Selektives Laden und Entladen von Prädikatsextensionen beim Constraint-basierten Datenbank-Caching

Joachim Klein, Susanne Braun, Gustavo Machado

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

Zusammenfassung: Um die Antwortzeit von Anfragen an Datenbanksysteme zu verringern und die Skalierbarkeit zu erhöhen, werden beim Datenbank-Caching Teilmengen von Daten in der Nähe von Anwendungen vorgehalten. Im Gegensatz zu anderen Ansätzen wird hierbei eine deklarative Anfragebearbeitung durch das Cache-System angestrebt, welche die Auswertung einzelner Prädikate unterstützt, die in häufig auszuwertenden Anfragen auftreten. Beim Constraint-basierten Datenbank-Caching werden hierzu Bedingungen definiert, die eine korrekte Anfrageauswertung garantieren und zudem leicht überprüfbar sind. Dabei beschreiben die Constraints einen Abhängigkeitsgraphen, der das Laden und Entladen von Cache-Inhalten beeinflusst. Um dennoch eine bestmögliche Anfragebearbeitung zu gewährleisten, ist es wichtig, leistungsstarke Methoden zu entwickeln, die ein selektives Laden und Entladen zu verwaltender Einheiten (so genannter Cache Units) ermöglicht. Dieser Aufsatz beschreibt die bisherigen Ansätze und evaluiert erstmals explizit deren Performance. Die neu eingeführten Begriffe Cache-Unit und Cache-Unit-Differenz helfen dabei, die Größenverhältnisse der zu verwaltenden Einheiten zu beschreiben. Darüber hinaus werden neue Umsetzungen vorgestellt, die ein effizienteres Laden und Entladen als bisher ermöglichen und die Adaptivität des Gesamtsystems steigern.

1 Motivation

Heutzutage gibt es kaum noch Anwendungen, die nicht auf entfernt gespeicherte Daten angewiesen sind. Immer mehr Anwendungen und Dienste werden bereits direkt über das Internet abgerufen. So kann man heute schon auf komplette Office-Anwendungen im Web (so genannte Web-Anwendungen) zugreifen. Sie alle sind auf Daten aus leistungsstarken Datenspeichern angewiesen. Im Zuge dieser Entwicklung gewinnen Datenbanken, zu denen vor allem große relationale Datenbanksysteme gehören, zunehmend an Bedeutung. Immer mehr Daten, von nunmehr weltweit agierenden Benutzern, werden gespeichert und verarbeitet. Zusätzlich werden die Systeme inzwischen immer mehr durch Analysen (z. B. durch Data-Mining-Techniken) belastet.

Um die Zugriffszeit auf weit entfernte Datenbanken (oder auch ganze Datenbank-Cluster) zu verringern, versucht das Datenbank-Caching, ähnlich wie das Web-Caching, häufig zugegriffene Daten in der Nähe der Anwendung vorzuhalten. Gleichzeitig entlas-

ten die Cache-Instanzen so den zentralen Datenbank-Server (Backend). Die Existenz eines Cache-Systems ist hierbei für Anwendungen stets transparent, wodurch sie wie gewöhnlich auf die Daten, wie sie im Backend-Schema definiert sind, zugreifen können.

Die einfachste Art, einen Datenbank-Cache zu realisieren, besteht darin, komplette Tabellen repliziert vorzuhalten, was auch als Full-Table Caching [Ora08] bezeichnet wird. Da solche Ansätze keine dynamische Anpassung des Caches-Inhaltes an die tatsächliche Anfragelast zulassen, wurden mehrere weiterführende Konzepte entwickelt. Viele dieser Ansätze benutzen materialisierte Sichten (bzw. Anpassungen davon) [APTP03, BDD+98, GL01, LMSS95, LGZ04, LGGZ04]. Da alle diese Ansätze auf die Auswertung der durch materialisierte Sichten definierten Tabellen beschränkt sind, versuchen Konkurrenzansätze PSJ-Anfragen[1] über mehrere Cache-Tabellen zu unterstützen [ABK+03, The02]. Zu diesen Ansätzen gehört auch das Constraint-basierte Datenbank-Caching (CbDBC). Auch in der Industrie haben Datenbankhersteller die Ansätze übernommen, um ihre Produkte um einen entsprechenden Datenbank-Cache zu erweitern [ABK+03, LGZ04, The02].

Der Grundgedanke des CbDBC besteht darin, vollständige Extensionen (also Satzmengen) einzelner Prädikate vorzuhalten, die hierdurch für die Beantwortung unterschiedlicher Anfragen verwendbar sind. Dabei wird die Vollständigkeit dieser Prädikatsextensionen durch einfache Bedingungen (Constraints) sichergestellt (vgl. Abschnitt 2).

Da diese Constraints zu jeder Zeit vom Cache-Inhalt erfüllt sein müssen, hat die Cache-Wartung mengenorientiert unter Beachtung der definierten Constraints zu erfolgen. Das Laden von Satzmengen in den Cache und das Entladen dieser ist beim CbDBC von besonderer Bedeutung, da sich der Caching-Effekt erhöht, je schneller die Daten aktuell referenzierter Prädikate geladen und veraltete entfernt werden können [BK07]. Die typischerweise hohe Latenz zwischen Backend und Cache erschwert das Laden hierbei zusätzlich. Darüber hinaus entstehen durch die definierten Constraints Abhängigkeitsketten (typischerweise zwischen Tabellen, die häufig Verbundpartner sind), deren Beachtung beim Laden und Entladen erhebliche Komplexität in sich birgt. Vor allem zyklische Abhängigkeiten führen große Probleme ein (vgl. Abschnitt 3.2).

In [BHM06] wurden bereits die konzeptionellen Grundlagen des Ladens sowie eine erste Implementierung vorgestellt. Gezielte Untersuchungen der Performance bezüglich grundlegender Cache-Strukturen (vgl. Abschnitt 4.1) bestätigen die bereits erwarteten Performance-Einbrüche bei längeren Abhängigkeitsketten. Dieser Aufsatz stellt zwei neue Implementierungskonzepte vor, das *indirekte Laden* (im Abschnitt 4.2) und das *vorbereitete Laden* (in Abschnitt 4.3), welche die aufgetretenen Probleme lösen. Dabei werden die Vor- und Nachteile der verschiedenen Konzepte gegenübergestellt und durch Messungen überprüft.

Bezüglich des Entladens wurde bisher vorgeschlagen, den kompletten Cache-Inhalt zu bestimmten Zeitpunkten zu löschen, da ein selektives Löschen in Zyklen zu gesonderten Problemen führt [HB07] (vgl. auch Abschnitt 3.2). Wir stellen in Abschnitt 5 eine neue Methode vor, die es erlaubt, das selektive Entladen performant durchzuführen. Zunächst werden jedoch in Abschnitt 2 die grundlegenden Definitionen des CbDBC (aus [HB07]) zum besseren Verständnis des Aufsatzes wiederholt.

[1] Projektion-Selektion-Join-Anfragen

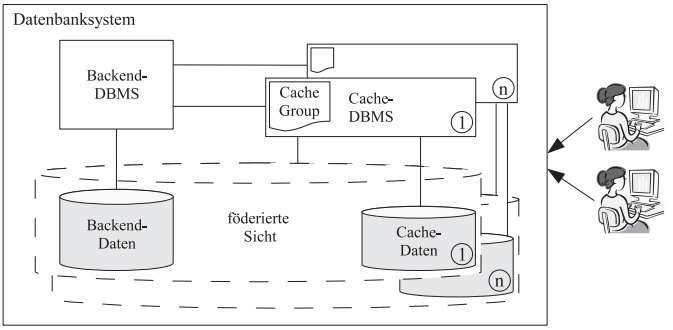

Abbildung 1: Der grundlegende Aufbau eines CbDBC-Systems

2 Constraint-basiertes Datenbank-Caching

Das CbDBC hält Satzmengen häufig angefragter Prädikate in Cache-Instanzen vor und beschleunigt so den lesenden Zugriff. Die Daten werden in Cache-Tabellen gespeichert und erfüllen dabei stets die aktuell gültige Menge von Constraints. Das Cache-Verwaltungssystem verwendet die Constraints, um entscheiden zu können, ob eine Anfrage oder Teilprädikate davon beantwortet werden können.

Die grundlegende Struktur des Cache-Systems wird durch Abbildung 1 verdeutlicht. Jede Cache-Instanz bedient sich einer föderierten Sicht, um in Anfragen sowohl auf Cache-Tabellen als auch auf Backend-Tabellen zugreifen zu können. Das komplette Datenbanksystem besteht nun aus einem Backend-DBS und eventuell mehreren Cache-Instanzen. Die Menge an Cache-Tabellen und die Menge aller Constraints, die eine Cache-Instanz verwaltet, werden in einer so genannten *Cache Group* zusammengefasst.

