Technische Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr.-Ing. Dr. h. c. Theo Härder
Lehrgebiet Datenverwaltungssysteme

Studienarbeit

Cache-Group-Optimierung zur Effizienzsteigerung von Datenbank-Caches

Betreuer: Dipl.-Inf. Andreas Bühmann

Bearbeiter: Wolfgang Scholl
Matrikel-Nr.: 345 905

Kaiserslautern, im April 2006

Inhaltsübersicht

Inhaltsübersicht 3
1 Zusammenfassung 5
2 Einführung 6
2.1 Caching als Optimierungsmethode für Datenbankabfragen 6
2.2 „Intelligentes" Caching mit Cache Groups 7
2.2.1 Grundlagen des constraintbasierten Datenbank-Cachings 7
2.2.2 Herausforderung 11
3 Regeln für die Cache-Group-Optimierung 14
3.1 Unique-Spalten 14
3.2 Induzierte Bereichsvollständigkeit 17
3.3 Gegenläufige RRCCs 20
3.4 RRCCs zu Geschwistern 24
4 Systematischer Optimierungsablauf 27
4.1 Notwendigkeit einer Regelanwendungs-Systematik 27
4.2 Grundlagen des Durchlaufkonzepts 27
4.3 Baumstruktur 28
4.4 Zyklenbehandlung 28
4.5 Weitere Verfahrensoptimierung 29
5 Grenzen des Optimierungskonzeptes 32
5.1 Unerkannte RRCCs 32
5.1.1 Umarmende RCCs 32
5.1.2 Problematik 33
5.2 Umgang mit RCC-Zyklen 33
5.2.1 Zyklen 33
5.2.2 Homogene Zyklen 34
5.2.3 Heterogene Zyklen 35
5.2.4 Stark verschachtelte Zyklen 35
6 Präsentation des Analysetools 38
6.1 Übersicht über die Klassen 38
6.2 Implementierung der Regeln und des Durchlaufprinzips 39
6.3 XML-Input- und -Output-Funktionalität 40
7 Ausblick 42
Literaturverzeichnis 43

1 Zusammenfassung

Caching ist ein vielseitig einsetzbares Konzept der Informatik zur Effizienzsteigerung unterschiedlicher Systeme. Hierbei sind relationale Datenbanken einer der herausforderndsten Anwendungsbereiche. Ein vielversprechender Ansatz auf diesem Gebiet ist das constraintbasierte Caching mit Cache Groups, bei dem man sich die auf einer Auswahl von zu cachenden Tabellen definierten Constraints zu Nutze macht, um mehr Unabhängigkeit des Caches von der Backend-Datenbank zu gewinnen.

Je mehr Constraints vorhanden sind, desto größer ist (trotz steigender Komplexität) die Fähigkeit des Caches, Anfragen eigenständig beantworten zu können. In früheren Publikationen wurde festgestellt, dass sich aus einer gegebenen Cache Group oft zusätzliche Constraints ableiten lassen, die allein aus dem Zusammenhang heraus gültig sind. Somit entstand der Bedarf für einen Algorithmus, der diese optimierenden Constraints systematisch aufdecken kann.

In dieser Arbeit werden vier Optimierungsregeln präsentiert, mit deren Hilfe solche zusätzlichen Constraints gefunden werden können. Darüber hinaus wird ein spezielles Analyseverfahren vorgestellt, welches unter Einbeziehung dieser Regeln eine effiziente Cache-Group-Optimierung ermöglicht. Im Rahmen der Erläuterungen wird außerdem auf Probleme von RCC-Zyklen in Cache Groups eingegangen; Dazu wird ein Verfahren vorgestellt, mit dem Zyklen systematisch aufgespürt und analysiert werden können, um einen Cache so vor möglichem Schaden zu schützen.

2 Einführung

Im Zeitalter des Internet sind Web-Anwendungen populär geworden, die Software-Anwendern unabhängig von ihrem Aufenthaltsort Zugriff auf entfernte Applikationen erlauben. Insbesondere die Nutzung von Datenbanken gewinnt im modernen Geschäftsleben immer mehr an Bedeutung. Dabei existiert eine breite Palette an Datenbank-Anwendungen, die von der einfachen E-Mail-Datenbank bis hin zum komplexen Kundenverwaltungssystem reicht.

2.1 Caching als Optimierungsmethode für Datenbankabfragen

Die technische Herausforderung bei Datenbanknutzung über das Internet ist die Bearbeitung von sehr vielen Datenbankanfragen, die gleichzeitig an den Datenbank-Server gestellt werden. Bei paralleler Abarbeitung verlängert sich die Bearbeitungszeit für jede einzelne Anfrage, bei alternativem sequenziellem Vorgehen entstehen lange Warteschlangen. Folglich wächst bei starker Anfragendichte die Dauer von der Anfragestellung bis zu ihrer Beantwortung stark an, was die entfernte Anwendung extrem ausbremst. Ziel der wissenschaftlichen Bemühungen auf diesem Feld ist es deshalb, die Belastung des zentralen Servers auf ein absolutes Minimum zu reduzieren, einerseits um Ressourcen (wie Rechenzeit des Servers und Bandbreite) zu schonen, andererseits um die Antwortgeschwindigkeit von entfernten Systemen idealerweise der der Nutzung vor Ort anzunähern.

Einen effektiven Lösungsweg stellt das Datenbank-Caching dar. Das zu Grunde liegende Prinzip lässt sich mit dem lokalen Cache eines Web-Browsers vergleichen: Dieser soll Daten, die schon einmal vom entfernten Web-Server abgerufen wurden, zwischenspeichern, um sie bei erneutem Abruf direkt und ohne Serveranfrage zur Verfügung stellen zu können. Dadurch kann die Ladezeit von Webseiten optimiert und das übertragene Datenvolumen bei wiederholten Anfragen deutlich reduziert werden. Beim World Wide Web ist diese Methode relativ einfach umzusetzen, da es sich bei den zu cachenden Daten um statische Objekte wie z.B. HTML-Seiten oder JPEG-Bilder handelt.

In der Welt von relationalen Datenbanken gestaltet sich dies ungleich schwieriger: Es handelt sich bei den angeforderten Informationen nicht um statische Objekte, sondern um via SQL-Anfragen individuell angeforderte Teilausschnitte der gesamten Datenbank. Daher ist das Caching-Verfahren wesentlich komplexer, denn als kleinste Einheit müssen einzelne Datensätze einzelner Tabellen gecacht werden. Im Idealfall kann der Cache Anfragen allein auf Basis seiner zwischengespeicherten Datensätze beantworten, ohne Rückfrage an die zentrale Haupt-Datenbank, das sogenannte *Backend*. Selbstverständlich müssen dabei hundertprozentige Korrektheit, Vollständigkeit und Aktualität des zurückgelieferten Abfrageergebnisses gewährleistet sein [HäBü04a].

Forschungsbedarf besteht nun dahingehend, möglichst effiziente Caching-Algorithmen zu entwickeln, die gute Prognosen über zukünftig benötigte Datensätze erstellen und solche Datensätze präventiv im Cache zwischenspeichern. Weiterhin werden auch Verfahren benö-

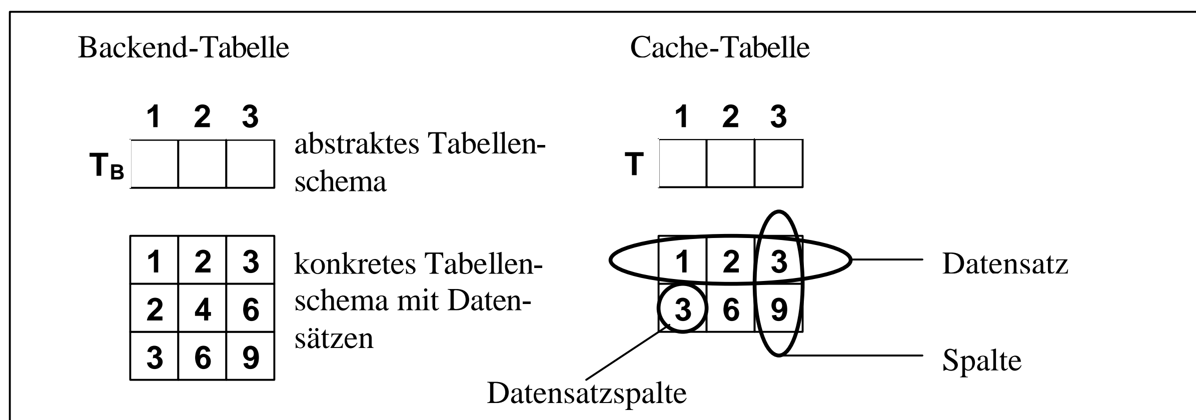

Abbildung 1: Begrifflichkeiten und Darstellung von Cache- und Backend-Tabellen

tigt, mit denen ein Cache die Korrektheit und Vollständigkeit eines von ihm ohne Rückgriff auf die Backend-Datenbank erstellten Anfrageergebnisses gewährleisten kann [Bühm05].

2.2 „Intelligentes“ Caching mit Cache Groups

Ein vielversprechender Ansatz ist der des constraintbasierten Datenbank-Cachings. Die Grundlagen dieses Konzepts werden im Folgenden kurz dargelegt (Kap. 2.2.1), woraus wiederum der eigentliche Zielgedanke dieser Arbeit abgeleitet wird (Kap. 2.2.2).

2.2.1 Grundlagen des constraintbasierten Datenbank-Cachings

Die relationale Backend-Datenbank eines Datenverwaltungssystems beinhaltet verschiedene Tabellen mit mehreren Spalten sowie sämtliche in diesen Tabellen befindliche Datensätze. Der Cache, der nun zwischen Client und Backend geschaltet wird, ist ebenfalls eine relationale Datenbank – allerdings spiegelt er nur eine kleine Auswahl der Tabellen des Backend-Servers wieder; Diese wiederum enthalten nur von ausgewählten Datensätzen der Backend-Datenbank eine Kopie.

In Abbildung 1 werden die in dieser Arbeit verwendeten Begrifflichkeiten und die Art der graphischen Darstellung von Tabellen veranschaulicht: Einerseits wird ein abstraktes Tabellenschema (oben) verwendet, um mehrere Tabellen übersichtlich in Beziehung zu setzten, andererseits ist es manchmal hilfreich, ein ausführliches Beispiel mit Hilfe des konkreten Tabellenschemas (unten) zu betrachten, in dem die aktuell in einer Tabelle enthaltenen Datensätze in Form von Zeilen dargestellt werden. Eine Datensatzspalte ist ein einzelnes Spaltenobjekt eines bestimmten Datensatzes, dieses muss von seiner Wertausprägung unterschieden werden. Eine Datensatzspalte existiert nur ein einziges Mal, allein ihr Wert kann auch in anderen Datensatzspalten vorkommen. Der Wert wird im Folgenden durch die Value-Funktion v dargestellt, beispielsweise würde der Wert von T.1, also der ersten Spalte von Tabelle T, als v(T.1) geschrieben.

Nun stellt sich die Frage, welche Tabellen und welche Datensätze im Cache vorgehalten werden sollen. Nach gegenwärtigem Stand der Forschung müssen die zu cachenden Tabellen vom

Administrator des Caches vorgegeben werden. Allerdings ist es denkbar, dass in Zukunft die Cache-Verwaltungs-Software, der sogenannte *Cache-Manager*, diese Tabellenauswahl automatisiert trifft und gegebenenfalls zu einem späteren Zeitpunkt weiter optimiert.

Die Tabellenauswahl für einen Cache wird durch seine *Cache Group* dargestellt. Hierbei handelt es sich um eine Struktur aus Tabellen mit ihren Spalten, die eine Teilmenge der Tabellenstruktur der Backend-Datenbank darstellt. Sinnvollerweise sollten die Tabellen, auf die am häufigsten zugegriffen wird, in der Cache Group abgebildet sein. (Es ist ebenfalls möglich, mehrere räumlich verteilte Caches für ein und dieselbe Backend-Datenbank einzurichten, dies wird als Cache-Föderation bezeichnet. In diesem Fall wird jeder einzelne dieser Caches durch seine eigene individuell gestaltbare Cache Group bestimmt. [BüHM06]) Das Vorhandensein einer Tabelle in der Cache Group bedeutet jedoch nicht, dass sämtliche ihrer Datensätze auch im Cache zwischengespeichert werden – die Cache-Tabelle muss jeweils nur eine Teilmenge an vollständigen Sätzen enthalten.

Gleichzeitig enthält eine Cache Group auch Constraints, mit deren Hilfe es später möglich ist, für eine Anfrage zu entscheiden, ob sie allein mit den Daten aus dem Cache beantwortet werden kann. Dabei können Constraints gezielt eingesetzt werden, um für bestimmte Anfragetypen das Caching zu optimieren. Diese Arbeit konzentriert sich hierbei auf zwei wichtige Arten von Constraints: *Bereichsvollständigkeits-Constraints* und *referentielle Cache-Constraints*.

