Mobile and Context-aware Database Technologies and Applications

Replikation und Caching für mobile Anwendungen

Christian Doppstadt
Matrikelnummer 353112

15. Juni 2007

Technische Universität Kaiserslautern
Fachbereich Informatik
Lehrgebiet Informationssysteme

Betreuer:
Joachim Klein

Inhaltsverzeichnis

1 Die Ausgangssituation

Die Nachfrage und der Bedarf nach mobilen Anwendungen steigen stetig an. Zu dieser Entwicklung tragen die immer weiter sinkenden Preise für mobile Geräte und deren damit wachsende Anzahl und Verbreitung bei. Der Anteil verkaufter Notebooks lag im Vergleich zur Anzahl verkaufter Personal Computern für den privaten Gebrauch bereits im Jahr 2006 bei 52 Prozent [1], was diesen Trend bestätigt. Neben Notebooks spielen aber auch PDAs[1] und Mobiltelefone als Plattform für mobile Anwendungen eine wichtige Rolle, da es sich hierbei um Geräte handelt, die man immer bei sich tragen und somit erst wirklich von Mobilität sprechen kann. Am 14. August 2006 gab der BITKOM[2] bekannt, dass die Anzahl der Mobilfunk-Anschlüsse mit 82,8 Millionen die Zahl der Einwohner in Deutschland überschritten hat [2]. Damit steht nahezu jedem Menschen in Deutschland das Mobiltelefon als Plattform für mobile Anwendungen zur Verfügung. Zudem ergibt sich hierdurch die Möglichkeit, auch außerhalb von lokalen Netzen über das Mobilfunknetz Zugang zu Firmennetzen oder zum Internet zu erlangen.

Ein weiterer Trend ist die steigende Nachfrage nach UMTS-Anschlüssen und die damit verbundene Erweiterung der Bandbreite für Datenverbindungen. So stieg die Anzahl der Anschlüsse zwischen Ende 2005 und Ende 2006 von 2,3 Millionen auf 6,5 Millionen an. Der BITKOM rechnet für das Jahr 2007 mit einem weiteren Wachstum von 60 Prozent im Vergleich zum Vorjahr [3]. Durch die ständig wachsende Bandbreite steigt auch das Potenzial für mobile Anwendungen, denn erst durch die gesteigerte Konnektivität werden Anwendungen die einen hohen Bedarf an Übertragungsressourcen haben – wie Videokonferenzen, Media-Streaming oder Datentransfer größerer Dateien – auch auf mobilen Endgeräten möglich.

Problematisch ist jedoch noch immer die Verfügbarkeit von Netzzugängen. So ist eine flächendeckende Bereitstellung von UMTS nur in städtischen Großräumen gewährleistet. Aber auch schmalbandige Onlineverbindungen über das GSM-Netz sind nicht überall möglich bzw. ausreichend. So sind Funklöcher selbst in Deutschland, mit einer sehr guten Netzabdeckung, nicht ungewöhnlich. Betrachtet man zudem andere Regionen der Welt, so stellt eine flächendeckende Netzabdeckung eher die Ausnahme dar. Netzzugänge über Wireless LAN (WLAN) ermöglichen zwar eine deutlich erhöhte Übertragungsrate selbst gegenüber UMTS, jedoch ist diese Technik durch die oft nur lokale Verfügbarkeit und die eingeschränkte Reichweite nicht für den dauerhaften, mobilen Netzzugang geeignet.

Unabhängig von der Umgebungssituation möchte der Benutzer seine mobile Anwendung nutzen, ohne dabei auf die Verfügbarkeit eines Netzzuganges und dessen bereitgestellte Bandbreite angewiesen zu sein. Dies ist zwar sicherlich nicht in jeder Situation möglich, da der genaue Bedarf an Informationen nicht vorhersehbar ist und zeitnahe Informationen nicht im Voraus bereitgestellt

[1] Personal Digital Assistant
[2] Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.

werden können. Durch gezieltes Zwischenspeichern von Informationen kann der Bedarf an Bandbreite jedoch reduziert und Ladezeiten verkürzt werden. Diese Arbeit soll aufzeigen, welche Verfahren hierzu zum Einsatz kommen und welche besonderen Anforderungen im mobilen Einsatz an diese gestellt werden.

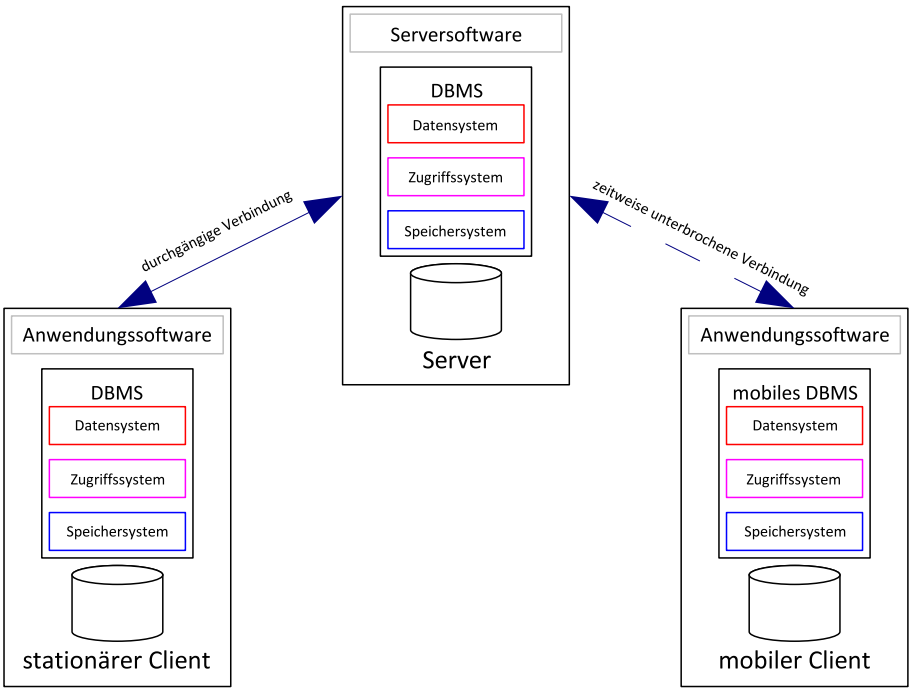

Abbildung 1. Allgemeiner Überblick über das Anwendungsszenario.

Abbildung 1 gibt einen Überblick über das Anwendungsszenario, an dem gezeigt werden soll, wie Caching und Replikation bei mobilen Anwendungen in eine Systemarchitektur integriert sind. Das Szenario enthält einen zentralen Server, sowie zwei Clients, von denen einer nur über eine zeitweise unterbrochene Verbindung angebunden ist. Auf allen Systemen läuft ein Datenbankmanagementsystem (DBMS), sowie weiter Software, die auf das DBMS zugreifen kann.

Nachfolgend werden zunächst Anwendungssoftware, Caching und Replikation im Allgemeinen betrachtet. Die Anwendung von Caching und Replikation wird dann auf das Datenbankumfeld erweitert. An verschiedenen Verfahren sollen danach Anpassungen bei Caching und Replikation für den Einsatz im mobilen Umfeld aufgezeigt werden. Abschließend wird noch auf zukünftige Entwicklungen, sowie weiteres Potential mobiler Anwendungen in Bezug auf Caching und Replikation eingegangen.

2 Umgebungsunabhängige Betrachtung von mobilen Anwendungen, Replikation und Caching

In diesem Abschnitt soll auf die drei namensgebenden Teilbereiche – mobile Anwendungen, Replikation und Caching – dieser Arbeit im Einzelnen und unabhängig voneinander eingegangen werden. Dabei geht es im Wesentlichen um die Spezifika von mobilen Anwendungen, sowie eine grundlegende Erläuterung von Replikation und Caching in stationären Anwendungen. In den einzelnen Unterabschnitten werden die jeweiligen Themen zudem durch kurze Beispiele verdeutlicht.

2.1 Mobile Anwendungen

Eine mobile Anwendung zeichnet sich, wie der Name bereits vermuten lässt, durch die Möglichkeit zum mobilen Einsatz aus. Dabei ist die Mobilität der Hardware, auf dem die Applikation ausgeführt werden soll, eine grundlegende Voraussetzung für den mobilen Einsatz. Damit ergibt sich aber auch zugleich das größte Problem mobiler Anwendungen, denn beim Einsatz mobiler Geräte erreicht man die gewonnene Bewegungsfreiheit nur durch Einbußen bei Rechenleistung, Akku-Laufzeit, Größe und Lesbarkeit des Anzeigefeldes oder ähnlichen Eigenschaften. Im mobilen Umfeld ist jedoch der gegenüber einem stationären PC nicht verfügbare oder nur eingeschränkt nutzbare Netzzugang ausschlaggebendes Motivationskriterium für den Einsatz von Replikation und Caching.

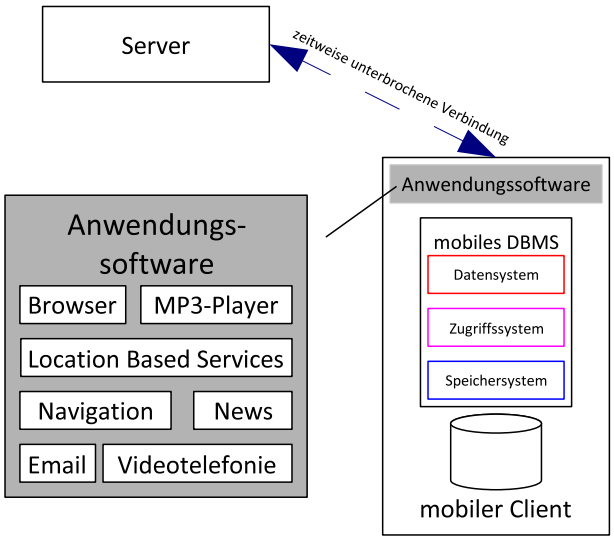

Abbildung 2. Anwendungssoftware auf mobilen Clients.

Abbildung 2 zeigt die Einordnung von mobilen Anwendungen in das beispielhafte Anwendungsszenario. Zu diesem Zeitpunkt wird die Anbindung an ein Datenbankverwaltungssystem jedoch noch nicht weiter betrachtet.

