Client 2 wird dadurch erkannt, daß der Primärschlüssel des zu löschenden Tupels im aktuellen Datenbestand des Server (CI_X^S) nicht mehr existiert, da es schon von Client 1 gelöscht wurde. Die einzige Lösung für diesen Konflikt besteht im Verwerfen (`discard`) der zweiten Löschung.

- Löschen-Ändern-Konflikt
  Will Client 2 eine Änderung am Tupel X einbringen und wurde dieses bereits gelöscht, wird dies wieder anhand des nicht mehr existierenden Primärschlüssels erkannt. Allerdings ist die Lösung hier nicht mehr trivial. Abhängig von der Anwendungssemantik gibt es prinzipiell zwei Möglichkeiten:

  – Die Änderung kann nicht eingebracht werden und wird verworfen (`discard`). Diese Variante könnte beispielsweise dann verwendet werden, wenn Client 2 eine geringere Priorität hat als Client 1.

  – Die Änderung soll integriert werden. Da jedoch das zugrundeliegende Tupel nicht mehr existiert, muß die Änderung als neues Tupel wieder eingefügt werden (`new insert`). Allerdings sollte hierbei genau beachtet werden, inwieweit dadurch die Semantik einer Anwendung gefährdet wird.

Das Einfügen von Tupel X auf Client 2 würde aufgrund der Annahme gemeinsam replizierter Datenbestände die Eindeutigkeit des Primärschlüssels verletzen und ist deswegen nicht anwendbar.

4.3.3.4 Änderungskonflikte

Ähnlich wie bei den Löschkonflikten im Abschnitt 4.3.3.3 hat eine Änderung von Tupel X durch Client 1 nur für Lösch- und Änderungsoperationen von Client 2 auf demselben Tupel problematische Auswirkungen (Tabelle 4.5). Das Einfügen von Tupel X auf Client 2 ist bei gleichzeitiger Änderung auf Client 1 und vorausgesetzter gleicher Replikationsmenge nicht möglich.

- Ändern-Löschen-Konflikt
  Auf den ersten Blick erscheint dieser Fall nicht kritisch, da das zu löschende Tupel X noch existiert. Allerdings wurde durch die Änderung von Client 1 das Tupel verändert, sodaß ein Vergleich von $BI_X^{C_2}$ und CI_X^S keine Übereinstimmung ergibt. Damit basiert die Löschung von Client 2 noch auf der mittlerweile veralteten Version von X. Zur Lösung bieten sich, je nach Semantik der Anwendung, zwei Möglichkeiten an. Zum einen kann die Löschung verworfen werden (`discard`), andererseits trotz inkorrekter Voraussetzungen durchgeführt werden. Für eine Entscheidung zwischen den beiden Varianten könnten Prioritäten von Nutzern verwendet werden.

	Ändern von X durch Client 1		
	Löschen	Ändern	Einfügen
Konflikt	Die Löschung hat veraltete Version von X als Grundlage	Die Änderung hat veraltete Version von X als Grundlage	n.a.
Erkennung	Vergleich von BI_X^C des Client 2 und CI_X^S des Server	Vergleich von BI_X^C des Client 2 und CI_X^S des Server	n.a.
Behandlung	z.B. `discard`	z.B. `priority, udf`	n.a.
Beispiel	Ein Nutzer im HERMES-System bringt die Schließung eines Restaurants als Veränderung des Besitzers ein, während ein zweiter Nutzer die Schließung als Löschung einbringt	Zwei Nutzer bringen unterschiedliche Besitzer für das gleiche Restaurant im HERMES-System ein	n.a.

Tabelle 4.5: Änderungkonflikte

- Ändern-Ändern-Konflikt
  Wurden durch Client 1 und Client 2 zwei verschiedene Änderungen am Tupel X vorgenommen, wie in Tabelle 4.6 dargestellt, so kann Client 2 nach Voraussetzung nicht konfliktfrei reintegriert werden, da $BI_X^{C_2} \neq CI_X^S$. Zur Konfliktauflösung stehen die in Abschnitt 4.2 vorgestellten Möglichkeiten der datenorientierten Behandlung zur Verfügung, d.h. die Änderung von Client 2 zu verwerfen oder die Änderung von Client 1 zu überschreiben. Alternativ können auch je nach Anwendungssemantik verschiedene Funktionen wie `sum` oder `max` auf den beiden Werten des Attributes `preis` angewandt werden.

Client 1	Client 2

`UPDATE produkte`
`SET preis = 5.50`
`WHERE id = 0815;`
	`UPDATE produkte`
	`SET preis = 6.00`
	`WHERE id = 0815;`

Tabelle 4.6: Beispiel für einen Änderungskonflikt

4.3.4 Konfliktauflösung

Für die automatische Konfliktauflösung stehen neben den Auflösungsmöglichkeiten der datenorientierten Konfliktbehandlung aus Abschnitt 4.2 noch folgende Möglichkeiten zur Verfügung:

Alternativen: Erzeugt ein einzufügendes oder geändertes Tupel einen Konflikt mit dem aktuellen Datenbestand, so besteht die Möglichkeit, ein Tupel mit alternativen Daten zu verwenden, um den Konflikt zu lösen. Allerdings müssen diese Alternativen

semantisch sinnvoll im Laufe der Anwendung erzeugt werden. Verwendung findet diese Variante vorallem bei Anwendungen wie Terminplaner, die für einen Termin (Datum, Zeit) mehrere Orte zur Auswahl zulassen, um somit beispielsweise konfligierende Raumbelegungen aufzulösen.

Überführung von `UPDATE` zu `INSERT`: Diese Auflösungsvariante speziell für Löschkonflikte kann dann eingesetzt werden, wenn eine lokale Änderung (`UPDATE`) nicht mehr eingebracht werden kann, weil das zugrundeliegende Tupel mittlerweile aus der Datenbank gelöscht wurde. Soll die Änderung dennoch integriert werden, kann dies als erneute Einfügung (`INSERT`) vorgenommen werden, wenn es die Anwendungssemantik zuläßt.

Überführung von `INSERT` zu `UPDATE`: Ähnlich einer Operationstransformation bei der Neueinfügung von Tupeln kann hier ein `INSERT`, welches ein Tupel mit bereits vorhandenem Primärschlüssel einzufügen versucht, zu einem `UPDATE` des bestehenden Tupels umgeformt werden. Auch hierbei ist zu prüfen, inwiefern dies der zugrundeliegenden Anwendungssemantik entspricht. Derartige Festlegungen sind, wie in unserem Ansatz in Kapitel 6 beschrieben, beim Design der Anwendung durch den Administrator zu definieren.

Mehrversionsverfahren: Eine etwas komplexere Variante der Auflösung von Konflikten bieten Konzepte von Mehrversionsverfahren [GB94]. Diese unterscheiden sich von anderen Methoden in der zeitweiligen Führung von verschiedenen Versionen eines Datenelementes. Dadurch lassen sich Lesetransaktionen von Schreibtransaktionen entkoppeln, indem Änderungsoperationen neue Versionen von Datenelementen erzeugen und Leseoperationen nicht immer auf der aktuellsten Version stattfinden. Zum einen muß dabei die für eine Transaktionsanfrage passende Version eines Datenobjektes bestimmt werden, andererseits müssen die erzeugten Versionen so verwaltet werden, daß nicht mehr benötigte Versionen verworfen werden können. Aufgrund der algorithmischen Komplexität dieses Ansatzes, wird auf eine ausführliche Beschreibung verzichtet. Detailiertere Informationen sind in [HR01] zu finden, eine Bewertung des Mehrversionsverfahren wurde in [Lie01] gegeben.

4.4 Transaktionsorientierte Konfliktbehandlung

4.4.1 Konfliktdefinition

Zwei Transaktionen T_1 und T_2 stehen in Konflikt zueinander, wenn sie nach der Konfliktdefinition in Abbildung 3.1 gleichzeitig auf dasselbe Datenobjekt X zugreifen, wobei mindestens einer der beiden Zugriffe schreibend ist. Um zu erkennen, ob Transaktionen konfligierend auf gleiche Datenelemente zugegriffen haben, können die im Kapitel 2 definierten Mengen RS und WS jeweils miteinander verglichen werden. Ein Konflikt zwischen zwei Transaktionen T_1 und T_2 verschiedener Clients besteht, wenn gilt:

$$RS(T_1) \cap WS(T_2) \neq \emptyset \quad \text{oder} \quad RS(T_2) \cap WS(T_1) \neq \emptyset$$

Dabei lassen sich folgende Konflikte genauer unterscheiden, wobei einige mit bekannten Anomalien und Inkonsistenzen aus dem klassischen Datenbankbetrieb verglichen werden. Grundlage hierfür ist das Szenario in Abbildung 4.5.

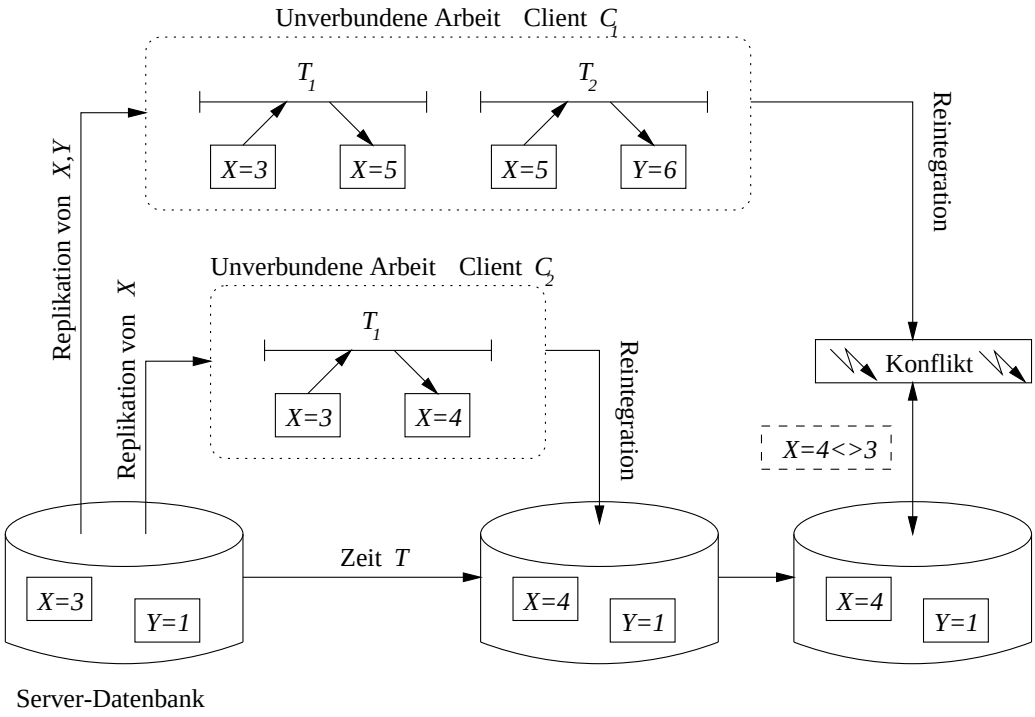


Abbildung 4.5: Klassische Anomalien beim mobilen Datenbankzugriff

4.4.1.1 Schreib-Lese-Konflikte (*Dirty Read*)

Eine lokale Transaktion T_2 des Client C_1 liest ein Objekt X, welches vorher von einer anderen lokalen, aufgrund der Unverbundenheit allerdings nur vorläufig abgeschlossenen, Transaktion T_1 desselben Client geschrieben wurde. Dies führt für Transaktion T_2, wie in Abbildung 4.5 veranschaulicht, zur Anomalie *Dirty Read*, da Transaktion T_1 beim Reintegrieren des Client C_1 aufgrund eines Konfliktes abgewiesen wird.

4.4.1.2 Lese-Schreib-Konflikte (*Unrepeatable Read*)

Die Transaktion T_1 des Client C_1 liest zu Beginn der unverbundenen Arbeit den Wert von X. Parallel dazu ändert Client C_2 diesen Wert auf $X = 4$ und reintegriert erfolgreich. Bei der Reintegration von Client C_1 werden lokale Transaktionen nach dem zugrundegelegten Two-Tier-Prinzip [GHOS96] erneut ausgeführt. Insbesondere kann die Leseoperation von Transaktion T_1 als wiederholtes Lesen innerhalb der langlaufenden Transaktion, d.h. der unverbundenen Arbeit von Client C_1, betrachtet werden, wobei es zur Anomalie des *Unrepeatable Read* kommt.

4.4.1.3 Schreib-Schreib-Konflikte (*Lost Update*)

Dieser Konflikt entsteht immer dann, wenn lokale Änderungen eines Client aufgrund von Konflikten und durch eine `server wins`-Regel nicht reintegriert werden können. Beispielsweise gehen in Abbildung 4.5 die Änderungen von C_1 verloren, weil diese auf einem veralteten Wert von X basieren, der mittlerweile durch Client C_2 geändert wurde.

4.4.1.4 Konflikte durch Integritätsverletzung

Zur Verletzung einer Integritätsbedingung kann es im einfachsten Fall kommen, wenn beim Replizieren von Daten eine zugehörige Bedingung B nicht mit repliziert wird. Dann unterliegen die unverbunden durchgeführten Änderungen keiner Integritätssicherung bezüglicher dieser Bedingung B. Als Lösung wäre die Replikation aller nötigen Integritätsbedingungen vorstellbar. Dennoch kann es beim Reintegrieren zu einer Verletzung der Integrität kommen, wenn im Zusammenhang mit anderen, nicht replizierten Daten, gewisse Bedingungen nicht efüllt sind.

Integritätsbedingung: $a + b > 0$	
Client 1	Client 2
Repliziere $a = 5, b = 4$	Repliziere $a = 5, b = 4$
Ändere $a = -2$	Ändere $b = -3$
Test$(-2 + 4 > 0)$=`TRUE`	Test$(5 - 3 > 0)$=`TRUE`
Test$(-2 - 3 > 0)$=`FALSE`	

Tabelle 4.7: Integritätsverletzung trotz vollständiger Replikation

Die schließlich letzte Möglichkeit, Konflikte dieser Art in dem hier zugrundegelegten Modell zu vermeiden, beinhaltet zusätzlich die Replikation aller Daten einer Datenbank. Abgesehen von der technisch unmöglichen Realisierung dieser Forderung für mobile Kleinstgeräte, kann man zeigen, daß es jedoch selbst bei vollständiger Replikation zur Verletzung einer Integritätsbedingung kommen kann, wie es das Beispiel in Tabelle 4.7 veranschaulicht. Trotz lokaler Erfüllung der Integrität ($a + b > 0$ sei die einzige Integritätsbedingung in der Datenbank), vollständiger Replikation (a und b seien die einzigen Daten) und disjunkter Änderung, liegt beim Synchronisieren beider Clients eine Verletzung der Integrität vor.

4.4.1.5 Konflikte durch Abhängigkeiten

Betrachtet man eine Folge von lokalen Transaktionen eines mobilen Client, dann entstehen gewisse Abhängigkeiten zwischen ihnen, wenn Transaktionen die vorläufigen Ergebnisse vorangegangener Transaktionen nutzen. Werden nun beim Reintegrieren lokale Transaktionen aufgrund von Konflikten zurückgewiesen oder mit einem anderen Ergebnis eingebracht, so ist für alle in Abhängigkeit stehenden Transaktionen des Client zu prüfen, ob sie möglicherweise ebenfalls zurückgewiesen werden müssen, es kommt zum *Cascading Reject* [1]. Ähnlich können lokale Änderungen oder Löschungen beim Reintegrieren zu weiteren Änderungen bzw. Löschungen führen, wenn Hierarchiebeziehungen (referentielle Integrität) zwischen Tabellen in der Datenbank existieren.

Sei für ein Beispiel im HERMES-System die Situation in Abbildung 4.6 betrachtet. Dabei steht die Tabelle `Bewertungen` in einer Fremdschlüsselbeziehung zur Tabelle `Restaurants`, d.h. Bewertungen für ein Restaurant können nur für bereits existierende Restaurants abgegeben werden. Hat nun ein Client im HERMES-System die Tabelle `Restaurants` repliziert, die Tabelle `Bewertungen` dagegen nicht, so führen unverbunden durchgeführte

[1] In Anlehnung an den Begriff des *Cascading Abort* im klassischen zentralisierten Datenbankbetrieb.

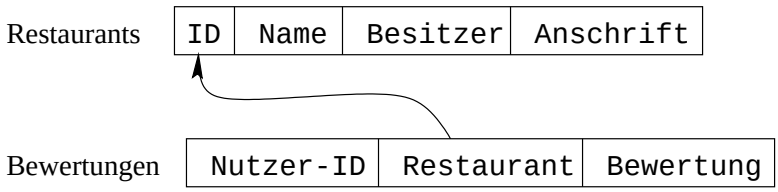


Abbildung 4.6: Beispiel für eine Fremdschlüsselbeziehung

Löschungen von Tupeln der `Restaurant`-Tabelle beim Synchronisieren zu Löschungen von Tupeln innerhalb der Tabelle `Bewertungen`, wenn diese Abhängigkeit mit dem SQL-Konstrukt `CASCADING DELETE` definiert wurde. Zum Konflikt kommt es hierbei, wenn solche Löschungen von der „Wurzel“ beginnend nicht erlaubt sind, beispielsweise bei Verwendung des SQL-Konstrukts `RESTRICT`. Es erzwingt, daß für eine Löschung von Tupeln der `Restaurants`-Tabelle die über einen Fremdschlüssel abhängigen Tupel der Tabelle `Bewertungen` zuvor gelöscht werden müssen. Die einzige Lösungsmöglichkeit besteht in der Abweisung solcher konfligierender Löschungen. Zur Vermeidung dieser Konflikte können entsprechende Hierarchie-Beziehungen bei der Replikation beachtet werden, wofür das in [Gol02] vorgestellte Konzept der Fragmente verwendet werden kann.

4.4.2 Konflikterkennung

Bei der transaktionsorientierten Konflikterkennung, d.h. einer Reintegration unter Beachtung des Transaktionskontextes, steht die Aufzeichnung des RS und WS als nötige Zusatzinformation für die Synchronisation im Mittelpunkt. Allerdings werden dabei nur die Primärschlüssel und nicht die eigentlichen Inhalte der Daten protokolliert und ausgewertet, wie im Kapitel 2 beschrieben. Voraussetzung ist hierbei, daß eine Transaktion vor einer Änderung das entsprechende Datum gelesen hat ($WS \subseteq RS$), d.h. es gibt keine sogenannten *Blind Writes*. Weiterhin müssen Informationen über die Transaktionsreihenfolge, beispielsweise in Form von Zählern oder Zeitstempeln, für die Auswertung vorliegen. Zur Erkennung von Abhängigkeitskonflikten zwischen Transaktionen könnte für replizierte Daten auf dem Client protokolliert werden, ob sie bereits von einer lokalen Transaktion T_1 verändert wurden. Greift eine weitere lokale Transaktion T_2 auf diese veränderten replizierten Daten zu, ergibt sich eine Abhängigkeit der Transaktion T_2 von T_1, welche dem RPS als Information zur Verfügung gestellt werden kann. Nach [HR01] kann man nun allgemein zwei Möglichkeiten unterscheiden, einzubringende Transaktionen entsprechend einer Serialisierungsfolge zu synchronisieren.

4.4.2.1 BOCC-Validierung

Bei der rückwärtsgerichteten Synchronisation (*Backward oriented Optimistic Concurrency Control, BOCC*) wird eine zu validierende Transaktion T nur mit schon abgeschlossenen Transaktionen T_i auf Konflikte untersucht. Dazu wird überprüft, ob ein Datenelement $X \in RS_T$ existiert, welches von einer Transaktion geschrieben wurde ($X \in WS_{T_i}$), die während der Ausführung von T abgeschlossen wurde. Das komplette Protokoll zur Validierung von T ist in Abbildung 4.7 dargestellt.

```
valid := TRUE;
FOR ALL T_i (T_i während der Ausführung von T beendet) DO
    IF (RS(T) ∩ WS(T_i) ≠ ∅) THEN valid := FALSE
    END IF;
END DO;
IF valid THEN Schreibphase(T)
         ELSE Reject(T)[2]
END IF;
```

Abbildung 4.7: Validierung von T nach dem BOCC-Verfahren

Im klassischen Datenbankszenario hat die BOCC-Validierung zum einen den Nachteil, daß Transaktionen unnötigerweise zurückgesetzt werden müssen. Dies gilt beispielsweise für Transaktion T_1 in Abbildung 4.8, da T_1 das Element $x \in WS(T_2)$ gelesen hat, allerdings erst nach der Schreiboperation von T_2 und damit den aktuellen Wert von x. Abhilfe schafft hier das sogenannte *BOCC+*-Verfahren, welches mit Versionsnummern arbeitet.

Zum anderen besteht die Gefahr, daß Transaktionen wiederholt abgewiesen werden können (*Starvation*). Dies ist vorallem für langlaufende bzw. lokale Transaktionen im mobilen Umfeld der Fall, weil diese durch die oft lange Zeit der Unverbundenheit mit vielen Transaktionen validiert werden müssen.