Wert- und Bereichsvollständigkeit

Wertvollständigkeit oder *value completeness* ist ein Attribut eines einzelnen Wertes in einer Tabellenspalte im Cache. Der Wert v(a) einer Datensatzspalte a ist wertvollständig in der (Cache-) Spalte T.s, wenn alle Datensätze der Backend-Tabelle mit $v(T_B.s) = v(a)$ in der Cache-Tabelle T vorhanden sind. Wertvollständigkeit eines Wertes einer Tabellenspalte ist wichtig, wenn eine Anfrage an den Cache sich auf diesen Wert bezieht. Denn wenn nicht alle Datensätze mit diesem Wert in der Cachetabellenspalte vorhanden sind, kann die Vollständigkeit des Ergebnisses dieser Anfrage nicht mehr garantiert werden, folglich muss trotzdem wieder die Backend-Datenbank in die Anfragebearbeitung mit einbezogen werden. Daher ist es hilfreich, wenn in häufig referenzierten Tabellenspalten *alle* aktuell vorkommenden Werte wertvollständig sind. In diesem Fall nennen wir eine solche Cache-Spalte *bereichsvollständig* oder *domain complete* (dc) bzw. alternativ *spaltenvollständig* oder *column complete* [HäBü05].

Prinzipiell gilt für Constraints, dass sie eingehalten werden müssen. Hierfür ist der Cache-Manager zuständig. Im Falle der Bereichsvollständigkeit müssen also entsprechend Datensätze aus dem Backend in den Cache nachladen werden, falls auf anderem Wege ein Datensatz mit einem neuen, noch nicht wertvollständigen Wert in eine bereichsvollständige Cache-Spalte gelangt ist.

T_B

1	2
1	3
1	4
2	5
2	6

T

1	2
dc	
1	3
1	4

SELECT * FROM T WHERE T.1 = 1 → OK

SELECT * FROM T WHERE T.1 = 2 → R → NOK

SELECT * FROM T WHERE T.1 = 3 → R → OK

Abbildung 2: Anfragenoptimierung durch Bereichsvollständigkeit

Mit der Information, dass eine Cache-Spalte bereichsvollständig ist, lässt sich oft entscheiden, ob eine Anfrage auf diese Spalte allein aus dem Cache beantwortet werden kann. Teilweise ist noch eine einfache Überprüfung mit Hilfe des Backends nötig um diese Frage zu beantworten; Diese wird durch die gegebene Wertvollständigkeit aber auf ein Mindestmaß an Komplexität reduziert [HäBü04b].

Dieser Zusammenhang wird durch das Beispiel in Abbildung 2 veranschaulicht: Hier ist Spalte T.1 von Cache-Tabelle T domain complete. Offensichtlich wird dieser Constraint eingehalten, denn alle Datensätze mit dem Wert „1“ in Spalte 1 sind in der Cache-Tabelle vorhanden. Da sich ansonsten keine weiteren Datensätze mit anderen Werten (z.B. „2“) in Spalte 1 befinden, ist T.1 bereichsvollständig.

Wird nun die erste der drei angegebenen SQL-Anfragen an den Cache gesendet, so weiß dieser ohne Rückfrage (R) an die Backend-Datenbank, dass er diese Anfrage völlig eigenständig beantworten kann (OK): Es sind nur Datensätze mit „1“ in Spalte T.1 gefragt. Da solche Datensätze vorhanden sind und T.1 zusätzlich als bereichsvollständig deklariert ist, ist klar, dass sich alle benötigten Datensätze im Cache befinden müssen.

Die zweite Beispiel-Anfrage ist komplizierter. Es werden Datensätze mit „2“ in Spalte T.1 gesucht, davon ist gegenwärtig kein einziger im Cache vorhanden. Es wäre aber denkbar, dass solche Datensätze im Backend existieren, daher muss dies kurz überprüft werden. Daran, dass diese Überprüfung positiv ausfällt, erkennt der Cache, dass er diese Anfrage nicht selbständig beantworten kann.

Anders sieht es bei der letzten SQL-Anfrage aus: Die gerade beschriebene einfache Rückfrage an den Backend-Server ergibt, dass es dort gar keine Datensätze mit dem Wert „3“ in Spalte T_B.1 gibt. Somit ist „3“ effektiv wertvollständig in Cachespalte T.1, auch wenn dort kein einziger derartiger Datensatz vorhanden ist. Der Cache kann die Anfrage nun also unabhängig von der Backend-Datenbank beantworten, in diesem Fall ist die Antwort eine leere Menge.

Referentielle Cache-Constraints

Die kurz *RCC* genannten Constraints verknüpfen zwei Spalten aus zwei verschiedenen Tabellen wertmäßig, was bei Anfragen mit Equijoins hilfreich ist. Ein RCC von Cache-Spalte T.1

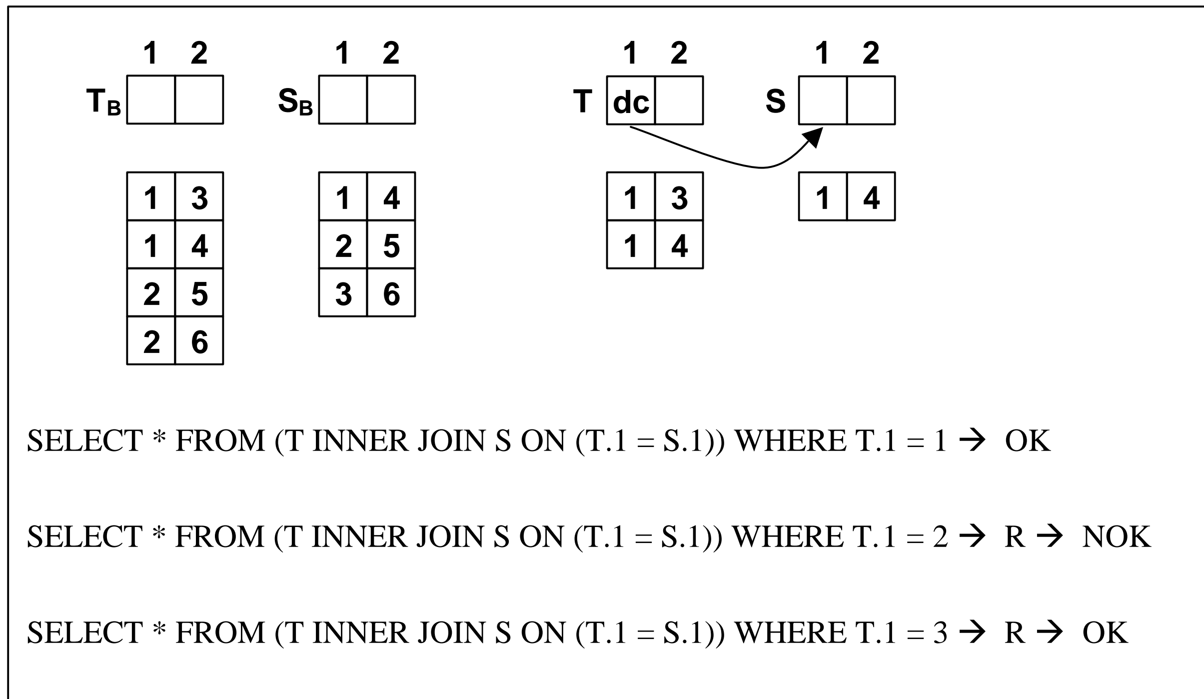

Abbildung 3: Anfragenoptimierung durch RCCs

zu Cache-Spalte S.1 ist genau dann erfüllt, wenn sämtliche aktuell in T.1 enthaltenen Werte in S.1 wertvollständig sind. Somit stellt ein RCC effektiv eine Art Nachladebefehl dar: Sobald im Cache in Tabelle T ein neuer Datensatz mit einem neuen Wert x in Spalte T.1 auftaucht, müssen alle Sätze aus S_B mit diesem Wert in Spalte S_B.1 auch in Tabelle S nachgeladen werden (falls sich sie sich nicht bereits dort befinden), damit x in S.1 wertvollständig ist und so der RCC nicht verletzt wird. Im Fall eines Equijoins zwischen zwei durch einen RCC verbundenen Spalten hat der Cache nun oft bereits vorgesorgt: Durch den RCC kann einfach entschieden werden, ob ein Nachladen aus der Backend-Datenbank notwendig ist oder nicht [Baye04].

Dies wird in Abbildung 3 verdeutlicht: Wie beim vorigen Beispiel in Abbildung 2 sind auch hier verschiedene Beispiel-Anfragen gegeben, die zu unterschiedlichen Ergebnissen führen.

Die erste SQL-Anfrage kann direkt aus dem Cache beantwortet werden, denn sie zielt nur auf Datensätze mit „1" in Spalte T.1 ab. Hiervon sind dank der vorausgesetzten Bereichsvollständigkeit alle im Cache vorhanden; Wäre T.1 nicht dc, so müssten hier gegebenenfalls Datensätze von T_B nach T nachgeladen werden. Der Vorteil des RCC liegt darin, dass durch ihn bereits der Wert „1" bereichsvollständig in S.1 ist, so dass hier keine weitere Prüfung notwendig ist. Es macht offensichtlich Sinn, Spalten, die häufig gejoint werden, mit einem RCC zu verbinden.

Beim nächsten Beispiel fehlen wieder sämtliche Datensätze mit „2" in T.1, wodurch die Anfrage nicht vom Cache erfüllt werden kann. Um verlängerte Antwortzeiten zu reduzieren, wird die Anfrage nun direkt vom Backend-Server beantwortet. Für zukünftige Anfragen können die gerade vermissten Datensätze mit „2" in T.1 in T nachgeladen werden. Hierdurch würde „2" zur Aufrechterhaltung des RCCs automatisch auch in S.1 wertvollständig gemacht,

womit der Cache bereits alle nötigen Daten für eine erneute Beantwortung der gleichen oder einer ähnlichen Anfrage erhalten hat.

Im dritten Beispiel findet der Cache per simpler Nachfrage an den Backend-Server heraus, dass es in T_B gar keine Datensätze mit „3“ in T_B.1 gibt. Daraus folgt, dass ein Join eine leere Menge wäre, der RCC muss hier also gar nicht in die Analyse einbezogen werden. Das Ergebnis kann nun auch der Cache zurückliefern.

Aufgabe des Cache-Managers ist es nun wie bereits angedeutet, die durch die Cache Group vorgegebenen Constraints aufrechtzuerhalten, indem er bei Bedarf fehlende Datensätze aus dem Backend in den Cache nachlädt. Ebenso muss er bei der Verdrängung von Daten aus dem Cache für Constraint-Einhaltung sorgen.

Lohn des Aufwandes ist, dass einfach geprüft werden kann, ob eine bestimmte Anfrage allein durch den Cache beantwortet werden kann. Diese Prüfung wird als *Sondierung* oder *probing* bezeichnet. Liefert das individuelle Probing für eine Anfrage ein positives Ergebnis, dann unterscheidet sich die Ausführung der Anfrage im Cache nicht von der direkten Ausführung im Backend. Dies wird dadurch gewährleistet, dass alle von für die Abfrage benötigten Datensätze im Cache vorgehalten werden – genau dies wird durch die Sondierung vor der Ausführung der Anfrage überprüft [Bühm04]. Diese Situation wurde mit der jeweils ersten Beispiel-SQL-Anfrage der beiden letzten Abbildungen gezeigt. Sollte der Cache-Manager feststellen, dass eventuell nicht sämtliche notwendigen Werte im Cache vorhanden sein könnten, so muss die Backend-Datenbank in die Anfrageausführung mit einbezogen werden. Standardmäßig würde dann die Backend-Datenbank die komplette Anfrage beantworten. Alternativ wäre es möglich, dass mit einer sehr einfachen Rückfrage des Caches an das Backend herausgefunden werden kann, ob die gecachten Datensätze nicht doch ausreichen, so wie in der jeweils letzten Anfrage der zwei vorigen Abbildungen. Trifft dies nicht zu, so würde die Anfrage nach gängiger Methode komplett an den Backend-Server weitergereicht werden. Allerdings wäre es auch denkbar, dass lösbare Teile einer Anfrage im Cache ausgeführt werden, und nur der Rest der Backend-Datenbank überlassen wird. Die zweite vorstellbare Alternative ist, dass der Cache sämtliche für die Abfrage benötigten Datensätze vom Backend nachlädt, und diese dann doch selbst beantworten kann. In jedem Fall aber sollten entsprechende Algorithmen im Cache-Manager vorhanden sein, die aus „Fehlern“ bzw. unbeantwortbaren Anfragen lernen und den Cacheinhalt besser auf künftige Anfragen vorbereiten.

2.2.2 Herausforderung

Um optimales Caching betreiben zu können, muss die Struktur der überwiegenden Datenbank-Nutzung bekannt sein. Das beinhaltet z.B., welche Spalten am meisten angefragt werden, und welche Spaltenpaare häufig gejoint werden. Dieses erwartete Nutzungsprofil wird durch die Cache Group dargestellt.