2.1 Elemente einer Cache Group

Cache-Tabellen sind logische Teilkopien entsprechender Backend-Tabellen, da sie stets nur eine (häufig benötigte) Teilmenge ihrer Daten enthalten. Dabei entspricht die Schema-Definition einer Cache-Tabelle der ihr zugeordneten Backend-Tabelle bis auf Fremdschlüssel, die nicht übernommen werden. Primärschlüsseldefinitionen und Unique-Constraints bleiben jedoch erhalten. Die zu einer Backend-Tabelle T gehörende Cache-Tabelle bezeichnen wir fortan als T_C.

Constraints. Die einzigen in einer Cache Group verwendeten Constraints sind *Füllspalten* (filling column, FC) und *referentielle Cache-Constraints* (referential cache constraint, RCC)[2]. Über die Inhalte einer Füllspalte (vgl. Abbildung 2a) entscheidet das Cache-System, ob Werte in den Cache zu laden sind. Ein Ladevorgang wird immer dann ausgeführt, wenn eine Anfrage einen Wert der Füllspalte explizit referenziert, z. B. durch die

[2]Inwiefern ein RCC von den bekannten Konzepten einer Primär-Fremdschlüsselbeziehung abweicht, wird auch in [HB07] ausführlich erläutert.

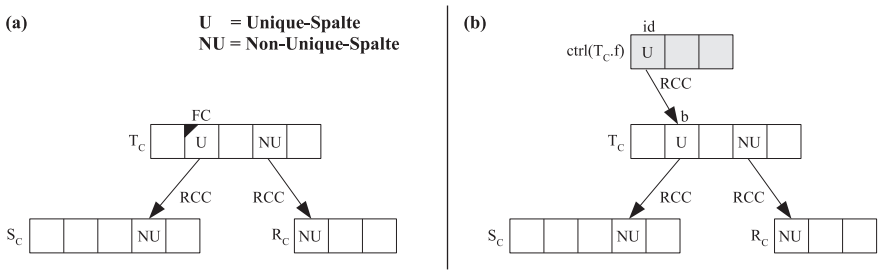

Abbildung 2: Konzeptionelle (a) und interne (b) Darstellung einer Cache Group

Anfrage $\sigma_{id=10}T$. Ist der referenzierte Wert in einer zuvor definierten Kandidatenmenge enthalten, so wird er, sofern noch nicht vorhanden, *wertvollständig* in den Cache geladen.

Definition 2.1 (Wertvollständigkeit) *Ein Wert einer Spalte wird als wertvollständig bezeichnet (oder kurz vollständig), wenn alle Sätze der zugehörigen Backend-Tabelle, die den gleichen Spaltenwert enthalten, auf dem Cache verfügbar sind. Somit ist der Wert v vollständig in $T_C.a$ genau dann, wenn alle Sätze $\sigma_{a=v}T$ im Cache verfügbar sind.*

Mit einem RCC lassen sich zwei Spalten aus Cache-Tabellen miteinander verbinden, die den gleichen Wertebereich haben. Er wird als gerichtete Kante zwischen den Spalten dargestellt (vgl. Abbildung 2). Ein RCC verlangt, dass alle Werte, die in der Quellspalte eines RCC enthalten sind, in der Zielspalte wertvollständig vorliegen. Es lässt sich nun bereits erkennen, dass alle über RCCs abhängigen Sätze (nach Beendigung eines Ladevorgangs) im Cache vorliegen müssen.

Definition 2.2 (Füllspalte) *Eine Cache-Spalte $T_C.b$, die als Füllspalte deklariert ist, lädt alle Werte $v \in K$, die explizit durch eine Anfrage $\sigma_{b=v}T$ referenziert werden, wertvollständig in den Cache. Hierbei beschreibt K die Menge aller ladbaren Füllspaltenwerte (Kandidatenmenge).*

Definition 2.3 (RCC) *Ein referenzieller Cache-Constraint $T_C.b \rightarrow S_C.a$ zwischen einer Quellspalte $T_C.b$ und einer Zielspalte $S_C.a$ ist genau dann erfüllt, wenn alle Werte v aus $T_C.b$ wertvollständig in $S_C.a$ sind.*

Da die Zusicherungen der beiden Constraints hauptsächlich auf der zugrundeliegenden Wertvollständigkeit basieren, lässt sich das Konzept noch weiter vereinfachen. Zunächst bezeichnen wir jede RCC-Quellspalte als *Kontrollspalte*, da sie für alle in ihr enthaltenen Werte die Wertvollständigkeit in der RCC-Zielspalte erzwingt. Um die Wertvollständigkeit einer Füllspalte (etwa $T_C.f$ aus Abbildung 2b) zu garantieren, wird jeweils eine interne (nur auf dem Cache verwaltete) Tabelle hinzugefügt. Diese wird *Kontrolltabelle* $\text{ctrl}(T_C.f)$ genannt und speichert in einer Kontrollspalte $\text{ctrl}(T_C.f).id$ alle Werte, die in $T_C.f$ wertvollständig geladen sind. Ein in die Spalte $\text{ctrl}(T_C.f).id$ eingelagerter Wert wird als *Füllwert* bezeichnet. Ein RCC realisiert die Eigenschaften der Füllspalte und ermöglicht es fortan, alle Funktionalitäten (wie auch das Laden und Entladen) nur noch mit Hilfe von RCCs und Cache-Tabellen zu beschreiben.

3 Grundlagen des Ladens und Entladens

Wodurch und wann das Laden von Sätzen in den Cache ausgelöst wird, wurde bereits im vorangegangenen Abschnitt beschrieben. In diesem Abschnitt wenden wir uns nun eingehend den speziellen Eigenschaften zu und den Problemen, die beim Laden und Entladen auftreten. Zunächst schaffen wir einige theoretische Grundlagen, um die Menge der Sätze, die zu laden und zu entladen ist, besser beschreiben zu können. Die Menge an Sätzen, die aufgrund eines Kontrollwertes v in $T_C.a$ auf dem Cache vorliegen müssen, wird als *RCC-Hülle* bezeichnet.

Definition 3.1 (RCC-Hülle) *Sei $T_C(a_1,\ldots,a_n)$ eine Cache-Tabelle mit einem ausgehenden RCC $T_C.a_i \rightarrow R_C.b$ und $s = (w_1,\ldots,w_n) \in T_C$ ein Satz mit $w_i = v$. Die Menge aller Sätze, die aufgrund von $s \in T_C$ mit $w_i = v$ im Cache vorliegen müssen, wird als RCC-Hülle von v in $T_C.a_i$ (oder kurz: $RH_{T_C.a_i}(v)$) bezeichnet.*

Für das Laden und Entladen von Sätzen sind die RCC-Hüllen der Füllwerte besonders wichtig, da durch sie die Mengen aller Sätze, die auf dem Cache vorliegen müssen, festgelegt sind. Aus diesem Grund bezeichnen wir die RCC-Hülle eines Füllwertes als *Cache Unit*.

Definition 3.2 (Cache Unit) *Sei $T_C.f$ eine Füllspalte. Die RCC-Hülle eines Füllwertes v aus* ctrl$(T_C.f)$*.id bezeichnen wir als Cache Unit von v in $T_C.f$ (oder kurz als $CU_{T_C.f}(v)$).*

Ein Satz, der in den Cache geladen wurde, kann zu mehreren Cache Units gehören. Die Anzahl der Sätze, die während eines Ladevorgangs tatsächlich geladen werden müssen bzw. während des Entladens gelöscht werden können, kann daher sehr viel kleiner sein als die Menge der Sätze in der Cache Unit. Die Menge der zu ladenden bzw. zu verdrängenden Sätze hängt also stark von den bereits geladenen Cache Units ab und wird daher erstmals als *Cache-Unit-Differenz* konkret definiert.

Definition 3.3 (Cache-Unit-Differenz) *Sei $CU_{T_C.f}(v)$ eine Cache Unit, die zu laden bzw. zu entladen ist, und I die Vereinigung aller übrigen Cache Units, die unverändert im Cache bleiben. Die Menge $CU_{T_C.f}(v) - I$ aller Sätze, die tatsächlich geladen bzw. verdrängt werden müssen, wird als Cache-Unit-Differenz von v in $T_C.f$ (oder kurz: CU-Diff$_{T_C.f}(v)$) bezeichnet.*[3]

Durch die vorangegangenen Betrachtungen wird deutlich, dass es aufgrund der Struktur einer Cache Group evtl. schwer sein kann, die Sätze zu finden, die zur jeweiligen Cache-Unit-Differenz gehören. Um ein besseres Verständnis für diese Schwierigkeiten zu erlangen, isolieren wir zunächst die Zyklen innerhalb einer Cache Group.

