Verwaltung replizierter Daten bei nutzerdefinierter Replikation

Christoph Gollmick, Gennadi Rabinovitch

Friedrich-Schiller-Universität Jena, Institut für Informatik
Ernst-Abbe-Platz 2, 07743 Jena
{gollmick,gennadi}@informatik.uni-jena.de

Kurzfassung: Ein Dienst zur Replikation und Synchronisation von Daten ist für mobile Datenbanklösungen unverzichtbar. Das Konzept der nutzerdefinierten Replikation erlaubt dabei die anwendungsgesteuerte, dynamische Auswahl relationaler Datenbankinhalte für die Replikation. Die Realisierung eines solchen Dienstes erfordert eine effiziente Verwaltung von Replikationsanforderungen tausender mobiler Clients. Der Beitrag stellt eine Lösung vor, die auf der Gruppierung von semantisch zusammengehörenden Daten zu Datenfragmenten basiert. Wir gehen dabei auf einen Algorithmus zur Bildung geeigneter Fragmentmengen sowie die Verwaltung von Fragmenten ein.

1 Einführung und Motivation

Dank der Fortschritte im Bereich der Hardware mobiler Geräte, können diese heute mit relationalen Datenbanksystemen ausgestattet werden. Kommerzielle mobile Datenbanklösungen [Fan00] erlauben dabei sowohl eine isolierte Arbeit mobiler Clients als auch ihre Integration in eine kooperative Datenverwaltung mit Workstation- und Großrechner-DBS. Solche mobilen Client/Server-Datenbankumgebungen bestehen aus einem zentralen Datenbank-Server, einem oder mehreren mobilen Datenbank-Clients und einer Komponente, die den Datenabgleich (Synchronisation) regelt. Mobile Geräte sind dabei abhängig von unterwegs verfügbaren Kommunikationsmitteln und einer mobilen Energieversorgung. Beide Voraussetzungen sind heute und auch in naher Zukunft noch mit erheblichen Einschränkungen versehen. Der Stand der Technik bietet zwar adäquate drahtlose Kommunikationsmittel wie WLAN oder UMTS, diese sind aber noch nicht weit verbreitet, teuer und energieintensiv in der Benutzung. Außerdem sind sie prinzipbedingt nicht so leistungsstark und sicher gegen Angriffe von Dritten wie feste Netze. Neben der klassischen Client/Server-Kommunikation ist deshalb für den jederzeitigen, sicheren und kostengünstigen Zugriff lokaler Anwendungen auf Datenbankinhalte ein geeigneter Dienst zur Replikation und Synchronisation von Daten notwendig [ÖV99, HHB96].

Wie ein solcher Replikationsdienst ausgestaltet werden muß, hängt maßgeblich von den Anforderungen des angenommenen Anwendungsszenarios ab. Die hier zugrundegelegte Anwendung ist das interaktive mobile Reiseinformationssystem HERMES [Bau03]. HER-

MES hat die Aufgabe, seine Nutzer bei der Reiseplanung zu unterstützen und ihnen unterwegs ortsabhängige Informationen bereitzustellen. Im Unterschied zu klassischen Touristenführern werden aber nur die Basisdaten, wie Name, Einwohnerzahl oder Historie von Orten, zentral bereitgestellt. Der weitaus größte Teil wird von den mobilen Nutzern selbst und in der Regel vor Ort ergänzt und aktualisiert (z. B. Reiseberichte, Speisekarten von Restaurants). Offensichtlich sind die Zugriffsprofile der HERMES-Nutzer abhängig vom Aufenthaltsort, der Zeit und der Situation, in der sie sich befinden. Eine Konsequenz daraus ist, daß die Auswahl der Daten für die Replikation nicht zentral vorgegeben werden kann. Vielmehr müssen wir der HERMES-Client-Anwendung die Möglichkeit geben, Replikate bezogen auf Nutzer- und Client-Bedürfnisse zur Laufzeit anzufordern und wieder freizugeben.

Leider unterstützen aktuelle kommerzielle mobile Datenbanklösungen eine Auswahl von Replikaten durch die Anwendung nur in sehr eingeschränktem Maße (z. B. einmalige Auswahl aus einer Liste vordefinierter Sichten und Ersetzen von Platzhaltern). Zur Realisierung des unverbundenen Arbeitens von HERMES und verwandten Anwendungen wurde deshalb ein Dienst zur *nutzerdefinierten Replikation* (NR) entwickelt, der auf einer vorhandenen mobilen Client/Server-Datenbankumgebung aufsetzt [Gol03a, Gol03b]. Die Arbeit mit dem nutzerdefinierten Replikationsdienst geschieht in zwei, in verbundenem Zustand auszuführenden, Schritten, der *Replikationsschemadefinition* und der *Replikatdefinition* (Abbildung 1).

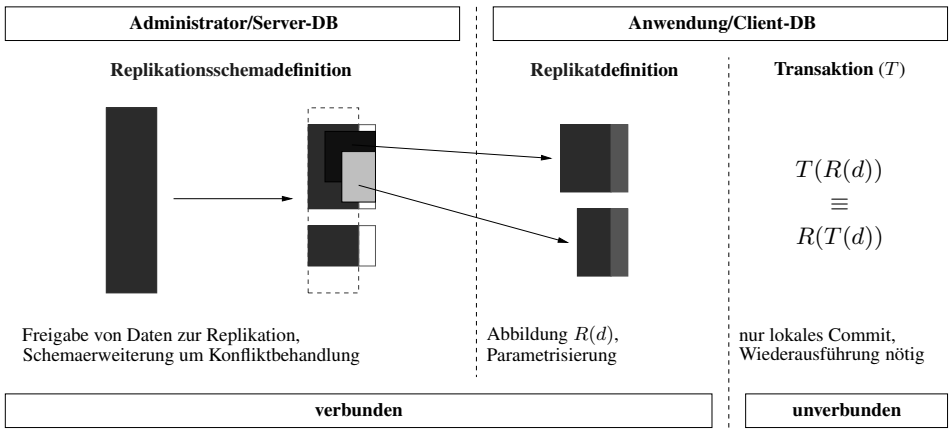


Abbildung 1: Ablauf der nutzerdefinierten Replikation

Das Replikationsschema wird i. d. R. einmalig vom Administrator auf dem Server erzeugt und beschreibt den Ausschnitt der Quelldatenbank, der für alle mobilen Clients zur Replikation freigegeben ist, erweitert um Konfliktbehandlungsmethoden. Darauf bezugnehmend kann ein mobiler Client bevor er die Verbindung trennt ein oder mehrere Replikate definieren. Dies geschieht mit Hilfe einer an die SQL-Sichtdefinition angelehnten Syntax (`CREATE REPLICATION VIEW`), erweitert um einen Modus (optimistisch änderbar, exklusiv änderbar, nur lesbar) und Parameter für die Konfliktbehandlung (z. B. Anzahl der

Schlüssel für Einfügungen). Nach erfolgreicher Synchronisation[1] kann unverbunden mit den Mitteln des Client-DBMS auf den *Replikationssichten* gearbeitet werden. Natürlich können Replikationssichten auch wieder gelöscht werden. Dies signalisiert dann dem Replikationsdienst, daß er die von der Sicht referenzierten Daten, sofern sie keine andere Sicht mehr benötigt, löschen kann.