Offensichtlich ist die Cache-Group-Konfiguration mit allen vorhandenen Constraints von essentieller Bedeutung für die Effektivität des Cachings. Folglich sollte angestrebt werden, für eine Datenbank eine möglichst optimale Cache Group zu finden. Gegenwärtig muss man sich

hierbei noch auf menschliche Vorgaben verlassen, so muss z.B. die initiale Tabellenauswahl für die Cache Group vom Administrator getroffen werden. Allerdings ist es durchaus vorstellbar, dass intelligente Cache-Manager später in der Lage sind, ein Cache-Group-Schema selbständig auf Basis von Zugriffshäufigkeiten für einzelne Tabellen zu modifizieren.

In dieser Arbeit soll es jedoch allein um die Aufgabe gehen, eine von Hand vorgegebene Cache Group aufzubereiten, bevor sie an den Cache-Manager des Caches übergeben wird. Diese Aufbereitung besteht darin, aus der initial gegebenen Cache Group „das meiste herauszuholen", also weitere gültige Constraints zu finden, die sich aus dem System ableiten lassen.

Hierzu wurde ein Analyseverfahren entwickelt, welches eine Cache-Group-Konfiguration aus Tabellen und vorgegebenen RCCs im Detail unter die Lupe nimmt, und auf folgende drei Arten optimiert:

- Erkennen und Markieren von bereichsvollständigen Spalten (als Einstiegspunkte). Wie im vorigen Abschnitt erläutert wurde, ermöglicht der Bereichsvollständigkeits-Status von Spalten mit Hilfe von Probing eine Aussage über die Beantwortbarkeit einer Anfrage aus dem Cache. Folglich erhöht die flächendeckende Markierung aller bereichsvollständigen Spalten in einer Cache Group die Chance, Anfragen unabhängig vom Backend aus dem Cache beantworten zu können.
- Erkennen und Hinzufügen von zusätzlich gültigen, redundanten Optimierungs-RCCs (später RRCCs genannt). Durch jeden zusätzlich gültigen RRCC unterstützt der Cache weitere Anfrage-Joins, was ebenfalls seine Eigenständigkeit erhöht.
- Erkennen und Markieren von existierenden RCCs als redundant, wenn diese sich aus den übrigen Constraints ableiten lassen. Gemäß dem Konzept der Cache Group muss der Cache-Manager stets für die Aufrechterhaltung sämtlicher Constraints sorgen, dies schließt auch RCCs ein. Werden in der Ursprungstabelle eines RCC Datensätze nachgeladen, so muss geprüft werden, ob der RCC noch gültig ist. Im negativen Fall müssen auch in der Zieltabelle dieses RCC Datensätze nachgeladen werden. Redundante RCCs werden zur Aufrechterhaltung der Constraints aber gar nicht benötigt – hierfür sorgen bereits die nicht-redundanten (N)RCCs. Somit kann der Cache-Manager die redundanten RCCs beim Nachladen von Datensätzen ignorieren. Durch das Erkennen von noch nicht als redundant markierten RCCs als redundant können folglich überflüssiger Aufwand und unnötige Backend-Anfragen vermieden werden.

Zusätzlich zu diesen drei Optimierungsansätzen sollen Zyklen erkannt werden, welche die Sicherheit des Caches durch unkontrolliertes Nachladen gefährden – hierauf wird in Kapitel 5 näher eingegangen.

In Kapitel 3 werden die erarbeiteten Regeln zur Erkennung dieser Optimierungsmöglichkeiten vorgestellt. Kapitel 4 beschäftigt sich mit der Frage, mit welcher Systematik diese Regeln auf eine Cache Group angewendet werden müssen, um das volle Optimierungspotential auszuschöpfen und den Analyseaufwand trotzdem so klein wie möglich zu halten. In Kapitel 5 wird den Grenzen der Cache-Optimierung nachgegangen, woraufhin die beispielhafte Implemen-

tierung der Optimierungsregeln in einem Java-Tool in Kapitel 6 präsentiert wird. Abschließend gibt Kapitel 7 einen Ausblick auf Perspektiven beim Datenbank-Caching.

3 Regeln für die Cache-Group-Optimierung

Eine vom Cache-Administrator vorgegebene Cache-Group-Konfiguration lässt sich in den meisten Fällen überarbeiten, um die Leistung des Cache-Managers zu verbessern. Erstens können die bereichsvollständigen Tabellenspalten identifiziert werden. Zweitens können weitere gültige aber redundante RCCs gefunden werden, die sich aus dem Kontext der übrigen RCCs regeben. Und drittens können initial vorgegebene RCCs als redundant erkannt werden, wenn sie sich aus dem Zusammenhang der anderen RCCs sowieso ergeben.

Wie bereits im vorigen Kapitel erläutert können all diese anfangs „verborgenen“ Zusatzinformationen zur Optimierung der Arbeit des Cache-Managers beitragen. Im Folgenden werden vier Regeln aufgestellt, erklärt und bewiesen, mit deren Hilfe diese impliziten Informationen einer Cache Group aufgespürt werden können.

An dieser Stelle wird der Verständlichkeit halber eine Spezialisierung von RCCs vorgenommen: Ein RCC ist immer entweder ein NRCC oder ein RRCC.

Definition: Ein *NRCC* ist ein *nicht-redundanter referentieller Cache-Constraint*, also ein RCC, der nicht aus einer Cache Group gestrichen werden kann, ohne dass sich deren Constraint-Semantik verändert. Anders ausgedrückt: Würde man einen NRCC aus der Cache Group entfernen, so würden sich im Cache-Manager unter gleichen Voraussetzungen andere Füllzustände einzelner Cache-Tabellen ergeben. NRCCs sind immer initial vorgegebene RCCs, durch Optimierung können keine neuen Exemplare entstehen. NRCCs werden graphisch als durchgehende (nicht-gestrichelte) Pfeile dargestellt.

Definition: Ein *RRCC* ist ein *redundanter referentieller Cache-Constraint*, also ein RCC, der sich zusätzlich zu den bereits vorhandenen RCCs aus dem Kontext der Cache Group mit ihren RCCs ergibt. RRCCs könnten aus einer Cache Group ersatzlos gestrichen werden, ohne dass sich deren Constraint-Semantik verändert – im Cache befänden sich genau dieselben Datensätze wie in dem Szenario mit den RRCCs. Geltende RRCCs können durch Cache-Group-Optimierung einerseits zusätzlich zu den bestehenden RCCs gefunden werden, andererseits können auch ursprüngliche NRCCs zu RRCCs degradiert werden, sofern sie sich doch aus anderen RCCs ableiten lassen. RRCCs werden graphisch als gestrichelte Pfeile dargestellt.

Nun aber zu den vier Optimierungsregeln für Cache Groups:

3.1 Unique-Spalten

Die im Folgenden beschriebene Regel erkannten schon Härder und Bühmann bei der Entwicklung ihres Cache-Group-Konzeptes [HäBü04], sie wird an dieser Stelle nochmals erläutert. Diesen Abschnitt abschließend wird an einem einfachen Beispiel die Wirkungsweise dieser Regel veranschaulicht, sowie ihre allgemeine Gültigkeit bewiesen.

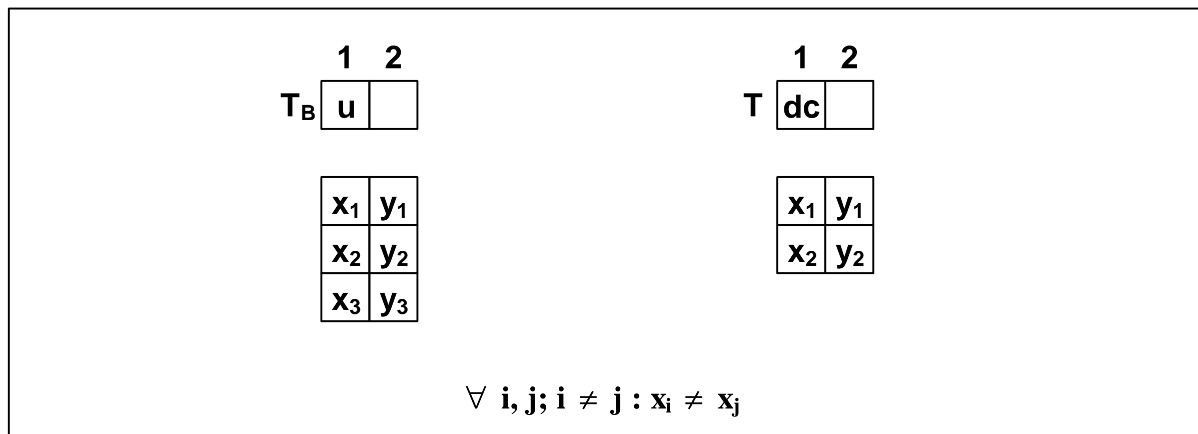

Abbildung 4: Regel 1, Unique-Spalten

Für Spalten, die in der Datenbank als unique gekennzeichnet sind, hat bekanntermaßen jeder Datensatz der Tabelle einen eigenen, einmalig vorkommenden Wert. Betrachtet man sich diese Konstellation genauer, so fällt auf, dass sich aus diesem Umstand ein Constraint für diese Spalte in der Cache Group ableiten lässt:

Wenn jeder Wert in einer Spalte maximal einmal auftritt, so ist er automatisch wertvollständig, sobald er im Cache vorkommt. Daraus lässt sich wiederum schlussfolgern, dass *alle* in dieser Cache-Spalte vorhandenen Werte wertvollständig und die Spalte somit bereichsvollständig ist.

Regel 1: Alle unique-Spalten sind in der Cache Group in jedem Fall bereichsvollständig [Bühm05].

Am Beispiel in Abbildung 4 in wird klar, wieso dies so ist: In der Backend-Datenbank befinde sich die Tabelle T_B mit den Spalten $T_B.1$ und $T_B.2$, wobei $T_B.1$ unique sei. T_B enthalte drei Datensätze, (x_1,y_1), (x_2,y_2), und (x_3,y_3). (Datensätze werden in Abbildungen der Einfachheit halber direkt unter der jeweiligen Tabelle aufgelistet.) Da Spalte $T_B.1$ unique ist, unterscheiden sich die Werte aller Datensätze in Spalte $T_B.1$ (x_1, x_2 und x_3) voneinander.

Im Cache wird T_B als T zwischengespeichert, allerdings befinden sich zur Zeit nur zwei der drei Datensätze aus T_B in T. Da Spalte $T_B.1$ unique ist, ist auch T.1 unique. Daraus lässt sich ableiten, dass jeder vorkommende Wert in T.1 immer wertvollständig ist, da ja schließlich nur ein einziger Datensatz mit diesem Wert in T.1 existiert. Daraus wiederum folgt gemäß Definition, dass T.1 bereichsvollständig ist, was der Aussage von Regel 1 entspricht.

Zum formalen Beweis der aufgestellten Optimierungsregeln wird in Zukunft eine spezielle Notation benutzt, welche sich an die der Mengenlehre anlehnt. Im Folgenden wird diese Notation kurz eingeführt:

Objekttypen

Prinzipiell muss eine Unterscheidung gemacht werden zwischen Datensatzspalten und Werten. Datensatzspalten sind Spalten einzelner Datensätze in Tabellen. Datensatzspalten haben keine Duplikate. Beispielsweise können zwei Personen mit gleichem Namen in einer Adress-

datenbank existieren, allerdings werden sich die zwei Datensätze anhand eines zusätzlichen Schlüsselelements unterscheiden lassen, sie sind also nicht identisch. Dementsprechend sind auch zwei Datensatzspalten, die z.B. den gleichen Namen enthalten, voneinander verschieden.

In unserer Notation gibt es also

- einzelne Datensatzspalten a sowie
- die Menge T.1 der Datensatzspalten aus Spalte 1 in Tabelle T.

Demgegenüber sind Werte Ausprägungen von Datensatzspalten, also etwa ein Name. Im Gegensatz zu Datensatzspalten können mehrere in einer Tabelle befindliche Werte identisch sein. In der Adressdatenbank können z.B. mehrere Personen mit demselben Namenswert existieren. Aus einer einzelnen Datensatzspalte oder einer Menge von Datensatzspalten kann die Wertemenge extrahiert werden, und zwar mit der value-Formel:

In der Notation ist

- v(a) der Wert der Datensatzspalte a, und
- v(T.1) ist die Wertemenge der Datensatzspalten in Spalte 1 von Tabelle T.

Mengenrelationen

Analog zur klassischen Mengenlehre werden die folgenden Mengenrelationen benutzt:

- $\cap$, $\cup$ Schnittmenge und Vereinigungsmenge von zwei Datensatzspalten- oder Wertemengen
- $\subset$, $\subseteq$, $\supset$, $\supseteq$ Teilmengenbeziehungen zwischen zwei Datensatzspalten- oder Wertemengen
- $\notin$, $\in$ (Nicht-) Element-von-Beziehungen von Datensatzspalten und Werten zu Datensatzspalten- und Wertemengen

Um den Gebrauch der Notation zu veranschaulichen, wird an dieser Stelle trotz seiner Trivialität der formale Beweis von Regel 1 durchgeführt.

