Technische Universität Kaiserslautern

Diplomarbeit

Implementierung und Analyse von Synchronisationsverfahren für das CbDBC

Susanne Margot Maria Braun

September 2008

Fachbereich Informatik
AG Datenbanken und Informationssysteme
Technische Universität Kaiserslautern

Betreuer:
Prof. Dr.-Ing. Dr. h.c. Theo Härder
Dipl.-Inform. Joachim Klein

.

Ich versichere, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kaiserslautern, 01.09.2008

(Susanne Margot Maria Braun)

Inhaltsverzeichnis

1 Einleitung

Diese Diplomarbeit wurde im Rahmen des ACCache-Projekt [ACC] (Adaptive Constraint-based Database Caching) durchgeführt. Das Projektteam beschäftigt sich mit dem Datenbank-Caching. Die Idee dabei ist ein großes, zentrales Backend-Datenbanksystem, durch den Einsatz von Datenbank-Caches zu entlasten. Das Zwischenspeichern der Backend-Daten erfolgt adaptiv und dynamisch. Jeder Cache speichert immer gerade die Daten, die von den Anwendungen referenziert werden. Diese Daten können der Anwendung vom Cache schneller bereit gestellt werden. Dadurch wird Last vom Backend genommen, Antwortzeiten werden verkürzt und eine Verbesserung der Skalierbarkeit wird erreicht.

Das dynamische und adaptive Verhalten wird bei dieser Form des Datenbank-Caching durch den Einsatz von Constraints ermöglicht. Die Constraints stellen sicher, dass ein Datenbank-Cache bezüglich seines Inhaltes in einem konsistenten Zustand ist. Dadurch ist es dem constraint-basierten Cache möglich, eine Vielzahl gängiger SQL-Anfragen selbständig ohne die Backend-Datenbank auszuwerten. Bisher besteht dabei allerdings eine Beschränkung auf lesende Zugriffe. Im Rahmen dieser Arbeit wurde ein erster Schritt zur Unterstützung schreibender Zugriffe unternommen. Dies erforderte zunächst eine tiefgehende theoretische Analyse des Constraint-basierten Datenbank-Caching im Hinblick auf ACID-Transaktionen und Synchronisation.

Auf der Grundlage der theoretischen Erkenntnisse wurde der ACCache-Prototyp erweitert, sodass im Kontext des ACCache-Systems AC(I)D-Transaktionen ausgeführt werden können. Das ACCache-System ist eine reine Java-Implementierung und der Zugriff auf das ACCache-System ist über einen JDBC-Treiber möglich. Weiterhin stellt das System keine proprietäre Erweiterung eines existierenden Datenbanksystems dar, sondern ist eine Middleware, die weitgehend unabhängig von einem spezifischen Datenbanksystemhersteller ist.

1.1 Motivation

Es gibt bereits sehr viele Replikationslösungen, welche durch Lastverteilung eine bessere Skalierbarkeit erzielen. Allerdings ist dem Autor kein System bekannt, das auf

globaler Ebene die Serialisierbarkeit aller Transaktionen sicherstellen kann und den Zugriff auf veraltete Daten ausschließt. Das Replizieren von Daten, bei gleichzeitigem Sicherstellen von vollständigem ACID-Transaktionsschutz kann als offenes Problem angesehen werden. Da es sich bei den existierenden Replikationssystemen um vollständige Kopien der Backend-Datenbank handelt, oder um starre Partitionierungen derselben, eröffnet das dynamische Cachen der Daten, völlig neue Möglichkeiten, insbesondere auch im Bereich der Synchronisation.

Im Rahmen dieser Arbeit wurde versucht die Tragweite dieser Möglichkeiten zu erfassen. Der Prototyp wurde um ein erstes Synchronisationsverfahren erweitert. Dieser bildet eine Basis für weitergehende Untersuchungen und Leistungsmessungen. Die gewonnen Erkenntnisse können weiter ausgebaut werden, und in weiteren Arbeiten kann vielleicht das Ideal 1-Copy-Serializability für den verteilten Fall erreicht werden.

1.2 Zielsetzung

Das Ziel dieser Arbeit war zunächst, das Ausführen von Update-Operationen auf den Daten des ACCache-Systems in einem für die Anwendungstransaktionen "zumutbaren" Rahmen zu ermöglichen. Das bedeutet Update-Operationen sollten überhaupt (und zum ersten Mal) möglich sein, und dabei sollte außerdem eine Form von Transaktionsschutz gegeben sein. Das langfristige Ziel der ACCache-Arbeitsgruppe ist natürlich eine höchst mögliche Konsistenzstufe für seine Anwendungstransaktionen zu erreichen. Allerdings ist dies im Zuge einer Diplomarbeit nicht möglich.

Ein weiterer Schwerpunkt war, dass das Synchronisationsverfahren möglichst optimal auf das Constraint-basierte Datenbank-Caching zugeschnitten sein sollte. Die Freiheitsgrade, die beim Caching gegeben sind, sollten in einem positiven Sinne für die Synchronisation ausgenutzt werden. Außerdem stellt diese Arbeit auch einen Proof-Of-Concept für die Synchronisation beim Constraint-basierten Datenbank-Caching dar.

1.3 Gliederung

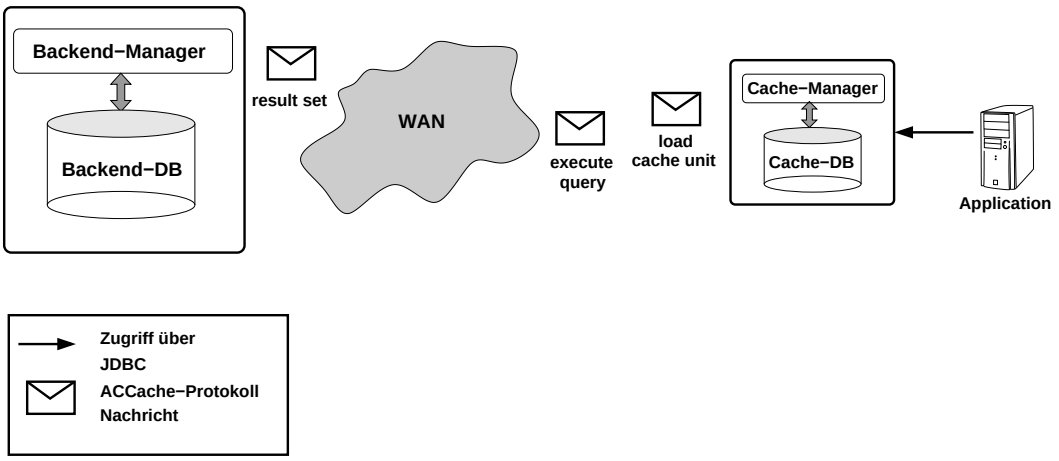
Als Einstieg wird zunächst in Kapitel 2 eine Einführung in das Constraint-basierte Datenbank-Caching gegeben. Im Anschluss daran, werden die wichtigsten Konzepte und Begriffe der Transaktionsverwaltung wiederholt. Bevor in das eigentliche Thema eingestiegen wird, werden in Kapitel 4 die Anforderungen an das zu erstellende Verfahren festgehalten und die bestehenden Kernprobleme identifiziert. Um die Konzeption des Verfahrens nachvollziehbar zu machen, beschäftigt sich Kapitel 5

allgemein mit der Datenreplikation. Es werden bestehende Replikationslösungen untersucht und analysiert. Die Folgerungen aus dieser Analyse fließen in die Umsetzung des Verfahrens ein. Schließlich wird in Kapitel 6 das entwickelte Verfahren vorgestellt. Dies beinhaltet einige Untersuchungen bezüglich der Korrektheit des Algorithmus. In Kapitel 7 werden die Erweiterungen, die am Prototyp vorgenommen wurden beschrieben. Die Arbeit schließt mit einer Zusammenfassung und Bewertung des Verfahrens.

2 Grundlagen des Constraint-basierten Datenbank-Caching

An dieser Stelle wird zunächst eine Einführung in das Constraint-basierte Datenbank-Caching gegeben. Ein Constraint-basierter Datenbank-Cache speichert die Daten einer zentralen Datenbank, der sogenannten *Backend-Datenbank*. Das Zwischenspeichern dieser Daten erfolgt dynamisch, und ist abhängig von den kürzlich erhaltenen Anfragen. Vorgehalten werden nur Daten mit hoher Lokalität, also Daten, die mit hoher Wahrscheinlichkeit in der nahen Zukunft referenziert werden. Ein Cache wird immer so installiert, dass zwischen Cache und Anwendung eine geringe Latenz und eine hohe Datenübertragungsrate gegeben ist. Bildlich befindet sich ein Cache in der Nähe der Anwendung. Zwischen Backend und Cache ist nicht zwingend eine geringe Latenz gegeben. Dazwischen liegt typischerweise ein Wide Area Network (WAN), wie zum Beispiel das Internet. In Abbildung 2.1 sind die beschriebenen Zusammenhänge graphisch dargestellt.

Abbildung 2.1 Constraint-basiertes Datenbank-Caching



Obwohl sich stets nur ein Bruchteil der Daten in der Backend-Datenbank im Cache befinden, ist es einem Constraint-basierten Cache möglich komplexe SQL-Anfragen

selbständig auszuwerten. Hierzu verwaltet der Cache zusätzliche Metadaten, die es ihm ermöglichen zu entscheiden, ob eine bestimmte Anfrage verarbeitet werden kann oder nicht. Dieser Entscheidungsprozess wird als *Sondierung* bezeichnet. Fällt diese negativ aus, so werden Teile der Anfrage oder die komplette Anfrage ans Backend delegiert.

2.1 Elemente eines Constraint-basierten Datenbank-Cache

Das Herzstück dieser Metadaten bildet die sogenannte *Cache Group*. Eine Cache Group besteht aus einer Menge von *Cache-Tabellen* und einer Menge von *Constraints*. Bei den Cache-Tabellen werden *Cached Tables* (oder: Cached-Tabellen) und Tabellen, die verschiedene Metadaten speichern, unterschieden. Eine Cached Table korrespondiert mit eine Tabelle der Backend-Datenbank. Das bedeutet eine Cached Table A kann die Tupel ihrer Backend-Tabelle A_B zwischenspeichern. Ein Constraint drückt eine Bedingung bezüglich der Existenz von Tupeln aus und betrifft maximal zwei Cached Tables. Der wichtigste Constraint ist der *Referential Cache Constraint* (kurz: *RCC*). Ist ein RCC zwischen zwei Cached Tables definiert, so ist unter anderem das Auswerten von Joins zwischen diesen beiden Cached Tables möglich. Ein weiteres Element bilden die sogenannten *Füllspalten*. Sie sind bei den Constraints einzuordnen und sind für die Steuerung der dynamischen Lade- und Entladevorgänge verantwortlich. In Abbildung 2.2 (a) ist ein Beispiel für ein Schema der Backend-Datenbank gegeben und in Teilabbildung (b) eine mögliche Cache Group zu diesem Schema. Der Cache kann Tupel aus den Backend-Tabellen $Mitarbeiter_B$ und $Abteilung_B$ zwischenspeichern. Zwischen diesen beiden Cached Tables gibt es einen RCC. Die Spalte $AbtNr$ ist eine Füllspalte. Die Bedeutung des RCC wird im Folgenden näher erläutert. Die Bedeutung der Füllspalte wird in Abschnitt 2.2 erklärt.

Dazu muss allerdings zunächst der Begriff der *Wertvollständigkeit* eingeführt werden. Die Forderung nach der Wertvollständigkeit eines Wertes w in einer Spalte $R.p$ ist zentral im Kontext des Constraint-basierten Datenbank-Caching. Ein Wert w ist wertvollständig in einer Spalte $R.p$, wenn alle Tupel aus R_B, die in Spalte $R_B.p$ den Wert w haben, in R sind (vgl. Definition 2.1). In Abbildung 2.3 ist die Backend-

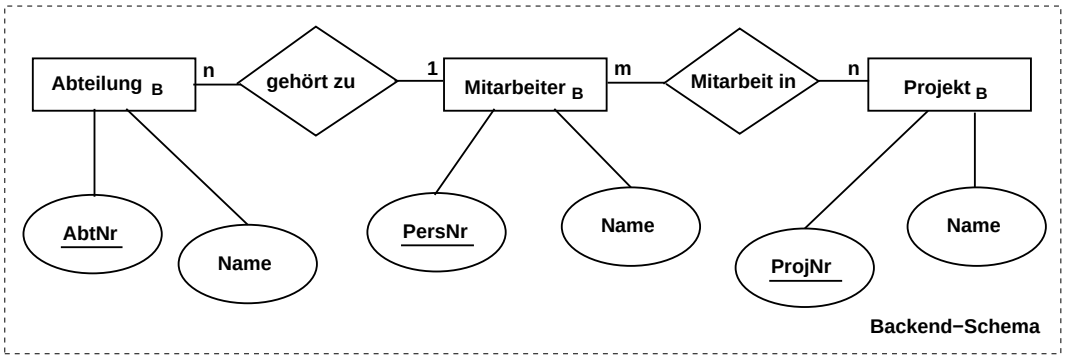
---

Definition 2.1 Wertvollständigkeit

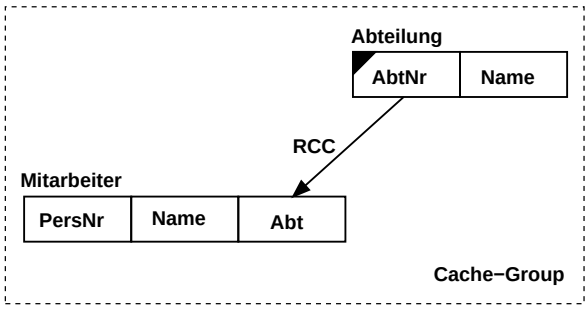
Ein Wert w ist wertvollständig (oder kurz vollständig) in einer Spalte $R.p \Leftrightarrow$ Alle Sätze aus $\sigma_{p=w}R_B$ sind in R. (vgl. [Büh06])

Tabelle $Mitarbeiter_B$ und die zugehörige Cached Table $Mitarbeiter$ dargestellt. Im Beispiel sind in der Spalte *Abt* die Werte 4711 und 4712 wertvollständig. Der Wert

Abbildung 2.2 Cache Group



(a) Beispielschema in der Backend-Datenbank



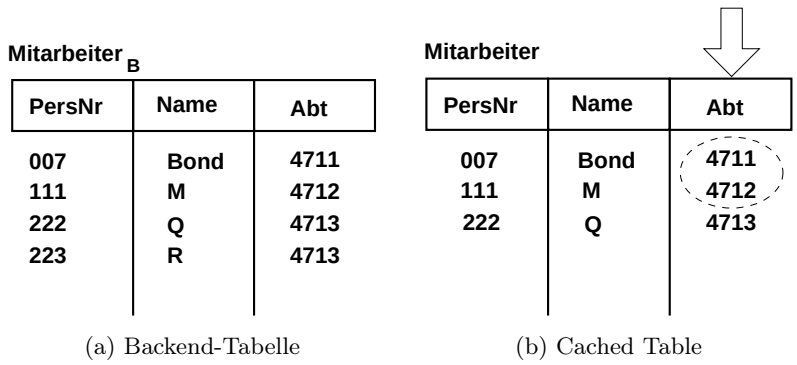
(b) Beispiel für eine Cache Group

4713 ist nicht wertvollständig, denn in der Cached Table ist nur der Mitarbeiter 'Q' geladen. Der Mitarbeiter 'R', der auch zu Abteilung 4713 gehört, wurde nicht in die Cached Table geladen.

Kommen wir nun zu der Definition des RCC. Ein RCC kann innerhalb einer Cache Group definiert werden und beschreibt eine Relation $(S.p, D.q)$ zwischen zwei Cached-Table-Spalten. Die Spalte $S.p$ ist die *Quellspalte* und die Spalte $D.q$ die *Zielspalte*. Deshalb ist auch die Notation $S.p \rightarrow D.q$ gebräuchlich. Ein RCC ist genau dann gültig, wenn für jeden Wert w, der in der Quellspalte $S.p$ auftritt, alle Tupel mit dem Wert w in $D_B.q$ in der Cached Table D geladen sind. Aus der Definition der Wertvollständigkeit ergibt sich dann: Ein RCC ist genau dann gültig, wenn jeder Wert w, der in der Quellspalte $S.p$ auftritt, wertvollständig in der Zielspalte $D.q$ ist.(vgl. Definition 2.2). S wird auch als *Qelltabelle* und D auch als *Zieltabelle* bezeichnet.

Für den RCC aus Abbildung 2.2 bedeutet dies: Wenn eine Abteilung in den Cache geladen wird, müssen auch alle Mitarbeiter dieser Abteilung in den Cache geladen werden. Betrachten wir dazu nochmals die Backend-Tabelle $Mitarbeiter_B$ aus

Abbildung 2.3 Wertvollständigkeit



(a) Backend-Tabelle (b) Cached Table

Abbildung 2.3 (a). Wenn die Abteilung mit der Nummer 4713 in den Cache geladen wird, dann muss 4713 in der Spalte *Mitarbeiter.Abt* wertvollständig sein. Und dann müssen in jedem Fall die beiden Mitarbeiter 'Q' und 'R' geladen werden.

Definition 2.2 Referentieller Cache-Constraint (RCC)

Ein RCC $S.p \rightarrow D.q$ von der Quellspalte $S.p$ zur Zielspalte $D.q$ ist erfüllt $\Leftrightarrow$ Alle Werte, die in $S.p$ existieren, sind wertvollständig in $D.q$. (vgl. [Büh06])

Aus der Definition des RCC (vgl. Definition 2.2) folgt direkt, dass jede Spalte $D.q$ wertvollständig ist für jeden Wert w, der in der Quellspalte $S.p$ eines eingehenden RCC vorkommt (siehe Beobachtung 2.1).

Beobachtung 2.1 Ableiten von Wertvollständigkeit

Sei $w \in S.p$, $S.p$ sei Quellspalte des RCC $S.p \rightarrow D.q \Rightarrow w$ ist wertvollständig in $D.q$. (vgl. [Büh06])

2.2 Vorgehensweise beim Laden von Backend-Daten

Das Lade- und Entladeverhalten wird vom Design der Cache Group bestimmt und dieses hängt gegenwärtig vom Expertenwissen des Cache-Administrators ab. Konzep-

tionell wird es durch die Füllspalten bestimmt. Mit jeder Füllspalte $R.f$ wird eine *Kandidatenmenge* K assoziiert. Wird in einer Anfrage ein *Kandidatenwert* $w \in K$ referenziert, so wird w wertvollständig in $R.f$ gemacht. Diese Forderung nach Wertvollständigkeit setzt sich rekursiv für jeden von R ausgehenden RCC fort. Das Referenzieren eines Wertes w aus einer Kandidatenmenge K löst somit einen rekursiven Ladevorgang aus. Anschaulich gesprochen pflanzt sich das Laden von Tupeln auf allen Tabellen entlang jeder Kette von ausgehenden RCCs fort.

In Abbildung 2.4 ist für die Cache Group aus Abbildung 2.2 ein Ladevorgang illustriert. In Teilabbildung (a) sind die Backend-Tabellen der Cache Group, und in Teilabbildung (b) eine Beispiel-Query mit Kandidatenmenge dargestellt. In der Query wird der Kandidatenwert 4713 referenziert. Deshalb wird, wie in Teilabbildung (c) dargestellt, zunächst dieser Wert vollständig in der Spalte *AbtNr* gemacht. Da es sich um eine Unique-Spalte handelt, muss nur ein Tupel geladen werden. Die Spalte besitzt aber noch einen ausgehenden RCC auf die Zieltabelle *Mitarbeiter*. Deshalb müssen auch alle Mitarbeiter der Abteilung in die Cached Table geladen werden.

Abbildung 2.4 Laden und Füllspalten

Abteilung $_B$

AbtNr	Name
4711	Agenten
4712	Chefs
4713	Genies

Mitarbeiter $_B$

PersNr	Name	Abt
007	Bond	4711
111	M	4712
222	Q	4713
223	R	4713

(a) Backend-Tabellen

```
SELECT Name
FROM Abteilung
WHERE AbtNr = 4713;
```

Kandidatenmenge = { 4711, 4713}

(b) Kandidatenmenge und Beispiel-Query

Abteilung

AbtNr	Name
4713	Genies

Mitarbeiter

PersNr	Name	Abt
222	Q	4713
223	R	4713

(c) Cache-Tabellen

Ein Ladevorgang wird ausschließlich durch die explizite Referenz eines Kandidatenwertes w ausgelöst. Backend-Daten können nur auf diesem Weg in den Cache gelangen. Die Tupelmengen, die aufgrund der Constraints zusammen mit einem Füllwert in den Cache geladen werden müssen, werden als *Cache-Units* bezeichnet (vgl. Definition 2.3). Sie stellen die Einheiten des Ladens und Entladens im Cache-System dar.

Definition 2.3 Cache-Unit

Sei $R.f$ eine Füllspalte mit der Kandidatenmenge K. Sei $w \in K$. Die Menge aller Tupel, die aufgrund von $(w) \in R.f$ im Cache existieren muss, heißt Cache-Unit von w in $R.f$ (kurz: $CU_{R.f}(w)$).

2.3 Sondierung

Ein Datenbank-Cache speichert, wie bereits erwähnt, zu jedem Zeitpunkt nur einen Bruchteil der Daten des Backend. Um eine Anfrage auswerten zu können, muss somit zunächst ermittelt werden, ob der Cache alle Daten enthält, die zur Ableitung der korrekten Ergebnismenge benötigt werden. Dieser Vorgang wird als *Sondierung* bezeichnet. Bei Datenmodellen, die deskriptive Anfragen auf ihren Daten zulassen, werden Prädikate zur Selektion der Datensätze verwendet. Dies gilt insbesondere für das relationale Datenmodell und die relationale Algebra. In diesem Zusammenhang werden zwei weitere wichtige Begriffe eingeführt, nämlich die *Prädikatextension* (vgl. Definition 2.4) und die *Prädikatvollständigkeit* (vgl. Definition 2.5).

Definition 2.4 Prädikatextension

Die Prädikatextension des Prädikates P ist die Menge aller Datensätze, die benötigt werden, um das Prädikat P auszuwerten. (vgl. [HB07])

Definition 2.5 Prädikatvollständigkeit

Eine Menge von Tabellen T ist prädikatvollständig bezüglich des Prädikats $P \Leftrightarrow$ In den Tabellen von T sind alle Tupel der Prädikatextension von P enthalten. (vgl. [HB07])

Aufgabe der Sondierung ist es damit, zu einer Anfrage A zu ermitteln, ob die Prädikatextension P_A von A Teilmenge einer bereits im Cache vorhandenen Tupelmenge ist. Die Sondierung muss schnell und effizient durchführbar sein. Denn Sie muss vor

jeder Anfrage durchgeführt werden. Ist die Sondierung negativ, muss die Anfrage oder Teile davon an das Backend delegiert werden.

Wertvollständigkeit von w in $R.p$ bedeutet, dass die Menge aller Tupel, die durch das Prädikat $R.p = w$ selektiert werden, im Cache ist. Deshalb bezeichnen wir die Selektion eines Wertes w auf einer Spalte $S.p$, in der w wertvollständig ist, als *Einstiegspunkt*, da bekannt ist, dass alle Tupel, die zur Auswertung der Selektion benötigt werden, im Cache sind. Hat S einen ausgehenden RCC $S.p \rightarrow D.q$ auf Spalte q, so wissen wir, dass jeder Wert w', der in $S.q$ auftritt, wertvollständig in der Zielspalte $D.r$ ist. Damit ist die Extension des Prädikates $S.p = w \wedge S.q = D.r$ im Cache. Dieses Prinzip lässt sich entlang jeder Kette von ausgehenden RCCs rekursiv fortsetzen. Hat man einen Einstiegspunkt gefunden, so sind alle Verbundpartner für eine Folge von Equijoins entlang jeder Kette von ausgehenden RCCs in den Tabellen der RCC-Kette vorhanden. Zusätzlich können auf jeder Tabelle der RCC-Kette beliebige Selektionen ausgeführt werden, da sie die Ergebnismenge lediglich weiter einschränken.

Betrachten wir am Beispiel aus Abbildung 2.4 die Anfrage aus Listing 2.1: In der `WHERE`-Klausel wird die wertvollständige Abteilungsnummer 4713 referenziert. Somit ist ein Einstiegspunkt gefunden. Der Join zwischen Mitarbeiter und Abteilung geht entlang eines ausgehenden RCC, und es kann gefolgert werden, dass alle Verbundpartner zu Abteilung 4713 im Cache sind. Die Sondierung ist positiv und kann vom Cache beantwortet werden. Aber auch die Anfrage aus Listing 2.2 kann vom Cache verarbeitet werden. Diese stellt noch eine zusätzliche Bedingung an die Ergebnismenge und ist damit in der Ergebnismenge der vorangegangenen enthalten.

Listing 2.1 Sondierung: Beispiel 1

```
SELECT A.Name AS Abteilung, M.Name AS Mitarbeiter
FORM Abteilung A, Mitarbeiter M
WHERE A.AbtNr = 4713
AND A.AbtNr = M.Abt
```

Zusammenfassend kann also folgender Anfragetyp vom Cache beantwortet werden: Eine Konjunktion von Equijoins entlang einer Kette von RCCs. Zusätzlich darf auf jeder Tabelle der RCC-Kette beliebig selektiert werden. Auf der ersten Relation der RCC-Kette muss es aber mindestens eine Selektion eines vollständigen Wertes geben. Für die Sondierung ist es also wichtig, zu wissen, welche Werte in einer Spalte wertvollständig sind. Für Füllspalten $R.f$ kann dies in trivialer Weise ermittelt werden: Jeder Wert in der Füllspalte hat diese Eigenschaft. Nach Beobachtung 2.1 ist außerdem jeder Wert w wertvollständig in einer Spalte $D.q$, wenn er in der Quellspalte $S.p$ eines RCCs $S.p \rightarrow D.q$ auftritt.

Listing 2.2 Sondierung: Beispiel 2

```
SELECT A.Name AS Abteilung, M.Name AS Mitarbeiter
FORM Abteilung A, Mitarbeiter M
WHERE A.AbtNr = 4713
AND A.AbtNr = M.Abt
AND M.Name = 'Q'
```

2.3.1 Erreichbarkeit in einer Cache Group

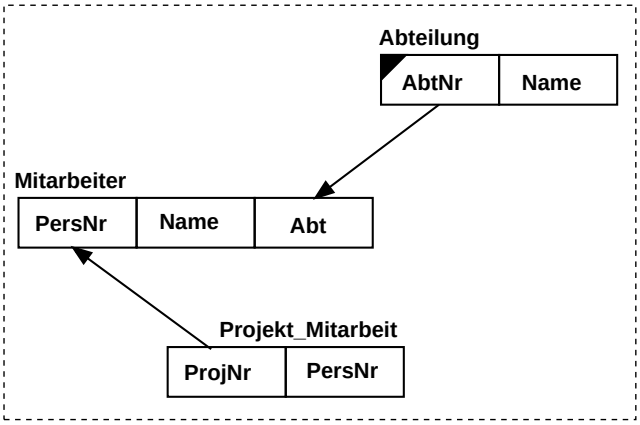
Im Folgenden soll etwas zur Sichtbarkeit bzw. Erreichbarkeit von Datenbankobjekten in einer Cache Group gesagt werden. Eine Anfrage wird nur dann auf dem Cache ausgewertet werden, wenn die Sondierung positiv war. Weil auch die Füllspalten intern über RCCs implementiert werden (vgl. [HB07]), ist dies aber nur dann möglich, wenn die Tupel direkt oder über mehrere Stufen von RCCs referenziert werden. Einzige Ausnahme bilden Einstiegspunkte auf Unique-Spalten. In einer Unique Spalte ist jeder geladene Wert direkt wertvollständig (vgl. [HB07]). Deshalb kann hier durch Überprüfung der Existenz eines Wertes auf Wertvollständigkeit geschlossen werden.

Bis auf diese Ausnahme gilt die folgende Regel: Jede Tabelle, die keine eingehenden RCCs besitzt ist im Cache unsichtbar oder unerreichbar. Damit Tupel jemals in eine Cached Table geladen werden können, genügt mindestens ein eingehender RCC nicht. Die Tabelle muss von einer Füllspalte über eine Kette von RCCs erreichbar sein. Wenn niemals Tupel in eine Cached Table geladen werden können, dann kann für diese auch niemals erfolgreich sondiert werden. In der Cache Group aus Abbildung 2.5 ist eine im Cache unerreichbare Tabelle *Projekt_Mitarbeit* gegeben.

2.4 Zusammenfassung

In diesem Kapitel wurden die Grundlagen des Constraint-basierten Datenbank-Caching vorgestellt. Dabei wurde die Cache Group als das zentrale Element eines Constraint-basierten Datenbank-Cache vorgestellt. Die RCCs einer Cache Group nehmen eine Sonderstellung ein. Sie ermöglichen das effiziente Ableiten von Informationen, insbesondere bezüglich des Cache-Inhalt. Die RCCs referenzieren bestimmte Datenmengen, und stellen gleichzeitig die Erreichbarkeit und Verfügbarkeit all dieser Daten sicher.

Abbildung 2.5 Erreichbarkeit in einer Cache Group



3 Grundlagen der Transaktionsverwaltung

Dieses Kapitel gibt eine Einführung in die Grundlagen der Transaktionsverwaltung. Zunächst werden grundlegende Begriffe, wie Transaktion, Transaktionsprogramm und Historie definiert. Anschließend werden die Anomalien, die im unkontrollierten Mehrbenutzerbetrieb auftreten können vorgestellt. Zur Vermeidung dieser Anomalien müssen Transaktionen synchronisiert bzw. *isoliert* werden. Für den Grad der Isolation gibt es eine Klassifikation in sogenannte *Isolation Levels* oder *Konsistenzstufen*. Auch die Transaktionen des ACCache-Systems müssen zur Vermeidung dieser Anomalien synchronisiert werden. Die Grundlagen sind wichtig für das allgemeine Verständnis der Synchronisationsprobleme im verteilten Fall. Sie werden außerdem benötigt, um zu zeigen, dass das entwickelte Synchronisationsverfahren wichtigen Anforderungen genügt.

3.1 Transaktionen

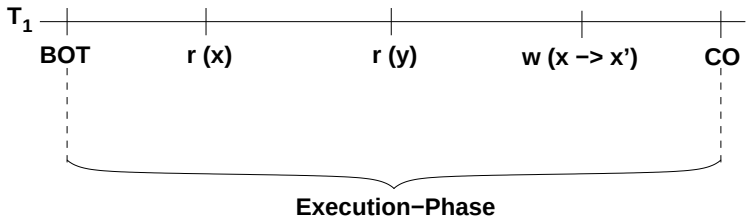
Jede *Datenbank-Transaktion* (kurz: *Transaktion*) besteht aus einer Folge von Datenbank-Operationen. Zusätzlich besitzt eine Transaktion einen eindeutigen Start- und Endzeitpunkt. Der Startzeitpunkt wird als *Begin of Transaction* (kurz: *BOT*) bezeichnet und durch die BOT-Operation eingeleitet. Das Ende einer Transaktion *End of Transaction* (kurz: *EOT*) wird durch die speziellen Operationen *Abort* oder *Commit* ausgelöst. Der eindeutige Commit-Zeitpunkt wird auch mit der Abkürzung *CO* bezeichnet. Bei allen weiteren Operationen handelt es sich um gewöhnliche Lese- oder Schreib-Operationen. Für letztere wird auch der synonyme Begriff Änderungs-Operationen gebraucht. Das Lesen eines Datenelementes x wird mit $r(x)$ abgekürzt, Schreiben entsprechend mit $w(x)$ oder $w(x \mapsto x')$. Eine Transaktion, die mindestens ein Datenelement schreibt, wird als schreibende Transaktion oder kurz als Schreiber bezeichnet. Besitzt eine Transaktion ausschließlich lesende Operationen, so handelt es sich um eine lesende Transaktion (kurz: Leser). Alle Abkürzungen werden in dieser Arbeit durchgängig verwendet und sind in Tabelle 3.1 aufgeschlüsselt. In Abbildung 3.1 ist die graphische Notation für eine beispielhafte Transaktionen T_1 dargestellt. Die Transaktion liest zunächst die Datenelemente x und y. Nach dem Schreiben von x führt T_1 die Commit-Operation aus. Der endgültige Commit-Zeitpunkt ist

entsprechend mit *CO* markiert. Der Abschnitt zwischen *BOT* und *CO* bzw. *AB* wird auch als *Ausführungsphase* von T_1 bezeichnet.

Tabelle 3.1 Transaktionen: Abkürzungen

BOT	Begin of Transaction Timestamp (Startzeitpunkt)
EOT	End of Transaction (Endzeitpunkt)
CO	Commit-Zeitpunkt (Spezialfall EOT)
AB	Abort-Zeitpunkt (Spezialfall EOT)
$r(x)$	Lesen des Datenelementes x
$w(x)$	Schreiben des Datenelementes x
$w(x \mapsto x')$	Schreiben des Datenelementes x

Abbildung 3.1 Transaktionen: Graphische Notation



Zusätzlich besitzen Transaktionen die *ACID-Eigenschaften* [HR83]:

Atomarität (atomicity) Die Atomarität einer Transaktion betrifft zwei wichtige Aspekte. Zum einen Atomarität aus Sicht des Transaktionsprogramms, und zum anderen Atomarität aus Sicht anderer Transaktionen. Atomarität für die Transaktion bedeutet, dass entweder alle ihre Änderungen wirksam werden oder gar keine. Endet eine Transaktion mit einer erfolgreichen Commit-Operation, so müssen alle Änderungen dieser Transaktion wirksam werden. Wird hingegen eine Abort-Operation ausgeführt, so darf keine Änderung in die Datenbank übernommen werden. Für die anderen Transaktionen bedeutet dies: Sobald eine Änderung einer erfolgreich abgeschlossenen Transaktion sichtbar ist, müssen auch alle anderen Änderungen dieser Transaktion sichtbar sein. Damit ist eine Transaktion auf einer logischen Ebene die kleinste, nicht weiter zerteilbare Einheit in der Änderungen an Datensätzen wirksam werden können.

Konsistenz (consistency) Eine Transaktion erzeugt einen konsistenten Zustand der Daten. Oder anders ausgedrückt: „Eine Transaktion überführt die Datenbank von einem konsistenten in einen wiederum konsistenten Datenbank-Zustand“ [HR01]. Während der Ausführung einer Transaktion können selbstverständlich

inkonsistente Zustände auftreten, aber diese sind für nebenläufige Transaktionen aufgrund der Atomarität und der Isolations-Eigenschaft (siehe unten) nicht sichtbar.