Bei den Programmen, die auf mobilen Endgeräten laufen, kann zwischen solchen, die keine, nur gelegentlich, schubweise oder wirklich dauerhaft eine Verbindung zur Außenwelt benötigen, unterschieden werden. Da die Notwendigkeit einer Verbindung für den Einsatz einer Anwendung im mobilen Umfeld grundlegend ist, nehme ich für den weiteren Verlauf eine Einteilung der Anwendungen in vier Klassen vor, wie Tabelle 1 zeigt. In der Literatur finden sich zur Klassifikation mobiler Anwendungen verschiedene andere Kriterien. Zu nennen sind hier insbesondere die Unterscheidung nach Teilnehmern der Anwendung – wie *business to consumer* (B2C) und *business to business* (B2B) – oder die Unterscheidung nach kommunikationszentrierten, transaktionsorientierten und inhaltszentrierten Anwendungen [4].

Klasse	Häufigkeit der Datenanforderung	Beispiele
1.	nie	E-Book-Reader, MP3-Player
2.	gelegentlich	E-Mail-Client, News-Reader
3.	schubweise	Internet-Browser
4.	dauerhaft	Videotelefonie, Media-Streaming

Tabelle 1. Klassifikation von Anwendungen nach notwendiger Verbindung

Zu der ersten Klasse von Anwendungen, die im mobilen Einsatz ohne Onlineverbindung auskommen, gehören Programme wie E-Book-Reader oder MP3-Player. Hier wird nur von Zeit zu Zeit ein Abgleich mit einem stationären Rechner durchgeführt, um Daten vom mobilen Gerät zu laden oder es mit neuen Daten zu füllen, jedoch stellt dieser Vorgang keine besonderen Anforderungen bezüglich der Synchronisation dar, weil der Benutzer gezielt auswählt, welche Daten wohin übertragen werden sollen.

Zu den Anwendungen, die gelegentlich eine Onlineverbindung brauchen, gehören E-Mail-Client und News-Reader. Hierbei können die Programme auch bei nur zeitweise verfügbarem Netzzugang ihre Daten aktualisieren und der Benutzer kann diese dann offline lesen.

Internet- und Wap-Browser, bei denen der Nutzer schubweise Daten anfordert, gehören eigentlich bereits in die Klasse der Anwendungen, die dauerhaft eine Netzwerkverbindung benötigen. Hier sind die Abstände zwischen den einzelnen Datenanforderungen so kurz, dass eine fehlende Verbindung dem Benutzer sofort auffallen würde und ein weiteres Arbeiten mit der Anwendung kaum möglich ist. Jedoch kann ein Browser mit zuvor geladenen Daten durchaus auch noch ohne Verbindung arbeiten, was den Unterschied zur vierten Klasse ausmacht.

Die vierte Klasse enthält Programme, die eine Onlineverbindung wirklich dauerhaft benötigen. Ohne oder mit unterbrochener Verbindung funktioniert ein Programm dieser Klasse nicht. Hier sind Anwendungen wie Videotelefonie,

Instant-Messaging oder Streaming von live gesendeten Inhalten einzuordnen. Kann man in einem Browser eine zuvor geladene Seite auch ohne Internetzugang noch weiter betrachten, arbeitet Videotelefonie ohne Netzzugang überhaupt nicht mehr. Bei dieser Klasse ist es nicht möglich, Daten im Voraus zu laden, da diese zu dem Zeitpunkt noch nicht existieren. Ein Zwischenspeichern der Daten für einen späteren Zugriff ist zwar möglich, allerdings kann damit die Verbindung nicht wirklich entlastet werden, da diese Daten durch die fortschreitende Zeit an Aktualität verlieren (eine bereits gesehen Nachrichtensendung am nächsten Tag nochmals zu sehen, dürfte nur für die wenigsten Nutzer von Interesse sein). Stattdessen benötigt der Benutzer bei diesen Anwendungen eher aktuelle Daten.

Alle bisher aufgeführten Anwendungen unterscheiden sich in Bezug auf die Verbindung neben der Bandbreite hauptsächlich durch die Intervalllänge zwischen den Zeitpunkten, zu denen Daten geladen werden müssen. Für den weiteren Verlauf dieser Arbeit wird hauptsächlich auf Programme der zweiten und dritten Klasse, also Programme, die gelegentlich oder schubweise ihre Daten übertragen, eingegangen, da durch Replikation und Caching auf der Seite des Clients nur auf diese Einfluss genommen werden kann. Die erste Klasse der Anwendungen ist weniger relevant, da der Zeitpunkt, zu dem wieder eine Verbindung zu einem stationären Rechner hergestellt wird, relativ unerheblich und damit unkritisch ist. Bei Anwendungen, die eine Echtzeitübertragung von Daten benötigen und somit der vierten Klasse zugeordnet sind, ist es nicht möglich, mit Techniken wie Caching oder Replikation eine verbesserte Konnektivität zu erzielen. Daher tritt diese Klasse für die weitere Betrachtung ebenfalls in den Hintergrund. Zu erwähnen ist noch, dass nicht jede Anwendung ganz eindeutig in eine der vier Klassen eingeordnet werden kann. Für den weiteren Verlauf dieser Arbeit reicht die Klassifikation jedoch aus.

Bei den im Weiteren betrachteten Klassen ist es das Hauptziel, die Länge der Intervalle, zu denen Daten übertragen werden müssen, zu maximieren und die Menge der zu übertragenden Daten zu minimieren. Idealfall hierbei wäre es natürlich, überhaupt keine Daten mehr übertragen zu müssen, was sich jedoch verständlicherweise nicht realisieren lässt. Selbst im Fall des MP3-Players, eigentlich die unkritischste der erwähnten Anwendungen der ersten Klasse, möchte der Benutzer früher oder später die Musik gegen etwas Neues austauschen.

2.2 Allgemeine Form von Caching

Unter Caching wird allgemein eine Technik verstanden, die durch den Einsatz einer mehrschichtigen Speicherstruktur versucht, den Zugriff auf Daten der unteren Schicht zu beschleunigen. Die Struktur ist dabei so ausgelegt, dass der Zugriff auf eine höhere Ebene schneller erfolgt, als auf eine darunter liegende Schicht. Caching versucht nun Daten, auf die bereits zugegriffen wurde und die mit großer Wahrscheinlichkeit nochmals benötigt werden, in einer höheren Schicht der Speicherstruktur zu halten und somit den Zugriff zu beschleunigen [5].

Neben Daten, die bereits gelesen wurden, können auch Daten in den Cache-Speicher geladen werden, die sich in physischer Nähe von Daten befinden, auf

die schon zugegriffen wurde. Grund für dieses Verfahren ist die Tatsache, dass die Daten in der Nähe mit einer erhöhten Wahrscheinlichkeit gegenüber anderen Daten ebenfalls für die Verarbeitung benötigt werden. Diese ergänzende Technik wird „Read Ahead“ genannt, da die Daten hierbei im Vorfeld der eigentlichen Bearbeitung gelesen werden. Diese Technik macht jedoch nur Sinn, wenn der Zugriff auf die im Voraus gelesenen Daten deutlich schneller erfolgen kann, als ein späteres Nachladen, da die zusätzlich geladenen Informationen möglicherweise gar nicht benötigt werden. Als Beispiel für einen solchen Fall sei der Zugriff auf eine Festplatte genannt, wobei Daten physikalisch vor und nach den benötigten Informationen – ohne erheblichen Mehraufwand – geladen werden können. Ein späterer Zugriff auf diese Daten würde durch die neue Positionierung des Lesekopfes jedoch erheblich länger dauern.

Da die Größe eines Cache-Speichers begrenzt ist, müssen geeignete Verfahren angewendet werden, um zu entscheiden, welche Daten im Speicher gehalten werden sollen. Man bezeichnet diese Verfahren als Verdrängungsstrategien, da sie über die Verdrängung von Daten aus dem Cache entscheiden. Ziel hierbei ist es Daten, die nicht mehr benötigt werden zu löschen und Daten, die wieder benötigt werden, im Cache zu halten. Grundsätzliche Strategien sind hierbei FIFO (*first in first out*)[3], LFU (*least frequently used*)[4] oder LRU (*least recently used*)[5]. Meistens findet jedoch eine Kombination dieser Ansätze statt, um die Effizienz des Cache-Speichers zu erhöhen.

Neben der Verdrängung muss sich der Cache auch noch um Änderungen an den Daten kümmern. Wird ein Datensatz geändert, so entsteht eine Inkonsistenz zwischen den Daten im Cache und der darunter liegenden Schicht. Hier besteht nun die Möglichkeit, die geänderten Daten aus dem Cache direkt in die darunter liegende Schicht weiter zu geben. Diese Schreibstrategie wird als *write through* bezeichnet. Sie hat jedoch den Nachteil, dass Daten die mehrfach geändert werden, unnötigerweise an die untere Schicht weiter gereicht werden. Alternativ können geänderte Daten auch erst bei der Verdrängung aus dem Cache an die untere Schicht weiter gereicht werden, was als *write back* bezeichnet wird. Allerdings muss hierbei sichergestellt werden, dass keine anderen Zugriffe direkt auf die untere Ebene erfolgen, da somit Daten gelesen würden, die durch die Änderung der oberen Speicherschicht nicht mehr gültig wären [6].

Wie bereits erwähnt, wird ein solcher Cache z. B. beim Zugriff auf Festplatten eingesetzt. Abbildung 3 zeigt den Cache einer Festplatte, der den physikalischen Speicherplatten (engl. Platters) vorgeschaltet ist. Dabei werden alle Anfragen von Daten immer direkt an den Cache gestellt, der die Daten dann mit Hilfe der oben beschriebenen Techniken nach außen zur Verfügung stellt.

2.3 Allgemeine Form von Replikation

Von Replikation wird immer dann gesprochen, wenn eine Kopie von Daten erzeugt wird und auf die ursprünglichen Daten weiter zugegriffen werden soll. Die

[3] Verdrängung des ältesten Eintrags.
[4] Verdrängung des am wenigsten gelesenen Eintrags.
[5] Verdrängung des am längsten nicht gelesenen Eintrags.