Beweis:

(1) Sei T.1 eine Spalte der Cache-Tabelle T, welche Datensätze der Backend-Tabelle T_B zwischenspeichern kann. Also befinden sich in T.1 nur Elemente, die auch in $T_B.1$ vorhanden sind:
$T.1 \subseteq T_B.1$

(2) Sei nun $a \in T_B.1$ eine beliebige Datensatzspalte aus Spalte $T_B.1$ der Backend-Tabelle T_B. Sei ferner $T_B.1$ unique, d.h. es gibt kein weiteres (von a verschiedenes) Element b in $T_B.1$, dessen Wert v(b) dem Wert v(a) von a gleicht:
für alle $a, b \in T_B.1$ *mit* $v(a) = v(b)$ *gilt:* $a = b$

(3) Aus (1) und (2) folgt, dass auch T.1 unique sein muss, wenn T_B.1 unique ist:
für alle a, b $\in$ T.1 mit v(a) = v(b) gilt: a = b

(4) Schließlich kann aus (3) die Bereichsvollständigkeit von T.1 gefolgert werden. Bereichsvollständigkeit ist definiert wie folgt: Wenn T.1 eine Datensatzspalte a mit dem Wert v(a) enthält, so befinden sich alle in T_B.1 vorhandenen von b verschiedenen Datensätze b, die aber den gleichen Wert v(a) enthalten auch in T.1. Da gemäß (3) keine Duplikate b des Elements a in T.1 existieren weil T.1 unique ist folgt, dass alle Elemente mit dem Wert v(a) in T.1 vorhanden sein müssen. Somit ist T.1 bereichsvollständig:
für alle a, b $\in$ T_B.1 mit a $\in$ T.1 und v(a) = v(b) gilt: b $\in$ T.1 q.e.d.

3.2 Induzierte Bereichsvollständigkeit

Zum besseren Verständnis der Ausführungen zur zweiten Optimierungsregel sollen an dieser Stelle ein paar grundlegende Zusammenhänge erläutert werden.

Eine wichtige Frage ist, wann und weshalb bestimmte Datensätze in den Cache geladen werden. Offensichtlich können Tabellen in einem Cache über RCCs „befüllt“ werden; anders ausgedrückt: Ein eingehender RCC kann das Nachladen gewisser Datensätze aus dem Backend in die Cache-Tabelle, auf die der RCC zeigt, provozieren. Der Nachladeprozess für eine Cache-Tabelle wird dadurch angestoßen, dass eine andere über einen RCC vorgelagerte Cache-Tabelle neue Datensätze nachgeladen hat. Bildlich gesehen entsteht so eine „Nachladekaskade“ entlang einer RCC-Reihe. Eine solche Kaskade beginnt immer bei einer *Füllspalte*. Eine Füllspalte ist eine Spalte, in der von außen die Wertvollständigkeit bestimmter Werte erzwungen wird. Dieses Prinzip wird vom Cache-Manager dazu genutzt, um gezielt bestimmte Datensätze in den Cache zu laden, um diesen so besser auf zukünftige Anfragen vorzubereiten.

Leider verkompliziert die Existenz von Füllspalten den Cache-Group-Optimierungsprozess, da neben den RCCs noch mit einem zweiten Konzept operiert werden muss. Um dieses Problem zu umgehen, kann eine Füllspalte auch mit Hilfe eines RCC abgebildet werden, was funktional äquivalent ist. Hierzu wird jede Füllspalte als Zielspalte eines NRCC betrachtet. Dieser geht von einer einspaltigen Kontrolltabelle im Cache aus, die nicht in der eigentlichen Backend-Datenbank existiert und nur zum gezielten Füllen der abhängigen Füllspalte dient. Das Konzept der Füllspalte wird durch diese Darstellung exakt abgebildet; Durch diese Umformung wird das ganze Cache-Group-Modell vereinfacht, da man Füllspalten nicht mehr als speziellen Spaltentyp betrachten muss, sondern sämtliche Optimierungsregeln allein auf dem RCC-Konzept aufbauen kann. Im weiteren Verlauf dieser Arbeit werden Füllspalten nur noch als eingehende RCCs dargestellt [Bühm04].

Nun aber zurück zur zweiten Optimierungsregel. Auch diese entstammt den Erkenntnissen von Härder und Bühmann [HäBü04], sie wird hier an einem Beispiel erläutert und bewiesen.

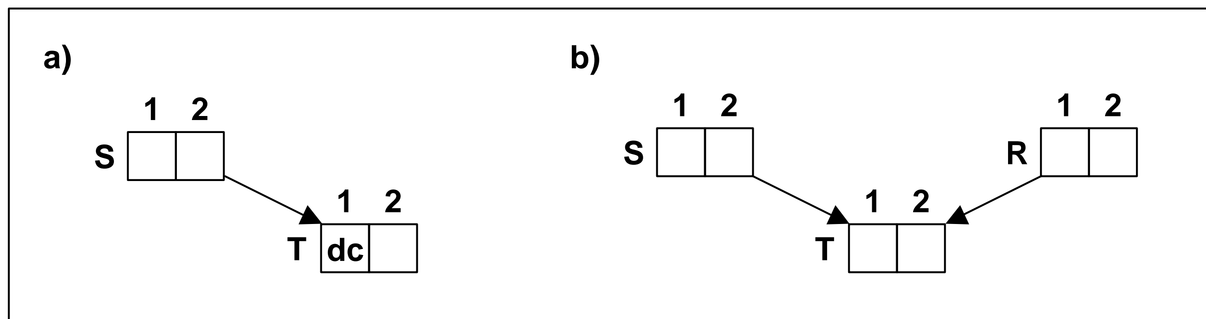

Abbildung 5: Regel 2, induzierte Bereichsvollständigkeit, Beispiel und Gegenbeispiel

Wenn in einer Tabelle der Cache Group ein RCC eingeht, dann bedeutet dies automatisch, dass die in der Ursprungsspalte des RCC vorkommenden Werte in seiner Zielspalte wertvollständig sind, allein aufgrund der Tatsache, dass RCCs vom Cache-Manager immer gültig gehalten werden.

Wenn nur in einer einzigen Spalte einer Cache-Group-Tabelle ein oder mehrere RCCs eingehen, handelt es sich um einen Spezialfall, aus dem sich ein weiterer Constraint ableiten lässt. Wie bereits erklärt, sind die Werte der RCC-Zielspalte alle wertvollständig. Da diese Zielspalte im genannten Spezialfall die einzige Stelle ist, über die (gemäß dem beschriebenen Kaskadenprinzip) neue Datensätze in die Cache-Tabelle gelangen können, können außer den wertvollständigen Werten in unserer Zielspalte keine weiteren mehr vorhanden sein. Somit ist diese Spalte in der beschriebenen RCC-Konstellation auch bereichsvollständig. Dieser Zusammenhang wird als *induzierte Bereichsvollständigkeit* bezeichnet.

Regel 2: Wenn in einer Cache-Group-Tabelle nur in einer einzigen Spalte NRCCs eingehen (was auch andere Füllspalten ausschließt), so ist die Zielspalte in jedem Fall bereichsvollständig [Bühm05].

An den Beispielen in den Abbildungen 5 und 6 wird dieser Zusammenhang verständlich. In Abbildung 5a ist die Voraussetzung der Regel 2 gegeben: Nur in der ersten Spalte von Tabelle T gehen NRCCs (in diesem Fall nur ein einziger) ein, daher ist T.1 bereichsvollständig. In Abbildung 6 wird diese Situation detailliert mit beispielhaften Datensätzen dargestellt. In der Cache-Spalte S.2 befinden sich die Werte x_1 sowie x_3, diese werden über den NRCC in T.1 wertvollständig gemacht. So werden alle Datensätze mit x_1 oder x_3 in T_B.1 in Tabelle T geladen, hierbei ist übrigens unwichtig, dass es für x_3 gar keine Datensätze in T_B gibt. Da kein anderer Weg in Form eines NRCCs existiert, über den weitere, eventuell nicht wertvollständige Werte in T.1 gelangen könnten, ist Spalte T.1 auch bereichsvollständig.

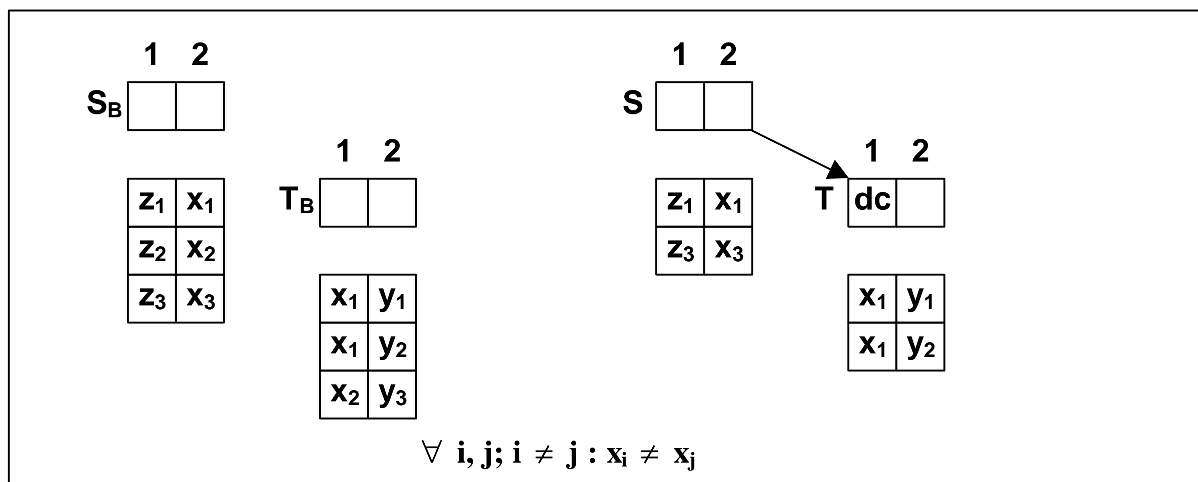

Abbildung 6: Regel 2, induzierte Bereichsvollständigkeit

Anders sieht es aus in Abbildung 5b: Hier geht ein weiterer NRCC in der zweiten Spalte von Tabelle T ein. Somit könnten hierüber weitere Datensätze in T gelangen, mit neuen Werten in T.1, die dort nicht zwingend wertvollständig sind. Folglich kann aus dieser Konstellation auch keine implizite Bereichsvollständigkeit von T.1 abgeleitet werden [.

Nachfolgend wird Regel 2 bewiesen.

Beweis:

(1) Seien T.1 und S.2 zwei Spalten der Cache-Tabellen T und S, welche jeweils Datensätze der Backend-Tabellen T_B und S_B zwischenspeichern können. Also befinden sich in T.1 nur Elemente, die auch in T_B.1, und in S.2 nur Elemente, die auch in S_B.2 vorhanden sind:
$T.1 \subseteq T_B.1$
$S.2 \subseteq S_B.2$

(2) Es existiere weiterhin ein RCC von S.2 nach T.1, der T.1 mit Datensatzspalten befüllt:
für alle $a \in T_B.1,\ c \in S.2$ *mit* $v(a) = v(c)$ *gilt:* $a \in T.1$

(3) Durch den Nachladeeffekt sind alle Werte in T.1, die von S.2 kamen, in T.1 wertvollständig:
für alle $a,b \in T_B.1,\ c \in S.2$ *mit* $b \in T.1$ *und* $v(a) = v(b) = v(c)$ *gilt:* $a \in T.1$

(4) Diese Wertvollständigkeit ist der erste Schritt in Richtung der zu beweisenden Bereichsvollständigkeit. Allerdings ist wäre es möglich, dass neben dem betrachteten noch weitere RCCs existieren, über die Werte in T.1 gelangen könnten – Werte, die eventuell nicht wertvollständig sein könnten, wodurch T.1 auch nicht bereichsvollständig wäre. Betrachten wir zuerst die Möglichkeit weiterer RCCs von beliebigen Spalten x, die direkt in T.1 eingehen:
für alle $a \in T_B.1,\ c \in x$ *mit* $v(a) = v(c)$ *gilt:* $a \in T.1$

(5) Offensichtlich sind sämtliche Werte, die über einen beliebigen RCC direkt auf T.1 in diese Spalte gekommen sind, immer noch wertvollständig:
für alle a,b $\in$ T_B*.1, c* $\in$ *(S.2* $\cap$ *x) mit b* $\in$ *T.1 und v(a) = v(b) = v(c) gilt: a* $\in$ *T.1*

(6) Die zweite Möglichkeit, über die Werte in T.1 gelangen könnten, wäre ein RCC auf eine der von T.1 verschiedenen Spalten in Tabelle T – die Voraussetzung von Regel 2 schließt diese jedoch aus. Somit sind alle Werte in T.1 über einen RCC direkt nach T.1 hierher gelangt:
für alle a $\in$ *T.1 gilt: es gibt ein c* $\in$ *(S.1* $\cap$ *x) mit v(a) = v(c)*

(7) Aus dieser Schlussfolgerung (6) lässt sich zusammen mit den Erkenntnissen (3) und (5), dass sämtliche Werte in T.1 wertvollständig sind, folgern, dass T.1 bereichsvollständig ist, womit Regel 2 bewiesen wäre:
für alle a,b $\in$ T_B*.1 mit b* $\in$ *T.1 und mit v(a) = v(b) gilt: a* $\in$ *T.1 q.e.d.*

Ein Spezialfall der Regel 2 ist ein sogenannter *Selbst-RCC*, der in derselben Spalte ein und ausgeht. Mit Hilfe eines solchen Konstrukts lässt sich sicherstellen, dass die betroffene Spalte in jedem Fall bereichsvollständig ist. Zum Teil ergibt sich dies direkt aus Regel 2, darüber hinaus ist ein Selbst-RCC aber auch unabhängig von etwaigen weiteren eingehenden NRCCs in anderen Spalten derselben Tabelle. Denn falls (auf welchem Weg auch immer) neue Werte in die betroffene Spalte gelangen, werden alle anderen Spaltenelemente mit dem gleichen Wert durch den Selbst-RCC unmittelbar nachgeladen, woraus die unbedingte Bereichsvollständigkeit folgt.

