Stets Wertvollständig! – Snapshot Isolation für Transaktionen beim Constraint-basierten Datenbank-Caching

Joachim Klein

AG Datenbanken und Informationssysteme
Fachbereich Informatik
Technische Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
jklein@informatik.uni-kl.de

Abstract: Das Constraint-basierte Datenbank-Caching (CbDBC) erlaubt es, feingranular und dynamisch, Satzmengen häufig verwendeter Prädikate, in der Nähe von Anwendungen vorzuhalten, um lesende Anfragen zu beschleunigen. Dabei lässt sich die Vollständigkeit bzgl. der Anfrageprädikate anhand einfacher Bedingungen (den Constraints) ableiten, welche alle auf dem zentralen Konzept der Wertvollständigkeit aufbauen. Durch das Kopieren der Daten auf den Cache entstehen Replikate, deren Konsistenz zu gewährleisten ist. Gleichzeitig muss jedoch auch die Wertvollständigkeit der Constraints jederzeit gewahrt sein. Wie lässt sich unter diesen Bedingungen für Transaktionen eine akzeptable Isolationsstufe erreichen, wenn der Zugriff auf die primäre Datenbank aufgrund der hohen Latenz teuer und daher zu vermeiden ist? Dieser Aufsatz zeigt, wie sich die Vollständigkeit ganzer Satzmengen im CbDBC wahren lässt, ohne den durch das Caching erreichten Vorteil aufzugeben. Dabei garantiert die für das CbDBC angepasste Synchronisation die Isolationsstufe *Snapshot Isolation* und erlaubt eine verzögerte (lazy) Aktualisierung der Replikate.

1 Motivation

Datenbank-Caching beschleunigt den lesenden Zugriff auf entfernte Datenbanken, indem es Satzmengen, die zur Beantwortung häufig gestellter Anfragen benötigt werden, in der Nähe der betreffenden Anwendungen zur Verfügung stellt. Für alle Datenbank-Caching-Verfahren [LGZ04, The02, ABK$^+$03, APTP03, BDD$^+$98, LAK10] ist dabei festzulegen, wie die Vollständigkeit der benötigten Sätze bereits im Cache (durch einen *lokalen* Zugriff) bestimmt werden kann. Constraint-basierte Datenbank-Caching (CbDBC) verwendet dazu einfache Bedingungen (*Constraints*), welche die Vollständigkeit von Sätzen bezüglich eines Wertes garantieren. Dies erlaubt es, die einer Anfrage vorgeschaltete Vollständigkeitsprüfung (*Probing* [HB07]) durch die Abfrage einzelner Werte zu realisieren, ohne die benötigten Sätze komplett zu lesen. Gleichzeitig bietet diese Vorgehensweise die Möglichkeit, feingranular und dynamisch zu entscheiden, von welchen Cache-Inhalten eine hohe Lokalität zu erwarten ist.

Eine große Herausforderung beim Datenbank-Caching ist die Realisierung von Synchronisationsverfahren, die für Transaktionen eine hohe Isolationsstufe garantieren. Aufgrund der hohen Latenz zwischen dem zentralen Datenbestand (*Backend*) und dem Cache wurden in der Vergangenheit Ansätze vorgeschlagen, die es erlauben, die Konsistenz stark abzuschwächen, um eine größere Skalierbarkeit zu erreichen [GLRG04a, GLRG04b]. Diese Ansätze haben jedoch zwei entscheidende Nachteile: Einerseits ist es oft nötig, die gewünschte Aktualität der Daten in Anfragen zu spezifizieren, was die Unabhängigkeit der Anwendung vom Datenbanksystem stark beeinträchtigt, und andererseits wird dabei keine wohldefinierte Isolationsstufe erreicht.

Ebenso wie das CbDBC-Verfahren halten auch partielle Replikationsverfahren nur einen Teil der Gesamtdatenmenge als Replikat vor. Die dabei verwendeten Synchronisationsmechanismen bilden deshalb zwar einen wichtigen Startpunkt für die Suche nach einem geeigneten Verfahren, aber die beim CbDBC vorherrschende Dynamik erschwert zumindest die Implementierung solcher Lösungen (vgl. Abschnitt 4).

Durch das redundante Vorhalten der Daten im Cache entstehen (dynamisch veränderliche) Replikate, weshalb die Kontrolle der verteilen Replikate zwingend in den Synchronisationsmechanismus integriert werden muss. Abschnitt 3 beantwortet deshalb kurz die grundlegenden Fragen zur Replikatskontrolle. Eine genaue Diskussion hierzu findet sich in [Kle10].

Zentral für die Auswahl einer geeigneten Nebenläufigkeitskontrolle ist die Forderung, dass der durch das Caching erreichte Geschwindigkeitsvorteil nicht wieder verloren gehen darf. Daher muss ein Verfahren gewählt werden, welches sowohl lesende Zugriffe als auch den Transaktionsabschluss möglichst nicht behindert. Wir konzentrieren uns beim CbDBC auf Verfahren, Snapshot Isolation für Transaktionen garantieren. In Abschnitt 4 erläutern wir diese Entscheidung und erklären die benötigten Grundlagen der Snapshot Isolation in Abschnitt 4.1.

Nachfolgend zeigen wir in Abschnitt 5, wie sich Snapshot Isolation im CbDBC garantieren lässt. Dabei stützen wir uns auf die zentrale Forderung aus [Kle10], auf ältere Snapshots im Backend zugreifen zu können. Der gezeigte Synchronisationsmechanismus erlaubt eine verzögerte (*lazy*) Aktualisierung der Replikate und kommt (abgesehen davon, dass die Änderungen einer Transaktion zum Cache zu übertragen sind) ohne zusätzliche Kommunikation aus. Insbesondere wird kein 2-Phasen-Commit-Protokoll (2PC) zum Abschluss einer Transaktion benötigt.

Bevor wir dies jedoch genau betrachten, wiederholen wir zunächst die zum Verständnis nötigen Grundlagen des CbDBC.

2 Grundlagen des CbDBC

Das Constraint-basierte Datenbank-Caching (CbDBC) beschleunigt Anfragen, indem es Satzmengen häufig angefragter Prädikate in Anwendungsnähe vorhält. Dabei werden die Sätze in sogenannten *Cache-Tabellen* gespeichert und stammen aus einer primären Datenbank, dem Backend. Zu jeder Cache-Tabelle T gehört immer genau eine Backend-Tabelle

T_B. Dabei entspricht die Definition einer Cache-Tabelle der ihr zugeordneten Backend-Tabelle bis auf die Fremdschlüsseldefinitionen, die nicht übernommen werden.

Damit der Cache zur Beantwortung eines Prädikats P einer Anfrage Q benutzt werden kann, muss die *Prädikatsextension* von P (dies umfasst alle Sätze, die zur Auswertung von P benötigt werden) im Cache vollständig vorhanden sein. Ist dies der Fall, so ist ist der Cache hinsichtlich P *prädikatsvollständig* und kann die Beantwortung von Q übernehmen.

Um die Prädikatsvollständigkeit effizient überprüfen und herstellen zu können, benötigt das CbDBC nur zwei Constraint-Typen: Den *referenziellen Cache-Constraint (RCC)* und die sogenannte *Füllspalte (filling column, FC)*. Beide bauen auf dem Konzept der *Wertvollständigkeit* auf, welche die Grundlage des Constraint-basierten Ansatzes bildet.

Definition 2.1 (Wertvollständigkeit) *Ein Wert w einer Spalte $T.a$ ist genau dann wertvollständig (oder kurz vollständig), wenn alle Sätze $\sigma_{a=w}T_B$ im Cache verfügbar sind.*

Ein RCC $S.a \rightarrow T.b$ kann zwischen zwei Spalten mit gleichem Wertebereich definiert werden. Er garantiert jederzeit die Wertvollständigkeit in $T.b$ für alle Werte w aus der *Quellspalte* $S.a$. Hierbei ist zu beachten, dass die Wertvollständigkeit nur für $T.b$, die sogenannte *Zielspalte*, garantiert wird. Dies erlaubt z. B. die Auswertung des Gleichverbundes $S \bowtie_{a=b} T$, falls die Wertvollständigkeit für einen Wert w in S (z. B. für w in $S.c$) gegeben ist, sodass die Anfrage $\sigma_{S.c=w}(S \bowtie_{a=b} T)$ das korrekte Ergebnis liefert. Im umgekehrten Fall (Wertvollständigkeit ist gegeben für einen Wert w aus T) ist die Auswertung des Verbunds nicht möglich[1].

Definition 2.2 (Referenzieller Cache-Constraint, RCC) *Ein RCC $S.a \rightarrow T.b$ von einer Quellspalte $S.a$ und zu einer Zielspalte $T.b$ ist genau dann erfüllt, wenn alle Werte w aus $S.a$ wertvollständig in $T.b$ sind.*

Jede Spalte einer Cache-Tabelle kann auch als Füllspalte ausgezeichnet werden. Mit Hilfe von Füllspalten wird festgelegt, wann und welche Werte vollständig in den Cache geladen werden. Hierzu verwaltet der Cache zu jeder Füllspalte (z. B. $S.f$) eine Menge von *Kandidatenwerten*, für die ein Ladevorgang ausgelöst werden darf.