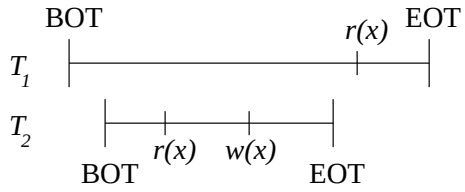


Abbildung 4.8: Probleme beim BOCC-Verfahren

4.4.2.2 FOCC-Validierung

Im Gegensatz zum BOCC-Verfahren wird bei der vorwärtsorientierten Synchronisation (*Forward oriented Optimistic Concurrency Control, FOCC*) eine Transaktion nur gegen aktive Transaktionen validiert. Dabei wird festgestellt, ob eine der laufenden Transaktionen ein Objekt gelesen hat, welches die zu validierende Transaktion schreiben will. Nur in diesem Fall liegt ein Konflikt vor und eine oder mehrere der beteiligten Transaktionen (aktive oder zu validierende) müssen korrekterweise zurückgesetzt werden. Der vollständige Algorithmus zur FOCC-Validierung ist in Abbildung 4.9 dargestellt.

Für den Einsatz in dem hier zugrundegelegten mobilen Datenbankszenario ist diese Art der Validierung allerdings ungeeignet, da es wie die BOCC-Validierung erst zum Zeitpunkt der Synchronisation eines mobilen Client verwendet werden kann und dann im allgemeinen andere mobile Clients nicht ebenfalls zum Server verbunden sind. Damit ist die Ermittlung

[2] Im klassischen festverbundenen Datenbankbetrieb wird stattdessen die bereits gestartete Transaktion T mittels `ROLLBACK` zurückgerollt.

```
valid := TRUE;
FOR ALL T_i (T_i wird ausgeführt) DO
    IF (RS(T) ∩ WS(T_i) ≠ ∅) THEN valid := FALSE
    END IF;
END DO;
IF valid THEN Schreibphase(T)
         ELSE Konfliktlösung
END IF;
```

Abbildung 4.9: Validierung von T nach dem FOCC-Verfahren

aller aktuell laufenden Transaktionen mobiler Clients nicht möglich. Es bleibt daher nur die BOCC-Validierung, welche aus oben genannten Gründen allerdings auch eine hohe Abbruchrate aufweist, da es für lange unverbundene Phasen zwangsläufig zur vermehrten Abweisungen von lokalen Transaktionen kommt.

4.4.3 Konfliktauflösung

Für die automatische Konfliktauflösung stehen folgende Möglichkeiten zur Auswahl:

Isoliertes Abweisen einer Transaktion: Erzeugt eine zu reintegrierende Transaktion mit den aktuellen Daten einen Konflikt, wird sie vollständig zurückgewiesen, um die Atomaritätseigenschaft zu gewährleisten. Dadurch gehen alle Änderungen dieser Transaktion verloren. Abhängigkeiten anderer Transaktionen werden, wie beispielsweise auch beim IBM DB2 DataPropagator [IBM02], nicht beachtet.

Kaskadierendes Abweisen einer Transaktion: Muß eine Transaktion aufgrund von Konflikten abgewiesen werden, so geschieht dies hier ebenfalls für alle abhängigen Transaktionen. Dadurch entstehen sogenannte *Abhängigkeits-Cluster*, d.h. Gruppen von untereinander abhängigen Transaktionen. Einsatz könnte diese Möglichkeit für Anwendungen haben, deren Ausführung sich über mehrere, voneinander abhängige, Transaktionen erstreckt und nur semantisch sinnvoll ist, wenn entweder alle oder keine der zugehörigen Transaktionen korrekt ausgeführt bzw. reintegriert werden können.

Transaktion mit anderem Ergebnis: Wird eine Transaktion trotz Konfliktsituation eingebracht, kann die dafür notwendige Wiederausführung auf den aktuellen Daten zu einem veränderten Transaktionsergebnis führen. Dies muß innerhalb der zugrundeliegenden Anwendungssemantik gültig sein, was beispielsweise im Bayou-System [GHOS96] durch ein sogenanntes *Akzeptanzkriterium* bzw. in MobiSnap [PBM+00] durch die *mobile Transaktion* realisiert wird.

Verletzung der Atomarität: Eine letzte Möglichkeit für die Reintegration konfligierender Transaktionen besteht darin, durch Auflösung der Transaktionsgrenzen soviele Operationen wie möglich einzubringen. Dadurch wird allerdings die Atomaritätseigenschaft verletzt.

Kapitel 5

Produktbeispiele und Forschungsansätze

Dieses Kapitel soll einen Überblick geben, inwieweit DBS-Produkte und ein ausgewähltes Forschungsprojekt mobile Clients unterstützen. Die hierbei untersuchten kommerziellen Systeme verwenden ebenfalls das im Kapitel 2 beschriebene und hier zugrundegelegte Arbeitsmodell mit seinen bereits vorgstellten Eigenschaften. Trotz der Tatsache, daß das Thema Mobilität im Zusammenhang mit Datenbanken erst seit einigen Jahren näher erforscht und betrachtet wird, bieten alle größeren Hersteller von Datenbankmanagementsysteme entsprechende Funktionalitäten an, wie die Arbeit von [Fan00] zeigt.

In dieser Arbeit werden beispielhaft zwei verschiedene Produktansätze vorgestellt. Dies ist zum einen als typisches mobiles Produkte das im Abschnitt 5.1 beschriebene *Oracle9i Lite* von Oracle. Andererseits wird im Abschnitt 5.2 auf den *DB2 DataPropagator 8* von IBM eingegangen, der als klassisches Replikations-Tool auch für mobile Datenbanken verwendet werden kann. Dem gegenüber steht das Forschungsprojekt *MobiSnap*, dessen Ziele und Inhalte im Abschnitt 5.3 näher erläutert werden.

5.1 Oracle9*i* Lite

5.1.1 Einführung und Architektur

Das für den mobilen Einsatz entwickelte Produkt Oracle9*i* Lite [Ora02a] umfaßt unter anderem Funktionalitäten zur Entwicklung und Implementierung von Anwendungen für die unverbundene Arbeit. Diese Anwendungen greifen über das sogenannte *Oracle Lite RDBMS*, ein für mobile Endgeräte optimiertes und speicherminimiertes relationales DBMS, auf replizierte Daten eines Client zu. Dafür stehen die Sprachelemente von *Mobile SQL (MSQL)* zur Verfügung. Die Aufbereitung der lokalen Änderungen zur Synchronisation übernimmt das *Mobile Sync Module*. Die eigentliche datenorientierte Reintegration der Daten sowie die Konfliktbehandlung ist Aufgabe des zwischen mobilen Client und Datenbankserver geschalteten *Mobile Server*, auf dessen Funktionalität in den folgenden Abschnitten eingegangen wird. Abbildung 5.1 zeigt die Architektur der einzelnen zu Oracle9*i* Lite gehörenden Komponenten.

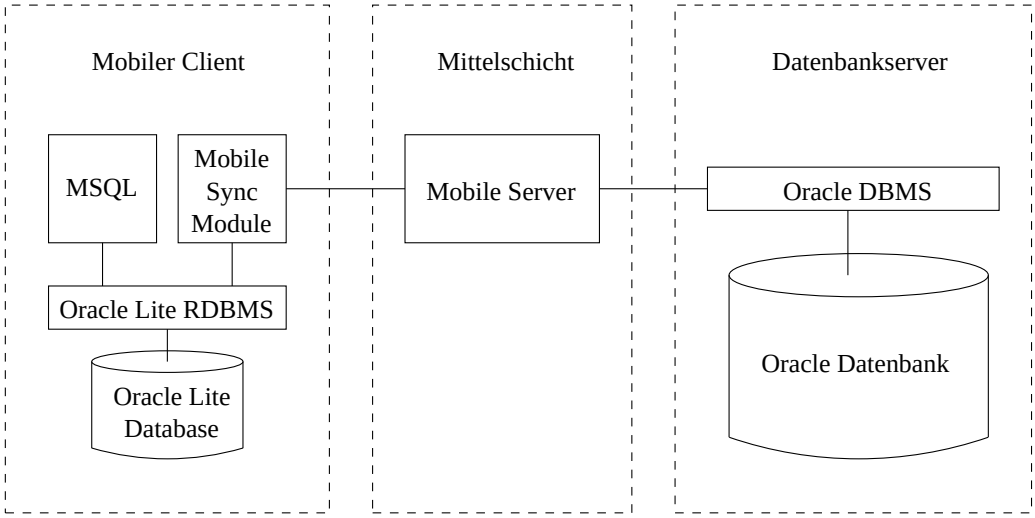


Abbildung 5.1: Architektur von Oracle9*i* Lite

Durch ein *Publish & Subscribe*-Verfahren werden für jede Anwendung eines mobilen Client die notwendigen Daten über sogenannte *Publikationen* definiert. Eine Publikation ist vergleichbar mit einem Datenbankschema und kann mehrere Publikationselemente enthalten, die wie eine parametrisierte Sicht einen bestimmten Ausschnitt der Serverdaten darstellen. Zusätzlich lassen sich auf prozedurale Weise einige Eigenschaften zu den Publikationselementen definieren, wie in Abbildung 5.2 dargestellt.

```
public static void AddPublicationItem
    (String publication,
     String item,
     String columns,
     String disabled_dml,
     String conflict_rule,
     String restricting-predicate,
     String weight) throws Throwable
```

Abbildung 5.2: Syntax der Anweisung `AddPublicationItem`

Folgende Parameter beeinflußen die Konfliktbehandlung eines zugrundeliegenden Publikationselementes, eine genauere Beschreibung möglicher Werte findet sich in [Ora02a].

`disabled_dml`
: Über diesen Parameter kann festgelegt werden, welche Änderungsoperationen auf den zugehörigen Publikationsdaten möglich sind. Dabei reicht das Spektrum von vollständig änderbar über nur lesenden Zugriff bis zur Freigabe einzelner Operationen (`INSERT`, `UPDATE`, `DELETE`).

`conflict_rule`
: Hier wird eine der beiden später erläuterten Konfliktauflösungen festgelegt.

5.1.2 Konfliktvermeidung

Für die Vermeidung von Konflikten sind in Oracle9*i* Lite keine expliziten Methoden vorgesehen, dies wird durch eine entsprechende Gestaltung der Anwendungen dem Administrator überlassen. Dabei ist eine Sperrung der replizierten Daten nicht möglich, um die Verfügbarkeit bei einer Vielzahl von Clients nicht zu gefährden.

5.1.3 Konflikterkennnung

Folgende Konflikte werden durch den *Mobile Server* automatisch erkannt:

Eindeutigkeitskonflikt
Ein Eindeutigkeitskonflikt wird erkannt, wenn eine Verletzung der Eindeutigkeitsbedingung für Primärschlüssel oder `UNIQUE`-Definition während eines `INSERT` oder `UPDATE` bezüglich eines replizierten Tupels auftritt.

Änderungskonflikt
Ein Änderungskonflikt wird erkannt, wenn es Unterschiede zwischen dem alten Wert vor der Replikation (BI_X^C) und dem aktuell auf dem Server vorliegenden Wert (CI_X^S) für ein Tupel X gibt, verursacht durch ein konkurrierendes `UPDATE` auf Client und Server. Zur Erkennung ist beim Synchronisieren der Änderungen des mobilen Client der alte Wert des Tupels mitzusenden. Stimmen dagegen alter Wert und aktueller Wert eines Tupels überein, wird der neue Wert AI_X^C dem entsprechenden Tupel zugewiesen.

Löschkonflikt
Ein Löschkonflikt wird erkannt, wenn kein Tupel für die einzubringenden Operationen `UPDATE` oder `DELETE` aufgrund des fehlenden Primärschlüssels gefunden wird.

Synchronisationsfehler
Hierunter versteht Oracle alle anderen Konflikte, die eine Verletzung bestehender Integritätsregeln darstellen, wie beispielsweise von Fremdschlüsselbeziehungen und `NOT NULL`-Bedingungen für Attribute.

Zur Datenidentifizierung nutzt Oracle seit einigen Produktversionen den Primärschlüssel. Besitzt eine Tabelle keinen Primärschlüssel, so muß vom Nutzer ein eindeutiger Schlüssel vorgegeben werden. Anwendungen sollten keine Änderungen an diesen Schlüsseln vornehmen dürfen, um die durch Oracle überwachte Integrität der Daten zu gewährleisten. Zusätzlich zum Primärschlüssel verwendet der *Mobile Server* eine interne Versionierung, um Konflikte zu erkennen. Dabei wird für jedes Tupel des Client bzw. des Server eine Versionsnummer verwaltet, die bei Nichtübereinstimmung zur regelbasierten Konfliktauflösung mittels sogenannter *Winning Rules* führt.

5.1.4 Konfliktauflösung

Alle durch konfligierende Operationen erzeugten Konflikte können mit Hilfe einer der folgenden Konfliktauflösungen automatisch auf dem *Mobile Server* gelöst werden:

client wins
: Dadurch werden alle Änderungen des mobilen Client auf dem *Mobile Server* eingebracht. Diese Konfliktlösungvariante kann angepaßt werden, indem zusätzliche `BEFORE INSERT/UPDATE/DELETE`-Trigger auf der betreffenden Tabelle definiert werden. Dieser vergleicht alte und neue Werte eines Tupels und behandelt entsprechend der definierten Trigger-Logik die Änderungen des Client.

server wins
: Gewinnt der *Mobile Server*, gehen die Änderungen des Client verloren und es werden entsprechende Änderungen zwischen den aktuellen Daten des *Mobile Server* und den Daten des Client zurückpropagiert.

Synchronisationsfehler werden dagegen nicht automatisch aufgelöst. Stattdessen wird die entsprechende Transaktion vom *Mobile Server* zurückgerollt bzw. abgewiesen und in eine sogenannte *Error Queue* eingefügt. Diese speichert Transaktionen, die aufgrund eines ungelösten Konfliktes fehlgeschlagen sind. Ein Administrator kann dann entweder durch eine manuelle Änderung der Daten von Transaktionen in der *Error Queue* oder der Daten des Server bestehende Konflikte auflösen.

5.2 DB2 DataPropagator Version 8

5.2.1 Einführung und Architektur

Ursprünglich als Komponente für die Replikation in verteilten Datenbankszenarien konzipiert, läßt sich der *DB2 DataPropagator* [IBM02] auch für die Replikation in sogenannten satellitenartigen Umgebungen einsetzen. Diese besteht aus mehreren Satellitendatenbanken, die jeweils als mobiles oder stationäres System realisierbar sind. Dabei ist der Einsatz für mobile Szenarien 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 Ziel- bzw. Replikattabellen angefordert (*Subscribe*) werden. 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.

Für die Ermittlung der Änderungen auf den Replikationsquellen liest *Capture* asynchron während der unverbundenen Arbeit das Log und speichert die aktuellen Änderungen in der sogenannten Änderungstabelle (*CD-Table, Change Data Table*) sowie abgeschlossene Transaktionen in der sogenannten UOW-Tabelle (*Unit Of Work*). Aus diesen Tabellen liest *Apply* und führt die entsprechenden Änderungen an Quell- und Zieltabellen durch. Dies geschieht dezentral und asynchron. Eine solche Synchronisation kann entweder nur für geänderte Daten (*Differential Refresh*) oder für alle Daten einer Tabelle (*Full Refresh*) erfolgen.

5.2.2 Konfliktvermeidung

Die Vermeidung von Konflikten wird ähnlich wie bei Oracle9*i* Lite dem Administrator bei der Gestaltung der Anwendungen überlassen, eine Verwendung von Sperren wird nicht angeboten. Allerdings werden folgende Konzepte zur Unterstützung vorgeschlagen:

- Fragmentierung nach Schlüssel:
  Die Anwendung wird so konzipiert, daß Änderungen von Daten der Replikatquelle in der konsolidierten Datenbank mit bestimmten Schlüsselbereichen nur von gewissen Clients (Replikatknoten) erzeugt werden können. Beispielsweise könnte eine Anwendung zur Erfassung von Produktumsätzen so gestaltet werden, daß ein Replikatknoten in New York nur die Verkaufsdaten der östlichen USA ändern kann, auf die Daten der westlichen USA aber dennoch lesend zugreifen kann. Für die Fragmentierung ist dabei beispielsweise der ZIP-Code (Postleitzahlen) gut geeignet.

- Fragmentierung nach Zeit:
  Eine weitere Möglichkeit zur Konfliktvermeidung, jedoch vorrangig für den Fall verteilter Datenbanken, besteht darin, die Anwendung so zu entwerfen, daß Änderungen an replizierten Daten zeitversetzt stattfinden, d.h. daß möglichst nur ein Replikatknoten zu einem bestimmten Zeitpunkt Änderungen an mehrfach replizierten Daten vornimmt und diese dann konfliktfrei konsolidieren kann. Grundlage hierfür kann beispielsweise die Aufteilung in Zeitzonen sein, wenn Änderungen nur in gewissen Zeitfenstern möglich sind.

5.2.3 Konflikterkennung

Operationsorientierte Konflikte zwischen einem Master und einem Replikatknoten in einer sogenannten *Update-Anywhere*-Umgebung werden von *Apply* dann erkannt, wenn bei der Einbringung einer Operation auf dem, durch den Primärschlüssel identifizierten, Tupel X dem ΔBI_X^S ein ΔBI_X^C gegenübersteht, d.h. wenn seit der letzten Synchronisation des Satelliten eine konkurrierende 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, d.h. die Auflösung der konfligierenden Änderungen des Client (Replikatknoten) erfolgt immer nach der Regel `server-wins`. Konflikte bezüglich `INSERT` und `DELETE` werden wie üblich durch Primärschlüsselvergleiche erkannt. Ein weiterer erkennbarer Konflikt ist die Verletzung einer Integritätsbedingung.

Für den Aufwand zur Konflikterkennung gibt es drei Level, der sich für jeden einzelnen Replikationsknoten individuell festlegen läßt. Bei einer Zusammenfassung von mehreren Replikationsknoten zu einer sogenannten *Subscribe*-Menge, verwendet *Apply* dann jedoch den höchsten aller innerhalb dieser Menge existierenden Level. Folgende Level zur Konflikterkennung von niedrig (`None`) bis hoch (`Enhanced`) stehen zur Auswahl:

- `None`:
  Es findet keine Konflikterkennung statt, sodaß konfligierende Änderungen zwischen der Mastertabelle und der Replikattabelle nicht erkannt werden. Diese Einstellung ist nur in Anwendungsfällen mit garantierter Konfliktfreiheit sinnvoll, um unnötige Performance-Belastungen zu vermeiden. Allgemein wird sie jedoch nicht für *Update-Anywhere*-Umgebungen empfohlen.

- `Standard`:
  Bei der normalen Konflikterkennung werden in jedem Zyklus von *Apply* jeweils die Schlüsselwerte der schon durch *Capture* erfaßten Tupel in den Änderungstabellen der Quelltabelle bzw. Zieltabelle miteinander verglichen. Tritt ein solcher Schlüsselwert in beiden Änderungstabellen auf, wurde seit der letzten Synchronisation eine Änderung an dem Tupel sowohl auf der Quelltabelle als auch auf der Zieltabelle (Replikat) vorgenommen und ein Konflikt ist erkannt. *Apply* fährt dann die zuvor abgeschlossene Transaktion auf dem Replikatknoten zurück und behält nur die Änderungen, die auf der Quelltabelle des Masterknoten erzeugt wurden.

- `Enhanced`:
  Die erweiterte Konflikterkennung arbeitet ähnlich wie die Konflikterkennung mit Level `Standard` durch Erkennung doppelter Schlüsselwerte in den Änderungstabellen. Allerdings werden hier alle Replikate innerhalb einer *Subscribe*-Menge gegen weitere Transaktionen durch *Apply* gesperrt und erst nach der Erfassung aller Änderungen, die vor der Sperrung durchgeführten wurden, beginnt *Apply* die Konflikterkennung durch Vergleich von Schlüsselwerten zwischen den Änderungstabellen. Die Behandlung von Konflikten erfolgt analog zu normalen Konflikterkennung. Diese Konflikterkennung soll laut [IBM02] die beste Datenintegrität zwischen den Quelldaten auf dem Master und den Replikaten bieten.

  Der `Enhanced`-Level ist jedoch nicht für mobile Umgebungen, in denen Clients nur sporadisch verbunden sind wie in dieser Arbeit zugrundegelegt, verfügbar und wird durch die normale Konflikterkennung ersetzt. Der Grund hierfür ist ähnlich wie bei der in Abschnitt 4.4 beschriebenen FOCC-Validierung, daß zum Synchronisationszeitpunkt eines mobilen Client nicht notwendigerweise alle anderen zur gleichen *Subscribe*-Menge gehörenden mobilen Clients ebenfalls im Netzwerk verbunden sind.