Isolation (isolation) Isolation bedeutet, dass jede Transaktion *logisch im Einbenutzerbetrieb* läuft und dass Anomalien (vgl. Abschnitt 3.2), die im unkontrollierten Mehrbenutzerbetrieb auftreten können vermieden werden. Insbesondere muss das Transaktionsprogramm nicht die Seiteneffekte der Nebenläufigkeit behandeln oder vorsehen.

Dauerhaftigkeit (durability) Diese Eigenschaft gewährleistet, dass alle Änderungen einer erfolgreich mit der Commit-Operation abgeschlossenen Transaktion dauerhaft gespeichert werden und alle Fehler und Systemabstürze überleben.

Programme, die im Kontext einer Transaktion ablaufen, werden auch als *Transaktionsprogramme* bezeichnet. Das klassische Beispiel für ein Transaktionsprogramm, ist das Überweisen eines Geldbetrages von einem Konto auf ein anderes. In Listing 3.1 ist ein JDBC-Programm gegeben, welches eine solche Umbuchung vornimmt. Bei JDBC führt die erste Anweisung implizit die BOT-Operation aus. Weiterhin verfügt JDBC über ein Auto-Commit-Feature. Wird dies nicht explizit deaktiviert, so wird nach jeder Anweisung ebenfalls die Commit-Operation vollautomatisch ausgeführt.

3.2 Anomalien des unkontrollierten Mehrbenutzerbetriebes

Unter den *Anomalien des unkontrollierten Mehrbenutzerbetriebes* werden alle Nebenläufigkeitseffekte zusammengefasst, die bei gleichzeitiger Ausführung mehrerer Transaktionen auftreten können. Diese sind auf Synchronisationsfehler zurückzuführen und können bei einer *seriellen* Ausführung aller Transaktionen nicht auftreten. Zur optimalen Ausnutzung der Hardware-Ressourcen sind zu einem gegebenen Zeitpunkt allerdings typischerweise mehrere Transaktionen gleichzeitig aktiv. Die Operationen dieser Transaktionen laufen dann zeitlich verschränkt ab. Werden diese Operationen nicht synchronisiert kann es zu Konflikten kommen und es können dabei unerwünschte Effekte auftreten. Für die zeitliche Abfolge der Operationen von einer oder mehr Transaktionen werden zwei Begriffe unterschieden. Bei laufenden Transaktionen wird *Schedule* verwendet. Sind die Transaktionen bereits beendet so lautet die korrekte Bezeichnung *Historie* . Überlappt die Ausführungsphase einer Transaktion mit der einer anderen, so sind diese beiden Transaktionen nebenläufig.

Verlorengegangene Änderungen (Lost Update) Diese Anomalie ist vergleichsweise schwerwiegend. Sie kann auftreten, wenn zwei Transaktionen ein Datenelement gleichzeitig lesen und auf der Basis dessen, was sie gelesen haben, das

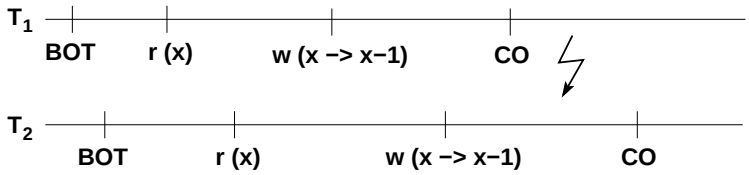
Listing 3.1 JDBC-Transaktionsprogramm: Umbuchung

```
// disable auto Commit operation
connection.setAutoCommit(false);

PreparedStatement readBalance = conection.prepareStatement(
        "SELECT BALANCE " +
        "FROM ACCOUNTS " +
        "WHERE ACCOUNT_NR = ?");
PreparedStatement updateBalance = connection.prepareStatement(
        "UPDATE ACCOUNTS " +
        "SET BALANCE = BALANCE + ? " +
        "WHERE ACCOUNT_NR = ?");

readBalance.setInt(1, fromAcctNr);
updateBalance.setLong(1, amount);
updateBalance.setInt(2, toAcctNr);

// imlicit BOT
ResultSet rs = readBalance.executeQuery();
if(rs.next() {
        balance = readBalance.getLong();
}

if(balance > 0) {
        updateBalance.executeUpdate();
}

// Commit operation
connection.commit;
```

Element schreiben. In Abbildung 3.2 ist eine Historie gegeben, bei der ein Lost Update auftritt. T_1 und T_2 lesen gleichzeitig das Datenelement x. Anschließend subtrahieren beide von dem gelesenen Wert Eins. Aber nur die Änderung von T_2 wird wirksam. Weil T_2 die Subtraktion von T_1 nicht mitbekommen hat überschreibt sie diese.

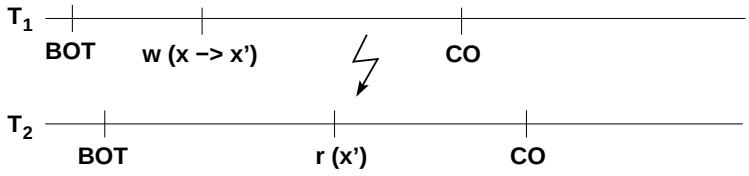
Dirty Read Aufgrund der Atomaritäts-Eigenschaft sollen die Änderungen einer aktiven Transaktion, erst mit ihrem endgültigen Commit sichtbar werden. Im engeren Sinne (nach [BBG+95]) ist das Lesen eines Datenelementes, der Änderungen einer Transaktion enthält, welche noch nicht erfolgreich die Commit-Operation ausgeführt hat, ein Dirty Read. In diesem Fall liegt allerdings nicht zwingend ein Synchronisationsfehler vor und aus Optimierungsgründen könnte man auch fordern, dass ein Dirty Read nur dann vorliegt, wenn die schreibende Transaktion niemals die Commit-Operation erfolgreich abschließt. Dies ist

Abbildung 3.2 Anomalien: Lost Update

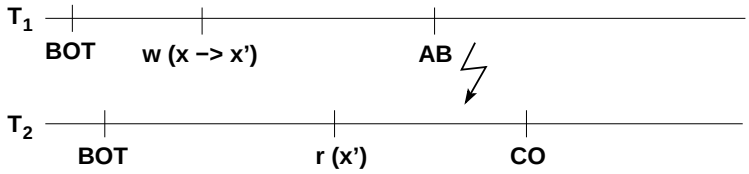


allerdings umstritten (vgl. [BBG+95]). In Abbildung 3.3 (a) ist ein Beispiel für einen Dirty Read im engeren Sinne gegeben und in Abbildung 3.3 (b) ein Dirty Read im weiteren Sinne.

Abbildung 3.3 Anomalien: Dirty Read



(a) Dirty Read im engeren Sinne



(b) Dirty Read im weiteren Sinne

Dirty Write Analog zum Dirty Read liegt ein Dirty Write vor falls eine Transaktion T_1 ein Datenelement schreibt, das bereits Änderungen von einer nebenläufigen, aktiven Transaktion T_2 enthält. Wird das Datenelement vor dem Schreiben von T_1 zuerst gelesen, so spricht man eher von einem Dirty Read. Wird das Element vorher nicht gelesen, sondern von T_1 *blind geschrieben*, liegt nicht zwingend ein Synchronisationsfehler vor. Denn die Änderung ist völlig unabhängig von dem aktuellen Wert des Elementes erfolgt. Jedes Lost Update ist ein Dirty-Write. Er kann aber nur auftreten, falls das Datenelement vorher auch gelesen wurde.

Non-Repeatable Read Der Non-Repeatable Read beschreibt die Situation, dass ein zweifaches Lesen zu unterschiedlichen Ergebnissen führt. Das Zulassen von Non-Repeatable-Reads ist schwerwiegend, weil sich die Programmierer von Transaktionsprogrammen quasi immer implizit darauf verlassen, dass der Wert

eines Datenelementes, den sie zu Beginn einer Transaktion lesen, während der Dauer der Transaktion unverändert bleibt. Es ist beispielsweise gängige Praxis, dass zuerst Daten gelesen werden, um eine Bedingung zu überprüfen, und abhängig davon Operationen auszuführen. Aufgrund dieser impliziten Annahme entsteht beispielsweise auch ein Lost Update.

Phantome Phantome sind eng verwandt mit den Non-Repeatable Reads. In der Tat können sie zu dazu führen, dass das mehrfache Ausführen identischer Leseoperationen unterschiedliche Werte liefert. Man spricht aber von Non-Repeatable Read, falls existierende Datenelemente bei mehrmaligem Lesen unterschiedliche Werte haben. Ein Phantom ist ein Tupel das nebenläufig neu erzeugt oder gelöscht wird. Dann kann es passieren, dass bei der wiederholten Ausführung einer Anfrage die neue Ergebnismenge größer oder kleiner ist als vorher. Insbesondere bei Anfragen mit Aggregat-Funktionen können Phantome aber auch dazu führen, dass unterschiedliche Werte zurückgeliefert werden.

3.3 Korrektheitskriterium der Synchronisation

Offensichtlich können die oben beschriebenen Anomalien nicht auftreten, falls alle Transaktionen seriell ausgeführt werden. Aus dieser Tatsache wird das allgemein anerkannte Korrektheitskriterium für die Synchronisation abgeleitet (vgl. [HR01]). Dieses besagt, dass die Historie einer Menge von gleichzeitig ablaufenden Transaktionen korrekt ist, wenn es mindestens eine *äquivalente*, serielle Historie gibt. Eine Historie, die im Sinne der Synchronisation korrekt ist, bezeichnet man auch als *serialisierbar* (vgl. Theorem 3.3). In welchem Fall die Äquivalenz einer Historie mit

---

Theorem 3.1 Korrektheitskriterium der Synchronisation

Sei H Historie einer Menge von parallel ablaufenden Transaktionen. Eine Historie H ist serialisierbar $\Leftrightarrow$ Es gibt mindestens eine serielle Historie S, die äquivalent ist zu H. (vgl. [HR01])

---

einer Seriellen vorliegt, kann formalisiert werden. An dieser Stelle findet allerdings eine Beschränkung auf eine anschauliche Variante statt, da sie für das weitere Verständnis im Rahmen dieser Arbeit ausreichend ist. Für die Äquivalenz mit einer seriellen Historie müssen zwei Bedingungen erfüllt sein. Zum einen muss jede Leseoperation der Historie die gleichen Ausgabewerte sehen, wie sie sie in der seriellen Historie sehen würde. Und zweitens muss nach Ausführen der Historie der gleiche Datenbank-Zustand vorliegen, wie nach dem seriellen Ausführen der Transaktionen (vgl. Theorem 3.3).

Theorem 3.2 Äquivalenz einer Historie zu einer seriellen Historie

Eine Historie H ist äquivalent zu einer seriellen Historie S, falls

1. jede Transaktion der Historie H sieht, für jede Leseoperation die gleichen Ausgabewerte, wie in der seriellen Historie S
2. Die Historie H erzeugt den gleichen Datenbank-Endzustand, wie S

Das Sicherstellen von serialisierbaren Historien ist für den verteilten Fall noch deutlich schwieriger. Denn die Transaktionen laufen dann auf unterschiedlichen Knoten und der Mangel an zentralisiertem Wissen, bzw. die Nachrichten-Latenz erschweren die Einhaltung der Serialisierbarkeit. *1-Copy-Serializability* beschreibt die Eigenschaft eines verteilten Systems serialisierbare Historien, wie bei einem nichtreplizierten Datenbanksystem, zu garantieren. Dies ist aber noch nicht zufriedenstellend gelöst, das heißt bei einem Replikationssystem ist immer mit einem niedrigeren Isolationsgrad zu rechnen, als bei einem Einzelknotensystem.

3.4 Synchronisationsverfahren

In der Forschung und auch in der Industrie wurden bereits viele Synchronisationsverfahren entwickelt, welche die Serialisierbarkeit ihrer Historien sicherstellen. Die eigentliche Kunst dabei ist, dass das Verfahren neben den nicht-serialisierbaren Historien, nicht gleichzeitig zu viele serialisierbare Historien ausschließt (vgl. [BBG+95]). Betrachtet man das Problem von der theoretischen Seite, dann gibt es zu n Transaktionen immerhin $n!$ serielle Ausführungsreihenfolgen (siehe [HR01]).

Es kann durchaus auch Anwendungsfälle geben, bei denen die Serialisierbarkeit nicht gegeben sein muss, und Anomalien in Kauf genommen werden können. Die Wahl eines geeigneten Synchronisationsverfahrens ist ein Optimierungsproblem und es muss ein sinnvoller Kompromiss zwischen dem Isolationsgrad und dem der Nebenläufigkeit gefunden werden.

3.4.1 Synchronisation durch Sperrverfahren

Viele Synchronisationsverfahren beruhen auf dem Setzen geeigneter Sperren auf den Datenobjekten. Der Zugriff auf ein Objekt kann nur erfolgen, falls vorher eine geeignete Sperre gesetzt werden konnte. Bei den Sperren werden unterschiedliche Sperrmodi unterschieden. Je nach Verträglichkeit der Sperrmodi können gleichzeitig mehrere Sperren auf einem Datenobjekt gesetzt sein. Ein Datenelement kann beispielsweise

generell von mehreren Transaktionen gelesen werden, wenn gleichzeitig keine Schreiboperation stattfindet.

Das einfachste Verfahren ist das sogenannte *RX-Sperrverfahren*. Dieses unterscheidet nur zwei Sperrmodi: Die exklusive Sperre (X-Sperre oder Schreibsperre) und die Lesesperre (R-Sperre). Ist ein Datenelement mit einer X-Sperre belegt können keine anderen Sperren gesetzt werden. Bei gesetzter R-Sperre können weitere R-Sperren von konkurrierenden Lesern gesetzt werden, allerdings keine X-Sperre.

Die Dauer in der eine Sperre gehalten wird und die Art und Weise, wie die Sperren angefordert und freigegeben werden, ist ausschlaggebend für den Isolationsgrad. Dies wirkt sich aber selbstverständlich auch auf den Grad der Nebenläufigkeit und den Transaktionsdurchsatz aus. Je länger eine Sperre gesetzt bleibt, umso länger werden konkurrierende Transaktionen blockiert. Je stärker Transaktionen blockiert werden, umso höher wird der *MPL* (multi-programming level = Anzahl aktiver Transaktionen). Mit steigendem MPL steigt wiederum die Konfliktrate, was wiederum zu verstärkten Blockierungen führt, usw.

Es kann gezeigt werden, dass durch das Anfordern der Sperren in einer ersten Phase, und das Freigeben der Sperren in einer zweiten Phase, die resultierende Historie *konfliktserialisierbar* ist. Konfliktserialisierbarkeit ist annähernd so gut wie Serialisierbarkeit. Das Auftreten von Phantomen kann hier allerdings nicht so ohne weiteres ausgeschlossen werden. Bei der Verwendung von Sperrverfahren kann es außerdem zu Deadlocks und kaskadierendem Rücksetzen kommen. Weitergehende Informationen hierzu finden sich in [HR01].

3.5 Isolationsgrade nach ANSI-SQL

Der SQL92-Standard [Ame92] beschreibt vier *Isolationsgrade* (Isolation Levels): *Read Uncommited, Read Commited, Repeatable Read* und *Serializable*. Die Definition dieser Konsistenzstufen sollte dabei unabhängig von konkreten Implementierungen erfolgen. Deshalb wurden die Konsistenzstufen auf der Basis von Anomalien definiert, welche bei der jeweiligen Stufen nicht ausgeschlossen werden können. Tabelle 3.2 enthält für jeden Grad eine Zeile. In den entsprechenden Spalten ist dann markiert, welche Anomalien bei der Konsistenzstufe noch auftreten können.

Der Standard schließt dabei das Auftreten von Lost Updates für jede seiner vier Konsistenzstufen generell aus. Die Definition von ANSI ist nicht unumstritten. Die Anomalien, die von ANSI zugrunde gelegt werden sind beispielsweise nicht allgemein genug definiert. Die Anomalie Dirty Write ist in dem Standard überhaupt nicht enthalten. Außerdem ist die eigentliche Menge der Isolation Levels unvollständig, und die Definition der Konsistenzstufen nicht eindeutig. Näheres findet man in [BBG$^+$95]).

Tabelle 3.2 ANSI SQL Isolation Levels (entnommen aus [HR01])

Konsistenzstufe	**Anomalie**		
	Dirty Read	Non-Repeatable Read	Phantome
Read Uncommited	+	+	+
Read Commited	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

Berenson et. al. führen in diesem Papier ein weiteres für diese Arbeit wichtiges Isolation Level ein. Die sogenannte *Snapshot Isolation* wird ausführlich in Kapitel 5.4 behandelt.

Für diese Arbeit sind insbesondere die Konsistenzstufen Repeatable Read und Serializable von Bedeutung. Repeatable Read ist das Minimum was an Isolation realisiert werden muss. Isolation Levels unterhalb von Repeatable Read werden von ihrem Isolationsgrad als zu schwach angesehen.

3.6 Zusammenfassung

In diesem Kapitel wurden grundlegende Begriffe und Konzepte der Transaktionsverwaltung vorgestellt. Dabei wurde eine Beschränkung auf diejenigen Bereiche dieses umfangreichen Themenkomplexes vorgenommen, die für den weiteren Verlauf dieser Arbeit wichtig sind. Insbesondere das Auftreten von Anomalien muss von jedem Synchronisationsverfahren bestmöglich unterbunden werden. Die Begriffe Konsistenzstufe oder Isolationsgrad werden im Laufe dieser Arbeit noch häufig verwendet werden.

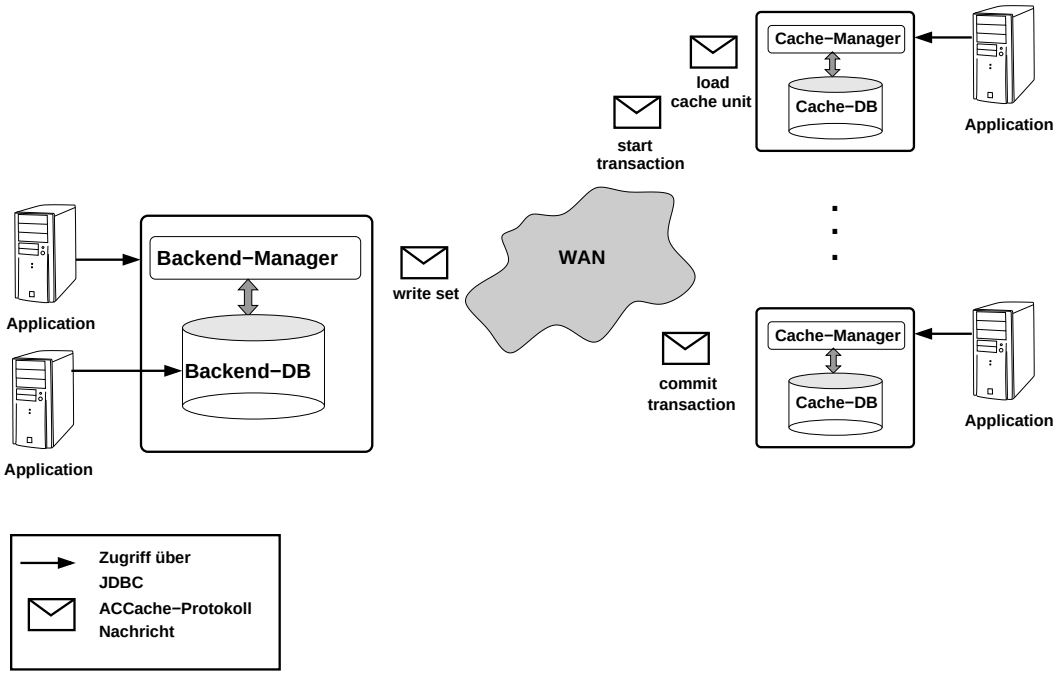
4 Anforderungen und Kernprobleme

In diesem Kapitel werden zunächst die Anforderungen an das zu erstellende Synchronisationsverfahren herausgearbeitet. Die verschiedenen Möglichkeiten und Freiheitsgrade, die bei der Konzeption in Frage kommen, können in der Arbeit stets anhand dieser Anforderungen bewertet werden. Anschließend werden die zentralen Probleme, die sich bei der Umsetzung des Verfahrens ergeben, aufgezeigt. Für die Darstellung der Anforderungen und Kernprobleme, bedarf es allerdings zunächst einer exakten Beschreibung der Ausgangssituation.

4.1 Ausgangssituation

In Abbildung 4.1 ist die Gesamtarchitektur eines Constraint-basierten Datenbank-Caching-Systems dargestellt. Der Ausgangspunkt aller Überlegungen sind die Daten einer großen, zentralen Datenbank, der *Backend-Datenbank*. Zu der Backend-Datenbank kann es eine beliebige Zahl an Cache-Datenbanken geben. Diese können weltweit verteilt sein. Jede dieser Cache-Datenbanken speichert eine Teilmenge der Backend-Daten als Kopien. Eine Cache-Datenbank wird vollständig von einem Cache-Manager kontrolliert, der exklusiven Zugang zu seiner Datenbank besitzt. Jeder Cache-Manager verwaltet jeweils seine eigene Cache Group, sowie eine Menge von Anwendungstransaktionen. Anwendungstransaktionen des Caching-Systems werden im Folgenden auch als *ACCachetransaktionen* bezeichnet. Das Laden- und Entladen von Daten in einen Cache ist ein vollkommen dynamischer Prozess. Die Menge der Daten, die sich zu einem bestimmten Zeitpunkt in einer Cache-Datenbank befindet, wird ausschließlich von der Cache Group und dem Zugriffsmuster der aktuellen Anwendungstransaktionen bestimmt. Der Backend-Manager hat Zugriff auf die Backend-Datenbank. Aber außer ihm kann es noch andere Benutzer geben, die völlig unabhängig vom Caching-System auf die Backend-Datenbank zugreifen. Backend-Manager und Cache-Manager kommunizieren über ein Protokoll. ACCache-Transaktionen können zu beliebigen Zeitpunkten auf den Caches und dem Backend starten und müssen synchronisiert werden.

Abbildung 4.1 Big Picture



4.2 Anforderungen

Im Folgenden soll näher auf die Anforderungen, die an das Synchronisationsverfahren gestellt werden, eingegangen werden. Dabei handelt es sich um das erste, speziell auf das Constraint-basierte Datenbank-Caching zugeschnittene Synchronisationsverfahren überhaupt. Diese Arbeit stellt somit auch einen Proof-Of-Concept dar und der entwickelte Algorithmus ist mit Sicherheit noch verbesserungswürdig. Nicht alle Anforderungen, konnten im Rahmen dieser Arbeit umgesetzt werden.

4.2.1 Transparenz (Replikationstransparenz)

Eine elementare Forderung ist Transparenz für die Anwender und Anwendungsprogramme. Das Implementieren und Warten von Software ist teuer. Aus diesem Grund wäre eine Anpassung bestehender Software an Besonderheiten des Datenbank-Caching nicht erwünscht. Das Programmieren von Transaktionsprogrammen muss über die JDBC-API möglich sein. Damit muss lediglich der JDBC-Treiber des Caching-Systems, anstatt des ursprünglichen Datenbanksystem-Treiber, geladen werden. Jeder Zugriff auf die Daten gestaltet sich wie ein Datenbank-Zugriff über JDBC.

Für das Anwendungsprogramm ist es völlig transparent, ob die Daten aus der Cache-Datenbank stammen oder aus der Backend-Datenbank. Weiterhin soll die Anwendung nicht zwischen rein lesenden und schreibenden Transaktionen unterscheiden müssen, indem sie dies beispielsweise beim Start einer Transaktion deklariert.

4.2.2 ACID-Transaktionen

Alle Transaktionen, die über das Cache-System abgewickelt werden, müssen aus Sicht der Anwendung den ACID-Eigenschaften (vgl. Abschnitt 3.1) genügen. Im Folgenden sollen die ACID-Eigenschaften noch einmal im Hinblick auf das Constraint-basierte Datenbank-Caching betrachtet werden.

Atomarität Atomarität bedeutet zum Einen, dass entweder alle Änderungen einer Transaktion wirksam werden, oder gar keine. Zum Anderen darf eine Transaktion immer nur transaktionskonsistente Zustände sehen. Das heißt, sobald die Änderung einer anderen Transaktion sichtbar ist, müssen alle Änderungen dieser Transaktion sichtbar sein. Im Hinblick auf das Constraint-basierte Datenbank-Caching bedeutet dies allerdings nicht, dass die Änderungen einer Transaktion tatsächlich synchron und atomar in allen beteiligten Datenbanken eingebracht werden müssen. Es muss lediglich sichergestellt werden, dass sobald eine Anwendungtransaktion eine Änderung einer anderen Transaktion sieht, alle Änderungen dieser Transaktion für die Anwendungstransaktion sichtbar sind. Bei Systemfehlern oder Abstürzen von beteiligten Knoten muss sichergestellt sein, dass alle Änderungen einer Transaktion wirksam werden oder keine.

Konsistenz Jede Transaktion erzeugt einen konsistenten Datenbankzustand. Während der Ausführung einer Transaktion können selbstverständlich inkonsistente Zustände entstehen, sind aber für nebenläufige Transaktionen, aufgrund der Isolations-Eigenschaft nicht sichtbar. Überträgt man dies auf das Constraint-basierte Datenbank-Caching, dann genügt es zu gewährleisten, dass jede Anwendungstransaktion stets einen transaktionskonsistenten Zustand sieht. Man kann sogar fordern, dass jede Transaktion, den bei ihrem BOT jüngsten transaktionskonsistenten Zustand sieht. Es muss nicht sichergestellt sein, dass jede Cache-Datenbank zu jedem Zeitpunkt in einem transaktionskonsistenten Zustand ist. Wohl aber die Backend-Datenbank, da auf ihr auch andere Benutzer als das ACCache-System zugelassen sind.

Isolation Isolation bedeutet, dass jede Transaktion logisch im Einbenutzerbetrieb läuft und dass Anomalien (vgl. Abschnitt 3.2), die im unkontrollierten Mehrbenutzerbetrieb auftreten können, unterbunden werden. Das Entwickeln eines Synchronisationsverfahrens, das eine geeignete Isolierung für ACCachetransaktionen sicherstellt, ist das eigentliche Thema dieser Arbeit. Die bestehenden Kernprobleme werden im nächsten Kapitel vertieft betrachtet.

Dauerhaftigkeit Diese Eigenschaft gewährleistet, dass alle Änderungen einer erfolgreich abgeschlossene Transaktion dauerhaft gespeichert werden und alle Fehler und Systemabstürze überleben. Dies muss für jeden beteiligten Knoten des Caching-Systems erfüllt sein. Von besonderer Bedeutung ist hierbei, dass mit zunehmender Anzahl an Caches, die Wahrscheinlichkeit dafür, dass ein Knoten ausfällt, steigt.

4.2.3 1-Copy Serializability

1-Copy-Serializability (vgl. 3.3) ist das Ziel für ein Replikationssystem schlechthin. Wenn 1-Copy-Serializability erreicht werden könnte, so wäre sichergestellt dass die Historien aller Transaktionen des verteilten Systems serialisierbar sind. Somit wäre das verteilte System im Hinblick auf Isolation und Synchronisation gleichwertig mit einem nicht-replizierten Datenbanksystem. Der Anwendungsprogrammierer wäre damit vor allen Anomalien des unkontrollierten Mehrbenutzerbetriebes geschützt (vgl. Kapitel 3.2). Er könnte seine Konzentration vollständig auf die eigentliche Programmlogik lenken. Somit ist 1-Copy-Serializability ein sehr erstrebenswertes Ziel für jede Replikationslösung. Sie ist prinzipiell auch im verteilten Fall erreichbar. Allerdings führt sie in der Praxis zu großen Problemen. Dies ist beispielsweise darauf zurückzuführen, dass die Konfliktrate bei einem replizierten System um ein Vielfaches größer ist, als bei einem Einzelknotensystem. Hinzu kommt der enorme Kommunikationsaufwand zwischen den einzelnen Replikaten.

4.2.4 Skalierbarkeit

Skalierbarkeit muss in mindestens zwei Dimensionen gegeben sein:

Größe Das System soll in der Anzahl der Cache-Knoten skalieren. Das dynamische Hinzunehmen eines Cache muss jederzeit möglich sein, wenn es die Anwendung erfordert. Mit der Hinzunahme eines Cache steigt immer auch die Anzahl an Transaktionen innerhalb des Systems, was im Allgemeinen eine höhere Konfliktrate zur Folge hat. Skalierbarkeit in der Größe des Systems ist also keine triviale Anforderung.

Geografie Weil die Caches weltweit verteilt sein dürfen, muss das System in der Geografie skalieren. Ein Datenbank-Caching-System ist nicht zu verwechseln mit einem Datenbank-Cluster. Die Knoten eines Cluster befinden sich im allgemeinen im selben LAN. Zwischen den Knoten eines Clusters kann eine schnelle Verbindung installiert werden, sodass sehr hohe Zugriffsraten möglich sind. Zwischen einem Datenbanken-Cache und dem Backend liegt aber ein WAN oder das Internet. Hier ist mit deutlich höheren Latenzzeiten zu rechnen als im LAN (vgl. Kapitel 2). Beim Internet kommen immer auch noch einige Unsicherheitsfaktoren hinzu. Wichtige Internetknoten können ausfallen, und die

RT-Laufzeiten fallen dann dementsprechend ab. Latenz ist ein wichtiger Punkt beim Datenbank-Caching und muss bei der Konzeption des Synchronisationsverfahrens beachtet werden.

4.2.5 Effizienz

Weiterhin soll das Verfahren möglichst effizient und performant arbeiten. Es ist von enormer Wichtigkeit, dass das Verfahren die ACCachetransaktionen so wenig wie möglich blockiert. Verminderte Blockierungen führen dazu, dass Ressourcen früh wieder freigegeben werden können, und die Konfliktrate minimiert wird.

4.2.6 Middleware-Architektur

Der existierende Datenbank-Cache-Prototyp ist als Middleware in der Programmiersprache Java implementiert. Folglich muss auch das zu entwickelnde Verfahren als Middleware konzipiert werden. Weil diese Middleware keine proprietäre Erweiterung eines existierenden Datenbanksystems ist, kann der Prototyp für ein beliebiges Datenbanksystem als Cache eingesetzt werden. Es sind sogar heterogene Systemlandschaften vorstellbar. Ein beliebiges relationales Altsystem beispielsweise könnte durch den Einsatz von Caches entlastet werden, wobei gleichzeitig bei den Caches völlig neue Datenbanksysteme eingesetzt werden könnten. Dieses Szenario ist durchaus realistisch, denn das Design und Rollout einer kompletten Systemlandschaft ist selten von Grund auf möglich. Vielmehr müssen bestehende Altsysteme mit neuen Technologien integriert werden.

4.3 Kernprobleme

Betrachten wir nun die Probleme, die beim Replizieren von Daten entstehen. Auf die hier beschriebenen Herausforderungen wird im Laufe dieser Arbeit immer wieder direkt oder indirekt eingegangen. Sie sind wichtige Grundlage für die Konzeption des entwickelten Verfahrens.

4.3.1 Konfliktrate

Der immense Anstieg der Konfliktrate bei einem replizierten System stellt wohl die größte Herausforderung dar. In Kapitel 5.2 wird gezeigt, dass grundsätzlich mit einem quadratischen Anstieg der Konfliktrate in Abhängigkeit der Zahl der Knoten des Replikationssystems zu rechnen ist.

4.3.2 Verteiltheit der Daten

Da es sich um ein verteiltes System handelt, gibt es einen Mangel an globalem und zentralisiertem Wissen. Dies kommt umso mehr zum Tragen, da es sich in diesem Fall um ein Transaktionssystem handelt. Wenn eine ACCachetransaktion in den Zustand commited wechseln soll, dann muss zwischen mindestens einer globalen Instanz und einem Replikatknoten ein Konsens über diesen Zustand herrschen. Erstrebenswert ist in jedem Fall auch, dass jeder andere Knoten zeitnah über den Ausgang der Transaktion erfährt. Dies erfordert den Einsatz eines verbindungsorientierten Protokolls wie TCP. Verbindungsorientierte Protokolle verfügen über eine Sicherungsschicht, die garantiert, dass versendete Nachrichten beim Empfänger angekommen sind. Dies ist selbstverständlich teuer und bedeutet zusätzliche Latenz.

4.3.3 Zugriffslücke „Netzwerk“

Da es sich bei der Architektur des ACCache-Systems um Datenbank-Caches handelt, die über eine Netzwerkverbindung mit einer Backend-Datenbank verbunden sind, liegt es nahe eine Analogie zu ziehen zwischen der Hauptspeicher – Externspeicher Zugriffslücke und der “Netzwerk-Zugriffslücke” zu ziehen. Der Zugriff auf die Daten aus der Cache-Datenbank erfolgt deutlich schneller als ein Zugriff auf die Backend-Datenbank. Immer wenn der Cache Anfragen ans Backend weiterleiten muss, führt dies zu einer Verlängerung der Transaktion. Dies hat zur Folge, dass die Transaktion länger Sperren halten muss, Konflikte potentiell erhöht werden. Von daher ist es insbesondere aus Sicht der Transaktionsverwaltung erstrebenswert, dass so viele Anfragen wie möglich von der Cache-Datenbankbeantwortet werden können.

Der Vergleich mit dem Externspeicher hinkt jedoch ein wenig. Bei mechanischen Platten gibt es eine natürliche Grenze für die Zugriffsrate. Davon ist bei den Netzwerktechnologien nicht unbedingt auszugehen. Hier liegt die Obergrenze für die Signalübertragung aufgrund der Glasfaserkabeltechnologie bei Lichtgeschwindigkeit. Momentan sieht es nicht so aus, als ob das Breitbandgeschäft bereits an seine Grenzen gestoßen wäre. Auch beim Internet ist mit weiteren Steigerungen der Datenraten zu rechnen. Kann man sich eine schnelle Anbindung erlauben, so hat man eine akzeptable Datenrate und entsprechend geringe Latenz.

4.3.4 Middleware-Architektur

Die Middleware-Architektur bringt einige Vorteile mit sich, gleichzeit führt sie jedoch zu größeren Komplikationen bei der Umsetzung. An verschiedenen Stellen wäre der Zugriff auf die internen Daten des zugrundeliegenden Datenbanksystems nützlich. Es kann aber nicht vorausgesetzt werden, dass ein Zugriff auf diese möglich ist. Weiterhin isoliert das zugrundeliegende Datenbanksystem alle Transaktionen lokal mit

einem eigenen Synchronisationsverfahren, und damit auch alle Transaktionen des ACCache-Systems selbst. Dies muss insbesondere berücksichtigt werden, da sonst Konflikte mit der globalen Isolierung und der Isolierung des Datenbanksystems auftreten können.