Definition 2.3 (Füllspalte, FC) *Eine Füllspalte $S.f$ lädt einen Wert w wertvollständig, sobald w von einer Anfrage durch das Prädikat $S.f = w$ explizit referenziert wird und w ein Kandidatenwert ist.*

2.1 Cache Groups

Ein Constraint-basierter Datenbank-Cache verwaltet die für ihn definierten Cache-Tabellen und Constraints (RCCs und FCs) in einer sogenannten *Cache Group*. Eine Cache Group

[1] Aus diesem Grund darf ein RCC nicht mit einem Fremdschlüssel gleichgesetzt werden.

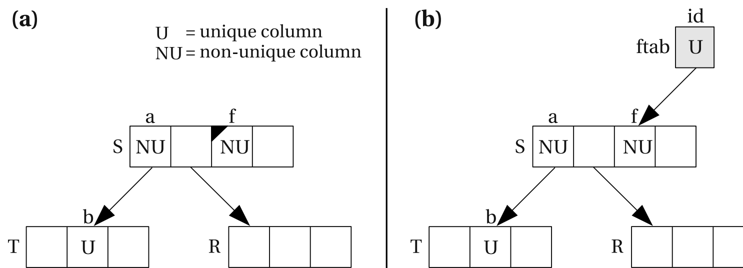

Abbildung 1: Konzeptionelle (a) und interne (b) Darstellung einer Cache Group

besteht dabei aus einer *Wurzeltabelle (root table)*, in der genau eine Spalte als Füllspalte deklariert ist und evtl. mehreren *Mitgliedstabellen (member tables)*, die über RCCs von der Wurzeltabelle abhängen. Ein einfaches Beispiel hierfür zeigt Abb. 1a, wobei S als Wurzeltabelle mit der Füllspalte $S.f$ darstellt und T bzw. R die Mitgliedstabellen über RCCs von S aus erreichbar sind. Um mehrere Prädikate gleichzeitig zu unterstützen, werden üblicherweise mehrere Cache Groups zu einer *Cache-Group-Föderation* zusammengefasst (vgl. [HB07]).

Um das Verhalten einer Füllspalte im Cache zu realisieren, wird für jede Füllspalte eine *interne Tabelle* angelegt. Im Gegensatz zu einer Cache-Tabelle hat eine interne Tabelle keine Beziehung zu einer Backend-Tabelle. Für die Füllspalte $S.f$ wird (wie in Abb. 1b gezeigt) eine interne Tabelle $ftab$ angelegt, deren Primärschlüssel, die Spalte $ftab.id$, über einen RCC mit der Füllspalte $S.f$ verbunden wird. Wird nun ein Ladevorgang für den Wert w der Füllspalte $S.f$ ausgelöst, so genügt es den Wert w in Spalte $ftab.id$ einzufügen. Der ausgehende RCC $ftab.id \rightarrow S.f$ erzwingt nachfolgend die erforderliche Wertvollständigkeit von w in $S.f$. Mit Hilfe dieser Vereinfachung kann die Cache-Verwaltung ausschließlich mithilfe von Tabellen und RCCs beschrieben werden.

2.2 Hülle eines Kontrollwertes

Wie zuvor für den Wert w, der durch Einfügung in $ftab.id$ die Wertvollständigkeit in $S.f$ erzwang, gezeigt wurde, kontrolliert jeder *RCC-Quellspaltenwert* die Wertvollständigkeit in den Zielspalten ausgehender RCCs. Aus diesem Grund nennen wir alle Werte, die in RCC-Quellspalten auftreten, *Kontrollwerte*.

Verfolgen wir das Beispiel bzgl. der Cache Group aus Abb. 1 weiter, so fällt auf, dass abhängig von w Sätze im Cache benötigt werden, die ihrerseits wieder Kontrollwerte (z. B. x, y, z in die Spalte $S.a$) einfügen. Die Werte x, y, z müssen dann wiederum in $T.b$ wertvollständig sein. Es entsteht eine rekursive Abhängigkeit von Sätzen unter Verfolgung der ausgehenden RCCs. Daher definieren wir die Menge aller Sätze, die abhängig von einem Kontrollwert w im Cache benötigt werden, als *Hülle des Kontrollwertes* und schreiben für die Hülle dieses Kontrollwertes $C(w)$.

Definition 2.4 (Hülle eines Kontrollwertes) *Sei w ein Kontrollwert des RCC $S.a \rightarrow T.b$ und daher $I = \sigma_{a=w} T_B$ die Menge aller Sätze, die in T wertvollständig vorliegen müssen. Die Hülle von w ist rekursiv definiert als Menge aller Sätze $C(w) = I \cup C(w_i)$, $\forall w_i \in W(I)$, wobei $W(I) = (w_1, ..., w_n)$ die Menge aller Kontrollwerte aus I beschreibt.*

3 Replikatskontrolle beim CbDBC

Durch das Kopieren von Sätzen in den Cache entstehen verteilte Replikate, deren Konsistenz zu gewährleisten ist. Dabei spielt die Frage, wo, wie und, vor allem, wann Replikate aktualisiert werden, für die Umsetzung der Nebenläufigkeitskontrolle eine wichtige Rolle. Darüber hinaus muss geklärt werden, auf welchen Replikaten Änderungen durchgeführt werden dürfen. Im Weg weisenden Aufsatz von Gray u. a. [GHOS96] werden die Verfahren daher auch im wesentlichen mittels zweier Parameter kategorisiert.

Der erste Parameter beschreibt, wo Änderungen ausgeführt werden dürfen, auf einer ausgezeichneten Kopie (*primary copy*) oder auf jeder Kopie (*everywhere*). Für das CbDBC wird ein Primary-Copy-Verfahren verwendet [Kle10], da das Backend in natürlicher Weise eine Primärkopie bereitstellt. Die Hauptproblematik besteht jedoch darin, dass die Zulässigkeit einer Update-Ausführung nicht allein durch eine Vollständigkeitsprüfung gewährleistet werden kann. Durch eine Änderung könnten z. B. Trigger ausgelöst oder Integritätsbedingungen verletzt werden. Die zugehörigen Metadaten müssten somit auch auf den Cache kopiert und synchronisiert werden. Darüber hinaus kann niemals sichergestellt werden, dass ein Cache alle Update-Anweisungen einer Transaktion bearbeiten kann. Zum Abschluss einer Transaktion lägen somit an zwei verschiedenen Stellen, im Cache und im Backend, Änderungsinformationen vor, die an andere Caches weiterzuleiten sind. Aus diesen Gründen sind Update-Everywhere-Verfahren für das CbDBC deutlich schwerer umzusetzen. Nach den Ergebnissen aus [GHOS96] benötigt man für Update-Everywhere-Verfahren entweder eine komplexe Nebenläufigkeitskontrolle oder Verfahren zur Konfliktauflösung, welche im CbDBC nicht automatisch, d. h. ohne Benutzerinteraktion, gewährleistet werden kann. Daher werden im CbDBC alle Update-Anweisungen an das Backend weitergeleitet, wo deren Ausführbarkeit stets überprüft werden kann.

Der zweite von Gray u. a. [GHOS96] verwendete Parameter definiert, wann die Replikate aktualisiert werden. Dies kann *direkt (eager)* oder *verzögert (lazy)* erfolgen. Direkt bedeutet hierbei, dass die Replikate vor Transaktionsabschluss (*Commit*) zu aktualisieren sind, wohingegen die verzögerte Aktualisierung irgendwann nach dem Commit (meist aber möglichst zeitnah) erfolgt.

Eager-Ansätze leiden generell darunter, dass sie den Transaktionsabschluss behindern, bis alle Replikate aktualisiert sind. Im CbDBC wird dieses Problem noch verstärkt, da zwischen dem Backend und den Caches eine hohe Latenz angenommen wird[2]. Es bietet sich daher an, eine verzögerte Aktualisierungsstrategie im CbDBC anzustreben. Entsprechende Ansätze leiden oft darunter, dass sie Inkonsistenzen erzeugen. Die höchste Isolationsstufe *Serialisierbarkeit* ist dabei meist nicht zu erreichen. Die aktuelle Entwicklung in For-

[2] Diese ist oft erst der Grund für den Einsatz eines Datenbank-Cache.

T_1 führt folgende Änderungen durch:

```
1) INSERT INTO Order_Lines
    VALUES (47,1,4);

2) UPDATE Order_Lines
    SET   IId = 22
    WHERE OId = 1;
```

In beiden Fällen wird durch die Übernahme der Änderung auf dem Cache der RCC *Order_Lines.IId* → *Items.Id* verletzt.