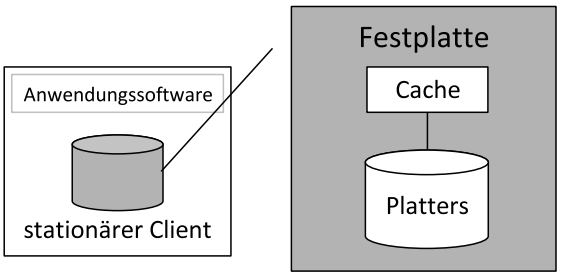

Abbildung 3. Der Einsatz von Cache in Festplatten.

Gründe für die Replikation von Daten können unterschiedlichsten Ursprungs sein. Als wichtigste Motivation sind jedoch sicherlich die Datensicherung oder die Erhöhung der Verfügbarkeit zu nennen. Auf Aspekte der Datensicherung soll an dieser Stelle jedoch nicht weiter eingegangen werden, da diese für mobile Anwendungen nicht zu einer Verbesserung der Konnektivität beitragen [5].

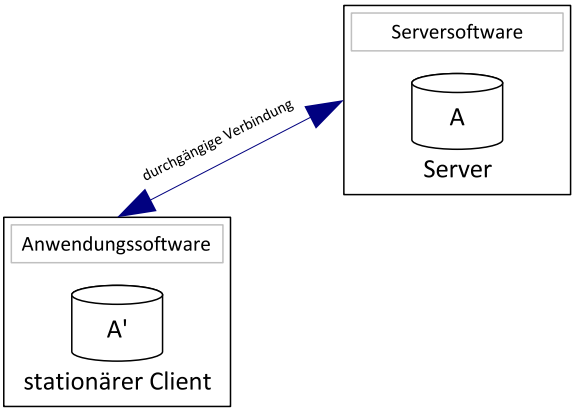

Abbildung 4. Replikation von Dateien in einem Netzwerk.

Abbildung 4 zeigt die Verteilung einer Datei in einem Netzwerk. Dabei befindet sich hier die ursprüngliche Datei A auf dem Server und deren Replikat A' auf dem stationären Client. Wird Replikation nun zur Verbesserung der Verfügbarkeit in einem Netzwerk eingesetzt, treten zwei wesentliche Vorteile auf. Zum einen können die Dateien schneller gelesen werden, da sie nicht mehr über das Netzwerk angefordert und verteilt werden müssen. Zum anderen sinkt die Auslastung des Netzwerkes, was sich direkt aus dem ersten Vorteil ergibt.

Die beiden bisher erwähnten Vorteile treten jedoch nur bei lesenden Zugriffen auf. Wird auf ein Replikat einer Datei oder deren Vorlage schreibend zugegriffen, so entsteht das Problem der Inkonsistenz zwischen Originaldatei und Replikat.

Zur Lösung dieses Problems wird die so genannte synchrone oder asynchrone Replikation eingesetzt, wobei sich die beiden Verfahren grundlegend unterscheiden [7].

Bei der synchronen Replikation wird gleichzeitig mit einer Änderung an einer Datei, die Änderung auch auf alle anderen verteilten Replikate angewendet (Atomarität). Die ändernde Anwendung erhält hierbei sofort die Information über Erfolg bzw. Misserfolg der Änderung aller Replikate. Wir betrachten an dieser Stelle ein verteiltes System, um dem Datenbankkontext noch nicht vorweg zu greifen. Bei dieser Betrachtung liegt der Vorteil in der Tatsache, dass immer nur eine Aktion ausgeführt wird und nicht, wie es bei Transaktionen nötig wäre, eine zusammenhängende Folge von Zugriffen. Damit muss das verteilte System nur eine identische Abfolge der Operationen auf den beteiligten Daten gewährleisten. Beispielhaft betrachten wir hier ein wie in Abbildung 4 gezeigtes System, an dem zwei Rechner beteiligt sind, die jeweils ein identisches Exemplar einer Datei halten. Der stationäre Client möchte nun eine Operation durchführen, die schreibend auf die Datei zugreift. Hierzu gibt es nun zwei Möglichkeiten: Zum einen kann er die Operation auf beiden Systemen gleichzeitig ausführen, wobei durch einen so genannten *atomic broadcast* sicher gestellt wird, dass die Reihenfolge der Operationen auf den Systemen identisch ist und nicht mit Operationen anderer Anwendungen vermischt wird. Hierbei erfolgt eine Rückmeldung über die Ausführung der Operation von jedem der beteiligten Systeme. Zum anderen kann der stationäre Client die Operation zuerst auf einem primären System ausführen, welches die Änderungen dann auf die andere Datei überträgt. Soll in Abbildung 4 der Server das primäre System sein, so beauftragt der stationäre Client den Server mit der Durchführung der Operation. Nach der Ausführung werden die Änderungen an der Datei auf den stationären Client übertragen. Die eigentliche Berechnung wird jedoch nicht auf dem stationären Client ausgeführt. Ein *view synchronous broadcast* stellt hierbei sicher, dass der Übergang von einem Zustand der Datei in einen Anderen in identischer Reihenfolge abläuft. Abschließend wird der stationäre Client noch vom Server über die Durchführung der Operation benachrichtigt [8].

Asynchrone Replikation hingegen wählt einen anderen Ansatz. Hierbei ist das Replikat nur zum Zeitpunkt der Erstellung mit der Originaldatei identisch. Anschließend kann auf beiden Exemplaren der Datei gearbeitet werden und es findet keine Kontrolle der Konsistenz statt. Somit muss zwischenzeitlich auch keine Netzwerkverbindung zwischen den verteilten Dateien bestehen. Nach einer bestimmten Zeit - z. B. wenn wieder eine Verbindung besteht - muss zwischen Replikat und Original geprüft werden, ob ein Konflikt besteht. Dabei gilt es zu unterscheiden, ob eines der beiden Elemente geändert wurde. Falls weder Replikat noch Original geändert wurden, liegt kein Konflikt vor. Hat sich eines der beiden Objekte geändert, so werden alle anderen Replikate durch das geänderte Objekt ersetzt. Hierbei ist jedoch zu beachten, dass die nicht geänderte Datei unter Umständen nach der Änderung der anderen Datei gelesen wurde. Im letzten Fall, wenn sich also beide Dateien geändert haben, ist die Lösung des Konfliktes deutlich komplizierter oder sogar unmöglich. Hier ist in der Regel

ein Eingreifen des Benutzers erforderlich, um festzulegen, welche Datei gültig ist oder welche Teile von welcher Datei zum neuen Original zusammengesetzt werden sollen[6]. Asynchrone Replikation kann natürlich auch bei mehr als einem Replikat angewendet werden, jedoch steigt die Anzahl der dadurch möglicherweise entstehenden Konflikte exponentiell an.

Als Beispiel für asynchrone Replikation seien hier die beiden Versionsverwaltungen *concurrent versions system* (CVS) und *subversion* (SVN) erwähnt, die mehreren Benutzern den gleichzeitigen Zugriff auf Dateien ermöglichen und ihn im Konfliktfall bei der Lösung unterstützen [9,10].

Wie aus der bisherigen Ausführung ersichtlich wird, senkt also eine steigende Anzahl von Replikaten die Kosten bei lesenden Zugriffen und steigert die Kosten bei schreibenden Zugriffen. Zudem wird mehr Speicherplatz benötigt, der bei den Kosten ebenfalls berücksichtigt werden muss. Da die somit in der Summe anfallenden Kosten von der Häufigkeit und Art der Zugriffe, sowie der Netzwerkstruktur abhängig sind, lässt sich der optimale Verteilungsgrad und -ort nur für einen konkreten Fall bestimmen. Dieses Problem wird in der Literatur auch als *file allocation problem* (FAP) bezeichnet und kann – bei bekannten Größen – relativ einfach gelöst werden [11].

3 Der Einsatz von Replikation und Caching in relationalen Datenbanksystemen

In diesem Abschnitt wird die Betrachtung des bisherigen Szenarios um Datenbankverwaltungssysteme erweitert. Dazu wird zunächst auf die Unterschiede zwischen Caching und Replikation eingegangen, um diese beiden Techniken besser voneinander abzugrenzen. Anschließend werden die für Caching und Replikation erforderlichen Anpassungen für den Einsatz im Umfeld von Datenbanken jeweils in einem eigenen Abschnitt erläutert. Innerhalb der Abschnitte wird zunächst das Vorgehen an einem einzelnen DBMS aufgezeigt und anschließend werden die Möglichkeiten für die Nutzung in verteilten Umgebungen beschrieben.

3.1 Abgrenzung der Aufgaben von Caching und Replikation

Auch nach der ersten Begriffserklärung in den Abschnitten 2.2 und 2.3 ist noch nicht wirklich klar, wie genau sich Caching und Replikation voneinander abgrenzen lassen. In der Literatur findet sich gelegentlich auch noch der Begriff des Hoardings von Daten, womit eine Abgrenzung nochmals erschwert wird [12]. Hoarding wird dabei als Technik bezüglich der Funktionalität zwischen Caching und Replikation eingeordnet. Um es bereits vorweg zu nehmen, auch hier wird eine absolut genaue Unterscheidung nicht möglich sein, einige wichtige Kriterien werden jedoch genauer betrachtet.

[6] Bei binären Dateien ist ein Zusammensetzen verschiedener Versionen in der Regel unmöglich.

Wichtigstes Unterscheidungskriterium zwischen Caching und Replikation ist der Einfluss, den der Benutzer auf die gespeicherten Daten nehmen kann. Während bei der reinen Replikation explizit durch den Anwender festgelegt wird, welche Daten repliziert werden, wird der Inhalt eines Cache durch seine Größe, die Einlagerungs- und Verdrängungsstrategie bestimmt. Der Benutzer nimmt auf den Inhalt nur indirekt Einfluss, indem er Daten anfordert, die dann im Cache zwischengespeichert werden. Diese Tatsache ist insbesondere relevant, wenn man einen mobilen Client betrachtet. Da der Inhalt eines Cache nicht durch den Benutzer beeinflusst werden kann, gibt es auch keine Garantie, bestimmte Daten im Cache zu finden. Daher eignet sich Caching nicht für ein völlig unverbundenes Arbeiten, sondern kann nur die zu übertragenden Daten reduzieren und somit die benötigte Bandbreite einer Verbindung verringern.