Die Menge der definierten Replikationssichten bildet ein „Fenster“ durch das ein bestimmter Ausschnitt der Quelldatenbank „sichtbar“ ist. Solange die Transaktionen der mobilen Anwendung nur durch dieses Fenster auf die Daten zugreifen (also die Replikationssichten benutzen), soll für sie *Replikattransparenz* gelten. Replikattransparenz heißt, daß unverbunden durchgeführte Anfragen und Änderungen an Daten dieselbe Ergebnismenge bzw. dasselbe Ergebnis haben, als würden sie, isoliert von anderen Nutzern, auf entsprechenden Sichten der Quelldatenbank ausgeführt ($T(R(d)) \equiv R(T(d))$, Abbildung 1). Durch Replikattransparenz werden falsche Entscheidungen der Anwendung und Reintegrationskonflikte vermieden, die durch unvollständige Daten und eine rein nachgelagerte, also bei der Reintegration der Änderungen durchgeführte, Integritätssicherung entstehen können.[2]

Die einfachste Möglichkeit, Replikattransparenz zu erzielen, ist, die ganze Quelldatenbank inklusive aller Integritätsbedingungen auf den Client zu replizieren. Allerdings wird dies nur in wenigen Ausnahmefällen, in denen die Datenbank klein genug ist, möglich und sinnvoll sein. Eine zentrale Aufgabe unseres Replikationsdienstes ist es also, anhand der Replikationssichtdefinitionen unter Berücksichtigung bereits auf dem Client vorhandener Daten eine geeignete Menge (neu) zu replizierender bzw. zu löschender Daten zu bestimmen. Das erfordert eine entprechende Metadaten-Verwaltung von Replikaten und Replikatdefintionen für die Synchronisation durch den Replikationsdienst.

Im Abschnitt 2 erläutern wir die Grundlagen für die Replikatverwaltung und führen anschließend den Fragment-Begriff in Abschnitt 3 ein. Ein Algorithmus zur initialen Fragmentbildung wird in Abschnitt 4 vorgestellt. Anschließend erläutern wir in Abschnitt 5 die Funktionsweise des Fragmentierungsalgorithmus anhand eines Beispiels und gehen auf eine Realisierung des Fragment-Konzepts ein. Abschnitt 6 schließt mit einer Zusammenfassung und einem Ausblick.

2 Grundlagen der Replikatverwaltung

Ein erster Ansatzpunkt für eine Replikatverwaltung ist das bereits gut erforschte und im Datawarehouse-Bereich eingesetzte Konzept der materialisierten Sichten [GM99]. Wir würden also das Ergebnis jeder Replikationssichtdefinition als materialisierte Sicht auf dem anfragenden mobilen Client anlegen. Dieses einfache Verfahren hat eine Reihe von Nachteilen, das Aufzeigen der wichtigsten von ihnen soll im folgenden dazu dienen, Ziele und Lösungen für eine effiziente Replikatverwaltung zu erarbeiten.

Bei überlappenden Sichtdefinitionen auf einem Client müßte ein u. U. erheblicher Teil

[1] Für die *Durchführung* der Synchronisation und Konfliktbehandlung werden von der NR die Standardkomponenten der Datenbanksysteme der mobilen Client/Server-Datenbankumgebung eingesetzt.

[2] Zu beachten ist, daß Konflikte aufgrund konkurrierender Änderungen verschiedener mobiler Nutzer dadurch nicht verhindert werden können [Lie03].

der Daten redundant vorgehalten werden. Diese Redundanz impliziert nicht nur die Verschwendung kostbaren Speicherplatzes, sondern auch einen erhöhten Synchronisationsaufwand. Ziel muß es also sein, angefragte Daten nur *einmal* auf den Client zu replizieren, egal, von wie vielen Replikationssichten sie benutzt werden.

Eine Übertragung und die lokale Prüfung von Integritätsbedingungen (z. B. Fremdschlüsselbedingung) ist mit materialisierten Sichten nur sehr eingeschränkt möglich (durch die Angabe von `WITH CHECK OPTION`), da nur genau die von der Sichtdefinition erfaßten Daten repliziert werden. Unser Replikationsdienst muß deshalb in der Lage sein, eventuell für die Integritätssicherung zusätzlich erforderliche Daten mit auf den Client zu übertragen.

Ausgehend von einer client-orientierten Replikatverwaltung (der Replikationsdienst sammelt für jeden Client während er nicht mit dem Server verbunden ist, alle ihn betreffenden Datenänderungen, die dann bei der Synchronisation sofort übertragen werden können) müßte der Replikationsdienst Änderungen an Daten, die von mehreren mobilen Clients repliziert wurden, redundant vorhalten. Wir brauchen also eine Verwaltung, die auf der Bildung von logischen Verwaltungseinheiten (im folgenden *Fragmente*) basiert, welche Daten zusammenfassen, die semantisch und nach Auswertung der Zugriffsprofile mobiler Nutzer zusammengehören. Da sich Zugriffsprofile ändern können, muß eine vorgenommene Fragmentierung zudem änderbar sein.