4.4 Zusammenfassung

In diesem Kapitel wurden zunächst die Anforderungen an ein zu erstellendes Synchronisationsverfahren festgelegt. Dabei stellt die Isolations-Eigenschaft des ACID-Paradigmas, bzw. die 1-Copy-Serializability die größte Herausforderung dar. Aber auch das Sicherstellen von Skalierbarkeit ist keine triviale Anforderung. Aus diesem Grund sollte das Verfahren möglichst effizient arbeiten, da sich sonst die bestehenden Kernprobleme nicht bewältigen lassen. Hier ist insbesondere die hohe Konfliktrate bei einem replizierten System zu nennen. Diese wird auch im nächsten Kapitel noch eingehend untersucht.

5 Replizierte Daten und Nebenläufigkeit

Für die Synchronisation von nebenläufigen Transaktionen, die auf einem einzigen System ablaufen, wurden sehr performante Verfahren entwickelt. Auch für den Fall, dass die Daten über mehrere Knoten repliziert werden, gibt es bereits sehr viele Ansätze und Verfahren. Die meisten dieser Verfahren nehmen allerdings in Kauf, dass die replizierten Daten über kurze Zeitspannen hinweg nicht aktuell sind. Dies ist darauf zurückzuführen, dass Änderungen erfolgreich abgeschlossener Transaktionen nicht auf allen Replikaten mit dem Commit der Transaktion wirksam werden. Die dadurch entstehenden veralteten Daten werden oft auch als *stale data* bezeichnet.

Die Inkaufnahme von stale data hat mehrere Gründe: Zum einen ist der Kommunikationsaufwand zwischen den Replikaten zu nennen, den das synchrone Durchführen von Änderungsoperationen mit sich bringt. Schwerwiegender ist aber wohl der Zeitaufwand, der benötigt wird, um die Änderungsoperationen auf allen Replikaten synchron durchzuführen. Das endgültige Commit einer solchen globalen Transaktion muss hinausgezögert werden, bis die Schreiboperationen auf allen Knoten des replizierten Systems ausgeführt wurden. Dies führt zu einer Verlängerung der schreibenden Transaktion und dazu, dass Sperren und Ressourcen erst verspätet freigegeben werden können. Aufgrund der dadurch enstehenden zusätzlichen Blockierungen steigt die Konfliktrate signifikant. Im Folgenden sollen die existierenden Ansätze und Verfahren für den verteilten Fall etwas genauer vorgestellt und untersucht werden.

5.1 Replikationstechniken

Ein Replikat enthält die vollständige Kopie aller Daten einer Datenbank. Es gibt verschiedene Replikationstechniken. Diese lassen sich jedoch anhand von zwei Dimensionen klassifizieren. Zum einen unterscheidet man, ob das Propagieren von Änderungen synchron oder asynchron erfolgt:

Eager Replication Hier werden die Änderungen einer schreibenden Transaktion synchron auf allen Replikaten wirksam. Hierdurch gibt es niemals stale data.

Lazy Replication Bei dieser Strategie werden die Änderungen einer erfolgreich abgeschlossen Transaktion asynchron auf allen anderen Replikaten nachgezogen. Nach dem Commit einer Transaktion gibt es ein kleines Zeitfenster δ in dem die geänderten Daten auf den anderen Replikaten veraltet sind.

Das Propagieren von Änderungen kann auf unterschiedliche Arten erfolgen. Sie könnte zum einen auf einer logischen Ebene durch das Versenden der Änderungsoperationen geschehen. Oder auf einer physischen durch das Versenden der aktuellen Werte eines geänderten Datensatzes. Letzteres erfordert die Extraktion des sogenannten *Write Sets* einer Transaktion. Das Write Set speichert alle geänderten Datensätze. Das Anwenden des Write Sets einer Transaktion muss den ACID-Eigenschaften genügen. Dies bedeutet insbesondere, dass alle Änderungen eines Write Sets atomar auf einem Replikat eingebracht werden müssen.

Replikationstechniken unterscheidet man weiterhin danach, welche Knoten bestimmte Datenobjekte schreiben dürfen:

Master Replication Jedes Datenobjekt hat eine *Master*-Datenbank. Änderungen auf diesem Datensatz müssen auf dem Master-Knoten erfolgen und werden somit auf diesem synchronisiert. Der Master propagiert die Änderungen auf diesem Datensatz an alle Nicht-Master-Replikate. Eine alternative Bezeichnung für diese Vorgehensweise ist *Primary-Copy-Verfahren*. Der Master wird dann als *Primärkopie* bezeichnet.

Group Replication Jeder Knoten darf alle seine Datensätze schreiben. Jeder Knoten muss alle seine Änderungen an alle anderen Replikate propagieren. Für diese Strategie wird auch das Synonym *Update everywhere* verwendet.

Group und Master Replication lassen sich jeweils mit Eager und Lazy Replication kombinieren.

5.2 Inhärente Probleme bei der Replikation von Daten

Im Folgenden sollen die Kernprobleme, die beim Replizieren von Daten entstehen, aufgezeigt werden. Die Erkenntnisse dieses Abschnitts stützen sich weitgehend auf eine Analyse von Grey et. al. [GHOS96]. Die der Datenreplikation innewohnende Problematik ist darauf zurückzuführen, dass mit steigender Anzahl von Replikaten, die Konfliktrate um eine Vielfaches stärker wächst. Je mehr Replikate es gibt, um so mehr Anwendungstransaktionen gibt es gleichzeitig. Die Größe der Datenbank wächst allerdings im Allgemeinen nicht, und wenn überhaupt dann nicht in der gleichen Größenordnung wie die Anzahl der Anwendungstransaktionen. Statistisch gesehen steigt die Konfliktrate, weil viel mehr Transaktionen gleichzeitig auf die gleiche Menge von Daten zugreifen.

5.2.1 Abschätzung der Konfliktrate

Grey et. al. haben in ihrem Papier einige einfache Formeln zur Abschätzung der Konfliktrate in Abhängigkeit

- von der Anzahl der Replikate und
- von der Transaktionsgröße

aufgestellt.

Zum besseren Verständnis der Analyse ist es erforderlich den Begriff der Anwendungstransaktionen von dem Begriff der Transaktion allgemein zu unterscheiden (vgl. Tabelle 5.1). Eine Anwendungstransaktion ist eine Transaktion, die von einer Anwendung der Daten gestartet wurde. In Abgrenzung dazu, kann es auch Transaktionen geben, die vom Replikationssystem gestartet werden. Das Einbringen des Write Sets einer Anwendungstransaktion auf einem Replikat muss beispielsweise den ACID-Eigenschaften genügen. Deshalb wird das Anwenden eines Write Set typischerweise in einer eigenen Transaktion stattfinden. Alle Variablen, die in die Analyse einge-

Tabelle 5.1 Abgrenzung der Begriffe Anwendungstransaktion und Transaktion

Anwendungstransaktion	Transaktion einer Anwendung des Replikationssystems
Transaktion	Transaktion allgemein. Anwendungstransaktionen und Transaktionen, die vom Replikationssystem gestartet werden

hen sind in Tabelle 5.2 aufgeschlüsselt. In dem Modell, das Grey et. al. ihrer Analyse zugrundelegen, ist der Zugriff auf die Daten gleichverteilt. Das heißt auf jedes Datenelement wird mit der gleichen Wahrscheinlichkeit $\frac{1}{db\_size}$ zugegriffen. Diese Annahme ist in der Realität im Allgemeinen nicht gegeben. Das Modell ist aber ausreichend, um die Größenordnung der Konfliktrate abzuschätzen.

Tabelle 5.2 Variablen bei der Abschätzung der Konfliktrate

nodes	Anzahl Knoten
transactions	Anzahl Transaktionen
tps	Anzahl gestarteter Anwendungstransaktionen pro Sekunde
actions	Anzahl Update-Operationen einer Transaktion
action_time	Zeit, die für eine Update-Operation benötigt wird
db_size	Größe der Datenbank

Abschätzung der Konfliktrate bei einem einzelnen Knoten

Zunächst soll die Rate für Schreibkonflikte abgeschätzt werden, wenn das System aus einem einzelnen Knoten besteht. Also für den Fall, dass noch keine Replikate vorhanden sind. Bei einer Rate von *tps* gestarteten Anwendungstransaktionen pro Sekunde, beläuft sich die Gesamtanzahl der Transaktionen pro Zeitpunkt auf:

$$transactions = tps \cdot actions \cdot action\_time \tag{5.1}$$

(vgl. [GHOS96]).

Im Durchschnitt sind zu einem Zeitpunkt alle Transaktion halb fertig. Dann haben diese insgesamt $\frac{transactions \cdot actions}{2}$ Datenobjekte geändert. Die Wahrscheinlichkeit dafür, dass eine Schreiboperation nun ein Datenelement ändert, der bereits geschrieben wurde, liegt dann bei $\frac{transactions \cdot actions}{2} \cdot \frac{1}{db\_size}$. Die Wahrscheinlichkeit $P(transaction)$, dass bei einer Transaktion ein Schreibkonflikt auftritt, ist gleich der Wahrscheinlichkeit dafür, dass die Transaktion mindestens ein Datenelement schreibt, der bereits von einer anderen Transaktion geschrieben wurde. Diese beträgt ungefähr:

$$\begin{aligned} P(transaction) &= 1 - (1 - \frac{transactions \cdot actions}{2 \cdot db\_size})^{actions} \\ &\approx \frac{transactions \cdot actions^2}{2 \cdot db\_size} \end{aligned} \tag{5.2}$$

(vgl. [GHOS96]).

Gleichung 5.1 eingesetzt in Gleichung 5.2 ergibt:

$$P(transaction) \approx \frac{tps \cdot action\_time \cdot actions^3}{2 \cdot db\_size} \tag{5.3}$$

In dem Modell ist das Auftreten eines Konflikts gleichwahrscheinlich für jede Schreiboperation. Weiterhin benötigt jede Operation im Mittel $action\_time$ Sekunden. Daraus folgt für die Konfliktrate $CR(transaction))$ einer Transaktion:

$$CR(transaction) \approx \frac{tps \cdot action\_time \cdot actions^3}{2 \cdot db\_size \cdot actions \cdot action\_time} = \frac{tps \cdot actions^2}{2 \cdot db\_size} \tag{5.4}$$

Auf dem Knoten laufen aber *transactions* Anwendungstransaktionen gleichzeitig und deshalb liegt die Wahrscheinlichkeit für das Auftreten eines Konfliktes in der nächsten Sekunde insgesamt bei:

$$CR(system) \approx \frac{tps \cdot actions^2}{2 \cdot db\_size} \cdot transactions = \frac{tps^2 \cdot actions^3 \cdot action\_time}{2 \cdot db\_size} \tag{5.5}$$

Abschätzung der Konfliktrate bei Eager Replication

Nun soll die Konfliktrate $CR_{eager}(system)$ für ein Replikationssystem, das Eager Replication zum Propagieren von Änderungen einsetzt, abgeschätzt werden. Bei *nodes* Knoten des Replikationssystems ergibt sich ein globaler Anstieg von *tps* auf *nodes*·*tps* Anwendungstransaktionen pro Sekunde:

$$tps\_eager = nodes \cdot tps \tag{5.6}$$

Bei Eager Replication wird jede Update-Operation synchron und spätestens bei Commit, zu allen anderen Replikaten propagiert. Dadurch vergrößert sich für jede Schreiboperation die Ausführungszeit *action_time*, um die Zeit die für das synchrone Propagieren der Änderung benötigt wird. Grey et. al. vergrößern hier um den Faktor *nodes* (vgl. [GHOS96]). Bei dieser Untersuchung wird allerdings nicht davon ausgegangen, dass sich *action_time* um den Faktor *nodes* vergrößert, weil das Propagieren der Änderungen auf die Replikate parallel erfolgen kann. Stattdessen erhöht sich *action_time* um die durchschnittliche Dauer *propagation_time*, die zum Propagieren einer Änderung benötigt wird. Die Gesamtanzahl der Anwendungstransaktionen pro Zeitpunkt erhöht sich dann auf $nodes \cdot tps \cdot (propagation\_time + action\_time) \cdot actions$:

$$transactions\_eager = nodes \cdot tps \cdot (propagation\_time + action\_time) \cdot actions. \tag{5.7}$$

Analog zu Gleichung 5.3 beträgt die Wahrscheinlichkeit für einen Schreibkonflikt bei einer Transaktion:

$$P_{eager}(transaction) \approx \frac{nodes \cdot tps \cdot (propagation\_time + action\_time) \cdot actions^3}{2 \cdot db\_size}. \tag{5.8}$$

Entsprechend ergibt sich die Wahrscheinlichkeit dafür, dass die Transaktion in der nächsten Sekunde ein Datenelement schreibt, der bereits von einer nebenläufigen Transaktion geschrieben wurde, durch die Division von $actions \cdot (propagation\_time + action\_time)$:

$$CR_{eager}(transaction) \approx \frac{nodes \cdot tps \cdot actions^2}{2 \cdot db\_size}. \tag{5.9}$$

Nun gibt es zu jedem Zeitpunkt aber insgesamt *transactions_eager* Anwendungstransaktionen. Damit gilt für die Gesamtkonfliktrate $CR_{eager}(system)$:

$$
\begin{aligned}
CR_{eager}(system) &\approx \frac{nodes \cdot tps \cdot actions^2}{2 \cdot db\_size} \cdot transactions\_eager \\
&= \frac{nodes^2 \cdot tps^2 \cdot (action\_time + propagation\_time) \cdot actions^3}{2 \cdot db\_size} \\
&= (1 + \frac{propagation\_time}{action\_time}) \cdot \frac{nodes^2 \cdot tps^2 \cdot action\_time \cdot actions^3}{2 \cdot db\_size}.
\end{aligned}
$$

Daraus folgt mit Gleichung 5.5:

$$
\begin{aligned}
CR_{eager}(system) &= (1 + \frac{propagation\_time}{action\_time}) \cdot nodes^2 \cdot CR(system) \\
&= c \cdot nodes^2 \cdot CR(system), \quad c \geq 1.
\end{aligned}
\tag{5.10}
$$

An Gleichung 5.10 wird die, der Eager Replication innewohnende Problematik, ersichtlich: Bei einem Wachstum der Knotenanzahl um den Faktor n steigt die Konfliktrate um mindestens n^2. Hinzu kommt noch der Faktor $c = 1 + \frac{propagation\_time}{action\_time}$. Ist $propagation\_time \geq action\_time$ steigt die Konfliktrate um mindestens $2 \cdot n^2$.

Die Zeit zum Propagieren einer Änderung $propagation\_time$ ist abhängig von der Bandbreite der Netzwerkverbindung und von der Geschwindigkeit in der das Anwenden einer Änderung auf den Replikaten möglich ist. Wenn beispielsweise mit Write Sets gearbeitet wird, dann kann das Anwenden des Write Set durchaus schneller sein, als das Ausführen der ursprünglichen Anweisung.

Die Zeit, die bei Eager Replication zum Propagieren von Änderungen benötigt wird bestimmt den Faktor c. Wenn das Propagieren der Änderungen optimiert werden kann, so wirkt sich das positiv auf die Konfliktrate aus. Dies soll anhand der theoretischen Überlegungen am Beispiel eines Replikationssystems mit 10 Knoten verdeutlicht werden:

1. Fall Sei $propagation\_time = 2 \cdot action\_time \Rightarrow c = 3$.

   Dann ist die Konfliktrate des Replikationssystems 300 mal so groß, wie bei einem System mit nur einem Knoten.

2. Fall Sei $propagation\_time = action\_time \Rightarrow c = 2$.

   Dann ist die Konfliktrate des Replikationssystems 200 mal so groß, wie bei einem System mit nur einem Knoten.

3. Fall Sei $propagation\_time = \frac{1}{2} \cdot action\_time \Rightarrow c = 1.5$.

   Dann ist die Konfliktrate des Replikationssystems 150 mal so groß, wie bei einem System mit nur einem Knoten.

Der beschriebene Verbesserungseffekt verstärkt sich, wenn die Knotenanzahl < 10 ist, da dann das Quadrat in der Anzahl der Knoten gegenüber dem Faktor c nicht mehr so stark ins Gewicht fällt. Im Gegenzug nimmt er mit steigender Knotenanzahl (> 10) ab. Allerdings erscheinen mir 10 Replikate weltweit für eine Datenbank realistischer als 20. Dies ist aber abhängig von der Anwendung.

Es ist trotzdem nicht davon auszugehen, dass die Zeit zum Propagieren der Änderungen $propagation\_time \lll action\_time$ ist, sodass der Quotient $\frac{propagation\_time}{action\_time}$ niemals ≈ 0 sein wird. Bei $c = 1.5$ müsste das Propagieren der Änderungen bereits doppelt so schnell möglich sein, wie das ursprüngliche Ausführen der Änderungen. Als einziges Beispiel fällt einem dazu zunächst *the-human-is-in-the-loop* ein. Anzunehmen, dass $c \approx 2$ ist, dürfte realistisch sein. Möglicherweise gibt es aber noch andere Möglichkeiten die Laufzeit einer Transaktion so wenig wie möglich durch das Replikationssystem zu vergrößern.

Abschätzung der Konfliktrate bei Lazy Replication

Betrachten wir nun Lazy Replication. Auch für die Bewertung dieser Strategie schließe ich mich der Argumentation von Grey et. al. an (vgl. [GHOS96]). Bei Lazy Replication werden die Änderungen einer Transaktion erst nach deren Commit asynchron auf den Replikaten eingebracht. Dies hat zwei bezüglich der Konfliktrate maßgebliche Vorteile: Zum einen wird die Laufzeit der Transaktion nicht durch das Propagieren der Änderungen vergrößert. Dies führt dazu, dass die Blockierungen geringer sind, und damit auch die Konflikte. Kommt eine Transaktion T nicht zu einem erfolgreichen Ende, so wurden keine wertvollen Ressourcen auf den anderen Knoten vergeudet, und es wurden keine anderen Transaktionen auf diesen Knoten durch T blockiert.

Da das Propagieren der Änderungen asynchron erfolgt, wirkt sich das Propagieren nicht auf die Laufzeit einer Transaktion aus. Entsprechend sind bei *nodes* Knoten zu einem Zeitpunkt $nodes{\cdot}tps{\cdot}actions{\cdot}action\_time$ Transaktionen aktiv:

$$transactions\_lazy = nodes \cdot tps \cdot actions \cdot action\_time. \tag{5.11}$$

Analog zu Gleichung 5.3 beträgt die Wahrscheinlichkeit dafür, dass eine Transaktion ein Datenelement schreibt, der nebenläufig von einer anderen Transaktion geschrieben wurde:

$$P_{lazy}(transaction) \approx \frac{nodes \cdot tps \cdot action\_time \cdot actions^3}{2 \cdot db\_size}. \tag{5.12}$$

Entsprechend ergibt sich wiederum die Wahrscheinlichkeit dafür, dass die Transaktion in der nächsten Sekunde ein Datenelement schreibt, der bereits von einer nebenläufigen Transaktion geschrieben wurde, durch die Division von $action{\cdot}action\_time$:

$$CR_{lazy}(transaction) \approx \frac{nodes \cdot tps \cdot actions^2}{2 \cdot db\_size}. \tag{5.13}$$

Die Gesamtkonfliktrate $CR_{lazy}(system)$ ergibt sich durch Multiplikation mit der Anzahl laufender Transaktionen $transactions\_lazy$:

$$\begin{aligned} CR_{lazy}(system) &\approx \frac{nodes \cdot tps \cdot actions^2}{2 \cdot db\_size} \cdot transactions\_lazy \\ &= \frac{nodes^2 \cdot tps^2 \cdot actions^3 \cdot action\_time}{2 \cdot db\_size}. \end{aligned}$$

Somit gilt mit Gleichung 5.5:

$$CR_{lazy}(system) = nodes^2 \cdot CR(system). \tag{5.14}$$

Offensichtlich steigt auch bei Lazy Replication die Konfliktrate um eine Größenordnung von n^2. Die Tatsache, dass sich die Konfliktrate, sowohl bei Eager als auch bei Lazy Replication, um eine Größenordnung von n^2 erhöht, ergibt sich in natürlicher Weise aus der Replikation und lässt sich nicht ändern. Denn bei Lazy Replication werden die Transaktionen durch das Replikationssystem überhaupt nicht blockiert. Die Anzahl der Konflikte ist bei Lazy Replication einzig auf die große Menge gleichzeitiger Anwendungstransaktionen, die durch die replizierten Knoten möglich werden, zurückzuführen.

Performancevorteile kommen bei der Lazy Replication gegenüber der Eager Replication dadurch zum Tragen, dass die Änderungen asynchron propagiert werden, und mit den Gleichungen 5.10 und 5.14 gilt dann:

$$CR_{eager}(system) = c \cdot CR_{lazy}(system) \tag{5.15}$$

Die Konfliktrate ist bei Eager Replication also um den Faktor c größer als bei Lazy Replication. Der Faktor c wird aber maßgeblich von der Zeit, die zum Propagieren der Änderungen benötigt wird, bestimmt (vgl. Abschnitt 5.2.1). Somit ist bei Eager-Systemen die Zeit, die für das Propagieren benötigt wird kritisch. Sie ist (der einzige) Ansatzpunkt für Optimierungen.

Kombiniert man Lazy Replication mit Group Replication so kann man prinzipiell den größten Durchsatz an Transaktionen erzielen. In diesem Fall kann dann jeder Knoten jede beliebige Transaktion selbständig abwickeln. Beim Propagieren der Änderungen können dann allerdings nach Commit noch Serialisierbarkeitsfehler festgestellt werden. In diesem Fall müssen Transaktionen gegebenenfalls kompensiert werden. Dies kann außerdem manuelles Eingreifen erfordern und ist vom Isolationsgrad des konkreten Replikationssystems abhängig.

Möchte man Kompensation und manuelles Eingreifen ausschließen, so wäre noch eine Kombination von Lazy Replication mit Master Replication möglich. Master-Replication hat generell den Vorteil, dass alle Schreiboperationen, die ein bestimmtes Datenelement betreffen auf seinem Master-Knoten synchronisiert werden. Konflikte werden dann auf dem Master-Knoten beispielsweise über Blockierungen aufgelöst. Im Allgemeinen kann dann ein beliebiger Knoten nicht alle Schreiboperationen einer Transaktionen selbständig durchführen, sondern muss gegebenenfalls an den Master delegieren. Dadurch steigt der Kommunikationsaufwand und die Dauer einer Transaktion wird dann wieder durch das Replikationssystem vergrößert. Master Replication kann generell immer dann sinnvoll eingesetzt werden, wenn es die Anwendung erlaubt, dass die Daten so auf die unterschiedlichen Master-Knoten verteilt werden können, dass eine sinnvolle Lastverteilung gegeben ist.

5.3 Synchronisation in verteilten Systemen

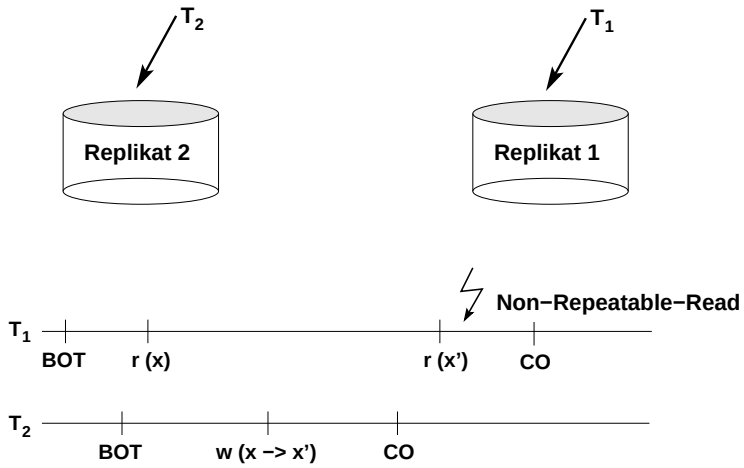
Für verteilte Systeme gibt es speziell entwickelte Synchronisationsverfahren. Dabei sind insbesondere das *Mehrversionenverfahren* und *Optimistische Synchronisationsverfahren* zu nennen. Die klassischen Sperrverfahren sind bei einem replizierten System nicht so gut geeignet. Um dies zu verdeutlichen werden die einzelnen Synchronisationsverfahren anhand des Beispiels aus Abbildung 5.1 näher untersucht. Wir betrachten zwei Knoten eines Replikationssystems und zwei nebenläufige Transaktionen T_1 und T_2. Würden die beiden Transaktionen vollkommen unsynchronisiert auf einem Knoten laufen, so träte bei T_1 ein Non-Repeatable-Read auf. Wir wollen nun aber betrachten was geschieht, wenn T_1 auf dem Replikat R_1, und T_2 auf dem Replikat R_2 läuft.

Erfolgt keine Synchronisation auf globaler Ebene, dann kommt es zu einem undefinierten und chaotischen Verhalten. Werden die Änderungen synchron mit Commit propagiert, tritt bei T_1 ein Non-Repeatable-Read auf. Bei Lazy Replication wäre der Ausgang unklar. Wird die Änderung zufällig erst nach der zweiten Leseoperation von T_1 propagiert, tritt kein Synchronisationsfehler auf und es gibt eine theoretische Serialisierbarkeitsreihenfolge von $T_1 \mid T_2$. Im Folgenden wird untersucht, wie das Verhalten bei dem Einsatz unterschiedlicher Synchronisationsverfahren ist.

5.3.1 Sperrverfahren

Wendet man die klassischen Sperrverfahren direkt auf ein Replikationssystem an, so müssten alle Sperren auf allen Replikaten auf globaler Ebene verwaltet werden. Dies könnte zum Beispiel über einen Transaktionsmanager realisiert werden, der alle Sperren verwaltet. Das Anfordern und Vergeben der Sperren wäre dann mit einem enormen Kommunikationsaufwand verbunden. Denn jede Sperranforderung

Abbildung 5.1 Synchronisation bei Replikaten



muss vom globalen Transaktionsmanager bewilligt werden. Denkbar wäre auch eine dezentrale Variante ohne globalen Manager. Dann müssten sich allerdings alle Replikate für jede Sperre einigen, ob diese an ein bestimmtes Replikat vergeben werden kann oder nicht. Dies dürfte einen noch deutlich höheren Kommunikationsaufwand mit sich bringen.

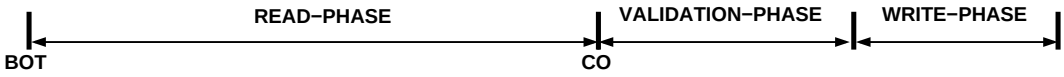
In dem Beispiel aus Abbildung 5.1 fordert T_1 als erstes eine Lesesperre für das Datenelement x an. Wenn T_2 das Datenelement x schreiben möchte, muss T_2 warten bis T_1 die Commit-Operation ausführt und die Lesesperre auf globaler Ebene wieder freigeben wird. Die Transaktion T_2 wird also nicht unerheblich blockiert, weil durch das Anfordern und Freigeben der Sperren zusätzlich insgesamt vier mal Latenz hinzukommt.

Wir haben bereits in Abschnitt 5.2 gesehen, dass die Anzahl der Transaktionen bzw. der MPL linear in der Anzahl der Knoten wächst. Die Konfliktrate steigt dabei mindestens quadratisch. Dies bedeutet, dass sich die Blockierungen, die durch die Sperren entstehen, gegenüber einem einzelnen Knoten sowieso schon stark erhöhen. Hinzu kommt, dass bei Einsatz eines Sperrverfahren auch Deadlocks auftreten können, die sich über mehr als ein Knoten erstrecken (vgl. [Thi07]). Es wurde im vorigen Abschnitt aber gezeigt, dass gerade Blockierungen so weit wie möglich zu vermeiden sind. Insbesondere dann, wenn ein hoher MPL gegeben ist. Es ist davon auszugehen, dass die durch die Sperren entstehenden zusätzlichen Blockierungen sich sehr negativ auf die Skalierbarkeit des replizierten Systems auswirken. Ein Verfahren, dass die Transaktionen so wenig wie möglich blockiert, sodass Ressourcen so schnell wie möglich wieder freigegeben werden, ist für den verteilten Fall angemessen.

5.3.2 Optimistische Synchronisationsverfahren

Bei optimistischen Synchronisationsverfahren werden Konflikte als Ausnahme angesehen. Dementsprechend darf jede Transaktion zunächst jede Operation ausführen und wird nicht blockiert. Änderungen führt die Transaktion dabei zunächst in ihrem eigenen privaten Transaktionspuffer durch (vgl. [HR01]). Wenn die Transaktion die Commit-Operation aufruft, findet zunächst eine *Validierungsphase* statt. Falls in der Validierungsphase ein möglicher Serialisierbarkeitsfehler festgestell wird, wird die validierende Transaktion oder die konfliktionäre Transaktion zurückgesetzt. Ansonsten tritt die Transaktion in die *Schreibphase* ein, d.h. alle Änderungen aus ihrem privaten Puffer werden in die Datenbank eingebracht und die Transaktion befindet sich im Zustand commited. In Abgrenzung dazu, wird die normale Ausführungsphase auch als *Lesephase* bezeichnet. Der Lebenszyklus einer Transaktion bei Optimistischer Synchronisation ist in Abbildung 5.2 graphisch dargestellt.

Abbildung 5.2 Lebenszyklus einer Transaktion bei Optimistischer Synchronisation



Weil die Schreibphase immer im Rahmen der Commit-Operation stattfindet, entspricht die Serialisierbarkeitsreihenfolge, der Reihenfolge der Commit-Operationen. Die Transaktion, die als letztes die Commit-Operation ausführt, muss dann in der Serialisierbarkeitsreihenfolge die letzte gewesen sein, da ihre Änderungen alle Vorangegangenen überschreiben können. Konflikte werden anhand der sogenannten *Read Sets* und *Write Sets* ermittelt. Ein Read Set enthält alle gelesenen und das Write Set alle geschriebenen Datensätze der Transaktion. In der Validierungsphase werden die Read Sets und Write Sets von nebenläufigen Transaktionen gegeneinander abgeglichen. Dabei können alle potentiellen Serialisierbarkeitsfehler festgestellt werden. Es ist dabei allerdings auch möglich, dass eine Transaktionen zurückgesetzt wird, obwohl eigentlich kein Konflikt vorgelegen hat.

Bei Optimistischen Synchronisationsverfahren entstehen zwar keine Blockierungen, dafür ist damit zu rechnen, dass erheblich mehr Transaktionen zurückgesetzt werden müssen. Denn ein Konflikt wird immer mit dem Abbruch einer Transaktion aufgelöst (vgl [HR01]). Dies ist insbesondere für langlaufende Transaktionen problematisch. Die Wahrscheinlichkeit, dass bei einer langlaufenden (meist lesenden) Transaktion ein Konflikt auftritt ist größer als bei einer kurzen Transaktionen, die gerade einmal einen Datensatz schreibt. Man bezeichnet dieses Problem als *starvation*. Das "Verhungern" einer Transaktion bezeichnet man hierbei die Situation, in der sie aufgrund auftretender Konflikte immer wieder abgebrochen und neu gestartet werden muss.

Hinzu kommt, dass die Validierungs- und Schreibphasen der Transaktionen trotzdem synchronisiert werden müssen. Die Transaktionen werden zwar während ihrer gesamten Lesephase nicht blockiert, dafür stellen die Commit-Operation einen Engpass dar, da diese echt sequentiell ausgeführt werden müssen.

Von Vorteil ist, dass bei optimistischen Synchronisationsverfahren keine Deadlocks auftreten können. Außerdem bieten sich optimistische Verfahren für den verteilten Fall an. Ein Replikat entspricht dann nämlich einem großen privaten Puffer, der von allen Transaktionen des Replikats verwendet wird. Auf dem Replikat können die Transaktionen lokal mit einem beliebigen Verfahren synchronisiert werden und auf globaler Ebene mit einem Optimistischen. Dazu bietet sich die *rückwärtsgerichtete Synchronisation* (*BOCC*) an (vgl. [HR01].

Bei BOCC validiert jede Transaktion gegen alle anderen Transaktionen, die während ihrer Laufzeit in den Zustand commited gewechselt sind. Potentielle Konflikte werden ermittelt, indem die Write Sets der in Frage kommenden Transaktionen mit dem Read Set der Transaktion abgeglichen werden. Falls ein Datensatz aus dem Read Set in mindestens einem Write Set enthalten ist, liegt ein potentieller Konflikt vor. Denn in diesem Fall hat die Transaktion die Änderung einer Transaktion, die in der Serialisierbarkeitsreihenfolge vor ihr ist, nicht mitbekommen. Dies ist auch an dem Beispiel aus Abbildung 5.1 ersichtlich. Hier ist das Datenelement x in dem Read Set von T_1 enthalten. T_2 hat dieses Datenelement aber nebenläufig zu T_1 geändert, und die Änderung vor dem Commit von T_1 in die Datenbank eingebracht. In diesem Fall würde T_1 zurückgesetzt werden. Hätte T_1 das Datenelement x auch noch geschrieben, so läge ein Lost-Update vor. Hätte T_1 das Datenelement nur geschrieben, und nicht gelesen, läge kein Konflikt vor. Denn dann hätte T_1 blind geschrieben. In diesem Fall würde T_1 aber zu Unrecht zurückgesetzt werden. Da T_1 insgesamt nur gelesen hat, wäre theoretisch eine Serialisierbarekeitsreihenfolge von $T_1 \mid T_2$ gegeben.

Für das Replikationssystem würde es also genügen einen globalen Transaktionsmanager zu installieren, der die globalen Commit-Zeitpunkte und die Write Sets aller Transaktionen kennt. Bei langlaufenden Transaktionen müssen diese Write Sets entsprechend lange aufgehoben werden. Die Transaktionen werden während ihrer Lesephase nicht durch das Replikationssystem blockiert. Allerdings erfordert die Commit-Operation das Versenden des Read Set und Write Set an den globalen Transaktionsmanager. Wenn dieser keinen Konflikt feststellt, kann die Transaktion in den Zustand commited wechseln. Dabei müssen alle Commit-Operationen von allen Transaktionen des Replikationssystems vom globalen Transaktionsmanager synchronisiert werden. Es findet also zumindest eine größere Blockierung am Ende der Transaktion statt.