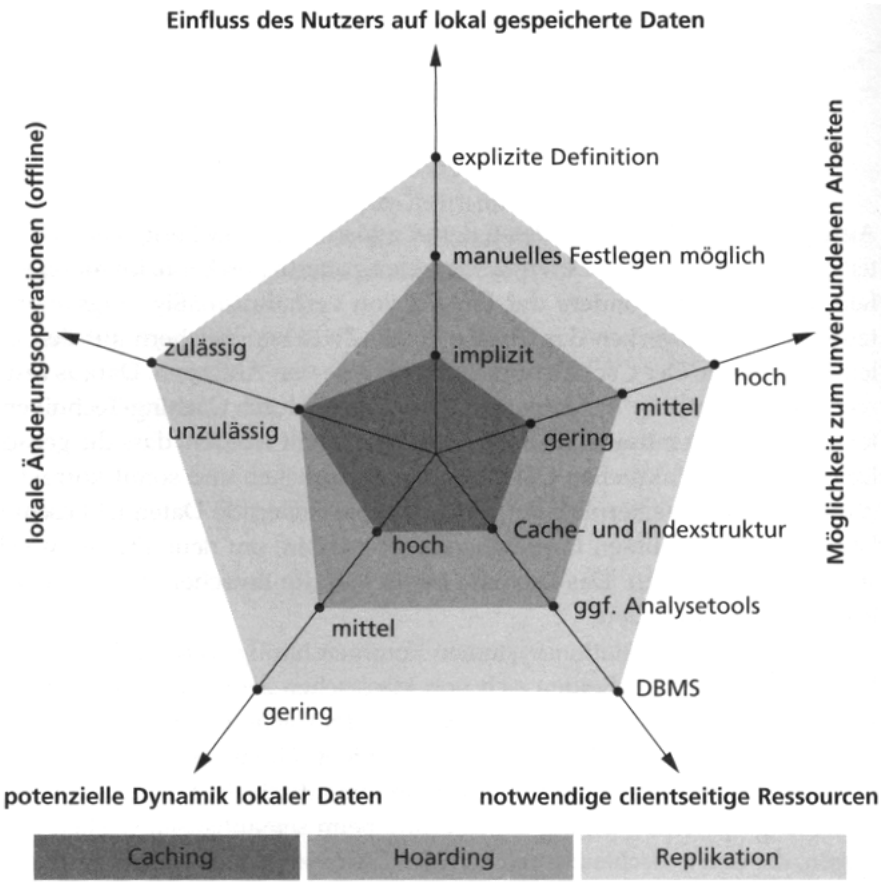

Abbildung 5. Abgrenzung von Caching, Hoarding und Replikation[13].

Aus dem Einfluss des Benutzers lässt sich direkt ein weiteres Kriterium zur Unterscheidung ableiten. Es handelt sich dabei um die Dynamik der gespeicherten Daten. Während ein Cache bedingt durch seine vergleichsweise geringe Größe einem ständigen Wechsel der Daten durch neue Anforderungen und somit bedingter Verdrängung unterliegt, erfolgt eine Änderung an Replikaten nur durch den schreibenden Zugriff eines Benutzers. Ein Cache hat somit eine sehr hohe Dynamik, wohingegen die Dynamik bei Replikation eher gering ist.

Abbildung 5 zeigt eine Abgrenzung zwischen Caching, Hoarding und Replikation. Neben den Kriterien „Einfluss des Nutzers auf lokal gespeicherte Daten“ und „potenzielle Dynamik lokaler Daten“ werden auch die weiteren Kriterien „notwendige clientseitige Ressourcen“, „lokale Änderungsoperationen (offline)“ und „Möglichkeit zum unverbundenen Arbeiten“ herangezogen. Wie die Auswahl der zusätzlichen Kriterien zeigt, wird hier bereits auf den mobilen Einsatz Bezug genommen. Es ist jedoch wichtig zu beachten, dass sich die Grenzen zwischen Caching, Replikation und Hoarding – je nach technischer Ausprägung – sehr leicht verschieben können. Nutzt ein Cache ein wie in Abschnitt 2.2 beschriebenes „Look Ahead“, wäre dieser bezüglich Abbildung 5 nicht mehr als Cache, sondern passender als Daten-Hoarding zu bezeichnen. Somit ist – wie bereits angedeutet – eine Einordnung vom eingesetzten System abhängig und nicht immer eindeutig möglich [13].

3.2 Caching in Datenbanksystemen

Für die Einführung von Caching bei Datenbanksystemen wird an dieser Stelle zunächst das in dieser Arbeit verwendete Schichtenmodell erläutert. Es kommt dabei ein vereinfachtes, auf drei Schichten reduziertes Modell zum Einsatz, wie es in [14] näher beschrieben ist. Dort wird auch der Grund für den Einsatz verschiedener Schichten erläutert, was jedoch für die weitere Darstellung von Replikation und Caching keine wesentliche Bedeutung hat. Abbildung 6 verdeutlicht die Einordnung in die bereits bekannte Struktur des Anwendungsszenarios. Der Grund für die Verwendung eines dreischichtigen Modells im Vergleich zum fünfschichtigen Modell, wie es in der Praxis bei relationalen Datenbanken zum Einsatz kommt, liegt in der Reduktion der Komplexität für den Betrachter. Um Caching und Replikation zu erklären, reicht das vereinfachte Modell an dieser Stelle aus.

Zunächst soll auf Caching innerhalb eines einzelnen Datenbanksystems (DBS) eingegangen werden. Neben dem Caching, das innerhalb des Festspeichers durch den Festplatten-Cache oder durch das Betriebssystem im Hauptspeicher erfolgt, ist eine eigene Schicht innerhalb des Datenbankmanagementsystems für Caching zuständig. Hierbei handelt es sich um das Speichersystem, welches sich neben der Pufferung von Daten auch um den Zugriff und die Verwaltung des externen Speichers kümmert[7]. Der Einsatz eines eigenständigen Caching-Mechanismus innerhalb des DBMS ist auf verschiedene Weise begründet. Dazu zählen eine optimale Anpassung an die Größe der Speichereinheiten – im Speichersystem des

[7] Im Modell mit fünf Schichten unterteilt sich das Speichersystem in zwei eigene Schichten für externen Zugriff und Pufferung.

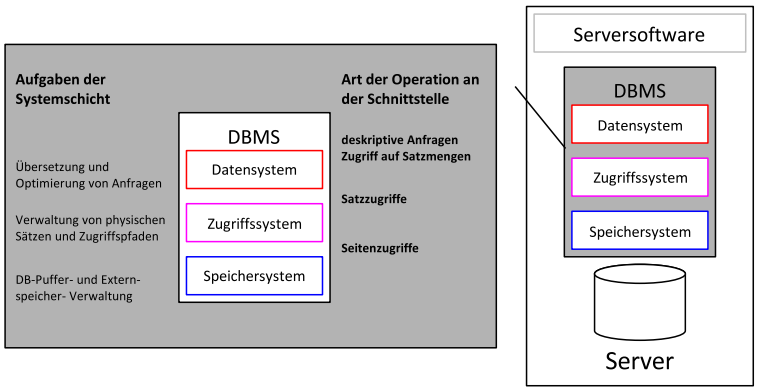

Abbildung 6. Dreischichtiges Architekturmodell.

DBMS sind dies üblicherweise Seiten – für den Zugriff durch das Zugriffssystem, das Sammeln von Logging-Informationen, um die Konsistenz der Daten auch im Fehlerfall zu gewährleisen, und der Einsatz verbesserter Verdrängungsstrategien, die auf den Zugriff der übergeordneten Schicht, wie das Durchlaufen von Suchbäumen, angepasst sind.

Der Mehrbenutzerbetrieb in DBS führt beim Caching zu dem Problem, dass eine gerade genutzte Seite bei einer notwendigen Verdrängung durch eine andere Anforderung nicht aus dem Cache entfernt wird. Hierzu wird bei der Anforderung der Seite explizit ein Flag gesetzt (FIX), das die Verdrängung der Seite verhindert. Hat die aufrufende Systemkomponente die Verarbeitung der Seite abgeschlossen, so wird das Flag wieder entfernt (UNFIX). Dementsprechend darf die eingesetzte Verdrängungsstrategie eine Seite mit einem solchen gesetzten Flag nicht verdrängen[8].

Zu berücksichtigen ist zudem, dass im Cache der Datenbank nicht nur Seiten gespeichert werden, die die eigentlichen Datenelemente enthalten. Es werden auch Zugriffsinformationen über die Datenelemente, wie Hashtabellen oder B*-Bäume, gespeichert. Durch die Verfügbarkeit solcher Informationen ist der Datenbank-Cache in der Lage ein gezieltes „Prefetching“ von Daten durchzuführen, wenn z. B. eine Baumsstruktur durchlaufen wird. Prefetching geht somit über ein einfaches „Read Ahead“ hinaus, da es sich nicht nur auf die physische Nähe von Datenelementen bezieht[9] [15].

In einer verteilten Datenbankumgebung kommt eine weitere Form des Caching zum Einsatz, welche die Zugriffslast auf die beteiligten Systeme minimieren soll. Sie kann auch dazu dienen, den notwendigen Datentransport zwischen den Systemen abzusenken. Diese Technik wird als Datenbank-Caching bezeichnet. Abbildung 7 zeigt den schematischen Aufbau. Datenbank-Caching kann dabei auch auf mehreren Clients eingesetzt werden. Die Idee beim Datenbank-Caching ist

[8] Dies wird auch als FIX/UNFIX-Mechanismus bezeichnet.
[9] Read Ahead kann als Teilmenge von Prefetching betrachtet werden.