5.2.4 Konfliktauflösung

Die Standard-Konfliktauflösung führt ein Rollback der konfliktverursachenden Transaktion auf dem Replikatknoten durch und beläßt die entsprechende Transaktion in der UOW-Tabelle. Damit würde beim nächsten *Apply*-Zyklus versucht werden, die Transaktion erneut einzubringen. Eine endgültige Lösung wird dem Nutzer bzw. Administrator durch einen manuellen Eingriff überlassen, beispielsweise indem er auf dem Satellitensystem eine kompensierende Transaktion für die konflikterzeugende Transaktion findet bzw. durch eine nutzerdefinierte Behandlung für zurückgewiesene Transaktionen, die automatisch nach jeder Beendigung eines *Apply*-Zyklus startet. Allerdings werden keine Lese-Abhängigkeiten nachfolgender Transaktionen von einer zurückgewiesenen Transaktion durch *Apply* erkannt.

5.3 MobiSnap

5.3.1 Einführung und Architektur

Das MobiSnap-Projekt [PBM+00, PBM+99] wurde im Jahre 1999 an der *Universidade Nova de Lisboa* in Portugal begonnen und befindet sich derzeit noch in der weiteren Entwicklung. Ziele diese Projektes sind die Entwicklung eines mobilen Transaktionsmodells

und Konzepte zur Kontrolle der Abweichung zwischen replizierten Daten. Um einerseits die Arbeit für mobile Nutzer im unverbundenen Zustand zu ermöglichen und andererseits gewisse Forderungen in Form von später erläuterten *Reservierungen* an die Replikate stellen zu können, bietet MobiSnap sowohl die optimistische als auch pessimistische Replikation an. Eine durch Änderungskonflikte verschiedener Nutzer verursachte Konsistenzgefährdung wird datenorientiert betrachtet, d.h. die Divergenz von Replikaten ist auf deren Inhalt definiert. Weiterhin dient ein zentraler Datenbank-Server als Besitzer aller Primärkopien, wobei ein erweitertes Client-Server-Szenario [JHE99], wie in Abbildung 5.3 dargestellt, verwendet wird.

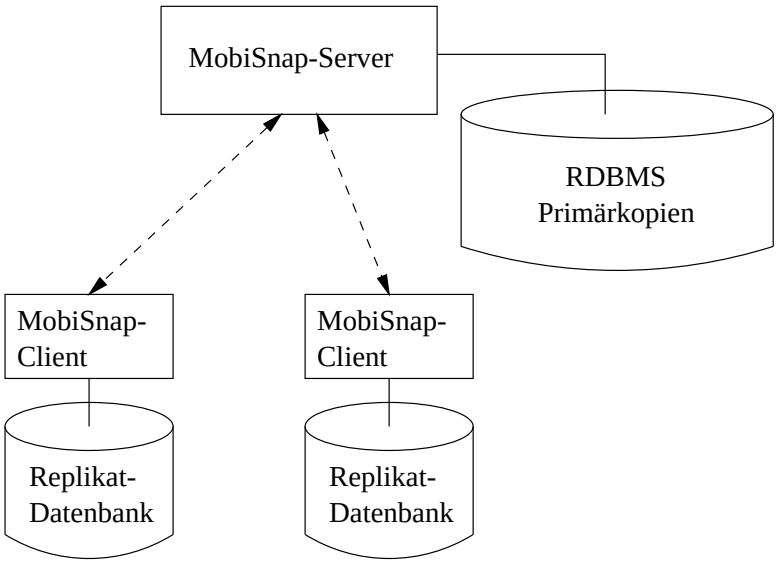


Abbildung 5.3: Erweitertes Client-Server-Szenario im MobiSnap-Projekt

Die Unterstützung der unverbundenen Arbeit wird in MobiSnap durch zwei Konzepte gewährleistet. Über sogenannte *mobile Transaktionen* kann ein Nutzer Änderungen des Datenbankzustandes mittels kleiner SQL-Programme durchführen. Diese werden in einer auf PL/SQL [Ora99] basierenden imperativen Programmiersprache geschrieben, die ein Spezifizieren der gewünschten Semantik einer Operation (Vorbedingung, Nachbedingung, Alternativen) abhängig vom aktuellen DB-Zustand ermöglicht. Damit ist für die klassische Konflikterkennung und eventuelle Auflösung die entsprechende mobile Transaktion, d.h. der Nutzer bzw. Administrator, verantwortlich. Abbildung 5.4 zeigt eine solche mobile Transaktion am Beispiel einer Verkaufsanwendung.

Vor der Ausführung mobiler Transaktionen können zusätzlich die im folgenden erläuterten Reservierungen angefordert werden, die gewisse Garantien bezüglich des Datenbankzustandes auf dem Server erlauben und zur Vermeidung konfligierender Änderungen dienen. Ähnlich dem Two-Tier-Modell [GHOS96] werden lokal zwei Versionen von Datenelementen gehalten, sogenannte vorläufige (*tentative*) und endgültige (*committed*) Versionen.

5.3.2 Reservierungen

In einigen mobilen Anwendungen kann es von großer Bedeutung sein, gewisse Änderungen an lokal replizierten Daten konfliktfrei reintegrieren zu können. In solchen Fällen bietet

```
------------------------- NEW ORDER -----------------------------
BEGIN
  SELECT price, stock INTO prd_price, prd_cnt
  FROM   products
  WHERE  name = 'BLUE THING';
  IF prd_price <= 10.00 AND prd_cnt >= 50 THEN
    -- update orders, current stock, ...
    NOTIFY('SMTP', 'sal-07@thingco.pt', order completed...');
    COMMIT;
  ENDIF;
  ROLLBACK;
  ON ROLLBACK NOTIFY('SMS', '351927435456', impossible order ...');
END;
-----------------------------------------------------------------
```

Abbildung 5.4: Beispiel für eine *mobile Transaktion*

ein rein optimistischer Ansatz keine adäquate Lösung. MobiSnap stellt dagegen zusätzliche Reservierungsgrade zur Verfügung, um gewisse Rechte und Garantien für replizierte Daten zu erhalten. Dieser hybride Ansatz zwischen rein optimistischer und rein pessimistischer Replikation wird in ähnlicher Weise von unserem im Kapitel 6 vorgestellten Modell verwendet, allerdings erfolgt die Anforderung solcher Rechte nicht auf dem in MobiSnap gewählten prozeduralen Weg.

- *Escrow*
  Hier findet eine Zerlegung von teilbaren Ressourcen, d.h. Mengen repräsentierenden Attributen, in für einzelne Clients exklusiv gesperrte Teilmengen statt. Beispielsweise können sich Verkäufer eine Gesamtmenge von Produkten eines Artikels teilen. Dieser Ansatz implementiert das bereits im Abschnitt 3.2.2 beschriebene `ESCROW`-Verfahren und ist damit ebenfalls nur auf wenige spezielle Attribute anwendbar. Zur Realisierung findet in der Datenbank eine Sperrung der reservierten Teilmengen statt.

- *Slot*
  Die Slot-Garantie reserviert das exklusive Einfügerecht über eine gewisse Menge von Tupeln, deren Attributwerte einer zu definierenden Bedingung genügen. Dabei spielt gerade im Umfeld vieler mobiler Nutzer die Granularität eine wichtige Rolle, um Änderungswünsche anderer Nutzer nicht zu versperren. Beispielsweise kann ein Nutzer das Recht zur Planung eines Treffens, d.h. Einfügung eines Tupel in die entsprechende Tabelle, für einen festgelegten Raum und eine bestimmte Zeitperiode anfordern.

- *Werteänderung*
  Diese Garantie reserviert das exklusive Änderungsrechts von Attributen bzw. Werten für einen Client, wenn das Attribut nicht von einer `ESCROW`-Reservierung abgedeckt werden kann. Beispielsweise kann dies die Änderung einer Produktbeschreibung sein, die je Produkt nur einmal auftritt. Zur Realisierung wird ein Trigger definiert, der jede konkurrierende Transaktion abbricht, welche die reservierten Daten ändern will.

Somit wird effektiv eine Sperrung der Daten durchgeführt. Auch hier ist das Granulat möglichst klein zu wählen.

- *Werteverwendung*
  Mit Hilfe dieser Garantie kann das Leserecht auf einem Attribut mit einem bestimmten Wert reserviert werden, d.h. es findet eine Zusicherung des Wertes statt, auch wenn das betreffende Attribut geändert wird. Beispielsweise kann ein mobiler Verkäufer ein Produkt zu einem gewissen Preis verkaufen, auch wenn der Preis geändert wird. Dafür wird jeweils der aktuelle Werte eines Datenelements durch den reservierten Wert innerhalb der Datenbank des Client ersetzt. Leider finden sich in [PMC02] keine genaueren Erläuterungen zu dieser Reservierung.

Solche Reservierungen werden nur für eine bestimmte Frist an mobile Clients vergeben, d.h. die besitzen eine sogenannte *Gültigkeitsdauer* wie bereits im Abschnitt 3.3.1 beschrieben. Die Ergebnisse einer mobilen Transaktion sind auf dem mobilen Client nur dann korrekt und endgültig, wenn diese nur auf reservierten Werten arbeitet und die Transaktion innerhalb der Gültigkeitsdauer der Reservierungen zum Server propagiert wird. Die Anforderungen solcher Garantien erfolgt anwendungsabhängig und skriptgesteuert beim Replizieren.

Zunächst findet eine Überprüfung der Gültigkeit einer Reservierungsanfrage eines Client statt. Dabei spielen erlaubte Reservierungen auf der Datenbank, aktuell zugesicherte Reservierungen, aktueller Datenbankzustand und die Client-Identität eine Rolle. Abhängig von der angeforderten Reservierungsart werden dann bei der Zusicherung, wie oben beschrieben, entsprechende „Sperren“ auf der Datenbank gesetzt. Schließlich müssen einige Informationen über die zugesicherten Reservierungen protokolliert werden, um vorgenommene Änderungen in der Datenbank nach Ablauf der Bereitstellungszeit rückgängig machen zu können.

5.3.3 Transaktionsausführung

Anwendungen auf dem Client führen mobile Transaktionen aus, die verschiedene Ergebnismeldungen haben können. Eine mit der Meldung *Reservation Commit* abgeschlossene Transaktion wurde erfolgreich auf reservierten Daten ausgeführt. Allerdings wird der endgültige Abschluss auf dem Server nur garantiert, falls die Reintegration vor Ablauf der Reservierung stattfindet. Wird im Gegensatz dazu eine Transaktion mit der Meldung *Tentative Commit* abgeschlossen, so ist diese zwar erfolgreich abgeschlossen, jedoch nur vorläufig und hat damit nur vorläufige Versionen erzeugt, da nicht ausreichend Reservierungen zur Unterstützung der Transaktion vorhanden waren. Die Reintegration derartiger Transaktionen kann zu Konflikten führen.

Wird eine Transaktion auf vorläufigen Daten ausgeführt und anschließend abgebrochen, so erhält sie den Status *Tentative Abort* und wird später nicht zum Server propagiert. Schließlich kann eine Transaktion auch mit dem Status *Unbekannt* abschließen, wenn die aktuell auf dem Client replizierten Daten nicht vollständig für die gewünschte Transaktionsausführung vorhanden sind, weil beispielsweise ein referenziertes Feld oder Datensatz nicht repliziert wurden.

Die Wiederausführung auf dem Server führt zum endgültigen Commit auf den Primärkopien der Daten oder zum Abbruch unter entsprechender Aktualisierung der Protokolldaten

des Servers und einer angemessenen Nutzerbenachrichtigung beispielsweise per E-Mail, da meist der Client zum Zeitpunkt des Auftretens von Konflikten bezüglich einzubringender Transaktionen nicht mehr erreichbar ist. Transaktionen, die aufgrund von gültigen Reservierungen anderer Transaktionen nicht ausgeführt werden können, verbleiben im System und werden nach Gültigkeitsablauf der betreffenden Reservierung oder Abbruch der entsprechenden Transaktion automatisch erneut ausgeführt.

5.3.4 Projektstatus

Nach [PMC02] ist ein Prototyp auf Java-Basis implementiert, der als Middleware zwischen Client und Server zur Bereitstellung oben beschriebener Funktionalität, wie Reservierungsmechanismus und Transaktionsausführung, dient. Die Auswertung von SQL-Statements übernimmt jeweils die zugrundeliegende Datenbank. Auf der Server-Seite wird hier derzeit Oracle 8 verwendet, es kann aber jede relationale Datenbank mit Unterstützung von Triggern zur Speicherung der Primärkopien eingesetzt werden. Auf dem Client wird die ebenfalls auf Java basierende Hypersonic-SQL Datenbank genutzt. Eine Benachrichtigung der Clients über den Erfolg ihrer eingebrachten Transaktionen kann wahlweise per Mail über einen SMTP-Server oder per SMS über ein HTTP-Gateway erfolgen. Für Leistungsbetrachtungen zum bereits implementierten Projekt wurde eine mobile Verkaufsanwendung zugrundegelegt.

Kapitel 6

Sprachentwurf

Im Gegensatz zu Produkten, in denen die Festlegung einer zu replizierenden Datenmenge für mobile Clients zentral und meist skriptgesteuert verläuft, überläßt die *nutzerdefinierte Replikation* [Gol03] die Auswahl der Daten der Anwendung auf dem mobilen Client. Dazu werden geeignete deskriptive Sprachmittel bereitgestellt. Dieses Kapitel beschäftigt sich mit der Integration der Konfliktvermeidung bzw. operationsorientierten Konfliktbehandlung in die Schnittstellen der nutzerdefinierten Replikation, abstrahiert dabei jedoch von Vorschlägen für eine konkrete Implementierung der Verfahren. Diese Schnittstellen werden von einem zwischen Client und Server geschalteten *RPS (Replication Proxy Server)* angeboten [Mül03]. Die verschiedenen Funktionalitäten an den Schnittstellen bzw. auf den Spezifikationsebenen sind als Übersicht in Tabelle 6.1 dargestellt.

6.1 Spezifikationsebenen der nutzerdefinierten Replikation

Als Erweiterung zum Client-Server-Modell aus Kapitel 2 ist im Modell der nutzerdefinierten Replikation, wie oben erwähnt und in Abbildung 6.1 dargestellt, ein RPS zwischen Client und Server geschaltet. Dadurch ergeben sich die im folgenden beschriebenen drei Spezifikationsebenen der Replikationsschemadefinitionsebene (Abschnitt 6.1.1), Replikatdefinitionsebene (Abschnitt 6.1.2) und Synchronisationsebene (Abschnitt 6.1.3).

6.1.1 Replikationsschemadefinitionsebene

Mit Sprachelementen auf dieser Ebene läßt sich die Struktur und Menge der zur Replikation freigegebenen Daten auf festlegen. Mit Hilfe des `CREATE CONSOLIDATED TABLE`-Statements (Abschnitt 6.2) werden Quelltabellen des Server mit weiteren Informationen und Festlegungen zur Konfliktvermeidung bzw. -auflösung angereichert und auf dem RPS als konsolidierte Tabellen zur Replikation bereitgestellt. Derartige Spezifikationen auf dem RPS sind Aufgaben eines Administrators, der entsprechende Rechte auf der Datenbank und semantisches Hintergrundwissen zu den Daten hat. Festlegungen für eine automatische Konfliktbehandlung sind nur auf Ebene der Datenbank-Operationen `INSERT`, `DELETE` und `UPDATE` sowie auf Ebene von Datentypen, beispielsweise durch Mittelwertbildung numerische Attributwerte, möglich. Diese durch den Administrator definierten Methoden zur

		Replikationsschemadefinitionsebene	Replikatdefinitionsebene	Synchronisationsebene	Bemerkungen
Konfliktvermeidung	INS	`KEY POOL`	`max-keys`, `REFILL`	automatisches Auffüllen	nur für künstliche Schlüssel
		`SLOT`	Slot-Tabelle angeben	–	nur für natürliche Schlüssel
	DEL	`RESTRICTION OF OPERATIONS`	–	–	–
		`RESTRICTION BY TIME`	`time-value`	–	
		`RESTRICTION BY REQUESTS`	–	–	
	UPD	`RESTRICTION OF OPERATIONS`	–	–	–
		`RESTRICTION BY TIME`	`time-value`	–	–
		`RESTRICTION BY REQUESTS`	–	–	–
		`ESCROW`	`column/quantity`, `REFILL`	automatisches Auffüllen	nur für `ESCROW`-Attribute
Konfliktauflösung	ALL	`DISCARD`	–	✓	–
		`OVERWRITE`	–	–	–
		`PRIORITY`	–	–	–
		`ERROR`	–	✓	–
	INS	`RENAME KEY`	–	–	nur für künstliche Schlüssel
		–	–	`ALTERNATIVES`	durch Anwendung erzeugt
	UPD	`NEW INSERT`	–	–	–
		`SUM, AVERAGE`	–	–	nur für zahlwertige Attribute
		`UDF`	Nutzerfunktion angeben	–	–

✓ : Verwendung möglich – : Verwendung nicht möglich

Tabelle 6.1: Konfliktvermeidung und Konfliktbehandlung auf den Spezifikationsebenen

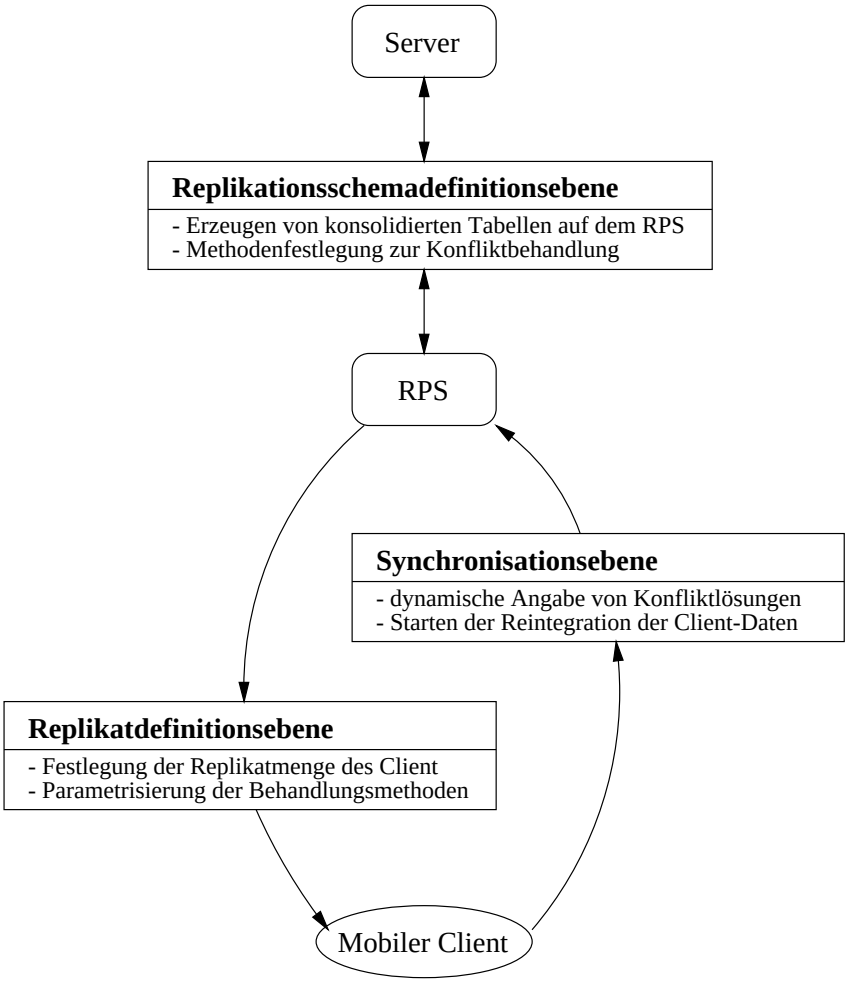


Abbildung 6.1: Spezifikationsebenen für den Sprachentwurf

Vermeidung und Auflösung von Konflikten sowie festgelegte `DEFAULT`-Werte für eine Tabelle werden automatisch bei einer entsprechenden Client-Anfrage für die Replikatdaten übernommen und können nutzerspezifisch parametrisiert werden.

6.1.2 Replikatdefinitionsebene

Wurden alle nötigen Tabellen auf der Replikationsschemadefinitionsebene mit entsprechenden Konfliktvermeidungs- und -auflösungsstrategien erweitert, muß nun als Voraussetzung für das mobile unverbundene Arbeiten der zu replizierende Ausschnitt festgelegt werden. Dies findet jeweils durch den Nutzer bzw. die Anwendung auf dem mobilen Client mit Hilfe des `CREATE REPLICATION VIEW`-Statements statt (Abschnitt 6.3). Neben der eigentlichen Auswahl der zu replizierenden Daten können die bereits durch den Administrator festgelegten Verfahren zur Konfliktvermeidung und Konfliktbehandlung anwendungsabhängig parametrisiert werden, beispielsweise mit Angaben über die Anzahl der zu reservierenden Primärschlüssel für Einfügungen mit dem `KEY-POOL`-Verfahren. Zusätzlich kann spezifiziert werden, ob benötigte Mengen für das `ESCROW`- bzw. `KEY-POOL`-Verfahren beim Synchro-

nisationszeitpunkt wieder bis zum definierten Maximum eines Client aufgefüllt werden sollen. Eine Veränderung der möglichen Parameter erfolgt über das in Abschnitt 6.4 beschriebene `ALTER REPLICATION VIEW`-Statement. Soll der zu replizierende Datenbankausschnitt geändert werden, kann dies nur durch eine Löschung mittels `DROP REPLICATION VIEW` [Mül03] und anschließender Neuerzeugung realisiert werden.