3.1 Atomare Zonen

Die Struktur einer Cache Group, bestehend aus Cache-Tabellen und RCCs, lässt sich als gerichteter Graph mit Tabellen als Knoten und RCCs als Kanten betrachten. Zyklische

[3] Auf eine noch formalere Definition wurde bewusst verzichtet, da sie in der Folge nicht benötigt wird.

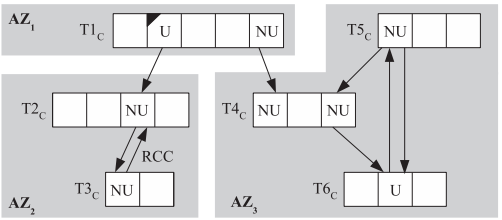

Abbildung 3: Die Aufspaltung einer Cache Group in ihre atomaren Zonen

Abhängigkeiten verursachen in diesem Graph die größten Probleme und werden daher separat in Abschnitt 3.2 betrachtet. Um einen azyklischen Graph zu bilden, werden die Zyklen in *atomaren Zonen (AZ)* zusammengefasst (vgl. auch [BHM06]), die sich isoliert betrachten lassen. Auch Cache-Tabellen, die nicht Teil eines Zyklus sind, stellen atomare Zonen dar. Wir nennen diese *triviale atomare Zonen*. Cache-Tabellen, die zu einem Zyklus gehören, werden in einer so genannten *nicht-trivialen atomaren Zone* zusammengefasst. Abbildung 3 zeigt ein Beispiel, in dem die Aufspaltung einer Cache Group in ihre Zonen verdeutlicht wird. Dabei wird ein RCC, der zwischen zwei Zonen verläuft, als *externer RCC* bezeichnet, ein RCC, der innerhalb einer atomaren Zone verläuft, als *interner RCC*.

Betrachten wir die atomaren Zonen als Knoten und die externen RCCs als Kanten, so entsteht ein gerichteter azyklischer Graph, der es uns ermöglicht, das Laden und Entladen von Werten auf einer höheren Abstraktionsebene zu betrachten, ohne Zyklen zu berücksichtigen.

Wir betrachten erneut Abbildung 3: Bereits in [BHM06] wurde diskutiert, in welcher Reihenfolge die Sätze in die Cache-Tabellen einzufügen sind (top-down oder bottom-up). Dabei wurde festgestellt, dass das Einladen der Daten von unten nach oben den enormen Vorteil bietet, pro atomarer Zone eine Fülltransaktion bilden zu können, ohne RCCs zu verletzen. Dadurch ist es z. B. möglich, die neu geladenen Sätze der RCCs in AZ_3 bereits vor der Beendigung des kompletten Ladevorgangs für anstehende Anfragen zu nutzen. Würden die Sätze von oben nach unten eingefüllt, dürfte das Commit der Ladetransaktion erst nach dem Laden aller atomaren Zonen ausgeführt werden, um die Konsistenz der Cache-Inhalte und eine korrekte Anfrageauswertung zu gewährleisten.

Das Entladen von Cache Units bietet die gleichen Vorteile, wenn die atomaren Zonen von oben nach unten entladen werden und dabei die betroffenen Sätze jeweils transaktionsgeschützt entfernt werden. Deshalb ist zuerst der Füllwert aus der Kontrolltabelle zu löschen. Danach sind die Daten aus AZ_1, dann aus AZ_2 und zuletzt aus AZ_3 zu löschen. Auch hier können die Transaktionen nach Bearbeitung einer atomaren Zone abgeschlossen werden.

Um jeweils die richtige Ausführungsreihenfolge zu finden, wird ein einfacher Algorithmus [CLRS82] eingesetzt, der die Zonen topologisch sortiert und somit eine totale Ordnung bestimmt.

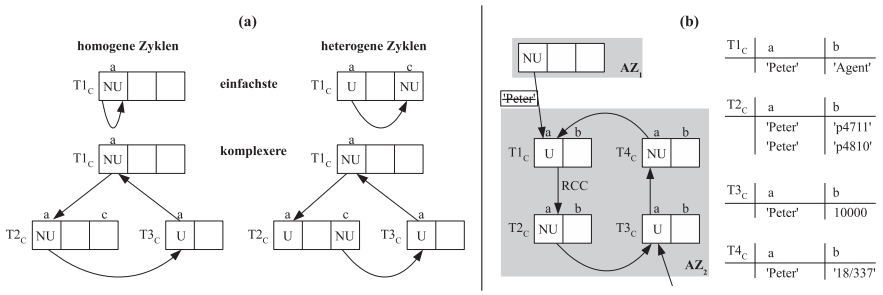

Abbildung 4: (a) Beispiele für homogene und heterogene Zyklen, (b) Wertebeispiel

3.2 Probleme in Zyklen

Nachdem wir bisher durch die Einführung der atomaren Zonen von Zyklen abstrahiert haben, werden wir uns nun mit den Problemen beschäftigen, die innerhalb nicht-trivialer Zonen auftreten. Die Zyklen werden nochmals unterschieden in *homogene Zyklen* und *heterogene Zyklen*. Abbildung 4a zeigt einige Beispiele von homogenen und heterogenen Zyklen. Wir nennen einen Zyklus aus RCCs homogen, wenn nur eine Spalte pro Tabelle Teil des Zyklus ist. Dies ist z. B. im Zyklus $T1_C.a \rightarrow T2_C.a \rightarrow T3_C.a \rightarrow T1_C.a$ der Fall. Im Gegensatz dazu nennen wir einen Zyklus heterogen, wenn mehr als eine Spalte pro Tabelle beteiligt ist [ABK+03].

Während des Ladens treten die Hauptprobleme in heterogenen Zyklen auf, weil dabei ein *rekursives Laden* ausgelöst werden kann. Bei einem Zyklusdurchlauf können nämlich Sätze eingelagert werden, bei denen Werte aus mehreren Spalten die Fortsetzung des Ladevorgangs bestimmen und somit erneute Zyklusdurchläufe erzwingen. Dieses Problem versucht Beispiel 3.1 zu verdeutlichen; es ist ausführlich in [HB07] beschrieben. Das gleiche Problem entsteht auch beim Entladen, wobei zu große Cache-Unit-Differenzen entstehen können, die dann zu entladen wären.

Homogene Zyklen sind hingegen gutartig. In ihnen stoppt der Ladevorgang stets spätestens nach einem Zyklusdurchlauf, weil in den bestimmenden Spalten beim Einfügen keine neuen Werte auftreten. Beim Entladen in homogenen Zyklen ist die Situation jedoch ungleich komplizierter. Die zyklische Abhängigkeit unter den Werten muss erkannt und aufgelöst werden. Wir erläutern dieses Problem ausführlich in Beispiel 3.2.

Beispiel 3.1. [Rekursives Laden/Entladen]
Wir betrachten den einfachsten heterogenen Zyklus aus Abbildung 4. Sobald ein Wert w_1 in Spalte $T1_C.a$ geladen wird, müssen auch alle abhängigen Sätze mit den Werten $T1_C.c = w_1$ in die Tabelle $T1_C$ geladen werden. Dabei können wiederum neue Werte $w_2, ..., w_k$, die zuvor nicht in der Tabelle vorhanden waren, in der Spalte $T1_C.a$ auftreten. Das Laden setzt sich rekursiv fort, bis keine neuen Werte mehr auftauchen.

Im schlimmsten Fall werden alle Daten des Backends in die Cache-Tabellen geladen. Um Situationen zu vermeiden, in denen ein rekursives Laden bzw. Entladen möglich ist, werden heterogene Zyklen beim Design einer Cache Group ausgeschlossen. In homogenen

Zyklen stoppt der Ladeprozess nach dem ersten Zyklusdurchlauf, da keine neuen Kontrollwerte mehr auftauchen können. Beim Entladen ist es jedoch schwer, die Abhängigkeiten innerhalb des Zyklus richtig zu erkennen, wie das folgende Beispiel belegt.