5.3.3 Mehrversionenverfahren

Der Einsatz eines Mehrversionenverfahren kann die Menge an Synchronisationskonflikten in erheblichem Maße reduzieren. Bei einem Mehrversionenverfahren müssen nur die Schreiboperationen synchronisiert werden. Die Leseoperationen werden bei einem reinen Mehrversionenverfahren nicht synchronisiert. Dies wird durch die Verwaltung eines *Versionen-Pool* ermöglicht (vgl. [HR01]). Bei jeder Schreiboperation, wird für das Datenobjekt eine neue Datenversion angelegt. Die von einer Transaktion erzeugten Datenversionen werden erst bei ihrem Commit für alle nachfolgenden Transaktionen sichtbar. Nebenläufige Transaktionen sehen während ihrer gesamten Dauer diejenige Datenversion, die zu ihrem BOT gültig war. Damit sieht jede Transaktion während ihrer gesamten Dauer den zu ihrem BOT jüngsten transaktionskonsistenten Zustand. Nebenläufige Schreiboperationen bleiben ihr verborgen. Ein Non-Repeatable-Read kann niemals auftreten.

Somit werden reine Lesetransaktionen niemals blockiert. Die Schreiboperationen können mit einem beliebigen anderen Verfahren synchronisiert werden. Von außen betrachtet ergibt sich in Anlehnung an Kemme (vgl. [LKPMJP05]) für jede Transaktion die folgende anschauliche Charakteristik:

- Die Transaktion liest alle Daten bei ihrem BOT
- und bei ihrem Commit schreibt sie alle ihre Daten.

Denn jede Transaktion sieht ausschließlich Änderungen von Transaktionen, die vor ihrem BOT bereits die Commit-Operation erfolgreich abgeschlossen haben. Und erst mit ihrem Commit werden alle von ihr angelegten Datenversionen gültig. Wird das reine Mehrversionenverfahren zur Synchronisation eingesetzt, dann sind die entstehenden Historien nicht serialisierbar. Die resultierende Konsistenzstufe wird als *Snapshot Isolation* bezeichnet und im nächsten Abschnitt ausführlich dargestellt.

Betrachten wir nun was geschieht, wenn das Mehrversionenverfahren bei einem Replikationssystem eingesetzt wird. Dann genügt es auch auf globaler Ebene nur die schreibenden Transaktionen zu synchronisieren. Dies könnte beispielsweise auch mit einem optimistischen Synchronisationsverfahren gelöst werden. Wenn zwei Transaktionen gleichzeitig starten und beide denselben Datensatz ändern, muss auf jeden Fall eine abgebrochen werden. Denn dann hat die Transaktion, die als letzte die Commit-Operation ausführen würde, in keinem Fall die Änderungen der anderen mitbekommen und es liegt potentiell ein Lost-Update vor. Der globale Transaktionsmanager müsste analog zur optimistischen Synchronisation die Write Sets und Commit-Zeitpunkte aller erfolgreich abgeschlossenen Transaktionen verwalten. Ebenso muss er alle Commit-Operationen synchronisieren. Schreibende Transaktionen werden also in ihrer Commit-Operation blockiert. Lesende Transaktionen überhaupt nicht. Wenn die Änderungen synchron propagiert werden können, liegt auf globaler Ebene Snapshot Isolation vor.

Betrachten wir dazu noch einmal kurz das Beispiel aus Abbildung 5.1. T_1 ist ein reiner Leser, somit muss die gesamte Transaktion nicht synchronisiert werden. Nebenläufig ändert T_2 das Datenelement x. Aber T_1 wird davon nichts mitbekommen. Selbst falls die Änderung von T_2 synchron propagiert und in die Datenbank von Replikat 1 eingebracht wird, so stellt dies eine Änderung dar, die in jedem Fall nebenläufig zu T_1 stattfindet. T_1 sieht aber nur die Datenversion von x, die zu ihrem BOT gültig war. Somit können in diesem Fall beide Transaktionen erfolgreich die Commit-Operation ausführen. T_2 wird dabei so lange vom globalen Transaktionsmanager blockiert, wie dieser benötigt, um festzustellen, ob ein Schreibkonflikt vorliegt oder nicht.

5.4 Snapshot Isolation

Eine weitere wichtige Konsistenzstufe wurde in dem Papier [BBG+95] eingeführt. Es handelt sich dabei um die *Snapshot Isolation*. Diese wurde durch die Entwicklung eines neuartigen Synchronisationsverfahren technisch möglich. Wenn ein Mehrversionenverfahren eingesetzt wird und alle Schreibkonflikte zusätzlich mit einem beliebigen anderen Verfahren behandelt werden, beträgt die Konsistenzstufe mindestens Snapshot Isolation. Die Funktionsweise eines Mehrversionenverfahrens wurde in Abschnitt 5.3.3 grob dargestellt. Ein wichtiger Vorteil des Mehrversionenverfahrens ist, dass Leseoperationen nicht synchronisiert werden müssen. Eine Transaktion liest während ihrer gesamten Dauer von einem *Snapshot*. Der Snapshot einer Transaktion entspricht dem jüngsten transaktionskonsistenten Zustand zum Zeitpunkt ihres BOT. Das bedeutet, dass die Transaktion nur Änderungen von Transaktionen sieht, die vor ihrem BOT erfolgreich die Commit-Operation ausgeführt haben. Die Änderungen nebenläufiger Transaktion sind nicht in ihrem Snapshot enthalten. Dadurch wird die Transaktion von allen nebenläufigen Transaktionen isoliert und Non-Repeatable-Reads können nicht auftreten. Schreiboperationen müssen nach wie vor synchronisiert werden.

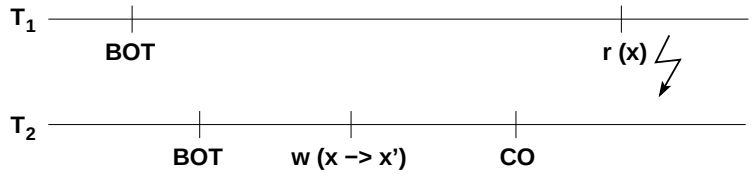
5.4.1 Anomalien bei Snapshot Isolation

Allerdings können bei Snapshot Isolation nicht-serialisierbare Historien auftreten. Dies ist vor allem darauf zurückzuführen, dass die Daten eines Snapshots schnell veraltet sein können. Dies kommt insbesondere dann zum Tragen, falls die Transaktion eine lange Laufzeit hat. Es können zwei für die Snapshot Isolation spezifische Anomalien identifiziert werden: *Read Skew* und *Write Skew*.

Read Skew

Ein Read Skew stellt das Lesen veralteter Daten dar. Ein Read Skew alleine ist noch kein Serialisierbarkeitsfehler, allerdings kann es trotzdem zu sehr unschönen Effekten kommen. Betrachten wir dazu die Transaktionen T_1 und T_2 aus Abbildung 5.3. Die beiden Transaktionen starten ungefähr gleichzeitig, d.h. $BOT_{T_1} \approx BOT_{T_2}$. Als erstes schreibt T_2 das Datenelement x und führt die Commit-Operation aus. Anschließend liest T_1 das Element x. Die Änderung von T_2 ist aber nicht in dem Snapshot von T_1 enthalten, folglich liest T_1 einen veralteten Wert. Nun ist dieser Ablauf trotzdem serialisierbar mit der Reihenfolge $T_1 \mid T_2$. Trotzdem kann es zu schwer nachvollziehbaren Folgeerscheinungen kommen. Denn T_1 läuft ja noch weiter, und führt die Commit-Operation unter Umständen erst viel später aus als T_2. Gleichzeitig wäre aber T_1 nach der Serialisierbarkeitsreihenfolge vollständig vor T_2 gewesen. Je länger T_2 läuft umso weniger aktuell sind die Daten. Weiterhin kann T_1 in Abhängigkeit von den gelesenen veralteten Daten weitere Aktionen ausführen. Insbesondere dann, wenn T_1, abhängig von dem Gelesenen, andere Daten schreibt, können Serialisierbarkeitsfehler auftreten.

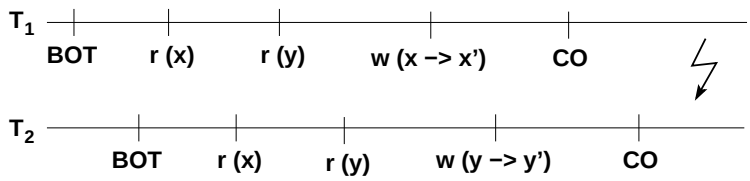
Abbildung 5.3 Snapshot Isolation: Read Skew



Write Skew

Ein Write Skew stellt in jedem Fall ein Serialisierbarkeitsfehler dar. Er ist aber eng verwandt mit dem Read Skew und ist ebenfalls darauf zurückzuführen, dass Transaktionen veraltete Daten in ihrem Snapshot haben. In Abbildung 5.4 ist ein Beispiel für einen Write Skew gegeben. Die Transaktionen T_1 und T_2 starten wiederum in etwa gleichzeitig ($BOT_{T_1} \approx BOT_{T_2}$). Anschließend lesen beide die Datenelemente x und y. Danach schreibt T_1 das Objekt x und T_2 schreibt y. In dieser Historie liegt insbesondere kein Schreibkonflikt vor, da T_1 und T_2 unterschiedliche Datenobjekte schreiben. Trotzdem ist die Historie nicht serialisierbar. Dies kann man sehr leicht zeigen. Wäre die Historie serialisierbar dann entweder in der Reihenfolge $T_1 \mid T_2$ oder $T_2 \mid T_1$.

Abbildung 5.4 Snapshot Isolation: Write Skew



1. Fall $T_1 \mid T_2$**:** Wäre T_1 vollständig vor T_2, dann müsste T_2 alle Änderungen von T_1 sehen. Die Änderung von T_1 auf dem Datenelement x ist aber nicht in dem Snapshot von T_2 enthalten.

$\Rightarrow$ Die Historie ist nicht äquivalent zu der seriellen Historie $T_1 \mid T_2$.

2. Fall $T_2 \mid T_1$**:** Hier gilt analog: Wäre T_2 vollständig vor T_1, dann müsste T_1 alle Änderungen von T_2 sehen. Die Änderung von T_2 auf dem Datenelement y ist aber nicht in dem Snapshot von T_1 enthalten.

$\Rightarrow$ Die Historie ist nicht äquivalent zu der seriellen Historie $T_2 \mid T_1$.

Damit wurde gezeigt, dass die Historie nicht serialisierbar ist. Das Auftreten eines Write Skew kann schwerwiegend sein, wenn die Anwendung dies nicht maskieren kann. In [BBG+95] wird folgendes Beispiel genannt: Seien die beiden Datenelemente x und y Kontostände. Die Anwendung erlaubt negative Kontostände, solange die Summe aller Kontostände positiv bleibt. Die Transaktionen T_1 und T_2 lesen zu Beginn die Kontostände. T_1 verkleinert den Kontostand x, sodass $x' + y > 0$. T_2 macht das entsprechende für Kontostand y. Wenn beide Transaktionen beendet sind, betragen die Kontostände x' bzw. y'. Sei nun $x = 10, y = 5, x' = -3, y' = 2$. Dann ist $x' + y = -3 + 5 = 2 > 0$ und $y' + x = 2 + 10 = 12 > 0$, aber $x' + y' = -3 + 2 = -1 < 0$.

Ein Write Skew kann also beispielsweise auftreten, wenn zwischen zwei unterschiedlichen Datenobjekten eine implizite Bedingung besteht. Dann kann es zwischen zwei Transaktionen, die gleichzeitig starten zu einem Write Skew kommen. Wenn jede Transaktion nur jeweils eins der beiden Objekte schreibt, führt dies zu keinem Schreibkonflikt. Die Transaktion, die als zweites die Commit-Operation ausführt, hat allerdings das Update der anderen Transaktion nicht mitbekommen und hat infolge dessen die Bedingung gegen einen veralteten Wert geprüft. Es kann dann passieren, dass sie das zweite Datenobjekt mit einem Wert beschreibt, der die Bedingung verletzt.

Allgemein beschreibt ein Write Skew die Situation, dass eine Menge nebenläufiger Transaktionen gleichzeitig mit einer bestimmten Menge von Daten arbeiten, diese lesen und jeweils eine Teilmenge davon auch schreiben. Und zwar so, dass kein Schreibkonflikt entsteht. Jede dieser Transaktionen hat aber die Änderungen der

anderen Transaktionen nicht mitbekommen und schreibt deswegen potentiell fehlerhafte Daten, weil aus den veralteten Daten falsche Rückschlüsse gezogen wurden.

5.5 Folgerungen für das Constraint-basierte Datenbank-Caching

Beim Constraint-basierten Datenbank-Caching werden ebenfalls Daten repliziert. Allerdings unterscheidet sich ein Datenbank-Cache in einigen Punkten von einem vollständigen Replikat:

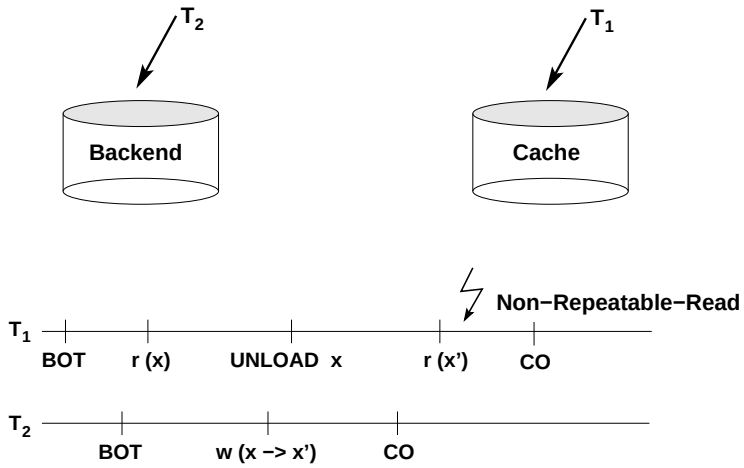
- Ein Cache speichert eine Teilmenge der Backend-Daten und ist keine vollständige Kopie.
- Ein Cache ist dynamisch. Daten können jederzeit geladen und entladen werden.
- $\Longrightarrow$ Ein Cache kann niemals gewährleisten alle Anfragen selbst beantworten zu können.

Zunächst einmal können die genannten Punkte bei der Synchronisation in einem positiven Sinne ausgenutzt werden. Dies wird in späteren Kapiteln noch deutlicher werden. Allerdings ergeben sich aus diesen Eigenheiten auch einige Spezialfälle, die bei der gewöhnlichen Replikation nicht auftreten können, aber beim Caching beachtet werden müssen. Beispielsweise kann die Historie aus Abbildung 5.1 unter dem Blickwinkel des Caching zu Komplikationen führen. Deswegen ist in Abbildung 5.5 diese Historie am Beispiel des Datenbank-Caching nochmals graphisch dargestellt. Transaktion T_1 ist ein reiner Leser, und liest zunächst das Datenelement x. Auf dem Backend ändert eine nebenläufige Transaktion T_2 das Datenelement x und bringt die Änderung mit der Commit-Operation erfolgreich in die Backend-Datenbank ein. Inzwischen wurde das Datenelement x vom Cache entladen. Nun liest T_1 das Element x erneut, und der Cache muss diese Anfrage ans Backend weiterleiten, da das Element nicht mehr geladen ist. T_1 liest von der Backend-Datenbank den geänderten Wert x', und wir haben einen Non-Repeatable-Read.

Es wurde bereits ausführlich dargelegt, dass es im verteilten Fall schwierig ist mit einem reinen Sperrverfahren zu arbeiten. Zum einen wird durch die Blockierungen, die bei Sperren immer entstehen, die Konfliktrate zusätzlich erhöht. Hinzu kommt dass das Anfordern einer Sperre vor jeder Leseoperation einen der Hauptvorteile des Datenbank-Caching untergräbt. Denn die kurzen Antwortzeiten, die von dem Cache zur Verfügung gestellt werden, sind dann weitgehend hinfällig, weil jedesmal wenn ein neues Datenelement gelesen wird, die Strecke zum globalen Transaktionsmanager zwei mal zurückgelegt werden muss, um die Sperre zu setzen.

Optimistische Synchronisation hilft an dieser Stelle auch nicht weiter, denn der Fehler tritt mitten in der Ausführungsphase auf. Ein möglicher Ansatz wäre es dem

Abbildung 5.5 Synchronisation beim Datenbank-Caching



Cache zu verbieten Datenelemente zu entladen, die von einer aktiven Transaktion referenziert wurden. Aber dies würde sehr massiv in die adaptiven Regularien des Caching-Systems eingreifen und das eigentliche Caching zu stark einschränken.

An dieser Stelle hat man nicht viele andere Möglichkeiten, als mit einem Mehrversionenverfahren zu arbeiten. Denn aufgrund des dynamischen Lade- und Entlademechanismus des Cache, muss für jede ACCache-Transaktion sichergestellt sein, dass die Datenversionen, welche zu ihrem globalen BOT gültig waren, bis zu ihrem EOT zumindestens auf der Backend-Datenbank gehalten werden. Es gibt keine Garantien, dass irgendein Cache, irgendeinen Datensatz zu einem bestimmten Zeitpunkt speichert. Deshalb ist es erforderlich, dass jede ACCache-Transaktion einen globalen Transaktionskontext auf dem Backend hat. Ein Mehrversionenverfahren auf Cache- und Backend-Datenbank bringt an dieser Stelle folgende Vorteile:

- Wenn der Datensatz x vom Cache entladen wird, kann T_1 ihn weiterhin von der Cache-Datenbank lesen, da der alte Datensatz x im Snapshot von T_1 ist.
- Jede Transaktion kann gegebenenfalls jeden Datensatz in der Datenversion, der zu ihrem BOT gültig war, von der Backend-Datenbank lesen.
- Leser müssen nicht synchronisiert werden. Dies führt zu einer verminderten Konfliktrate.
- Transaktionen werden deutlich weniger blockiert als bei einem Sperrverfahren.

5.6 Zusammenfassung

In diesem Kapitel wurde zunächst eine Einführung in bestehende Replikationstechniken gegeben. Anschließend wurden diese bezüglich ihrer Konfliktrate in Abhängigkeit der Knotenzahl und der Transaktionsgröße untersucht. Dabei wurde festgestellt, dass bei einem replizierten System die Konfliktrate immer in einer Größenordnung von n^2 in Abhängigkeit der Knotenzahl n wächst. Bei der synchronen Ausführung aller Änderungsoperationen auf allen Knoten des replizierten Systems kommt noch eine Konstante c hinzu. Diese wird von der Zeit, die zum Propagieren der Änderungen benötigt wird, bestimmt.

Nun ist dieses Ergebnis sehr ernüchternd. Hinzu kommt, dass eigentlich alle Änderungen synchron auf allen Caches vorhanden sein sollten. Dies ist jedenfalls erforderlich, falls auf der globalen Ebene des Replikationssystem die Konsistenzstufe serialisierbar erreicht werden soll. Soll diese hohe Konsistenzstufe erreicht werden, so muss c so niedrig sein wie möglich. Wenn c vernachlässigbar klein wäre, wäre die Konfliktrate vergleichbar mit der eines Lazy Replication Systems. An diesem Punkt setzt das in der Arbeit entwickelte Verfahren an. Es wird versucht die Zeit, die zum Propagieren benötigt wird, so weit herabzudrücken wie möglich.

In diesem Kapitel wurden außerdem verschiedene Synchronisationsverfahren auf ihre Tauglichkeit im verteilten Fall untersucht. Dabei erscheint insbesondere das Mehrversionenverfahren vielversprechend. Denn dadurch ist sichergestellt, dass lesende Transaktionen überhaupt nicht blockiert werden. Der Einsatz eines Mehrversionenverfahrens bietet sich auch an, weil der Cache immer nur Teilmengen der Backend-Daten enthält. Es wurde gezeigt, dass andernfalls ein Non-Repeatable-Read auftreten könnte, der sich nur schwer maskieren ließe.

Eine weitere wichtige Erkenntnis ist, dass insbesondere im verteilten Fall eine Transaktion so wenig wie möglich blockiert werden sollten. Denn lange Blockierungen erhöhen die Wahrscheinlichkeit für das Auftreten eines Konfliktes. Dies schließt Sperrverfahren praktisch von vorneherein aus. Hinzu kommt das Anfordern und Freigeben der Sperren, was sehr viel Latenz bedeutet, und letztendlich auch zu Blockierungen führt. Schon das Anfordern der Lesesperren würde einen großen Vorteil des Datenbank-Caching untergraben: Die verkürzten Antwortzeiten, die durch den Cache bereitgestellt werden. Prefetching-Techniken müssen ausgeschlossen werden, da sie von JDBC nicht unterstützt werden. Optimistische Verfahren sind auch nicht das Allheilmittel, denn die Commit-Operation ist sehr umfangreich, und führt damit am Ende auch zu Blockierungen und einem hohen Synchronisationsaufwand an einer kritischen Stelle. Insbesondere führen auch die Rücksetzungen dazu, dass sich die Anzahl der aktiven Transaktionen erhöht, was im Endergebnis auch zu einer Erhöhung der Konfliktrate führt. Insgesamt muss ein guter Kompromiss gefunden werden zwischen dem Blockieren und dem Rücksetzen von Transaktionen.

6 Synchronisationsverfahren

Im Folgenden wird das im Zuge dieser Arbeit, entwickelten Synchronisationsverfahrens vorgestellt. Das Verfahren bietet den Anwendungstransaktionen des ACCache-Systems globale Snapshot Isolation. Das bedeutet, jede Anwendungstransaktion liest von einem Snapshot der, den zu ihrem BOT jüngsten, transaktionskonsisten Zustand entspricht. 1-Copy-Serializability wird also nicht erreicht. Es wird aber sichergestellt, dass jede Anwendungstransaktion für jede Anfrage ausschließlich die Datenversionen ihres Snapshot sieht. Dabei wird vorrausgesetzt, dass das zugrundeliegende Datenbanksystem ein Mehrversionenverfahren zur Synchronisation einsetzt. Dadurch ist keine vollständige Unabhängigkeit vom Datenbanksystemhersteller gegeben, allerdings synchronisieren bereits sehr viele Datenbanksysteme mit einem Mehrversionenverfahren. Beispielsweise Oracle seit Version 7, Microsoft SQL Server seit Version 2005, PostgreSQL und MySQL.

6.1 Konzeption

Das Verfahren implementiert ein Primary-Copy-Verfahren. Die Primärkopie kann eine beliebige Datenbank sein und entspricht im ACCache-System der Backend-Datenbank. Zu der Primärkopie kann es eine beliebige Anzahl von Cache-Instanzen geben, die jeweils eine Teilmenge der Daten der Primärkopie zwischenspeichern. Im Zusammenhang mit den Replikationsstrategien aus Kapitel 5.1 ist die Primärkopie ein Master, denn die Primärkopie ist die einzige Datenbank auf der Änderungen durchgeführt werden dürfen. Dies ist allerdings völlig transparent für die Anwendungstransaktion. Update-Operationen werden vom Cache an das Backend weitergeleitet. Insgesamt werden damit alle Schreiboperationen von der Backend-Datenbank synchronisiert. Alle globalen Schreibkonflikte werden von ihr behandelt.

Die Strategie zum Propagieren der Änderungen ist weder synchron noch asynchron. Sie ist aus Sicht der Anwendung eager und aus Sicht des Caching-Systems lazy. Man könnte sagen es handelt sich um ein halbsynchrones Verfahren, das möglichst performant ist, und optimal zugeschnitten ist auf das Constraint-basierte Datenbank-Caching. Die Vorteile die man aus der Tatsache ziehen kann, das es sich um Caches handelt und nicht um Replikate werden ausgenutzt. Beim Caching hat man viele

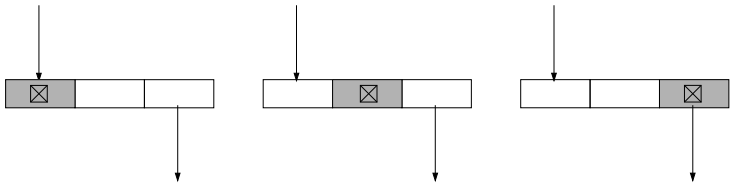
Freiheitsgrade, die man bei der Replikation nicht hat. Zur Erinnerung, Replikate sind vollständige Kopien und Caches speichern nur eine Teilmenge der Daten. Das naivste Verfahren zum Propagieren von Änderungen ist das Folgende: Wenn eine Transaktion einen Datensatz schreibt, entlade bei ihrem Commit auf den Caches alle Cache Units, die diesen Datensatz enthalten. Dann würden automatisch alle Anfragen, die Daten aus diesen Cache Units referenzieren, an das Backend weitergeleitet werden. Dort sehen sie die aktuelle Datenversion. Nach und nach füllt sich der Cache wieder und er kann die Anfragen wieder selbst auswerten.

Außerdem verfügt ein Cache über zusätzliche Metadaten in Form der Cache Group-Definition. Ziel ist es, so viele Informationen wie möglich aus diesen Constraints abzuleiten, die für die Synchronisation von Nutzen sein könnten. Es wurde bereits beim Probing bewiesen, das sich aus den Cache-Constraints effizient Informationen über den Inhalt des Caches ableiten lassen.

6.2 RCC-Strukturen und Updates

Zunächst soll untersucht werden, wie sich Update-Operationen auf eine RCC-Struktur auswirken. Dabei können drei Fälle unterschieden werden: Geändert wurde ein Wert in einer Spalte mit eingehenden RCCs, in einer Spalte mit ausgehenden RCCs, oder in einer Spalte ohne RCCs. In Abbildung 6.1 sind die drei Fälle, mit jeweils einem RCC, graphisch dargestellt. Diese schließen sich nicht gegenseitig aus. Die Änderung kann beispielsweise auch auf einer Spalte mit eingehenden und ausgehenden RCCs stattgefunden haben. Im Folgenden wird die Änderung eines Wertes abstrakt als eine Transition $old\_value \mapsto new\_value$ betrachtet.

Abbildung 6.1 Update-Operationen auf RCC-Strukturen



(a) Änderung auf einer Spalte mit eingehenden RCCs

(b) Änderung auf einer Spalte ohne RCC

(c) Änderung auf einer Spalte mit ausgehenden RCCs

1. Fall: Änderung auf einer Spalte mit eingehenden RCCs Bei der Änderung eines Datenelementes aus einer Spalte mit eingehenden RCCs, wird die Konsistenz des Cache nicht verletzt. Im schlechtesten Fall existiert *new_value* in

keiner Qellspalte der RCCs. Dann ist dieser Datensatz im Cache praktisch unerreichbar. Man könnte in diesem Fall auch erzwingen, dass der Datensatz gelöscht wird.

2. **Fall: Änderung auf einer Spalte ohne RCC** Da keine Spalte mit RCC betroffen ist, wird auch keiner der gültigen Constraints verletzt.
3. **Fall: Änderung auf einer Spalte mit ausgehenden RCCs** Das Ändern eines Datenelementes aus einer Spalte mit ausgehendem RCC ist vergleichsweise schwerwiegend. Eine solche Änderung kann die Konsistenz des Cache verletzen, und zwar dann, wenn in dieser Spalte *new_value* bis jetzt nicht existiert hat. Falls *new_value* vorher schon in der Spalte war, sind in den Zieltabellen bereits alle Tupel mit dem Wert *new_value* in den Zielspalten geladen, und der Cache ist in einem konsistenten Zustand. Sofern nach der Änderung *old_value* nicht mehr in der Spalte existiert, sind alle Tupel die von *old_value* über ausgehende RCC-Ketten referenziert werden potentiell obsolet und unerreichbar. Auch in diesem Fall könnte ein Entladen all derjenigen Tupel, die aufgrund der Änderungsoperation unerreichbar sind, durchgeführt werden. Es ist dabei aber genauso gut möglich, das aufgrund von Überlappungen in den geladenen Cache Units keine Tupel entladen werden können.

Insgesamt sind also alle Änderungen auf Spalten mit RCCs vergleichsweise teuer, da sie Lade- und Entladevorgänge auslösen können. Es lässt sich jedoch argumentieren, dass Änderungen auf Spalten mit RCCs vergleichsweise selten sind, da der Großteil aller RCCs natürlicherweise auf Primär- und Fremdschlüsselspalten definiert sind. Und Änderungen auf Primär-und Fremdschlüsselspalten sind eher unüblich. Folgt man dieser Argumentation, dann wären durch Update-Operationen ausgelöste Lade- und Entladevorgänge eher die Ausnahme.

Aber selbst wenn man davon ausgeht, dass die Nachlade- und Löschvorgänge beim Propagieren der Änderungen nicht stark ins Gewicht fallen, so wurde bereits in Kapitel 5.2 ausführlich dargestellt, dass bereits das synchrone Ausführen einer Änderungsoperation auf allen Knoten die Konfliktrate erheblich erhöht. Deshalb wird in diesem Synchronisationsverfahren eine neuer Ansatz erfolgt, der sich auf die Ausnutzung der Cache-Constraints stützt.

6.3 Strategie zum Propagieren von Änderungen

Ziel ist es alle Datensätze, die von den Änderungen einer schreibenden Transaktion betroffen sind synchron, also spätestens bei Commit dieser Transaktion, auf allen Caches unerreichbar zu machen. Geladene Tupel können in einer Cache Group beispielsweise unerreichbar gemacht werden, indem bestimmte RCCs temporär für

ungültig erklärt werden. Sind Datensätze unerreichbar, dann sind sie quasi unsichtbar, so als ob sie nicht im Cache geladen worden wären. Bezieht die Auswertung einer Anfrage unerreichbare Tupel ein, so muss die Anfrage ans Backend weitergeleitet werden, da der Cache sie nicht beantworten kann. Das Anwenden der Änderungen auf den Caches und eventuell notwendige Lade- und Entladevorgänge werden asynchron, also erst nach der Commit-Operation, durchgeführt. Sobald der Cache wieder in einem konsistenten Zustand ist, können die betroffenen Datensätze wieder erreichbar gemacht werden.

Zur Maskierung von Inkonsistenzen in einer Cache Group wurden in dieser Arbeit zwei Ansätze verfolgt und näher untersucht. Die erste Herangehensweise zeichnet sich durch Einfachheit und Eleganz aus, tendiert aber dazu, dass potentiell zu viele Datensätze unerreichbar werden. Bei der zweiten Vorgehensweise werden deutlich weniger Datensätze unsichtbar. Diese ist aber im Gegenzug erheblich komplizierter und führt zu einigen Komplikationen. Beispielsweise wären einige Anpassungen am Probing erforderlich. Im Folgenden werden die beiden Invalidierungsmöglichkeiten vorgestellt, wobei zunächst auf die feingranularere Variante eingegangen wird.

6.3.1 Invalidierung von RCC-Werten

Um zu verstehen wie man die Unerreichbarkeit von Datensätzen mittels RCC-Wert-Invalidierungen realisieren kann, betrachten wir zunächst den RCC $q_1 \rightarrow z_1$ aus Abbildung 6.2. Die Menge aller Werte, die in der Quellspalte q_1 vorkommen, wird im Folgenden als die Menge der Werte, die *an dem RCC anliegen* bezeichnet (vgl. Definition 6.1). Entsprechend wird ein solches Paar aus RCC und Wert auch *RCC-Wert* genannt. Die *Invalidierung eines RCC-Wertes* bedeutet dann, dass ein bestimmter Wert, der an einem RCC anliegt, invalidiert wird.

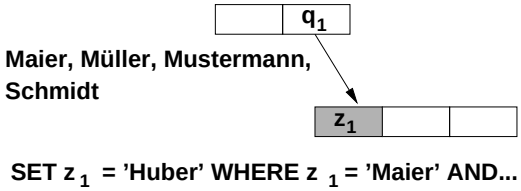
Definition 6.1 RCC-Werte

Die Menge aller Werte die an einem RCC anliegen, ist gleich der Menge aller Werte, die in der Quellspalte des RCC vorkommen.

In Abbildung 6.2 sind diese Werte neben dem RCC aufgetragen. Zur Wiederholung erläutere ich nochmals die Bedeutung dieser Werte für die Zieltabelle: Für jeden einzelnen, an einem RCC anliegenden Wert w, müssen in der Zieltabelle alle Tupel geladen sein, die in der Zielspalte den Wert w haben. Im Beispiel aus Abbildung 6.2 müssen in der Zieltabelle also alle Tupel vorhanden sein, die in der ersten Spalte z_1 den Wert *Maier, Müller, Mustermann* oder *Schmidt* haben.

Nehmen wir weiter an, eine Transaktion führt die Änderung $Maier \mapsto Huber$ für mindestens einen Datensatz der Zieltabelle durch. Dann müssen nach dem Com-

Abbildung 6.2 RCC-Wert-Invalidierung: Beispiel 1



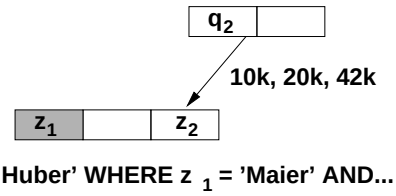
mit dieser Transaktion in den Caches die veralteten Datensätze unerreichbar sein. Für die Cache Group aus der Abbildung bedeutet dies beispielsweise, dass der RCC zumindestens für den Wert *Maier* ungültig erklärt werden muss. Er ist aber insbesondere für den Wert *Huber* ungültig, denn für *Huber* können nicht alle Verbundpartner ermittelt werden, da einige fälschlicherweise noch den Wert *Maier* in Spalte z_1 haben. Man könnte auch den ganzen RCC für temporär ungültig erklären. Sobald die Änderungen nachgezogen wurden und die Cache Group wieder in einem konsistenten Zustand ist, kann die Invalidierung wieder aufgehoben werden. Damit würden allerdings potentiell Anfragen ans Backend delegiert werden, die noch vom Cache beantwortet werden könnten. Theoretisch genügt es den RCC für die Werte *Maier* und *Huber* als ungültig zu markieren (alternative Bezeichnung: zu invalidieren).