Was die Bereichsvollständigkeit betrifft, so gehen wir davon aus, dass die bisher vorgestellten Regeln vollständig sind, dass also durch konsequente Anwendung dieser Regeln auf alle Tabellen eines Caches alle bereichsvollständigen Spalten identifiziert werden. Hierfür liegt kein formaler Beweis vor, allerdings sprechen die folgenden Tatsachen für die Richtigkeit dieser Aussage: Bereichsvollständigkeit ist eher ein Ausnahmezustand, der nur unter ganz speziellen Voraussetzungen vorliegt. Gehen in mehr als einer Spalte einer Tabelle NRCCs ein, so kann im Allgemeinen über die Wertvollständigkeit der Werte der ganzen Tabelle praktisch keine verlässliche Aussage getroffen werden, da es viele Wege gibt, auf denen Einzelwerte „hineingeschmuggelt“ werden können. Somit kann für keine der Spalten dieser Tabelle generelle Bereichsvollständigkeit angenommen werden. Die große Ausnahme bilden hierbei unique-Spalten, die allein auf Grund ihrer eingeschränkten Wertemenge immer bereichsvollständig sind. Genau diese einsichtigen Zusammenhänge werden durch die beiden Regeln 1 und 2 abgebildet.

3.3 Gegenläufige RRCCs

Bevor wir näher auf die Regel für gegenläufige RRCCs eingehen, wird eine weitere Eigenschaft definiert, die Grundlage für das Vorhandensein von RRCCs ist: Die *Spaltenabhängigkeit* oder *column dependency*.

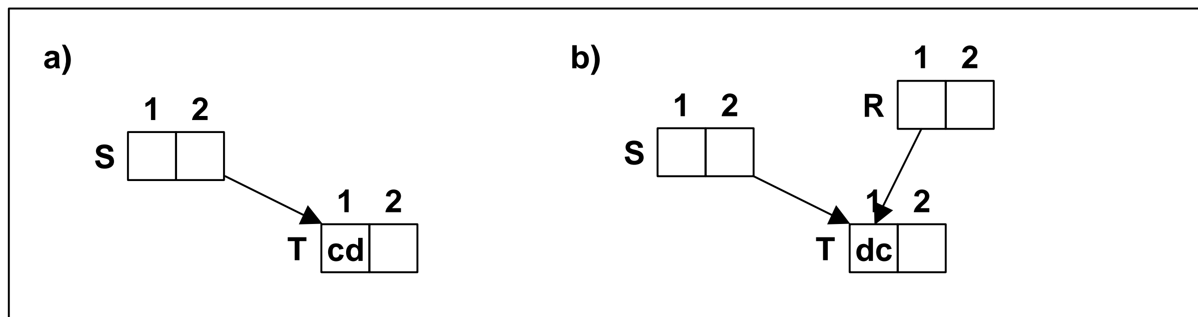

Abbildung 7: Spaltenabhängigkeit, Beispiel und Gegenbeispiel

Definition: Eine Spalte T.x einer Cache-Group-Tabelle T ist genau dann *spaltenabhängig* oder *column dependent* (cd) von der Spalte S.y einer anderen Cache-Group-Tabelle S, wenn T.x bereichsvollständig ist und außer einem NRCC von S.y kein anderer NRCC nach T zeigt.

In diesem Fall gilt:

$v(T.x) \subseteq v(S.y)$

Da der eine RCC nach T.1 der einzige Weg ist, auf dem neue Werte in T.1 gelangen können, ist die Wertemenge von T.1 Teilmenge der Wertemenge von S.2, im Extremfall entspricht T.1 S.2 genau.

Abbildung 7 stellt diesen Zusammenhang graphisch dar. In Abbildung 7a ist T.1 Ziel eines NRCC, außerdem hat keine weitere Spalte in T einen eingehenden NRCC, folglich ist T.1 wertvollständig (dc). Ferner zeigt kein weiterer NRCC auf T.1, somit ist T.1 auch spaltenabhängig (cd) (von S.2).

Bei Abbildung 7b wurde eine weitere Cache-Tabelle R ergänzt, mit einem NRCC auf T.1. Hierdurch ist T.1 nach wie vor wertvollständig (dc), aber nicht mehr spaltenabhängig (cd).

Mit der Definition von Spaltenabhängigkeit können wir fortschreiten zur nächsten Regel.

Wurde eine Spalte einer Cache-Tabellen als spaltenabhängig identifiziert, und ist ihre Vorgängerspalte (die, von der sie spaltenabhängig ist) wertvollständig, so kann dem eingehenden NRCC ein RRCC in entgegengesetzter Richtung (*Rück-RCC*) hinzugefügt werden. Dieser verläuft also von der spaltenabhängigen Spalte zu deren Vorgängerspalte.

Dieser Constraint gilt in jedem Fall, da der eingehende NRCC der einzige Weg ist, über den in eine spaltenabhängige Spalte neue Werte nachgeladen werden können. Somit müssen sämtliche in dieser Spalte befindliche Werte sich auch in der Vorgängerspalte befinden, da sie derentwegen überhaupt erst in die abhängige Spalte geladen wurden. Ist die Vorgängerspalte zusätzlich noch bereichsvollständig, so sind ihre Werte alle wertvollständig. Daher gilt auch der Constraint, dass in der Vorgängerspalte sämtliche Werte der abhängigen Spalte wertvollständig sind, was durch einen RCC, in diesem Fall einen RRCC, dargestellt werden kann.

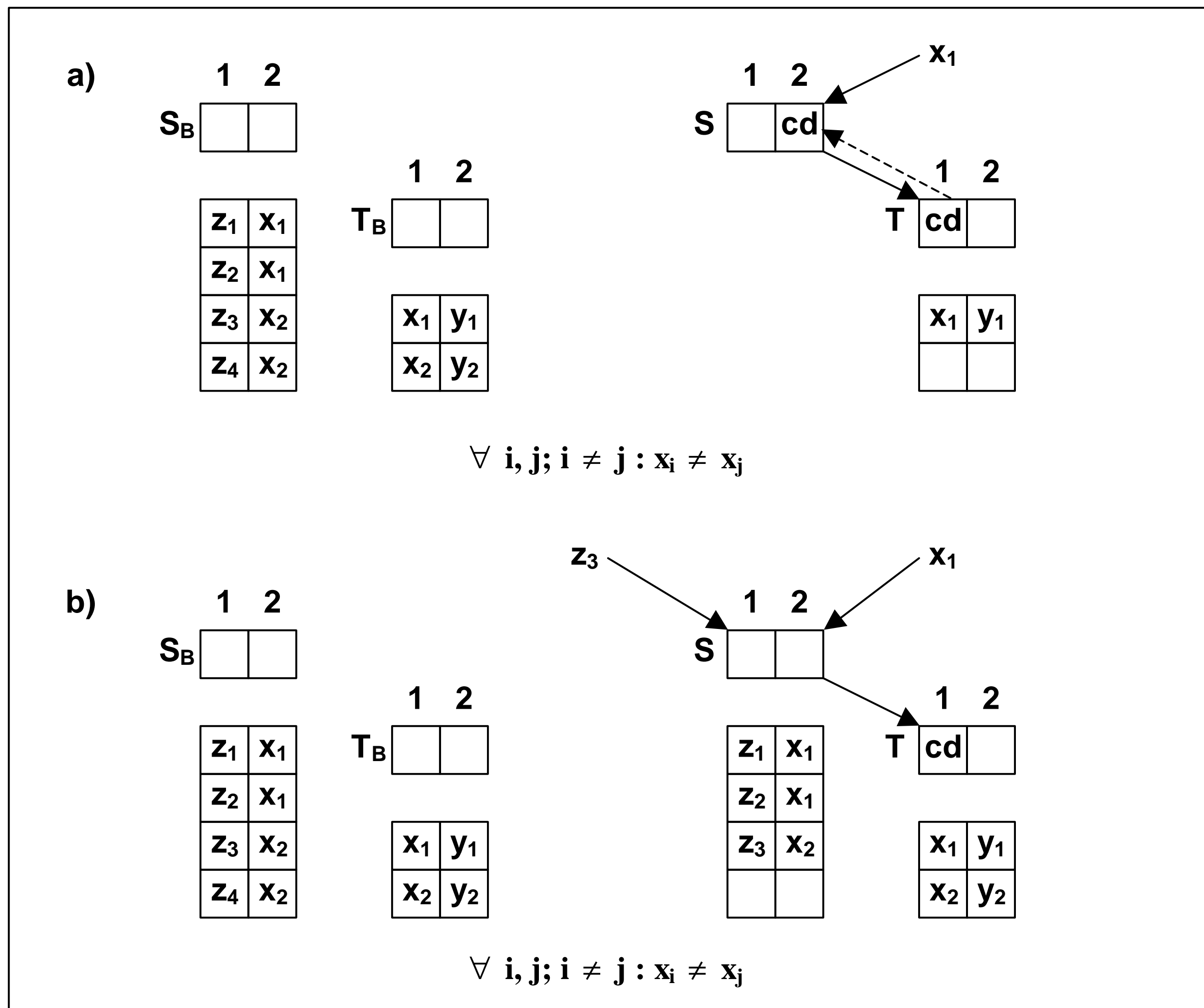

Abbildung 8: Regel 3, gegenläufige RRCCs, Beispiel und Gegenbeispiel

Regel 3: Ist eine Spalte einer Cache-Tabelle über einen NRCC spaltenabhängig von einer bereichsvollständigen Spalte einer anderen Tabelle, so gilt ein dem NRCC entgegengesetzter RRCC von der abhängigen Spalte zur Ursprungsspalte des NRCC.

Anhand von Abbildung 8 wird die Logik dieser Regel verständlich: In Abbildung 8a existieren zwei Tabellen im Backend, S_B und T_B. In der Cache Group wurden zwei NRCCs vorgegeben, einer von S.2 nach T.1, sowie einer von einer nur angedeuteten Tabelle nach S.2. Offensichtlich sind S.2 und T.1 bereichsvollständig, und T.1 ist zusätzlich spaltenabhängig von S.2. (S.2 ist ebenfalls spaltenabhängig, was bei diesem Beispiel jedoch nichts zur Sache tut.) Da mit S.2 der Vorgänger der spaltenabhängigen Spalte T.1 bereichsvollständig ist, greift die gerade vorgestellte Regel, somit kann ein RRCC von T.1 nach S.2 eingezeichnet werden.

Abbildung 8b wurde gegenüber Abbildung 8a um einen NRCC nach S.1 erweitert, hierdurch fällt die Bereichsvollständigkeit für S.2 weg. Durch die gewählte Konstellation wird deutlich, warum nun der RRCC aus 5a nicht mehr gültig ist: Durch den neuen NRCC wird zusätzlich zu den beiden Datensätzen mit x_1 auch der Datensatz (z_3, x_2) aus S_B in S geladen. Wegen des NRCC nach T muss nun auch der Datensatz (x_2, y_2) aus T_B in T geladen werden. Ein RRCC von T.1 nach S.2 kann nun offensichtlich nicht mehr gelten, da dieser voraussetzen würde,

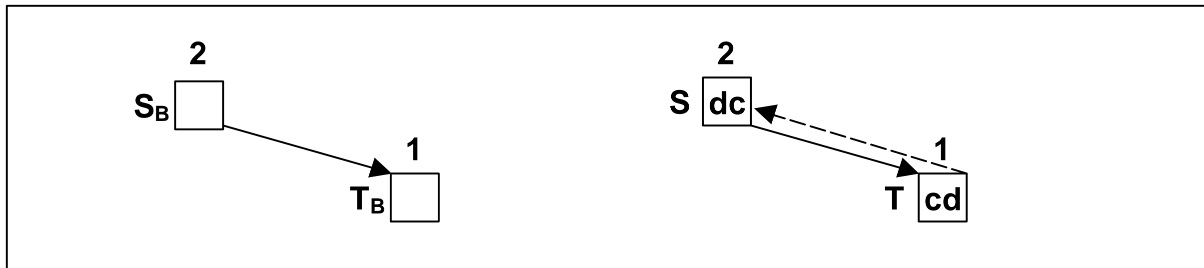

Abbildung 9: Regel 3, gegenläufige RRCCs, Beweismodell

dass sich auch (z_4, x_2) in S befindet. Da S aber nicht bereichsvollständig ist, ist dem nicht so, folglich darf der RRCC in diesem Fall nicht gesetzt werden.