Für die bei der Replikatdefinition durch die Client-Anwendung spezifizierbaren Modi der Replikation (z. B. exklusives Einfügen) benötigen wir auf Dienstseite eine geeignete Sperrverwaltung. Im Fall konkurrierender materialisierter Sichten entspricht die Aufgabenstellung der bei der Realisierung von *Prädikatssperren* [HR01]. Der semantische Vergleich beliebiger Selektionsprädikate ist allerdings NP-schwer und demzufolge sehr „teuer“ in der Ausführung. Wir greifen deshalb auf das bekannte Konzept der Granulatsperren [GR93] zurück. Sperren oder vergleichbare Attribute werden hierbei nicht für beliebige Prädikate gesetzt, sondern nur für eine feste Menge vorgegebener Prädikate. Hierfür bieten sich die Fragmente an, die wir im vorangegangenen Absatz gefordert haben. Dazu müssen wir bei der Bildung von Fragmenten darauf achten, daß sich die Transformation *Replikationssicht* $\rightarrow$ *Fragmentmenge* (siehe Abschnitt 5.3) einfach ausführen läßt.

3 Fragmente als Verwaltungseinheit

Im Unterschied zu Dateisystemen (Dateien) und objektorientierten Datenbanksystemen (Objekte) bieten sich als Fragmentgranulat in relationalen Datenbanken verschiedene logische Verwaltungseinheiten an.[3] Eine Verwaltung auf Tabellenebene ist naheliegend aber zu grob, wenn beispielsweise aus einer großen zentralen Tabelle jeweils nur kleine Datenmengen repliziert werden. Eine feinere Auswahl erlauben *Tabellenpartitionen* [Now01], die über Selektion oder Projektion aus einzelnen Datenbanktabellen gebildet werden. Hierbei können allerdings Abhängigkeiten zwischen verschiedenen Tabellen nicht berücksichtigt werden. Wir definieren die für die Replikation (Auswahl zu replizierender Daten) und

[3] Physische Verwaltungseinheiten wie Seiten sind für eine inhaltsbasierte Verknüpfung von Daten denkbar ungeeignet, weswegen wir sie hier nicht weiter betrachten.

die Sperrverwaltung (siehe auch Abschnitt 5.1) verwendete Einheit deshalb wie folgt:

> Ein *Fragment* ist eine Menge horizontaler, vertikaler oder kombinierter Tabellenpartitionen aus einer oder mehreren Tabellen, angereichert mit Zusatzinformationen (z. B. Beziehungen zu anderen Fragmenten).

Prinzipiell kann ein Fragment aus nur einem Tupel (mit einem Attribut) oder aus der gesamten Datenbank bestehen. Es gilt also bei der Fragmentbildung einen Kompromiß zwischen geringer Fragmentanzahl (Fragmente möglichst groß) und zu viel übertragenen Daten (Fragmente möglichst klein) zu finden. Wir betrachten in diesem Beitrag nur Fragmente, die aus horizontalen Tabellenpartitionen bestehen.

4 Bildung von Fragmenten

Die Problemstellung der logischen und physischen Fragmentierung von Daten zur Verbesserung der Skalierbarkeit der Replikatsynchronisation ist indes nicht neu und es wurden bereits verschiedene Lösungsansätze publiziert [YDN00, PB99]. Die zitierten Ansätze gehen davon aus, daß dem Administrator das Zugriffsprofil der mobilen Clients detailiert bekannt ist, d. h. welche Daten von welchem Client angefordert werden und welche Zugriffs- bzw. Änderungshäufigkeit die Daten aufweisen. Entsprechend wird ein globales Optimum approximiert und in Form von Indexstrukturen oder einer physischen Partitionierung realisiert.

Da das detailierte Zugriffsprofil der mobilen Anwender bei nutzerdefinierter Replikation zunächst nicht zur Verfügung steht, verfolgen wir einen zweiphasigen Ansatz. In der Initialisierungsphase vor Inbetriebnahme des Systems oder bei gravierenden Änderungen im Datenschema, wird aus den vorab gegebenen Informationen zunächst ein geeigneter, nicht zwingend optimaler, Anfangszustand von Fragmenten erstellt, der anschließend in der Anpassungsphase im laufenden Betrieb automatisch und inkrementell anhand von Zugriffsstatistiken angepaßt wird. Wir konzentrieren uns in diesem Beitrag auf die Beschreibung eines Algorithmus zur initialen Fragmentbildung.[4] Die initiale Fragmentbildung erfolgt in zwei Schritten:

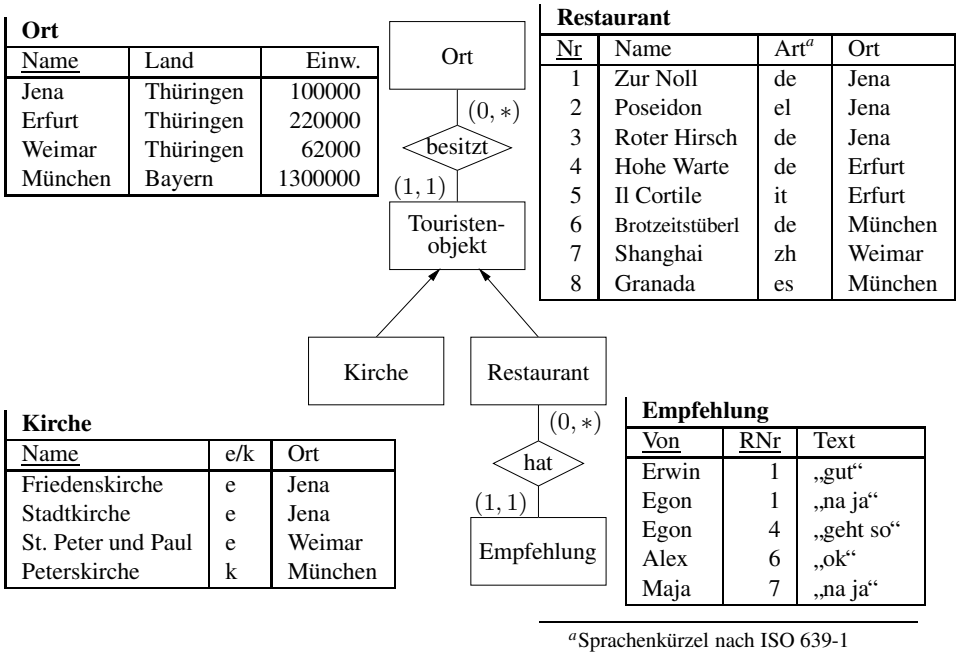
1. Im ersten Schritt, der *Schemafragmentbildung*, werden jeweils Tabellen zu einem Schemafragment zusammengefaßt, die in Beziehung stehen und bei der Replikation immer zusammen übertragen werden sollen.
2. Im zweiten Schritt, der *Datenfragmentbildung*, wird jedes Schemafragment in einzelne (Daten-)Fragmente zerlegt, die jeweils Tupel mit gemeinsamen Merkmalen zusammenfassen.