Die Hüllen der Kontrollwerte 47 und 22 müssen zunächst nachgeladen werden.

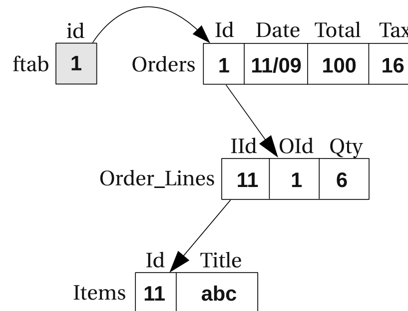

Abbildung 2: Übernahme von Änderungen erfordert das Nachladen von Sätzen

schung und Praxis zielt heute jedoch meist auf eine schwächere Isolationsstufe, die der Snapshot Isolation [BBG+95, LKPMJP05, WK05]. In [Kle10] wurden die beiden Umsetzungsvarianten und die dabei auftretenden Herausforderungen für das CbDBC ausführlich diskutiert. Dabei wurde deutlich, das sich Snapshot Isolation trotz einer verzögerten Aktualisierungsstrategie erreicht lässt, falls die Caches auf den Zustand (Snapshot) im Backend zugreifen können, der ihrem derzeit lokalen entspricht.

In diesem Aufsatz stützen wir uns auf diese Erkenntnisse und verwenden daher für den in Abschnitt 5 entwickelten Synchronisationsmechanismus eine Replikatskontrolle mit verzögerter Aktualisierung der Replikate und einer Primärkopie im Backend.

3.1 Änderungspropagierung

Nachdem wir die Fragen, wo und wann Replikate aktualisiert werden, erörtert haben, betrachten wir in diesem Abschnitt die Besonderheiten bei der Übernahme von Änderungen im Cache und somit die Frage, wie die Replikate (Caches) aktualisiert werden. Wie die Erfassung von Änderungen durch eine geeignete Change-Data-Capture-Strategie erfolgt und durch welche Verfahren die Änderungen selektiv an die Caches gelangen, wird in diesem Aufsatz nicht betrachtet. Erklärungen hierzu finden Sie in [Kle10] und werden zum Verständnis der in diesem Aufsatz entwickelten Nebenläufigkeitskontrolle nicht benötigt.

Eine besondere Bedeutung hat jedoch die Übernahme von Änderungen im Cache. Hierbei sind die dort definierten Constraints jederzeit einzuhalten, wodurch Sätze während der Übernahme nachgeladen werden müssen (vgl. auch [Kle10]). Zur Verdeutlichung dieses Umstandes betrachten wir Abb. 2.

Wir nehmen an, dass die Transaktion T_1 die in Abb. 2 aufgeführten Änderungen im Backend vorgenommen hat. Das ausgeführte Insert-Statement sowie das Update-Statement beziehen sich auf Sätze, die aufgrund des Kontrollwertes $w = 1$ der Tabelle $Orders$ im

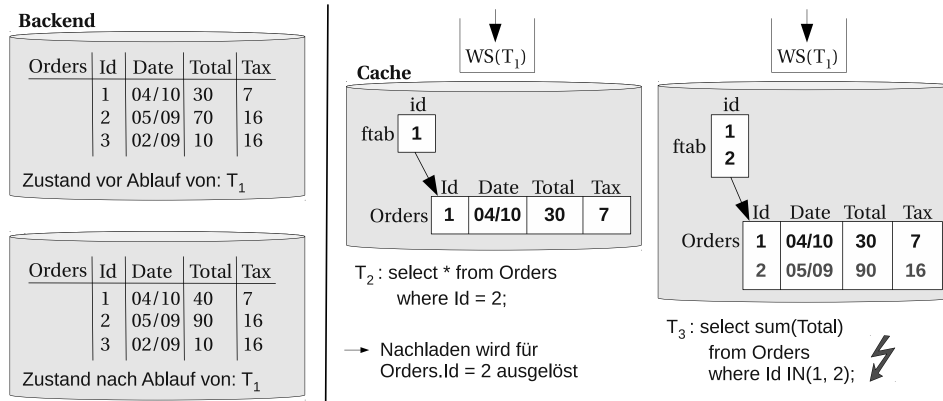

Abbildung 3: Nachladen verursacht ein Einlagern verschiedener transaktionskonsistenter Zustände

Cache benötigt werden. Sie gehören zur Hülle von w. Aus diesem Grund müssen die Sätze $(47, 1, 4)$ sowie $(22, 1, 6)$ (nach Änderung) in den Cache eingelagert werden. Damit der RCC $Order_Lines.IId \rightarrow Items.Id$ nicht durch die Übernahme der Änderungen verletzt wird, müssen zunächst die Hüllen der Kontrollwerte $u = 22$ und $v = 47$ aus $Order_Lines.IId$ nachgeladen werden.

Durch Änderungen können Sätze auch ihre Abhängigkeiten verlieren. Im Beispiel könnte der Satz $(11, abc)$ in $Items$ z. B. entladen werden. Durch Wegfall von Abhängigkeiten werden jedoch niemals RCCs verletzt und deshalb ist hierbei auch kein Nachladen nötig. Ein *Garbage Collector* sucht und löscht nicht mehr benötigte Sätze nebenläufig.

Die Tatsache, dass bei der Übernahme von Änderungen (und auch durch Füllspalten) Ladeprozesse ausgelöst werden, erschwert die Gewährleistung der Konsistenz im Cache nachhaltig. Wir wollen dies durch ein weiteres Beispiel (vgl. Abb. 3) verdeutlichen, wobei wir annehmen, dass die Caches verzögert aktualisiert werden und im Backend nur die neueste Version von Sätzen gelesen werden kann.

Im Beispiel nehmen wir an, der Cache hat die Bestellung 1 ($Orders.Id = 1$) vor dem Ablauf der Transaktion T_1 geladen. Nachfolgend wird im Backend die Änderungstransaktion T_1 ausgeführt, welche die Summe ($Orders.Total$) für die Bestellung 1 und 2 auf die Werte 40 bzw. 90 ändert. Die Änderungen von T_1 wurden bereits in Form eines *Write Set (WS)* an den Cache übertragen, der aber die Änderungen zu diesem Zeitpunkt noch nicht übernommen hat. In der Zwischenzeit wird über den Cache mit der Ausführung der Transaktion T_2 begonnen, welche das Nachladen der Bestellung 2 auslöst, wobei nun im Backend nur noch der Zustand nach Ablauf von T_1 zugreifbar ist. Auf dem Cache befinden sich nun Sätze, die den Zustand vor Ablauf von T_1 repräsentieren, und Sätze, die den Zustand nach Ablauf von T_1 aufweisen, wodurch die Ausführung von T_3 ein *Incorrect Summary* erzeugt.

Die aufgeführten Beispiele machen deutlich, dass im Cache nur Sätze des gleichen transaktionskonsistenten Zustandes eingelagert sein dürfen, damit dieser überhaupt verwendet

werden kann. Dies ist nur dann möglich, wenn der Cache während des Nachladens noch Zugriff auf den Zustand hat, den er selbst gerade repräsentiert.

Basierend auf den bisher gewonnenen Erkenntnissen wollen wir nachfolgend untersuchen, welche Verfahren zur Nebenläufigkeitskontrolle im CbDBC einsetzbar sind. Dabei werden wir zeigen, dass sich die erwünschten Eigenschaften und die notwendige Konsistenz nur durch den Einsatz von Mehrversionenverfahren garantieren lassen, wobei die gewählte Isolationsstufe der Snapshot Isolation lesende Zugriffe nicht behindert.

4 Nebenläufigkeitskontrolle beim CbDBC

Die gewählte Art der Replikatskontrolle muss mit einer geeigneten Nebenläufigkeitskontrolle integriert werden [LKPMJP05, WK05]. In diesem Aufsatz bewerten wir dabei die verschiedenen Verfahren zur Nebenläufigkeitskontrolle nur im Bezug auf die von uns gewünschte verzögerte Änderungsübernahme und unter dem Aspekt, dass eine Primärkopie existiert. Das wichtigste Kriterium für die Auswahl eines geeigneten Verfahrens für das CbDBC ist, dass der durch das Caching erzielte Vorteil (die Performance, die durch eine lokale Anfragebeantwortung gewonnen wurde) nicht verloren gehen darf. Daher sind Verfahren, die einen Zugriff aufs Backend benötigen, z. B. um eine Sperre anzufordern wie beim verteilten 2-Phasen-Sperrprotokoll (D2PL [BG81]), nicht geeignet. Beim Einsatz von optimistischen Verfahren, wie z. B. *Forward Oriented Concurreny Control (FOCC)* oder *Backward Oriented Concurrency Control (BOCC)*, müsste das *Read Set* der Transaktion im Cache erfasst und zur Verifikation an das Backend übermittelt werden.