Seien nun für den Beweis die Tabellen aus Abbildung 8 auf die benötigten Spalten reduziert, wie in Abbildung 9 dargestellt.

Beweis:

(1) Seien T.1 und S.2 zwei Spalten der Cache-Tabellen T und S, welche jeweils Datensätze der Backend-Tabellen T_B und S_B zwischenspeichern können. Also befinden sich in T.1 nur Elemente, die auch in T_B.1, und in S.2 nur Elemente, die auch in S_B.2 vorhanden sind:
$T.1 \subseteq T_B.1$
$S.2 \subseteq S_B.2$

(2) Es existiere weiterhin ein NRCC von S.2 nach T.1, der T.1 mit Datensatzspalten befüllt, wobei dieser NRCC der einzige ist, der in Tabelle T eingeht:
für alle $a \in T_B.1,\ b \in S.2$ *mit* $v(a) = v(b)$ *gilt:* $a \in T.1$

(3) Sei außerdem S.2 bereichsvollständig:
für alle $a,b \in S_B.2$ *mit* $b \in S.2$ *und mit* $v(a) = v(b)$ *gilt:* $a \in S.2$

(4) Gemäß Regel 2 folgt aus (1) und (2) die Bereichsvollständigkeit von T.1:
für alle $a,b \in T_B.1$ *mit* $b \in T.1$ *und mit* $v(a) = v(b)$ *gilt:* $a \in T.1$

(5) Darüber hinaus folgt aus der Definition der Spaltenabhängigkeit und der Einschränkung in (2), dass nur ein einziger NRCC in T eingeht, dass T.1 auch spaltenabhängig von S.2 ist:
$v(T.1) \subseteq v(S.2)$

(6) Durch die wertmäßige Teilmengenschreibweise der Spaltenabhängigkeit (5) wird offensichtlich, dass sich nur Werte in T.1 befinden, die auch mindestens einmal in S.2 vorkommen.
für alle $a \in T.1$ *gilt: es gibt ein* $b \in S.2$ *mit* $v(a) = v(b)$

(7) Da S.2 auch bereichsvollständig ist (3), sind alle Werte in T.1 dank (6) bereichsvollständig in S.2. Damit ergibt sich als Resultat aus (3) und (6) ein RRCC von T.1 nach S.2, womit Regel 3 bewiesen wäre:
für alle $a \in S_B.2,\ b \in T.1$ *mit* $v(a) = v(b)$ *gilt:* $a \in S.2$ *q.e.d.*

3.4 RRCCs zu Geschwistern

Auch diese Regel baut auf der Spaltenabhängigkeit auf, allerdings ist sie vielseitiger als die vorhergehende Regel – in bestimmten Konstellationen können durch sie extrem viele RRCCs hinzugefügt werden.

Wenn eine Spalte als spaltenabhängig erkannt wurde, und wenn von ihrer Vorgängerspalte weitere RCCs auf andere Spalten ausgehen, dann können von der abhängigen Spalte zu jeder Nachfolgerspalte des Vorgängers RRCCs eingezeichnet werden. Das besondere hierbei ist, dass es nicht auf Bereichsvollständigkeit des Vorgängers und aller anderen beteiligten Spalten ankommt. Es ist nicht einmal erforderlich, dass die anderen Nachfolger über einen NRCC vom Vorgänger abhängen, selbst ein RRCC reicht für die Anwendung dieser Regel aus.

Da eine spaltenabhängige Cache-Spalte definitionsgemäß nur Werte beinhalten kann, die auch schon in der Vorgängerspalte vorhanden sind, lassen sich Schlüsse auf die „Geschwister-Spalten“ ziehen, die ebenfalls über einen RCC von der gleichen Vorgängerspalte gefüllt werden. Wir wollen uns nochmals ins Gedächtnis rufen, dass ein RCC die in der Ausgangsspalte vorkommenden Werte in der Zielspalte wertvollständig macht, unabhängig davon, ob diese dort überhaupt vorkommen und ebenso unabhängig von weiteren eingehenden RCCs der Zieltabelle. Somit sind in einer spaltenabhängigen Spalte mindestens dieselben Werte wertvollständig wie in allen Geschwistern, was einem RCC zu den Geschwistern des abhängigen Spalte entspricht.

Regel 4: Ist eine Spalte einer Cache-Tabelle über einen NRCC spaltenabhängig von einer Spalte einer anderen Tabelle, und gehen von der Ursprungsspalte des NRCC weitere RCCs (NRCCs und RRCCs) aus, so gilt ein zusätzlicher RRCC von der abhängigen Spalte zu jeder Zielspalte eines dieser RCCs.

In Abbildung 10 wird dies verdeutlicht: Im Backend befinden sich die drei Tabellen R_B, T_B und S_B, alle werden im Cache vorgehalten als R, T und S. Hierbei ist T.1 durch einen NRCC von S.1 bereichsvollständig und ebenfalls spaltenabhängig von S.1. Dadurch ist Regel 4 anwendbar. S.1 als Vorgängerspalte zu T.1 hat einen weiteren NRCC zu R.1. Gemäß der Regel gilt nun ein RRCC von T.1 nach R.1.

In der Cache-Tabellen S befinden sich drei der Datensätze aus S_B: x_1, x_2 und x_4; x_3 ist nicht gecacht. Somit werden wegen des NRCCs nach T.1 zwei der drei Datensätze aus T_B in T geladen: x_1 und x_2; x_3 ist nicht vorhanden. Obwohl nun die zweite Nachfolgertabelle von S.1 mehr als einen weiteren RCC enthält und somit R.1 nicht einmal bereichsvollständig ist, ist der neue RRCC für alle möglichen Fälle gültig.

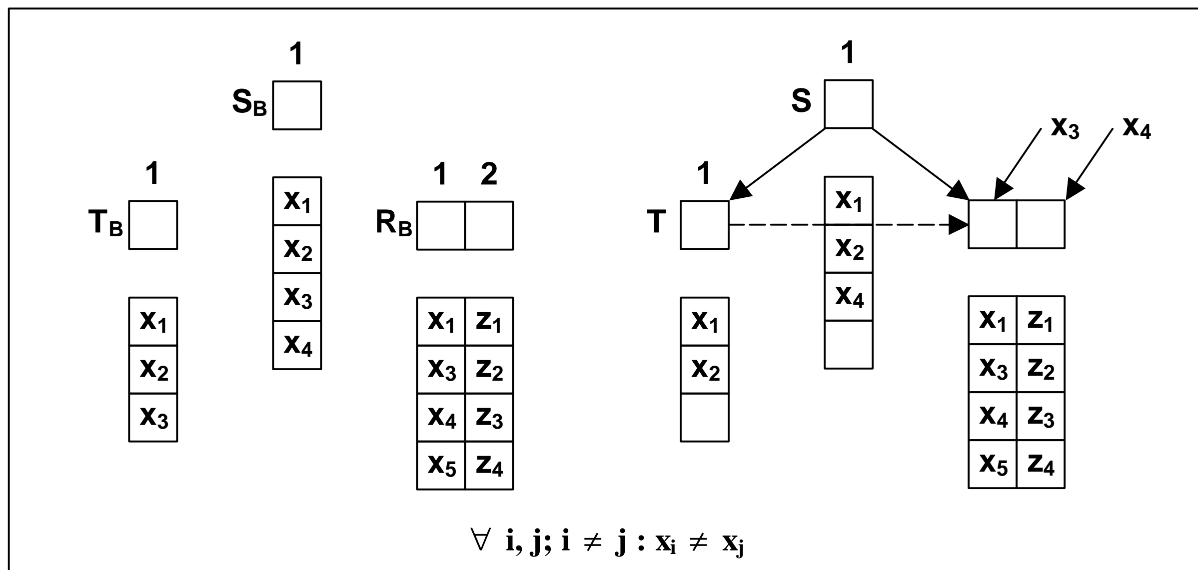

Abbildung 10: Regel 4, RRCCs zu Geschwistern

- x_1 ist in T.1 und auch in R.1 wertvollständig, da x_1 auch in S.1 vorhanden ist und so in beide Cache-Spalten „eingeschleust" wird.
- x_2 ist in T.1 vorhanden, in R.1 aber nicht – trotzdem gilt der RRCC, da x_2 durch das Vorkommen von in S.1 wieder in beiden Spalten wertvollständig ist, auch wenn in R_B.1 und somit auch in R.1 gar kein Exemplar vorhanden ist.
- x_4 ist in R.1 vorhanden, in T.1 aber nicht. Der neue RRCC fordert also gar nicht, dass x_2 in R.1 existiert. Trotzdem ist x_4 wieder in beiden Spalten wertvollständig über das Vorkommen von x_4 in S.1.
- x_3 und x_5 werden über zwei RCCs zu zwei unterschiedlichen Spalten von R in R.1 geladen, allerdings nicht über S.1, folglich sind diese Werte auch in T.1 nicht vorhanden. Selbstverständlich verletzen diese zusätzlich vorhandenen Werte die Gültigkeit des neuen RRCC nicht.

Beweis:

(1) Seien T.1, S.1 und R.1 zwei Spalten der Cache-Tabellen T, S und R, welche jeweils Datensätze der Backend-Tabellen T_B, S_B und R_B zwischenspeichern können:
$T.1 \subseteq T_B.1$
$S.1 \subseteq S_B.1$
$R.1 \subseteq R_B.1$

(2) Es existiere weiterhin ein NRCC von S.1 nach T.1, der T.1 mit Datensatzspalten befüllt, wobei dieser NRCC der einzige ist, der in Tabelle T eingeht:
für alle $a \in T_B.1$, $b \in S.1$ *mit* $v(a) = v(b)$ *gilt:* $a \in T.1$

(3) Es existiere ein zweiter RCC von S.1 nach R.1, der R.1 mit Datensatzspalten befüllt:
für alle $a \in R_B.1$, $b \in S.1$ *mit* $v(a) = v(b)$ *gilt:* $a \in R.1$

(4) Gemäß Regel 2 folgt aus (1) und (2) die Bereichsvollständigkeit von T.1:
für alle $a,b \in T_B.1$ *mit* $b \in T.1$ *und mit* $v(a) = v(b)$ *gilt:* $a \in T.1$

(5) Darüber hinaus folgt aus der Definition der Spaltenabhängigkeit und der Einschränkung in (2), dass T.1 auch spaltenabhängig von S.1 ist:
$v(T.1) \subseteq v(S.1)$

(6) Durch die wertmäßige Teilmengenschreibweise der Spaltenabhängigkeit (5) wird offensichtlich, dass sich nur Werte in T.1 befinden, die auch mindestens einmal in S.1 vorkommen.
für alle $a \in T.1$ *gilt: es gibt ein* $b \in S.1$ *mit* $v(a) = v(b)$

(7) Durch den RCC aus (3) werden alle in S.1 vorkommenden Werte in R.1 wertvollständig gemacht. Zu beachten ist dabei, dass dazu nicht all diese Werte in $R_B.1$ vorkommen müssen, denn ein Wert ist immer vollständig in einer Spalte, wenn in der zugehörigen Backendtabelle kein einziger Datensatz mit diesem Wert in dieser Spalte existiert. In Verbindung mit (6) lässt sich hieraus ein RRCC von T.1 nach R.1 ableiten, was der Aussage von Regel 4 entspricht:
für alle $a \in R_B.1$, $c \in T.1$ *mit* $v(a) = v(c)$ *gilt:* $a \in R.1$ *q.e.d.*

4 Systematischer Optimierungsablauf

4.1 Notwendigkeit einer Regelanwendungs-Systematik

Im vorgehenden Kapitel wurden vier Optimierungsregeln aufgezeigt und bewiesen, mit denen Datenbankcaching im Rahmen des Cache-Group-Konzeptes verbessert werden kann. Da sich diese Arbeit nicht allein auf die theoretischen Grundlagen beschränken soll, stellt sich die Frage, wie diese Regeln auf eine Cache Group angewendet werden können. Betrachtet man die dargestellten Regeln mit besonderem Augenmerk auf ihre Eintrittsbedingungen und ihre Schlussfolgerungen, so fällt auf, dass die Reihenfolge der Regelanwendung Auswirkung auf das Ergebnis haben kann.

Die Voraussetzungen der Regeln bestehen aus Eigenschaften einer Spalte und deren Vorgängern, wie zum Beispiel Bereichsvollständigkeit oder Spaltenabhängigkeit. Genau diese Eigenschaften lassen sich wiederum aus der Regelanwendung ableiten. Ebenso zählt auch die Existenz von RCCs, die in einer Tabellenspalte eingehen, zu den Anwendungsbedingungen anderer Regeln, die wiederum selbst neue RCCs erzeugen oder bestehende NRCCs zu redundanten RRCCs degradieren. Hierdurch bedingt lässt sich an manchen Stellen in der Cache Group eine Regel erst anwenden, nachdem ihre Voraussetzungen durch die Anwendung einer anderen Regel geschaffen wurden. Je nachdem, in welcher Reihenfolge man die Gültigkeit einzelner Regeln an bestimmten Stellen prüft, wird mehr oder weniger Optimierungspotential ausgeschöpft. Wenn eine schlechtere Reihenfolge gewählt wird, sind mehrere Regelanwendungs-Iterationen notwendig, um das gleiche maximale Optimierungsergebnis zu erreichen.