Zur Entscheidung, welche Tabellen in einem Schemafragment und welche Tupel in einem Fragment verwaltet werden sollten, können Informationen aus dem Datenbankschema (z. B. Fremdschlüssel, Ortsattribute, Integritätsbedingungen), den Daten selbst (z. B.

[4] Prinzipiell können die dabei verwendeten Techniken bzw. deren Umkehrung auch für die spätere inkrementelle Anpassung der Fragmente verwendet werden [Rab03].

nutzer- oder gruppenbezogene Daten) oder von der Anwendung (z. B. nur-lese Daten, Zugriffsrechte) ausgewertet werden. Die Informationen des Datenbankschemas sind weitgehend automatisch auswertbar und stehen bereits vor Inbetriebnahme des Systems fest. Die Informationen über die potentielle Daten- und Anwendungsnutzung müssen vom Anwendungsentwickler bereitgestellt werden, sie stehen aber auch später in der Anpassungsphase durch automatische Auswertung der Zugriffsstatistiken zur Verfügung.

4.1 Schemafragmentbildung



Ort

Name	Land	Einw.
Jena	Thüringen	100000
Erfurt	Thüringen	220000
Weimar	Thüringen	62000
München	Bayern	1300000

Restaurant

Nr	Name	Art[a]	Ort
1	Zur Noll	de	Jena
2	Poseidon	el	Jena
3	Roter Hirsch	de	Jena
4	Hohe Warte	de	Erfurt
5	Il Cortile	it	Erfurt
6	Brotzeitstüberl	de	München
7	Shanghai	zh	Weimar
8	Granada	es	München

Kirche

Name	e/k	Ort
Friedenskirche	e	Jena
Stadtkirche	e	Jena
St. Peter und Paul	e	Weimar
Peterskirche	k	München

Empfehlung

Von	RNr	Text
Erwin	1	„gut“
Egon	1	„na ja“
Egon	4	„geht so“
Alex	6	„ok“
Maja	7	„na ja“

[a] Sprachenkürzel nach ISO 639-1

Abbildung 2: Beispiel: E/R-Diagramm und Daten

Wir beschreiben im folgenden die Schemafragmentbildung anhand der, u. a. für die spätere Änderungsfähigkeit von Replikaten wichtigen, Fremdschlüsselbeziehung zwischen Tabellen. Es können jedoch mit leichten Modifikationen bzgl. der transitiven Partitionierung (siehe Abschnitt 4.2) auch andere Abhängigkeiten und Integritätsbedingungen verwendet werden. Zur Illustration des Verfahrens verwenden wir den in Abbildung 2 angegebenen Ausschnitt aus einer HERMES-Datenbank. *Restaurant.Ort* und *Kirche.Ort* sind jeweils Fremdschlüssel bzgl. Tabelle *Ort*, *Empfehlung.RNr* ist Fremdschlüssel bzgl. *Restaurant*.

Wir betrachten die im Schema vorhandenen Tabellen als Knoten ($X_{1 \leq i \leq n}$) und die Fremdschlüsselbeziehungen als zur Primärschlüssel haltenden Tabelle hin gerichtete Kanten eines Graphen (Abbildung 3 zeigt den Graphen für das Beispielschema). Ziel der Sche-

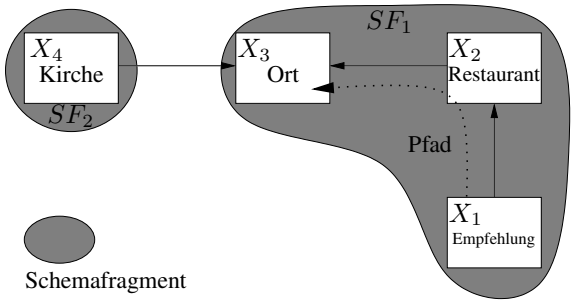


Abbildung 3: Beispiel für Schemafragmentbildung

mafragmentierung ist es, rekursiv die Primärschlüssel haltende Tabelle zu den abhängigen Tabellen in ein Schemafragment zu gruppieren. Bedingung für die spätere Durchführung der Datenfragmentierung (Abschnitt 4.2) ist, daß jedes Schemafragment höchstens einen Fremdschlüsselpfad enthalten darf. Die Schemafragmentierung ist nicht eindeutig und i. d. R. Aufgabe eines Administrators. Da wir die zu erwartenden Zugriffsprofile noch nicht im Detail kennen (Wird sich z. B. die Mehrzahl der Reisenden in besuchten Orten eher für Kirchen oder Restaurants interessieren?), können wir die initiale Schemafragmentierung für eine gegebene Knotenliste automatisch mit dem in Abbildung 4 angegebenen Algorithmus erzeugen lassen. Über die Reihenfolge der Knoten in der Eingabeliste, kann der Administrator die Schemafragmentierung steuern.

SchemaFrag($<X_1, \ldots, X_n>$)

```
1:  for each X_{1≤i≤n} do
2:      if not Visited(X_i) do
3:          Visited(X_i) = true;
4:          select X_p = parent(X_i)
            with not Visited(X_p)
5:          if exists X_p do
6:              SchemaFrag(<X_p>);
7:              SF(X_i) ∪ SF(X_p);
8:          od;
9:      od;
```

```
SELECT DISTINCT tabname
FROM syscat.tables
WHERE tabschema='HERMES'
      AND tabname NOT IN (
          SELECT reftabname
          FROM syscat.references
          WHERE tabschema='HERMES');

WITH temp (mode, path, len) AS (
      SELECT p.tabname, p.tabname, 1
      FROM syscat.references p
      WHERE p.tabname = 'EMPFEHLUNG'
UNION ALL
      SELECT c.reftabname,
          p.path || '+' || c.reftabname, p.len+1
      FROM temp p, syscat.references c
      WHERE p.mode = c.tabname)
SELECT path, len FROM temp
ORDER BY len DESC;
```

Abbildung 4: Algorithmus SchemaFrag und SQL-Anweisungen (DB2-Notation)