Beispiel 3.2. [Abhängigkeiten in homogenen Zyklen]
Abbildung 4b zeigt einen homogenen Zyklus, in dem der Wert *'Peter'* bereits aus AZ_1 gelöscht wurde. Versuchen wir nun herauszufinden, ob der Wert *'Peter'* auch innerhalb von AZ_2 gelöscht werden kann, so muss zunächst die zyklische Abhängigkeit $T1_C.a \rightarrow T2_C.a \rightarrow T3_C.a \rightarrow T4_C.a \rightarrow T1_C.a$ innerhalb der atomaren Zone erkannt werden. Der Wert *'Peter'* kann aber auch von externen RCCs abhängig sein, wie es z. B. durch Tabelle $T3_C$ möglich wäre. Referenziert der externe RCC den Wert, so kann *'Peter'* aus keiner Tabelle entfernt werden. Tut er dies nicht, so sind alle in Abbildung 4b gezeigten Sätze löschbar. Darüber hinaus ist stets zu beachten, dass die Abhängigkeitskette innerhalb eines Zyklus unterbrochen sein kann. Dies ist der Fall, wenn zu einem Kontrollwert in der Zieltabelle des RCC kein Satz mit dem entsprechenden Wert existiert. Nehmen wir an, dass in Tabelle $T3_C$ oder $T4_C$ kein Satz mit dem Wert *'Peter'* existiert, so können zumindest die entsprechenden Sätze der Tabellen $T1_C$ und $T2_C$ gelöscht werden.

Aufgrund dieser Probleme wurde in [HB07] als vorläufige Lösung vorgeschlagen, den kompletten Inhalt des Cache zu bestimmten Zeitpunkten zu löschen. Selbst wenn man einen günstigen Zeitpunkt für einen solchen Leerstart abwartet, so ist der zu erwartende Aufwand des Nachladens, bis der Cache wieder gefüllt ist, beträchtlich. Ein *selektives Entladen* einzelner Cache Units, wie es in Abschnitt 5 beschrieben wird, vermeidet das Löschen des gesamten Inhalts.

3.3 Messungen

Die in den folgenden Abschnitten diskutierten Messungen wurden mit dem Ziel durchgeführt, den Aufwand für das Laden und Entladen grundlegender Cache-Group-Strukturen abschätzen zu können. In diesen Experimenten wurde bewusst nicht die Performance des Gesamtsystems vermessen, damit gezielte Aussagen über die hier betrachteten Hauptfunktionen möglich sind. Darüber hinaus wurde zur Vereinfachung die Latenz zwischen Backend und Cache nicht variiert. In allen Messungen wird die Anzahl der zu ladenden bzw. zu löschenden Sätze schrittweise erhöht. Die dabei angegebene Anzahl der zu ladenden Sätze entspricht der Größe der Cache-Unit-Differenz des jeweils geladenen oder entladenen Füllwertes. Dabei wurden die Sätze möglichst gleichmäßig über die Menge der Cache-Tabellen in der Cache-Group-Struktur verteilt.

Wichtige Cache-Group-Strukturen. Wie die vorangegangenen Ausführungen verdeutlichen, sind vor allem unterschiedlich lange RCC-Ketten und homogene Zyklen für die Messungen interessant. Wir haben dieser Menge grundlegender Strukturen noch Bäume hinzugefügt, bei denen jede innere Cache-Tabelle zwei ausgehende RCCs aufweist. In Abbildung 5 sind einige Beispiele dieser Strukturen aufgeführt, wobei die jeweilige Kontrolltabelle mit angegeben ist. Die einfachste Struktur, die vermessen wurde, bestand aus

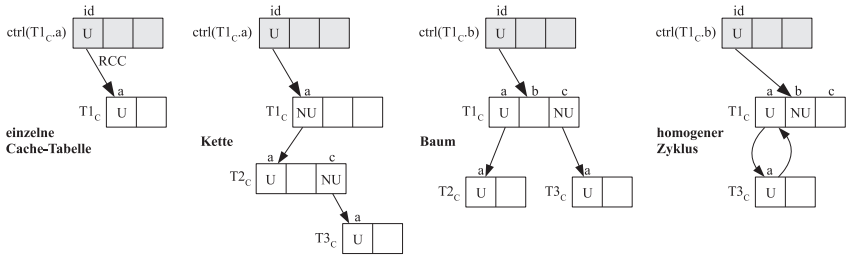

Abbildung 5: Beispiele wichtiger Cache-Group-Strukturen

einer einzelnen Cache-Tabelle mit einer Füllspalte, sodass eine Kette mit einem RCC (zwischen Kontrolltabelle und Cache-Tabelle) entstand.

Datengenerator. Ein eigens entwickelter Datengenerator analysiert vor jeder Messung die ihm übergebene Cache Group und versucht, eine Cache-Unit-Differenz bestimmter Größe (z. B. mit 2000 Sätzen) zu generieren, die er auf alle Cache-Tabellen gleich verteilt[4]. Dies bedeutet für die Vermessung einer RCC-Kette mit drei Cache-Tabellen, dass im Beispiel 666 Sätze für die erste Cache-Tabelle, 666 für die zweite und 667 für die dritte Tabelle generiert werden. Die so generierten Sätze werden im Backend hinterlegt.

Alle verwendeten Tabellen besitzen sieben vollbesetzte Spalten. Sind diese Teil einer RCC-Definition, wurde als Datentyp `INTEGER` definiert, für alle anderen Spalten wurde der Datentyp `VARCHAR(300)` verwendet.

Messaufbau. Als Cache-System wurde der Prototyp des ACCache-Projekts [Mer05, BHM06] eingesetzt. Die Funktionalität des Systems wurde durch die in Abschnitt 4 und 5 beschriebenen Methoden entsprechend erweitert.

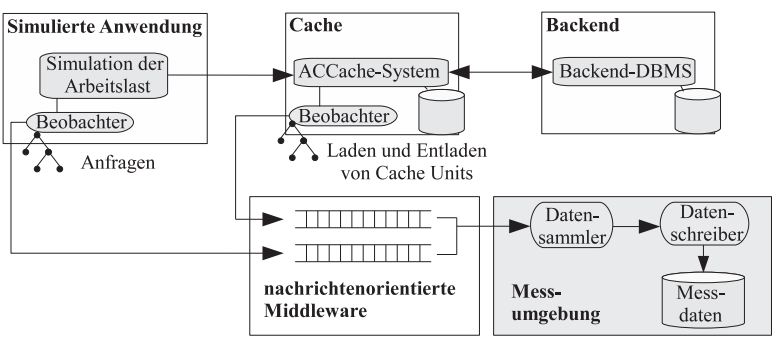

Abbildung 6: Aufbau der Messungen

[4]Es gibt Cache-Group-Strukturen, bei denen eine Gleichverteilung nicht gewährleistet werden kann. Diese leisten jedoch für eine Aufwandsabschätzung keinen gesonderten Beitrag.

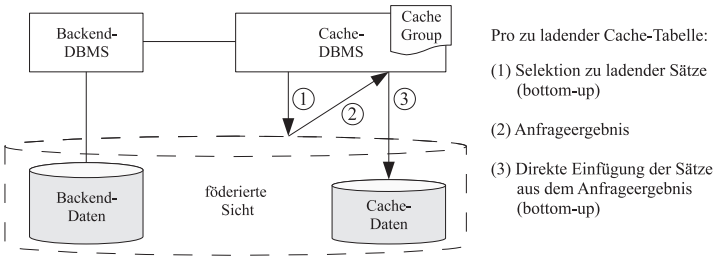

Abbildung 7: Ablaufdiagramm für das direkte Laden

Die verschiedenen Messungen wurden mit einer von uns selbst entwickelten Messumgebung durchgeführt, wobei jeder Messlauf sechsmal ausgeführt wurde. Vor jedem Durchlauf wurden die Daten der Cache Units neu generiert, sodass die Anzahl der Sätze in der Cache-Unit-Differenz zwar gleich blieb, die Daten sich aber jedes mal änderten. Alle Anwendungen wurden durch die Messumgebung jeweils gestoppt und wieder neu gestartet. Die in den folgenden Abschnitten gezeigten Grafiken zeigen jeweils den Mittelwert der Messergebnisse aus den sechs zusammengehörigen Durchläufen.

Die Messumgebung beschreibt die an einer Messung teilnehmenden Anwendungen mittels *Arbeitsknoten*. In allen Messungen wurden drei Arbeitsknoten verwendet: die simulierte Client-Anwendung, die durch entsprechende Anfragen das Laden der Cache Units auslöst, das ACCache-System und das Backend-DBS (vgl. Abbildung 6).

4 Laden von Cache Units

4.1 Direktes Laden

Wie bereits in Abschnitt 3.1 erwähnt, betrachten wir zuerst das in [BHM06] vorgestellte Konzept, wobei die Sätze über die atomaren Zonen von unten nach oben, *direkt* in den Cache eingefügt werden. Wir bezeichnen diese Methode daher als *direktes Laden*. Abbildung 7 zeigt den Ablauf des Ladevorgangs.