6.1.3 Synchronisationsebene

Wurden unverbundene Änderungen auf replizierten Daten durchgeführt, sind diese schließlich wieder mit dem RPS zu synchronisieren. Für diesen Vorgang sieht die Spracherweiterung das im Abschnitt 6.5 erläuterte `SYNCHRONIZE`-Statement vor, welches in Bezug auf die Konfliktbehandlung vorallem die Möglichkeit der Angabe alternativer Änderungen erlaubt.

6.1.4 Syntaxdiagramme

Die in den folgenden Abschnitten näher betrachtete Funktionalität auf den einzelnen Spezifikationsebenen wird zur besseren Lesbarkeit und Verständlichkeit in Syntaxdiagrammen dargestellt. Die Darstellung dieser Diagramme orientiert sich im wesentlichen an Elementen, wie sie beispielsweise auch in [IBM02] verwendet werden.

Eine mit einem Syntaxdiagramm dargestellte Anweisung beginnt mit `>>` und ist von links oben nach rechts unten zu lesen, wobei das Ende der Anweisung durch `><` gekennzeichnet ist. Sind Zeilenumbrüche notwendig, werden diese durch `->` verdeutlicht und mit `>-` auf der nächsten Zeile fortgesetzt. Schlüsselwörter, wie beispielsweise `FOR`, werden groß geschrieben, Variablen, wie beispielweise `table-name`, werden klein geschrieben. Tritt ein Minuszeichen (`-`) mehr als einmal hintereinander auf, so kennzeichnet es eine Trennung zwischen zwei Wörtern der Anweisung, d.h. es stellt ein Leerzeichen dar. Ansonsten gehört es zum Bezeichner einer Variablen. Wahlmöglichkeiten werden durch Verzweigungen mit `+` bzw. `'` dargestellt, Wiederholungen durch stilisierte Linien mit auf den Rücksprungpunkt zeigenden Pfeilen (`V`).

6.2 Die Anweisung `CREATE CONSOLIDATED TABLE`

Die in Abbildung 6.2 dargestellte Syntax dient zur Erstellung einer mit Informationen zu Konfliktbehandlungsmethoden angereicherten Tabelle, die auf dem RPS für die Replikation durch mobile Clients zur Verfügung steht. Dabei müssen die in [Mül03] beschriebenen Mechanismen für den Abgleich zur Quelltabelle auf dem Server sorgen. Einer Operation (`INSERT`, `UPDATE`, `DELETE`) bzw. einer Kombination aus Operation und Tabellenspalte kann maximal eine Methode zur Konfliktvermeidung bzw. Konfliktauflösung zugeordnet werden, die Auswahl des Verfahrens hängt dabei von der Operation ab und unterliegt gewissen Einschränkungen.

6.2.1 Syntax

Die Syntax für das `CREATE CONSOLIDATED TABLE`-Statement ist in Abbildung 6.2 dargestellt, die Bedeutung aller Variablen und Schlüsselwörter wird im folgenden erläutert.

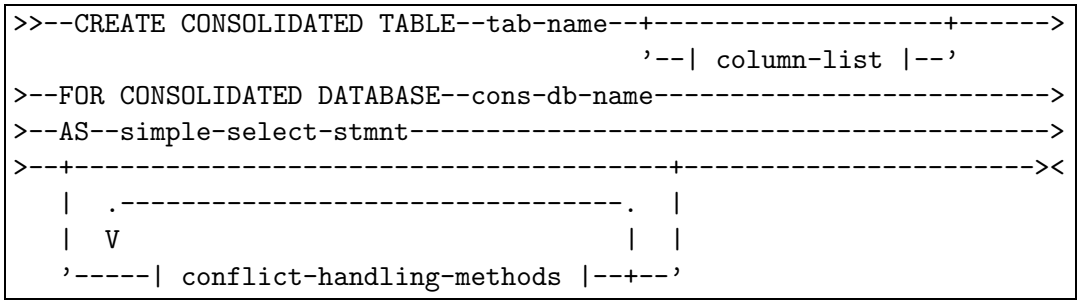


Abbildung 6.2: Syntaxdiagramm der Anweisung `CREATE CONSOLIDATED TABLE`

Dabei betreffen die in Tabelle 6.2 aufgeführten Teile der Syntax nicht unmittelbar die hier betrachteten Konzepte zur Konfliktbehandlung und werden nur kurz erläutert. Genauere Informationen diesbezüglich finden sich in der Arbeit von [Mül03].

Syntaxelement	Kurzbeschreibung
`tab-name`	Name der zu erzeugenden konsolidierten Tabelle
`column-list`	Auswahl von Tabellenspalten
`cons-db-name`	Name der konsolidierten Datenbank, in welcher die konsolidierte Tabelle erzeugt wird
`simple-select-stmnt`	Selektionsanweisung zur Auswahl der Daten

Tabelle 6.2: Weitere Elemente der `CREATE CONSOLIDATED TABLE`-Syntax

`| conflict-handling-methods |`
: Dieser in Abbildung 6.3 dargestellte Teil der Syntax beschreibt die Varianten zur Vermeidung (Abschnitt 6.2.1.1) und Auflösung (Abschnitt 6.2.1.2) von Konflikten für die unverbunden ausgeführten Änderungsoperationen `INSERT`, `DELETE` und `UPDATE`.

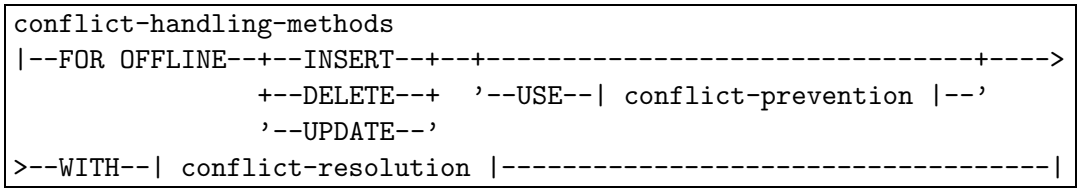


Abbildung 6.3: Syntaxdiagramm für `conflict-handling-methods`

6.2.1.1 Konfliktvermeidung

Über die in Abbildung 6.4 dargestellte Anweisung `| conflict-prevention |` lassen sich Verfahren zur Konfliktvermeidung wählen. Die ausführliche Beschreibung der einzelnen

Methoden wurde bereits im Kapitel 3 vorgenommen, die Auswahl und eventuelle Belegung mit DEFAULT-Werten ist, wie einleitend erläutert, Aufgabe des Administrators. Werden keine Vorkehrungen zur Konfliktvermeidung getroffen werden, kann nur durch eine exklusive Anforderung der Daten während der CREATE REPLICATION VIEW-Anweisung in Abschnitt 6.3 eine konfliktfreie Änderung garantiert werden, ansonsten unterliegen die Daten dem optimistischen Zugriff. Dieser ist sinnvoll, wenn es aufgrund der Anwendungslogik zu keinerlei Konflikten kommen kann oder Daten ausschließlich zum Lesen bereitgestellt werden.

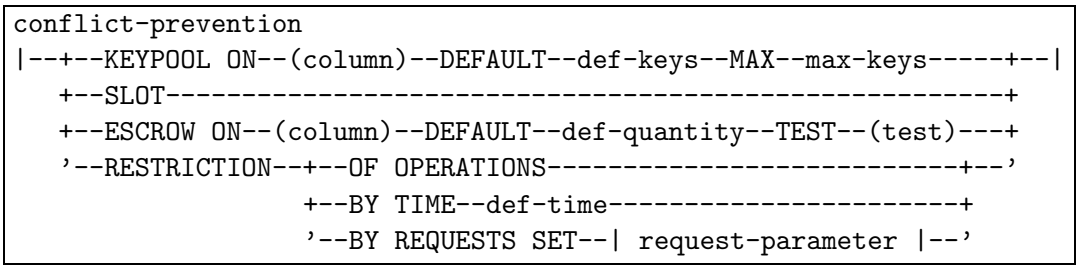


Abbildung 6.4: Syntaxdiagramm für conflict-prevention

KEYPOOL ON (column) DEFAULT def-keys MAX max-keys
: Besitzt eine zu replizierende Tabelle mit dem Attribut column einen künstlichen Primärschlüssel bzw. eine mit UNIQUE definierte zahlwertige Spalte, so kann zur Vermeidung von Einfügekonflikten das im Abschnitt 3.2.1 vorgestellte KEY-POOL-Verfahren verwendet werden, welches mit dem Schlüsselwort KEYPOOL eingeleitet wird. Der Parameter def-keys bezeichnet hierbei die Menge der standardmäßig vergebenen Anzahl von reservierten Primärschlüsseln je replizierenden Client, max-keys legt die maximal anforderbare Menge von Schlüsseln fest. Der Nutzer bzw. die Anwendung kann dann die tatsächlich gewünschte Menge zum Zeitpunkt der eigentlichen Replikatdefinition im CREATE REPLICATION VIEW-Statement spezifizieren.

SLOT
: Diese Klausel deklariert die Möglichkeit, INSERT-Konflikte in der entsprechenden Tabelle nach dem im Abschnitt 3.2.3 beschriebenen SLOT-Verfahren zu vermeiden. Die Belegung notwendiger Attribute einzufügender Tupel durch ein PRE-INSERT findet allerdings erst zum Zeitpunkt der Replikation durch den Nutzer bzw. die Anwendung mittels einer anzugebenden Tabelle statt, da ein Automatismus wie beim KEY-POOL-Verfahren nicht möglich ist.

  Beispielsweise könnte ein Nutzer im Reiseinformationssystem HERMES [Bau03] beabsichtigen während einer Stadtbesichtigung vorhandene Sehenswürdigkeiten näher zu beschreiben bzw. zu erfassen. Dazu reserviert er sich die entsprechenden Einfügerechte über bekannte Schlüsselwerte (Name der Stadt, Name der Sehenswürdigkeit). Weitere Attribute, wie beispielsweise Baujahr, Stil oder Zustand einer Sehenswürdigkeit, können dann unverbunden eingetragen werden, während sie auf dem Server mit NULL-Werten vorbelegt werden.

ESCROW ON (column) DEFAULT def-quantity TEST (test)
: Diese Konfliktvermeidung ist nur für die Operation UPDATE auf einzelnen Attri-

buten anwendbar. Dazu muß das durch `column` referenzierte Attribut die in Abschnitt 3.2.2 beschriebenen Eigenschaften der Zahl- und Mengenwertigkeit erfüllen. Der mit `def-quantity` bezeichnete Parameter ist die per Default vergebene maximale „Teilmenge“ des Attributs `column`, diese läßt sich in der `CREATE REPLICATION VIEW`-Anweisung durch den Nutzer anwendungsbedingt verringern. Dagegen kann der Parameter `test`, der für die Integritätssicherung im Sinne der Anwendung zuständig ist, nur vom Administrator spezifiziert werden. Die Syntax zur Formulierung einer solchen Testbedingung orientiert sich dabei an den Restriktionen der `CHECK`-Option innerhalb einer Tabellendefinition und ist beispielhaft in Abbildung 6.7 dargestellt.

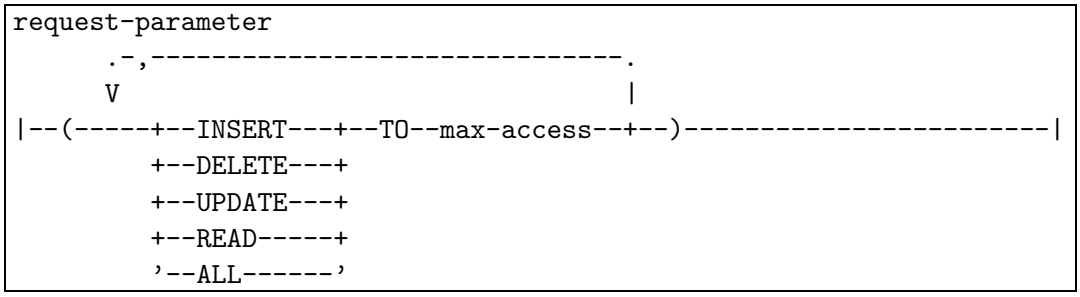


Abbildung 6.5: Syntaxdiagramm für `request-parameter`

`RESTRICTION`

Mit Hilfe dieses Schlüsselwortes werden Verfahren ausgewählt, die ein konfliktvermeidendes Löschen bzw. Ändern von Tupeln erlauben. Dazu lassen sich folgende Beschränkungen für die möglichen Operationen `UPDATE` und `DELETE` spezifizieren:

- `OF OPERATIONS`
  Bei Auswahl dieser Option wird ein konfliktfreies Löschen bzw. Ändern durch die Einschränkung möglicher Operationen für andere Clients erreicht, die eine gleiche oder überlappende Anforderung von Replikatdaten haben. Grundlage ist hierfür die im Abschnitt 3.3.2 bereits erläuterte Verträglichkeitsmatrix in Tabelle 3.1.
- `BY TIME def-time`
  Durch diese Option wird die Zusicherung der konfliktfreien Löschung bzw. Änderung in ihrer Gültigkeit zeitlich befristet, d.h. eine Synchronisation von Änderungen basierend auf solchen Reservierungen nach Ablauf der Gültigkeit kann zu Konflikten führen wie im Abschnitt 3.3.1 beschrieben. Die per Default vorgegebene Dauer der Reservierungen wird durch den Parameter `def-time` festgelegt und kann vom Nutzer im `CREATE REPLICATION VIEW`-Statement verändert werden. Ist dabei die angeforderte Zeitspanne größer als `def-time`, wird die Anforderung abgewiesen.
- `BY REQUESTS SET | request-parameters |`
  Die Zusicherung für eine konfliktfreie Löschung bzw. Änderung wird hier ebenfalls beschränkt, allerdings durch einen vom Administrator festzulegenden Wert `max-access` von Zugriffsanforderungen je Operation anderer Clients auf die reservierten Daten, wie in Abbildung 6.5 dargestellt. Eine Veränderung dieser

Werte durch den Nutzer ist nicht möglich, da deren Bestimmung nur durch die recht aufwendige Auswertung von Zugriffsstatistiken erfolgen kann. Wird das Schlüsselwort `ALL` verwendet, gilt für alle Operationen der gleiche Schwellwert. Einsatz findet diese Möglichkeit beispielsweise bei der Aufrechterhaltung der Verfügbarkeit eines Systems, wenn die Anzahl der Zugriffe auf bestimmte Daten zeitabhängig und nicht konstant ist.

6.2.1.2 Konfliktauflösung

Der in Abbildung 6.6 dargestellte Syntaxteil `| conflict-resolution |` erlaubt die Definition von Methoden zur Konfliktauflösung. Wie bei den Verfahren zur Konfliktvermeidung sind auch hier nicht alle Operationen mit allen Methoden verträglich, was durch entsprechende Hinweise in der folgenden Beschreibung verdeutlicht wird.

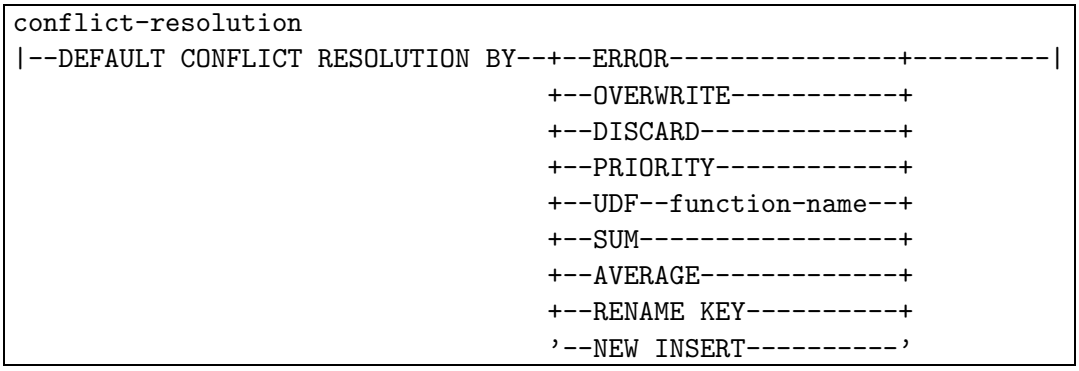


Abbildung 6.6: Syntaxdiagramm für `conflict-resolution`

`ERROR`
: Erkannte Synchronisationskonflikte bezüglich Einfügen, Ändern oder Löschen werden nicht automatisch gelöst, sondern nur für eine manuelle Bearbeitung protokolliert. Dies ist vorallem dann notwendig, wenn sich eine automatische Konfliktlösung im Sinne der Anwendungssemantik nicht erreichen läßt.

`OVERWRITE`
: Synchronisationskonflikte bezüglich Einfügen oder Ändern werden durch Überschreiben des aktuell auf dem RPS vorliegenden Tupels mit dem einzubringenden Tupel des mobilen Client gelöst. Durch Anwendung dieser `client-wins`-Regel bleiben jeweils die zuletzt eingebrachten, aber nicht notwendigerweise aktuellsten, Tupel erhalten.

`DISCARD`
: Synchronisationskonflikte bezüglich Einfügen, Ändern oder Löschen werden durch das Abweisen des einzubringenden Tupels gelöst. Sollen Einfügekonflikte bezüglich gleichen Inhalts aber mit unterschiedlichem Primärschlüssel ebenfalls erkannt werden, ist hierzu eine `UNIQUE`-Definition über alle nicht zum Schlüssel gehörende Attribute notwendig, wie im Abschnitt 4.3.3.2 beschrieben. Effektiv wird also die Regel `server-wins` realisiert.

`PRIORITY`
: Synchronisationskonflikte bezüglich Einfügen oder Ändern werden durch Vergleich von Prioritäten verschiedener Nutzer bzw. Geräte gelöst, indem die Daten mit höherer Priorität im System integriert bzw. beibehalten werden. Effektiv entscheidet also ein Nutzerkonzept über die bereits erläuterten Lösungsmethoden `OVERWRITE` und `DISCARD`. Eine Möglichkeit zur Realisierung wäre die Erweiterung jedes Tabellenschemas um ein zahlwertiges Attribut, welches den Nutzer identifiziert, der beispielsweise ein Tupel neu angelegt oder geändert hat. Diese könnte dann für Vergleichszwecke herangezogen werden.

`UDF function-name`
: Synchronisationskonflikte bezüglich Einfügen oder Ändern können durch Anwendung einer vom Administrator definierten Funktion `function-name` gelöst werden. Eine solche Funktion kann sowohl auf vollständigen Tupeln als auch für einzelne Attribute definiert sein, entsprechend muß mit dem Namen der Funktion auch deren Signatur in Form von Ein- und Ausgabeparametern angegeben werden.

`SUM/AVERAGE`
: Änderungskonflikte durch verschiedene Werte für ein zahlwertiges Attribut können durch Anwendung numerischer Funktionen gelöst werden, beispielsweise indem für das entsprechende Attribut die Summe bzw. der Durchschnitt aus beiden im Konflikt stehenden Attributen übernommen wird.

`RENAME KEY`
: Einfügekonflikte durch Tupel mit gleichem Primärschlüssel werden hier durch Umbenennung des Primärschlüssels des einzufügenden Tupels gelöst, wobei Folgeabhängigkeiten, beispielsweise in Fremdschlüsselbeziehungen, ebenfalls beachtet und entsprechende aktualisiert werden müssen. Diese Auflösungsvariante steht nur für künstliche Schlüssel zur Verfügung, deren Umbenennung durch Konzepte zur Verwaltung freier Primarschlüssel realisiert werden kann. Ähnliche Verfahren finden sich beispielsweise auch in der Freiplatzverwaltung eines Speichersystems.

`NEW INSERT`
: Änderungskonflikte durch Änderungen auf bereits gelöschten Tupeln werden durch ein Wiedereinfügen des einzubringenden Tupels gelöst. Diese Variante bedarf einer genauen Analyse der Anwendungssemantik vom Administrator.

6.2.2 Beispiel

Als Beispiel sei das in Abschnitt 2.3.1 beschriebene Verkaufsszenario betrachtet. Auf dem Server des Herstellers existiere eine Tabelle `lagerbestand`, die zu jedem über die Produktnummer (`PNr`) eindeutig identifizierten Produkt einige Eigenschaften wie die Beschreibung und die im Lager vorhandene Menge enthält. Ausserdem sind die Mitarbeiter explizit dazu angehalten, nach neuen innovativen Produkten zu suchen und diese in Eigenverantwortung einzukaufen. Für die Erstellung einer entsprechenden konsolidierten Tabelle `rp-lagerbestand` auf dem RPS wird das in Abbildung 6.7 dargestellte Statement verwendet.