Zu Beginn des SchemaFrag-Algorithmus bildet jeder Knoten in der Knotenliste ein eigenes Schemafragment. Während der Abarbeitung werden die auf einem Pfad liegenden Schemafragmente vereinigt, wobei jeder Knoten nur einmal betrachtet wird. Nach Abschluß des Algorithmus verfügen wir über eine Anzahl disjunkter Schemafragmente, deren jeweilige Tabellen über einen Fremdschlüsselpfad verbunden sind (Abbildung 3 zeigt eine mögliche Zerlegung unseres Beispiels in zwei Schemafragmente). Die bei der Zerlegung nicht berücksichtigten Fremdschlüsselbeziehungen werden als Zusatzinformationen in die späteren Fragmente aufgenommen (siehe Abschnitt 5.2). Die Schemafragmentbildung bzgl. Fremdschlüssel läßt sich auch leicht über SQL-Anfragen an den Datenbankkatalog unterstützen. Mögliche Pfadanfänge (Tabellen im Schema, auf die keine Fremdschlüsselreferenz verweist) erhält man mit Hilfe der ersten in Abbildung 4 angegebenen Kataloganfrage. Die zweite, rekursive, Anfrage bestimmt für die Tabelle *Empfehlung* alle möglichen Fremdschlüsselpfade.

4.2 Datenfragmentbildung

Der nächste Schritt ist die Zerlegung der Schemafragmente anhand der vorhandenen Basisdaten in geeignete Fragmente. Für die initiale Fragmentbildung müssen allgemeingültige Kriterien aus der Anwendung abgeleitet werden. Betrachten wir HERMES, so ist es für einen Touristen in Jena wahrscheinlicher, daß er sich für Jenaer Gastronomiebetriebe und Kulturobjekte interessiert als für die in München. Es gilt also für Zugriffsprofile von HERMES-Nutzern eine *Ortsabhängigkeit* im Datenzugriff. Eine weitere Klassifizierung von Daten ist anhand der Anzahl der auf sie (potentiell) zugreifenden Nutzer in *nutzerspezifisch* (z. B. wenn HERMES die Verwaltung privater Reiseaufzeichnungen erlaubt) und *gemeinsam genutzt* möglich.

DataFrag(F, *fmin*, $<A_1, \ldots, A_n>$)

1: let A_i be the first attribute with ($1 \leq i \leq n$) and more than one distinct value (V) in corresponding table
2: **if** exists A_i **do**
3: **for** each $V^i_{1 \leq j \leq m}$ **do**
4: **if** (SELECT count(*) FROM $X(A_i)$ WHERE $A^i = V^i_j) \geq$ *fmin* **do**
5: create new fragment with condition: „*Condition*(F) AND $A^i = V^i_j$ “
6: **else**
7: insert V^i_j in *SetOfResidValues*
8: **if** new fragments were created **and** *SetOfResidValues* is not empty **do**
9: extend F's condition to „*Condition*(F) AND A^i IN (*SetOfResidValues*)“
10: **for** each new or changed fragment $F'_{1 \leq k \leq o}$ **do**
11: DataFrag(F'_k, *fmin*, $<A_{i+1}, \ldots, A_n>$);
12: **od**;

Abbildung 5: Algorithmus DataFrag

Die genannten Kriterien haben die angenehme Eigenschaft, daß sie in der Regel explizit mit Attributen im Schema verbunden sind (Ortsattribute, Besitzer). Wir können damit

einen Algorithmus (Abbildung 5) angeben, der bei Eingabe einer Attributliste und eines Grenzwertes (*fmin*) die Daten eines Schemafragments horizontal in Fragmente zerlegt. Die Attributliste gibt die Reihenfolge der Partitionierungsattribute auf dem *Pfad* vor, der *vom Blatt zur Wurzel* durchlaufen wird. Für jeden angenommenen Wert eines Partitionierungsattributs wird vom Algorithmus ein neues Fragment abgespalten. Über *fmin* wird sichergestellt, daß solche Fragmente mindestens eine Tabelle mit $\geq$ *fmin* Tupeln enthalten. Die Liste der Partitionierungsattribute für jede Tabelle kann entweder vom Administrator vorgegeben oder, wenn keine Informationen vorliegen, durch aufsteigende Sortierung aller Attribute der jeweils betrachteten Tabelle auf dem Pfad nach der Anzahl der verschiedenen Attributwerte automatisch ermittelt werden [Rab03].

5 Beispiel und Realisierung

5.1 Beispiel

Am Beispiel des Schemafragments 1 (Abbildung 3) aus unserem HERMES-Schema soll die Funktionsweise von DataFrag demonstriert werden (Attributliste[5]: *Ort.Land*, *Restaurant.Ort*, *Restaurant.Art* und *fmin* $= 2$). Das erste vom Algorithmus ausgewählte Partitionierungsattribut ist *Ort.Land*. Es wird ein neues Fragment (Fragment 1) für den Wert 'Thüringen' (3 Tupel in *Ort 1*) erzeugt und es verbleibt ein Fragment (Fragment 2) mit *Ort.Land*='Bayern' (Abbildung 6).

Ort **1**

Name	**Land**	Einwohner
Erfurt	**Thüringen**	220000
Jena	**Thüringen**	100000
Weimar	**Thüringen**	62000

Restaurant **1**

Nr	Name	Art	Ort
1	Zur Noll	de	Jena
2	Poseidon	el	Jena
3	Roter Hirsch	de	Jena
4	Hohe Warte	de	Erfurt
5	Il Cortile	it	Erfurt
7	Shanghai	zh	Weimar

Empfehlung **1**

Von	RNr	Text
Erwin	1	„gut“
Egon	1	„na ja“
Egon	4	„geht so“
Maja	7	„na ja“

Ort **2**

Name	**Land**	Einwohner
München	**Bayern**	1300000

Restaurant **2**

Nr	Name	Art	Ort
6	Brotzeitstüberl	de	München
8	Granada	es	München

Empfehlung **2**

Von	RNr	Text
Alex	6	„ok“

Abbildung 6: Beispiel für Fragmentierung – Schritt 1

Im nächsten Schritt betrachtet der Algorithmus für die Fragmente 1 und 2 das Partitionierungsattribut *Restaurant.Ort* (Abbildung 7). Der Algorithmus spaltet nun zuerst 'Jena' als neues Fragment (Fragment 1.1) ab. Als nächstes spaltet der Algorithmus die Tupel für Erfurt ab (Fragment 1.2) und beläßt die Tupel für 'Weimar' in Fragment 1.3. Das Fragment 2 wird nicht weiter unterteilt, weil das Partitionierungsattribut *Restaurant.Ort* nur einen Attributwert hat ('München').