Für jede Cache-Tabelle wird vom Cache-System ein `INSERT`-Statement an die Datenbank gesendet, welches die zu ladenden Sätze auswählt und direkt in die Tabelle einfügt. Das Cache-System greift dabei über die föderierte Sicht auf die Datenbank zu, um die ausgewählten Sätze der Cache Unit mit den bereits geladenen vergleichen zu können (vgl. auch [BHM06]).

Messergebnisse. Abbildung 8 zeigt die Messergebnisse für die in Abschnitt 3.3 besprochenen Cache-Group-Strukturen. Um die übrigen Messungen besser einordnen zu können, betrachten wir zunächst das Laden in die einzelne Cache-Tabelle. Die verbleibenden Messungen lassen sich somit relativ zu dieser bewerten. Die in den Messungen gewählte maximale Anzahl von 3000 zu ladenden Sätzen ist recht gering, sodass es keinen erhöhten Aufwand darstellt, diese auszuwählen und einzufügen. Das auf der Cache-Instanz verwen-

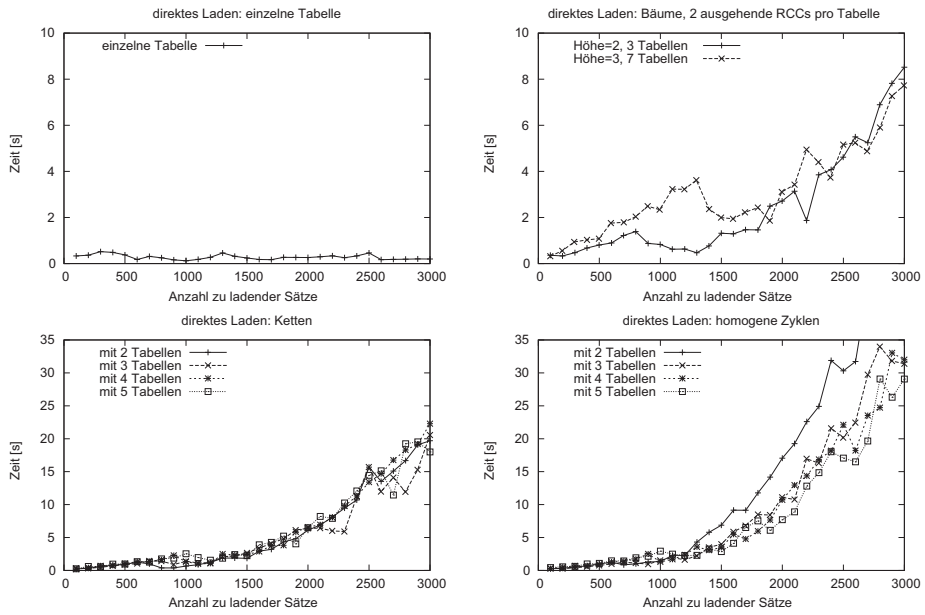

Abbildung 8: Ergebnisse des direkten Ladens: einzelne Knoten, Ketten, Bäume, homogene Zyklen

dete Datenbanksystem erfüllt die Aufgabe stets in einer Geschwindigkeit, die unter 500 ms liegt.

Bereits bei der Auswertung einer Kette mit nur zwei Cache-Tabellen ergibt sich ein inakzeptabel erhöhter Aufwand von bis zu 23 Sekunden, der auch bei der Vermessung längerer Ketten stabil bleibt. Dieses Verhalten gründet sich auf zwei gegenläufige Effekte beim Auswerten der für das Laden erforderlichen Anfragen: Zum einen erhöht sich der Aufwand durch die Hinzunahme einer weiteren Cache-Tabelle, da für das Laden in die unterste Tabelle eine zusätzliche Anfrage benötigt wird, in der ein Verbund mit allen darüber liegenden Tabellen vorzunehmen ist. Zum anderen verringert sich die Anzahl der jeweils zu selektierenden Sätze pro Tabelle, da die Sätze der Cache Unit auf alle Tabellen gleichmäßig verteilt wurden. Hierdurch sinkt der Aufwand des jeweils auszuführenden Verbunds.

Innerhalb von Zyklen ist der Aufwand sogar so hoch, dass teilweise für das Laden einer Cache Unit bis zu 55 Sekunden notwendig waren. Da in Bäumen die Anzahl der notwendigen Verbunde von ihrer Höhe abhängt, ist der Aufwand hier deutlich geringer, jedoch immer noch sehr hoch (bis zu 9 Sekunden).

Für alle Strukturen ist klar erkennbar, dass der Aufwand stark ansteigt, sobald die Menge der zu ladenden Sätze signifikant hoch ist (> 1000 Sätze). Bei kleinen Mengen (< 1000 Sätze) liegt der Aufwand hingegen meist unter einer Sekunde.

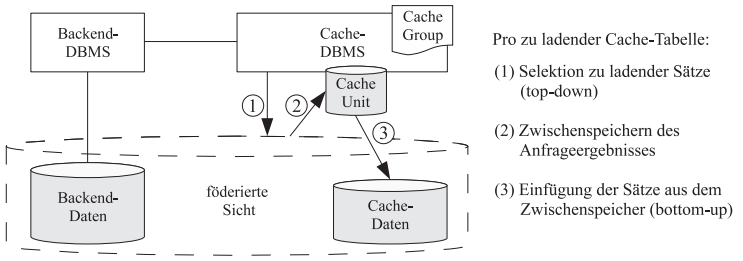

Abbildung 9: Ablauf des indirekten Ladens

4.2 Indirektes Laden

Der Ansatz des indirekten Ladens versucht den hohen Selektionsaufwand des direkten Ladens zu vermeiden, indem die Sätze einer Cache Unit zuvor in so genannte *Staging-Tabellen* geladen werden. Dies ermöglicht es, ausgehend vom Füllwert, die Sätze einer Cache Unit von oben nach unten (top-down) zu bestimmen. Dabei werden neu geladene Kontrollwerte in so genannte *Kontrollwerttabellen* geschrieben, anhand derer die zu ladenden Sätze nachfolgender Tabellen bestimmt werden. Sobald eine Cache Unit komplett zwischengespeichert wurde, kann sie wie bisher von unten nach oben geladen werden, wodurch keine RCCs verletzt werden. Abbildung 9 zeigt den schematischen Ablauf des Ladens.

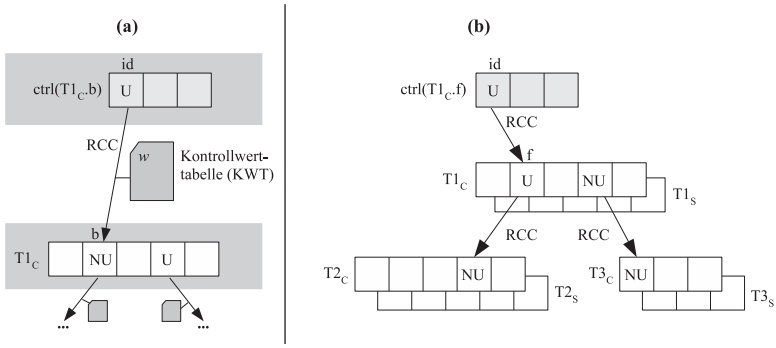

Abbildung 10: Die neuen Elemente: Kontrollwerttabellen (a), Staging-Tabellen (b)

Zusammenstellen der Cache Unit. Für jede Cache-Tabelle wird eine Staging-Tabelle mit identischen Spaltendefinitionen angelegt. Die zu einer Cache-Tabelle $T1_C$ gehörende Staging-Tabelle bezeichnen wir mit $T1_S$. Um alle Sätze einer Cache Unit aufzusammeln, wird zunächst der zu ladende Füllwert (z. B. *w*, vgl. Abbildung 10a) in die Kontrollwerttabelle (KWT) des ausgehenden RCCs eingefügt. Jeder RCC verfügt über eine solche Tabelle, um geladene oder gelöschte Kontrollwerte weiterzureichen. Das Zusammenstellen der Daten kann nun durch einen einfachen rekursiven Algorithmus realisiert werden,

der für jede Tabelle, die noch zu verarbeitende KWTs eingehender RCCs aufweist, fehlende Sätze anfragt und neu geladene Kontrollwerte weiterreicht. Besitzt keine KWT mehr zu verarbeitende Kontrollwerte, ist das Zusammenstellen der Cache Unit beendet. Diese Methode, ist unabhängig von der Struktur der Cache Group und würde auch das Laden in heterogenen Zyklen ermöglichen.