Als zweites zentrales Problem muss der Cache, um die Wertvollständigkeit seiner Constraints zu erhalten, Sätze nachladen. Dazu ist es notwendig, auf einen bestimmten transaktionskonsistenten Zustand im Backend zugreifen zu können, wie bereits im vorangehenden Abschnitt gezeigt. Um dies effizient gewährleisten zu können, muss auf Cache und Backend eine Mehrversionenverfahren zum Einsatz kommen.

Der Einsatz von Mehrversionenverfahren ist auch dadurch motiviert, dass für eine Transaktion zu jeder Zeit gewährleistet sein muss, dass sie den gleichen transaktionskonsistenten Zustand liest, egal ob sie auf den Cache oder das Backend zugreift. Da sich der Cache-Inhalt fortlaufend ändert, kann nicht gewährleistet werden, dass das Lesen einer Transaktion nur auf den Cache beschränkt ist. Durch die verzögerte Aktualisierungsstrategie unterscheiden sich die neuesten transaktionskonsistenten Zustände der Datenbanken voneinander. Somit müssen ältere transaktionskonsistente Zustände im Backend aufgehoben werden, damit sie für den Cache zugreifbar sind, wodurch zwingend der Einsatz eines Mehrversionenverfahrens notwendig wird.

Zur effizienten Implementierung von Snapshot Isolation [Fek09] werden auch Mehrversionenverfahren eingesetzt. Als wesentliche Eigenschaft blockiert Snapshot Isolation niemals lesende Zugriffe. Gelesene Objekte müssen somit auch nicht aufgezeichnet werden, was für CbDBC von großem Vorteil ist. So wird die Implementierung Middleware-basierter Lösungen sehr erleichtert, da es oft keine Unterstützung gibt, alle gelesenen Objekte einer Transaktion abzufragen bzw. in einem Read Set zu speichern.

Im Nachfolgenden betrachten wir Snapshot Isolation und später deren Umsetzung im CbDBC genauer. Dabei gehen wir stets davon aus, dass auf eine Objektmenge oder ein Objekt O abstrakt zugegriffen wird. Für die Nebenläufigkeitskontrolle spielt es nämlich keine Rolle, ob O einzelne Attributwerte, Sätze oder Seiten darstellt. Wichtig ist nur, das über die Art der Versionskontrolle immer der gleiche transaktionskonsistente Zustand zugegriffen wird.

4.1 Snapshot Isolation

Die Isolationsstufe Snapshot Isolation wurde maßgeblich in [BBG+95] motiviert. Snapshot Isolation gründet direkt auf dem Einsatz von Mehrversionenverfahren, die einer Transaktion stets den Zustand (Snapshot), der zu ihrem Startzeitpunkt (*Begin of Transaction, BOT*) gerade gültig war, präsentieren. Dabei muss dieser Zeitpunkt irgendwann vor dem ersten Lesezugriff durch die Transaktion festgelegt werden. Lesende Zugriffe werden niemals blockiert. Sie sind also immer möglich, was für das Datenbank-Caching von besonderer Bedeutung ist. Will eine Transaktion T_x ein Objekt O schreiben, so darf sie dies im einfachsten Fall nur dann, wenn nach der zu lesenden Version V_i keine weitere festgeschriebene Version $V_{j>i}$ existiert. Ist dies der Fall, wird T_x zurückgesetzt. Dieses Vorgehen bezeichnet man auch als *First Committer Wins* [BBG+95]. Es werden also nur sogenannte Schreib/Schreib-Konflikte erkannt und aufgelöst. Schreibende Transaktionen müssen also nicht durch ein spezielles Protokoll (im einfachsten Fall ein RX-Sperrprotokoll) synchronisiert werden.

Um den BOT einer Transaktion bzw. den Commit-Zeitpunkt einer Version festlegen zu können, benötigt man eine logische Uhr, die zumindest nach dem erfolgreichen Commit einer schreibenden Transaktion erhöht werden muss. Um auch den BOT einer Transaktion eindeutig zuordnen zu können, wird diese Uhr, welche wir nachfolgend als *Systemuhr* t bezeichnen, meist auch bei BOT um eins erhöht.

In unserer Implementierung gehen wir entsprechend vor: Bei Beginn einer Transaktion wird der Wert von t als BOT festgehalten; danach wird t um eins erhöht. Gleiches gilt beim erfolgreichen Abschluss einer Transaktion (z. B. von T_x), wobei alle von ihr geschriebenen Versionen mit dem Zeitpunkt $EOT(T_x) = t$ markiert werden.

Bei der Umsetzung von Snapshot Isolation für verteilte Datenbanksysteme unterscheidet man üblicherweise zwischen *starker (strong)* und *schwacher (weak)* Snapshot Isolation [DS06]. Der Unterschied besteht darin, dass es im verteilten Fall dazu kommen kann, dass eine Transaktion ihre eigenen Änderungen nicht sofort sieht, je nachdem wann Replikate aktualisiert werden. Bei starker Snapshot Isolation kann dies nicht vorkommen, bei schwacher Snapshot Isolation wird dies toleriert.

Die nachfolgend beschriebene Vorgehensweise beim CbDBC garantiert zunächst nur schwache Snapshot Isolation. In Abschnitt 5.4 erklären wir aber auch, wie sich starke Snapshot Isolation garantieren lässt.

5 Snapshot Isolation für CbDBC

Wir betrachten nun, wie sich Snapshot Isolation beim CbDBC erreichen lässt. Da das Backend und die Caches auf unterschiedlichen Hosts agieren, unterscheiden wir die *Backend-seitige Systemuhr* t_{BE} von der *Cache-seitigen Systemuhr* t_{C_i}. Wir benötigen diese Unterscheidung auch, um deutlich zu machen, dass alle Instanzen ihre eigene (lokal verwaltete) Versionskontrolle besitzen. Eine global eindeutige Systemuhr wird somit nicht benötigt.

In den nachfolgenden Betrachtungen beziehen sich alle Erklärungen direkt auf die Versionen, die bei Änderung eines Objektes angelegt werden. Dabei spielt es jedoch keine Rolle, ob die vorgeschlagene Versionskontrolle direkt in ein eigenständiges (evtl. proprietäres) CbDBC-System integriert wird oder ob das Caching-System Middleware-basiert umgesetzt ist, wobei jedoch nur Datenbanksysteme benutzt werden können, die lokal auch Snapshot Isolation anbieten, wie z. B. PostgreSQL [Pos10], Oracle [Ora10] oder MS-SQL-Server [Mic10] (vgl. hierzu auch Abschnitt 5.7).

Im Folgenden bezeichnen wir eine durch den Benutzer (bzw. zugreifende Applikation) initialisierte Transaktion T_x als *global* bzw. als *Benutzertransaktion*. Jede globale Transaktion wird immer genau von einem Cache C_i zusammen mit dem Backend ausgeführt, wobei das *Cache-Management-System (CacheMS)* die Transaktion T_x kontrolliert. Für den Teil der Transaktion T_x, der über das Backend ausgeführt wird, schreiben wir T_x^{be} und für den Cache-Zugriff $T_x^{c_i}$. Wir bezeichnen diese Teile von T_x als *lokale* Transaktionen, da sie nur einen lokalen Zugriff, entweder auf die Cache-Datenbank oder auf die Backend-Datenbank zulassen. Bei einer Middleware-basierten Umsetzung sind für die Zugriffswege T_x^{be} und $T_x^{c_i}$ eigenständige, sogenannte *echte* lokale Teiltransaktionen nötig, die vom CacheMS initialisiert werden. Darüber hinaus gibt es weitere lokale Transaktionen, die durch das CacheMS initialisiert werden, um bestimmte Verwaltungsaufgaben (z. B. das Nachladen) von Werten durchzuführen. Diese Transaktionen werden einfach mit ihrem Zweck markiert. Wir schreiben T_y^{load} für eine lokale Ladetransaktion, die auf das Backend zugreift, um die zu landenden Sätze zu lesen, und T_y^{insert} für deren Einfügeoperationen im Cache. Lokale Transaktionen, die die Übernahme von Änderungen aus dem WS realisieren, markieren wir mit *accept*, so ergibt sich z. B. T_z^{accept}. Lokale Transaktionen, die auf das Backend zugreifen, nennen wir auch einfach *Backend-Transaktionen* und dementsprechend lokale Transaktionen, die auf den Cache verweisen *Cache-Transaktionen*. Auch wenn die Ausführung lokaler Transaktionen nicht Middleware-basiert, sondern integriert (intern) abläuft, gehen wir davon aus, dass für diese Transaktionen die ACID-Eigenschaften unter Isolationsstufe Snapshot Isolation eingehalten werden. Den zugewiesenen Zugriffszeitpunkt von Backend-Transaktionen benennen wir t_Z^{be} und von Cache-Transaktionen $t_Z^{c_i}$.