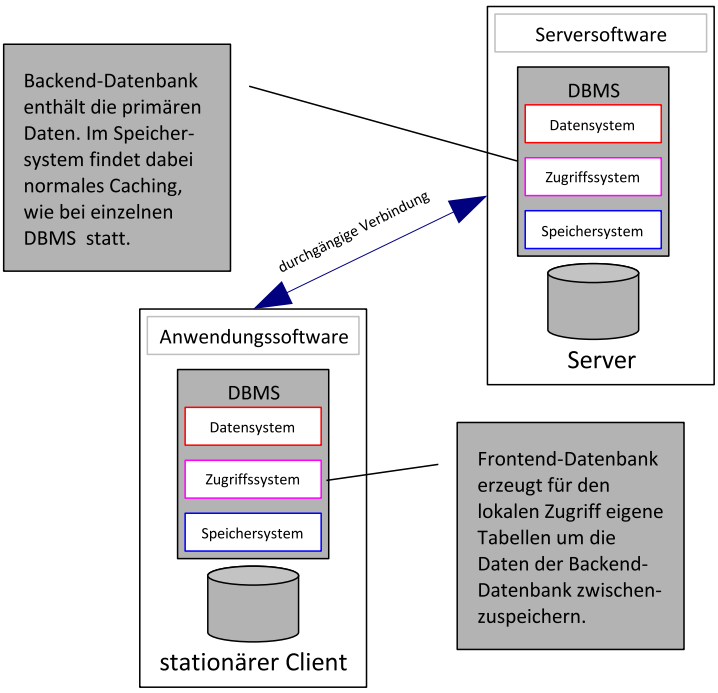

Abbildung 7. Datenbank-Caching

es, einen Teil der Daten des Backend-Datenbankservers, der die primären Daten hält, auch auf dem Client, wo die so genannte Frontend-Datenbank läuft, zu speichern. Kann eine Anfrage dann vollständig mit den Daten der Frontend-Datenbank beantwortet werden, muss die Anfrage nicht mehr an die Backend-Datenbank gerichtet werden. Die zusätzlichen Daten der Frontend-Datenbank werden dabei in normalen Relationen gespeichert, befinden sich also nicht in flüchtigem Speicher. Änderungen werden direkt auf der Backend-Datenbank ausgeführt. Diese ist auch dafür verantwortlich, Änderungen im eigenen Datenbestand an die Frontend-Datenbank zu propagandieren, sofern die Daten dort ebenfalls gespeichert sind. Datenbank-Caching bietet im Vergleich zu Caching in der Speichersystemebene einer einzelnen Datenbank deutlich mehr Setup-Parameter, was jedoch mit einem zusätzlichen administrativen Aufwand verbunden ist. Besteht beim Speichersystem nur die Möglichkeit, neben der Cachegröße die Verdrängungsstrategie festzulegen, muss beim Datenbank-Caching beispielsweise noch festgelegt werden, welche Art von Daten genau im Cache gehalten werden sollen[10].

Der Einsatz von Datenbank-Caching ist nur sinnvoll, wenn auf der Backend-Datenbank so viele Änderungsoperationen durchgeführt werden, dass eine Replikation der Daten – also eine vollständige Duplikation von Relationen – we-

[10] Möglich wäre z. B. nur das Caching von Daten einer bestimmten Relation.

gen des dadurch bedingten Synchronisationsaufwands keinen Leistungsvorteil bringt, aber zugleich so wenig Änderungen durchgeführt werden, dass durch Caching noch eine Reduktion der zu transportierenden Daten bewirkt werden kann. Datenbank-Caching ist, obwohl in der Frontend-Datenbank Replikate gehalten werden, bei den Caching-Verfahren einzuordnen, da das Kriterium der expliziten Festlegung der zu replizierenden Daten – wie dies für Replikations-Verfahren erforderlich wäre – nicht gegeben ist [16].

3.3 Replikation in Datenbanksystemen

Replikation innerhalb eines einzelnen DBMS mag auf den ersten Blick nicht sonderlich sinnvoll erscheinen, jedoch bringt ein solches Vorgehen bei gewissen Aufgaben durchaus einen Vorteil. Für ein einführendes Beispiel seien hier die beiden Begriffe *online transaction processing* (OLTP) und *online analytical processing* (OLAP) kurz erläutert. Das wichtigste Merkmal von OLTP sind kurze und häufig ausgeführte Trankaktionen, die unter Einhaltung der ACID-Eigenschaften ausgeführt werden und somit Transaktionssicherheit gewährleisten. Ziel ist hierbei eine genaue Abbildung der realen Welt innerhalb des DBMS. OLTP kommt bei Aufgaben zum Einsatz, bei denen die Daten im logischen Einbenutzerbetrieb verarbeitet werden müssen und Transaktionssicherheit einen hohen Stellenwert hat. OLAP hingegen wird bei der Analyse großer Datenmengen eingesetzt. Transaktionen haben hierbei sehr lange Laufzeiten, womit die Einhaltung der ACID-Eigenschaften nicht immer garantiert werden kann. Durch die – mit der langen Laufzeit von Transaktionen verbundenen – Sperren eines Zwei-Phasen-Commit Protokolls, ist Transaktionssicherheit kaum möglich, da die Wahrscheinlichkeit von Synchronisationskonflikten quadratisch zur Transaktionsdauer steigt [14,17,18].

Sollen nun jedoch Daten, die normalerweise unter Einhaltung der ACID-Eigenschaften verarbeitet werden, analysiert werden, bietet sich eine Replikation dieser Daten an, um den normalen Betrieb nicht durch die Analyse zu beeinträchtigen. Die Änderung der primären Daten während der Analyse werden bei den Ergebnissen somit jedoch nicht berücksichtig, was aber gerade in diesem Fall auch keine große Bedeutung hat. Die Ergebnisse beziehen sich somit auf den Zeitpunkt, zu dem das Replikat erstellt wurde. Aus diesem Grund bezeichnet man diese Art der Replikation auch Snapshot-Replikation [19].

Die Replikation von Daten für analytische Zwecke stellt einen relativ einfachen Fall der Replikation dar, da die Replikate nach der Analyse verworfen werden können und keine Synchronisation mit den Ausgangsdaten erfolgen muss. Soll jedoch mit den Replikaten gearbeitet werden, so wird eine aufwendige Synchronisation erforderlich, um die verschiedenen Versionen konsistent zu halten. Um genauer auf diese Problematik einzugehen, soll zunächst der Begriff der Serialisierbarkeit erläutert werden. In DBS wird im Verglich zu verteilten Systemen – wie in Abschnitt 2.3 beschrieben – statt mit einzelnen Operationen auf den Systemen, mit Transaktionen gearbeitet. Dabei kann eine Transaktion aus vielen Operationen bestehen, die auf verschiedene Objekte lesend und schreibend

zugreifen. Im Mehrbenutzerbetrieb könnte nun verschiedene Transaktionen nacheinander ausgeführt werden, um die Isolation zu gewährleisten. Dies ist jedoch aus Gründen der Systemleistung nicht akzeptabel, da immer auf das Ende einer Transaktion gewartet werden muss, bevor die Nächste begonnen werden kann. Daher werden auch einzelne Operationen verschiedener Transaktionen gemischt ausgeführt. Das Serialisierbarkeitskriterium gewährleistet hierbei die Isolation verschiedener Transaktionen. Dazu wird sichergestellt, dass das gemischte Ausführen von Operationen (Schedules) verschiedener Transaktionen mit der getrennten Ausführung einer beliebigen Transaktionsreihenfolge identisch ist [20].

Innerhalb eines einzelnen DBMS reicht der Mechanismus der Serialisierbarkeit bereits aus, um replizierte Daten synchron zu halten. Hierbei muss bei einer Änderung auf Daten innerhalb derselben Transaktion auch das Replikat geändert werden. Durch die Isolation von Transaktionen, kann eine andere Transaktion so entweder nur den Zustand vor oder nach der Änderung sehen.

Wie bei der allgemeinen Form von Replikation (Abschnitt 4), kann auch die bei der Replikation in DBMS zwischen synchroner und asynchroner Replikation unterschieden werden. Gerade im Umfeld von verteilten DBS sind hierfür jedoch die Begriffe *eager replication* und *lazy replication* gebräuchlicher. Analog zur synchronen Replikation bedeutet eager replication, dass alle verteilten Replikate aus Sicht des Benutzers gleichzeitig geändert werden, womit Mehrbenutzeranomalien ausgeschlossen werden. Lazy replication führt eine Transaktion hingegen zunächst aus und prüft später die Korrektheit der Ausführung [21].

In diesem Abschnitt soll zunächst nur auf eager replication eingegangen werden. Lazy replication wird in Abschnitt 4.2 näher betrachtet, da im mobilen Umfeld eager replication quasi nicht möglich ist. Betrachtet man Replikation für den Einsatz in verteilten DBS, wo in jedem DBS ein zuvor festgelegtes Replikat z. B. einer Relation gespeichert ist, reicht die einfache Serialisierbarkeit nicht aus, um die verteilten Daten synchron zu halten. Hier wird das Kriterium der 1-Kopie-Serialisierbarkeit[11] benötigt. Das Problem liegt darin, dass die Schedules zwar auf allen beteiligten DBS serialisierbar sein können, die Transaktionen auf den einzelnen Systemen jedoch nicht in derselben Reihenfolge ablaufen. Da Transaktionen im Allgemeinen nicht kommutativ sind, kann dies zur Inkonsistenz der Daten führen. Die 1-Kopie-Serialisierbarkeit fordert daher zusätzlich, dass die Transaktionen wie auf einer Datenbank ohne Redundanz serialisierbar sind [22].

Zwei Ansätze, die die Konsistenz der Daten sicherstellen, sollen an dieser Stelle etwas genauer betrachtet werden. Hierbei handelt es sich um die so genannten Write-All-Ansätze und die Primary-Copy-Verfahren. Das bekannteste der Verfahren der Write-All-Ansätze ist *read once write all* (ROWA). Hierbei werden alle Replikate sofort geändert, wodurch es irrelevant ist, welches der verschiedenen Replikate von einer Transaktion gelesen wird. Für den lesenden Zugriff stellt dies einen hohen Vorteil dar, da eine Transaktion nur eine Lesesperre auf ein einziges Replikat setzen muss. Bei schreibenden Zugriffen müssen jedoch alle Replikate gleichzeitig gesperrt werden. Zudem wird ein Schreiben unmöglich, sobald ein einziger Rechner, der ein Replikat hält, nicht mehr er-

[11] Meistens *one-copy serializability* genannt.

reichbar ist. Eine Abschwächung, die dieses Problem umgeht, ist die Variante *write all available,* wobei nur alle erreichbaren Replikate geschrieben werden. Ist ein Replikat dann wieder verfügbar, darf es jedoch erst wieder gelesen werden, nachdem es aktualisiert wurde [19,22].