Anschließend betrachtet der Algorithmus das Partitionierungsattribut *Restaurant.Art* (Abbildung 8). Es wird ein neues Fragment (Fragment 1.1.1) mit *Restaurant.Art* $=$ 'de' vom

[5] entspricht einer Liste, wie sie sich bei automatischer Sortierung der Ortsattribute ergeben würde

Ort **1.1**

Name	Land	Einwohner
Jena	Thüringen	100000

Restaurant **1.1**

Nr	Name	Art	**Ort**
1	Zur Noll	de	**Jena**
2	Poseidon	el	**Jena**
3	Roter Hirsch	de	**Jena**

Empfehlung **1.1**

Von	RNr	Text
Erwin	1	„gut“
Egon	1	„na ja“

Ort **1.2**

Name	Land	Einwohner
Erfurt	Thüringen	220000

Restaurant **1.2**

Nr	Name	Art	**Ort**
4	Hohe Warte	de	**Erfurt**
5	Il Cortile	it	**Erfurt**

Empfehlung **1.2**

Von	RNr	Text
Egon	4	„geht so“

Ort **1.3**

Name	Land	Einwohner
Weimar	Thüringen	62000

Restaurant **1.3**

Nr	Name	Art	**Ort**
7	Shanghai	zh	**Weimar**

Empfehlung **1.3**

Von	RNr	Text
Maja	7	„na ja“

Abbildung 7: Beispiel für Fragmentierung – Schritt 2

Fragment 1.2 abgespalten. Die Tupel für 'el' verbleiben im Restfragment (Fragment 1.1.2). Die Fragmente für 'Weimar' und für 'München' werden nicht weiter unterteilt. Es ist zu bemerken, daß bei dieser Art der Fragmentbildung sowohl (zunächst) leere Tabellenpartitionen entstehen können (siehe Empfehlung 1.1.2), als auch ein Tupel in mehreren Fragmenten auftreten kann (siehe Ort 1.1.1 und Ort 1.1.2). Da Datenfragmente somit nicht disjunkt sein müssen, sind die Überscheidungen zwischen Fragmenten in einem entsprechenden (hierarchischen) Sperrverfahren zu berücksichtigen [GR93, Rab03].

Ort **1.1.1**

Name	Land	Einwohner
Jena	Thüringen	100000

Restaurant **1.1.1**

Nr	Name	**Art**	Ort
1	Zur Noll	**de**	Jena
3	Roter Hirsch	**de**	Jena

Empfehlung **1.1.1**

Von	RNr	Text
Erwin	1	„gut“
Egon	1	„na ja“

Ort **1.1.2**

Name	Land	Einwohner
Jena	Thüringen	100000

Restaurant **1.1.2**

Nr	Name	**Art**	Ort
2	Poseidon	**el**	Jena

Empfehlung **1.1.2**

Von	RNr	Text

Abbildung 8: Beispiel für Fragmentierung – Schritt 3

Abbildung 9 zeigt am Beispiel des Fragment 1.1.1 die SQL-Anfragen zur Beschreibung der Tabellenpartitionen, die mit Hilfe der von DataFrag generierten Auswahlbedingungen erzeugt wurden.

5.2 Metadaten

Die Tabelle *Fragments* in Abbildung 10 enthält die Metadaten zur Verwaltung der Fragmente. In der Tabelle werden für jedes Fragment und die darin enthaltenen Tabellen und Attribute, wenn vorhanden, die Auswahlprädikate gespeichert. Wir machen uns dabei die spezielle Struktur der vom DataFrag-Algorithmus erzeugten Prädikate zunutze, die in konjunktiver Normalform (KNF) vorliegen und nur „=“-Vergleiche mit Konstanten enthalten. Die Spalte *Condition* enthält jeweils die möglichen Attributwerte (Oder-Verknüpfung) in einem String-Format, das die einfache Abfrage mit SQL-`LIKE` erlaubt. Zur Auswahl der Daten eines bestimmten Fragments aus der Datenbank, müssen alle seine Bedingungen

```
Ort:          SELECT DISTINCT o.Name, o.Land, o.Einwohner
              FROM (Empfehlung e RIGHT OUTER JOIN Restaurant r ON e.RNr=r.Nr)
                 RIGHT OUTER JOIN Ort o ON r.Ort=o.Name
              WHERE o.Land='Thüringen' AND r.Ort='Jena' AND r.Art='de';

Restaurant:   SELECT DISTINCT r.Nr, r.Name, r.Art, r.Ort
              FROM (Empfehlung e RIGHT OUTER JOIN Restaurant r ON e.RNr=r.Nr)
                 RIGHT OUTER JOIN Ort o ON r.Ort=o.Name
              WHERE o.Land='Thüringen' AND r.Ort='Jena' AND r.Art='de';

Empfehlung:   SELECT DISTINCT e.Von, e.RNr, e.Text
              FROM (Empfehlung e RIGHT OUTER JOIN Restaurant r ON e.RNr=r.Nr)
                 RIGHT OUTER JOIN Ort o ON r.Ort=o.Name
              WHERE o.Land='Thüringen' AND r.Ort='Jena' AND r.Art='de';
```

Abbildung 9: SQL-Anfragen für Fragment 1.1.1 (DB2-Notation)

erfüllt sein (Und-Verknüpfung).

Wenn das vom DataFrag-Algorithmus zur Partitionierung verwendete Attribut ein Primär- oder Fremdschlüsselattribut war, so wird dessen Bedingung zum *Condition*-String des jeweiligen Beziehungspartners hinzugefügt (z. B. *Ort.Name* = 'Jena'). Damit wir später auch Replikationssichtdefinitionen mit Verbunden effizient ausführen können, speichern wir in der *Condition*-Spalte neben den Attributwerten mögliche Join-Attribute (auch schemafragmentübergreifend) anderer Tabellen. So wird z. B. die Beziehung zwischen den Tabellen *Restaurant* und *Ort* einmal beim Fremdschlüsselattribut *Restaurant.Ort* (*Ort.Name*) und einmal beim Primärschlüssel haltenden Attribut *Ort.Name* (*Restaurant.Ort*) in der *Fragments*-Tabelle gespeichert. Im normalen Betrieb wird eine Anpassung der Tabelle nur dann notwendig, wenn sich die Fragmentierung (z. B. durch Einfügen eines Restaurants mit einer neuen Stilrichtung) oder das Schema ändern.