Findet ein Mitarbeiter neue Produkte, kann er diese nach dem für den Primärschlüssel `PNr` definierten `KEY-POOL`-Prinzip einfügen. Über die maximal reservierbare Menge von

```
CREATE CONSOLIDATED TABLE rp-lagerbestand
FOR CONSOLIDATED DATABASE my-rps-db
AS SELECT * FROM lagerbestand
FOR OFFLINE INSERT
         USE KEYPOOL ON (PNr) DEFAULT 10 MAX 20
FOR OFFLINE DELETE
         WITH DEFAULT CONFLICT RESOLUTION BY DISCARD
FOR OFFLINE UPDATE
         USE ESCROW ON (menge) DEFAULT 5 TEST (menge > 10);
```

Abbildung 6.7: Beispiel für `CREATE CONSOLIDATED TABLE`

20 Primärschlüsseln hinaus sind innerhalb einer unverbundenen Arbeitsphase keine weiteren Einfügungen durchführbar, auch nicht optimistisch. Dadurch können keine `INSERT`-Konflikte während der Reintegration entstehen. Für Löschungen gibt es keine Konfliktvermeidung, doppelte Löschungen desselben Produktes werden ignoriert. Für Änderungen des Attributes `menge` wird das `ESCROW`-Verfahren eingesetzt, wobei die vergebene Default-Menge 5 ist und Anforderungen solcher Teilmengen die Bedingung `(menge > 10)` einhalten müssen, d.h. der Bestand des Servers bezüglich dieses Attributes muß nach einer Reservierung größer als 10 sein.

6.3 Die Anweisung `CREATE REPLICATION VIEW`

```
CREATE TABLE replication-view-parameter
    (rv-name              VARCHAR  NOT NULL,
     operation            CHAR(6)  NOT NULL CHECK (operation IN
                                     ('INSERT', 'UPDATE', 'DELETE')),
     attribut-name        VARCHAR  NOT NULL,
     key-pool-value       INTEGER,
     restrict-time        TIME,
     max-ins-access       INTEGER,
     max-upd-access       INTEGER,
     max-del-access       INTEGER,
     max-read-access      INTEGER,
     PRIMARY KEY (rv-name, operation, attribut-name)
    )
```

Abbildung 6.8: SQL-Statement zur Erzeugung der RV-Parameter-Tabelle

Über die Anweisung `CREATE REPLICATION VIEW` wird auf dem mobilen Client durch den Nutzer bzw. die Anwendung eine Replikationssicht mit der angeforderten Datenmenge und den entsprechenden Zusicherungen bzw. exklusiven Reservierungen angelegt. Ist diese so spezifizierte Definition einer Replikationssicht ungültig, beispielsweise aufgrund von Rechtekonflikten oder nichterfüllbaren Datenanforderungen, wird sie mit einer Fehlermeldung zurückgewiesen. Ist dagegen die Definition der Replikationssicht erfolgreich, so wird diese

als Grundlage der unverbundenen Arbeit innerhalb der spezifizierten Replikatdatenbank auf dem Client angelegt. Desweiteren werden auf dem Server alle nötigen Informationen zu den verwendeten Parametern in der sogenannten *RV-Parameter-Tabelle (Replication View Parameter Table)* festgehalten, wie sie in Abbildung 6.8 dargestellt ist. Diese enthält für jede Replikationssicht die zu einer Operation und Attribut festgelegten bzw. per `DEFAULT` übernommenen Parameter, wie beispielsweise die Gültigkeitsdauer `restrict-time`.

6.3.1 Syntax

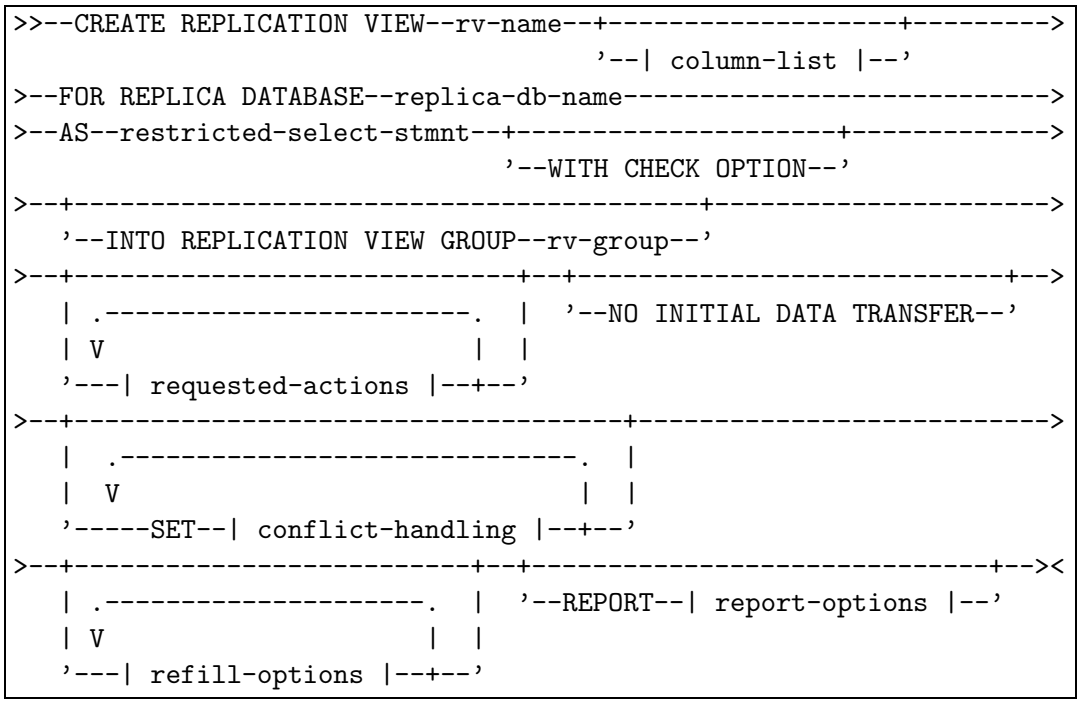


Abbildung 6.9: Syntaxdiagramm der Anweisung `CREATE REPLICATION VIEW`

Im folgenden wird die Semantik der in Abbildung 6.9 dargestellten Syntax des `CREATE REPLICATION VIEW`-Statements erläutert, wobei wir uns jedoch nur auf die zur Konfliktbehandlung relevanten Teile beschränken. Alle anderen in Tabelle 6.3 zusammengefaßten Elemente dieser Syntaxdefinition basieren auf der Arbeit von [Mül03] und werden dort ausführlich erläutert.

6.3.1.1 Rechteanforderung

Durch das in Abbildung 6.10 dargestellte `| requested-actions |`-Konstrukt können optional sowohl optimistische als auch exklusive Änderungsanforderungen an die selektierten Daten gestellt werden, wie im folgenden beschrieben.

`FOR OFFLINE MODIFICATION`
: Wird diese Option verwendet, so sind die replizierten Daten für optimistische Änderungen freigegeben. Es können, wie im Kapitel 4 beschrieben, beim Synchronisieren der so geänderten Daten Konflikte entstehen.

Syntaxelement	Kurzbeschreibung
`rv-name`	Name der zu erzeugenden Replikationssicht
`column-list`	Auswahl von Tabellenspalten
`replica-db-name`	Name der Replikatdatenbank des Client, in welcher die Replikationssicht erzeugt wird
`restricted-select-stmnt`	Selektionsanweisung zur Auswahl der Daten
`WITH CHECK OPTION`	einzufügende bzw. zu ändernde Daten müssen der Selektionsbedingung genügen
`rv-group`	Zuordnung der Replikationssicht zu einer Replikationssichtengruppe
`NO INITIAL DATA TRANSFER`	Replikationssicht wird nur zur Einfügung verwendet, keine Datenübertragung notwendig

Tabelle 6.3: Weitere Elemente der `CREATE REPLICATION VIEW`-Syntax

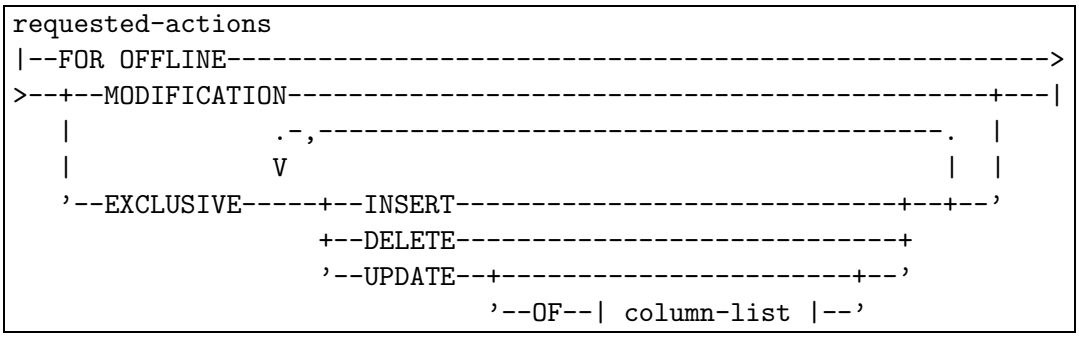


Abbildung 6.10: Syntaxdiagramm für `requested-actions`

`FOR OFFLINE EXCLUSIVE`
: Ist eine Konfliktvermeidung basierend auf den Eigenschaften eines Datentyps oder einer Operation, wie beispielsweise beim `KEY-POOL`-Verfahren, nicht möglich, können konfliktfreie Änderungen auch durch Anforderung exklusiver Operationsrechte auf den Daten garantiert werden. Dabei werden die betroffenen Daten in der konsolidierten Tabelle des RPS für andere Nutzer mit konfligierenden Zugriffen gesperrt. Das Recht zur exklusiven Anforderung besitzen allerdings nur ausgewählte Nutzer, um die Verfügbarkeit nicht zu stark einzuschränken. Durch eine `FOR OFFLINE EXCLUSIVE`-Anweisung wird nicht auch automatisch das optimistische Änderungsrecht vergeben, sondern muß explizit angefordert werden.

6.3.1.2 Konfliktbehandlung

Mit der in Abbildung 6.11 dargestellten `| conflict-handling |`-Option lassen sich Methoden zur Konfliktvermeidung und -behandlung, die für eine konsolidierte Tabelle vom Administrator im Abschnitt 6.2 festgelegt wurden, für ein konkretes Anwendungsszenario parametrisieren. Dabei werden vorgegebene Werte überschrieben, die jedoch durch Aus-

wahl der DEFAULT-Option während der Änderung einer Replikationssicht im Abschnitt 6.4 wieder hergestellt werden können.

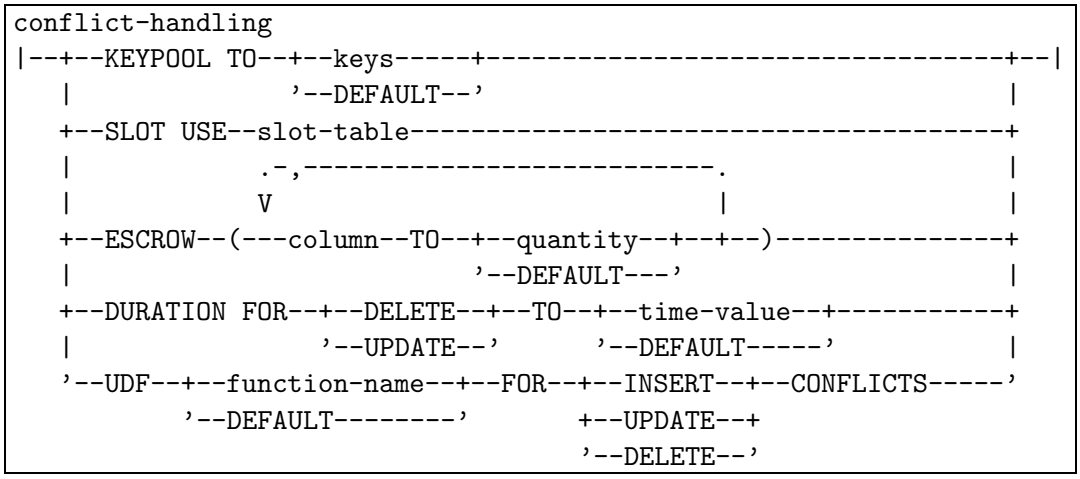


Abbildung 6.11: Syntaxdiagramm für conflict-handling

KEYPOOL TO keys
: Die über den Parameter keys anzugebende Menge an Primärschlüsseln darf das Maximum max-keys der KEY-POOL-Anweisung im CREATE CONSOLIDATED TABLE-Statement nicht überschreiten, d.h. es wird effektiv das Maximum aus beiden Größen für die Erzeugung der Replikationssicht verwendet.

SLOT USE slot-table
: Wurde die SLOT-Option zur Vermeidung von Einfügekonflikten festgelegt, muß hier im Zuge der Definition einer Replikationssicht der Nutzer bzw. die Anwendung eine Tabelle über den Parameter slot-table angeben. Diese Tabelle enthält alle bereits mit notwendigen Werten belegten Tupel, für die das Einfügerecht reserviert werden soll, wie das in Tabelle 6.4 beispielhaft für die im Abschnitt 3.2.3 beschriebene Terminplanungsanwendung dargestellt ist.

Raum	Datum	Zeit	Leiter	Zweckbeschreibung
SR123	2003-02-13	15:00	<NULL>	<NULL>
SR123	2003-02-20	15:00	<NULL>	<NULL>
⋮	⋮	⋮		

Tabelle 6.4: Beispiel einer SLOT-Tabelle

ESCROW (column TO quantity)
: Für die mit column bezeichneten Spalten können nutzerdefinierte ESCROW-Mengen über den Parameter quantity angegeben werden, die allerdings nicht größer als die durch den Administrator festgelegte DEFAULT-Menge def-quantity sein darf.

`DURATION FOR DELETE/UPDATE TO time-value`
: Soll die vom Administrator definierte Gültigkeitsdauer einer, wie im Abschnitt 3.3.1 beschriebenen, zeitlich beschränkten Änderungs- bzw. Löschreservierung geändert werden, kann dazu ein nutzerspezifizierter Wert für `time-value` angegeben werden. Ist dieser größer als der `DEFAULT`-Wert, kann es zur Abweisung einer so angeforderten Replikationssicht kommen.

`UDF function-name`
: Die einzige Möglichkeit des Nutzers, in die vom Administrator festgelegte Konfliktauflösung während der Erzeugung einer Replikationssicht einzugreifen, besteht in der Angabe einer eigenen definierten Funktion inklusive Signatur über den Parameter `function-name` zur Konfliktauflösung per `UDF`.

6.3.1.3 Wiederauffüllung und Benachrichtigung

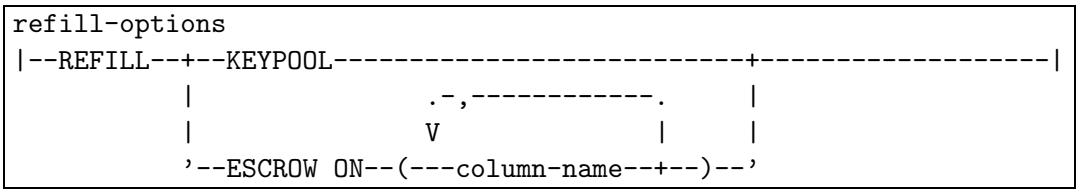


Abbildung 6.12: Syntaxdiagramm für `refill-options`

`| refill-options |`
: Mit dieser in Abbildung 6.12 dargestellten Option können die zu einer Replikationssicht möglicherweise gehörenden `KEY-POOL`- bzw. `ESCROW`-Mengen beim Reintegrieren auf die angeforderte Menge aufgefüllt werden. Wird diese Option nicht verwendet, werden standardmäßig die noch auf dem Client vorhandenen und nicht verwendeten Primärschlüssel bzw. Teilmengen von `ESCROW`-Attributen an den RPS rücküberführt.

`| report-options |`
: Über den in Abbildung 6.13 dargestellten `REPORT`-Syntaxteil kann der Nutzer spezifizieren, in welchem Umfang er über seine eingebrachten Änderungen Informationen erhalten will. Dabei steht sowohl die Art von Konflikten (`ALL, SOLVED, UNSOLVED`) als auch das Informationsmedium (`MAIL, SMS, LOG`) zur freien Kombinationsauswahl. Dabei müssen die Daten zu Mailadresse oder Handy-Nummer in vorher eingerichteten Nutzerprofilen verwaltet sein. Die Implementierung setzt dann ein entsprechendes Mail- und SMS-Gate voraus (siehe Ausblick im Kapitel 8).

6.3.2 Beispiel

Legt man wieder das Szenario einer Lagerverwaltung aus Abschnitt 2.3.1 zugrunde, so könnte sich ein Verkäufer mit der in Abbildung 6.14 dargestellten Anweisung eine Replikationssicht `hoover-sales-view` innerhalb seiner lokal vorhandenen Replikatdatenbank `my-client-db` erzeugen, welche die Daten aller Produkte repliziert, die als 'Staubsauger' beschrieben sind. Damit er eine Menge von Staubsaugern neben anderen unabhängig arbeitenden Verkäufern auch konfliktfrei verkaufen kann, ohne diese physisch mit sich zu

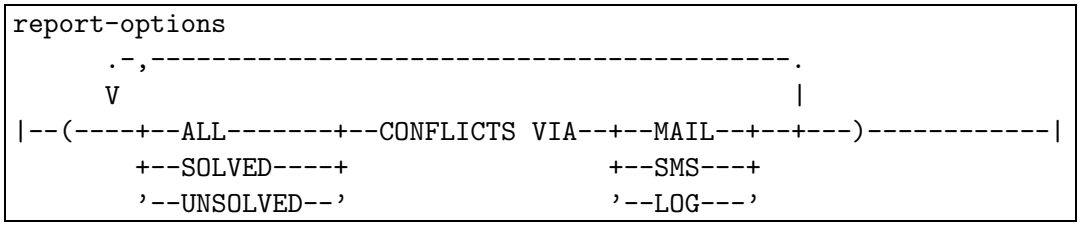


Abbildung 6.13: Syntaxdiagramm für `report-options`

transportieren, reserviert er sich 7 Einheiten vom Attribut `menge`. Außerdem wird festgelegt, daß beim Reintegrieren sowohl verwendete `ESCROW`- als auch `KEY-POOL`-Mengen wieder aufgefüllt werden.

```
CREATE REPLICATION VIEW hoover-sales-view
FOR REPLICA DATABASE my-client-db
AS SELECT *
   FROM   rp-lagerbestand lager
   WHERE  lager.beschreibung = 'Staubsauger'
FOR OFFLINE MODIFICATION
SET ESCROW (lager.menge TO 7)
REFILL ESCROW ON (lager.menge)
REFILL KEYPOOL;
```

Abbildung 6.14: Beispiel für `CREATE REPLICATION VIEW`

6.4 Die Anweisung ALTER REPLICATION VIEW

Die Anweisung `ALTER REPLICATION VIEW` erlaubt im verbundenen Zustand des Client die Änderung bereits festgelegter Parameter einer Replikationssicht `rv-name` innerhalb der Replikatdatenbank `replica-db-name`. Eine derart geänderte Replikationssicht kann dann wiederum, beispielsweise aufgrund zusätzlicher konfligierender Rechteanforderungen, zur Abweisung führen. Ist dagegen die Anforderung erfolgreich, müssen entsprechend die Replikationssicht-Parameter der in Abbildung 6.8 dargestellten RV-Parameter-Tabelle auf dem RPS aktualisiert werden. Weitere Auswirkungen einer solchen Replikationssichtänderung sind in [Mül03] beschrieben.

6.4.1 Syntax

In diesem Abschnitt wird die zur Änderung einer Replikationssicht in Abbildung 6.15 abgebildete Syntax der `ALTER REPLICATION VIEW`-Anweisung erläutert. Die vorrangig in [Mül03] beschriebenen Elemente sind dabei wieder in der Tabelle 6.5 zusammengefaßt.