Vergleich: Direktes/indirektes Laden. In Abbildung 11 sind die Ergebnisse des Vergleichs zwischen direktem und indirektem Laden aufgeführt. Hierzu wurden genau die Cache-Group-Strukturen vermessen, bei denen die Performance-Einbrüche am deutlichsten waren, da das indirekte Laden primär dazu entwickelt wurde, diese Problemsituationen zu bewältigen. Das Ergebnis ist überdeutlich: In beiden Fällen bleibt der Aufwand für das indirekte Laden unter 3 Sekunden. Meistens wird sogar nur eine Sekunde benötigt.

Dieses Ergebnis kann eventuell noch verbessert werden. Das indirekte und auch das direkte Laden werden vom Cache allein durchgeführt. In beiden Fällen sind daher zum Zusammenstellen der Cache Unit mehrere Anfragen notwendig, die auch Backend-Tabellen enthalten. Ist die Latenz zwischen Backend und Cache hoch, wovon grundsätzlich ausgegangen wird, muss diese somit auch mehrfach in Kauf genommen werden. Das nachfolgend vorgestellte Konzept des *vorbereiteten Ladens*, versucht diesen Nachteil zu vermeiden.

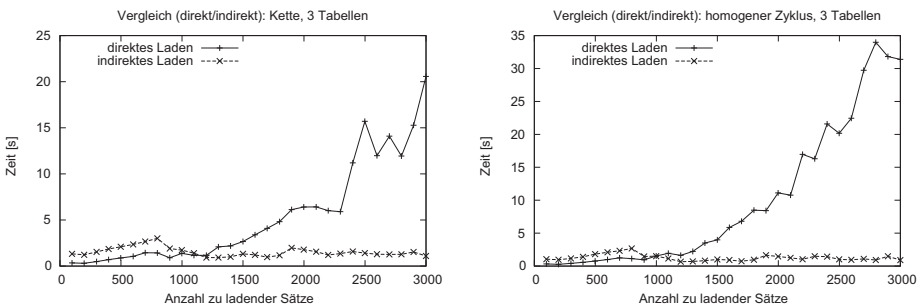

Abbildung 11: Vergleich des direkten und indirekten Ladens

4.3 Vorbereitetes Laden

In dieser Variante übernimmt das Backend-DBMS die Zusammenstellung der Cache Unit (vgl. Abbildung 12). Das Cache-DBMS fordert die Cache Unit an, indem es den zu ladenden Füllwert an das Backend-System übergibt. Das Auswählen der Sätze erfolgt dabei wie beim indirekten Laden. Damit dies möglich ist, muss das Backend-DBMS die Definition der Cache Group kennen. Es wird also durch die Selektion der Daten und durch die Wartung zusätzlicher Metadaten belastet. Da es jedoch für andere Funktionalitäten (z. B. die Synchronisation) sowieso notwendig ist, Metadaten der Cache-Instanzen auf dem Backend zu verwalten, wird der Zusatzaufwand für das Verwalten der Metainformationen relativiert.

Sobald die Cache Unit zusammengestellt ist, wird sie an den Cache gesendet, der die Sätze direkt von unten nach oben einfügt. Ein Zwischenlagern der Daten in Staging-Tabellen entfällt. Diese Methode kann vermutlich nur dann einen zusätzlichen Vorteil bieten, wenn zwischen Backend und Cache eine hohe Latenz herrscht. Eine Vermessung der Auswirkungen hoher Latenzen wurde im Rahmen dieser Arbeit noch nicht durchgeführt und sollte Bestandteil zukünftiger Aktivitäten sein.

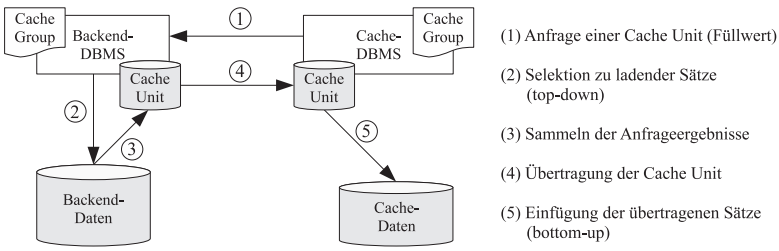

Abbildung 12: Ablauf des vorbereiteten Ladens

4.4 Zusammenfassung

Die durchgeführten Messungen zeigen, dass das direkte Laden nur schlecht skaliert und zum Einfügen größerer Cache-Unit-Differenzen nicht gut geeignet ist. Die schlechte Performance führt dazu, dass die zu ladenden Prädikatsextensionen im Cache erst viel zu spät nutzbar werden. Die Methode des indirekten Ladens beschleunigt das Laden der Sätze enorm und somit auch die Geschwindigkeit, in der die neu geladenen Prädikate für die Anfragebearbeitung nutzbar werden. Bei sehr kleinen Cache Units bietet das direkte Einladen jedoch Vorteile, da hierbei eine Indirektion (Zwischenspeichern der Daten) vermieden wird. Mit dem vorbereiteten Laden wurde eine weitere Methode entwickelt, die mehrere entfernte Anfragen auf Backend-Daten vermeidet und dadurch evtl. hohe Latenzzeiten reduzieren kann. Ein CbDBC-System, welches alle Methoden implementiert, kann leicht in die Lage versetzt werden, je nach Situation die richtige Variante dynamisch während der Laufzeit auszuwählen. Gleichzeitig zeigen die Messungen, dass hierfür die Größe der Cache Unit, die Struktur der Cache Group und die Höhe der Latenz wichtige Kenngrößen darstellen.

5 Entladen von Cache Units

In diesem Abschnitt stellen wir ein Konzept zum selektiven Entladen von Cache Units vor. Wie bereits in Abschnitt 3.1 erklärt, werden beim Entladen die atomaren Zonen der Cache Group von oben nach unten durchlaufen. Wir bezeichnen diese Vorgehensweise auch als *vorwärtsgerichtetes Entladen*, bei welchem die gelöschten Kontrollwerte wiederum mithilfe der in Abschnitt 4.2 vorgestellten Kontrollwerttabellen weitergereicht werden.

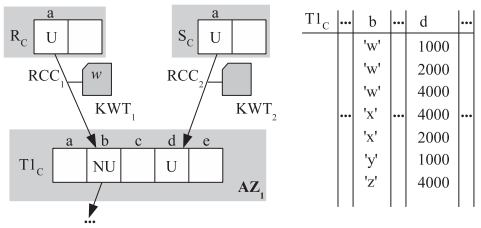

Abbildung 13: Löschen innerhalb einer trivialen Zone

Im Folgenden betrachten wir zunächst das vergleichsweise einfache Entfernen der Sätze aus einer trivialen atomaren Zone. Danach wird das Entladen aus einer nicht-trivialen atomaren Zone besprochen. Abschnitt 5.3 diskutiert die Ergebnisse der Messungen. Zuletzt wird in Abschnitt 5.4 die Verdrängungsstrategie des Cache-Systems vorgestellt.

5.1 Entladen in trivialen atomare Zonen

Zur besseren Veranschaulichung benutzen wir die in Abbildung 13 gezeigte Cache Group. Durch die vorberechnete Reihenfolge, in der die atomaren Zonen abzuarbeiteten sind, ist sichergestellt, dass alle KWTs eingehender externer RCCs mit den vollständig gelöschten Werten aus vorangegangenen Zonen gefüllt sind.

In Abbildung 13 wird der Löschvorgang durch den Wert w in KWT_1 ausgelöst. KWT_1 ist somit *Ausgangspunkt* für den Löschvorgang, der in Tabelle $T1_C$ gestartet werden muss. Wir nennen die Zieltabelle eines externen RCCs, dessen KWT gefüllt ist, daher auch *Starttabelle*.