5.1 Referenzierung eines Snapshots

Wie bereits erwähnt, muss der Cache ältere Versionen (Snapshots) im Backend referenzieren und zugreifen können. Dies wird durch die Übermittlung des gewünschten *Zugriffs-*

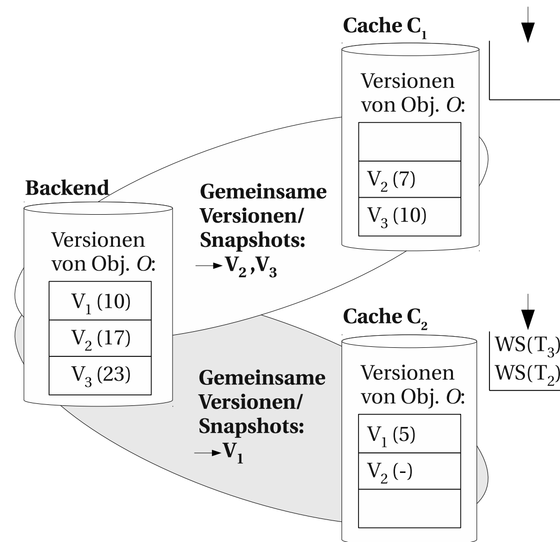

Abbildung 4: Gemeinsame Versionen von Backend und Cache bei verzögerter Aktualisierung

zeitpunktes t_Z^{be} ans Backend ermöglicht. Es genügt beim ersten Lesen einer Transaktion im Backend, diesen Zeitpunkt mitzuteilen. Die Frage ist jedoch, woher der Cache diesen Zeitpunkt kennt bzw. wie er ihn herausfindet. Das Backend überträgt den *Zustandszeitpunkt* t_Z beim Abschluss einer globalen Transaktion T zusammen mit den Änderungen von T im WS. Dabei ist $EOT(T) = t_{BE} = t_Z$. Sobald der Cache alle Änderungen aus diesem WS übernommen hat, stellt er zusammen mit dem Backend den neuen Zustand (Snapshot), der nach T erreicht wurde, bereit und kann daher fortan unter Angabe des im WS übermittelten Zustandszeitpunktes t_Z auf das Backend zugreifen. Die Initialisierung stellt dabei einen Sonderfall dar, weil hierbei zunächst ein Startzeitpunkt vom Backend zugewiesen wird (vgl. Abschnitt 5.2). Wir betrachten hierzu Abb. 4, welche einen Überblick über die angestrebte verteilte Versionsverwaltung bietet.

In Abb. 4 sind im Backend die Versionen V_1, V_2 und V_3 des Objektes O zu den Zeitpunkten 10, 17 und 23, welche in Klammern angegeben sind, festgeschrieben worden. Wir nehmen an, dass diese Versionen durch die Transaktionen T_1, T_2 und T_3 erzeugt wurden, die zu den gleichen Zeitpunkten (z. B. $EOT(T_1) = 10$) erfolgreich abgeschlossen wurden. Da auf einem Cache niemals Änderungen durch einen Benutzer stattfinden, werden dort nur Versionen von O erzeugt, wenn O neu geladen wird oder wenn ein WS nachfolgend eine Änderung für O signalisiert.

In unserem Beispiel lädt der Cache C_1 die Version V_2 und schreibt diese zum Zeitpunkt 7 seiner lokalen Systemuhr (t_{C_1}) fest. Das Überspringen von älteren Versionen entsteht, wenn der Cache erst zu einem Zeitpunkt $t_{BE} > 17$ das Objekt O lädt. Dies wäre z. B. der Fall, wenn die WSs von T_1 und T_2 übernommen werden, bevor der Cache O lädt. Im Beispiel hat C_1 auch bereits das WS von T_3 eingespielt und die Änderung auf O zum Zeitpunkt $t_{C_1} = 10$ festgeschrieben.

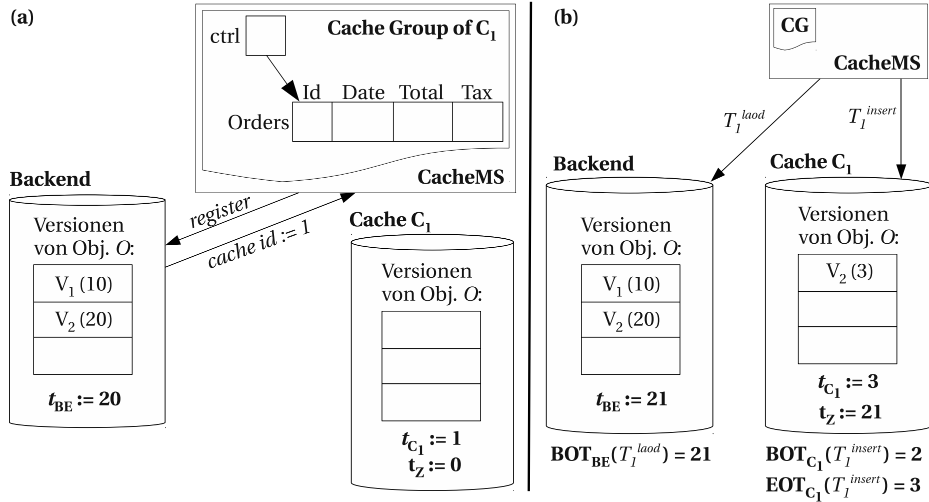

Abbildung 5: Initialisierung des Caches

Nehmen wir an, eine Transaktion T_{neu} startet, bevor das WS von T_3 eingespielt wird. In diesem Fall wird der Snapshot von T_{neu} korrekt referenziert, wenn zum Cache im Zeitintervall $t_{C_1} \geq 7 \wedge t_{C_1} < 10$ zugegriffen wird und zum Backend zum Zeitpunkt t_Z von 17. Die Wahl eines späteren Zeitpunktes als $t_Z^{be} = 17$ würde im Backend möglicherweise einen anderen Snapshot adressieren, wenn nämlich eine Transaktion zum Zeitpunkt 18 Versionen anderer Objekte eingebracht hätte. Im Cache repräsentieren alle Zeitpunkte t_{c_i} zwischen der Übernahme zweier WSs den gleichen transaktionskonsistenten Backend-Zustand. Daher sind hier mehrere Zugriffszeitpunkte $t_Z^{c_i}$ wählbar.

Der Cache C_2 hat bereits die Version V_1 von O geladen und das WS von T_1 übernommen. Die Änderungen der WSs von T_2 und T_3 stehen noch aus. Somit kann C_2 einer Transaktion T_{neu} zurzeit nur einen gemeinsamen Snapshot basierend auf der gemeinsamen Version V_1 anbieten.

5.2 Initialisierung

Ein neu zu initialisierender Cache ist zunächst leer. Damit er seine Arbeit aufnehmen kann, muss er sich zunächst am Backend registrieren (vgl. Abb. 5a). Dabei erhält er eine eindeutige Identifikation, die *Cache-Id*, um sich während der weiteren Verarbeitung gegenüber dem Backend ausweisen zu können. Der Cache ist bereits vor dieser Registrierung in der Lage, Benutzertransaktionen entgegenzunehmen. Diese werden solange komplett an das Backend weitergeleitet, bis die Initialisierung vollständig abgeschlossen ist.

Nachdem die Registrierung abgeschlossen ist, greift der Cache auf das Backend durch Initialisierung einer Ladetransaktion T_1^{load} erstmalig zu. Da der Cache noch leer ist, wird

für T_1^{load} der Wert $t_Z^{be} = t_Z = 0$ als Zugriffsparameter angegeben. Dies signalisiert dem Backend, dass T_1^{load} Zugriff auf den letzten festgeschriebenen Datenbankzustand (den neuesten Snapshot) erhalten soll. Diese Situation ist in Abb. 5b dargestellt, wobei der tatsächliche Zugriffszeitpunkt für T_1^{load} auf $BOT_{BE}(T_1^{load}) = t_{BE} = 21$ korrigiert wird. Dieser erstmals festgelegte Zeitpunkt wird als *Ausgangszeitpunkt* des Caches im Backend hinterlegt. Der Cache ist nun initialisiert und kann seine normale Verarbeitung aufnehmen. Muss der Cache in dieser Phase ($t_Z = 0$) noch weitere lokale Transaktionen anlegen, die auf das Backend zugreifen (z. B. falls eine Benutzertransaktion neu startet), so werden diese auf den Ausgangszeitpunkt (hier 21) festgelegt. Auf diese Weise muss dem Cache niemals durch eine gesonderte Kommunikation der Ausgangszeitpunkt explizit mitgeteilt werden.

Start der Auslieferung von WSs. Sobald der Cache erstmalig eine lokale Transaktion zum Zugriff auf das Backend gestartet hat (T_1^{load}), werden alle WSs von Änderungstransaktionen mit einem $EOT_{BE} \geq 21$ an den Cache ausgeliefert. Ändern sich also Objekte, die mittels T_1^{load} geladen wurden, kann der Cache alle Folgeänderungen anhand der ihm zugestellten WSs nachvollziehen. Da in jedem WS der nachfolgend zu verwendende Zustandszeitpunkt t_Z mitgeschickt wird, ist dieser nach der ersten WS-Übernahme im Cache eindeutig definiert.