`ADD/REVOKE | requested-actions |`

Diese Klausel erlaubt das Hinzufügen (`ADD`) von weiteren bzw. Entfernen (`REVOKE`)

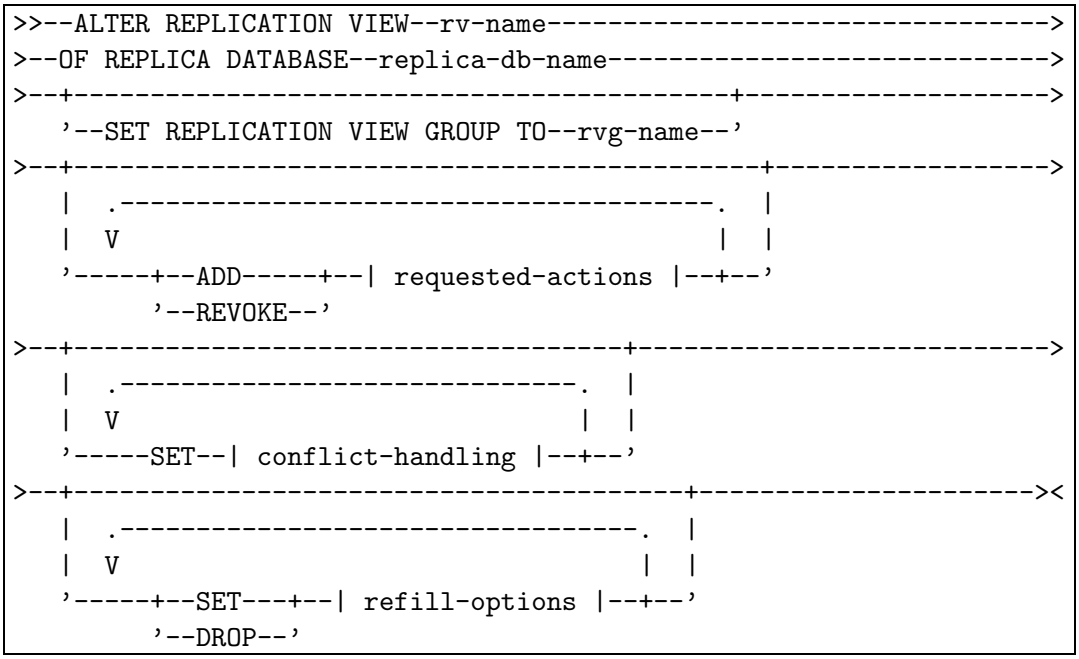


Abbildung 6.15: Syntaxdiagramm der Anweisung `ALTER REPLICATION VIEW`

Syntaxelement	Kurzbeschreibung
`rv-name`	Name der zu ändernden Replikationssicht
`replica-db-name`	Name der Replikatdatenbank des Client, in welcher die zu ändernde Replikationssicht existiert
`rv-group`	Name der neuen Replikationssichtengruppe

Tabelle 6.5: Weitere Elemente der `ALTER REPLICATION VIEW`-Syntax

von bestehenden Zusicherungen auf den angeforderten Daten. Dabei können, wie bereits im Abschnitt 6.3 erläutert, über den in Abbildung 6.10 dargestellten Anweisungsteil `| requested-actions |` sowohl optimistische als auch exklusive Änderungsrechte voneinander getrennt festgelegt werden.

`SET | conflict-handling |`
: Sollen für Verfahren zur Konfliktvermeidung bzw. -auflösung durch den Nutzer festgelegte Parameter wieder auf `DEFAULT`-Werte zurückgesetzt bzw. bestehende Parameter verändert werden, so kann dies durch Modifikation der in Abbildung 6.11 dargestellten und im Abschnitt 6.3 beschriebenen `| conflict-handling |`-Anweisung erreicht werden. Dabei ist zu beachten, daß nur die Möglichkeit zur Veränderung von Parametern besteht, jedoch nicht die vom Administrator im `CREATE CONSOLIDATED TABLE`-Statement festgelegten Methoden zu Konfliktvermeidung bzw. -auflösung verändert werden können.

`SET/DROP | refill-options |`
: Durch diese Anweisung kann über die in Abbildung 6.12 dargestellte Klausel die ent-

sprechende Verwaltung der ESCROW- und KEY-POOL-Mengen einer Replikationssicht zur Synchronisation verändert werden.

6.4.2 Beispiel

Betrachtet man das Beispiel in Abbildung 6.14 als Grundlage, so könnte der mobile Verkäufer entscheiden, sein optimistisches Recht für jede Art der unverbundenen Änderung auf der bereits definierten hoover-sales-view abzugeben und statt dessen das exklusive Einfügerecht anzufordern. Unabhängig davon sieht er aktuell seine Chance auf einen vermehrten Staubsauger-Absatz und reserviert sich eine Verkaufsmenge von 15 Einheiten. Diese Änderungen realisiert die in Abbildung 6.16 dargestellte Anweisung.

```
ALTER REPLICATION VIEW hoover-sales-view
FOR REPLICA DATABASE my-client-db
REVOKE FOR OFFLINE MODIFICATION
ADD FOR OFFLINE EXCLUSIVE INSERT
SET ESCROW (hoover-sales-view.menge TO 15);
```

Abbildung 6.16: Beispiel für ALTER REPLICATION VIEW

6.5 Die Anweisung SYNCHRONIZE

Die SYNCHRONIZE-Anweisung ermöglicht auf deklarativem Weg das Starten der Synchronisation von unverbunden durchgeführten Änderungen innerhalb einer Replikationssichtengruppe mit der Möglichkeit einer dynamischen Konfliktauflösung durch Alternativen. Dafür wird, wie bei allen anderen in diesem Kapitel besprochenen Anweisungen, eine Verbindung zwischen Server und RPS vorausgesetzt [Mül03].

6.5.1 Syntax

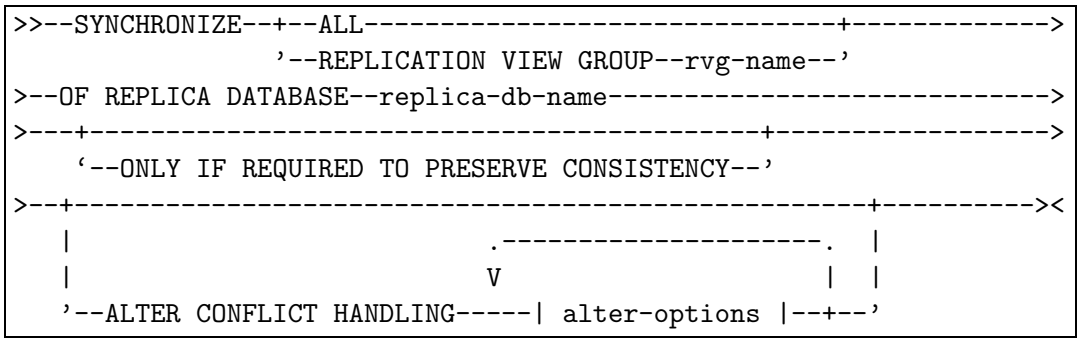


Abbildung 6.17: Syntaxdiagramm der Anweisung SYNCHRONIZE

Dieser Abschnitt beschreibt die in Abbildung 6.17 abgebildete Syntax der SYNCHRONIZE-Anweisung. Wie in den Beschreibungen zuvor wurden auch hier die in [Mül03] ausführlich erläuterten Elemente in Tabelle 6.6 zusammengefaßt.

Syntaxelement	Kurzbeschreibung
`ALL`	Synchronisation aller Replikationssichtengruppen der betreffenden Replikatdatenbank eines Client
`rvg-name`	Synchronisation der Replikationssichten, die in der Replikationssichtengruppe `rvg-name` existieren
`replica-db-name`	Name der zu synchronisierenden Replikatdatenbank des Client
`ONLY IF REQUIRED`	Synchronisation wird nur durchgeführt, wenn aus Konsistenzgründen notwendig

Tabelle 6.6: Weitere Elemente der `SYNCHRONIZE`-Syntax

`| alter-options |`

Für die Veränderung von Konfliktauflösungsmethoden zum Zeitpunkt der Synchronisation sind nur die in Abbildung 6.18 dargestellten Möglichkeiten realisierbar:

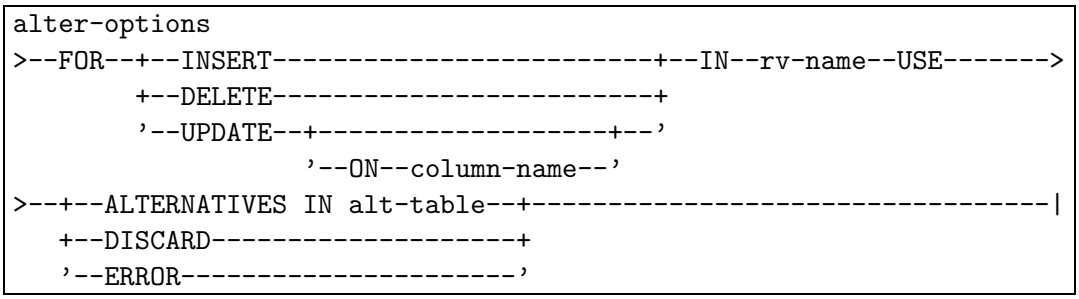


Abbildung 6.18: Syntaxdiagramm für `alter-options`

- Abschwächung der Konfliktauflösung:
  Durch Auswahl zwischen `DISCARD` oder `ERROR` für einen Konflikt bezüglich einer der Operationen `INSERT`, `DELETE` oder `UPDATE` kann die vom Administrator während des `CREATE CONSOLIDATED TABLE`-Statments in Abschnitt 6.2 festgelegte Auflösungsvariante überschrieben werden. Dies ist auch ohne Absprache mit dem Administrator möglich, da durch die beiden Varianten die Konsistenz der Serverdatenbank nicht gefährdet ist.

- Festlegung von Alternativen:
  Über den Parameter `alt-table` kann eine Tabelle angegeben werden, in der alternative Tupel zur Auflösung von Konfliktfällen zur Verfügung stehen, die zuvor durch die Anwendung erzeugt werden müssen. Dabei beschränken wir uns hier auf Einfügekonflikte. Eine Verwendung von Alternativen für Änderungen ist ebenfalls möglich, allerdings im Zusammenhang mit der Konflikterkennung weitaus komplizierter. Die `ALTERNATIVES`-Methode ersetzt nun die ursprünglich definierte Konfliktauflösung, testet die existierenden Alternativen für ein Tupel der Reihe nach und wird mit `ERROR` beendet, falls die letztmögliche Alternative ebenfalls einen Konflikt verursacht.

```
INSERT INTO table-name
       VALUES (...)
ALTERNATIVES INTO alt-table
       VALUES (...)
```

Abbildung 6.19: Integration einer ALTERNATIVES-Klausel

Eine Variante, neben der Verwendung zusätzlicher INSERT-Anweisungen, für die Erzeugung von Alternativen wäre die Erweiterung von SQL um ein entsprechendes in Abbildung 6.19 dargestelltes ALTERNATIVES-Konstrukt. Das Schema einer solchen Alternativen-Tabelle ähnelt dem der zugrundeliegenden Replikationssicht, erweitert um eine zusätzliche laufende Nummer für die Alternativen je Original-Tupel. Die Identifizierung eines alternativen Tupels erfolgt dann durch die Verkettung des Tupel-Primärschlüssel und der Alternativen-Nummer.

6.5.2 Beispiel

Sei neben der bisher betrachteten Verkaufsanwendung zusätzlich die im Abschnitt 3.2.3 beschriebene Terminplanungsanwendung auf dem mobilen Client installiert, die auf Daten der Sicht termine-view innerhalb der Replikatdatenbank my-client-db zugreift. Dann würde die in Abbildung 6.20 dargestellte Anweisung alle Replikationssichten der Replikatdatenbank synchronisieren und für konfligierende Termine entsprechende Alternativen in der Tabelle alternative-termine testen, deren Struktur in Tabelle 6.7 dargestellt ist, wobei der Primärschlüssel über die Attribute Alt-ID, Raum, Datum und Zeit definiert ist.

```
SYNCHRONIZE ALL
OF REPLICA DATABASE my-client-db
ALTER CONFLICT HANDLING FOR INSERT
      IN termine-view USE ALTERNATVIES IN alternative-termine;
```

Abbildung 6.20: Beispiel für SYNCHRONIZE

Alt-ID	Raum	Datum	Zeit	Leiter	Zweckbeschreibung
01	SR123	2003-02-13	16:00	C. Gollmick	Abschlußvortrag DA
02	SR123	2003-02-13	17:00	C. Gollmick	Abschlußvortrag DA
01	SR123	2003-02-20	17:00	J. Lufter	Abschlußvortrag SA

Tabelle 6.7: Beispiel einer Alternativen-Tabelle

Kapitel 7

Bewertung

Dieses Kapitel hat zwei größere Schwerpunkte. Einerseits werden die in dieser Arbeit vorgestellten Konzepte und Verfahren für die Konfliktvermeidung (Abschnitt 7.1) und Konfliktbehandlung (Abschnitt 7.2) nach jeweils aufgestellten Kriterien miteinander verglichen und bewertet. Dabei wird vorallem bei den Verfahren zur Konfliktbehandlung der Aufwand für notwendige Zusatzinformationen untersucht. Andererseits findet im Abschnitt 7.3 eine Bewertung und Gegenüberstellung der beschriebenen Produkte und Projekte statt.

7.1 Bewertung von Konfliktvermeidungsverfahren

Anhand folgender Kriterien werden in diesem Abschnitt Verfahren zur Konfliktvermeidung bewertet und miteinander verglichen. Tabelle 7.1 zeigt anschließend alle hier betrachteten Verfahren in der Übersicht.

- Wie stark schränkt das Verfahren die Verfügbarkeit der Daten auf dem Server ein bzw. welchen Beschränkungen unterliegen die replizierten Daten des Client?
- Verwendet das Verfahren zusätzliches Wissen über Inhalt oder Typ zugrundeliegender Daten zur Vermeidung von Konflikten?
- Ist der Nutzer bzw. die Anwendung bei der Umsetzung des Verfahrens beteiligt?
- Wie groß ist der Aufwand für die Integration des Verfahrens in bestehende DBMS bzw. welche Probleme können bei der Implementierung auftreten?
- Gibt es praxisrelevante Beispiele für den Einsatz des Verfahrens?

7.1.1 Konfliktvermeidung mit Check-out/Check-in

Das in Abschnitt 3.1.3 beschriebene Verfahren schränkt die Verfügbarkeit der exklusiv replizierten Daten auf dem Server stark ein, allerdings sind diese dann für den Client vollständig verfügbar und unterliegen keinen weiteren Einschränkungen. Die Sperrung und Eingliederung von Objekten erfolgt dabei ohne spezielle Informationen über die Daten selbst. Ein Mitwirken von Nutzer oder Anwendung ist ebenfalls nicht notwendig, sieht man

	Verfügbarkeitseinschränkung Server	Verfügbarkeitseinschränkung Client	Verwendung von Semantik	Mitwirkung von Nutzer/Anwendung	Integrationsaufwand	Praxisrelevanz
Konfliktvermeidung durch						
Check-out/Check-in	⊕	⊖	–	–	⊖	⊕
`KEY-POOL`-Verfahren	⊖	○	✓	–	○	⊕
`ESCROW`-Verfahren	○	○	✓	–	○	○
`SLOT`-Verfahren	⊖	⊕	–	✓	⊖	⊖
Begrenzung von Sperren	○	○	–	✓	⊕	⊖
Einschränkung von Operationen	⊖	⊕	–	✓	⊕	⊖

✓: ja –: nein ⊖: gering ○: mittel ⊕: hoch

Tabelle 7.1: Vergleich von Konfliktvermeidungsverfahren

von den anwendungsbezogenen Informationen zur Zusammenfassung von Daten für die Definition eines Objektes ab. Zum Einsatz kommt dieses Verfahren beispielsweise im Bereich komplexer Konstruktionsanwendungen, bei der mehrere Teams an verschiedenen Teilen eines Gesamtobjektes gleichzeitig arbeiten. Durch die strukturelle Zuordnung von Objektteilen zu bearbeitenden Gruppen werden Verfügbarkeitsprobleme vermieden. Da das betrachtete *Check-out/Check-in*-Verfahren effektiv das strikte Zwei-Phasen-Sperrprotokoll implementiert, ist somit der Aufwand zur Integration und Nutzung in bestehende DBMS sehr gering. Allerdings verwenden aktuelle Produkte aus Gründen der Performance nur das normale 2PL. Dadurch ist diese Konfliktvermeidung meist in proprietären Systemen bzw. speziellen Anwendungslösungen zu finden, wie beispielsweise in [HSD+94, SS99] beschrieben.

7.1.2 Konfliktvermeidung mit dem `KEY-POOL`-Verfahren

Wird für konfliktfreie unverbunden durchgeführte Einfügungen das `KEY-POOL`-Verfahren verwendet, muß dazu nicht die gesamte Tabelle für andere Nutzer gesperrt werden. Dadurch bleibt die Verfügbarkeit der Daten auf dem Server auf einem hohen Niveau, allerdings kann der Client nur in dem Umfang der vorher reservierten Primärschlüssel Einfügungen vornehmen, was seine Verfügbarkeit bezüglich möglicher `INSERT`-Operationen einschränkt. Voraussetzung für das Verfahren ist ein künstlicher Primärschlüssel, der für die betreffende Tabelle existieren muß. Zur Ermittlung benötigter Mengen an Primärschlü-

sseln für einen Client können automatisch Statistiken ausgewertet und Anwendungsprofile erstellt werden, allerdings ist ebenso genaues Wissen über die Semantik der Nutzeranwendungen notwendig.

Für die Implementierung eines solchen Verfahrens ist ein gewisser Aufwand zur Verwaltung und Vergabe von client-spezifischen `KEY-POOL`-Mengen nötig. Problematisch im Zusammenhang mit den notwendigen künstlichen Primärschlüsseln ist, daß weder die SQL-Norm noch die Produkte die Festlegung eines Primärschlüssels für eine Tabelle erzwingen. Um für ein einzufügendes Tupel einen der reservierten Primärschlüssel zu vergeben, ist zusätzliche Programmlogik notwendig. Diese kann entweder in das DBMS integriert sein, wie im Beispiel von Sybase [Syb02b], durch andere interne Mechanismen, wie beispielsweise Trigger und UDF, oder durch eine externe Mittelschicht zwischen Anwendung und DBMS realisiert werden.

7.1.3 Konfliktvermeidung mit dem `ESCROW`-Verfahren

Durch die exklusive Reservierung von Teilmengen eines `ESCROW`-Attributes bleibt die restliche Menge für konkurrierende Änderungen auf dem Server verfügbar. Dadurch sind jedoch gleichzeitig die Änderungsmöglichkeiten auf dem mobilen Client für solche Attribute eingeschränkt, da sie nur auf dem reservierten Teilbereich wirken können. Um das `ESCROW`-Verfahren anzuwenden, ist genaues Wissen über die Semantik der betreffenden Attribute notwendig. Desweiteren muß recht genau bekannt sein, in welchem Umfang ein Client unverbundene Änderungen durchführen will. Für die Ermittlung der daraus resultierenden Größe zu reservierender Teilmengen können einerseits Vorgaben des Nutzers bzw. der Anwendung verwendet werden, andererseits sollte durch Auswertung von Statistiken ähnlich wie beim `KEY-POOL`-Verfahren eine automatische Gewinnung entsprechender Parameter angestrebt werden.

Zur Implementierung des `ESCROW`-Verfahrens sind einige Mechanismen und Konzepte zu realisieren, die unter anderem für die Reservierung und Zurückführung von Teilmengen verantworlich sind oder auch die Einhaltung von angegebenen Bedingungen überwachen. Hinzu kommt die Festlegung von `ESCROW`-Attributen und die bereits im Abschnitt 3.2.2 diskutierte Semantik von Reservierungen auf ihnen. Hinweise zur Implementierung des Verfahrens im IMS/VS Fast Path sowie für langlaufende Transaktionen zur Vermeidung von Verfügbarkeitseinbußen auf *Hot Spot*-Objekten finden sich in [GK85, O'N86]. Dort, wie auch im mobilen Umfeld, ist das `ESCROW`-Verfahren jedoch nur für spezielle Attributtypen geeignet.

7.1.4 Konfliktvermeidung mit dem `SLOT`-Verfahren

Werden Einfügungen durch das `SLOT`-Verfahren realisiert, ändert sich die Verfügbarkeit der betroffenen Tabelle kaum, da nur die über `PRE-INSERT` eingefügten Tupel gegen konkurrierende Änderungen gesperrt werden müssen. Auf dem Client sind allerdings konfliktfreie Einfügungen, wie beim `KEY-POOL`-Verfahren, nur im Rahmen der reservierten Tupel möglich. Da die Konfliktvermeidung effektiv durch ein normales `INSERT` zum Zeitpunkt der Replikation realisiert wird, sind hierfür keine zusätzlichen semantischen Informationen notwendig. Die Werte zur Belegung des natürlichen Schlüssels einzufügender Tupel müssen jedoch vom Nutzer bzw. der Anwendung bereitgestellt werden.

Eine Implementierung dieses Verfahrens in bestehende DBMS kann insofern ohne großen Aufwand vorgenommen werden, da es nur auf bereits existierende Mechanismen (`INSERT`, Sperren von Tupeln) aufbaut. Einzig die Bereitstellung der einzufügenden Schlüsselinformation vor der eigentlichen (unverbundenen) Einfügung erfordert die Kenntnis über das genaue Anwendungs- bzw. Nutzerprofil. Diese Daten können also nur bereitgestellt werden, wenn schon zur Replikation feststeht, welche Einfügungen vorgenommen werden sollen. Dies ist nicht für alle Anwendungsszenarien möglich. Eine praxisrelevante Verwendung für das `SLOT`-Verfahren ist, im Gegensatz zum `KEY-POOL`-Verfahren, nicht bekannt.