Um die löschbaren Sätze zu bestimmen, muss jeder Satz mit $T1_C.b = \text{'}w\text{'}$ überprüft werden. Im Beispiel können alle diejenigen Sätze entfernt werden, deren Anwesenheit nicht durch die Kontrollwerte des RCC_2 verlangt wird. Enthält die Quellspalte des RCC_2 z. B. nur den Wert 1000, so können alle Sätze $\sigma_{(b=\text{'}w\text{'} \wedge d \neq 1000)} T1_C$ gelöscht werden. Das Entladen in trivialen atomaren Zonen kann also durch die Ausführung einer einzigen `DELETE`-Anweisung vorgenommen werden. Das folgende, vereinfacht dargestellte Statement findet und löscht die Sätze der atomaren Zone im Beispiel, wobei alle KWTs eingehender RCCs (hier KWT_1 und KWT_2) parallel berücksichtigt werden.

```
delete from T1_C
where (b in (select KW from KWT_1) or
       d in (select KW from KWT_2))
  and (b not in (select distinct R_C.a from R_C) and
       d not in (select distinct S_C.a from S_C))
```

5.2 Entladen in nicht-trivialen atomaren Zonen

Wir beschäftigen uns nun detailliert mit den Problemen, die im Beispiel 3.2 aufgezeigt wurden. Dort wurde bereits festgestellt, dass es notwendig ist, interne und externe Abhängigkeiten zu unterscheiden. Anhand des in Abbildung 14 gezeigten homogenen Zyklus wird nun erklärt, wie diese Abhängigkeiten aufgelöst werden können.

Von besonderer Bedeutung sind die Werte der Tabellenspalten, die Teil des homogenen Zyklus sind. Wir nennen diese Werte *Zykluswerte*. Die entscheidende Idee, um die Abhängigkeiten innerhalb der atomaren Zone schnell aufzulösen, besteht darin, diejenigen Zykluswerte zu bestimmen, für deren Sätze es keinerlei externe Abhängigkeiten mehr gibt. In Abbildung 14 gilt dies nur für den Wert 1000, da die Werte 2000 und 3000 durch $R_C.a = \text{'}x\text{'}$ und $S_C.a = \text{'}j\text{'}$ extern referenziert werden. Löschen wir nun alle Sätze mit diesem Wert aus allen Cache-Tabellen der Zone (*globales Löschen*), so verbleiben in der atomaren Zone nur noch Sätze, die entweder gar keine Abhängigkeiten mehr haben oder die aufgrund bestehender Abhängigkeiten nicht gelöscht werden dürfen. Die verbleibenden, noch löschbaren Sätze lassen sich nun durch ein intern (nur innerhalb der atomaren Zone) ausgeführtes, vorwärtsgerichtetes Entladen ermitteln (*internes Löschen*). Im Beispiel werden dabei die Sätze mit dem Wert 3000 aus $T1_C$ und $T2_C$ gelöscht; der Satz in $T3_C$ muss jedoch, aufgrund seiner externen Abhängigkeit, erhalten bleiben.

Nachfolgend betrachten wir die einzelnen Schritte dieser Vorgehensweise im Detail. Sie sind für jeden externen RCC, der zu löschende Kontrollwerte übermittelt, durchzuführen, um alle Sätze der Zone zu entladen[5].

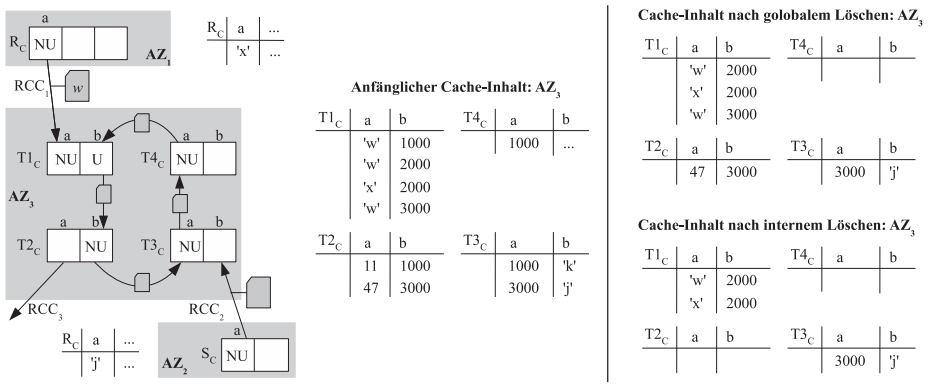

Abbildung 14: Löschen innerhalb nicht-trivialer Zonen

Globales Löschen. Wir betrachten nochmals Abbildung 14 und nehmen an, dass der Wert *w* wie angedeutet entladen werden soll. Um die Zykluswerte zu ermitteln, die global im Zyklus löschbar sind, genügt es, einen Verbund zwischen denjenigen Tabellen auszuführen, die einen externen eingehenden RCC aufweisen. Im Beispiel sind das die Tabellen

[5] Es ist möglich, den Algorithmus so zu konzipieren, dass er alle übermittelten Werte gleichzeitig betrachtet. Anhand dieser komplexen Variante lassen sich die Konzepte jedoch nicht gut erklären.

$T1_C$ und $T3_C$. Die folgende Anfrage bestimmt die zu löschenden Zykluswerte:

```
select T1_C.b from T1_C, T3_C
where  T1_C.b = T3_C.a
and    T1_C.a IN (select KW from KWT_1)
and    T1_C.a NOT IN (select distinct R_C.a from R_C)
and    T3_C.b NOT IN (select distinct S_C.a from S_C)
```

Es genügt also die Abhängigkeitsprüfung über die Quellspalten der externen RCCs vorzunehmen. Interne Abhängigkeiten (interne RCCs) müssen nicht überprüft werden, wodurch die Tatsache, dass in allen am Zyklus beteiligten Spalten die selben Werte miteinander verbunden werden, bestmöglich ausgenutzt wird. Im Beispiel wird auf diese Weise der Wert 1000 aus allen Tabellen innerhalb der atomaren Zone entfernt. Während jeder Löschung ist darauf zu achten, ob hierbei ein Kontrollwert vollständig entfernt wird, welcher somit in die entsprechende Kontrollwerttabelle ausgehender RCCs einzustellen ist. Dies ist im Beispiel für den Wert 11 der Fall, welcher vollständig aus $T2_C.a$ entfernt wurde und somit über die KWT von RCC_3 weitergeleitet wird.

Internes Löschen. Das interne Löschen kann nun auf die gleiche Weise vorgenommen werden wie das Löschen in trivialen atomaren Zonen. Für alle eingehenden RCCs (externe sowie interne) wird beginnend in der Starttabelle überprüft, ob es noch zu löschende Sätze gibt. Ist dies der Fall, wie im Beispiel aus Abbildung 14, so werden diese gelöscht, wobei die dabei vollständig gelöschten Kontrollwerte über **alle** ausgehenden RCCs (also externe wie interne) weitergeleitet werden. Der Löschvorgang wird innerhalb der atomaren Zone so lange fortgesetzt, bis keine interne KWT mehr zu verarbeitende Werte aufweist. Im Beispiel stoppt die Bearbeitung also in Tabelle $T3_C$, da der Wert 3000 aufgrund der externen Abhängigkeit nicht gelöscht werden kann.

Hätte es vor Beginn des internen Löschens einen Satz mit dem Wert 3000 in $T4_C$ gegeben, so wäre das interne Laden bereits sofort nach der ersten Überprüfung in der Starttabelle beendet. Weitere Prüfungen sind überflüssig, da jeder Zykluswert durch eine externe Abhängigkeit motiviert ist.

5.3 Messergebnisse

Abbildung 15 zeigt die Zeit, die benötigt wurde, um die beim Laden generierten Einheiten jeweils zu entladen. In allen Strukturen läuft der Entladeprozess dabei enorm schnell ab (meist unter 200 ms). Der anfänglich vergleichsweise hohe Aufwand (bei 100 Sätzen) entsteht durch die erstmalige Optimierung vorbereiteter Anfragen (Prepared Statements) durch die Datenbank. Dieser Aufwand ist auch in allen vorangegangenen Messungen enthalten, wo er aber aufgrund der hohen Aufwände in den Diagrammen nicht zu sehen ist.

Erfreulich ist vor allem das Ergebnis beim Löschen innerhalb homogener Zyklen. Das Löschen lief sogar noch etwas schneller ab als in vergleichbar langen Ketten. Denn für das globale Löschen, welches in unseren Tests die meisten Sätze löscht, wurden weniger Anfragen benötigt als beim Löschen über mehrere atomare Zonen hinweg. Wir konnten somit zeigen, dass ein selektives Entladen in Zyklen effizient durchführbar ist.