Betrachten wir als nächstes den Fall, dass die Änderung eine Spalte betrifft, die über keinen eingehenden RCC verfügt. Weiterhin muss es aber mindestens einen eingehenden RCC geben, sonst wäre die ganze Cached Table unerreichbar und kann ignoriert werden (vgl. Kapitel 2.3.1). Ein Beispiel ist in Abbildung 6.3 gegeben. Wie bereits im vorigen Abschnitt 6.2 festgestellt, führt diese Änderung nicht zur Verletzung bestehender Cache-Constraints. Jedoch müssen auch in diesem Fall die geänderten Datensätze unsichtbar gemacht werden, da sie auf den Caches veraltet sind. Im Folgenden wird beschrieben, welche Schritte pro geändertes Tupel durchzuführen sind. Um ein geändertes Tupel unerreichbar zu machen benötigen wir das *Before-Image* dieses Tupels, also die Werte in den Spalten des Tupels bevor die Änderung stattfand. Am RCC ist dann der Wert zu invalidieren, den das geänderte Tupel in der Spalte mit dem eingehenden RCC hatte (und auch immer noch hat). Aus der Anfrage im Beispiel geht hervor, dass dieser Wert *Maier* betrug. Er ist entsprechend zu invalidieren.

Damit ergibt sich zusammenfassend die folgende Vorgehensweise zur Behandlung eingehender RCCs.

1. Falls die Tabelle keine eingehenden RCCs hat: Abbruch. Es ist nichts weiter zu tun, da die Tabelle unsichtbar ist.
2. Für jeden eingehenden RCC der Tabelle:

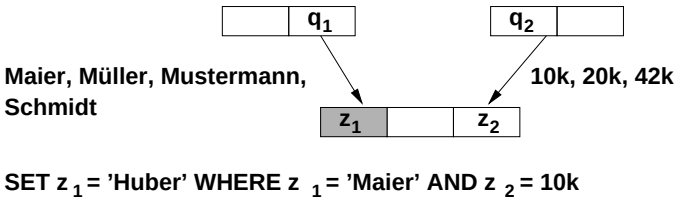
Abbildung 6.3 RCC-Wert-Invalidierung: Beispiel 2



- Falls auf der Zielspalte des eingehenden RCC eine Änderung stattfand: Sperre *old_value* und *new_value* an diesem RCC.
- Falls auf der Zielspalte des eingehenden RCC keine Änderung stattfand: Projiziere das Before-Image auf die Spalte mit dem eingehenden RCC. Invalidiere den durch die Projektion erhaltenen Wert w_i am eingehenden RCC.

In Abbildung 6.4 ist ein Beispiel mit mehreren eingehenden RCCs gegeben. Wiederum wird ein Wert aus Spalte z_1 geändert. Entsprechend müssen am RCC $q_1 \rightarrow z_1$ die Werte $Maier$ und $Huber$ invalidiert werden. Es gibt noch eine weiteren eingehenden RCC auf Spalte z_2. Aus der Anfrage geht hervor, dass das geänderte Tupel in dieser Spalte den Wert $10k$ hat. Um das Tupel über alle eingehenden RCCs unerreichbar zu machen, muss an diesem RCC der Wert $10k$ invalidiert werden.

Abbildung 6.4 RCC-Wert-Invalidierung: Beispiel 3



Die Invalidierungen auf den eingehenden RCCs sorgen dafür, dass veraltete Datensätze unerreichbar sind. Der verbleibende zu untersuchende Fall ist der, wenn auf der Spalte, des von der Änderung betroffenen Wertes, ein ausgehender RCC existiert. In Abschnitt 6.2 wurde bereits aufgezeigt, dass die Konsistenz des Caches verletzt sein kann, falls *new_value* bis jetzt nicht in dieser Spalte existiert hat. Ist also *new_value* nicht vorhanden, dann muss *new_value* an dem ausgehenden RCC invalidiert werden. Andernfalls könnte während des Probing fälschlicherweise gefolgert werden, das *new_value* in der Zielspalte vollständig ist. Eine Inkonsistenz liegt

in diesem Fall nur dann nicht vor, falls *new_value* überhaupt nicht in der Zielspalte der Backend-Tabelle vorkommt. Ob tatsächlich eine Inkonsistenz vorliegt, kann dann jedenfalls nur unter Einbeziehung der Backend-Datenbank ermittelt werden. Tritt hingegen *new_value* bereits in der Quellspalte der Cache-Tabelle auf, dann ist er bereits vollständig in der Zielspalte.

Zusammenfassend ergibt sich damit folgender Invalidierungs-Algorithmus:

1. Falls die Tabelle keine eingehenden RCCs hat: Abbruch. Es ist nichts weiter zu tun, weil die Tabelle unsichtbar ist.
2. Für jeden eingehenden RCC der Tabelle:
   - Falls auf der Zielspalte des eingehenden RCC eine Änderung stattfand: Sperre *old_value* und *new_value* an diesem RCC.
   - Falls auf der Zielspalte des eingehenden RCC keine Änderung stattfand: Projiziere das Before-Image auf die Spalte mit dem eingehenden RCC. Invalidiere den durch die Projektion erhaltenen Wert w_i am eingehenden RCC.
3. Für alle ausgehenden RCCs der Tabelle:
   - Falls auf der Quellspalte des ausgehenden RCC eine Änderung stattfand und *new_value* in der Spalte der Cache-Tabelle nicht existiert: Invalidiere *new_value* an diesen RCCs.

6.3.2 Invalidierung von RCCs

Die zweite Vorgehensweise, um veraltete Daten unerreichbar zu machen, ist nach den vorausgegangenen Überlegungen vergleichsweise naiv. Wurde ein Datensatz geändert, so sind alle eingehenden RCCs der Tabelle dieses Datensatze zu sperren. Damit ist der Datensatz unerreichbar und fehlerhafte Joins mit der Cache-Tabelle werden vermieden. Betraf die Änderung eine Spalte mit ausgehenden RCCs, so sind diese ebenfalls zu sperren. Damit wird ausgeschlossen, dass fälschlicherweise auf Wertvollständigkeit in den Zielspalten geschlossen wird.

Man kann nun argumentieren, dass durch diese Vorgehensweise zu viele Datensätze unerreichbar sind. Im Worst Case kann der gesamte Cache zeitweilig außer Funktion gesetzt sein, wenn beispielsweise eine Transaktion Änderungen auf allen Tabellen der Cache Group vorgenommen hat.

Andererseits zeichnet sich dieses Verfahren durch Schnelligkeit und Einfachheit aus. Die zu invalidierenden RCCs können schnell und direkt aus einem Write Set ermittelt werden. Es gibt wenige, aber dafür umso restriktivere Invalidierungen. Im Verlauf dieses Kapitels werden sich noch weitere Vorteile dieses simplen Verfahrens herausstellen. Bei der RCC-Wert-Invalidierung können beispielsweise Write Skews auftreten, die entsprechend maskiert werden müssen.

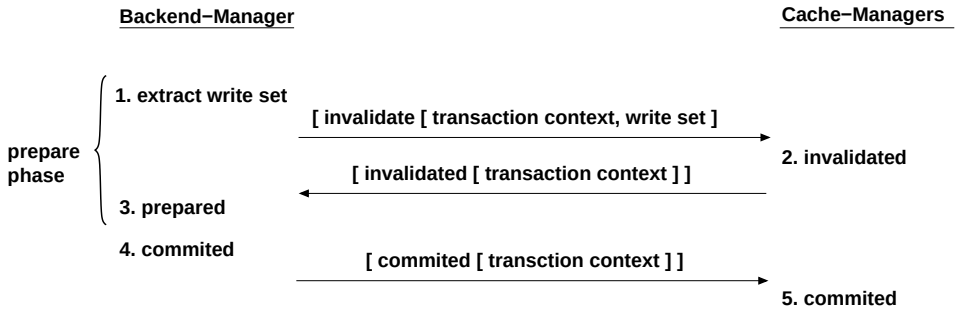
6.4 ACCache-Commit-Protokoll

Aufgrund der Replikation vergrößert sich im Allgemeinen die Dauer von schreibenden Transaktionen. Ziel des Verfahrens ist es, diese Erhöhung so gering wie möglich zu halten, damit Blockierungen möglichst vermieden werden. In Abbildung 6.5 (a) ist das Commit-Protokoll anschaulich dargestellt. Analog zum 2-Phase-Commit-Protokoll gibt es unmittelbar vor dem endgültigen Commit eine *Prepare-Phase*. Wenn eine schreibende Transaktion die Commit-Operation aufruft, dann ermittelt das Backend das Write Set dieser Transaktion. Anschließend wird an jeden Cache-Manager eine Nachricht geschickt, mit der Aufforderung die Invalidierungen, die sich aus den Änderungen der schreibenden Transaktion ergeben, durchzuführen. Die Nachricht enthält den globalen Transaktionskontext und das vollständige Write Set der Transaktion. Auf Grundlage des Write Sets ermittelt jeder Cache die für seine Cache Group erforderlichen Invalidierungen. Sobald der Backend-Manager von allen Caches die Nachricht erhalten hat, dass die Invalidierungen durchgeführt worden sind, ist die Prepare-Phase abgeschlossen und die Commit-Operation kann auf der Backend-Datenbank ausgeführt werden. Da innerhalb der Prepare-Phase alle nötigen Invalidierungen vorgenommen werden, wird die Prepare-Phase auch als *Invalidierungs-Phase* bezeichnet. Die abschließende *commited*-Nachricht muss in jedem Fall erfolgen, damit die Caches das Write Set der Transaktion anwenden. Es kann in der Prepare-Phase noch jederzeit dazu kommen, dass der Backend-Manager die Transaktion abbricht. Dies kann zum Beispiel eintreten, wenn auf der Backend-Datenbank ein Schreibkonflikt festgestellt wird oder, weil die *invalidated*-Nachricht eines Caches nicht eintrifft und dann aufgrund eines Timeouts abgebrochen werden muss. In diesem Fall wird eine *aborted*-Nachricht verschickt (siehe Abbildung 6.5 (b)).

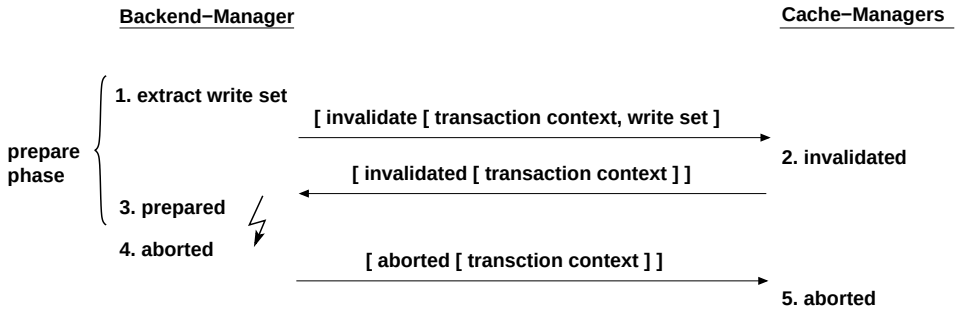
Die Kosten für eine Commit-Operation ergeben sich aus den Kosten für die Extraktion des Write Sets, plus drei Nachrichten, bzw. drei mal Latenz zu dem Cache, der am langsamsten angebunden ist. Hinzu kommen die Kosten für das Ermitteln und Durchführen der Invalidierungen auf den Caches. Da Letzteres parallelisiert werden kann, ist hierbei wiederum der Cache, der am längsten braucht, maßgeblich. Das eigentliche Bestimmen und Anwenden der Invalidierungen kann in jedem Fall vollständig durch Hauptspeicheroperationen abgewickelt werden, und verursacht keine teuren IO-Vorgänge. Die Kosten für das Bestimmen des Write Sets hingegen hängen stark von der Implementierung und vom Backend-Datenbanksystem ab. Sie ist jedoch im Allgemeinen mit IO-Aufwand verbunden. Manche Datenbank-Hersteller ermöglichen es asynchron auf das Redo-Log und Undo-Log der Transaktion per API zuzugreifen. Dadurch kann das Write Set sehr effizient ermittelt werden. Bei dem Datenbanksystem Oracle ist dies allerdings erst nach Commit möglich.

Eine weitere Alternative wäre es, das Ermitteln der Invalidierungen dem Backend zu überlassen. Dazu ist es erforderlich, dass das Backend alle RCCs aller Caches kennt. Zum Beispiel indem der Backendmanager eine “große” Cache Group verwaltet, die die Vereinigung der Cache Groups aller Caches darstellt. Dies hat den Vorteil, dass

Abbildung 6.5 Das Commit-Protokoll



(a) Erfolgreiches Commit



(b) Abbruch der Transaktion

Invalidierungen, die auf mehreren Caches angewendet werden müssen, nur einmal berechnet werden. Es ist weiterhin davon auszugehen, dass die Invalidierungen erheblich weniger Traffic verursachen, als das Write Set. Es entsteht allerdings zusätzliche Last auf dem Backend und das Write Set müsste dann nach dem endgültigen Commit sowieso an alle Caches geschickt werden.

6.5 Globales Transaktionsmanagement

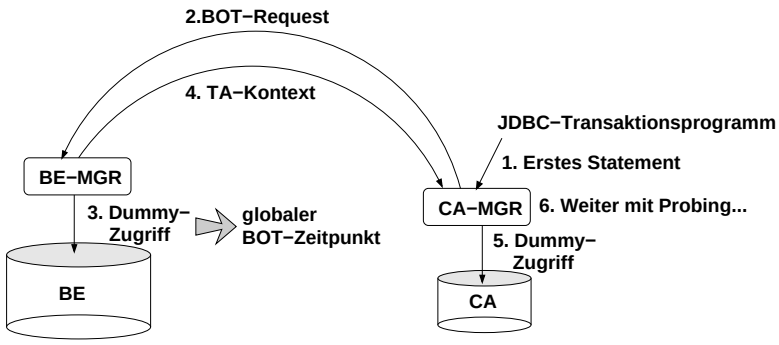
Bei einer ACCache-Anwendungstransaktion handelt es sich um eine globale Transaktion in einer verteilten Software-Architektur. Zwar sollen möglichst viele Operationen einer Transaktion vom Cache abgearbeitet werden, zumindestens aber die Update-Operationen müssen auf der Primärkopie des Backend-Managers ausgeführt werden. Somit können einer globalen ACCache-Transaktion zwei Datenbank-Verbindungen

zugeordnet werden: Eine zu der Cache-Datenbank und eine zu der Backend-Datenbank. Diese Verbindungen selbst stellen lokale Transaktionen auf der jeweiligen Datenbank dar. Beide lokalen Transaktionen besitzen eigene BOT- und Commit-Zeitpunkte. Einer globalen ACCache-Transaktion lässt sich also nicht direkt ein globaler BOT- und Commit-Zeitpunkt zuordnen. Für die Synchronisation der globalen Transaktionen ist dies aber nötig. Die genannten Konzepte ergeben sich aufgrund der Verteiltheit der globalen Transaktion nicht in natürlicher Art und Weise, sondern müssen künstlich nachgebildet werden. Welche Maßnahmen dazu getroffen werden müssen wird im im Folgenden gezeigt.

6.5.1 Globaler BOT-Zeitpunkt einer ACCache-Transaktion

Als erstes wird gezeigt, wie die beiden BOT-Zeitpunkte der lokalen Transaktionen mit dem globalen BOT-Zeitpunkt der ACCache-Transaktion zusammenhängt. Der globale BOT-Zeitpunkt bestimmt den globalen Snapshot, von dem die ACCache-Transaktion während ihrer gesamten Dauer liest. Dazu muss zunächst einmal sichergestellt werden, dass die Datenversionen, die für eine beliebige Transaktion bei ihrem globalen BOT gültig waren, bis zu ihrem globalen EOT aufgehoben werden. Da der Cache immer nur eine Teilmenge der Daten speichert, muss insbesondere sichergestellt werden, dass die entsprechenden Versionen in der Backend-Datenbank gehalten werden. Deshalb muss für jede ACCache-Transaktion ein entsprechender globaler Transaktionskontext auf dem Backend vorhanden sein, um zu verhindern, dass die Backend-Datenbank Versionen verwirft, die später noch benötigt werden.

Abbildung 6.6 Globaler BOT einer ACCache-Transaktion



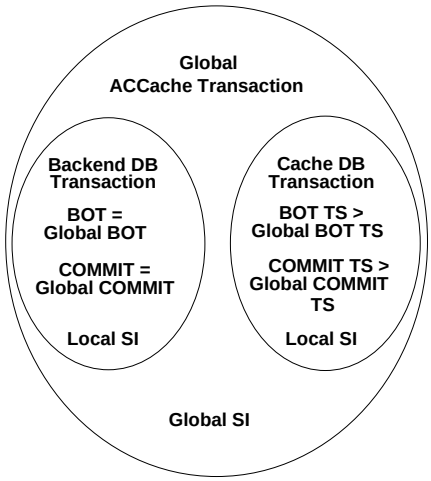
Bei JDBC-Transaktionen löst das erste Statement implizit die BOT-Operation aus. Bevor das erste Statement ausgeführt wird, muss also auf jeden Fall eine Transaktion

auf der Backend-Datenbank gestartet werden, damit, falls nebenläufige Schreiber Daten ändern, die veralteten Versionen aufgehoben werden. Es gibt dann jeweils einen BOT-Zeitstempel auf der Cache-Datenbank, und einen auf der Backend-Datenbank. Einzige Ausnahme zu dieser Regel sind sogenannte *Ad-hoc Queries*.

Zur Herstellung eines globalen BOT-Zeitpunktes muss also das erste Statement einer ACCache-Transaktion gesondert behandelt werden. In Abbildung 6.6 ist schematisch dargestellt, wie beim ersten Statement vorgegangen wird. Bevor das Statement verarbeitet wird, wird zunächst eine Nachricht ans Backend geschickt, dass eine neue globale Anwendungstransaktion gestartet werden soll. Der Backend-Manager startet dann eine eigene Transaktion auf der Backend-Datenbank, indem er über JDBC auf eine Dummy-Tabelle zugreift. Dieser Zugriff markiert dann den BOT-Zeitstempel auf der Backend-Datenbank. Gleichzeitig soll dies aber auch der globale BOT-Zeitpunkt sein. Dies ist sinnvoll, da sowieso alle Transaktionen auf der Backend-Datenbank synchronisiert werden. Der Backend-Manager erzeugt außerdem einen globalen Transaktionskontext und ordnet die lokale Transaktion dem globalen Kontext zu. Anschließend sendet das Backend den globalen Transaktionskontext an den Cache-Manager, den dieser ab sofort dem Anwendungstransaktionsprogramm zuordnet. Über den globalen Transaktionskontext wird sichergestellt, dass der globalen Anwendungstransaktion immer die gleiche vom Backend-Manager gestartete Transaktion auf der Primärkopie zugeordnet wird. Anschließend startet auch der Cache-Manager eine Transaktion auf der Cache-Datenbank und ordnet sie dem globalen Transaktionskontext zu. Auch dies geschieht implizit durch einen Dummy-Zugriff auf die Cache-Datenbank. Dieser markiert dann entsprechend den BOT-Zeitpunkt auf der Cache-Datenbank. Anschließend fährt der Cache-Manager, unter Benutzung des globalen Transaktionskontextes, wie gewohnt mit der Auswertung des Statements fort.

Ein Nachteil dieser Vorgehensweise ist, dass zum einen der Backend-Manager sehr viele offene Transaktionskontexte verwalten muss. Nach der Analyse aus Kapitel 5.2.1 handelt es sich dabei um $transactions\_eager$-viele bei $nodes - 1$ Caches (siehe Gleichung 5.7). Da die Caches alle Daten mit hoher Lokalität vorhalten und von einem Anteil von 20% Schreibern ausgegangen wird, erzeugen diese Transaktionen aber nicht viel Last auf dem Backend. Außerdem kostet die Operation zweimal Latenz. Die Transaktion müsste dann mindestens drei Anfragen ausführen, die ausschließlich vom Cache beantwortet werden können, um den Overhead der BOT-Operation in Bezug auf Antwortzeit auszugleichen und eine Verbesserung zu erzielen. Ansonsten ist der direkte Zugriff auf das Backend genauso schnell oder schneller. Allerdings sollte bei einer solchen Analyse nicht außer Acht gelassen werden, dass ein primäres Ziel des Datenbank-Caching Skalierbarkeit darstellt. Denn durch die Caches kann das Backend erheblich entlastet werden (80% lesende Transaktionen), sodass ein viel höherer Transaktionsdurchsatz insgesamt erreicht werden kann. Gleichzeitig schützt das Synchronisationsverfahren, durch die halbsynchrone Strategie zum Propagieren von Änderungen, seine Anwender vor stale data.

Abbildung 6.7 Globale ACCache-Transaktion



6.5.2 Globaler Commit-Zeitpunkt einer ACCache-Transaktion

Analog zum globalen BOT-Zeitpunkt entscheidet auch das Backend über den globalen Commit-Zeitpunkt. Jede Transaktion, welche die Invalidierungs-Phase abgeschlossen hat, kann die Commit-Operation auf der Backend-Datenbank ausführen. Der Zeitpunkt dieser Operation entspricht dem globalen Commit-Zeitpunkt. Anschließend wird an alle Caches die Nachricht geschickt, dass die Transaktion erfolgreich die Commit-Operation ausgeführt hat. Der Cache, auf dem die Transaktion gestartet wurde, führt ebenfalls die Commit-Operation auf seiner lokalen Transaktion aus. Jeder Cache kann dann asynchron beginnen das Write Set der Transaktion anzuwenden. Wenn dies abgeschlossen ist und alle eventuell entstandenen Inkonsistenzen beseitigt wurden, können die Invalidierungen der Transaktion wieder aufgehoben (oder: revalidiert) werden. Deshalb wird die Phase unmittelbar nach dem endgültigen Commit bis zum Aufheben der Invalidierungen als *Revalidierungs-Phase* bezeichnet. Das Aufheben der Invalidierungen markiert den *Revalidierungs*-Zeitpunkt. Dies ist der Zeitpunkt zu dem die Änderungen der Transaktion auch auf dem Cache atomar sichtbar werden.

6.5.3 Lebenszyklus einer ACCache-Transaktion

Insgesamt ergibt sich damit folgendes Transaktionsmanagement: Jeder globalen Anwendungstransaktion wird genau eine vom Backend-Manager verwaltete Transaktion auf der Backend-Datenbank und genau eine vom Cache-Manager verwaltete

Transaktion auf der Cache-Datenbank zugeordnet (vgl. Abbildung 6.7). Das Isolation Level der beiden lokalen Transacktionen muss mindestens Snapshot Isolation sein. Dies bedeutet, dass das zur lokalen Synchronisation eingesetzte Verfahren auf einem Mehrversionenverfahren basieren muss, welches außerdem alle Schreibkonflikte behandelt. Der für die Anwendungstransaktion globale BOT- und Commit-Zeitpunkt wird durch den BOT- bzw. Commit-Zeitstempel der Backend-Transaktion bestimmt.

Außerdem ergibt sich der in Abbildung 6.8 dargestellte Lebenszyklus für eine ACCache-Transaktion. Die einzelnen Phasen der Transaktion werden durch bestimmte Zeitpunkte eingeleitet oder beendet. Die für die Zeitpunkte verwendeten, Abkürzungen sind in Tabelle 6.1 aufgeschlüsselt.

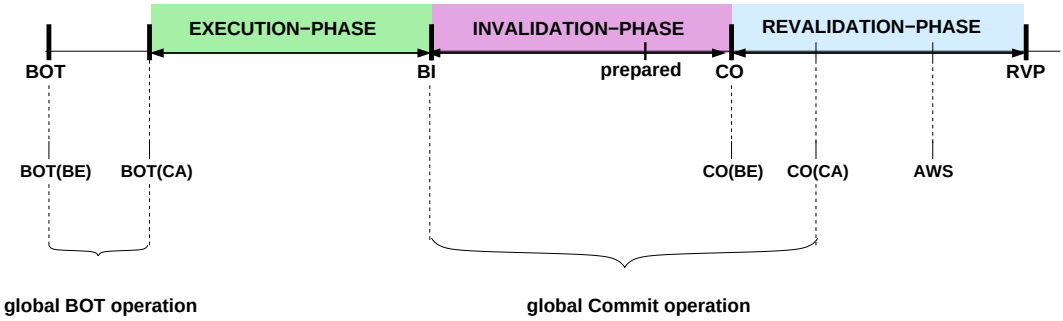
Tabelle 6.1 Abkürzungen für die wichtigen Zeitpunkte im Lebenszyklus einer ACCache-Transaktion

BOT	Globaler BOT-Zeitpunkt der ACCache-Transaktion
BOT(BE)	BOT-Zeitpunkt der lokalen Transaktion auf der Backend-Datenbank
BOT(CA)	BOT-Zeitpunkt der lokalen Transaktion auf der Cache-Datenbank
BI	Start der Invalidierungs-Phase
CO	Globaler CO-Zeitpunkt der ACCache-Transaktion
CO(BE)	CO-Zeitpunkt der lokalen Transaktion auf der Backend-Datenbank
CO(CA)	CO-Zeitpunkt der lokalen Transaktion auf der Cache-Datenbank
AWS	Zeitpunkt, zu dem das Write Set der ACCache-Transaktion vollständig auf einem Cache eingebracht ist
RVP	Revalidierungs-Zeitpunkt der ACCache-Transaktion auf einem Cache

Bevor die Transaktion mit dem Ausführen von Client-Anfragen beginnen kann, muss sie die globale BOT-Operation ausführen. Diese erfordert das Starten von zwei lokalen Transaktionen (vgl. Abschnitt 6.5.1). Anschließend wechselt die Transaktion in die Ausführungs-Phase. Dies ist die Phase in der die Transaktion Anfragen des Clients in ihrem globalen Kontext ausführt. Zum Zeitpunkt *BI* fordert der Client die globale Commit-Operation an. Damit beginnt für die Transaktion die Invalidierungs-Phase: Extraktion des Write Set durch den Backend-Manager, warten bis alle Caches die benötigten Invalidierungen vorgenommen haben. Sobald alle Caches das Durchführen der Invalidierungen bestätigt haben, ist die Transaktion global im Zustand *prepared* und kann die Commit-Operation auf der Backend-Datenbank ausführen. Die Transaktion befindet sich jetzt im Zustand *commited* und wechselt in die Revalidierungs-Phase. Die globale Commit-Operation ist allerdings erst beendet nachdem auch die lokale Transaktion zum Zeitpunkt *CO(CA)* auf der Cache-Datenbank beendet wurde. Denn erst nachdem auch diese lokale Transaktion beendet

wurde erhält der Client die Nachricht, dass die globale Commit-Operation erfolgreich ausgeführt wurde.

Abbildung 6.8 Lebenszyklus einer ACCache-Transaktion



Die Revalidierungs-Phase schließt mit dem Revalidierungs-Zeitpunkt *RVP* ab. Zwischen dem Beginn der Revalidierungs-Phase *CO* und *RVP* liegt noch der Zeitpunkt *AWS*. Dieser markiert den Zeitpunkt, zu dem alle Änderungen des Write Set vollständig in die Cache-Datenbank eingebracht sind.

Zur besseren Verdeutlichung sind außerdem die BOT- und Commit-Zeitpunkte der lokalen Transaktionen eingezeichnet. Dabei fallen der globale BOT- und Commit-Zeitpunkt der ACCache-Transaktion mit denen der lokalen Backend-Transaktion zusammen. Dies wurde oben bereits ausführlich dargestellt. Im nächsten Abschnitt wird nachgewiesen wie trotz dieser Voraussetzungen, globale Snapshot Isolation erreicht werden kann.

6.5.4 Globale Snapshot Isolation einer ACCache-Transaktion

Weil die BOT-Zeitpunkte auf der Cache- und Backend-Datenbank nicht identisch sind, muss noch gezeigt werden, dass die Anwendungstransaktion zu jedem Zeitpunkt die korrekten Datenversionen ihres globalen Snapshot sieht. Da alle Änderungen auf der Primärkopie stattfinden müssen, entspricht der globale Snapshot einer ACCache-Transaktion dem jüngsten transaktionskonsistenten Zustand der Backend-Datenbank zu ihrem globalen BOT-Zeitpunkt. Der globale BOT-Zeitpunkt einer ACCache-Transaktion entspricht wiederum dem BOT-Zeitpunkt $BOT(BE)$ auf der Backend-Datenbank. Daraus folgt, dass der globale Snapshot einer ACCache-Transaktion stets identisch ist mit dem Snapshot der Backend-Datenbank zum Zeitpunkt $BOT(BE)$ (vgl. Definition 6.2).

Definition 6.2 Globaler Snapshot einer ACCache-Transaktion

Sei $BOT(BE)$ der BOT-Zeitpunkt der lokalen Backend-Transaktion einer ACCache-Transaktion. Der globale Snapshot dieser ACCache-Transaktion ist identisch mit dem Snapshot der Backend-Datenbank zum Zeitpunkt $BOT(BE)$.

Somit sieht eine ACCache-Transaktion auf der Backend-Datenbank immer die korrekten Datenversionen ihres globalen Snapshot. Es muss jedoch gezeigt werden, dass die Transaktion auch auf der Cache-Datenbank immer die Datenversionen dieses Snapshot sieht. Immer dann, wenn dies nicht gewährleistet werden kann, muss der Cache an das Backend delegieren.

Sei also T_1 eine beliebige Anwendungstransaktion, die gerade die globale BOT-Operation durchführt. Der BOT auf der Backend-Datenbank finde zum Zeitpunkt $BOT_{T_1}(BE)$, und der BOT auf der Cache-Datenbank zum Zeitpunkt $BOT_{T_1}(CA)$ statt. Sei weiterhin T_2 eine schreibende Transaktion, die zum Zeitpunkt $CO_{T_2}(BE)$ die Commit-Operation auf der Backend-Datenbank ausführt. Es müssen drei Fälle unterschieden werden:

1.Fall: $CO_{T_2}(BE) < BOT_{T_1}(BE)$ (vgl. Abbildung 6.9)

Daraus folgt, dass die Änderungen von T_2 im globalen Snapshot der Transaktion T_1 enthalten sind.

Zu zeigen ist nun, dass T_1 auf dem Cache alle Änderungen von T_2 sieht

1. Fall: Bei $BOT_{T_1}(CA)$ wurde das Write Set von T_2 bereits vollständig in die Cache-Datenbank eingebracht. Dann sieht T_1 die geänderten Datenversionen auf der Cache-Datenbank.
2. Fall: Bei $BOT_{T_1}(CA)$ ist das Write Set von T_2 noch nicht in die Cache-Datenbank eingebracht. Dann sind bei $BOT_{T_1}(CA)$ die Invalidierungen von T_2 noch nicht aufgehoben und T_1 muss zunächst alle von T_2 geänderten Daten von der Backend-Datenbank lesen. Auf der Backend-Datenbank sieht T_1 die geänderten Datenversionen.

$\Rightarrow T_1$ sieht alle Änderungen von T_2 auf dem Cache.

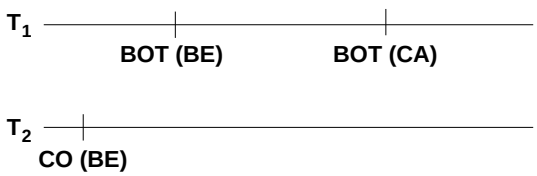
2. Fall: $CO_{T_2}(BE) > BOT_{T_1}(CA) > BOT_{T_1}(BE)$ (vgl. Abbildung 6.10)

Aus $CO_{T_2}(BE) > BOT_{T_1}(BE)$ folgt, dass alle Änderungen von T_2 nicht im globalen Snapshot der Transaktion T_1 enthalten sind.

Zu zeigen ist nun, dass T_1 auf dem Cache keine Änderungen von T_2 sieht

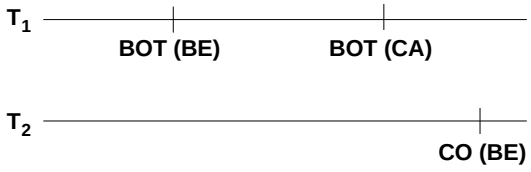
Das Anwenden des Write Sets von T_2 kann auf der Cache-Datenbank erst nach dem globalen Commit von T_2 stattfinden. Das heißt die Änderungen von T_2

Abbildung 6.9 Globale Snapshot Isolation: Fall 1



werden erst nach dem Zeitpunkt $CO_{T_2}(BE)$ auf der Cache-Datenbank wirksam. Weil außerdem $BOT_{T_1}(CA) < CO_{T_2}(BE)$ gilt, sind die Änderungen des Write Set in keinem Fall in dem Snapshot der lokalen Cache-Transaktion enthalten. T_1 sieht also keine Änderungen aus dem Write Set von T_2 auf der Cache-Datenbank.
$\Rightarrow T_1$ sieht keine Änderungen von T_2 auf dem Cache.

Abbildung 6.10 Globale Snapshot Isolation: Fall 2



3.Fall: $BOT_{T_1}(BE) < CO_{T_2}(BE) < CO_{T_2}(CA)$ (vgl. Abbildung 6.11)

Aus $BOT_{T_1}(BE) < CO_{T_2}(BE)$ folgt, dass die Änderungen von T_2 nicht im globalen Snapshot der Transaktion T_1 enthalten sind.

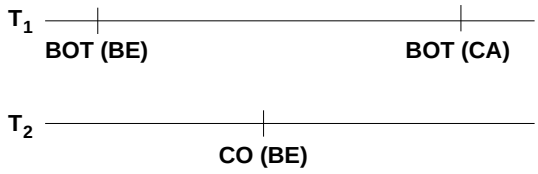
Zu zeigen ist nun, dass T_1 auf dem Cache keine Änderungen von T_2 sieht

1. Fall: Bei $BOT_{T_1}(CA)$ hat T_2 die Revalidierungs-Phase auf dem Cache bereits abgeschlossen. Dann wurde das Write Set von T_2 bereits vollständig in die Cache-Datenbank eingebracht und alle Invalidierungen wurden aufgehoben. Nun sieht T_1 die geänderten Datenversionen auf der Cache-Datenbank. Dies ist ein Fehler. Es muss sichergestellt werden, dass die Invalidierungen von T_2 erst nach $BOT_{T_1}(CA)$ aufgehoben werden. Denn solange diese gültig sind, muss T_1 alle von T_2 geänderten Daten von der Backend-Datenbank lesen. Auf der Backend-Datenbank sieht T_1 die alten Datenversionen.
2. Fall: Bei $BOT_{T_1}(CA)$ hat T_2 die Revalidierungs-Phase noch nicht abgeschlossen. Dann sind bei $BOT_{T_1}(CA)$ die Invalidierungen von T_2 noch

nicht aufgehoben und T_1 muss zunächst alle von T_2 geänderten Daten von der Backend-Datenbank lesen. Auf der Backend-Datenbank sieht T_1 die geänderten Datenversionen.

⇒ Wenn gewährleistet wird, dass die Invalidierungen der Transaktion T_2 erst nach $BOT_{T_1}(CA)$ aufgehoben werden, sieht T_1 die alten Datenversionen.