Primary-Copy-Verfahren folgen einem anderen Ansatz, hierbei wird ein Repliktat als Primärkopie ausgewählt. Ändernde Zugriffe erfolgen immer auf diese Primärkopie. Die Änderung der Replikate erfolgt im Anschluss asynchron, durch das DBMS, welches die Primärkopie verwaltet. Durch das Halten von Sperren auf den anderen Replikaten wird hierbei jedoch eine Inkonsistenz vermieden, womit dieses Verfahren der Eager-Replikation zuzuordnen ist. Schreibende Zugriffe werden dabei im Vergleich zu ROWA-Strategien beschleunigt, da die Sperre auf der Primärkopie aufgehoben werden kann, bevor alle Replikate geändert wurden. Lesende Zugriffe werden jedoch im Vergleich zu Write-All-Verfahren verlangsamt, da Schreibsperren durch die asynchrone Änderung auf den Replikaten länger gehalten werden müssen. Zudem entsteht durch den Einsatz der Primärkopie ein kritischer Ausfallpunkt[12] [19,23].

Bei den verschiedenen Replikationstechniken bleibt jedoch genau wie bei den Caching-Ansätzen zu beachten, dass der durch die erhöhte Verfügbarkeit erzielte Leistungsvorteil durch die parallele Verarbeitung und die Entlastung des Netzwerkes nicht wieder durch die zusätzlichen Kosten für Synchronisation und den erhöhten Speicherplatzbedarf, sowie den zusätzlichen administrativen Aufwand aufgebraucht wird. Dabei ist die Wahl der eingesetzten Replikationstechnik in Abhängigkeit von der Nutzung der Daten entscheidend, wie die beiden vorherigen Ansätze zeigen [24].

4 Replikation und Caching im Umfeld mobiler Anwendungen

Dieser Abschnitt soll die Überführung der bisher beschriebenen Ansätze für Replikation und Caching in den Kontext mobiler Anwendungen aufzeigen. Ein großer Teil, der bisher vorgestellten Techniken, kann durch leichte Modifikationen dabei auch für den Einsatz auf mobilen Endgeräten eingesetzt werden. Zudem werden Ansätze und Ideen aufgezeigt, die speziell für den mobilen Einsatz angedacht sind und somit eine bessere Entlastung des Übertragungsweges erzielen, als die bisher vorgestellten Verfahren.

4.1 Caching im mobilen Einsatz

Kommt Caching auf einem mobilen Client zum Einsatz, kann zunächst einmal unterschieden werden, ob die verwendete Anwendung direkt mit einem Netzwerk kommuniziert, oder auf einem DBS aufsetzt. Im ersten Fall können Techniken eingesetzt werden, die sich bereits im stationären Betrieb an schmalbandigen Internetzugängen bewährt haben, um das zu übertragende Datenvolumen zu reduzieren. Beispielhaft sei hier das Caching von Webinhalten genannt, wo gerade

[12] Auch *single point of failure* genannt.

bei statischen Seiten und Bildern eine erhebliche Einsparung bei der Datenübertragung erreicht werden kann, wenn eine Internetseite mehrfach besucht wird. In [25,26] finden sich, neben dem Caching für Webinhalte auf Seite des Clients, Aufbau und Strukturierung von Web-Caching auf dem gesamten Weg vom Internetserver, der die Seite beherbergt, bis zum Client, auf dem sie dargestellt werden soll.

Aufgrund des Kontextes dieser Arbeit – Mobile and Context-aware Database Technologies and Applications – soll jedoch insbesondere auf Caching bei Anwendungen eingegangen werden, die auf einer Datenbank aufsetzen. Neben den in Abschnitt 3.2 beschriebenen Möglichkeiten, ergeben sich im mobilen Umfeld einige Änderungen, auf die im Folgenden eingegangen werden soll.

Änderung der Granularität. Eine wesentliche Idee zur Steigerung der Leistungsfähigkeit eines Datenbankcache, der auf einem moblilen Client ausgeführt wird, besteht darin, die Granularität der im Cache gespeicherten Elemente zu ändern. Während im Speichersystem normalerweise Seiten gespeichert werden, wird in [27] vorgeschlagen stattdessen die Granularität auf das Speichern von Attributen oder Objekten zu ändern[13]. Dabei wird gezeigt, dass sich durch diese Änderung mehr Daten aus dem Cache zur Beantwortung von Anfragen auf dem Client nutzen lassen und somit weniger Daten vom Server übertragen werden müssen. Der Ablauf einer solchen Anfrage ist dabei in drei Übertragungsstufen vorgesehen wie Abbildung 8 zeigt. Zunächst wird auf dem Client eine Anfrage

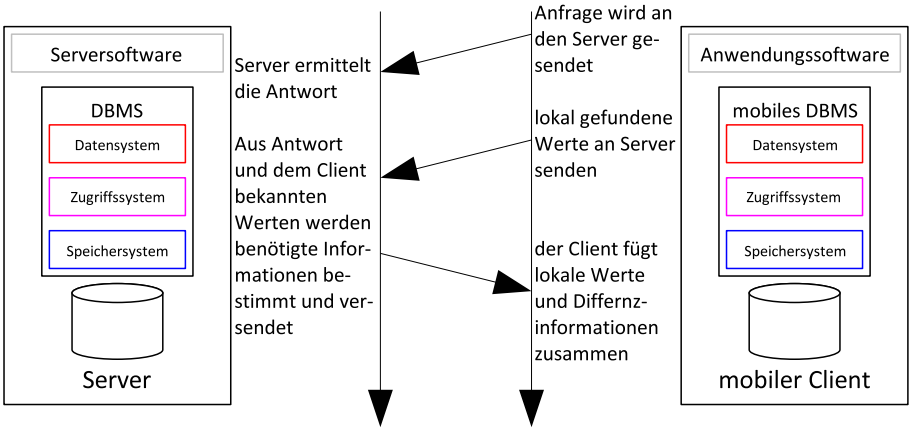

Abbildung 8. Ausführung einer Anfrage unter Ausnutzung des lokalen Cache [27].

gestellt, die direkt an den Server weitergereicht wird. Der Client bestimmt auf

[13] Diese Granularität bezieht sich auf objektorientierte Datenbanksysteme (OODB), kann jedoch auch auf Tupel in relationalen Datenbanken übertragen werden.

seiner Seite die Attribute, die er zur Beantwortung der Anfrage nutzen kann und sendet diese Information ebenfalls an den Server. Auf der Seite des Servers kann nun aus der vollständigen Antwort und den Informationen über die Daten auf der Seite des Clients ermittelt werden, welche Daten noch zu übertragen sind. Wurden Daten, die der Client noch im Cache hält, im Datenbestand des Servers geändert, kann dies ebenfalls durch den Server feststellt werden. Die geänderten, sowie noch nicht auf dem Client vorhandenen Informationen werden dann vom Server übertragen. Der Client muss nun nur noch die lokalen Daten um die Änderungen und die zusätzlichen Werte ergänzen und kann somit die Anfrage vollständig beantworten.

Semantisches Caching. Einen ähnlichen Schritt wählt das so genannte semantische Caching, wobei hier direkt auf die Speicherung von Tupeln eingegangen wird [28]. Semantisches Caching verwendet hierbei im Vergleich zum zuvor beschriebenen Verfahren eine genauere Beschreibung der im Cache gehaltenen Werte, die eine Auswertung und Verwaltung deutlich vereinfacht. Zur Beschreibung des Cache-Inhaltes werden dabei Prädikate eingesetzt. Es kann zwischen drei verschiedenen Ausprägungen der Beschreibung des Cache unterschieden werden. Bei der *exact cache description* (ECD) entspricht die Beschreibung des Cache-Inhaltes auch tatsächlich dem Inhalt. Die *conservative cache description* (CCD) ist eine Teilmenge der ECD wohingegen die *liberal cach description* (LCD) auch Elemente enthält, die nicht im Cache gespeichert sind [13]. Es gilt also $CCD \subseteq ECD \subseteq LCD$. Von der exakten Beschreibung des Cache abzuweichen macht durchaus Sinn, da eine genaue Beschreibung in der Regel mehr Prädikate erfordert und die Auswertung, ob eine Anfrage durch den Cache beantwortet werden kann, erschwert. LCE eignet sich um zu prüfen, ob eine Anfrage überhaupt mit dem Cache gelöst werden kann, wohingegen eine ECD und LCD direkt zum Beantworten von Fragen genutzt werden können[14]. Der Ablauf bei semantischem Caching ist an einem kurzen Beispiel sehr einfach zu verstehen:

> Wir betrachten eine Relation, die Informationen zu Restaurants (R) enthält und sich aus Name (NA), Adresse (AD) und dem durchschnittlichen Preis (DP) einer Speise in Euro zusammensetzt. Der Name des Restaurants soll der Primärschlüssel sein. Zur Beantwortung einer Anfrage sind vier Informationen notwendig: Die Cache-Beschreibung C (hier sei ECD angenommen), die eigentlich Anfrage Q, die Kompensationsanfrage Q^K – der Teil der Anfrage, der nicht mit dem Cache beantwortet werden kann – und die Filteranfrage Q^F – welche die benötigten Daten aus dem Cache anfordert. Der Verbund von Filter- und Kompensationsanfrage ergibt damit die Lösung der Anfrage.

[14] Auch mit LCD können Anfragen beantwortet werden, jedoch ist hierbei eine zusätzliche Überpüfung erforderlich, ob alle benötigten Daten tatsächlich im Cache vorhanden sind.

Cache-Beschreibung:	C = ({NA, DP},{DP<30}, R)
Anfrage:	Q = ({NA, AD, DP}, {DP<20}, R)
Kompensationsanfrage:	Q^K=({NA, AD}, {DP<20}, R)
Filteranfrage:	Q^F=({NA, DP}, {DP<20}, R)

Der Aufbau der vier Elemente ist hierbei gleich und beinhaltet eine Menge von Attributen, eine Menge von Prädikaten sowie die Relation auf die diese sich beziehen. Im Cache sind hier Name und durchschnittlicher Preis der Restaurants gespeichert, bei denen eine Speise im Durchschnitt weniger als 30 Euro kostet. Die Anfrage sucht nun Name, Adresse und Preise der Restaurants, bei denen der durchschnittliche Preis unter 20 Euro liegt. Daraus ergibt sich, dass noch Name und Adresse der Restaurants über die Kompensationsanfrage vom Server bezogen werden müssen, wo der Durchschnittspreis geringer als 20 Euro ist[15]. Die Daten aus dem Cache müssen noch gefiltert werden, da hier auch noch Restaurants enthalten sind, bei denen der durchschnittliche Preis zwischen 20 und 30 Euro liegt [13].