7.1.5 Konfliktvermeidung durch Begrenzung von Sperren

Werden Sperren in ihrer Gültigkeit begrenzt, ist die Verfügbarkeit gesperrter Daten auf dem Server nicht mehr vom Verhalten des mobilen Client abhängig. Durch die Freigabe exklusiv gesperrter Daten nach einer gewissen Frist oder bei anwachsenden Zugriffsanforderungen ist die Einschränkung der Verfügbarkeit nicht so groß gegenüber dem *Check-out/Check-in*-Verfahren. Entsprechend muß der Client mit einer Beschränkung seiner Zugriffsrechte während der mobilen Arbeit rechnen. Zusätzliche semantische Informationen zu den gesperrten Daten sind hierfür nicht nötig. Die Festlegung von Fristen muß in Abstimmung zwischen Nutzer, Anwendung und dem System bekannten Zugriffsstatistiken erfolgen.

Bestehende DBMS besitzen zwar im allgemeinen bereits ein Sperrprotokoll, allerdings müßte dies so modifiziert bzw. erweitert werden, daß es auch für zeitlich befristete Sperren geeignet ist. Zusätzlich werden Heuristiken oder ähnliche Quellen zur Bestimmung von Parametern benötigt, um beispielsweise die Dauer einer Sperre als Kompromiß zwischen Verfügbarkeit und Nutzeranforderung festzulegen. Dieser hohe Aufwand ist auch ein Grund dafür, daß das Konzept der begrenzten Sperren bisher nur im Forschungsprojekt MobiSnap bzw. in unserem in Kapitel 6 vorgestellten verwendet wird. Eine Implementierung in existierenden Produkten ist dagegen nicht bekannt.

7.1.6 Konfliktvermeidung durch Einschränkung von Operationen

Werden Daten nur für einzelne Operationen exklusiv repliziert, so können dieselben Daten für weitere, nicht konfligierende, Operationen anderer Clients ebenfalls freigegeben werden. Dadurch ist die Verfügbarkeit nicht so stark eingeschränkt im Vergleich zu einem exklusiven, aber operationsunspezifischen, Zugriff eines Client. Dagegen hat der Client auch nur einen auf die festgelegten Operationen beschränkten Zugriff auf die replizierten Daten. Für die Realisierung dieses Vorgehens sind keine weiteren semantischen Informationen notwendig, da hierfür die in Tabelle 3.1 dargestellten Verträglichkeiten ausgewertet werden. Dafür ist allerdings vom Nutzer bzw. der Anwendung genau festzulegen, welche Operationen auf welchen Daten geplant sind, damit dieses Wissen zur Freigabe kompatibler Operationen von konkurrierenden Clients verwendet werden kann.

Problematisch ist die Implementierung in bestehende Systeme aufgrund der Tatsache, daß diese keine Unterscheidung zwischen Sperren für Änderungs-, Einfüge- und Löschoperationen vornehmen, sondern kategorisch eine Schreibsperre vergeben. Erschwerend kommt hinzu, daß es keine Schnittstelle zur Sperrverwaltung eines DBMS gibt, die man über eine Mittelschicht ansprechen könnte sowie der recht hohe Aufwand zur Protokollierung

der einzelnen Operationsanforderungen auf den verschiedenen Daten. In der Praxis wird dieses Verfahren nicht eingesetzt.

7.2 Bewertung von Konfliktbehandlungsverfahren

Abhängig vom verwendeten Verfahren zur Konfliktbehandlung, d.h. Konflikterkennung oder Konfliktauflösung, werden bestimmte zusätzliche Informationen für deren Einsatz benötigt. Diese Zusatzinformationen lassen sich wie folgt klassifizieren:

- **intern:**
  Die benötigten Informationen können ohne äußere Einflüsse automatisch vom DBMS erzeugt werden bzw. direkt oder indirekt aus den Protokolldaten gewonnen werden.
- **extern:**
  Die benötigten Informationen können nur aus externen Quellen gewonnen werden.
  - **automatische Erzeugung:**
    Die benötigten Informationen können automatisch durch externe Quellen bereitgestellt werden.
  - **manuelle Erzeugung:**
    Die benötigten Informationen können nur durch den manuellen Eingriff eines Nutzers oder Administrators erzeugt werden.

Der DBMS-interne Aufwand für die Beschaffung von Zusatzinformationen wird ganz allgemein durch drei große Faktoren beeinflußt [HR01], die im Zusammenspiel die *Performance* eines Systems maßgeblich festlegen, wobei im Datenbankbereich vorallem die Anzahl von Lese- und Schreibzugriffen (I/O) eine entscheidende Rollen spielen.

I/O: Darunter versteht man den Aufwand, eine bestimmte Information über die Ein- und Ausgabekanäle eines Computers zu transferieren. Im speziellen soll darunter der Aufwand für Plattenzugriffe sowohl lesender aber vorallem auch schreibender Art zusammengefaßt sein.

CPU: Mit CPU ist der eigentliche Berechnungsaufwand gemeint, um eine Information aus gegebenen Parametern zu ermitteln. Meist ist dieser Anteil im Vergleich zu den anderen nicht signifikant, unter anderem begründet durch die immer leistungsfähigeren Prozessoren in stationären wie auch mobilen Geräten.

Speicher: Müssen die gewonnen Informationen bis zum nächsten Synchronisationszeitpunkt persistent aufbewahrt werden, wird der dafür benötigte zusätzliche Speicherplatz von dieser Kenngröße erfaßt. Auch hier ist zu beobachten, daß immer größere Kapazitäten auf immer kleinerem Raum möglich sind und dadurch der Speicher-Faktor analog zum CPU-Faktor nicht von allzu großer Bedeutung ist.

Zusätzlich zur obigen Klassifikation muß unterschieden werden, ob die benötigten Informationen auf dem Client oder dem Server gewonnen werden. Eine genauere Betrachtung dieser sowie weiterer Kriterien findet in den nächsten Abschnitten statt. Dabei werden entsprechend der Gliederung dieser Arbeit sowohl Verfahren zur Konflikterkennung im Abschnitt 7.2.1 als auch zur Konfliktauflösung im Abschnitt 7.2.2 bewertet und verglichen.

7.2.1 Verfahren zur Konflikterkennung

Anhand folgender Kriterien sollen die in der Arbeit betrachteten Verfahren zur Konflikterkennung bewertet und verglichen werden. Eine zusammenfassende Übersicht findet sich anschließend in Tabelle 7.2.

- Welche zusätzlichen Informationen zu den einzubringenden Daten bzw. über deren Inhalt oder Typ benötigt die Konflikterkennung?
- Mit welchem Aufwand ist die Beschaffung der benötigten Zusatzinformationen verbunden, wo sind diese in obiger Klassifikation einzuordnen?
- Ist die Gewinnung der Zusatzinformationen synchron während der Transaktionsausführung notwendig oder asynchron zum Reintegrationszeitpunkt möglich?
- Werden die zusätzlichen Informationen auf dem Client oder Server gewonnen?
- Benötigt das Verfahren eine operations- oder datenorientierte Reintegration?
- Wie groß ist der Aufwand für die Integration des Verfahrens in bestehende DBMS bzw. welche Probleme können bei der Implementierung auftreten?
- Gibt es praxisrelevante Beispiele für den Einsatz des Verfahrens?

	Zusatzinformationen					Art der Reintegration	Integrationsaufwand	Praxisrelevanz
	Art der Information	Zusätzlicher Aufwand	Verantwortlichkeit	Herkunft	synchron/asynchron			
Konflikterkennung mit								
Abbildern	ΔBI	−	i/a	C/S	−/✓	da/op	⊖	⊕
Zeitstempeln	$TS(T)$	✓	e/a	C/S	✓/−	op	⊕	○
RS/WS	RS	✓	i/a	C/S	✓/✓	op	○	⊖

✓: ja −: nein ⊖: gering ○: mittel ⊕: hoch
da: datenorientiert m: manuell i: intern C: Client
op: operationsorientiert a: automatisch e: extern S: Server

Tabelle 7.2: Vergleich von Konflikterkennungsverfahren

7.2.1.1 Konflikterkennung mit Abbildern

Bei der Konflikterkennung über den Abbildervergleich von replizierten Daten steht, wie im Abschnitt 4.3.3 beschrieben, vorallem die Beschaffung der Änderungen zwischen dem

BI und AI sowohl des Client als auch des Server im Vordergrund, d.h. es werden ΔBI^C und ΔBI^S benötigt. Diese lassen sich jedoch ohne zusätzlichen Aufwand nach Vollendung der unverbundenen Arbeit aus dem Log des Client bzw. Server gewinnen. Dazu ist allerdings die fortlaufende Protokollierung mit einem sogenannten *Recovery-Log* notwendig, d.h. es darf kein zyklisches Log verwendet werden. Die Gewinnung des ΔBI ist also ohne Veränderung des DBMS und automatisch möglich. Zusätzlich läßt sich aus dem Log die verursachende Änderungsoperation ermitteln, sodaß eine Reintegration auf daten- und operationsorientierte Art und Weise möglich ist. Der Aufwand für die Implementierung ist aufgrund der einfach und ohne zusätzliche Kosten zu gewinnenden Informationen sehr gering. Dadurch hat dieses Verfahren, allerdings auf einer datenorientierten Reintegration basierend, eine sehr große Praxisrelevanz und wird von allen kommerziellen System verwendet.

7.2.1.2 Konflikterkennung mit Zeitstempeln

Wie im Abschnitt 4.3.2 beschrieben, benötigt die operationsorientierte Reintegration mit Zeitstempeln entsprechende zeitliche Informationen zu Daten und Operationen, d.h. wann welches Datenobjekt durch welche Operation verändert wurde. Diese können nicht durch das DBMS selbst bereitgestellt werden, sondern müssen von einer externen Zeittaktung automatisch erzeugt werden. Da jedoch Zeitstempel während der Transaktionsausführung zwischen Client und Server synchron vergeben werden müssen, ist aufgrund der vorausgesetzten unverbundenen Arbeit des mobilen Client dieses Verfahren nicht bzw. nur unter hohem zusätzlichen Aufwand verwendbar. Die im Abschnitt 4.3.2 beschriebene Modifikation für das mobile Szenario ist dabei eine restriktivere Variante der Konflikterkennung mit RS/WS und deswegen nicht für eine Implementierung geeignet. Im Gegensatz dazu werden mitunter Zeitstempel zur Konflikterkennung in festverbundenen replizierten Umgebungen eingesetzt, wie beispielsweise bei *Oracle9i Replication* [Ora02b].

7.2.1.3 Konflikterkennung mit *RS/WS*

Für diese Art der Konflikterkennung ist die Bereitstellung des *Read Set (RS)* bzw. *Write Set (WS)* sowohl auf dem Client als auch auf dem Server notwendig. Diese können automatisch durch Auswertung gewisser DBMS-interner Informationen gewonnen werden. Dabei wird jeweils nur der notwendig existierende Primärschlüssel und nicht der Inhalt eines gelesenen bzw. geschriebenen Tupels aufgezeichnet. Das WS kann relativ einfach durch Auswertung des Log gewonnen werden und wird hier nicht weiter betrachtet. Dagegen ist die Gewinnung des RS extrem aufwendig, vorallem wenn synchron für jeden lesenden Zugriff einer Transaktion gleichzeitig eine Schreibaktion zwecks Protokollierung stattfindet. Dadurch wird die I/O-Performance stark eingeschränkt.

Eine etwas günstigere Variante ist die asynchrone Auswertung lokaler Sperrtabellen auf dem mobilen Client, aus denen ebenfalls die für lesenden und schreibenden Zugriff gesperrten Daten gewonnen werden können. Der Aufwand zur Integration dieses Verfahrens in bestehende DBMS beschränkt sich auf die Entwicklung von Methoden zur Gewinnung des RS und WS, die jedoch möglichst geringe Performance-Einschränkungen zur Folge haben sollten. Aufgrund fehlender Implementierungen dieser Methoden vorallem für das RS, wird dieses Verfahren zur Konflikterkennung in keinem Produkt im Sinne theoretischer Ansätze verwendet.

7.2.2 Verfahren zur Konfliktauflösung

Nach den im folgenden vorgestellten Kriterien sollen die betrachteten Verfahren zur Konfliktauflösung bewertet und verglichen werden. Die Übersicht in Tabelle 7.2 stellt alle Verfahren zusammenfassend gegenüber.

- Welche zusätzlichen Informationen zu den einzubringenden Daten bzw. über deren Inhalt oder Typ benötigt die Konfliktauflösung?
- Mit welchem Aufwand ist die Beschaffung der benötigten Zusatzinformationen verbunden, wo sind diese in der anfangs aufgezeigten Klassifikation einzuordnen?
- Ist die Gewinnung der Zusatzinformationen synchron während der Transaktionsausführung notwendig oder asynchron zum Reintegrationszeitpunkt möglich?
- Werden die zusätzlichen Informationen auf dem Client oder Server gewonnen?
- Benötigt das Verfahren eine operations- oder datenorientierte Reintegration?
- Führt die Konfliktauflösung garantiert oder nur mit einer gewissen Wahrscheinlichkeit zum Erfolg?
- Wie groß ist der Aufwand für die Integration des Verfahrens in bestehende DBMS bzw. welche Probleme können bei der Implementierung auftreten?
- Gibt es praxisrelevante Beispiele für den Einsatz des Verfahrens?

7.2.2.1 Konfliktauflösung durch Tupelauswahl

Hierunter werden alle Verfahren zusammengefaßt, die eine Entscheidung zwischen zwei konfligierenden Tupeln während der Reintegrationsphase zugunsten *eines* Tupels treffen. Dazu muß zuvor auf dem Server durch den Administrator eine solche Regel, welche entweder die Änderung des Client verwirft (`server wins`) oder den Zustand des Server überschreibt (`client wins`), festgelegt werden. Durch diese Entscheidung ist in jedem Fall ein Konflikt gelöst. Abgesehen von dem einmaligen Aufwand, die zur Anwendungssemantik passende Entscheidungsregel festzulegen, entstehen hierbei keine weiteren Kosten. Eine mögliche Verfeinerung dieser recht starren inhaltsunabhängigen Regeln ist durch Auswertung existierender Prioritäten (`priority`) von Nutzern oder Geräten möglich. Diese simple Konfliktauflösung ist sowohl für eine datenorientierte als auch operationsorientierte Reintegration ohne Probleme implementierbar und wird von allen Produkten für den mobilen Datenbankbereich unterstützt.

7.2.2.2 Konfliktauflösung durch Alternativen

Sollen Einfüge- oder Änderungskonflikte durch alternative Einfügungen bzw. Änderungen gelöst werden, müssen dafür zu verwendende Tupel als zusätzliche Informationen generiert werden. Hierbei ist vorallem der Nutzer bzw. die Anwendung eines mobilen Client gefragt, da ein DBMS diese Aufgabe nicht automatisieren kann. Dadurch entsteht einerseits für den Nutzer ein gewisser Aufwand, alternative Tupel im Sinne der Anwendungssemantik zu

	Zusatzinformationen								
	Art der Information	Zusätzlicher Aufwand	Verantwortlichkeit	Herkunft	synchron/asynchron	Art der Reintegration	Garantierte Konfliktauflösung	Integrationsaufwand	Praxisrelevanz
Konfliktauflösung durch									
Tupelauswahl	Regel	−	e/m	S	−/✓	da	✓	⊖	⊕
Alternativen	Tupel	✓	e/m	C	✓/−	op	−	○	⊖
Datentransformation	Funktion	✓	e/m	S	−/✓	da/op	✓	○	⊕
Operationstransf.	Regel	−	e/m	S	−/✓	op	✓	⊖	⊖
Akzeptanzkriterium	Test	✓	e/m	C/S	✓/✓	op	−	⊕	⊖
Versionierung	Versionen	✓	i/a	S	−/✓	op	−	⊕	○

✓: ja −: nein ⊖: gering ○: mittel ⊕: hoch
da: datenorientiert m: manuell i: intern C: Client
op: operationsorientiert a: automatisch e: extern S: Server

Tabelle 7.3: Vergleich von Konfliktauflösungsverfahren

erzeugen, andererseits benötigen diese zusätzlichen Speicherplatz und müssen synchron zur Ausführung der eigentlichen Operation angelegt werden. Diese Art der Konfliktauflösung, welche nur mit Kenntnis der zugrundeliegenden Operation möglich ist, hat zudem den Nachteil nicht immer einen Konflikt auflösen zu können, nämlich dann, wenn alle angegebenen Alternativen ebenfalls zum Konflikt führen. Eine Implementierung dieser Konfliktauflösung wurde bereits im Bayou-System [GHOS96] auf prozeduralem Weg vollzogen und ist auf deklarative Art und Weise in unserem vorgestellen Ansatz der nutzerdefinierten Replikation vorgesehen. In Produkten hat die Bereitstellung von Alternativen bisher noch keinen Einsatz gefunden.

7.2.2.3 Konfliktauflösung durch Datentransformation

In dieser Kategorie werden Verfahren zusammengefaßt, die Konflikte auflösen, indem sie konfligierende Daten so transformieren, daß sie konfliktfrei reintegriert werden können. Hierzu zählt sowohl die Umbenennung eines Primärschlüssels zur Auflösung von Eindeutigkeitsverletzungen als auch die Kombination zweier konfligierender Tupel zu einem neuen. Zusätzliche Informationen sind vorallem entsprechende Funktionen, die im einfachsten Fall schon vom DBMS bereitgestellt (`avg, sum`) oder aufwendiger vom Administrator selbst definiert (`udf`) werden können. Diese auf dem Server festgelegten Funktionen müssen allerdings Ergebnisse im Sinne der Anwendungssemantik produzieren und werden

zum Reintegrationszeitpunkt aufgerufen. Basierend auf einer daten- oder operationsorientierten Reintegration kann eine erfolgreiche Konfliktauflösung bei der `udf`-Verwendung durch ein entsprechendes Funktionsdesign garantiert werden, die vom DBMS bereitgestellten numerischen Funktionen wie `avg` ergeben für passende Attributtypen ebenfalls immer eine Lösung. Allerdings sind bei einer Umbenennung des Primärschlüssels mögliche Abhängigkeiten, beispielsweise bei Fremdschlüsselbeziehungen, zu beachten. Der Aufwand zur Implementierung solcher Verfahren liegt dabei vorallem in der Definition sinnvoller und nützlicher Funktionen, die automatisch Konflikte im Sinne der Anwendung lösen. Produkte bieten oft nur eine kleine Auswahl numerischer Funktionen sowie die Einbindung nutzerdefinierter Funktionen über die *UDF*-Schnittstelle.

7.2.2.4 Konfliktauflösung durch Operationstransformation

Wurde ein Änderungskonflikt aufgrund eines bereits gelöschten Tupels festgestellt, so kann das einzubringende `UPDATE` auf diesem Tupel zu einem `INSERT` umgewandelt werden. In ähnlicher Weise kann ein Einfügekonflikt durch das `INSERT` eines bereits existierenden Tupels mit gleichem Primärschlüssel in ein `UPDATE` auf das Tupel transformiert werden. Ob und wann solche Transformationen vorgenommen werden können, muß vom Administrator unter Beachtung der Anwendungssemantik in einer einfachen Regelung für den Server festgelegt werden. Für die Anwendung während der operationsorientierten Reintegration eines Client ist das Wissen über die verursachende Operation notwendig. Die Konfliktauflösung ist dann in jedem Fall erfolgreich, da entweder ein nicht existierendes Tupel eingefügt wird oder ein bestehendes Tupel geändert wird. Eine Implementierung ist demnach ohne größeren Aufwand möglich, wurde in den aktuellen Produkten aber nicht realisiert, da diese mit einer datenorientierten Reintegrationsstrategie arbeiten.

7.2.2.5 Konfliktauflösung durch Akzeptanzkriterium

Für die operationsorientierte Reintegration von lokal abgeschlossenen Transaktionen auf dem Server besteht auf Grundlage des Two-Tier-Modells [GHOS96] die Möglichkeit, die Konfliktbehandlung durch Angabe und Auswertung eines Akzeptanzkriteriums zu ersetzen. Dieser auf der Semantik einer Anwendung basierende Test muß dafür jedoch zuvor definiert werden. Dies ist üblicherweise die Aufgabe eines Administrator, kann aber auch von einem Nutzer durchgeführt werden, dann allerdings synchron zu lokalen Änderungstransaktionen auf dem mobilen Client. Das Akzeptanzkriterium prüft anwendungsbezogen, ob die einzubringenden Änderungen auf den aktuellen Daten gültig sind. Dadurch ist im Fall der Nichterfüllung des Tests nicht immer eine sichere Auflösung garantiert. Vorallem der manuelle administrative Aufwand zur Erzeugung solcher Testkriterien ist extrem hoch und erschwert die Implementierung in bestehende DBMS. Abgesehen von einer Anwendung dieses Verfahrens im Bayou-System [GHOS96] gibt ist keine Realisierung in existierenden Produkten.