Transaktionskonsistenter Zustand. Wir betrachten nochmals Abb. 5b. Solange C_1 kein WS abschließend eingespielt hat (also auch während der Übernahme eines WS) werden alle geladenen Sätze über T_1^{load} abgefragt. Somit erreichen nur Objekte des gleichen Snapshots (d. h. des gleichen transaktionskonsistenten Zustands) den Cache. Somit spielt es keine Rolle, wann und wie oft Sätze in die Cache-Datenbank eingefügt werden. In unserem Beispiel aus Abb. 5b wurde das Objekt O zum Zeitpunkt $EOT_{C_1}(T_1^{insert}) = 3$ in die Cache-Datenbank eingebracht. Eine lokale Transaktion $T_{alt}^{c_1}$ mit $BOT_{C_1}(T_{alt}^{c_1}) = 1$ kann auf das Objekt O im Cache nicht zugreifen, eine Transaktion $T_{neu}^{c_1}$ mit $BOT_{C_1}(T_{neu}^{c_1}) = 4$ hingegen schon. Die durch das CbDBC vorgegebenen Regeln, die eine korrekte Cache-Verwaltung (bzgl. Laden, Entladen, Probing) garantieren, bleiben dabei unangetastet und müssen natürlich eingehalten werden [KB09, HB07].

5.3 Übergang zum Nachfolgezustand

Sobald ein WS im Cache eintrifft, muss dieses schnellstmöglich verarbeitet werden. Dabei muss sichergestellt werden, dass die WSs in der Commit-Reihenfolge ihrer zugehörigen Änderungstransaktionen abgearbeitet werden. Jede WS-Übernahme wird durch genau eine lokale Transaktion im Cache abgewickelt (z. B. von T_1^{accept}, wie in Abb. 6a gezeigt).

Für jede Änderung im WS ist zu prüfen, ob der Cache den als geändert aufgeführten Satz[3] überhaupt enthält. Ist der Satz im Cache, wird er durch T_1^{accept} angepasst. Wie in

[3] Im WS werden alle Attribute eines geänderten Satzes für jede Cache-Tabelle in ihrem alten und neuen Zustand übertragen (vgl. [Kle10]).

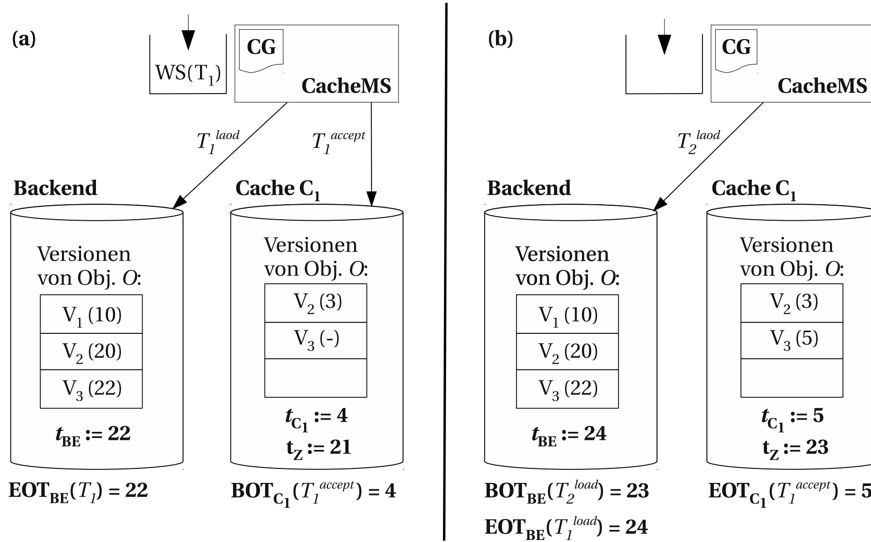

Abbildung 6: Übergang zum nächsten transaktionskonsistenten Zustand.

Abschnitt 3 gezeigt, können hierbei Ladeoperationen ausgelöst werden. Dabei ist es sogar möglich, dass neue Sätze den Cache erreichen, für die im gerade abzuarbeitenden WS eine Änderung aufgeführt ist, welche dann noch zusätzlich (nach dem entsprechenden Ladevorgang) durchzuführen ist. Diese könnte wieder ein Nachladen auslösen usw. Es entsteht eine rekursive Abarbeitung des WS. Dabei wird jedoch jede im WS aufgeführte Änderung höchstens ein Mal angewandt, wodurch der Übernahmeprozess stets terminiert. Erst wenn alle anwendbaren Änderungen des einzuspielenden WS übernommen wurden und die hierdurch ausgelösten Nachladeoperationen beendet sind, wird T_1^{accept} abgeschlossen. Hierdurch wird der Cache atomar von einem transaktionskonsistenten Zustand in den nachfolgenden überführt.

Nach dem erfolgreichen Abschluss von T_1^{accept} müssen alle nachfolgenden Ladeprozesse mit einer lokalen Backend-Transaktion durchgeführt werden, die den neuen transaktionskonsistenten Zustand repräsentiert. In Abb. 6b wurde dazu die Transaktion T_2^{load} neu eingerichtet, welche mit dem Zugriffszeitpunkt $t_Z^{be} = t_Z = 23$ aus dem WS von T_1 initialisiert wurde. Ebenso muss jede neu zu initialisierende lokale Backend-Transaktion T_{neu}^{be} mit $t_Z^{be} = 23$ angelegt werden. Die genaue Initialisierung der zugehörigen Benutzertransaktion T_{neu} wird im anschließenden Abschnitt besprochen.

5.4 Snapshot-isolierte Benutzertransaktion

Durch die in den beiden vorangegangenen Abschnitten beschriebene Versionsverwaltung sind wir nun in der Lage einer Benutzertransaktion (z. B. T_1) einen eindeutigen, globalen

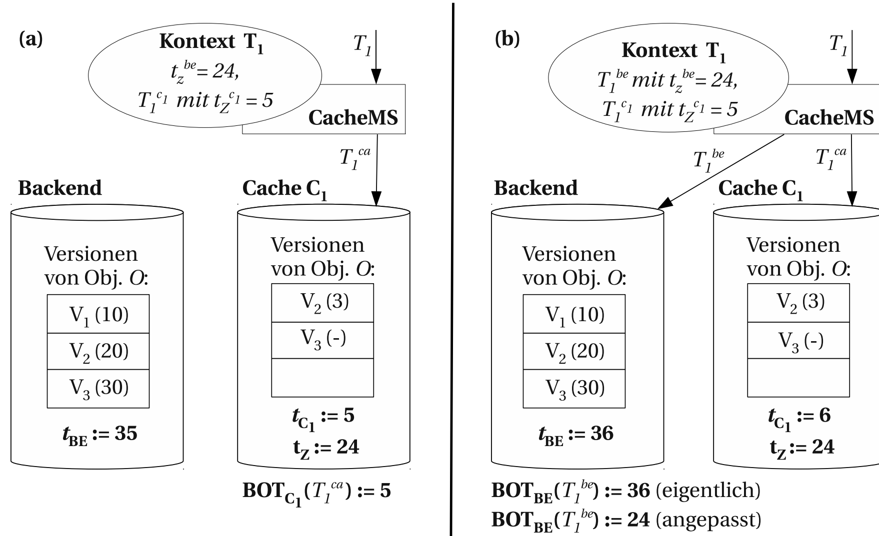

Abbildung 7: Initialisierung einer Benutzertransaktion.

Snapshot zuzuweisen. Dieser ergibt sich durch die Festlegung der Zugriffszeitpunkte für die lokalen Teiltransaktionen T_1^{be} und $T_1^{c_i}$. Sie werden im *Transaktionskontext* (oder kurz: Kontext) von T_1 hinterlegt, der vom CacheMS verwaltet wird. Auf diese Weise wird durch die Informationen im Kontext der Snapshot der Transaktion definiert. Aus diesem Grund bezeichnen wir solche Transaktionskontexte fortan synonym als Snapshot der Transaktion.

Jeder Snapshot wird gebildet, indem zunächst $T_1^{c_i}$ für die Cache-Datenbank angelegt wird. Hierbei wird kein Zugriffszeitpunkt vorgegeben. Der im Snapshot hinterlegte Zeitpunkt für den Cache-Zugriff ergibt sich direkt aus dem Startzeitpunkt von $T_1^{c_i}$. Somit ist $t_Z^{c_i} = BOT_{c_i}(T_1^{c_i}) = t_{c_i}$ der im Kontext hinterlegte Cache-Zugriffszeitpunkt. Gleichzeitig wird im Kontext von T_1 der aktuelle Wert von t_Z für den lokalen Backend-Zugriff hinterlegt ($t_Z^{be} = t_Z$). Die Transaktion T_1^{be} wird erst initialisiert, wenn ein Zugriff auf das Backend tatsächlich nötig wird. So wird für Transaktionen, die nur Lesezugriffe ausführen, die durch den Cache beantwortbar sind, das Backend nicht involviert. Wir betrachten das Anlegen einer Benutzertransaktion nochmals genau mit Hilfe der Beispiele aus Abb. 7.