Das Beispiel zeigt, wie im Cache nicht vorhandene Attribute vom Server angefordert werden. Dies lässt sich jedoch auch einfach auf Fälle übertragen, bei denen der Wertebereich einer Abfrage nicht vollständig im Cache abgebildet ist. Der große Vorteil bei diesem Verfahren liegt in der semantischen Ähnlichkeit von Anfragen, die häufig im mobilen Umfeld bei diesen auftritt. So fragt ein Nutzer eines digitalen Restaurantführers mit hoher Wahrscheinlichkeit nur Restaurants im Umfeld seiner aktuellen Position ab. Durch diese Tatsache bedingt kann zudem noch durch eine semantische Analyse der Anfragen auf die Verdrängung von Daten aus dem Cache Einfluss genommen werden. So könnte nach mehreren Anfragen bzgl. günstiger Restaurants entschieden werden, eher teurere Restaurants aus dem Cache zu verdrängen [28].

Für Caching im mobilen Umfeld ist noch zu erwähnen, dass ein Prefetching in der Regel nur sinnvoll ist, wenn die Verbindung gerade ungenutzte Kapazität hat, denn ein Zugriff auf Daten ist hierbei in der Regel immer gleich schnell. Wohingegen bei Festplatten das Laden von benachbarten Daten zu einem späteren Zeitpunkt erheblich länger dauert (vgl. auch 2.2).

4.2 Replikation im mobilen Einsatz

Replikation in mobilen Datenbanken hat einen entscheidenden Unterschied zu Replikation in stationären verteilten Datenbanken. Sollen bei Mehrrechnerdatenbanken Leistung und Verfügbarkeit verbessert werden, so geht es bei Replikation auf einem mobilen Client darum, überhaupt arbeiten zu können, auch wenn keine Verbindung zu einer stationären Datenbank besteht.

Wir gehen für die folgende Betrachtung davon aus, dass der Client festgelegte Replikate einer Serverdatenbank enthält, die mit den Ausgangsdaten identisch

[15] Der Name wird vom Server als Primärschlüssel für den Verbund von Filter- und Kompensationsanfrage benötigt, auch wenn er bereits im Cache gespeichert ist.

(synchron) sind. Zudem nehmen wir an, dass die Verbindung zwischen Client und Server zumindest zeitweise unterbrochen ist und die Daten nach dieser Unterbrechung wieder synchronisiert werden sollen[16]. Hiermit ist klar, dass eine eager replication ohne Netzverbindung nicht möglich ist, da eine atomare Änderung der Daten auf dem Server nicht eingebracht werden kann [21]. Trotzdem gibt es Verfahren für den mobilen Einsatz, die sicherstellen, dass Transaktionen erfolgreich ausgeführt werden (konfliktvermeidende Synchronisation). Aus dieser Gruppe sollen hier *check-out/check-in*, das Escrow- und das Key-Pool-Verfahren kurz beschrieben werden.

Check-out/Check-in. Bei der mobilen Transaktionsverarbeitung könnten theoretisch auch normale Sperrverfahren eingesetzt werden, indem ein Client die benötigten Sperren setzt bevor die Verbindung unterbrochen wird. Im unverbundenen Zustand kann dann auf den gesperrten Objekten gearbeitet werden. Sobald wieder eine Verbindung besteht, werden geänderte Daten auf dem Server übernommen und die Sperren wieder frei gegeben. Check-out/check-in stellt hierbei einige Erweiterungen für Transaktionen mobiler Clients bereit. Neben einfachen Lese- und Schreibsperren werden noch Langzeitsperren und eine Check-in-Sperre eingesetzt. Tabelle 2 zeigt die Verträglichkeit der verschiedenen Sperren bei check-out/check-in. Die Anpassungen im Vergleich zu normalen Sperrverfahren

	R	W	LR	LW	C
R	+	-	+	+	-
W	-	-	-	-	-
LR	+	-	+	-	-
LW	+	-	-	-	-
C	-	-	-	-	-

R	Lesesperre (read)
W	Schreibsperre (write)
LR	Langzeitlesesperre (long read)
LW	Langzeitschreibsperre (long write)
C	Check-in-Sperre

Tabelle 2. Verträglichkeitsmatrix bei Langzeitsperren[13].

werden benötigt, da die Sperrverwaltung normalerweise im flüchtigen Speicher angesiedelt ist. Langzeitsperren müssen daher in den Festspeicher geschrieben werden, um auch nach einem Fehlerfall noch gesetzt zu bleiben. Zudem kann bei einer Langzeit-Schreibsperre noch lokal auf dem Server ein lesender Zugriff erfolgen, was bei einer reinen Schreibsperre nicht möglich wäre. Bei der Synchronisation werden die Langzeitsperren in Check-in-Sperren umgewandelt, wodurch

[16] Ohne eine abschließende Synchronistion würde auch die Snapshot-Replikation aus Abschnitt 3.3 ausreichen.

alle anderen Transaktionen, die auf Objekte mit Langzeitsperren arbeiten, zuerst beendet werden müssen (commit oder abort). Zudem darf eine neue Transaktion auf die gesperrten Objekte erst wieder nach dem Synchronisieren zugreifen. Damit wird die Serialisierbarkeit bei check-out/check-in sichergestellt. Der große Nachteil ist jedoch durch die langen Sperrdauern bedingt, zwar können weiterhin alle gesperrten Objekte gelesen werden, doch Änderungen sind ausgeschlossen. Diese Einschränkung schließt Check-in/Check-out somit aus, wenn weiterhin Schreibzugriffe möglich sein sollen [17].

Escrow-Verfahren. Das Escrow-Verfahren wurde ursprünglich entwickelt, um auf Datenobjekten, die häufig geschrieben werden müssen[17], ohne Sperren zugreifen zu können. Hierzu muss der zu ändernde Wert eine Zahl sein, welche die Kardinalität eine Menge repräsentiert (z. B. freie Sitze). Die Änderungen an dem Wert können zudem nur erhöhend oder herabsenkend sein [29]. Zu dem eigentlichen Wert müssen auch noch ein Infimum und ein Supremum gespeichert werden, die den niedrigsten bzw. höchsten möglichen Wert speichern[18]. Die genaue Menge kann bei noch laufenden Transaktionen nicht bestimmt werden. Die Funktion des Escrow-Verfahrens soll an einem kurzen Beispiel aufgezeigt werden.

> Wir betrachten ein Reservierungssystem, bei dem es 15 freie Plätze gibt. Damit dürfen die Werte 0 als Minimum und 15 als Maximum nicht überschritten werden[19]. Zudem nehmen wir zwei Transaktionen an, die auf den Wert zugreifen wollen. Tabelle 3 zeigt den Ablauf der Transaktionen und die zugehörigen Werte für Infimum, Kardinalität und Supremum. Zuerst verringert die erste Transaktion den Wert um 5. Da die

T_1	T_2	INF	Q	SUP
		10	10	10
-5		5	5	10
	+3	5	8	13
	COMMIT	8	8	13
COMMIT		8	8	8

Tabelle 3. Ein Beispiel für das Escrow-Verfahren.

> Transaktion noch nicht abgeschlossen ist, kann der Wert zwischen dem Infimum 5 (bei erfolgreichem Ende) und dem Supremum 10 (bei nicht erfolgreichem Transaktionsende) liegen. Die zweite Transaktion erhöht den Wert anschließend um 3. Damit kann der Wert zwischen 5 (T1 erfolgreich, T2 nicht erfolgreich) und 13 (T1 nicht erfolgreich, T2 erfolgreich)

[17] Auch *hotspots* genannt.
[18] Es handelt sich hierbei nicht um Minumum und Maximum!
[19] Eine mögliche Überbuchung sei hier ausgeschlossen.

liegen. Im nächsten Schritt wird die zweite Transaktion erfolgreich beendet, womit nur noch T1 scheitern kann. Somit liegt der Wert zwischen 8 und 13. Nachdem auch T1 erfolgreich beendet worden ist, kann die Kardinalität mit 8 wieder genau bestimmt werden.

Wichtig ist noch, dass Minimum und Maximum nicht unter- bzw. überschritten werden, falls solche Grenzen bestehen. Dazu muss zu jedem Zeitpunkt gelten: $INF \geq MIN$ und $SUP \leq MAX$. Soll das Escrow-Verfahren nun auf einem mobilen Client angewendet werden, muss vor der Unterbrechung festgelegt werden, um wie viel der mobile Client einen Wert maximal verringern und/oder erhöhen darf. Infimum und Supremum werden dann entsprechend verändert. Besteht später wieder eine Verbindung, so wird die tatsächlich ausgeübte Veränderung auf dem Server übernommen. Die Phase ohne Verbindung kann dabei wie im obigen Beispiel als lange Transaktion aufgefasst werden. Da keine Sperren benötigt werden, können lokale Transaktionen weiterhin auf den Daten arbeiten. Einzige Einschränkung ist, dass der mobile Client evtl. Änderungen „reserviert", sie aber nicht nutzt und eine lokale Transaktion daher nicht ausgeführt werden kann [17].

Key-Pool-Verfahren. Ein weiterer Problemfall kann durch das Key-Pool-Verfahren gelöst werden. Hier wird bei bestehender Verbindung zwischen Client und Server eine Menge von Primärschlüsseln vereinbart, die der Client nutzen darf, um neue Daten in eine Relation einzufügen. Bei der nächsten Synchronisation können die neu angelegten Datensätze des Clients einfach auf dem Server eingefügt werden, da dieser die Primärschlüssel wegen der vorherigen Absprachen nicht verwendet hat. Hier können jedoch nur synthetische Primärschlüssel[20] verwendet werden. Abhilfe schafft hier das Slot-Verfahren, das Primärschlüssel mit inhaltlicher Bedeutung, wie Raum- und Zeitinformationen in einer Raumverwaltung, reservieren kann. Hierbei ist es jedoch auf dem Server nicht möglich, vom Client reservierte Raum- und Zeitkombinationen zu nutzen [30].