Es wird deutlich, dass ein durchdachtes Konzept benötigt wird, mit dem der Optimalzustand bei jeder Cache Group sicher und trotzdem mit geringem Aufwand gezielt erreicht werden kann. Im Rahmen dieser Arbeit wurde hierfür das sogenannte *Durchlaufkonzept* entwickelt, welches im folgenden Abschnitt vorgestellt wird.

4.2 Grundlagen des Durchlaufkonzepts

Das Durchlaufkonzept basiert auf der Erkenntnis, dass in einer Cache Group neue Datensätze nur über eingehende NRCCs in eine Cache-Tabelle gelangen können. Stellen wir uns die Grundaufgabe eines Cache-Managers einmal vor: Wenn in einer Tabellenspalte neue Werte auftauchen, weil neue Datensätze in die Tabelle geladen wurden, so muss der Cache-Manager in sämtlichen von dieser Tabellenspalte über einen NRCC abhängigen Tabellen neue Datensätze nachladen oder zumindest prüfen, ob dies zur Aufrechterhaltung der Constraints notwendig ist. Hierdurch entsteht bildlich gesprochen ein fortwährender Datenfluss entlang der NRCCs in einer Cache Group. Dieser beginnt in den Tabellen mit Füllspalten, da in diesen durch „Außeneinwirkung“ Datensätze nachgeladen werden, und endet in Tabellen ohne ausgehende NRCCs.

Aufgrund dieser Erkenntnis geht das Durchlaufkonzept die Tabellen entlang dieser Flussrichtung durch. Für jede Tabelle mit mindestens einer Füllspalte erfolgt ein Durchlauf der Cache Group, ausgehend von eben dieser Tabelle. In jeder Tabelle werden, die Eintrittsbedingungen vorausgesetzt, alle vier beschriebenen Regeln angewendet. Daraufhin verlagert sich der Fokus auf den Nachfolger der aktuellen Tabelle, also eine Tabelle, die über einen NRCC direkt von der aktuellen abhängt. Auf diese werden wiederum die Regeln angewendet, wonach es zum nächsten Nachfolger geht und so weiter, bis schließlich eine Tabelle erreicht wird, die keine ausgehenden NRCCs hat. Hierbei ist zu beachten, dass NRCC-Zyklen in einer Cache Group einen endlosen Durchlauf provozieren könnten, und daher gesondert behandelt werden müssen. Diese Problematik wird in Kapitel 4.4 behandelt und später in Kapitel 5.2 ausführlicher besprochen.

4.3 Baumstruktur

Diese Vorgehensweise verkompliziert sich etwas, da Tabellen prinzipiell mehrere nachfolgende Tabellen haben können. Die Lösung hierfür liegt in einer Baumstruktur: Die Tabellen stellen die Knoten eines Baumes dar, und ausgehende RCCs die von einem Knoten ausgehenden Kanten. Von der Starttabelle mit der Füllspalte als Wurzel ausgehend lässt sich so ein Baum aufspannen, der mit einer der gängigen Methoden strukturiert durchlaufen werden kann. Aus einem Grund, der im nächsten Absatz deutlich wird, wird in dieser Arbeit die Tiefensuche bevorzugt. Hierbei wird ein eingeschlagener RCC-Pfad so lange weiterverfolgt, bis keine weiteren ausgehenden RCCs mehr gegeben sind, woraufhin jeweils von der letzten besuchten Tabelle mit einer bisher noch nicht verfolgten Verzweigungsmöglichkeit fortgefahren wird, so lange, bis alle Tabellen besucht wurden. Bei jeder neu besuchten Tabelle werden selbstverständlich die bekannten Regeln angewendet.

4.4 Zyklenbehandlung

Allerdings hat die bisherige Darstellung noch einen Schönheitsfehler: Sie geht von einer Cache Group aus, in der keine Zyklen auftreten, also einer Menge von Tabellen, die über NRCCs ringförmig so verbunden sind, dass sie praktisch endlos reihum durchlaufen werden könnten. Ein solcher Endloszyklus muss verständlicherweise vermieden werden, damit ein entgültiger Optimierungszustand erreicht werden kann. Die Vorgehensweise des Durchlaufkonzeptes ist hierbei die folgende: Wenn bei der Tiefensuche im Tabellenbaum eine Tabelle erreicht wird, die auf dem aktuellen Pfad von der Wurzel aus bereits besucht wurde, so wurde hier ein Zyklus gefunden. An dieser Stelle muss der NRCC, der bereits verfolgt wurde, ignoriert werden, da ansonsten der entdeckte Zyklus immer wieder durchlaufen würde, und dadurch der Algorithmus nicht terminieren würde. Andere von dieser Tabelle ausgehende NRCCs sind hiervon nicht betroffen und müssen normal abgearbeitet werden. Dadurch wird die Cache Group sozusagen wieder zu einem „normalen“ Baum ohne Zyklen „zurechtgestutzt“, und es entsteht keine Endlosschleife beim Durchlauf.

Allerdings wird durch dieses Verfahren kein Zyklus in einer Cache Group beseitigt. Zyklen können aber beim Ablauf des Cache-Managers die Effizienz und gegebenenfalls auch die Sicherheit des Caches erheblich gefährden – auf diese Problematik wird jedoch in Kapitel 5.2 genauer eingegangen.

4.5 Weitere Verfahrensoptimierung

Nach einem einzelnen Cache-Group-Durchlauf von einer Füllspaltentabelle aus kann es sein, dass durch die Regelanwendung die Voraussetzungen für weitere Regeln geschaffen wurden, die die Cache Group weiter optimieren können. Um dieser Möglichkeit zu begegnen, wird die Cache Group so oft durchlaufen, bis im letzten Gesamtdurchlauf keine Änderungen durch Regelanwendung mehr vorgenommen werden konnten. Hierdurch wird gewährleistet, dass die Cache Group bestmöglich optimiert wird. Ein Zustand, in dem keine weiteren Regeln angewendet werden können wird immer erreicht, es kann also nicht zu einem Zustand kommen, in dem die Cache Group „oszilliert“, weil eine Regel eine Änderung vornimmt, die im nächsten Durchlauf wieder rückgängig gemacht wird und so weiter. Dies kann garantiert werden, da keine zwei Regeln derart gegensätzliche Veränderungen vornehmen. Es werden nur Bereichsvollständigkeitsmarkierungen gesetzt, und RRCCs hinzugefügt bzw. NRCCs zu RRCCs degradiert, aber keine neuen NRCCs hinzugefügt oder Bereichsvollständigkeitsmarkierungen entfernt.

Von jeder Tabelle mit Füllspalten wird nacheinander ein Durchlauf mit dieser Tabelle als Baumwurzel wie beschrieben durchgeführt. Dies ergibt einen Gesamtdurchlauf. Wurde ein Gesamtdurchlauf beendet, kommt es darauf an, ob durch ihn weitere Optimierungen vorgenommen wurden. Wenn ja, dann wird ein erneuter Gesamtdurchlauf ausgeführt. Erst, wenn in einem Gesamtdurchlauf keine Änderungen mehr vorkamen, terminiert der Durchlaufalgorithmus, da folglich der mit den gegebenen Regeln maximal mögliche Optimierungszustand erreicht wurde.

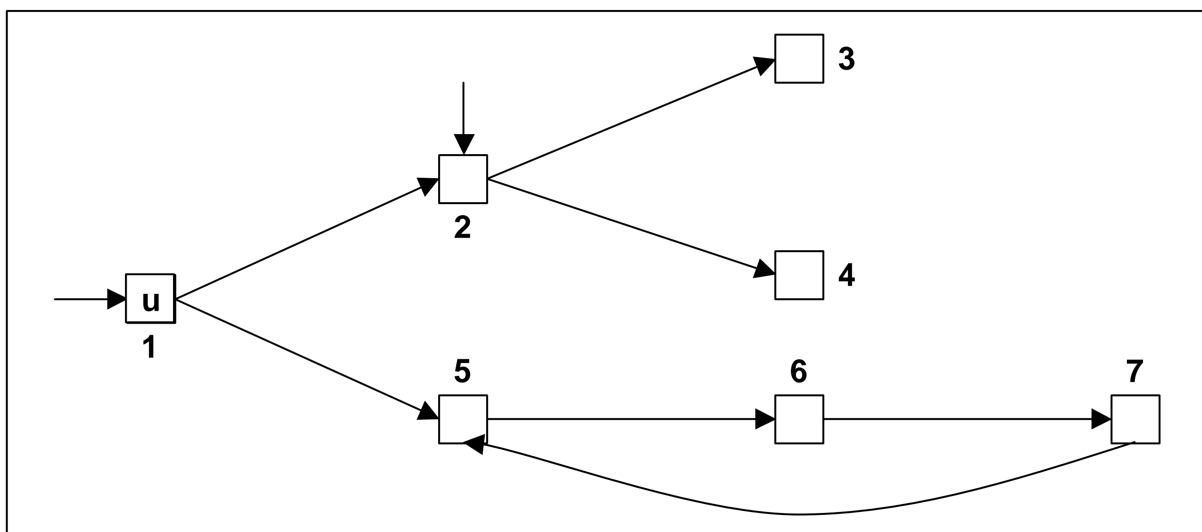

Abbildung 11: Beispiel für Durchlaufkonzept, Ausgangssituation

Eine weitere Effizienzsteigerung des Analyseverfahrens kann ohne Einbußen der Optimierungsqualität vorgenommen werden, indem die Anwendung von Regel 1 vorgezogen wird. Diese besagt, dass unique-Spalten auch bereichsvollständig sind. Da durch Anwendung keiner der Regeln eine Spalte unique werden kann, die es vorher nicht war, müssen die Voraussetzungen von Regel 1 nicht wiederholt geprüft werden. Es reicht aus, diese vor dem eigentlichen Durchlauf einmal für jede Spalte der Cache Group anzuwenden. Danach kann man sich beim eigentlichen Durchlaufverfahren auf die Regeln zwei bis vier konzentrieren, dies spart insbesondere bei komplexeren Cache Groups Ressourcen für die Optimierung ein. Anhand des folgenden etwas komplizierteren Beispiels wird die Funktionsweise des Durchlaufverfahrens gezeigt. Ausgehend von der Konstellation in Abbildung 11 werden mehrere Durchläufe abgearbeitet wie beschrieben, wodurch sich die optimierte Endsituation wie in Abbildung 12 dargestellt ergibt.

Die im Folgenden verwendete Optimierungsnotation ist so zu lesen: Zuerst wird die Tabelle bzw. Spalte genannt, die im Rahmen des Durchlaufkonzeptes gerade betrachtet wird. Darauf folgt gegebenenfalls die Art der vorgenommenen Optimierung mit „dc“ bzw. „cd“ für eine hinzugefügte Bereichsvollständigkeits- bzw. Spaltenabhängigkeitsmarkierung. Alternativ wird ein neuer RRCC mit Von- und Zu-Spalte bezeichnet. Dahinter wird jeweils in Klammern die angewendete Optimierungsregel erwähnt.

Nun aber zurück zu dem Beispiel aus Abbildung 11:

- Vor dem Durchlauf
    - 1 dc (R1)
- Erster Gesamtdurchlauf
    - Erster Durchlauf ausgehend von Füllspaltentabelle 2
        - 2 dc (R2)
        - 3 dc (R2), cd, RRCC 3 → 2 (R3), RRCC 3 → 4 (R4)
        - 4 dc (R2), cd, RRCC 4 → 2 (R3), RRCC 4 → 3 (R4)
    - Zweiter Durchlauf ausgehend von Füllspaltentabelle 1
        - 1 cd
        - 2, 3, 4 Keine Änderungen
        - 5 dc (R2)
        - 6 dc (R2), cd, RRCC 6 → 5 (R3)
        - 7 dc (R2), cd, RRCC 7 → 6 (R3), RRCC 7 → 5 (R4) (NRRC wird zu RRCC!)
- Zweiter Gesamtdurchlauf
    - Erster Durchlauf ausgehend von Füllspaltentabelle 2

- 2, 3, 4 Keine Änderungen
- Zweiter Durchlauf ausgehend von Füllspaltentabelle 1
  - 1, 2, 3, 4 Keine Änderungen
  - 5 cd, RRCC 5 → 1 (R3), RRCC 5 → 2 (R4)
  - 6 RRCC 6 → 1 (R4), RRCC 6 → 2 (R4)
  - 7 RRCC 7 → 1 (R4), RRCC 7 → 2 (R4)
- Dritter Gesamtdurchlauf
  - Erster Durchlauf ausgehend von Füllspaltentabelle 2
    - 2, 3, 4 Keine Änderungen
  - Zweiter Durchlauf ausgehend von Füllspaltentabelle 1
    - 1, 2, 3, 4, 5, 6, 7 Keine Änderungen

Hervorgehoben sei an dieser Stelle, dass anfänglich ein Zyklus über die Spalten 5, 6 und 7 existiert. Interessanterweise löst sich dieser von selbst auf, indem bei der Analyse von Spalte 7 mit Anwendung der Regel 4 der NRCC von 7 nach 5 durch einen RRCC ersetzt wird – und RRCCs werden, was den Tabellendurchlauf betrifft, ignoriert. In Fällen, in denen sich Zyklen nicht so problemlos auflösen, müsste der NRCC, der den Kreis schließt, vom Algorithmus ignoriert werden. In unserem Beispiel würde der NRCC von 7 nach 5 nicht weiter verfolgt, und da ansonsten bereits alle Tabellen besucht worden wären, wäre ein einfacher Durchlauf beendet.