Abbildung 6.11 Globale Snapshot Isolation: Fall 3



Damit wurde gezeigt, dass – mit gewissen Vorkehrungen – eine ACCache-Transaktion immer genau die Datenversionen ihres globalen Snapshot sieht. Weiterhin wurde gezeigt, dass die Commit- und BOT-Operationen verzahnt ablaufen können. Es bedarf dabei keiner Koordination zwischen Cache und Backend. Allerdings muss jeder Cache-Manager die BOT-Operationen mit dem Aufheben von Invalidierungen synchronisieren. Zwischen $BOT(BE)$ und $BOT(CA)$ dürfen keine Invalidierungen von Transaktionen aufgehoben werden, deren Commit-Operation ebenfalls in dieser Zeitspanne liegt. Eine mögliche Lösung wäre das Aufheben von Invalidierungen während

Beobachtung 6.1 Synchronisation von globaler BOT- und Commit-Operation

Globale BOT- und Commit-Operationen müssen nicht synchronisiert werden, solange für jede schreibende Transaktion T_1 der Revalidierungs-Zeitpunkt solange blockiert wird, bis keine weitere Transaktion T_2, mit $BOT_{T_2}(BE) < CO_{T_1}(BE)$, die globale BOT-Operation auf dem Cache durchführt.

einer aktiven BOT-Operation grundsätzlich zu verbieten. Dies ist allerdings sehr restriktiv. Besser wäre es den Revalidierungs-Zeitpunkt einer Transaktionen T_2 solange hinauszuzögern, bis keine BOT-Operation einer weiteren Transaktion T_1, mit $BOT_{T_1}(BE) < CO_{T_2}(BE)$, mehr aktiv ist (vgl. Beobachtung 6.1). Dies lässt sich leicht über einen Pointer auf die älteste BOT-Anforderung eines Clients implementieren. Immer dann, wenn eine globale BOT-Operation abgeschlossen wird, ändert sich potentiell dieser Pointer. Und wenn sich der Pointer ändert, kann überprüft werden, ob Invalidierungen aufgehoben werden können. Konkret: **Die Transaktion T_2 kann sicher revalidieren, falls der älteste BOT-Anforderungs-Zeitpunkt $> CO_{T_2}(BE)$ ist.**

6.6 Gültigkeit von Invalidierungen

Im Folgenden soll näher untersucht werden, wie sich Invalidierungen und Revalidierungen auf aktive Transaktionen auswirken. Dazu wird folgende These aufgestellt: Aufgrund der Isolierung des zugrundeliegenden Datenbanksystems muss jede Transaktion T während ihrer gesamten Laufzeit, ausschließlich die zum Zeitpunkt $BOT_T(CA)$ gültigen Invalidierungen beachten (vgl. Beobachtung 6.2). $BOT_T(CA)$ bezeichne wie bisher, den BOT-Zeitpunkt der lokalen Transaktion auf der Cache-Datenbank. Während der Laufzeit hinzukommende Invalidierungen können ignoriert werden. Vom Cache vorgenommene Revalidierungen können nicht übernommen werden. Von den Revalidierungen profitieren ausschließlich diejenigen Transaktionen, die neu starten. Dabei müssen zwei Fälle unterschieden werden:

Beobachtung 6.2 Gültigkeit von Invalidierungen für eine ACCache-Transaktion

Sei T eine ACCache-Transaktion und darüber hinaus $BOT_T(CA)$ der BOT-Zeitpunkt ihrer lokalen Transaktion auf der Cache-Datenbank.
Dann sieht T ausschließlich die Datenversionen ihres globalen Snapshot, wenn T während ihrer gesamten Ausführungsphase alle zum Zeitpunkt $BOT_T(CA)$ gültigen Invalidierungen beachtet.

---

1. Die Invalidierungen einer Transaktion T_1 seien zum Zeitpunkt $BOT_{T_2}(CA)$ einer anderen Transaktion T_2 gültig. Während der Laufzeit von T_2 werden diese revalidiert. Wie wirkt sich das auf T_2 aus?
2. Eine Transaktion T_1 invalidiert nach $BOT_{T_2}(CA)$ von Transaktion T_2. Wie wirken sich die Invalidierungen auf T_2 aus?

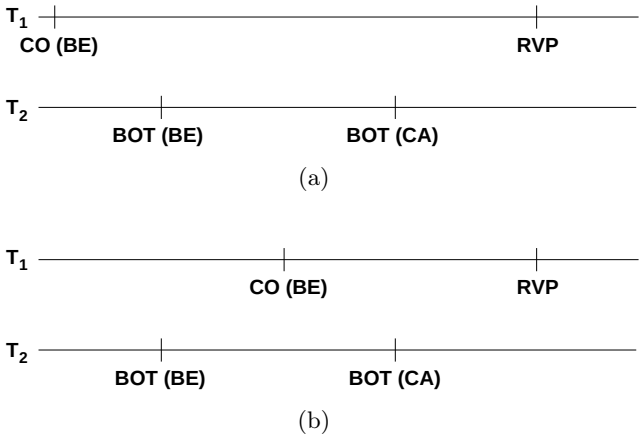
1. Fall Es gilt dann: $CO_{T_1}(BE) < BOT_{T_2}(CA) < RVP_{T_1}$.

Dafür gibt es nur zwei mögliche Abläufe (vgl. Abbildung 6.12):

- Ablauf (a): $CO_{T_1}(BE) < BOT_{T_2}(BE)$
  Dies bedeutet, dass alle Änderungen von T_1 in dem globalen Snapshot von T_2 enthalten sind. Da das Einbringen der Änderungen in der Cache-Datenbank erst zum Zeitpunkt $RVP_{T_1} > BOT_{T_2}(CA)$ erfolgt, sind die Änderungen von T_1 nicht im Snapshot der lokalen Cache-Transaktion von T_2. Die Transaktion sieht Änderungen von T_1 also auch nach deren Einbringung durch das Write Set nicht auf der Cache-Datenbank.
  $\Rightarrow$ T_1 muss die Invalidierungen weiterhin beachten und die geänderten Daten vom Backend lesen.
- Ablauf (b): $CO_{T_1}(BE) > BOT_{T_2}(BE)$
  Daraus folgt, dass alle Änderungen von T_1 nicht in dem globalen Snapshot von T_2 enthalten sind. Somit wäre das Beachten der Invalidierungen

durch T_1 niemals nötig gewesen. Allerdings führt das Beachten der Invalidierungen auch nicht zu einem Synchronisationsfehler.

Abbildung 6.12 Gültigkeit von Invalidierungen: Nebenläufige Revalidierungen



2. Fall Es gilt dann: $CO_{T_1}(BE) > BOT_{T_2}(CA) > BOT_{T_2}(BE)$.

Dafür gibt es nur einen möglichen Ablauf (vgl. Abbildung 6.13):

Aus $CO_{T_1}(BE) > BOT_{T_2}(CA)$ folgt direkt, dass die Änderungen von T_1 nicht im globalen Snapshot von T_2 enthalten sind. Sie können auch niemals im Snapshot der lokalen Cache-Transaktion sichtbar werden da, $RVP_{T_1} > CO_{T_1}(BE) > BOT_{T_2}(CA)$.
$\Rightarrow T_2$ muss die Invalidierungen, die nach $BOT_{T_2}(CA)$ gültig werden nicht beachten.

Abbildung 6.13 Gültigkeit von Invalidierungen: Nebenläufige Invalidierungen

Damit ist die Behauptung bewiesen: Es genügt, wenn der Cache-Manager jeweils zum Zeitpunkt $BOT(CA)$ ermittelt, welche RCC-Werte bis zu diesem Zeitpunkt ungültig waren und diese mit der Transaktion assoziiert. **Die Transaktion muss während ihrer Laufzeit maximal diese Invalidierungen beachten. Neu hinzukommende Invalidierungen betreffen sie aufgrund der Isolierung, die durch**

die zugrundegelegten Datenbanksysteme bereits vorgenommen werden, nicht (vgl. Beobachtung 6.2).

6.6.1 Gesonderte Behandlung von schreibenden Transaktionen

In den beiden vorangegangenen Abschnitten wurde gezeigt, wie das Verfahren sicherstellt, dass jede Transaktion jederzeit den zu ihrem BOT gültigen Snapshot sieht. Damit wurde auch das Auftreten von Non-Repeatable-Reads ausgeschlossen. Dabei wurde allerdings nicht beachtet, dass jede schreibende Transaktion ihre eigenen Änderungen sehen muss. Wenn eine Transaktion einen Datensatz schreibt, der auch im Cache ist, dann wird dieser Datensatz erst bei ihrem Commit im Cache unerreichbar. Würde die Transaktion den geänderten Datensatz nach der Schreiboperation nochmals lesen, so würde die Transaktion die ungeänderte Kopie aus dem Cache sehen. Dies ist selbstverständlich zu vermeiden, denn es widerspricht den Anforderungen einer globalen Snapshot Isolation.

Dazu ist es erforderlich das Write Set schritthaltend nach jeder Schreiboperation zu ermitteln und zu sammeln. Nach der erfolgreichen Ausführung einer Schreiboperation muss das Write Set dieser Operation dem Cache, auf dem die schreibende Transaktion läuft, mitgeteilt werden. Das Write Set der Update-Operation kann beispielsweise zusammen mit dem Return-Wert der Operation geschickt werden. Der Cache kann dann die Invalidierungen die aus diesem Write Set folgen, den bereits bestehenden der schreibenden Transaktion hinzufügen. Ab sofort greift dann die Transaktion auf ihre geänderten Daten nur noch über die Backend-Datenbank zu. Alle anderen Caches erhalten das vollständige Write Set erst in der Invalidierungs-Phase.

Zusätzlich zu den bei $BOT(CA)$ einmalig ermittelten Invalidierungen kommen bei Schreibern nach jeder Änderungsoperation, diejenigen Invalidierungen der jeweils geänderten Datensätze hinzu. Dies bedeutet, dass die Invalidierungen nicht auf einmal am Ende der Transaktion ermittelt werden, sondern schritthaltend. In der Prepare-Phase ist dann das bereits ermittelte Write Set an die restlichen Cache-Manager zu senden.

6.7 Die Revalidierungs-Phase

Eine ACCache-Transaktion befindet sich im Zustand *commited*, sobald auf der Backend-Datenbank erfolgreich die Commit-Operation abgeschlossen wurde. Direkt im Anschluss werden alle Cache-Manager über den erfolgreichen Ausgang der Operation informiert. Anhand des globalen Transaktionskontextes können die Cache-Manager

die Invalidierungen und das Write Set der Transaktion identifizieren. Die Cache-Manager können nun beginnen, die Änderungen asynchron zu übernehmen und entstehende Inkonsistenzen durch gezieltes Nachladen von Tupeln zu beseitigen. Sobald das Write Set der Transaktion vollständig angewandt wurde und alle entstandenen Inkonsistenzen beseitigt wurden, können alle Invalidierungen aufgehoben werden. Bei dem Aufheben der Invalidierungen muss darauf geachtet werden, dass nur diejenigen aufgehoben werden, die nicht bereits von anderen Transaktionen wiederholt angefordert wurden.

6.7.1 Asynchrones Anwenden des Write Sets

Bei der Konstruktion des Write Sets müssen einige Punkte beachtet werden. Eine SQL-Update-Anweisung kann gleichzeitig mehrere Tupel betreffen. Das Write Set sollte allerdings für jedes geänderte Tupel genau einen Eintrag enthalten. Dieser sollte zumindestens die folgenden Informationen enthalten:

- DML-Typ der SQL-Update-Operation (`INSERT`, `DELETE` oder `UPDATE`).
- Name der Backend-Tabelle
- Für jede `SQL-INSERT`-Anweisung alle Spaltenwerte des eingefügten Tupels
- Für jede `SQL-DELETE`-Anweisung ein Tupelidentifikator für das gelöschte Tupel
- Für jede `SQL-UPDATE`-Anweisung ein Tupelidentifikator für das geänderte Tupel und alle neuen Spaltenwerte des Tupels nach der Update-Operation
- Für `SQL-UPDATE`- oder `SQL-DELETE`-Anweisungen zusätzlich alle Spaltenwerte des Tupels vor der Update-Operation

Die alten Werte des Tupels werden zum Ableiten der Invalidierungen benötigt. Dies wurde bereits in Abschnitt 6.3.1 motiviert. Da für jede Tabelle in einer Cache Group ein Primärschlüssel angegeben werden muss, bietet sich der Primärschlüssel an. Bei Verwendung des Primärschlüssels als Identifikator ist allerdings Vorsicht geboten bei Update-Operationen auf Primärschlüsseln (vgl. Abschnitt 6.8.2). Manche Datenbanksysteme erlauben auch den Zugriff auf einen internen Tupelidentifikator. Dieser muss dann allerdings vom ACCache-System auf den Cache-Datenbanken selbst gepflegt werden.

Da jeder Write-Set-Eintrag mit der Änderung auf genau einem Tupel korrespondiert, kann aus jedem Eintrag ein SQL-Statement generiert werden, welches die alten Werte in dem Tupel blind mit den neuen Werten überschreibt. Dadurch kann ein vollständiges Write Set auf den Cache-Datenbanken blind geschrieben werden. Die Write Sets werden in der gleichen Reihenfolge angewendet, wie die Commit-Operationen auf der Backend-Datenbank auftraten. Auf jedem Cache befindet sich zu einem Zeitpunkt maximal ein Write Set in der Anwendung.

In Listing 6.1 ist ein Ausschnitt der Implementierung des Write-Set-Eintrages der SQL-UPDATE-Anweisung gegeben. Jeder Write Set-Eintrag ist ein Java-Objekt vom Typ `UpdateOperation`. Das Update-Objekt speichert im Attribut `tableName` den Namen der Backend-Tabelle, den Primärschlüssel des geänderten Tupels im Feld `primaryKey`, eine vollständige Liste der alten Werte des Tupels (`oldValues`) und eine Liste mit den neuen Werten (`newValues`). Das Speichern der Tupelwerte in einer `Map` erlaubt den Zugriff auf diese via Spaltenname. Der Primärschlüssel dient als Tupelidentifikator. In Listing 9.1.1 ist ein weiterer Ausschnitt aus dieser Klasse gegeben.

Listing 6.1 Implementierung des Write-Set-Eintrages einer SQL UPDATE-Anweisung

```
public class Update implements UpdateOperation {

        private Name tableName;

        private PrimaryKey primaryKey;

        private Map<Name, Object> newValues;

        private Map<Name, Object> oldValues;

//          . . .
}
```

Dabei handelt es sich um die Funktionalität zur Generierung des Statements, welches die Änderungen blind in das Tupel schreibt. Dazu wird ein `PreparedStatementData`-Objekt erzeugt. Dieses speichert die String-Repräsentation eines Java-`PreparedStatement` und dessen Parameter. Die Parameter werden so gespeichert, dass der Zugriff auf sie per `PreparedStatement`-Index möglich ist.

Bei der Ausführung dieser Statements ist jedoch zu beachten, dass es zu Wechselwirkungen mit dem allgemeinen Lade- und Entlademechanismus des Cache kommen kann. Diese werden im Folgenden für jeden DML-Typ einzeln untersucht.

INSERT-Anweisung Da die Write Sets auf den Caches asynchron, also nicht zum globalen Commit der Transaktion angewendet werden, liegt zwischen dem Sichtbarwerden des eingefügten Tupels auf der Backend-Datenbank und der Ausführung des `Insert`-Statements auf dem Cache eine Zeitspanne δ. In dieser Zeitspanne könnte eine Cache Unit geladen worden sein, die das neue Tupel enthält. Dann entsteht bei der Ausführung des Statement auf der Cache-Datenbank ein SQL-Fehler, weil bereits ein Tupel mit dem Primärschlüssel existiert. Der Fehler kann durch Abfrage des SQL-Error-Code maskiert werden. Allerdings muss dann eine für jeden Datenbanksystemhersteller spezifische Fehlerbehandlung

entwickelt werden. Eine andere Möglichkeit besteht darin, vorher eine Query auszuführen, die überprüft, ob das Tupel bereits geladen ist.

Je nach Implementierung des Entladevorgangs muss vorher überprüft werden, ob das einzufügende Tupel überhaupt eine Daseinsberechtigung hat (vgl. [Thi07]). Mit anderen Worten, es muss festgestellt werden, ob mindestens eine Cache Unit geladen ist, die dieses Tupel enthält. Ansonsten kann das einzufügende Tupel eine Datenleiche werden, die frühestens wieder entfernt werden kann, nachdem zufällig eine Cache Unit geladen worden ist, die dieses Tupel enthält.

Falls der Entlademechanismus so arbeitet, dass generell alle Tupel entfernt werden, die keine Daseinsberechtigung mehr haben, kann standardmäßig versucht werden das Tupel einzufügen. Gegebenenfalls wird dann das Tupel bei dem nächsten Entladevorgang wieder entfernt. Falls die Zeitspanne δ nicht zu groß ist, dann ist aus Sicht der Lokalität das Tupel allein schon deswegen berechtigterweise im Cache, weil ein schreibender Zugriff auf dieses Tupel vor δ Zeiteinheiten stattgefunden hat, und wonach die Wahrscheinlichkeit für einen erneuten Zugriff erhöht wäre. Es ist dann wiederum wahrscheinlicher, dass bereits eine Cache Unit geladen ist, die dieses Tupel enthält. Diese Vorgehenswiese kann sich nur negativ bemerkbar machen, falls es Transaktionen gibt, die Unmengen an neuen Tupel erzeugen. Dies ist normalerweise aber nur bei Data-Mining-Anwendungen, der Fall.

Muss die Daseinsberechtigung hingegen vorher überprüft werden, so verkompliziert und verteuert sie das Anwenden des Write Sets erheblich. Denn dann darf auch parallel zum Anwenden des Write Sets keinesfalls ein Lade- oder Entladevorgang aktiv sein. Sonst könnte es aufgrund von Synchronisationsfehlern sowieso zu Datenleichen kommen. Außerdem kann es dann auch passieren, dass die Daseinsberechtigung festgestellt wird, aber nachdem das Anwenden des Write Sets abgeschlossen ist, ein Entladevorgang startet, der das Tupel sofort wieder entfernt.

`DELETE`-Anweisung Das `DELETE`-Statement kann jederzeit ausgeführt werden. Falls das zu löschende Tupel nicht im Cache war, selektiert die `WHERE`-Klausel eine leere Menge zu löschender Tupel und nichts passiert.

Falls die Tabelle ausgehende RCCs hat, kann es passieren, dass größere Tupelmengen, die von den Werten des Tupel direkt oder indirekt referenziert wurden, ihre Daseinsberechtigung verlieren. Falls die Tabelle ausgehende RCCs auf Unique-Spalten hat, ist dies sogar sehr wahrscheinlich. Denn dann gibt es keine anderen Tupel in der Tabelle, die in der Spalte denselben Wert haben. Es gilt das gleiche wie oben, falls der Entlademechanismus nicht alle Tupel entfernt, die ihre Daseinsberechtigung verloren haben, verkompliziert sich das Anwenden einer `DELETE`-Anweisung nicht unerheblich. Es müssen dann nämlich alle referenzierten Tupel ohne Daseinsberechtigung entladen werden. Der Ausgangspunkt eines solchen Entladevorganges ist dann nicht mehr eine Füll-

spalte (vgl. [KMB], [Bra08]), sondern startet auf der Tabelle des gelöschten Tupel. Diese Funktionalität ist aber im gegenwärtigen Prototyp nicht implementiert, obwohl sie prinzipiell genauso arbeitet, wie das Entladen einer Cache Unit.

`UPDATE`-Anweisung Eine `UPDATE`-Anweisung kann ebenfalls blind ausgeführt werden. Falls das Tupel nicht geladen ist, hat dies keine Auswirkungen. Wenn das alte Tupel geladen ist, wird das Update angewendet. Für den Fall, dass bereits das neue Tupel geladen ist, wird das Update ein zweites Mal angewendet.

Die oben ausführlich beschriebenen Probleme mit den Datenleichen können hier an zwei Stellen auftreten. Zum einen, falls das Update auf einer Spalte mit eingehendem RCC stattfand. Dann ist es möglich, dass das Tupel die Daseinsberechtigung verliert, sofern der neue Wert nicht in der Quellspalte des eingehenden RCC vorkommt. Fand das Update auf einer Spalte mit ausgehenden RCC statt, können ebenfalls referenzierte Tupel ihre Daseinsberechtigung verlieren. Gegebenefalls müsste dann wiederum das Tupel gelöscht und entlang der ausgehenden RCC-Ketten alle unreferenzierten Tupel gelöscht werden.

Unter dem Blickwinkel der Revalidierungs-Phase sollte ein Entladevorgang immer auch alle unreferenzierten Tupel entfernen. Dies führt dazu, dass das Write Set blind angewendet werden kann. Einzig das Einfügen von Tupeln kann zu Fehlern führen falls das Tupel bereits geladen ist. Dieser Fehler kann jedoch relativ leicht behoben werden. Zwar kann ein Entladevorgang weiterhin von einer zu entfernenden Cache Unit getriggert werden, er sollte sich aber nicht ausschließlich auf die Cache Unit beschränken. Der Entladevorgang kann weiterhin gestartet werden, indem der Füllwert, der zu entladenden Cache Unit gelöscht wird. Dann werden alle von der Füllspalte aus erreichbaren atomaren Zonen, in topologisch sortierter Halbordnung, abgearbeitet (siehe [KMB],[Bra08]). In jeder erreichbaren Zone werden alle Tupel gelöscht, die von keinem RCC mehr referenziert werden. Dadurch werden die entladbaren Tupel der Cache Unit in jedem Fall gelöscht, aber auch alle anderen, die aus welchen Gründen auch immer, keine Daseinsberechtigung mehr haben. Ein solcher Entladevorgang, der gleichzeitig das Auftreten von Datenleichen verhindert ist wesentlich fehlertoleranter und beschleunigt das Anwenden von Write Sets.

Durch das Auftreten von Datenleichen kann es nicht zu Inkonsistenzen in der Cache Group kommen. Tupel ohne Daseinsberechtigung führen höchstens dazu, dass sich der Cache unnötig aufbläht. Die Tupel sind dann zwar in der Datenbank, aber kein Anwender kann auf sie zugreifen, da sie unerreichbar sind. Zwingend zu vermeiden sind allerdings Verletzung von RCC-Bedingungen. Diese können durch neue Werte in Spalten mit ausgehenden RCCs entstehen. Das Verletzen dieser Constraints muss in jedem Fall durch gezieltes Nachladen verhindert werden.

6.7.2 Asynchrones Nachladen von Datensätzen

Es wurde im Verlauf dieses Kapitels bereits ausführlich dargestellt, wann das Nachladen von Tupeln erforderlich sein kann (siehe 6.2). Immer wenn eine Änderung auf einer Spalte p mit ausgehendem RCC vorgenommen wurde, muss potentiell nachgeladen werden. Wenn der neue Spaltenwert w_n, der aus der Änderung resultiert, nicht bereits in der Spalte p existiert, muss in jedem Fall das Nachladen aller von w_n referenzierten Tupel ausgelöst werden. Andernfalls ist der Wert aufgrund der RCC-Bedingung bereits vollständig in der Zielspalte. Das Nachladen dieser Tupel funktioniert prinzipiell genauso, wie das Laden einer Cache Unit, bis auf den Unterschied, das das Laden in einer gewöhnlichen Spalte, anstatt einer Füllspalte startet.

6.7.3 Aufheben der Invalidierungen

Nachdem das Write Set vollständig angewendet wurde, und die nachzuladenden RCC-Hüllen in der Cache-Datenbank sind, können alle Invalidierungen der Transaktion aufgehoben werden. Dieser Zeitpunkt wird als *Revalidierungs-Zeitpunkt* bezeichnet. Er markiert den Zeitpunkt, in dem alle Änderungen der Transaktion im Cache atomar sichtbar werden.

Es muss allerdings beachtet werden, dass Invalidierungen, die auch von nachfolgenden Transaktionen angefordert wurden, nicht freigegeben werden. Deshalb gibt es für jede Invalidierung genau eine Java-Objekt. Jedes Invalidierungs-Objekt verwaltet einen internen Zähler. Immer wenn eine Invalidierung angefordert wird, die bereits besteht, wird der Zähler um eins erhöht. Bei Freigabe entsprechend um eins erniedrigt. Eine Invalidierung kann gelöscht werden, falls es keine Transaktionen in der Revalidierungs-Phase gibt, die diese Invalidierung angefordert haben.

6.7.4 Synchronisation der Revalidierungsphase mit Lade- und Entladevorgängen

Untersucht man die Nebeneffekte, die beim gleichzeitigen Ausführen der Revalidierungs-Phase und dem Laden oder Entladen von Cache Units entstehen können, so kommt man auf sehr viel mögliche Synchronisationsfehler. Nebenläufiges Laden und Entladen schließt sich sowieso aus. Bei gleichzeitigem Laden und Entladen kann es passieren, dass der Lader von unten beginnt eine Cache Unit zu laden und der Entlader eine Überlappende, von oben beginnend, entlädt. Dann kann es weiterhin dazu kommen, dass der Entlader Tupel löscht, die der Lader zuvor eingefügt hat. Der Lader setzt seinen Prozess fort, indem er weiter oben Tupel lädt, die die gelöschten referenzieren. Damit wäre der Cache in einem inkonsistenten Zustand.

Um Synchronisationsfehler zu vermeiden schließen sich deshalb diese drei Operationen aus. Zu einem Zeitpunkt lädt, entlädt oder revalidiert der Cache. Alle diese Aktionen werden sowieso asynchron ausgeführt, sodass sie keinen Einfluss auf die Antwortzeiten des Caches haben, und nicht gnadenlos parallelisiert werden müssen. Falls zu viele Jobs anstehen kann es allerdings passieren, dass der Cache nicht mehr voll ausgenutzt werden kann, weil die benötigten Tupel einfach zu spät in den Cache geladen werden. Für diese drei Vorgänge müsste ein sinnvoller Scheduler entwickelt werden, der die drei Vorgänge je nach Last entsprechend priorisiert.

Ein großer Vorteil dieser Vorgehensweise ist, dass auf der Cache-Datenbank dann keine Deadlocks auftreten können. Denn die Anwendungstransaktionen lesen nur, und Leseoperationen müssen bei Snapshot Isolation nicht synchronisiert werden. Der Cache-Manager ist damit der einzige Schreiber auf der Cache-Datenbank. Wenn er schreibt, dann weil er gerade lädt, entlädt oder revalidiert. Laden, Entladen und Revalidieren werden aber echt sequentiell ausgeführt.

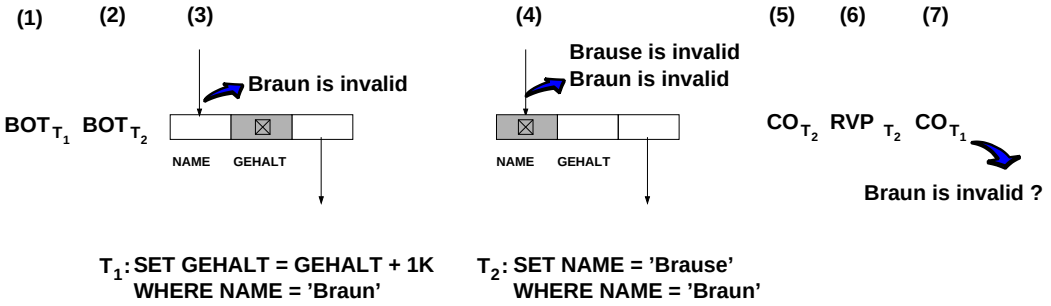
6.8 Unerwünschte Wechselwirkungen mit der Isolierung des zugrundeliegenden Datenbanksystems

Es wurde bereits darauf eingegangen, dass das zugrundeliegende Datenbanksystem für seine Transaktionen zumindestens Snapshot Isolation bieten muss. In Kapitel 5.4.1 wurde auf die Anomalien eingegangen, die bei Snapshot Isolation noch auftreten können. Die Probleme, die damit für die Umsetzung des beschriebenen Verfahrens einhergehen werden nun untersucht. Alle für Anomalien anfälligen Stellen werden ermittelt, damit sie entsprechend maskiert werden können.

6.8.1 Falsche RCC-Wert-Invalidierungen

Bei der Verwendung von RCC-Wert-Invalidierungen kann ein Write Skew auftreten. In Abbildung 6.14 ist die Problematik skizziert. Zwei Transaktionen T_1 und T_2 starten quasi gleichzeitig, d.h. $BOT_{T_1} \approx BOT_{T_2}$ und lesen vom gleichen Snapshot. T_1 macht eine Änderung auf einer RCC-freien Spalte: Eine Gehaltserhöhung für den Mitarbeiter 'Braun'. Sie ermittelt für den eingehenden RCC die zu sperrenden Werte, um den geänderten Datensatz unerreichbar zu machen. Aus Sicht von T_1 genügt es den Wert 'Braun' zu invalidieren. Gleichzeitig schreibt T_2 auf der Spalte mit dem eingehenden RCC. Der Mitarbeiter Braun hat geheiratet und heißt jetzt 'Brause'. T_2 führt die Commit-Operation aus und invalidiert dazu den alten und den neuen Wert am RCC. Noch bevor T_1 die Commit-Operation ausführt, werden die Änderungen von T_2 in die Cache-Datenbank eingebracht. Der Mitarbeiter 'Braun', der die Gehaltserhöhung bekommen soll, heißt also auf Backend- und Cache-Datenbank

Abbildung 6.14 Write Skew bei der RCC-Wert-Invalidierung



'Brause'. Dies hat T_1 aber nicht mitbekommen. Wenn T_1 die Commit-Operation ausführt liegt auch kein Schreibkonflikt vor, denn T_1 und T_2 haben unterschiedliche Datenbank-Objekte geschrieben. T_1 wird die Commit-Operation also erfolgreich abschließen und fälschlicherweise 'Braun' am RCC invalidieren. Nochmal zur Wiederholung: T_1 sieht die Änderung von T_2 nicht, denn sie war nebenläufig, und ist damit nicht im Snapshot von T_1 enthalten. Der geänderte Datensatz ist also auf dem Cache erreichbar, denn 'Brause' ist nicht gesperrt. Stattdessen ist 'Braun' unnötigerweise gesperrt.

Erste Lösungsansätze

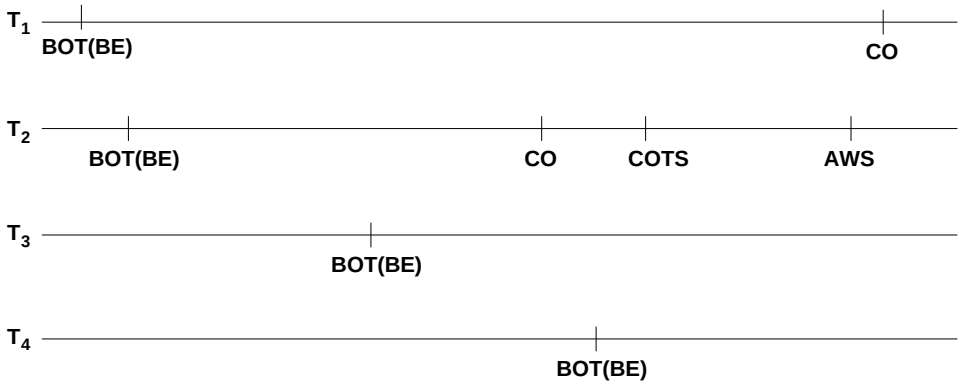
Das Problem entsteht an dieser Stelle, weil eine Änderung auf einer Spalte z mit eingehendem RCC gemacht wurde. Änderungen auf Spalten mit RCCs sind immer vergleichsweise teuer. Das geänderte Tupel bezeichne ich fortan als t_1. Man bedenke, dass diese Anomalie beim Verwenden gewöhnlicher RCC-Invalidierungen nicht auftreten kann. Neben der Einfachheit ist dies ein weiterer Pluspunkt, der für das Verwenden gewöhnlicher RCC-Invalidierungen spricht. Trotzdem soll eine Lösung zur Maskierung der fehlerhaften Invalidierung bei der RCC-Wert-Invalidierung erarbeitet werden.

Ein erster naiver Ansatz bestünde darin die RCC-Wert Invalidierung 'Braun' auf die RCC-Wert Invalidierung 'Brause' zu mappen. Also für jede nebenläufige Transaktion, die den Wert 'Braun' am RCC invalidieren möchte, stattdessen 'Brause' zu invalidieren. Dies ist aber nicht in jedem Fall korrekt, denn möglicherweise hat die Transaktion ein anderes Tupel t_2 geändert, dass in der Spalte NAME ebenfalls den Wert 'Braun' enthält. Dann müsste nämlich nach wie vor 'Braun' invalidiert werden. Ein solches Mapping kann nur erfolgreich durchgeführt werden, falls die Identität des Tupels t_1 bekannt ist. Dies ist aber aufwändig und nicht trivial, denn die Spalte NAME könnte auch Teil des Primärschlüssels sein. Damit kann die Identität eines

Tupels nur ausgenutzt werden, falls das zugrunde liegende Datenbanksystem den Zugriff auf einen internen Tupel-Identifikator erlaubt.

Ein weiterer Ansatz zur Maskierung dieser Anomalie besteht darin, zusätzlich zu 'Braun' an diesem RCC immer auch 'Brause' zu sperren. Also für jede nebenläufige Transaktionen, die 'Braun' invalidieren möchte, muss zusätzlich 'Brause' an dem RCC invalidiert werden. Denn alle nebenläufigen Transaktionen können die Änderungen von T_2 nicht sehen und damit liegt ein potentieller Write Skew vor. In Abbildung 6.15 ist das Beispiel aus Abbildung 6.14 mit einigen Erweiterungen schematisch dargestellt. T_1 und T_2 starten mehr oder weniger gleichzeitig. Es wird nicht davon ausgegangen, dass der exakte Commit-Zeitpunkt auf der Backend-Datenbank ermittelt werden kann. Deswegen wird ein Zeitpunkt möglichst unmittelbar nach dem Commit auf der Backend-Datenbank $COTS$ ermittelt. Für alle aktiven Transaktionen, deren $BOT(BE)$-Zeitpunkt kleiner als $COTS$ ist, und die 'Braun' am RCC invalidieren, muss zusätzlich 'Brause' invalidiert werden. Im Beispiel sind dies die Transaktionen T_1, T_3 und leider auch T_4. Bei T_4 wäre diese Maßnahme eigentlich nicht erforderlich, denn ihr globales BOT liegt hinter der Commit-Operation von T_2. Alle Änderungen von T_2 sind damit in ihrem Snapshot enthalten. Andererseits wird diese Situation nicht allzu oft auftreten, denn zum einen muss dazu $BOT(BE)$ in diesem ungünstigen Zeitintervall liegen. Und T_4 muss ein Tupel schreiben, für dass $NAME =$'Braun' gilt.