In Abb. 7a greift die Benutzertransaktion T_1 erstmalig über das CacheMS von C_1 auf das CbDBC-System zu. Die lokale Systemuhr des Caches hat derzeit den Wert $t_{c_1} = 5$ und im letzten, bereits eingespielten WS wurde der Zustandszeitpunkt $t_Z = 24$ übermittelt. Der Cache initialisiert den Snapshot von T_1, indem er die Zeitpunkte $t_Z^{c_1} = BOT_{c_1}(T_1^{c_1}) = t_{c_1} = 5$ und $t_Z^{be} = t_Z = 24$ für T_1 festlegt. Danach wird die lokale Systemuhr t_{c_1} um eins erhöht ($t_{c_1} = 6$, vgl. Abb. 7b). Die lokale Backend-Transaktion T_1^{be} ist noch nicht initialisiert. T_1 kann zurzeit also nur die Version V_2 von O im Cache lesen. Im Bild ist auch angedeutet, dass der Cache gerade die Version V_3 übernimmt (z. B. durch T_1^{accept}), die im Backend bereits festgeschrieben wurde. T_1^{accept} ist jedoch noch nicht abgeschlossen und daher zu T_1 nebenläufig. Sie endet erst zu einem Zeitpunkt $t_{c_1} > 5$, sodass T_1 die Version

V_3 niemals lesen kann.

Der Cache hat den Snapshot gebildet und ist nun in der Lage, falls T_1 auf das Backend zugreifen muss, T_1^{be} jederzeit korrekt anzulegen (vgl. Abb. 7b). Der Zugriffszeitpunkt von T_1^{be} würde normalerweise (unangepasste Snapshot Isolation) auf den tatsächlichen Beginn der Transaktion $BOT_{BE}(T_1^{be}) = t_{BE} = 36$ festgelegt. Da der Cache für T_1^{be} aber den Zugriffszeitpunkt 24 hinterlegt hat, wird der Beginn auf $BOT_{BE}(T_1^{be})$ auf den Wert 24 vorverlegt. Egal ob T_1 nun auf den Cache oder das Backend zugreifen muss, er sieht immer die gleiche Version des Objektes O (hier V_2). Da bereits eine weitere Version (V_3) von O durch eine andere Transaktion festgeschrieben wurde, kann T_1 auf O keine Änderung vornehmen. Dies würde zu einem Schreib/Schreib-Konflikt führen, der im Backend wie üblich (nach den Regeln für SI) erkannt wird. T_1 müsste dann zurückgesetzt werden (First Committer Wins).

Wir konnten somit zeigen, wie im Cache ein transaktionskonsistenter Zustand hergestellt und gewahrt werden kann. Durch die Möglichkeit, ältere Versionen im Backend zu referenzieren, kann jeder Benutzertransaktionen ein einheitlich aufgebauter, global konsistenter Snapshot zur Verfügung gestellt werden. Da Änderungen nur im Backend erfolgen, lassen sich dort alle Schreib/Schreib-Konflikte korrekt erkennen. Insbesondere der erfolgreiche Transaktionsabschluss kann allein durch das Backend verifiziert werden. Ein 2PC-Protokoll ist nicht notwendig, da auf den Caches keine Dauerhaftigkeit von Objekten benötigt wird. Tritt während der Verarbeitung ein schwerwiegender Fehler auf (wenn z. B. der Cache ausfällt oder ein übermitteltes WS nicht lesbar ist), kann der Cache im einfachsten Fall geleert und neu initialisiert werden.

Der beschriebene Ansatz garantiert jedoch nur schwache Snapshot Isolation, da es vorkommen kann, dass eine Transaktion ein Objekt ändert (Backend), welches sie nachfolgend nochmal liest (Cache). Wir diskutieren im nachfolgenden Abschnitt kurz einige Möglichkeiten, starke Snapshot Isolation zu erreichen, falls schwache Snapshot Isolation nicht ausreicht.

5.5 Starke Snapshot Isolation

Um starke Snapshot Isolation zu erreichen, muss sichergestellt sein, dass eine Transaktion T_1 keine Sätze im Cache liest, die sie selbst bereits geändert hat. Am einfachsten gelingt dies, wenn einer Transaktion der Zugriff auf den Cache verwehrt bleibt, sobald sie einmal ein Update-Statement ausführt hat. Dadurch wird jedoch die Nutzung des Caches sehr stark eingeschränkt. Reine Lesetransaktionen und Transaktionen, die erst am Ende schreiben, würden jedoch kaum oder gar nicht beeinflusst.

Um diese radikale Lösung zu vermeiden, kann man selektiv nur Zugriffe auf Cache-Tabellen verbieten, deren zugeordnete Backend-Tabelle zuvor von einem Update-Statement geändert wurde. Da jedes Statement einer Transaktion zunächst am Cache ankommt kann dieser die betroffene Tabelle auslesen und das Zugriffsverbot für die Transaktion in deren Kontext vermerken.

Als weitere Verfeinerung können alle Primärschlüssel (oder die *Record Identifier*) geän-

derter Sätze an die Antwortnachricht einer Update-Anweisung angehängt werden (per *Piggybacking*). Dies macht nur dann Sinn, wenn nicht zu viele Informationen zu übertragen sind. Der Cache kann so prüfen, ob zu lesende Sätze bereits durch die Transaktion geändert wurden. Wenn ja, wird die Leseoperation vom Backend beantwortet. Wurden zu viele Sätze geändert, so dass der Verwaltungsaufwand zu hoch ist, werden nachfolgend ungeprüft alle Operationen der Transaktion direkt ans Backend geschickt. Der große Nachteil dieser Vorgehensweise besteht darin, dass die Änderungen einer Anweisung direkt im Backend ermittelt werden müssen (z. B. durch Trigger). Das im Aufsatz gezeigte Verfahren vermeidet dies gerade, da hierbei die Änderungen erst nach Transaktionsabschluss (z. B. durch Log-Sniffing-Techniken) ermittelt werden können.

Eine viel versprechende Methode ist die Durchführung einer Vollständigkeitsprüfung (Probing) für Update-Anweisungen. Ist das Probing erfolgreich, kann der Cache selbst die geänderten Sätze ermitteln und daraus auch die neuen Werte berechnen. So kann der Cache selbst einer Transaktion ihre geänderten Sätze zurückliefern, falls die Ausführung im Backend erfolgreich war. Hierbei ist es jedoch zu beachten, dass dem Backend das erfolgreiche Probing signalisiert wird. Das Backend muss im Gegenzug die Zulässigkeit des Update bestätigen. Werden beim Update Folgeänderungen ausgelöst, die wiederum den momentanen Cache-Inhalt beeinflussen, muss dies zurückgemeldet werden, da der Cache solche Situationen nicht erkennen kann.

Es lassen sich auch Kombinationen der vorgeschlagenen Verfahren implementieren. Sobald der Cache jedenfalls transaktionslokal geänderte Sätze – in der Regel bei wiederholtem Lesen – nicht selbst zur Verfügung stellen kann, muss die Leseoperation im Backend erfolgen.

Alle hier aufgeführten Methoden können starke Snapshot Isolation erreichen. In unserem CbDBC-Prototyp ACCache [BHM06] ist jedoch bisher nur schwache Snapshot Isolation umgesetzt.

5.6 Löschung alter Versionen

Alte Versionen von Objekten dürfen gelöscht werden, wenn keine Transaktion mehr auf sie zugreifen kann. Sobald die Caches ein WS übernehmen, erhöht sich ihr lokaler Zustandszeitpunkt t_Z. Legt der Cache eine Backend-Transaktion an, wird der Zugriffszeitpunkt t_Z^{be} ans Backend zurück übermittelt. t_Z und somit auch t_Z^{be} werden stets nur erhöht. Eine vom Cache angelegte Benutzertransaktion greift also niemals mit einem Zugriffszeitpunkt t_Z^{be}, der kleiner als ein zuvor übermittelter Wert ist, auf das Backend zu. Aus dem Minimum aller übermittelten Zugriffszeitpunkte lässt sich so ermitteln, welche Versionen noch zugreifbar sind. Nicht mehr zugreifbare Versionen können gelöscht werden. Werden Benutzertransaktionen nur im Backend ausgeführt, ohne das ein Cache involviert ist, muss der kleinste BOT aller laufenden Transaktionen in die Analyse aufgenommen werden.

Auf dem Cache gelten die Standard-Regeln für die Löschung von Versionen. Das heißt, dass dort nur das Minimum über alle BOTs von laufenden Transaktionen herangezogen wird, um zu entscheiden, welche Versionen zu löschen sind.