5.3 Fragmentauswahl

Eine der wichtigsten Aufgaben eines Dienstes für die nutzerdefinierte Replikation ist die Abbildung einer Replikationssichtdefinition auf eine möglichst minimale Menge von Fragmenten. Für Selektionsprädikate in KNF die nur „=“-Vergleiche von Attributen mit Konstanten oder anderen Attributen (für Verbunde) enthalten, kann diese Aufgabe von dem in Abbildung 11 dargestellten Algorithmus SimplePred2Frag mit Hilfe der *Fragments*-Tabelle geleistet werden. Wie wir bei der Konzeption von HERMES feststellen konnten, lassen sich fast alle denkbaren Replikatanforderungen, die ja i. d. R. *nicht* die „normalen“ Anfragen an die Datenbank ersetzen, in der geforderten Form darstellen [Bau03].

Für die Ausführung von SimplePred2Frag nehmen wir an, daß der für die erste Auswertung der Replikationssichtdefinition verwendete Parser das Selektionsprädikat wie vom Algorithmus gefordert aufbereitet. Beispielsweise werden alle Join-Prädikate der Form $Attr_1 = Attr_2$ in die semantisch äquivalente Form $Attr_1 = Attr_2$ `OR` $Attr_2 = Attr_1$

FId	Table	Attribute	Condition
1	Ort	Name	+Jena+Restaurant.Ort+Kirche.Ort+
1	Ort	Land	+Thüringen+
1	Ort	Einwohner	–
1	Restaurant	Nr	+Empfehlung.Rnr+
1	Restaurant	Name	–
1	Restaurant	Art	+de+
1	Restaurant	Ort	+Jena+Ort.Name+
1	Empfehlung	Von	–
1	Empfehlung	Rnr	+Restaurant.Nr+
1	Empfehlung	Text	–
2	Ort	Name	+Jena+Restaurant.Ort+Kirche.Ort+
2	Ort	Land	+Thüringen+
2	Ort	Einwohner	–
2	Restaurant	Nr	+Empfehlung.Rnr+
2	Restaurant	Name	–
2	Restaurant	Art	+el+
2	Restaurant	Ort	+Jena+Ort.Name+
2	Empfehlung	Von	–
2	Empfehlung	Rnr	+Restaurant.Nr+
2	Empfehlung	Text	–
3	Ort	Name	+Erfurt+Restaurant.Ort+Kirche.Ort+
3	Ort	Land	+Thüringen+
3	Ort	Einwohner	–
3	Restaurant	Nr	+Empfehlung.Rnr+
3	Restaurant	Name	–
3	Restaurant	Art	+de+it+
3	Restaurant	Ort	+Erfurt+Ort.Name+
3	Empfehlung	Von	–
3	Empfehlung	Rnr	+Restaurant.Nr+
3	Empfehlung	Text	–

Abbildung 10: Ausschnitt aus *Fragments* (Fragmente 1.1.1, 1.1.2, 1.2)

überführt. Der Algorithmus erzeugt für jede Disjunktion der KNF eine Anfrage an die Tabelle *Fragments*. Jeder Vergleich *Tab.Attr* = *Value* in der Disjunktion wird dem Prädikat der `SELECT`-Anweisung in der Form *Table* = 'Tab' `AND` *Attribute* = 'Attr' `AND (` *Condition* `LIKE` '%+Value+%' `OR` *Condition* `IS NULL )` mit Oder-Verknüpfung hinzugefügt. Die Konjunktionen zwischen den Disjunktionen in der KNF realisieren wir durch Bestimmung der Schnittmenge aus allen Disjunktions-Anfragen (`INTERSECT ALL`). Die Abarbeitung der KNF erfolgt dabei getrennt für jedes Schemafragment. Die für die Schemafragmente erzeugten Anfragen werden über `UNION ALL` verknüpft. Die Ausführung der generierten Anfrage liefert die Nummern der zum Client aufgrund der Replikationssichtdefinition zu übertragenden Fragmente.

Abbildung 12 zeigt ein Beispiel für eine Replikationssichtdefinition und Abbildung 13 deren Übersetzung in eine Anfrage an die Tabelle *Fragments* durch den SimplePred2Frag-

```
SimplePred2Frag(preprocessed predicate) returns set of FId's
1:    for each schemafragment S do
2:       for each disjunction in CNF containing attributes which are also contained in S do
3:          for each triple (table, attribute, condition) with (table, attribute) exists in S do
4:             if wherecond is not empty then wherecond = wherecond + „ OR “;
5:             wherecond = wherecond + „(f.table = “ + table + „ AND f.attribute = “ +
                  attribute + „ AND (f.condition LIKE '%+“ + condition + „+%' OR
                  f.condition IS NULL))“;
6:          od;
7:          if Query_SF is not empty then Query_SF = Query_SF + „ INTERSECT ALL “;
8:          Query_SF = Query_SF + „(SELECT FId FROM Fragments f WHERE “ +
                  wherecond + „)“;
9:       od;
10:      Query_SF = „(“ + Query_SF + „)“;
11:      if Query is not empty then Query = Query + „ UNION ALL “;
12:      Query = Query + Query_SF;
13:   od;
14:   Executing of Query returns set of FId's
```

Abbildung 11: Algorithmus SimplePred2Frag

```
CREATE REPLICATION VIEW dtEssenInJenaUndErfurt
SOURCE Hermes FOR
    SELECT r.Name, e.Text
    FROM Restaurant r, Empfehlung e
    WHERE e.RNr=r.Nr AND (r.Ort='Jena' OR r.Ort='Erfurt') AND r.Art='de';
```

Abbildung 12: Anfragebeispiel

Algorithmus. Die Ausführung der Anfrage liefert die Fragmente 1.1.1 ($FId = 1$) und 1.2 ($FId = 3$) zurück. Die im Beispiel vorgenommene Fragmentierung ist offensichtlich dann günstig, wenn Nutzer ihre Restaurants in der Regel nach Empfehlungen auswählen und sich in diesem Zusammenhang nur wenig für Kirchen interessieren. Es werden nur solche Anfragen effizient unterstützt, die ihre Datenauswahl mit Hilfe der für die Fragmentbildung verwendeten Partitionierungsattribute durchführen. Werden beispielsweise Empfehlungen nach ihrem Urheber selektiert, so erhält die mobile Anwendung das komplette Schemafragment 1! Tritt eine solcher Zugriff häufiger auf, so ist eine Anpassung der Fragmentierung notwendig (Partitionierung der Fragmente des Schemafragments 1 nach *Empfehlung.Von*).