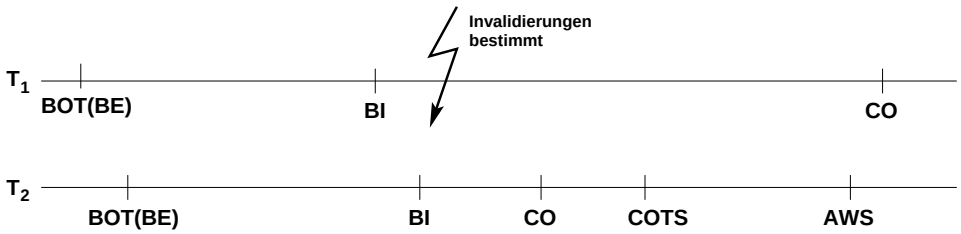
Abbildung 6.15 Maskierung von Write Skews bei der RCC-Wert-Invalidierung durch zusätzliche Invalidierungen



Alle Änderungen auf Spalten mit eingehenden RCCs werden in einer Hauptspeicher-Datenstruktur vorgehalten, und müssen so lange aufgehoben werden, wie noch Transaktionen aktiv sind, deren $BOT(BE)$-Zeitpunkt kleiner oder gleich $COTS$ ist. Ein dabei entstehender schwerwiegender Nachteil ist, dass nun auf dem Backend die Prepare-Phasen und Commit-Operationen synchronisiert werden müssen. Zu einem Zeitpunkt kann sich dann nur eine Transaktion in der Prepare-Phase befinden, die

Commits auf der Backend-Datenbank müssen in der gleichen Reihenfolge wie die korrespondierenden Prepare-Phasen ausgeführt werden. Das dies zwingend erforderlich ist, kann man sich leicht an dem Beispiel aus Abbildung 6.16 verdeutlichen. Dabei handelt es sich wieder um die beiden aus den vorangegangenen Beispielen betrachteten Transaktionen T_1 und T_2. Zusätzlich sollen die Prepare-Phase von T_1 und T_2 verzahnt ablaufen. Dann kann es passieren, dass T_1 das Ermitteln aller ungültigen RCC-Werte abschließt, bevor alle ungültigen RCC-Werte von T_2 bekannt sind. Dann könnten den Invalidierungen von T_1 die zusätzliche Invalidierung des Wertes 'Brause' nicht zugänglich gemacht werden, und der geänderte Datensatz wäre nach dem Commit von T_1 wiederum fälschlicherweise erreichbar. Weiterhin gilt, dass wenn T_1 zuerst die Prepare-Phase ausführt, T_1 dann auch zuerst die Commit-Operation ausführen muss.

Abbildung 6.16 Synchronisation von Prepare-Phasen und Commit-Operationen bei der RCC-Wert-Invalidierung



Das Synchronisieren der Prepare-Phasen und Commit-Operationen kann sich sehr negativ auf das Gesamtsystem auswirken. Zu einem Zeitpunkt kann sich dann maximal eine Transaktion in der Prepare-Phase befinden. Alle anderen Transaktionen, die gleichzeitig in die Prepare-Phase eintreten wollen werden blockiert. Das Bestimmen der Invalidierungen mehrerer Transaktionen kann nicht mehr sinnvoll parallelisiert werden. Eine Transaktion die bereits im Zustand *prepared* ist, kann erst die endgültige Commit-Operation auf dem Backend ausführen, bis alle Transaktionen, die vor ihr in der Prepare-Phase waren, die Commit-Operation ausgeführt haben.

Lange RCC-Wert-Invalidierungen

Zur Lösung des Problems führe ich die *lange RCC-Wert-Invalidierungen* ein. Eine lange RCC-Wert-Invalidierung wird nicht, wie die gewöhnliche (kurze) RCC-Wert-Invalidierung nach der Revalidierungs-Phase einer Transaktion aufgehoben. Jede Änderung $old\_value \mapsto new\_value$ auf einer Spalte mit eingehendem RCC hat eine lange RCC-Wert-Invalidierung von new_value zur Folge. Bleiben wir bei dem Beispiel aus Abbildung 6.14. Dann resultiert aus der Änderung 'Braun' $\mapsto$ 'Brause'

eine lange RCC-Wert-Invalidierung von 'Brause'. T_1 bekommt von alledem nichts mit und invalidiert lediglich 'Braun', was zur Folge hätte, dass nach ihrer Commit-Operation der geänderte Datensatz erreichbar wäre. Ohne Synchronisation bekäme T_1 auch von der RCC-Wert-Invalidierung 'Brause' von T_2 nichts mit, da T_1 ihre Prepare-Phase abschließt bevor T_2 ihre beendet (vgl. Abbildung 6.16). Zur Behebung des Problems wird die lange RCC-Wert-Invalidierungen nicht zum Revalidierungs-Zeitpunkt RVP_{T_2} aufgehoben, sondern erst nachdem jede Transaktion, die einen BOT-Zeitpunkt kleiner $COTS_{T_2}$ hat, revalidiert hat. Sie bleibt also so lange gültig, bis alle nebenläufigen Transaktionen die Revalidierungs-Phase abgeschlossen haben. Damit bleibt in dem Beispiel die RCC-Wert-Invalidierung 'Brause', so lange bestehen bis die Änderungen von T_1 eingebracht worden sind.

Im Folgenden soll gezeigt werden, dass lange RCC-Wert-Invalidierungen fehlerhafte Invalidierungen, die aus einem Write Skew resultieren, maskieren können. Sei dazu wiederum T_2 eine Transaktion, die ein Update auf einer Spalte mit eingehendem RCC ausführt. Bei der nebenläufigen Transaktion T_1 tritt aufgrund des Updates ein Write Skew auf. Die fehlerhafte Invalidierung von T_1 kann nur auftreten falls folgendes gilt:

1. $BOT_{T_1} \approx BOT_{T_2} < CO_{T_2} \Rightarrow$ Die Transaktionen starten nebenläufig
2. $CO_{T_2} < CO_{T_1} \Rightarrow$ Das Write Set von T_2 wird vor dem Write Set von T_1 eingebracht

Die lange RCC-Wert-Invalidierung von T_2 gilt spätestens zum Zeitpunkt CO_{T_2}. Da $CO_{T_2} < CO_{T_1}$ gilt die Invalidierung bei Commit von T_1. Sie bleibt per Definition bestehen, bis T_1 revalidiert hat, da $BOT_{T_1} < CO_{T_2}$. Somit besteht die lange RCC-Wert-Invalidierung über die gesamte Dauer der Revalidierungs-Phase von T_1 und die fehlerhafte Invalidierung kann durch ihre Gültigkeit maskiert werden.

Zusammenfassend ist eine lange RCC-Wert-Invalidierung eine RCC-Wert-Invalidierung, die nicht bereits zum Revalidierungs-Zeitpunkt ihrer Transaktion T aufgehoben wird. Sie muss so lange bestehen bleiben, bis alle zu T nebenläufigen Transaktionen revalidiert haben. Denn bei all diesen nebenläufigen Transaktionen tritt potentiell ein Write Skew, der eine fehlerhafte RCC-Wert-Invalidierung zur Folge hätte, auf. Nachteil dieses Verfahrens ist, dass die Invalidierung auch dann besteht, wenn bei der nebenläufigen Transaktion gar kein Write Skew aufgetreten ist. Außerdem macht die Transaktion, bei der der Write Skew auftritt, eine unnötige Invalidierung. Von Vorteil ist allerdings, dass die Prepare-Phasen und Commit-Operationen nicht synchronisiert werden müssen. Weiterhin ist auch in diesem Fall davon auszugehen, dass Änderungen auf Spalten mit eingehenden RCCs eher die Ausnahme als die Regel sind.

6.8.2 Cache Lost Updates

Werden bei der Anwendung von Write Sets Primärschlüssel zur Identifikation des zu ändernden Tupel herangezogen, dann kann es auch hier zu einem Write Skew kommen. Das Szenario in dem es zu der Anomalie kommt, ist ähnlich zu dem bereits diskutierten Fall mit den RCC-Wert-Invalidierungen. Zwei Transaktionen starten quasi gleichzeitig, beide schreiben ein bestimmtes Tupel, und zwar so, dass es zu keinem Schreibkonflikt kommt. Die Transaktion T_1, die zuerst die Commit-Operation auf der Backend-Datenbank ausführt hat den Primärschlüssel dieses Tupel geschrieben. Wenn die zweite Transaktion T_2 später ihr Write Set extrahiert, identifiziert sie das Tupel mit dem veralteten Primärschlüssel. Weil die Write Sets in Commit-Reihenfolge angewendet werden, hat das Tupel zum Zeitpunkt der Anwendung ihres Write Sets von T_2 bereits den neuen Primärschlüssel. Das generierte Update-Statement identifiziert dann im Cache gar kein Tupel (oder ein Falsches). Dies hat zur Folge, das das Update der Transaktion T_2 verlorgen geht.

Das Problem würde nicht auftreten, wenn der Cache die internen Tupel-Identifikatoren des Backend-Datenbanksystems pflegen könnte. Ein Zugriff auf diese Identifikatoren über das Datenbanksystem ist aber Voraussetzung. Denkbar wäre auch, das das gesamte ACCache-System eigene Tupelidentifikatoren verwalten würde. Jedoch sollte prinzipiell jedes beliebiges Datenbankschema unterstützt werden. Das Speichern von internen Tupelidentifikatoren in der Backend-Datenbank stellt eine Schemaänderung dar und ist mit Aufwand verbunden. Im nächsten Kapitel wird deshalb eine Lösung entwickelt, die diese Lost Updates verhindert. Da diese ausschließlich in den Cache-Datenbanken auftreten können bezeichne ich sie im folgenden als *Cache Lost Updates*.

Maskierung von Cache Lost Updates

Um die Cache Lost Updates maskieren zu können, müssen bei der Anwendung eines Write Sets einige zusätzliche Punkte beachtet werden. Zum einen müssen alle Primärschlüssel-Updates in einer Hauptspeicherdatenstruktur gespeichert und allgemein zugänglich gemacht werden. Zum anderen muss bei der Anwendung eines Write Sets überprüft werden, ob die Transaktion bei der Extraktion des Write Sets einem Write Skew unterlegen war. Dabei muss wie folgt vorgegangen werden:

1. Für jeden Update- und für jeden Delete-Eintrag des Write Set:
2. Gibt es für den Tupelidentifikator des Eintrages ein Primärschlüssel-Update?
3. Wenn ja, liegt der globale Commit-Zeitpunkt der Transaktion, die das Primärschlüssel-Update durchgeführt hat, zeitlich zwischen dem globalen BOT und dem globalen Commit der Transaktion dieses Write Sets? Dann war das

Primärschlüssel-Update nebenläufig und die Transaktion des Write Sets unterlag einem Write Skew.

4. Wenn die Transaktion des Write Sets einem Write Skew unterlag, ändere den Tupelidentifikator des Eintrages in den aktuellen Primärschlüssel

Die Primärschlüssel-Updates müssen so lange aufgehoben werden, bis die letzte Transaktion revalidiert hat, deren globaler BOT-Zeitpunkt kleiner ist als der globale Commit-Zeitpunkt der Transaktion mit dem Primärschlüssel-Update. Bei jeder dieser Transaktionen kann ein potentieller Write Skew auftreten.

Das System sollte sich die älteste Transaktion, also diejenige mit dem kleinsten globalen BOT-Zeitpunkt, merken. Immer wenn eine Transaktion vollständig revalidiert hat, muss dieser Pointer gegebenenfalls umgesetzt werden. Wenn sich die Referenz auf die älteste Transaktion ändert, sollte überprüft werden, ob Primärschlüssel-Updates gelöscht werden können.

6.9 Crash

Eine weitere wichtige Frage ist die nach den Folgen eines Crashes von einem der beteiligten Knoten eines ACCache-Systems. Der Crash des Backend ist schwerwiegend. Wenn das Backend nicht mehr da ist, dann "geht fast nichts mehr". Wenn der Cache merkt, dass die Verbindung zum Backend zusammengebrochen ist, könnte er in einen Read-only-Modus wechseln. In diesem Zustand werden alle Update-Requests von Transaktionen mit einer Fehlermeldung quittiert, um damit anzuzeigen, dass nur noch gelesen werden kann. Es ist dann davon auszugehen, dass es global keine Updates mehr gibt, da das Backend nicht mehr da ist. Von all denjenigen Transaktionen, die vor dem Crash noch erfolgreich die Commit-Operation abschließen konnten, wurden zuvor die Invalidierungen angewandt. Somit sind die Daten im Cache nicht veraltet und können noch gelesen werden. Man sollte an dieser Stelle nicht vergessen, dass das ACCache-System kein Replikationssystem ist. Es handelt sich eben "nur" um Caches für eine große Master-Datenbank.

Wenn ein Cache zusammenbricht, bekommt dies das Backend spätestens mit, wenn die Verbindung zum Cache geschlossen wird. Das Backend kann dann davon ausgehen, dass der Cache nicht mehr da ist. Ab sofort ist er dann kein aktiver Teilnehmer des Transaktions-Management-Protokolls. Das heißt das Backend schickt keine Invalidierungs- und Commit-Nachrichten an den Cache. Alle offenen Transaktionen, die dem unerreichbaren Cache zugeordnet werden können, müssen dann abgebrochen werden. Falls sich die abzubrechende Transaktion noch nicht in der Invalidierungs-Phase befand, genügt es die Transaktion auf der Backend-Datenbank abzubrechen und den globalen Transaktionskontext als *aborted* zu markieren. Andernfalls muss

außerdem an alle anderen Caches eine Abort-Nachricht für die entsprechende Transaktion geschickt werden. Diese können dann gegebenenfalls bereits vorgenommene Invalidierungen sofort aufheben.

Je nach Konfiguration braucht eine TCP-Verbindung relativ lange bis sie einen Timeout bekommt. Deswegen kann es vorkommen, dass der Zusammenbruch eines Caches nicht unbedingt sofort bemerkt wird. Dies ist für Transaktionen, die sich gerade in der Invalidierungs-Phase befinden kritisch. Denn die Transaktion kann solange nicht die Commit-Operation auf der Backend-Datenbank ausführen, bis von allen Caches die Invalidierungs-Bestätigung eingetroffen ist. An dieser Stelle sollte deswegen der Backend-Manager zum Beispiel mit TCP-Keepalive arbeiten, oder einen Heartbeat implementieren. Gegebenenfalls können in einer solchen Übergangs-Phase, in der noch nicht klar ist, ob der Cache wirklich zusammengebrochen ist, vereinzelte Transaktionen über die Abort-Operation abgebrochen werden.

Ein Cache ist nach seinem Boot-Vorgang leer, und muss sich erst wieder neu laden. Deshalb ist ein Crash seitens des Cache relativ unkritisch. Der Cache lädt dann automatisch die aktuellen Datenversionen. Er weiß nach dem Reboot allerdings nicht mehr, welche Transaktionen bei seinem Crash noch offen waren und welche vom Backend-Manager global commited wurden. Denn der Cache verwaltet darüber keine persistenten Informationen. All diejenigen, die noch nicht die globale Commit-Operation auf de Backend-Datenbank ausgeführt hatten wurden dann vom Backend-Manager abgebrochen. Alle anderen gelten offiziell als commited. Deren Änderungen sind dann persistent in der Backend-Datenbank und werden von den Caches entsprechend übernommen.

6.10 Zusammenfassung

In diesem Kapitel wurde ein auf das Constraint-basierte Datenbank-Caching zugeschnittenes Synchronisationsverfahren vorgestellt. Das Ausnutzen der Cache-Constraints ermöglicht das Implementieren einer halbsynchronen Strategie zum Propagieren von Änderungen. Dadurch werden Transaktionen bei ihrer Commit-Operation minimal blockiert. Selbst ein 2-Phase-Commit-Protokoll benötigt an dieser Stelle mindestens drei Nachrichten (bzw. drei Mal Latenz). Gleichzeitig ist sichergestellt, dass jede Anwendungstransaktion stets den bei ihrem BOT jüngsten transaktionskonsistenten Zustand sieht. Es gibt also aus Sicht der Anwendung keine Stale-Data-Situation. Aufgrund der Snapshot Isolation kann allerdings bei lang laufenden Transaktionen nicht ausgeschlossen werden, dass die Daten im Snapshot veralten. Der hohen Konfliktrate wird Rechnung getragen, indem alle Schreiboperationen auf der Backend-Datenbank synchronisiert werden müssen. Hier kann es dann zu Blockierungen der Schreiber kommen. Allerdings werden dadurch Deadlocks, die sich bei Update-Everywhere über mehrere Knoten verteilt bilden können ausgeschlossen (vgl.

[Thi07]). Das Behandeln von solch globalen Deadlocks dürfte noch deutlich teurer sein.

Der Lebenszyklus einer ACCache-Transaktion wurde vorgestellt. Dieser unterteilt sich in die drei Phasen: Ausführungs-Phase, Invalidierungs-Phase und Revalidierungs-Phase. Die Invalidieruns-Phase ist kritisch, weil ihre Dauer *propagation_time* entspricht (vgl. 5.2.1), und damit bestimmt, wie lange die Transaktion durch das ACCache-System blockiert wird. Es wurde in Kapitel 5.2.1 außerdem gezeigt, dass diese Zeit, die zum Propagieren der Änderungen benötigt wird, direkten Einfluss auf die Konfliktrate hat.

Weiterhin wurden zwei Invalidierungs-Techniken vorgestellt und untersucht, sowie deren Vor- und Nachteile aufgezeigt. Ein endgültiges Urteil, welche Technik dabei zu favorisieren ist, kann erst nach einigen praktischen Tests und Messungen mit realen Daten und einer realistischen Arbeitslast an Transaktionsprogrammen, gefällt werden. Beide Invalidierungstechniken zeichnen sich dadurch aus, dass sie vollständig im Hauptspeicher abgewickelt werden können. Dies ist ein wichtiger Punkt, denn IO-Operationen sind bei einem Transaktionssystem ein stark limitierender Faktor. Die Revalidierungs-Phase erfolgt asynchron und ist auf allen Caches unterschiedlich. Sie ist nicht kritisch, allerdings sollte auch sie möglichst zügig durchgezogen werden, damit auf dem Cache zu jedem Zeitpunkt möglichst alle zwischengespeicherten Daten erreichbar sind.

Zum Schluss wurde untersucht, inwieweit die Isolation und damit einhergehende Anomalien der zugrundegelegten Datenbanksysteme das ACCache-System betreffen. Die von potentiellen Anomalien betroffenen Stellen wurden identifiziert, und Lösungsansätze zu deren Maskierung wurden vorgestellt.

7 Implementierung

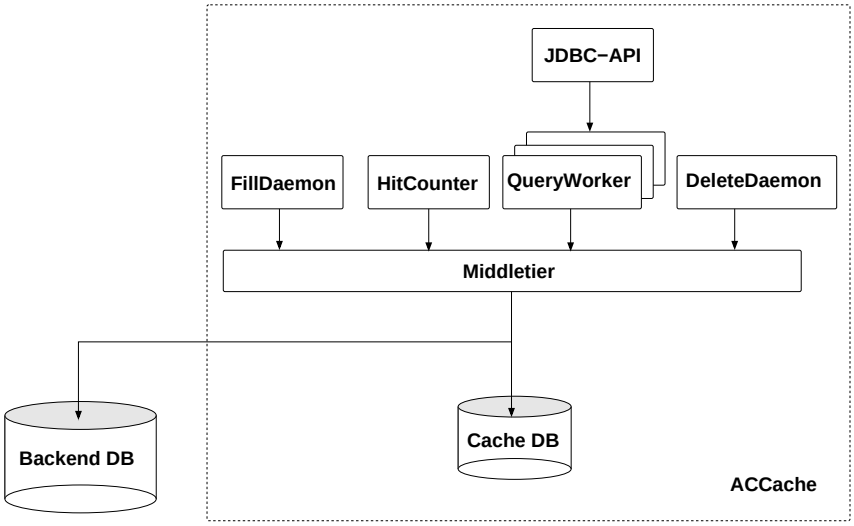
In diesem Kapitel wird die prototypische Implementierung des entwickelten Verfahrens vorgestellt. Als Einstieg wird eine kurze Zusammenfassung über den Stand des ACCache-Systems zu Beginn der Arbeit gegeben. Anschließend werden die vorgenommenen Erweiterungen vorgestellt. Dabei wird zunächst auf einer abstrakten Ebene auf die einzelnen Komponenten eingegangen, um dann anschließend etwas tiefer in die konkrete Implementierung einzuführen.

7.1 Stand zu Beginn der Arbeit

In Abbildung 7.1 ist der Stand des ACCache-Prototyps zu Beginn dieser Arbeit dargestellt. Dieser bestand dabei im wesentlichen aus dem Cache-Manager, ein Backend-Manager war nicht vorhanden. Der Zugriff auf die Daten der Backend-Datenbank erfolgte daher stets über eine *Middletier* genannte Softwareschicht. Deren Implementierung beruht im wesentlichen auf dem Föderationsfeature der IBM DB2. Die Middletier kapselte eine Verbindung zur Cache-Datenbank. Durch die Föderation zwischen Cache- und Backend-Datenbank konnte über diese Verbindung außerdem völlig transparent auf die Backend-Datenbank zugegriffen werden.

Für die Anwender eines ACCache lag eine JDBC-Treiber-Implementierung vor. Allerdings wurden hier wesentliche Teile, nämlich alle transaktionalen Funktionalitäten, ausgeklammert. Ein Client konnte lesende Anfragen ausführen. Zur Verarbeitung der Anfragen, wurden ein Pool von *QueryWorkern* vorgesehen. Jeder Query-Worker benutzte ein Middletier-Objekt für seine Datenbankzugriffe. Vor jeder Auswertung wurde zunächst eine Sondierung durchgeführt (siehe Kapitel 2.3). Alle Tabellen, die unter Ausnutzung der gültigen Cache-Constraints zur Auswerten der Anfrage herangezogen werden konnten wurden verwendet (siehe [Mer05]). Die verbleibenden Tupel der Prädikatextension wurden aus Backend-Tabellen gelesen. Aufgrund der Föderation war es möglich dass der Zugriff auf Backend- und Cache-Tabellen gleichzeitig in einer SQL-Anfrage stattfinden konnte. Die Query-Worker waren waren über Java RMI ansprechbar, und das Absetzen einer Query wurde vom Treiber über einen entfernten Methoden-Aufruf implementiert. Somit ist davon auszugehen, dass jede Query in ihrem eigenen Thread ausgeführt wurde.

Abbildung 7.1 Stand ACCache-Prototyp (entnommen aus [Mer05])



Weitere wichtige Komponenten betreffen das Laden und Entladen. Diese finden auch jeweils in einem eigenen Thread statt. Das Laden der Cache Units übernimmt der *FillDaemon*, das Entladen entsprechend der *DeleteDaemon*. Der *HitCounter* erfasst die Anzahl und Zeitpunkte der Cache Units-Zugriffe. Auf Grundlage der so erfassten Daten implementiert der *DeleteDaemon* eine Verdrängungsstrategie für die Cache Units.

7.2 Vorgenommene Erweiterungen

Im Folgenden werden die am ACCache-Prototyp vorgenommenen Erweiterungen beschrieben. Die wichtigste Komponente ist die Transaktionsmanagementkomponente. Sie bildet das Herzstück der Erweiterung. Diese Komponente benutzt eine IO-Managementkomponente. Letztere bietet eine Abstraktion von den TCP-Ein- und Ausgabeströmen. Außerdem verwaltet jede Transaktionsmanagementkomponente einen Pool an Verbindungen zu ihrer lokalen Datenbank. Sowohl Cache als auch Backend verfügen über diese Komponenten. Beide Seiten überschreiben dazu Standard-Implementierungen aus dem `acc.transactions`-Paket. Zwischen Cache- und Backend gibt es jeweils genau eine TCP-Verbindung. Jeder Cache hat eine eindeutige numerische ID, die er über eine Registrierung mit die Management-Schnittstelle des Backend-Managers erhält. Für das Transaktionsmanagement wurde ein Protokoll entwickelt, über das Cache und Backend kommunizieren. Jeder Cache-Manager besitzt weiterhin eine Komponente, die zuständig ist für die Invalidierungen.

Zusätzlich wurde ein neuer JDBC-Treiber implementiert. Dieser kommuniziert mit dem ACCache-System über das entwickelte Transaktionsmanagement-Protokoll. Dabei muss der Anwender insbesondere nicht zwischen Cache und Backend unterscheiden. Der Treiber arbeitet sowohl mit dem Cache-Manager als auch mit dem Backend-Manager zusammen.

7.2.1 Select-Schleife

Eine wichtige Anforderung an das Synchronisationsverfahren war, dass das Verfahren möglichst effizient arbeiten sollte. Es ist beispielsweise wichtig, dass die Nachrichten, des Transaktionsmanagementprotokolls schnell bearbeitet werden können. Da es sich um eine Datenbank-Technologie handelt, ist bei der Bearbeitung mit einem hohen IO-Aufwand zu rechnen. Dies bedeutet, dass die Nachrichten in keinem Fall seriell und von einem einzigen Thread bearbeitet werden können. Sonst würde bei jeder IO-Operation die gesamte Transaktionsmanagementkompononete blockiert werden. Weiterhin muss das System zeitweilig, hohe Nachrichtenraten aushalten können. Diese erfordert, dass das System Nachrichten asynchron Versenden und Empfangen kann.

Damit würde die Implementierung eines Java-RMI-Protokolls diesen Anforderungen nicht genügen. RMI bietet eine wunderbare Abstraktion von den technischen Details der Ein- und Ausgabe über eine Netzwerkverbindung. Allerdings ist RMI für das Implementieren eines hochskalierenden Server nicht geeignet. Dies hat mehrere Gründe:

- RMI öffnet beim Exportieren seiner Remote-Objekte sehr viele lokale TCP-Ports.
- Ein dezidierter Server-Port muss künstlich nachgebildet werden.
- Es kann kein direkter Einfluss auf das Scheduling der Verarbeitung der eingehenden Nachrichten und Verbindungsanforderungen genommen werden.
- Es kann kein direkter Einfluss auf das TCP-Verbindungs-Management genommen werden. RMI öffnet sehr viele TCP-Verbindungen, und das Management dieser Verbindungen muss vom Betriebssystem gehandelt werden und ist nicht unbedingt als optimal anzusehen.
- Es kann kein Einfluss auf das Thread-Management genommen werden. Es ist abhängig von der Implementierung der Java Virtual Machine (kurz: JVM), ob für jede Remote Method Invocation ein neuer Thread gestartet wird, oder ob ein Thread-Pooling stattfindet.
- Es ist von der Implementierung der JVM abhängig, wie performant eingehende Verbindungsanfragen und Nachrichten verarbeitet werden.

- Es müssen serialisierte Java-Objekte über die TCP-Verbindung geschickt werden. Serialisierte Java-Objekte können sehr groß sein, und aufgrund der Abstraktion durch RMI, ist sich der Anwendungsprogrammierer dessen nicht immer bewusst.

Aus diesen Gründen wurde das Ein- und Ausgabemanagement auf der unteren TCP-Ebene mit Java NIO implementiert (siehe [Hit02]). Java NIO bietet viele Klassen für ein effizientes und asynchrones Verarbeiten von IO-Vorgängen. Insbesondere kann die Selektor-Implementierung des Betriebssystems (siehe [IEE01]) aus Java heraus verwendet werden.

Ein Kernelement bilden die `java.nio.channels.Channel`-Implementierungen. Diese Kanäle erlauben den performanten Zugriff auf IO-Services des Betriebssystems (siehe [Hit02]). Das Lesen von einem Kanal, und das Schreiben von Daten in diesen erfolgt über `java.nio.Buffer`-Objekte. Daten werden nicht direkt in den Kanal geschrieben, sondern zunächst in einen Puffer. Das Kanal-Objekt wiederum liest die Daten aus dem Puffer und sendet sie über seinen Sockt zum Empfänger. Umgekehrt schreibt der Kanal die Daten, die über den Socket empfangen werden in einen Puffer, vom dem die Anwendung die empfangenen Bytes liest. Dieser Zusammenhang ist anschaulich in Abbildung 7.2 dargestellt. Der `ByteBuffer`-Wrapper greift direkt auf einen Betriebssystempuffer zu. Java-Byte-Puffer können auch "direkt" allokiert werden, dann befinden sie sich nicht im Heap, und der Datentransfer kann noch schneller abgewickelt werden.

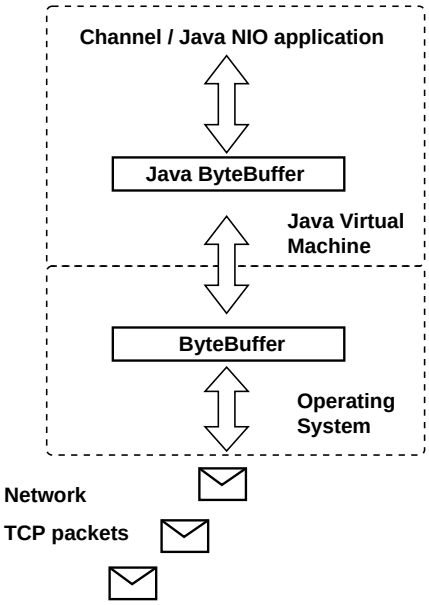
Die Unterklasse `SelectableChannel` arbeitet zusammen mit einem `java.nio.channels.Selector`. Der Selektor ermöglicht es die Zustände einer Menge von Kanälen gleichzeitig abzufragen. Die API-Aufrufe der Java-NIO-Selektor-Implementierung werden mehr oder weniger direkt auf Betriebssystemaufrufe umgesetzt und sind dementsprechend performant. Der Funktionsweise einer Select-Schleife wird im Folgenden etwas näher vorgestellt.

Funktionsweise einer Select-Schleife

Ein Selektor ermöglicht es, eine Vielzahl von TCP-Verbindungen gleichzeitig durch einen einzigen Thread zu verwalten (vgl. [Hit02]). Dem Thread wird es ermöglicht, schnell und effizient, den Zustand seiner Kanäle abzufragen, ohne selbst dabei blockiert zu werden. Falls für keinen Kanal das Ausführen von Aktionen erforderlich ist, kann mit einer anderen Aufgabe fortgefahren werden (siehe [Hit02]). Für den Fall dass mehrere Kanäle behandelt werden müssen, kann ein Scheduling der einzelnen Tasks vorgenommen werden.

Der Selektor ermöglicht dies, indem er für jeden registrierten Kanal ein *Interest Set* in Form einer Bitmap verwaltet. Das Interest Set enthält alle Operationen, über die der Thread informiert werden möchte, sobald sie auf dem Kanal verfügbar werden. Also beispielsweise, ob auf dem Kanal eine Leseoperation ausgeführt werden kann,

Abbildung 7.2 Java NIO: Channel und ByteBuffer (entnommen aus: [Hit02])



weil sich dort noch ungelesene Daten befinden. Für einen TCP-(Server)-Socket sind die folgenden Operationen mit dem Selektor registrierbar:

- `ACCEPT`: Informiert darüber, dass eine neue TCP-Verbindungsanfrage vorliegt.
- `READ`: Informiert darüber, dass Daten vom Kanal gelesen werden können.
- `WRITE`: Informiert darüber, dass Daten auf den Kanal geschrieben werden können.

Wird ein Kanal bei einem Selektor registriert, so gibt die entsprechende Methode ein `SelectionKey`-Objekt zurück. Dieser Schlüssel identifiziert den Kanal eindeutig, und wird für alle den Kanal betreffenden Operationen mit dem Selektor verwendet. In Listing 7.1 und Listing 7.2 sind Ausschnitte aus dem `Selector`- bzw. `SelectionKey`-Interface gegeben. Die Selektor-Schnittstelle enthält Methoden für blockierende und nicht-blockierende Select-Operationen. Bei einer Select-Operation werden alle Kanäle auf die Ausführbarkeit der Operationen ihres Interest-Set überprüft. Anschließend kann mit der `selectedKeys`-Methode auf diejenigen Schlüssel zugegriffen werden, auf deren Kanäle Operationen ausgeführt werden können.

Über die Schnittstelle des `SelectionKey`-Objektes kann dann mit der Verarbeitung fortgefahren werden. Über die `channel`-Methode kann man eine Referenz auf das zugehörige Kanal-Objekt erhalten. Weiterhin gibt es Methoden zum Überprüfen der

Listing 7.1 Ausschnitt aus der `java.nio.channels.Selector`-Schnittstelle

```
public abstract class Selector {

        // perform a blocking select operation
        public abstract int select( ) throws IOException;

        // perform a non-blocking select operation
        public abstract int selectNow( ) throws IOException;

        // retrieve all registered channels (via their key)
        public abstract Set<SelectionKey> keys( );

        // retrieve all ready channels (via their key)
        public abstract Set<SelectionKey> selectedKeys( );

        . . .
}
```

Ausführbarkeit der einzelnen Operationen (`isReadable`, `isWritable`, `isAcceptable`). Ein weiteres Feature ist, dass mit jedem Schlüssel-Kanal-Paar ein beliebiges Java-Objekt über die `attach`-Methode assoziiert werden kann. Dieses Objekt kann beispielsweise Metadaten zum Kanales aufnehmen.

In Listing 7.3 ist nun ein Beispiel für eine einfache Select-Schleife gegeben.

IO-Managementkomponente

Das `acc.transactions.connectionMgmt`-Paket stellt grundlegende Ein- und Ausgabefunktionalitäten auf der Socket-Ebene bereit. Alle gelesenen Bytes werden in Java-Nachrichten-Objekte umgewandelt, mit denen die höheren Schichten des Transaktionsmanagements bequem arbeiten können. Im Folgenden sollen die Klassen dieses Paketes etwas näher vorgestellt werden. Das Zusammenspiel dieser Klassen ist außerdem in Abbildung 7.3 illustriert.

`ServerSelectSockets` Diese abstrakte Klasse implementiert eine Select-Schleife. Sie ist außerdem vom Typ Runnable, sodass sie als Thread ausgeführt werden kann. Dieser Thread blockiert so lange, bis mindestens ein Kanal beschreib- oder lesbar ist, oder eine neue Verbindungsanfrage eintrifft. Für jeden lesbaren Kanal wird ein `SocketReader`-Objekt erzeugt. Dieser Worker erhält eine Referenz auf den Kanal. Anschließend wird der Worker dem `SocketWorkerExecutor` übergeben. Dieser übernimmt die Ausführung des Workers, durch seinen Thread-Pool.