7.2.2.6 Konfliktauflösung durch Versionierung

Neben den bisher betrachteten Varianten zur Konfliktauflösung bieten Mehrversionsverfahren die Möglichkeit, Konflikte durch Verwaltung zweitweise mehrerer Versionen eines

Datenelementes zu lösen. Die dafür nötigen zusätzlichen Versionen auf Seiten des Server benötigen einerseits erheblich mehr Speicherplatz, andererseits ist der algorithmische Aufwand zur Bestimmung passender Versionen sehr hoch. Die Versionierung übernimmt dabei das entsprechend ausgestattete DBMS automatisch. Die Verwendung und Erzeugung der Versionen erfolgt asynchron zur eigentlichen Arbeit des mobilen Client im Zuge der operationsorientierten Reintegration auf dem Server, d.h. der Client ist hiervon nicht weiter betroffen. Eine sichere Konfliktauflösung kann nicht garantiert werden, da nicht in jedem Fall eine passende Datenversion gefunden werden kann und lokale Änderungen dann abgewiesen bzw. manuell aufgelöst werden müssen. Vorallem aufgrund des hohen algorithmischen Aufwandes, wie unter anderem in [Lie01] beschrieben, ist die Implementierung dieses Verfahrens nur mit entsprechenden Performance-Einbußen möglich. Eine Versionierung könnte dabei durch eine jeweils zusätzliche Tabellenspalte als Versionsnummer simuliert werden. Nach [HR01] sind jedoch bereits einige Produkte, beispielweise von Oracle, mit dem Mehrversionskonzept realisiert.

7.3 Bewertung von Produkten und Forschungsprojekten

Zum Abschluß des Bewertungsteils sollen die beiden vorgestellten Produkte Oracle9*i* Lite und IBM DB2 DataPropagator 8 sowie die beiden Forschungsprojekte MobiSnap und unser eigener Ansatz zur Integration von Konfliktbehandlungsmethoden in die nutzerdefinierte Replikation miteinander verglichen werden. Hierzu werden folgende Kriterien verwendet:

- Wie und durch wen werden die zu replizierenden Daten eines mobilen Client ausgewählt bzw. festgelegt?
- Welche Möglichkeiten zur Konfliktvermeidung bietet das Produkt bzw. Forschungsprojekt an, wie und von wem werden diese festgelegt?
- Welche Möglichkeiten zur Konflikterkennung und -auflösung stehen zur Verfügung, wie und von wem werden diese ausgewählt?

In Tabelle 7.4 werden Produkte und Forschungsprojekte in einer Gesamtübersicht dargestellt.

7.3.1 Oracle9*i* Lite

Für die Replikation werden Daten über ein *Publish & Subscribe*-Verfahren bereitgestellt. Dabei werden in einer ersten Stufe mit der Methode *Publish* die für die Replikation zur Verfügung stehenden Daten vom Administrator festgelegt. Dieser ist ebenfalls in einer zweiten Phase für die Auswahl der client-spezifischen Daten über *Subscribe* verantwortlich, da mobile Anwendungen zentral definiert und erzeugt werden. Hierbei ist gleichzeitig darauf zu achten, Konflikte zwischen unverbunden ausgeführten Änderungen durch ein entsprechendes Anwendungsdesign zu vermeiden, da Oracle9*i* Lite keinerlei Möglichkeiten hierzu anbietet. Die Identifizierung von Tupeln erfolgt über den Primärschlüssel, sodaß Einfüge- und Löschkonflikte entsprechend durch doppelte bzw. fehlende Primärschlüssel erkannt werden. Ein Änderungskonflikt entsteht, falls es einen Unterschied zwischen dem replizierten (BI_X^C) und aktuell auf dem Server vorliegenden Wert (CI_X^S) eines Tupels X

gibt [Ora02a]. Wurde ein Konflikt erkannt, gibt es für dessen automatische Auflösung nur die Möglichkeit, zwischen der Regel `server wins` und `client wins` zu wählen. Diese Entscheidung wird anwendungsabhängig von einem Administrator getroffen. Durch Definition von Triggern können für eine Datenbanktabelle genauere und nutzerdefinierte Auflösungsmethoden integriert werden. Alle nicht automatisch lösbaren Konflikte müssen manuell durch den Nutzer bzw. Administrator behandelt werden.

7.3.2 IBM DB2 DataPropagator

Ähnlich wie bei Oracle9*i* Lite werden auch hier die zur Replikation freigegebenen Daten von Benutzertabellen über ein zweistufiges *Publish & Subscribe*-Verfahren vom Administrator definiert. Zur Vermeidung von Konflikten bietet auch der IBM DB2 DataPropagator keinerlei vordefinierten Konzepte, lediglich einige Hinweise für den entsprechenden Anwendungsentwurf bezüglich einer möglichen Fragmentierung der Daten werden gegeben. Die Konflikterkennung durch *Apply* arbeitet datenorientiert auf Grundlage des Vergleiches der Änderungen ΔBI korrespondierender Tupel auf dem Server und Client. Es lassen sich dabei verschiedene Level zur Konflikterkennung von einem zuständigen Administrator wählen, wie im Abschnitt 5.2 beschrieben. Da von Grund auf der Zustand des Servers als korrekt und konsistent angenommen wird, werden Konflikte nach der Regel `server wins` gelöst, d.h. konfligierende lokale Transaktionen vorerst abgewiesen und beim nächsten Synchronisationsvorgang erneut versucht zu reintegrieren. In der Regel ist in einem solchen Fall allerdings ein manueller Eingriff zur endgültigen Auflösung notwendig.

7.3.3 MobiSnap

Die unverbundene Arbeit orientiert sich in MobiSnap, wie im Abschnitt 5.3 beschrieben, sehr stark an der zugrundeliegenden Anwendungssemantik. Für deren Umsetzung ist eine vom Nutzer erzeugte *mobile Transaktion* in Form einer Prozedur zuständig, wie in Abbildung 5.4 dargestellt. In dieser wird über ein SQL-Statement die Menge der benötigten Daten zur Replikation angefordert. Zur Konfliktvermeidung lassen sich in MobiSnap einige Reservierungen auf gewisse Daten anfordern, wie beispielsweise die Verwendung einer bestimmten Mengeneinheit (`ESCROW`) oder das Recht auf Einfügung bestimmter Tupel. Um dennoch die Verfügbarkeit nicht zu sehr einzuschränken, sind vergebene Reservierungen nur eine bestimmte Zeitdauer gültig. Optimistisch vorgenommene lokale Änderungen müssen beim Reintegrieren auf Konflikte überprüft werden. Dies wird ebenfalls durch entsprechende Anwendungslogik innerhalb der mobilen Transaktion realisiert, beispielsweise durch Abfrage bestimmter Werte. Dieses Vorgehen ähnelt dem Akzeptanzkritierium, welches für das Bayou-System [GHOS96] entwickelt wurde. Schließlich müssen erkannte Konflikte ebenfalls durch Definition geeigneter Auflösungsmethoden innerhalb der mobilen Transaktion verarbeitet werden, wobei MobiSnap hier großen Wert auf die gezielte Benachrichtigung des Nutzers legt.

7.3.4 Eigener Ansatz

Unser im Kapitel 6 vorgestellter Ansatz zur Integration der Konfliktbehandlung in die nutzerdefinierte Replikation hat vorallem das Ziel, die zur unverbundenen Arbeit nötigen

Festlegungen und Definitionen auf deklarative Weise zu ermöglichen. Hierzu zählt insbesondere der Prozeß der Replikation, bei dem analog zum *Publish & Subscribe*-Verfahren, zuerst die konsolidierten Tabellen als Replikationsgrundlage mittels der deklarativen Anweisung `CREATE CONSOLIDATED TABLE` durch den Administrator festgelegt werden. Dieser wählt hierbei von Datentyp bzw. Operation abhängige Methoden und Default-Werte zur Konfliktvermeidung bzw. Konfliktauflösung für eine solche Tabelle aus. Neben der exklusiven Reservierung und zeitlich befristeten Zusicherungen wird hier auch das `KEY-POOL`-Verfahren für konfliktfreie Einfügungen sowie das `ESCROW`-Konzept angeboten. Für optimistisch durchgeführte Änderungen können basierend auf einer Konflikterkennung durch Abbildervergleich je nach zugrundeliegender Operation verschiedene Auflösungsmöglichkeiten festgelegt werden, wie beispielsweise einfache Regelanwendungen zur Tupelauswahl oder Methoden zur Daten- bzw. Operationstransformation. In einer zweiten Stufe repliziert der Nutzer bzw. die Anwendung benötigte Daten über die Anweisung `CREATE REPLICATION VIEW` auf den mobilen Client. Dabei können Default-Werte für entsprechende Konfliktvermeidungsverfahren durch anwendungsspezifische Parameter ersetzt werden. Schließlich können in der letzten Phase der Synchronisation wiederrum auf deklarativem Weg alternative Einfügungen für konfligierende Tupel angegeben werden, die allerdings zuvor vom Nutzer erzeugt werden müssen.

	Replikation				Konfliktvermeidung					K.-erkennung			Konfliktauflösung					
	Publish		Subscribe			Verfahren								Verfahren				
	Verantwortlichkeit	Verfahren	Verantwortlichkeit	Verfahren	Auswahl/Festlegung	Exklusives Sperren	KEY-POOL-Verfahren	ESCROW-Verfahren	Sperrbegrenzung	Primärschlüssel-Identifizierung	Abbildervergleich	Anwendungsdesign	Auswahl/Festlegung	Tupelauswahl	Alternativen	Datentransformation	Operationstransformation	Manuelle Auflösung
Oracle9*i* Lite	A	P	A	P	A[1]/–	–	–	–	–	✓	✓	–	A/P	✓	–	✓	–	✓
IBM DB2 DataPropagator 8	A	P	A	P	A[1]/–	–	–	–	–	✓	✓	–	A/P	✓[2]	–	–	–	✓
MobiSnap	–	–	N	P	N/P	✓	–	✓	✓	✓	–	✓[3]	N[3]/P	–	✓	–	–	✓
Eigener Ansatz	A	D	N	D	A/N[4]/D	✓	✓	✓	✓	✓	✓	–	A/N[5]/D	✓	✓	✓	✓	✓

✓: ja A: Administrator P: prozedural

–: nein N: Nutzer/Anwendung D: deklarativ

Tabelle 7.4: Vergleich von Produkten und Forschungsprojekten

[1] Konfliktvermeidung über das Anwendungsdesign

[2] es gibt nur die vom System festgelegte Regel `server wins`

[3] Konflikterkennung und -behandlung findet innerhalb der sogenannten *mobilen Transaktion* statt

[4] Festlegung durch den Administrator, Parametrisierung durch den Nutzer

[5] Festlegung durch den Administrator, Veränderung während der Synchronisation durch den Nutzer möglich

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

Die Arbeit hat einen Überblick zu Verfahren gegeben, die zur Vermeidung bzw. Erkennung und Auflösung von Konflikten dienen, die im hier zugrundegelegten Szenario der unverbundenen Arbeit auftreten können. Diese Verfahren wurden unter Aspekten wie Verfügbarkeit und notwendigen Zusatzinformationen miteinander verglichen und bewertet. Dabei hat sich gezeigt, daß strikte Sperrverfahren aufgrund der schlechten Verfügbarkeit, beispielsweise für Anwendungen wie das HERMES-System, nur eingeschränkt verwendet werden können. Um dennoch möglichst viele Konflikte zu vermeiden, kann die bekannte Semantik von Daten oder Operationen genutzt werden, wie beispielsweise mit dem `KEY-POOL`- oder `ESCROW`-Verfahren. Müssen Konflikte erkannt werden, hat sich gegenüber den anderen in Kapitel 4 betrachteten Verfahren hierfür nur ein Vergleich der Primärschlüssel und Datenabbilder von Tupeln als sinnvoll erwiesen. Dieses Verfahren läßt sich praktisch implementieren, da nur geringer Aufwand für die Beschaffung zusätzlicher Informationen notwendig ist. Allerdings sollte die Reintegration operationsorientiert durchgeführt werden, um gegenüber der rein datenorientierten Reintegration, wie sie in den meisten Produkten eingesetzt wird, vielfältigere Varianten zur Konfliktauflösung anwenden zu können. Diese basieren neben einfacher Regelentscheidung größtenteils auf der zugrundeliegenden Anwendungssemantik und ermöglichen dadurch häufiger eine erfolgreiche automatische Konfliktbehandlung. Dafür muß jedoch zuvor diese Semantik dem System in geeigneter Weise bekannt gemacht werden.

In einem zweiten großen Schwerpunkt dieser Arbeit wurde eine Integration der vorgestellten Verfahren in das Konzept der nutzerdefinierten Replikation vorgenommen. Durch die strikte Gliederung in Replikationsschemadefinitionsebene, Replikatdefinitionsebene und Synchronisationsebene wird eine klare Trennung von Konzept und Ausführung der zur Replikation notwendigen Aufgaben in einem mobilen Szenario gewährleistet. Die dabei verwendete deskriptive Schnittstelle führt zu einer besseren Übersichtlichkeit und vereinfacht die Benutzung durch Administrator bzw. Anwendung. Eine größere Skalierbarkeit gegenüber herkömmlichen Systemen wird durch die Aufgabenverlagerung zum Nutzer bzw. seiner Anwendungen erreicht. Zwar sind auch in unserem Ansatz gewisse Festlegungen zentral über das `CREATE CONSOLIDATED TABLE`-Statement und weitere in [Mül03] beschriebene Anweisungen vom Administrator durchzuführen, diese sind allerdings nur zur allgemeinen nutzerunabhängigen Initialisierung einer Replikationsumgebung notwen-

dig. Die individuelle Datenauswahl und Parametrisierung von Konfliktvermeidungsmethoden ist durch die Anwendung auf dem mobilen Client mit Hilfe der Anweisung `CREATE REPLICATION VIEW` vorzunehmen. Im Gegensatz dazu wird diese Aufgabe in den aktuellen Produkten von Skripten übernommen, die zuvor aufwendig und in Abstimmung mit dem Nutzer durch den Administrator erzeugt werden müssen.

8.2 Ausblick

Ausgehend vom hier zugrundegelegten Modell der unverbundenen Arbeit, könnte durch Erweiterung der Kommunikations- und Synchronisationsmöglichkeiten zusätzliche Funktionalität eingebracht werden. Hierzu zählt unter anderem die Nutzung asynchroner drahtloser Kommunikationsmittel, wie Broadcast oder SMS, um beispielsweise während der unverbundenen Arbeitsphase den mobilen Client über die Gültigkeit seiner zeitlich begrenzten Reservierungen in Form eines sogenannten *Invalidation Report* zu informieren. Läßt man bezüglich langlaufender Transaktionen zwischenzeitliche Synchronisationen mobiler Clients zu, könnten einerseits nicht mehr benötigte Sperren freigeben und entsprechende lokale Änderungen reintegiert werden, andererseits ließen sich zeitlich begrenzte Reservierungen verlängern sowie Mengen für das `KEY-POOL`- bzw. `ESCROW`-Verfahren nachfordern. Für eine vorzeitige Freigabe von Sperren könnte beispielsweise das in [KS91] beschriebene Verfahren eines *graphenbasierten Sperrprotokolls* verwendet werden, welches auf einer physischen oder logischen Zugriffshierarchie bezüglich der betrachteten Daten aufbaut. Allerdings erfordern diese Erweiterungen umfangreiche Eingriffe in das DBMS und benötigen neue Transaktionsmodelle.

Nachdem im Kapitel 6 beschriebenen Sprachentwurf zur Integration von Verfahren zur Konfliktvermeidung und -auflösung in die nutzerdefinierte Replikation, könnte in einer anschließenden Arbeit die prototypische Realisierung und Implementierung folgen. Ein entsprechender Ansatz für die Funktionalitäten des *Replication Proxy Server (RPS)* wird in [Mül03] beschrieben. Denkbar ist hierbei auch die Einführung einer Hierarchie von mehreren RPS, um ein verteiltes Konfliktmanagement zu gewährleisten. Gleichzeitig kann eine solche Hierarchie für die Verteilung lokal relevanter Daten verwendet werden, wie dies mit dem Konzept der *Fragmente* in [Gol02] beschrieben wird. Dabei müssen auch Probleme im Zusammenhang mit überlappenden Replikatdefinition betrachtet werden. Schließlich ist für die Konfliktauflösung mit Hilfe von Alternativen die Problematik der Abhängigkeiten zu klären, d.h. welche Konsequenzen beispielsweise die Einfügung eines alternativen Tupels für andere Daten hat. Hierfür sollte, anders als im Bayou-System [GHOS96], eine deklarative Lösung bereitgestellt werden.

Literaturverzeichnis

[Bau03] K. Baumgarten. Konzept und Realisierung des interaktiven Reiseinformationssystems HERMES. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2003. Laufend.

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

[EN01] R. Elmasri und S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, Menlo Park, CA, 2001.

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

[GB94] P. Graham und K. Barker. Effective Optimistic Concurrency Control in Multiversion Object Bases. In *Proceedings of the Object-Oriented Methodologies and Systems*, Band 858, Seiten 313–328. Springer-Verlag, September 1994.

[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.

[GK85] D. Gawlick und D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering*, 8(2):3–10, Juni 1985.

[Gol02] C. Gollmick. Konzept und Anforderungen der nutzerdefinierten Replikation zur Realisierung neuer mobiler Datenbankanwendungen. Jenaer Schriften zur Mathematik und Informatik Math/Inf/15/02, Institut für Informatik, Friedrich-Schiller-Universität Jena, September 2002.

[Gol03] C. Gollmick. Nutzerdefinierte Replikation zur Realisierung neuer mobiler Datenbankanwendungen. In *Tagungsband der 10. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), Leipzig*, Februar 2003. Angenommener Beitrag.

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

[HR01] T. Härder und E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer, Berlin, 2001.

[HSD+94] M. Hardwick, D. Spooner, B. Downie, M. Ferris, Z. Jiang und T. DeWeese. A STEP Entity Control System for Concurrent Engineering. In *Proceedings of the First International Conference on Concurrent Engineering: Research and Applications*, Pittsburgh, PA, August 1994.

[IBM02] IBM. *IBM DB2 Universal Database Replication Guide and Reference Version 8*, 2002.

[JHE99] J. Jing, A. S. Helal und A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys (CSUR)*, 31(2):117–157, 1999.

[KS91] H. F. Korth und A. Silberschatz. *Database System Concepts*. McGraw-Hill, New York, NY, 1991.

[Lie01] M. Liebisch. Evaluation von Transaktionsmodellen und transaktionsorientierten Reintegrationsstrategien für mobile Datenbanksysteme. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Dezember 2001.

[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.

[Mül03] T. Müller. Architektur und Prototyp eines Replication Proxy Server für die nutzerdefinierte Replikation von Datenbankinhalten. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2003. Laufend.

[O'N86] P. E. O'Neil. The Escrow Transactional Method. *TODS*, 11(4):405–430, 1986.

[Ora99] Oracle. *PL/SQL Users's Guide and Reference (Release 8.1.6)*, Dezember 1999.

[Ora02a] Oracle. *Oracle9i Lite Developer's Guide for Windows 32 (Release 5.0.2)*, September 2002.

[Ora02b] Oracle. *Oracle9i Replication Management API Reference (Release 9.2)*, März 2002.

[PBM+99] N. M. Preguica, C. Baquero, F. Moura, J. L. Martins, R. Oliveira, H. Joao, J. O. Pereira und S. Duarte. MobiSnap: Managing Database Snapshots in a Mobile Environment. In *Actas do 1º Encontro Português de Computação Móvel*, November 1999.

[PBM+00] N. M. Preguica, C. Baquero, F. Moura, J. L. Martins, R. Oliveira, H. Joao, J. O. Pereira und S. Duarte. Mobile Transaction Management in MobiSnap. In *Proceedings of the 2000 ADBIS-DASFAA Symposium on Advances in Databases and Information Systems Special Sessions on Mobile Database Technology (LNCS 1884)*, Seiten 379–386, September 2000.

[PMC02] N. M. Preguica, J. L. Martins und M. Cunha. An SQL-based Mobile Database Combining Conflict Avoidance and Conflict Resolution. Technischer Bericht, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2002.

[Reu82] A. Reuter. Concurrency on High-Traffic Data Elements. In *Proceedings of the ACM Symposium on Principles of Database Systems*, Seiten 83–92, März 1982.

[RG00] R. Ramakrishnan und J. Gehrke. *Database Management Systems.* McGraw-Hill, Boston, 2000.

[SS99] M. Schönhoff und M. Strässler. Global Version Management for a Federated Turbine Design Environment. In *Proceedings of the 4th International Workshop on Next Generation Information Technologies and Systems (NGITS)*, Seiten 203–220, Zikhron-Yaakov, Israel, Juli 1999.

[Syb02a] Sybase, Inc. *Adaptive Server Anywhere SQL User's Guide (Version 8.0.2)*, Oktober 2002.

[Syb02b] Sybase, Inc. *SQL Remote User's Guide (Version 8.0.2)*, Oktober 2002.

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung

Ich erkläre, daß ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, den 21. Februar 2003