5.7 Middleware-basierter Zugriff auf ältere Snapshots

In der derzeitigen Implementierung unseres Prototyps (ACCache) wird der Zugriff auf ältere Snapshots durch einen speziellen Transaktionspool simuliert. Dieser basiert auf der Idee, dass nach dem Abschluss einer Transaktion, die den Datenbankzustand verändert hat, ein Pool von nebenläufigen Transaktionen[4] angelegt wird, deren Startzeitpunkte direkt aufeinander folgen (z. B. $BOT_{BE}(T_1^{be}) = 10$, $BOT_{BE}(T_2^{be}) = 11$ usw.). Dabei wird sichergestellt, dass, während ein solcher Pool angelegt wird, keine andere Transaktion ihr Commit ausführt. Die Transaktionen innerhalb eines Pools greifen also alle auf den gleichen logischen Snapshot zu. Jeder Pool erhält eine Nummer, die gleichzeitig als Zugriffszeitpunkt dient und im WS an den Cache übermittelt wird. Wenn ein Cache eine lokale Backend-Transaktion anlegen will, referenziert er den entsprechenden Pool über den mitgelieferten Zustandszeitpunkt. Aus diesem bekommt er dann eine Transaktion zugewiesen. Ist der Pool von Transaktionen erschöpft, ist der Snapshot für den Cache nicht mehr erreichbar. Der Cache wird gezwungen, weitere WSs zu übernehmen, um einen neueren Pool referenzieren zu können. Wenn die Initialisierung einer lokalen Backend-Transaktion T_1^{be} fehlschlägt (z. B. wenn der Pool leer ist), wird die Benutzertransaktion T_1 einfach abgebrochen.

6 Fazit

Der vorliegende Aufsatz zeigt, dass der Einsatz eines (verteilten) Mehrversionenverfahrens für das CbDBC zwingend erforderlich ist, um einer Benutzertransaktion stets einen transaktionskonsistenten Zustand anbieten zu können. Damit lesende Zugriffe bevorzugt werden, wurde die Isolationsstufe Snapshot Isolation angestrebt. Dabei ist die im Aufsatz entwickelte Art der Nebenläufigkeitskontrolle bestens auf die Eigenschaften des CbDBC abgestimmt. Lesende Zugriff werden niemals blockiert, ein Erstellen von Read Sets ist nicht erforderlich, die Replikate in den Caches lassen sich verzögert aktualisieren und es gibt keinerlei zusätzliche Kommunikation zwischen Cache und Backend. Besonders diese letztgenannte Eigenschaft ist wichtig. Der Cache ist sogar in speziellen Fällen in der Lage, eine Benutzertransaktion alleine, ohne Involvierung des Backends durchzuführen. Der erfolgreiche Abschluss einer Transaktion kann allein vom Backend verifiziert werden, sodass ein 2PC-Protokoll nicht notwendig ist. Durch Sondermaßnahmen lässt sich starke Snapshot Isolation erreichen, wobei der Cache oft selbst verifizieren kann, ob die zu ändernden Sätze vollständig im Cache vorliegen.

Das Poolen von Transaktionen ermöglicht, falls erforderlich, den Middleware-basierten Zugriff auf ältere Snapshots, wobei eine Integration dieser Funktionalität in ein bestehendes Datenbanksystem (z. B. PostgreSQL) anzustreben ist. Einerseits sind dadurch die älteren Zustände durch beliebig viele Benutzertransaktionen zugreifbar und andererseits ist ein deutlich besseres Leistungsverhalten zu erwarten.

[4] Durch diesen Trick können wir später bei Bedarf noch mehrfach einen existierenden Snapshot nutzen.

7 Kritik und Ausblick

Der einzige echte Nachteil der vorgestellten Nebenläufigkeitskontrolle liegt darin, dass der Backend-Zugriff von Benutzertransaktion auf den Zustand zurückgesetzt wird, der im Cache als letztes übernommen wurde. Der Cache bestimmt somit durch die Geschwindigkeit, mit der er WSs übernimmt, wie stark sein Zustand von aktuellen Zustand im Backend abweicht. Dieser Abstand sollte nur gering sein, da sonst die Gefahr von Schreibkonflikten enorm steigt, was zum Abbruch vieler Transaktionen führen würde. Zur Verhinderung einer solchen Situation kann der Cache die Ausführung von Benutzertransaktionen künstlich verlangsamen, um mit höherer Priorität WSs einzuspielen. In der Regel ist jedoch zu erwarten, dass bei hoher Leselast im Gesamtsystem (60-80%) Konsistenz und Vollständigkeit in den Caches nur wenig vom aktuellen Datenbankzustand abweichen.

Die Eigenschaften der vorgestellten Synchronisation (Konfliktrate, Leistung, Skalierbarkeit) müssen durch Messungen noch genau untersucht werden. Dies stellt den Hauptanteil weiterer Forschungsbemühungen dar. Wünschenswert wäre dabei eine Integration in ein bestehendes Datenbanksystem (z. B. PostgreSQL). Außerdem müssen die Ideen und Methoden, um starke Snapshot Isolation zu garantieren, noch umgesetzt werden.

Literatur

[ABK+03] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh und Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *VLDB*, Seiten 718–729, 2003.

[APTP03] Khalil Amiri, Sanghyun Park, Renu Tewari und Sriram Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *ICDE*, Seiten 821–831, 2003.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil und Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, Seiten 1–10, 1995.

[BDD+98] Randall G. Bello, Karl Dias, Alan Downing, James J. Feenan, Jr., James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski und Mohamed Ziauddin. Materialized Views in Oracle. In *VLDB*, Seiten 659–664, 1998.

[BG81] Philip A. Bernstein und Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[BHM06] Andreas Bühmann, Theo Härder und Christian Merker. A Middleware-Based Approach to Database Caching. In Y. Manolopoulos, J. Pokorný und T. Sellis, Hrsg., *ADBIS 2006*, LNCS 4152, Seiten 182–199, Springer, 2006.

[DS06] Khuzaima Daudjee und Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *VLDB 2006: Proceedings of the 32nd International Conference on Very Large Data Bases*, Seiten 715–726, 2006.

[Fek09] Alan Fekete. Snapshot Isolation. In *Ency. of Database Systems*, Seiten 2659–2664. 2009.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil und Dennis Shasha. The Dangers of Replication and a Solution. In *SIGMOD Conference*, Seiten 173–182, 1996.

[GLRG04a] Hongfei Guo, Per-Åke Larson, Raghu Ramakrishnan und Jonathan Goldstein. Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. In Gerhard Weikum, Arnd Christian König und Stefan Deßloch, Hrsg., *SIGMOD*, Seiten 815–826. ACM, 2004.

[GLRG04b] Hongfei Guo, Per-Ake Larson, Raghu Ramakrishnan und Jonathan Goldstein. Support for Relaxed Currency and Consistency Constraints in MTCache. In *SIGMOD*, Seiten 937–938, New York, NY, USA, 2004. ACM.

[HB07] Theo Härder und Andreas Bühmann. Value Complete, Column Complete, Predicate Complete – Magic Words Driving the Design of Cache Groups. *VLDB Journal*, Seiten 805–826, 2007.

[KB09] Joachim Klein und Susanne Braun. Optimizing Maintenance of Constraint-Based Database Caches. In *ADBIS*, Seiten 219–234, 2009.

[Kle10] Joachim Klein. Concurrency and Replica Control for Constraint-based Database Caching. In *ADBIS*, Seiten 305–319. Springer, 2010.

[LAK10] LAKSHYA Solutions Ltd. CSQL - Der Full-Table-Cache für MySQL, PostgreSQL and Oracle - Dokumentation, 2010. http://www.csqldb.com/prod-Documentation.html.

[LGZ04] Per-Åke Larson, Jonathan Goldstein und Jingren Zhou. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE*, Seiten 177–189. IEEE Computer Society, 2004.

[LKPMJP05] Yi Lin, Bettina Kemme, Marta Patiño-Martínez und Ricardo Jiménez-Peris. Middleware-based Data Replication providing Snapshot Isolation. In *SIGMOD*, Seiten 419–430, 2005.

[Mic10] Microsoft. Microsoft SQL Server 2008 R2 Dokumentation, 2010. http://www.microsoft.com/sqlserver/.

[Ora10] Oracle. Oracle 11g R2 Dokumentation, 2010. http://www.oracle.com/technetwork/database/enterprise-edition/documentation/index.html.

[Pos10] PostgreSQL Global Development Group. PostgreSQL Dokumentation, 2010. http://www.postgresql.org/docs/.

[The02] The TimesTen Team. Mid-tier Caching: The TimesTen Approach. In *SIGMOD*, Seiten 588–593, 2002.

[WK05] Shuqing Wu und Bettina Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In *ICDE*, Seiten 422–433, 2005.