Listing 7.2 Ausschnitt aus dem `java.nio.channels.SelectionKey`-Interface

```
public abstract class SelectionKey {

        public static final int OP_READ
        public static final int OP_WRITE
        public static final int OP_ACCEPT

        // retrieve the channel of this key
        public abstract SelectableChannel channel();

        // retrieve or set the interest set (bitmap)
        public abstract int interestOps( );
        public abstract void interestOps (int ops);

        // retrieve all ready operations (bitmap)
        public abstract int readyOps( );

        // check readiness of operations
        public final boolean isReadable( );
        public final boolean isWritable( );
        public final boolean isAcceptable( );

        // retrieve or set an arbitrary object, that
        // is associated with the channel of this key
        public final Object attach (Object ob);
        public final Object attachment( );

        . . .
}
```

Für einen schreibbaren Channel wird analog mit einem `SocketWriter` verfahren. Durch diese Vorgehensweise wird der Server-Thread nicht durch das Lesen und Schreiben von Nachrichten blockiert, weil er bei Bedarf immer an einen Worker delegiert.

`ChannelIdentifier` In einem Kanalidentifikator werden Metadaten über einen Kanal gespeichert. Der Identifikator selbst, wird über die `attach`-Methode mit dem `SelectionKey` des Kanales assoziiert. Wenn der Kanal beispielsweise eine Verbindung zu zu einem Cache herstellt, so kann die ID dieses Cache im Identifikator gespeichert werden.

`SocketReader` **und** `SocketWriter` Die Implementierungen der Worker-Klassen SocketReader und SocketWriter sind zustandslos. Dadurch kann jeder SocketReader jeden beliebigen lesbaren Kanal verarbeiten. Entsprechendes gilt für be-

Listing 7.3 Beispiel für eine Select-Schleife

```
while (true) {

        // block until at least one channel is ready
        int numOfChangedChannels = this.selector.select();
        Set<SelectionKey> keys = selector.selectedKeys();

        for(SelectionKey key: keys) {

                if (key.isAcceptable()) {
                        this.acceptables.add(key);
                }
                if (key.isReadable()) {
                        this.readables.add(key);
                }
                if (key.isWritable()) {
                        this.writables.add(key);
                }

                // remove the key, as
                // it will be processed
                keys.remove(key);
        }

        // handle all ready channels
        this.handleReadables();
        this.handleWritables();
        this.handleAcceptables();
}
```

schreibbare Kanäle und SocketWriter. Jeder dieser Socket Worker ist vom Typ `Runnable`. Dies ermöglicht die Ausführung der Worker über einen `SocketWorkerExecutor`. Weitherhin werden alle vom ServerSelectSockets-Objekt verwalteten Kanäle als nicht-blockierend konfiguriert. Deswegen liest oder schreibt ein solcher Worker nur solange Daten, wie dies auf dem Kanal möglich ist. Der Worker wird niemals blockiert, sobald nichts mehr zu tun ist, beendet er sich. Weiterhin kann der SocketReader mit Fragmenten von Protokoll-Nachrichten umgehen. Wenn eine Nachricht zu groß ist, und nicht auf einmal gelesen werden kann, so erzeugt er ein `MessagePart`-Objekt. Dieses wird im Kanalidentifikator gespeichert. Sobald auf dem Kanal wieder neue Daten gelesen werden können, fährt er mit dem Lesen der Nachricht fort.

`SocketWorkerExecutor` Die Implementierung des SocketWorkerExecutor basiert auf den Bibliotheken aus `java.util.concurrent.*`, die seit Java 1.5 verfügbar

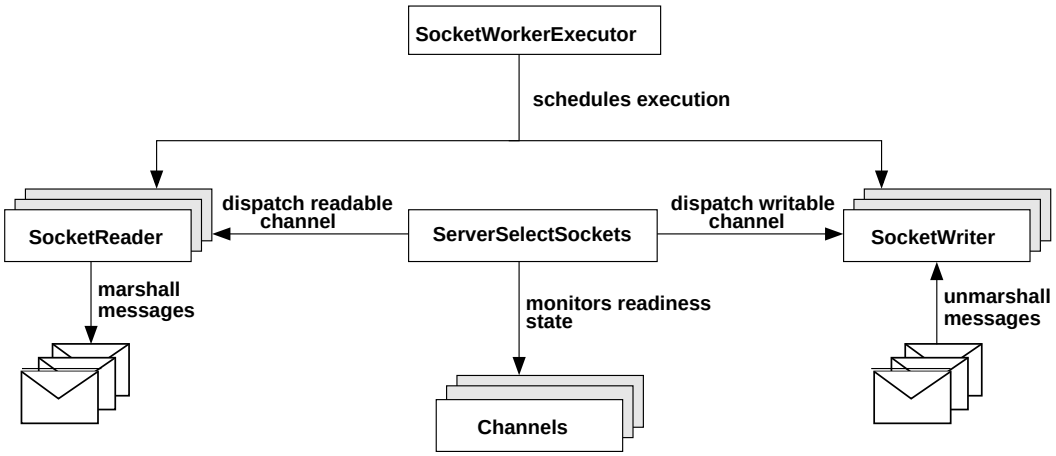
sind. Der Executor verwaltet einen Thread-Pool zur Ausführung der Socket-Worker. Es wäre in keinem Fall sinnvoll, sobald Daten von einem Kanal gelesen oder geschrieben werden müssen, jedesmal einen neuen Thread zu starten. Denn dies würde unter Last zu einem sinnlosen Kontext-Switching zwischen den einzelnen Threads führen, weil dann zu viele Threads im System sind. Genauso wenig wäre es sinnvoll das Lesen und Versenden der Nachrichten von einem einzigen Thread durchführen zu lassen. Dann könnte das System unter Last keine Anfragen mehr entgegennehmen. Deshalb wird mit einem Thread-Pool gearbeitet, der außerdem über einige Parameter verfügt, die sich an die Hardware und die Anwendung anpassen lassen.

- `int corePoolSize`: Die Grundgröße des Thread-Pool. Spezifiziert die Anzahl der Threads, die immer in dem Pool gehalten werden, bzw. verfügbar sind.
- `int maximumPoolSize`: Die maximale Größe des Pool, d.h. der Pool enthält höchstens `maximumPoolSize` Threads.
- `int queueSize`: Die Größe der Schlange, die zum vorübergehenden Speichern unerledigter Tasks verwendet wird. Wenn ein Worker nicht direkt einem Thread zugeordnet werden kann, wird er in einer Queue zwischengespeichert.
- `long keepAliveTime`: Die Zeit in Millisekunden, bevor Threads, die nicht zum Grundbestand des Pooles gehören, beendet werden. Das bedeutet, wenn es mehr als `corePoolSize` Threads gibt, und diese nicht länger als `keepAliveTime` nicht benötigt werden, werden diese beendet.
- `RejectedExecutionHandler rejHandler`: Dabei handelt es sich um die Strategie, die eingesetzt wird, falls keine Ressourcen mehr verfügbar sind. Wenn sich `maximumPoolSize` Threads bereits in Ausführung befinden, und die Queue voll ist, dann wird gemäß dieser Strategie vorgegangen. Die Voreinstellung dieses Parameters ist wie folgt: Wenn kein Thread mehr frei ist und die Queue voll ist, dann wird der Worker von dem Thread, der die Task an den Pool übergeben hat ausgeführt. Dies hat den Effekt, dass der aufrufende Thread ausgebremst wird, und so schnell keine neuen Tasks in die Queue stellen kann. Dies kann dazu führen, dass sich das System selbständig reguliert.

7.2.2 Transaktionsmanagementkomponente

Zentrales Element der Transaktionsmanagementkomponente sind die `TransactionManager`-Implementierungen. Die abstrakte Oberklasse und weitere Hilfsklassen befinden sich im Paket `acc.transactions`. Backend und Cache implementieren jeweils eine eigene Subklasse. Ein Transaktionsmanager koordiniert grundsätzlich alle lokalen Abläufe im Transaktionsmanagement. Der Transaktionsmanager auf der Backend-Seite verwaltet, gemäß der Konzeption, alle globalen Transaktionen des

Abbildung 7.3 Die IO-Management-Komponente

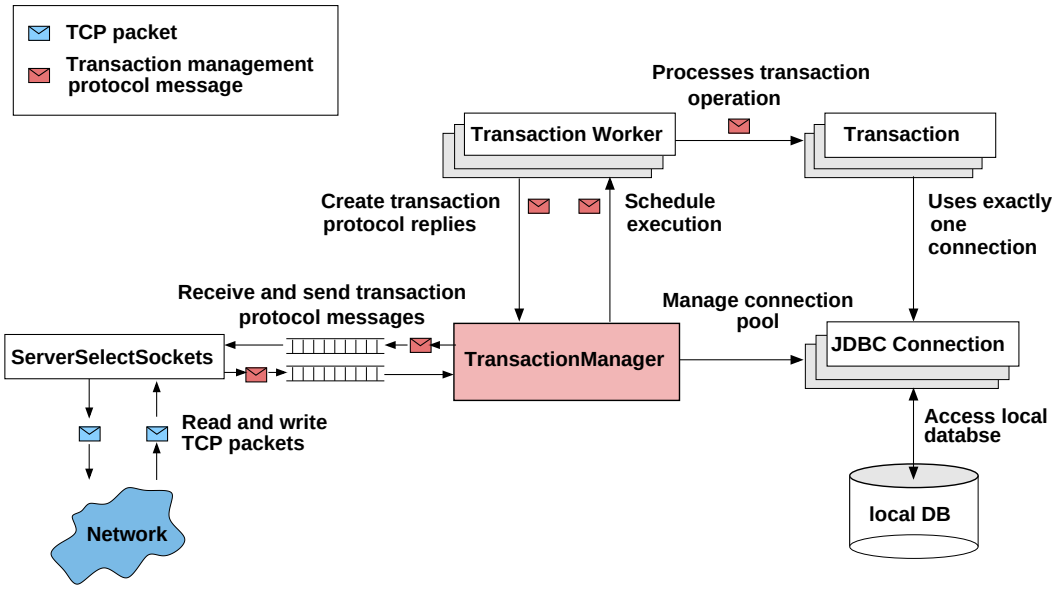


ACCache-Systems. Abbildung 7.4 gibt einen Überblick über die Transaktionsmanagementkomponente und illustriert zusätzlich das Zusammenspiel mit den IO-Komponenten.

Ein Transaktionsmanager benutzt die im vorigen Abschnitt beschriebene IO-Komponente, um die Nachrichten des Transaktionsmanagement-Protokolls zu empfangen und zu versenden. Neu empfangene Nachrichten werden zur weiteren Verarbeitung in einer FIFO-Schlange zwischengespeichert. Alle Protokollnachrichten verfügen über einen Typ und zu jedem Nachrichtentyp gibt es eine Worker-Implementierung. Der Manager ist für das Routing der empfangenen Nachrichten zu einem entsprechenden Worker verantwortlich. Jeder Nachrichtentyp korrespondiert mit einer grobgranularen Transaktionsoperation, wie beispielsweise der globalen BOT- oder Commit-Operation. Der Worker implementiert die Logik zur Ausführung dieser Operation und zum Generieren einer entsprechende Protokoll-Antwort. Diese übergibt er dem Transaktionsmanager, der das Routen der Nachricht zum korrekten Empfänger übernimmt. Das Scheduling zwischen Senden und Empfangen von Nachrichten soll fair sein. Deshalb werden die Nachrichten, die zum Versenden ausstehen, in der gleichen FIFO-Schlange gespeichert, wie die empfangenen Nachrichten. Das korrekte Routen der Nachrichten erfordert es, dass jede Protokoll-Nachricht in einen "Umschlag" vom Typ `acc.transactions.MessageEnvelope` verpackt wird. Dieser speichert unter anderem die Information, ob die Nachricht gelesen oder versendet werden muss. Vor der Übergabe einer Nachricht an die Netzwerk-IO-Komponente wird der Umschlag wieder entfernt.

Auch Protokoll-Nachrichten-Worker sind generell zustandslos, sodass jeder Worker alle Nachrichten seines Typ verarbeiten kann. Außerdem implementieren sie `Run-`

Abbildung 7.4 Transaktionsmanagement



nable, sodass sie in einem separaten Thread von einem `java.util.concurrent.Executor` ausgeführt werden können. Dabei werden alle Worker bis auf zwei Ausnahmen über einen Thread-Pool ausgeführt. Da das Parallelisieren der BOT- und endgültigen Commit-Operationen auf dem Backend nicht möglich ist, werden diese nur von einem einzigen Thread ausgeführt. Dadurch ist sichergestellt, dass diese Operationen echt sequentiell stattfinden. Die jeweiligen vom ACCache-System im Rahmen dieser Operationen ermittelten BOT- und Commit-Zeitpunkte entsprechen dann zwar nicht exakt denen auf der Backen-Datenbank, sie sind aber ausreichend, um die Nebenläufigkeit zweier Transaktionen festzustellen. Das sequentielle Ausführen der Commit-Operation auf dem Backend beinhaltet nicht das sequentielle Ausführen der Invalidierungsphasen.

Der Transaktionsmanager speichert alle für das Transaktionsmanagement relevanten Informationen. Dies hat zwei Gründe:

- Die Worker sollen zustandslos sein.
- Der Transaktionsmanager soll alle wichtigen Informationen zentral verwalten, und alle wichtigen Abläufe kontrollieren.

Diese Informationen umfassen alle aktiven Transaktionen, Clients und lokale Datenbankverbindungen. Dabei werden alle Datenbankverbindungen in einem Pool gehalten. Weil jede Transaktion lokal genau eine `JDBC-Connection` verwendet, kontrol-

liert der Transaktionsmanager die Zuordnung von Transaktion zu physischer Verbindung.

7.2.3 Transaktionsmanagementprotokoll

In dem Paket `acc.transactions.protocol` befinden sich die Klassen der Protokoll-Implementierung. Jede protokollkonforme Nachricht besitzt einen 20 Byte großen Header. Die eigentlichen Daten müssen in einem serialisierbaren Java-Objekt gekapselt werden. Dazu enthält der Header ein Feld, welches die Größe dieses serialisierten Objektes in Bytes aufnimmt. Zum Transport der Nutzdaten wurden jeweils leichtgewichtige Data Transfer Objects (kurz: DTOs) implementiert, um die Nachrichtengröße in einem annehmbaren Rahmen zu halten, wie beispielsweise die Klassen `ClientContext` und `GlobalTransactionContext` des Paketes `acc.transactions`. Jede Nachricht wird als Bytestrom versendet. Dieser wird gebildet aus dem Header, gefolgt von den Bytes des serialisierten Objektes.

Für die Weiterverarbeitung einer Protokollnachricht werden Header und Nutzdatenobjekt in abstrakteren Java-Objekten gekapselt. Die höheren Schichten können dann mit den entsprechend höherwertigen Objekten arbeiten. Die IO-Managementkomponente übernimmt dabei das Transformieren zwischen dem Bytestrom und den höherwertigen Objekten. Eine Nachricht des Transaktionsmanagement-Protokolls ist vom Typ `Message`. Ein Ausschnitt der Subklasse `MessageImpl` ist in Listing 7.4 dargestellt.

Listing 7.4 Ausschnitt aus der `MessageImpl`-Implementierung

```
public class MessageImpl implements Message {

        // message header
        private MessageHeader header;

        // message payload
        private MessageData data;

. . .
```

Ein Nachrichten-Header der momentanen Implementierung enthält die folgenden festen Felder:

- `long taId`: Speichert die global eindeutige ID der Transaktion.
- `int cacheId`: Speichert eine intern eindeutige ID des Cache, auf dem die Transaktion gestartet wurde. Eine ID gleich Null bedeutet, dass die Transaktion von

einem Client, der direkt mit dem Backend verbunden ist, gestartet wurde. IDs größer Null identifizieren einen Cache.

- `MessageType type`: Speichert den Nachrichtentyp, in einen `int`-Wert kodiert (siehe auch Listing 9.1.2).
- `int dataSize`: Speichert die Größe der Nutzdaten.

Dabei wurde eine Beschränkung auf die grundlegenden Informationen einer Nachricht genommen. Alle wichtigen Informationen, die schnell zugreifbar und immer vorhanden sein müssen, werden im Header gespeichert. Der Header erleichtert das Weiterleiten der Nachricht an die korrekte Softwarekomponente. Die bereits erwähnten Nachrichtentypen werden vollständig in dem Java-Enum `MessageType` aufgezählt und werden im Header als `int`-Wert kodiert. In Listing 9.1.2 ist ein Ausschnitt mit den wichtigsten Nachrichtentypen gegeben.

7.2.4 Invalidatorkomponente

Die Invalidator-Komponente befindet sich in dem Paket `acc.transactions.invalidations`. Das Management der Invalidierungen wird auf Cache-Seite durch einen `RCCValueManager` kontrolliert. Der Manager verwaltet alle ungültigen RCC-Werte der Cache Group. Für jedes Paar aus RCC und Wert gibt es genau ein `RCCValue`-Objekt. Dieses speichert intern die Anzahl aller angeforderten Invalidierungen dieses RCC-Wertes. Falls für diesen RCC-Wert eine lange Invalidierung durch eine Transaktion erforderlich gewesen ist, wird zusätzlich der globale-Commit-Zeitpunkt der Transaktion gespeichert. Das Invalidieren und Revalidieren der RCC-Werte liegt in der ausschließlichen Verantwortung dieses Managers. In Listing 7.5 ist ein Ausschnitt aus der Schnittstelle dieser Klasse gegeben. Zur Ableitung der eigentlichen Invalidierungen aus einem Write Set wird ein `Invalildator` benötigt. Listing 7.6 enthält die Schnittstelle dieser Invalidierer.

7.2.5 Anpassungen an existierenden Komponenten

Zum einen mussten transaktionsfähige und lokale Query-Worker-Implementierungen entwickelt werden. Zum anderen musste eine Probing-Komponente entwickelt werden, die mit Invalidierungen umgehen kann, und diese beachtet. Dabei konnte das Verarbeiten von RCC-Wert-Invalidierungen beim Probing nicht vollständig umgesetzt werden. RCC-Wert-Invalidierungen stellen einen erheblichen Eingriff in den existierenden, einfachen und effizienten Probingmechanismus dar. Bisher musste lediglich einen Einstiegspunkt gefunden werden, und für jeden Join die entsprechende RCC-Bedingung überprüft werden. Anstelle dieser RCC-Bedingung müsste nun sichergestellt werden, dass bei jedem Join kein Tupel in der Quelltabelle involviert ist, welches in einer Quellspalte einen invalidieren Wert besitzt. Dies ist ohne einen *RCC-Index* nur schwer möglich, da der ACCache die Auswertung von Joins dem

Listing 7.5 Ausschnitt aus der `RCCValueManager`-Schnittstelle

```
public class RCCValueManager {

        // invalidate
        public synchronized Set<RCCValue> invalidate(
                Map<RCC, Set<Object>> shortInvalidations,
                Map<RCC, Set<Object>> longInvalidations,
                long coTS);

        // revalidate
        public synchronized void cancelInvalidations(
                Set<RCCValue> invalidations);

        // retrieve all invalid rccs
        public synchronized Set<RCC> getAllInvalidRCCs();

        // retrieve all rcc value invalidations
        public synchronized Set<RCCValue> getAllInvalidRCCValues();

        // get singleton instance
        public static RCCValueManager getInstance();
}
```

Datenbanksystem überlässt. Ein RCC-Index würde zu jedem RCC, alle an ihm anliegenden Werte speichern.

Das Konzipieren und Implementieren eines erweiterten RCC-Index, der im Probing effizient eingesetzt werden kann, war im Rahmen dieser Arbeit nicht möglich. Deshalb wird jede RCC-Bedingung als unerfüllt angesehen, falls mindestens eine Wert-Invalidierung für diesen RCC existiert. Die einzige Ausnahme bildet hier die Suche nach einem Einstiegspunkt. Hier können RCC-Wert-Invalidierungen einfach eingesetzt werden. Alle Werte, die am RCC nicht invalidiert sind, und die in der Quellspalte existieren, stellen einen Einstiegspunkt dar.

7.3 Zusammenfassung

In diesem Kapitel wurde die prototypische Umsetzung des Verfahrens vorgestellt. Um eine gut skalierende Server-Architektur zu erzielen wurde ein effizient arbeitender IO-Layer für die Netzwerkein- und Ausgabe implementiert. Da es sich um eine Transaktionsmanagementkomponente handelt ist außerdem mit häufigen Blockierungen beim Zugriff auf die Externspeichermedien zu rechnen. Um trotzdem eine gute

Listing 7.6 Die `Invalidator`-Schnittstelle

```
public interface Invalidator {

        public Map<RCC, Set<Object>> determineInvalidations(
                WriteSet ws) throws WriteSetCorruptedException;

        public CustomWriteSet buildCustomWriteSet(
                WriteSet ws) throws WriteSetCorruptedException;

        public void determineInvalidations(
                CustomWriteSet cws,
                Map<RCC, Set<Object>> shortInvalidations,
                Map<RCC, Set<Object>> longInvalidations);
}
```

Konsolidierung zu erreichen, wurde mit Thread-Pools gearbeitet. Durch den Thread-Pool können je nach Anfragesituation beliebige zustandslose Worker ausgeführt werden. Der Transaktionsmanager ist das Herzstück, er koordiniert alle Vorgänge und verwaltet alle wichtigen Informationen.

Leider konnte die Implementierung im Rahmen der Diplomarbeit nicht vollständig abgeschlossen werden. Das Ausführen von Updates innerhalb einer JDBC-Transaktion wird aber bereits unterstützt. Die Transaktion wird auf globaler Ebene mit Snapshot Isolation isoliert. Das heißt sie sieht ausschließlich die Versionen ihres globalen Snapshot. Schreiber sehen alle ihre Änderungen. Bei Commit sind auf den Caches alle RCC-Wert-Invalidierungen der schreibenden Transaktion gültig. Diese werden allerdings noch nicht wieder aufgehoben. Die Implementierung der Ausführungs-und Invalidierungsphase wurde fertiggestellt, wohingegen die Revalidierungs-Phase noch abgeschlossen werden muss. Die Funktionalität zum Anwenden eines Write-Set liegt bereits vor, sie wird allerdings vom Transaktionsmanager noch nicht angestossen. Vollständig zu Implementieren wäre noch das Nachladen der referenzierten Tupel, aufgrund einer Änderung auf einer Spalte mit ausgehenden RCCs (vgl. Kapitel 6.7.2). Da im Prototyp bereits das Laden ganzer Cache Units implementiert ist, sollte diese Teilaufgabe keine größeren Schwierigkeiten bereiten.

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Synchronisationsverfahren für das Constraint-basierte Datenbank-Caching entwickelt. Außerdem wurde mit einer ersten prototypischen Umsetzung begonnen. An das Verfahren wurden die folgenden Anforderungen gestellt:

- Transparenz
- ACID-Transaktionen
- 1-Copy-Serializability
- Skalierbarkeit
- Effizienz
- Middleware-Architektur

Zur abschließenden Bewertung soll noch einmal kurz auf die einzelnen Punkte und deren Realisierung eingegangen werden.

Transparenz Transparenz wurde im vollen Umfang erreicht. Benutzer des ACCache-Systems brauchen nicht zu unterscheiden, ob sie auf das Backend oder den Cache zugreifen. Schreibende Transaktionen müssen nicht gesondert behandelt werden. Weiterhin wurde ein JDBC-Treiber entwickelt, sodass der Anwendungsprogrammierer mit der vertrauten JDBC-API arbeiten kann.

ACID-Transaktionen Diese Anforderung konnte nicht vollständig umgesetzt werden. Isolierung, wie sie im Sinne des ACID-Paradigmas definiert ist, kann nicht erreicht werden. Vollständiger ACID-Schutz würde Serialisierbarkeit auf globaler Ebene erfordern. Es kann aber nur Snapshot Isolation bereitgestellt werden. Die verbleibenden drei ACID-Eigenschaften A, C und D werden aber eingehalten.

1-Copy-Serializability Wurde nicht erreicht. Dies ergibt sich bereits aus dem vorangegangenen Punkt.

Skalierbarkeit Dieser Punkt ist schwierig zu beurteilen und es konnte zum Erstellungszeitpunkt noch keine endgültige Aussage getroffen werden. Es wurde viel Aufwand in diese Anforderung investiert. Beispielsweise wurde die Konfliktrate eingehend untersucht. Aus den gewonnen Erkenntnissen wurde versucht der stark steigenden Konfliktrate, wie sie bei einem replizierten System zu

erwarten ist, entgegenzuwirken. Unter anderem durch die Entwicklung eines halbsynchronen Vefahrens zum Propagieren von Änderungen. Aber auch der Einsatz des Mehrversionenverfahrens liegt hier begründet. Ziel war es, Transaktionen möglichst wenig durch das ACCache-System zu blockieren. Dazu wurden auch die Stellen untersucht, welche einer zwingenden Synchronisation durch das ACCache-System bedürfen, sodass überall, wo dies nicht erforderlich ist, eine Parallelisierung möglich ist. Der Punkt Skalierbarkeit spielte auch eine wichtige Rolle bei der prototypischen Implementierung. Beispielsweise wurde Java NIO eingesetzt, um eine gut skalierende Serverarchitektur zu gewährleisten. Der Kommunikationsaufwand konnte teilweise minimiert werden, durch die Entwicklung eines eigenen Protokolls und schlanke Kommunikations-Übertragungsobjekte. Zur abschließenden Beurteilung der Skalierbarkeit bedarf es eingehender Leistungsmessungen. Diese konnten aber im Rahmen dieser Arbeit nicht mehr durchgeführt werden.

Effizienz Diese Anforderung ist ähnlich schwierig zu bewerten, wie die vorangegangene. Aus Sicht des Autors wurde sie umgesetzt, sowohl bei der Konzeption, als auch bei der Implementierung. Das Verfahren wurde speziell für das Constraint-basierte Datenbank-Caching entwickelt und ist darauf zugeschnitten. Die Vorteile, die aus dem Caching-Ansatz, in Abgrenzung zum Replikationsansatz, gezogen werden können wurden ausgenutzt. Durch das halbsynchrone Propagieren der Änderungen kann die Commit-Operation als effizient angesehen werden. Es wurde stets versucht Latenz zu vermeiden. Die Implementierung eines eigenen Protokolls unterstützt dies zusätzlich.

Middleware-Architektur Bei dem entwickelten Verfahren handelt sich nicht um eine proprietäre Erweiterung eines existierenden Datenbanksystems. Es wurde so konzipiert, dass eine Umsetzung möglich ist, ohne auf bestimmte Features eines Datenbanksystemherstellers angewiesen zu sein. Somit wurde dies Anforderung in vollem Maße berücksichtigt.

Was zu tun bleibt ist der Abschluss der prototypischen Implementierung. Die Ausführungs- und Invalidierungsphase wurden vollständig implementiert. Dabei wurde auf die Umsetzung einer hochskalierenden Server-Architektur geachtet. Das Probing wurde so angepasst, dass Invalidierungen beachtet werden. Die Revalidierungsphase konnte nicht mehr abgeschlossen werden. Trotzdem werden nun Update-Operationen im Rahmen einer AC(I)D-Transaktion von dem ACCache-System unterstützt. Alle Invalidierungen liegen dann als RCC-Wert-Invalidierungen (inklusive langen RCC-Wert-Invalidierungen) vor, werden allerdings noch nicht wieder aufgehoben. Der Programmcode zum Anwenden eines Write Set ist bereits vorhanden. Allerdings wird dieser Prozess noch nicht angestossen. Vollständig zu implementieren wäre noch das Nachladen der Tupel, die zur Beseitigung eventuell entstandener Inkonsistenzen in der Cache Group benötigt werden, sowie das Aufheben der Invalidierungen. Dabei muss beachtet werden, dass der letzte Vorgang mit den BOT-Operationen zu synchronisieren ist, wie es in Beobachtung 6.1) festgehalten wurde. Außerdem müssen,

wie in Kapitel 6.8.1 beschrieben, die langen RCC-Wert-Invalidierungen beachtet werden. Bei dem Anwenden von Write Sets muss außerdem an die Cache-Lost-Updates gedacht werden (vgl. Kapitel 6.8.2).

Insgesamt wurde aber gezeigt, dass Update-Operationen im Rahmen des Constraint-basierten Datenbank-Caching möglich sind. Das ACCache-System wurde transaktionsfähig gemacht, und dabei wurde eine Konsistenzstufe erzielt, die leicht schwächer ist als Serialisierbarkeit. Insgesamt kann in diesem Bereich von dem Constraint-basierten Datenbank-Caching noch viel erwartet werden, weil es einige Vorteile gegenüber der reinen Replikation besitzt.

9 Anhang

9.1 Listings

9.1.1 Generierung des Prepared-Statements in Update.java

```
// map the parameters of the prepared statement by their indices
Map<Integer, Object> parameters = new HashMap<Integer, Object>();
int index = 1;

// generate UPDATE statement
StringBuilder stmt = new StringBuilder("UPDATE ");
stmt.append(this.tableName);

// SET clause
stmtm.append(" SET ");

// iterate over the columns that have been updated
Iterator<Name> colNameIt = this.newValues.keySet().iterator();

// there must be at least one  column, that has been updated
Name firstName = colNameIt.next();
stmt.append(firstName).append(" = ?");
parameters.put(index++, this.newValues.get(firstName));

while(colNameIt.hasNext()) {
        Name colName = colNameIt.next();
        stmt.append(", ").append(colName).append(" = ?");
        parameters.put(index++, this.newValues.get(colName));
}

// WHERE clause
stmt.append(" WHERE ");

// iterate over the primary key columns
PrimaryKey pk = this.primaryKey;
Map<Name, Object> primaryKeyValues = pk.getPrimaryKeyValues();
colNameIt = primaryKeyValues.keySet().iterator();
```

```
// there must be at least one primary key column
firstName = colNameIt.next();
stmt.append(firstName).append("␣=␣?␣");
parameters.put(index++, primaryKeyValues.get(firstName));

while(colNameIt.hasNext()) {
        Name colName = colNameIt.next();
        stmt.append("AND␣").append(colName).append("␣=␣?␣");
        parameters.put(index++, primaryKeyValues.get(colName));
}

return new PreparedStatementData(stmt.toString(), parameters);
```

9.1.2 Ausschnitt aus der Enum `MessageType`

```
public enum MessageType {

        // Request new global BOT.
        BOT_REQUEST(1),

         // Global BOT request has been processed.
         // Message data contains new
         // global transaction context.
        BOT_REPLY(2),

        // Request global commit.
        CO_REQUEST(3),

         // Global commit operation has been successfull.
        CO_REPLY(4),

        // Request global abort.
        ABORT_REQUEST(5),

        // Global abort operation has been succesfull.
        ABORT_REPLY(6),

         // Invalidation request. Message data contains
         // write set / invalidations of a transaction.
        INV_REQUEST(7),

        // Invalidations have been performed.
        // Cache is in prepared state.
        INV_REPLY(8),

        // Execute query request.
        EXEC_QUERY_REQUEST(9),
```

```
        // Query has been executed.
        // Message Data contains result set.
        EXEC_QUERY_REPLY(10),

        // Execute update request.
        EXEC_UPDATE_REQUEST(11),

         // Update has been executed.
         // Message data contains number
         // of updated rows (and invalidations).
        EXEC_UPDATE_REPLY(12),

        // Error occured.
        ERROR_REPLY(13),

        // Registration requested by a cache.
        REGISTER_CACHE_REQUEST(14),

        // Cache registration was successfull.
        REGISTER_CACHE_REPLY(15),

        // Registration requested by a client.
        REGISTER_CLIENT_REQUEST(17),

        // Client registration  was successfull.
        REGISTER_CLIENT_REPLY(18),

// . . .
}
```

Abbildungen

Definitionen

Beobachtungen

Literaturverzeichnis

[ACC] http://lgis.informatik.uni-kl.de/cms/index.php?id=104.

[Ame92] American National Standards Institute. *ANSI X3.135-1992, American National Standard for Information Systems – Database Language – SQL*, nov 1992.

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. In Michael J. Carey and Donovan A. Schneider [CS95], pages 1–10.

[Bra08] Susanne Braun. Entwicklung von automatisierten Tests zur Vermessung der Lade- und Entladecharakteristika von Cache-Units, März 2008. Projektarbeit, AG Datenbanken und Informationssysteme, Technische Universität Kaiserslautern, Fachbereich Informatik.

[Büh06] Andreas Bühmann. Ein Schritt zurück ist kein Rückschritt. *Inform., Forsch. Entwickl.*, 20(4):184–195, 2006.

[CS95] Michael J. Carey and Donovan A. Schneider, editors. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*. ACM Press, 1995.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In H. V. Jagadish and Inderpal Singh Mumick [JM96], pages 173–182.

[HB07] Theo Härder and Andreas Bühmann. Value Complete, Column Complete, Predicate Complete – Magic Words Driving the Design of Cache Groups. *The VLDB Journal (Online First)*, Januar 2007.

[Hit02] Ron Hitchens. *Java NIO*. O'Reilly, 2002.

[HR83] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung*. Springer, 2001.

[IEE01] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue*

6. 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[JM96] H. V. Jagadish and Inderpal Singh Mumick, editors. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. ACM Press, 1996.

[Kle06] Joachim Klein. Entwicklung einer automatisierten Messumgebung für das Constraint-basierte Datenbank-Caching. Master's thesis, Technische Universität Kaiserslautern, Fachbereich Informatik, AG Datenbanken und Informationssysteme, Oktober 2006.

[KMB] Joachim Klein, Gustavo Machado, and Andreas Bühmann. Customizable Garbage Collection for Constraint-based Database-Caching. Internes Paper der AG Datenbanken und Informationssysteme, Technische Universität Kaiserslautern, Fachbereich Informatik.

[LKPMJP05] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In Özcan [Özc05], pages 419–430.

[Mer05] Christian Merker. Konzeption und Realisierung eines Constraint-basierten Datenbank-Cache. Master's thesis, Technische Universität Kaiserslautern, Fachbereich Informatik, AG Datenbanken und Informationssysteme, Oktober 2005.

[Özc05] Fatma Özcan, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 2005.

[Thi07] Julia Thiele. Grundlegende Aktualisierungsprobleme beim Constraint-basierten Datenbank-Caching. Master's thesis, Technische Universität Kaiserslautern, Fachbereich Informatik, AG Datenbanken und Informationssysteme, September 2007.