Optimistische Synchronisation. Neben den konfliktvermeidenden Synchronisationsverfahren gibt es auch noch konfliktauflösende Synchronisation. Hierbei werden Transaktionen zunächst auf den replizierten Daten des Clients ausgeführt und bei der spätern Synchronisation wird versucht Änderungen mit dem Server abzugleichen. Man spricht daher auch von optimistischer Synchronisation. Die optimistische Synchronisation lauft allgemein immer in drei Phasen ab, wie Abbildung 9 zeigt. Dabei ist es egal, ob dieses Verfahren in einer einzelnen Datenbank, in einem verteilten DBS oder zwischen einer mobilen und einer stationären Datenbank zum Einsatz kommt. In der Lesephase findet die normale Transaktionsverarbeitung statt. Dabei werden Änderungen jedoch in einer Kopie gespeichert, die nur einer Transaktion zugeordnet ist. Nach dem Commit

[20] Syntetische Primärschlüssel dienen nur zur Identifikation und enthalten keine Informationen über das gespeicherte Tupel.

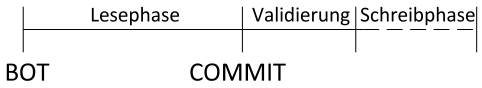

Abbildung 9. Phasen der optimistischen Synchronistion.

beginnt die Validierung, hierbei wird überprüft, ob ein Konflikt aufgetreten ist. Ist dies der Fall, so müssen eine oder mehrere Transaktionen zurückgesetzt werden, bis keine Konflikte mehr vorhanden sind. Ist eine schreibende Transaktion bei der Validierung nicht zurückgesetzt worden, so werden die Änderungen in die Datenbank übernommen [14].

In mobilen DBMS entspricht die Lesephase der Zeit, in der keine Verbindung zum Server besteht. Für die Validierung ist dann jedoch wieder eine Verbindung notwendig. Problematisch ist hierbei, dass Transaktionen bis zur nächsten Synchronisation nicht als ausgeführt bestätigt werden können. Dies stellt für den mobilen Einsatz eine erhebliche Einschränkung dar. Zudem steigt mit der Länge der Transaktion (bzw. der Zeit ohne Verbindung) die Wahrscheinlichkeit eines Konfliktes, was ein Zurücksetzen zur Folge hätte. Zwar kann die Transaktion dann wiederholt werden, jedoch besteht zu diesem Zeitpunkt wieder eine Verbindung und es ergibt sich kein Vorteil für ein unverbundenes Arbeiten [13].

5 Neue technische Ansätze und abschließendes Fazit

Neben den bisher aufgezeigten Methoden gibt es noch viele weite Möglichkeiten für den Einsatz von Caching und Replikation auf mobilen Clients. So wurde hier ein doch relativ homogenes Datenbankumfeld angenommen, wie es in der Realität wohl nur selten zum Einsatz kommt. In [31] wird ein dreischichtiges Architekturmodell vorgestellt, wobei eine Middleware für den Zugriff auch auf heterogene Datenquellen eingesetzt wird.

Ferner gibt es noch viel Forschungsarbeiten im Bereich von Caching und Replikation, die jedoch teilweise einen eher theoretischen Charakter haben. Ein Idee beschreibt den Einsatz eines verteilten Cache bei verschiedenen mobilen Clients [32,33]. Hierbei soll durch die Verteilung der zu cachenden Daten in einem – quasi ad-hoc – zusammengesetzten Cache die höhere Bandbreite der Kurzstreckentechniken (Bluetooth und WLAN) ausgenutzt werden, um den schmalbandigen Zugang zu entlasten. Dies ist wohl mehr eine futuristische Vision, denn selbst in dicht besiedeltem Gebiet wird man wohl in naher Zukunft keine ausreichende Dichte mobiler Endgeräte erreichen, um einen solchen Ansatz in der Praxis sinnvoll nutzen zu können.

Wie diese Arbeit aufgezeigt hat, ist es durchaus möglich durch den gezielten Einsatz von Caching und Replikation die Notwendigkeit einer dauerhaften Netzverbindung zumindest in einigen Anwendungsfällen aufzuheben und die für

die Nutzung der Anwendung zu übertragende Datenmenge deutlich zu reduzieren. Der technologische Fortschritt im Bereich der Übertragungstechniken, die immer weiter wachsende Rechenleistung von mobilen Geräten und größere Festspeicher erleichtern zudem die Arbeit auf mobilen Endgeräten und ermöglichen neue Anwendungen, die zuvor nur stationär möglich waren. [34] gibt einen Überblick über technische Entwicklungen im Bereich mobiler Kommunikation. Zudem wird eine größere Abstraktion von den technischen Details angedacht, um den Benutzer zu entlasten, damit er sich mehr mit den eigentlichen Anwendungen beschäftigen kann. Die Benutzerfreundlichkeit ist sicherlich einer der wichtigsten Aspekte für die weitere Entwicklung von Caching- und Replikationstechniken, denn bei allen neuen Entwicklungen sollte der Endanwender, als Nutzer dieser Techniken, stärker berücksichtigt werden.

Literatur

1. Kuri, J.: Gartner: Deutscher PC-Markt wächst um 3,6 Prozent. heise online - Heise Zeitschriften Verlag http://www.heise.de/newsticker/meldung/80426 (2006) Abgerufen am 28.04.2007.
2. Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.: Mehr Handys als Einwohner in Deutschland. http://www.bitkom.de/de/presse/43408_40990.aspx (2006) Abgerufen am 02.05.2007.
3. Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.: Mobilkommunikation. http://www.bitkom.de/de/markt_statistik/46261_38544.aspx (2007) Abgerufen am 04.06.2007.
4. Mutschler, B., Specht, G.: Mobile Datenbanksysteme. Springer-Verlag (2004)
5. Brockhaus Verlag: Brockhaus - Die Enzyklopädie Digital. Bibliographisches Institut & F. A. Brockhaus AG (2002)
6. Wikipedia.de: Cache. http://de.wikipedia.org/wiki/Cache Abgerufen am 01.05.2007.
7. Wikipedia.de: Replikation (Datenverarbeitung). http://de.wikipedia.org/wiki/Replikation_(Datenverarbeitung) Abgerufen am 01.05.2007.
8. Pedone, F., Wiesmann, M., Schiper, A., Kemme, B., Alonso, G.: Understanding replication in databases and distributed systems. In: ICDCS. (2000) 464–474
9. Homepage: Concurrent Versions System. http://www.nongnu.org/cvs/ Abgerufen am 18.06.2007.
10. Homepage: Subversion. http://subversion.tigris.org/ Abgerufen am 18.06.2007.
11. Gosh, Murthy, Moffett: File Allocation Problem: Comparision of Models with worst case and average Communication Delays. Operartion Research **40** (1992) 1074–1085
12. Kuenning, G.H., Popek, G.J.: Automated Hoarding for Mobile Computers. Computer Science Department, University of California, Los Angeles. (1997)
13. Höpfner, H., Türker, C., König-Ries, B.: Mobile Datenbanken und Informationssysteme - Konzepte und Techniken. dpunkt.verlag (2005)
14. Härder, T., Rahm, E.: Datenbanksysteme - Konzepte und Techniken der Implementierung 2. Auflage. Springer-Verlag (2001)
15. Elmasri, R., Navathe, S.B.: Grundlagen von Datenbanksystemen - 3. Auflage. Pearson Studium (2005)
16. Härder, T., Bühmann, A.: Datenbank-Caching - Eine systematische Analyse möglicher Verfahren. University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany. (2004)
17. Heuer, A., Saake, G.: Datenbanken: Konzepte und Sprachen - 2., aktualisierte und erweiterte Auflage. mitp-Verlag (2000)
18. Chamoni, P., Gluchowski, P.: Analytische Informationssysteme: Business Intelligence-Technologien und -Anwendungen. Springer-Verlag (2006)
19. Rahm, E.: Mehrrechner-Datenbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung. Addison-Wesley (1994)
20. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
21. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The dangers of replication and a solution. In: SIGMOD Conference. (1996) 173–182

22. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
23. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed ingres. In: Berkeley Workshop. (1978) 235–258
24. Buretta, M.: Data Replication: Tools and Techniques for Managing Distributed Information. John Wiley & Sons, Inc., New York, NY, USA (1997)
25. Rabinovich, M., Spatscheck, O.: Web Caching and Replication. Addison-Wesley (2001)
26. GeneSys: World-Wide Web Caching. Universität Kaiserslautern - Fachbereich Informatik. (2000)
27. Chan, B.Y.L., Si, A., Leong, H.V.: A Framework for Cache Management for Mobile Databases: Design and Evaluation. Distributed and Parallel Databases **10**(1) (2001) 23–57
28. Dar, S., Franklin, M.J., Jónsson, B.T., Srivastava, D., Tan, M.: Semantic data caching and replacement. In: VLDB. (1996) 330–341
29. O'Neil, P.E.: The escrow transactional method. ACM Trans. Database Syst. **11**(4) (1986) 405–430
30. Preguiça, N.M., Baquero, C., Moura, F., Martins, J.L., Oliveira, R.C., Domingos, H.J.L., Pereira, J.O., Duarte, S.: Mobile transaction management in mobisnap. In: ADBIS-DASFAA. (2000) 379–386
31. Helal, S., Hammer, J., Zhang, J., Khushraj, A.: A three-tier architecture for ubiquitous data access. In: AICCSA. (2001) 177–180
32. Chow, C.Y., Leong, H.V., Chan, A.T.S.: Distributed group-based cooperative caching in a mobile broadcast environment. In: Mobile Data Management. (2005) 97–106
33. Liu, B., Lee, W.C., Lee, D.L.: Distributed caching of multi-dimensional data in mobile environments. In: Mobile Data Management. (2005) 229–233
34. Arbanowski, S., van der Meer, S., Steglich, S.: I-centric Communications. Technische Universität Berlin. (2001)