Nicht unerwähnt bleiben sollen die Schwächen der hier vorgestellten Realisierung des Fragmentkonzepts. So ist zwar eine Anpassung der Fragmentierung möglich, jedoch erhöht die Zahl der verwendeten Partitionierungsattribute den Overhead bei der Fragmentverwaltung und beim Arbeiten mit der Datenbank. `UNIQUE`-Attribute wie Primärschlüssel oder Attribute deren Werteanzahl starken Schwankungen unterliegt, sind für eine Partitionie-

```
SELECT DISTINCT * FROM
    ( SELECT FId
      FROM Fragments f
      WHERE (f.Table='Empfehlung' AND f.Attribute='RNr' AND
            (f.Condition LIKE '%+Restaurant.Nr+%' OR f.Condition IS NULL)) OR
            (f.Table='Restaurant' AND f.Attribute='Nr' AND
            (f.Condition LIKE '%+Empfehlung.RNr+%' OR f.Condition IS NULL)) )
    INTERSECT ALL
    ( SELECT FId
      FROM Fragments f
      WHERE (f.Table = 'Restaurant' AND f.Attribute = 'Ort' AND
            (f.Condition LIKE '%+Jena+%' OR f.Condition IS NULL)) OR
            (f.Table = 'Restaurant' AND f.Attribute = 'Ort' AND
            (f.Condition LIKE '%+Erfurt+%' OR f.Condition IS NULL)) )
    INTERSECT ALL
    ( SELECT FId
      FROM Fragments f
      WHERE f.Table='Restaurant' AND f.Attribute='Art' AND
            (f.Condition LIKE '%+de+%' OR f.Condition IS NULL) ) );
```

Abbildung 13: Beispiel einer Fragmentanfrage

rung nicht geeignet. Auch unterscheidet der Algorithmus zur Zeit nicht zwischen Anfragen mit reiner Leseabsicht (hierbei kann auf eine Integritätssicherung auf Client-Seite verzichtet werden) und solchen mit Schreibberechtigung. Lösungsmöglichkeiten für diese Probleme sehen wir in der Einführung von Fragmenthierarchien, die analog zu hierarchischen Sperren [GR93] verwendet werden, und der Zulassung vertikaler Partitionen.

6 Zusammenfassung und Ausblick

Für die Replikatverwaltung der nutzerdefinierten Replikation haben wir den Einsatz von Fragmenten als Verwaltungsgranulat motiviert, einen Fragmentbildungsalgorithmus vorgestellt und eine mögliche Realisierung angegeben. Bei Verwendung von Fragmenten als Replikationsgranulat muß der Replikationsdienst für jeden mobilen Client nur die von ihm referenzierten Fragmente speichern. Die Auswertung einer Replikationssichtdefinition reduziert sich auf die Abbildung des angegebenen Prädikats auf die Menge der Fragmente. Für Prädikate in KNF, die nur „=“-Vergleiche enhalten, wurde ein Abbildungsalgorithmus angegeben. Da bei einer günstigen Fragmentierung die Fragmentmenge stabil und unabhängig von der Anzahl der mobilen Clients ist, können wir eine gute Skalierbarkeit des Fragmentkonzepts erwarten. Am Beispiel des Reiseinformationssystems HERMES konnte gezeigt werden, daß bereits mit einem automatischen Verfahren günstige Fragmentierungen erzeugt werden können. Die weiteren Ziele sind die Implementierung des Fragmentkonzepts inklusive dynamischer Fragmentanpassung in einem Prototyp für den nutzerdefinierten Replikationsdienst [Mül03], Komplexitätsbetrachtungen für Metadaten und Abbildungsalgorithmen sowie deren Erweiterung um Abbildungsfunktionalität für komplexere Prädikate.

Literaturverzeichnis

[Bau03] K. Baumgarten. Konzeption und Schemaentwurf eines interaktiven mobilen Reiseinformationssystems. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Juli 2003.

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

[GM99] A. Gupta und I. Mumick. *Materialized Views: Techniques, Implementations, and Applications*. The MIT Press, Cambridge, Massachusetts, 1999.

[Gol03a] C. Gollmick. Client-Oriented Replication in Mobile Database Environments. Jenaer Schriften zur Mathematik und Informatik Math/Inf/08/03, Institut für Informatik, Friedrich-Schiller-Universität Jena, Mai 2003.

[Gol03b] C. Gollmick. Nutzerdefinierte Replikation zur Realisierung neuer mobiler Datenbankanwendungen. In *Tagungsband der 10. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW), Leipzig*, Band P-26 aus *Lecture Notes in Informatics (LNI)*, Seiten 453–462, Februar 2003.

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

[HHB96] A. Helal, A. Heddaya und B. Bhargava. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, August 1996.

[HR01] T. Härder und E. Rahm. *Datenbanksysteme – Konzepte und Techniken der Implementierung*. Springer-Verlag, 2. Auflage, 2001.

[Lie03] M. Liebisch. Synchronisationskonflikte beim mobilen Datenbankzugriff: Vermeidung, Erkennung, Behandlung. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, Februar 2003.

[Mül03] T. Müller. Architektur und Realisierung eines Replication Proxy Server zur Unterstützung neuartiger mobiler Datenbankanwendungen. Diplomarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, April 2003.

[Now01] J. Nowitzky. Partitionierungstechniken in Datenbanksystemen: Motivation und Überblick. *Informatik Spektrum*, 24(6):345–356, Dezember 2001.

[ÖV99] T. Özsu und P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Inc., 2. Auflage, 1999.

[PB99] S. Phatak und B. Badrinath. Data Partitioning for Disconnected Client Server Databases. In *Proc. of the International Workshop on Data Engineering for Wireless and Mobile Access, Seattle, USA*, Seiten 102–109. ACM, 1999.

[Rab03] G. Rabinovitch. Replikatverwaltung bei der nutzerdefinierten Replikation. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2003. Laufend.

[YDN00] W. Yee, M. Donahoo und S. Navathe. A Framework for Designing Update Objects to Improve Server Scalability in Intermittently Synchronized Databases. In *Proc. of 9th Intl. Conference On Information And Knowledge Management, USA, 2000*, Seiten 54–61, 2000.