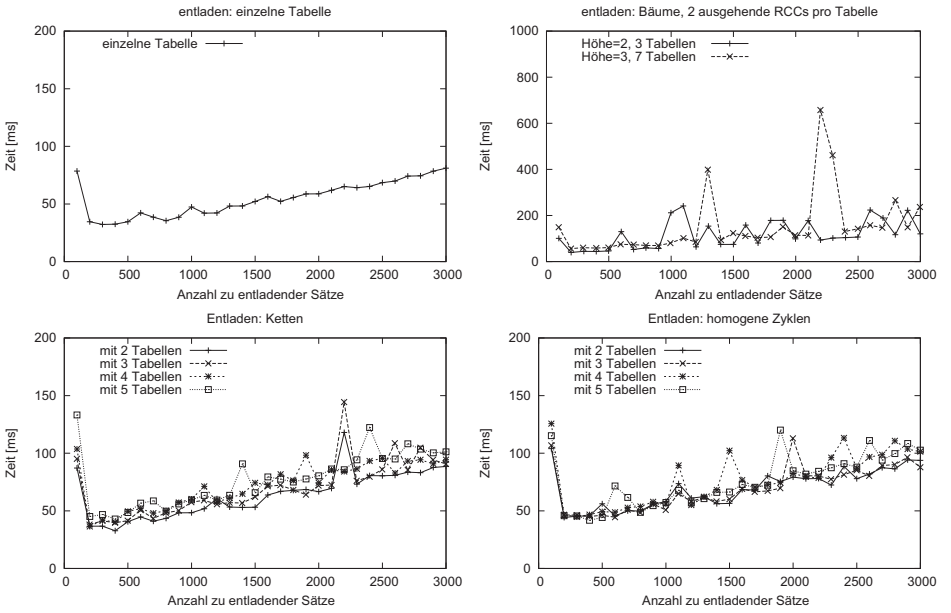

Abbildung 15: Entladen von Cache Units

Überraschend ist das Ergebnis beim Löschen in Bäumen. Beim Baum der Höhe drei, der sieben Tabellen und somit auch sieben atomare Zonen umfasst, steigt der Aufwand zweimal vergleichsweise stark an. In allen Messdurchläufen bei jeweils unterschiedlichen Daten gleichermaßen. Da für die Messungen ein so genannter zirkulärer Log-Schreibmodus verwendet wurde, mussten aufgrund der geringen Größe der Log-Dateien (von dreimal 4096 kB, Seitengröße 4 kB) zwischenzeitlich Seiten aus dem DB-Puffer verdrängt werden, um in den Log-Dateien vorangegangene Einträge überschreiben zu können. Dies wurde durch eine nachträglich vorgenommene Messung mit einem zehnmal größeren Log bestätigt und erklärt somit die zyklisch auftretenden Spitzen. Trotzdem bleibt auch in diesem Fall der Aufwand, um die Cache Units zu entladen, sehr gering.

5.4 Verdrängungsstrategie

Als Verdrängungsstrategie wird die allgemein bekannte LRU-k-Strategie (least recently used) eingesetzt. Zur Umsetzung wird jeweils der Zugriffszeitpunkt auf eine Cache Unit in der Kontrolltabelle für den entsprechenden Füllwert vermerkt (vgl. [BHM06]).

Die Entscheidung, wann das Entladen von Cache Units notwendig ist, wird anhand zweier Füllstandswerte getroffen. Der erste definiert die Maximalanzahl an Cache Units, die in eine Füllspalte eingelagert werden dürfen. Der zweite Wert definiert auf die gleiche Weise, bis zu welchem Füllstand entladen werden soll, wenn das Maximum erreicht ist. Der aktuelle Füllstand wird als Verhältnis zwischen der aktuellen Anzahl geladener

Cache Units und der Anzahl der Kandidatenwerte (der Anzahl aller ladbaren Cache Units) ermittelt. Auf diese Weise ist die Verdrängung von Cache Units pro Füllspalte einstellbar, was eine höhere Adaptivität gewährleistet als die Definition von Schranken für die ganze Cache Group. Es ist jedoch nur sehr schwer überprüfbar, ob diese Strategie zum Verdrängen der Cache Units ähnlich gut geeignet ist wie für den Einsatz im Datenbankpuffer. Der größte Unterschied ist, dass beim CbDBC ganze Prädikatsextensionen verdrängt werden und nicht einzelne Seiten. Auch das Ziel des Löschens ist unterschiedlich. Beim CbDBC steht nicht das Freigeben von Speicher im Vordergrund. Wir können davon ausgehen, das stets genug Speicher zur Verfügung gestellt werden kann, um alle möglichen Cache Units vorzuhalten. Es ist vielmehr wichtig, die Zahl der vorgehaltenen Cache Units einschränken zu können, um den Aufwand, z. B. für die Synchronisation von Replikaten möglichst gering zu halten und eine hohe Lokalität zu erreichen.

5.5 Zusammenfassung

Die vorgestellte Lösung ermöglicht es, einzelne Cache Units selektiv und effizient zu entladen. Sie ermöglicht jedoch nicht automatisch auch die Verarbeitung heterogener Zyklen, wie dies beim indirekten und vorbereiteten Laden der Fall ist. Das Entladen in homogenen Zyklen konnte so weit verbessert werden, dass nur noch ein sehr geringer Mehraufwand im Vergleich zu anderen Strukturen zu erwarten ist. Darüber hinaus beschränken die aufgezeigten Optimierungen den Aufwand beim Löschen in einfacheren homogenen Zyklen deutlich. Die Parameter der Verdrängungsstrategie lassen sich während der Laufzeit für jede Füllspalte separat anpassen. Hierdurch ist das Gesamtsystem prinzipiell in der Lage, mehr oder weniger Cache Units auf einer Cache-Instanz zuzulassen. Einzelne Cache Units können über die Kandidatenmenge von Füllspalten auch komplett ausgeschlossen werden.

6 Fazit

Das CbDBC versucht in seiner Konzeption, deklarative Anfragen auf einem Datenbank-Cache zu unterstützen. Die dazu eingeführten Constraints und die somit entstehenden Abhängigkeitsketten stellen jedoch eine Herausforderung für die Umsetzung der benötigten Funktionalitäten dar (alle Bedingungen müssen jederzeit erfüllt sein). Um weiterhin alle Vorteile einer relationalen Bearbeitung von Anfragen zu nutzen, ist es jedoch wichtig, die neuen Strukturen und Bedingungen auf relationaler Ebene durch leistungsstarke und einfache Methoden handhaben zu können. Mit den in diesem Aufsatz vorgestellten Konzepten und Algorithmen gelingt es, das selektive Laden von Cache Units deutlich zu verbessern und gleichzeitig das Entladen ebenfalls selektiv zu gestalten. Mit den neu eingeführten Definitionen Cache Unit und Cache-Unit-Differenz gelingt es nun, die Größenverhältnisse zu ladender bzw. zu entladender Einheiten genau zu beschreiben. Die gezeigten Verfahren erhöhen die Anpassungsfähigkeit des Cache-Systems, da es möglich ist, das beste Verfahren aus einer Menge von Alternativen auszuwählen. Darüber hinaus lässt sich das Füll- und Entladeverhalten mithilfe von Kandidatenwerten und verschiedener Füllgrade für die Füllspalten sehr gezielt beeinflussen.

7 Ausblick

Die vorgestellten Untersuchungen belegen zwar den Nutzen der neuen Verfahren, analysieren aber deren Verhalten immer noch nicht genau genug, um die Größenordnungen entscheidender Faktoren zu bestimmen. Ziel wäre es, z. B. anhand statistisch bzw. direkt erfassbarer Daten (wie die Größe der zu ladenden Cache Unit) entscheiden zu können, welche Ladestrategie zu wählen ist. Hierzu müsste vor allem der Einfluss verschiedener Latenzen noch genauer untersucht werden. Ebenso wurde nicht geklärt, inwieweit die Verbesserungen andere Funktionen beeinflussen. In der Zukunft planen wir daher, weitere Messverfahren zu entwickeln, die uns helfen, wichtige Entscheidungsrößen zu definieren.

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.

[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.

[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.

[BK07] Andreas Bühmann und Joachim Klein. Examining the Performance of a Constraint-Based Database Cache. In *IDEAS*, Seiten 290–295. IEEE Computer Society, 2007.

[CLRS82] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*, Kapitel 22.4, Seiten 549–552. The MIT Press, 2. Auflage, 1982.

[GL01] Jonathan Goldstein und Per-Åke Larson. Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD*, Seiten 331–342, 2001.

[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.

[LGGZ04] Per-Åke Larson, Jonathan Goldstein, Hongfei Guo und Jingren Zhou. MTCache: Mid-Tier Database Caching for SQL Server. *Data Engineering Bulletin*, 27(2):35–40, Juni 2004.

[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.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv und Divesh Srivastava. Answering Queries Using Views. In *PODS*, Seiten 95–104, 1995.

[Mer05] Christian Merker. Konzeption und Realisierung eines Constraint-basierten Datenbank-Cache. Diplomarbeit, TU Kaiserslautern, 2005.

[Ora08] Oracle Corporation. Internet Application Server Documentation Library, 2008.

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