Im zweiten Gesamtdurchlauf wird deutlich, dass sich durch die Vorarbeit vom ersten Gesamtdurchlauf neue Möglichkeiten eröffnet haben: Nachdem der NRCC von 7 nach 5 zum RRCC degradiert wurde, zählt er nicht mehr als eingehender RCC in 5 im Sinne der Definition von Spaltenabhängigkeit, worauf Spalte 5 als cd markiert werden kann. Dies wiederum löst eine Kettenreaktion aus, die weitere RRCCs in die Cache Group einfügt.

Abbildung 12: Beispiel für Durchlaufkonzept, maximale Optimierung

5 Grenzen des Optimierungskonzeptes

Mit Hilfe der Optimierungsregeln und des Durchlaufkonzepts können Cache-Group-Konfigurationen meist erfolgreich für den effizienten Einsatz im Cache-Manager optimiert werden. Dennoch gibt es Grenzen für die Optimierung:

Einerseits kann mit den bestehenden Regeln nicht jeder gültige zusätzliche Constraint gefunden werden; Hierauf wird in Kapitel 5.1 näher eingegangen. Andererseits können Cache Groups auch so angeordnet sein, dass sie beim Einsatz im Cache-Datenbank-Server durch unkontrolliertes Nachladen den Caching-Betrieb stark beeinträchtigen. Bei derart gestalteten Cache Groups hilft auch das dargestellten Optimierungskonzept nicht weiter. Um solche Konstellationen geht es in Kapitel 5.2.

5.1 Unerkannte RRCCs

Die vier vorgestellten Regeln zur Cache-Group-Optimierung decken einen Großteil der Optimierungsmöglichkeiten ab, aber nicht alle. Teilweise wären extrem aufwändige Regeln notwendig, um Einzelfälle abzudecken, die in bei realen Datenbank-Konfigurationen selten bis gar nicht auftreten. Im Folgenden wird ein solcher Fall gezeigt, in dem nicht alle gültigen RRCCs erkannt werden.

5.1.1 Umarmende RCCs

In Abbildung 13 ist eine Cache-Group-Konstellation dargestellt, die hier anschaulich „umarmende“ RCCs genannt werden soll. Hierbei gehen von einer Spalte mehrere NRCCs aus, von deren Zielspalten wieder NRCCs ausgehen, die sich in einer gemeinsamen Zielspalte treffen. Dabei ist es übrigens unerheblich, ob die „RCC-Arme“ wie dargestellt jeweils durch nur eine Tabelle (S bzw. R) oder mehrere Tabellen führen, solange die ausgehenden NRCCs jeweils von der Spalte (S.1 bzw. R.1) ausgehen, in der die Vorgänger eingegangen sind.

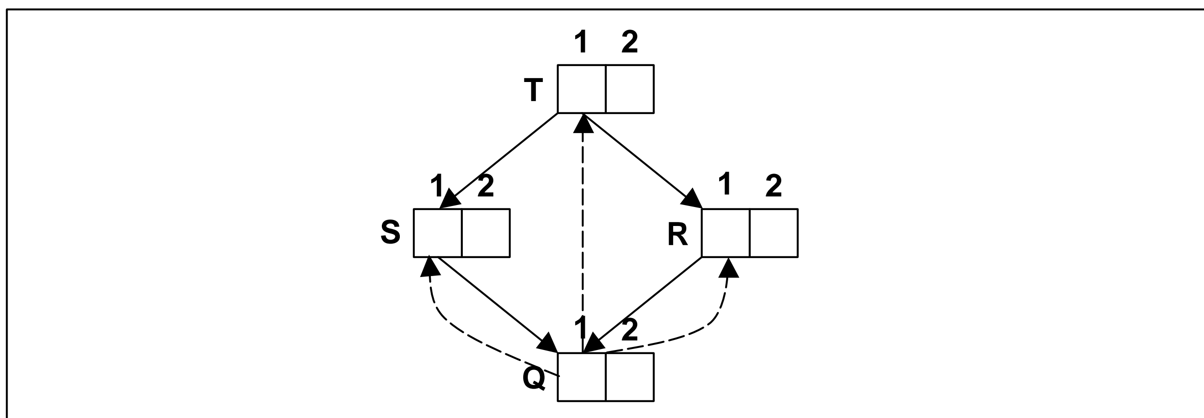

Abbildung 13: Beispiel für nicht erkannte RRCCs

5.1.2 Problematik

Bei der in diesem Beispiel vorgegebenen Cache Group ist es allein mit Hilfe unserer vier Optimierungsregeln nicht möglich, die drei RRCCs von Spalte Q.1 zu den Spalten S.1, R.1 und T.1 abzuleiten, die bei genauer Betrachtung jedoch gelten. Das Problem hierbei ist, dass Spalte Q.1 von zwei NRCCs abhängt und somit nicht spaltenabhängig sein kann. Da die zwei Quellspalten S.1, und R.1 dieser NRCCs jedoch beide allein von derselben Ursprungsspalte T.1 abhängen, würden die drei Rück-RRCCs von Spalte Q.1 dennoch gelten. Diese Problematik umfasst sämtliche Konstellationen, in denen eine Spalte über beliebig viele Zwischentabellen auf zwei Wegen von einer anderen Spalte aus erreicht werden kann. Voraussetzung ist, dass die NRCCs der Zwischentabellen jeweils in derselben Spalte ein- und ausgehen. Um solche Situationen zu erkennen, müsste der Analysealgorithmus mit verhältnismäßig großem Aufwand eine Liste der im aktuellen Durchlauf bereits besuchten Tabellen führen, mit deren Hilfe sich der erneute Besuch von Spalte Q.1 in unserem Beispiel erkennen lassen könnte.

Bei der Suche nach eventuell nicht erkannten RCCs sind wir sehr sorgfältig vorgegangen, dennoch lässt sich nicht ausschließen, dass eventuell weitere Cache-Group-Anordnungen existieren, bei denen nicht alle RRCCs gefunden werden. Sollte es solche geben, so dürfte es sich aber ebenfalls um seltene Ausprägungen handeln, so dass die Nichterkennung praktisch keine Relevanz haben dürfte.

5.2 Umgang mit RCC-Zyklen

Es gibt Cache-Group-Konfigurationen, die den Caching-Prozess nicht wie gewünscht optimieren, sondern ihn im Gegenteil behindern; Dies ist insbesondere der Fall, wenn RCC-Zyklen vorkommen. Folglich sollte die Optimierung der Cache Group derartige Konstellationen verhindern. Der folgende Abschnitt stellt einführend dar, was Zyklen sind, und warum sie gefährlich für einen Cache sein können. Nachfolgend wird darauf eingegangen, wie der in dieser Arbeit vorgestellte Algorithmus mit Zyklen umgeht.

5.2.1 Zyklen

Um einen Zyklus handelt es sich dann, wenn eine Reihe von Cache-Group-Tabellen mit NRCCs so untereinander verbunden sind, dass eine Endlosschleife (bei Analysieren wie auch beim Laden in den Cache) möglich ist. In Abbildung 14 sind solche Fälle dargestellt. Wie bereits in Kapitel 4 erwähnt, könnte ein Zyklus beim Durchlaufkonzept zu einem nicht endenden Analyseprozess führen, wenn er nicht vom Algorithmus erkannt wird (was der hier vorgestellte aber beherrscht). Viel schlimmer wirken sich zyklische Cache Groups jedoch später beim Einsatz im Cache-Datenbankserver aus. Hier bestimmt die vorgegebene Cache Group über ihre RCCs, welche Datensätze vom Cache-Manager in die Cache-Tabellen geladen werden. Befindet sich ein Zyklus in der benutzten Cache Group, so lädt der Cache-Manager unkontrolliert Datensätze vom Backend nach, gegebenenfalls so lange, bis der Inhalt einiger Cache-Tabellen dem ihrer Backend-Tabellen gleicht. Diese führt einerseits dazu, dass im Cache näherungsweise eine Kopie des Backends entsteht, was weder beabsichtigt noch

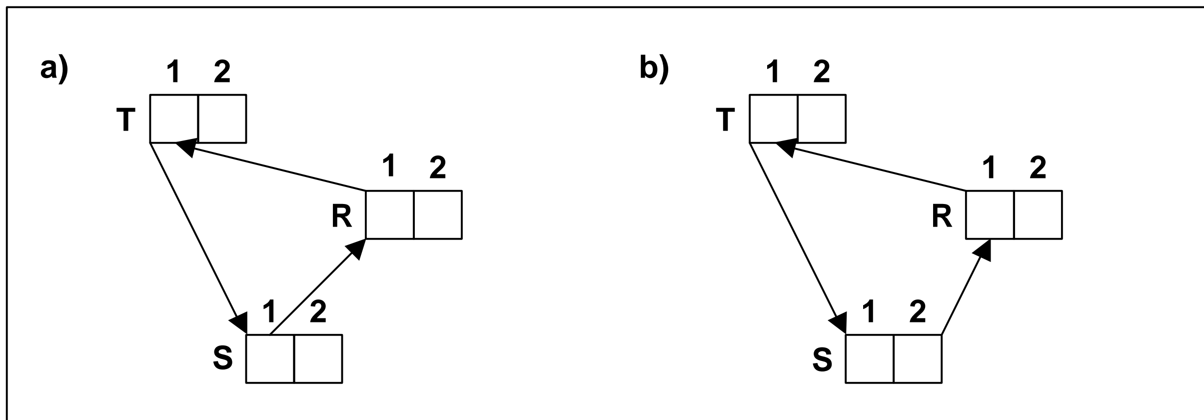

Abbildung 14: homogener Zyklus (a), heterogener Zyklus (b)

hilfreich ist. Andererseits wird das ständige Nachladen zur Hauptbeschäftigung des Cache-Servers, der dadurch weniger Ressourcen für seine eigentliche Hauptaufgabe, das Beantworten von Benutzeranfragen, zur Verfügung hat.

Prinzipiell muss unterschieden werden zwischen zwei verschiedenen Arten von Zyklen: homogenen und heterogenen [HäBü04c].

5.2.2 Homogene Zyklen

Wenn jeweils nur eine einzige Spalte pro Tabelle von einem Zyklus betroffen ist, so wird dieser Zyklus als homogen bezeichnet. Anders ausgedrückt: In jeder zum Zyklus gehörigen Tabelle verlässt der ausgehende Zyklus-RCC die Tabelle in derselben Spalte, in der auch der andere Zyklus-RCC eingeht (Abbildung 14a).

Diese Konstellation ist eigentlich ungefährlich für die Sicherheit des Caches: Wird in eine der Zyklus-Tabellen (z.B. T in Abbildung 14a) ein neuer Datensatz geladen, und gelangt hierdurch ein neuer Wert (z.B. „7“) in die Zyklus-Spalte, so lädt der Cache-Manager in der von dieser Spalte abhängigen Zyklus-Tabelle (S) gegebenenfalls weitere Datensätze nach, bis dieser Wert („7“) in der Zielspalte des RCC (S.1) wertvollständig ist. So lange in der zugehörigen Backendspalte der Zielspalte des jeweils nächsten Zyklus-RCC der Wert („7“) vorhanden ist, werden die betroffenen Datensätze auch in der nächsten Tabelle des Zyklus nachgeladen [Bühm04]. Im Extremfall setzt sich diese Prozedur durch sämtliche dem Zyklus zugehörige Tabellen (R) fort, bis sie wieder bei der ursprünglichen Tabelle (T) ankommt, und auch hier der eine neue Wert („7“) wertvollständig gemacht wird. Spätestens jetzt ist die ausgelöste Nachlade-Kaskade aber beendet, da der neue Wert in der folgenden abhängigen Spalte (S.1) ja während des Durchlaufs bereits wertvollständig gemacht wurde. Allerdings wird der beschriebene Prozess für jeden neuen Wert in einer zum Zyklus gehörenden Spalte neu angestoßen und ausgeführt, wodurch ein Mehraufwand für den Cache-Manager im Vergleich zu zyklusfreien Cache Groups entsteht.

Folglich müssen (reine) homogene Zyklen nicht vom Optimierungs-Algorithmus behandelt werden; Eine solche Cache Group kann unbesorgt dem Cache-Manager übergeben werden.

Trotzdem können homogene Zyklen in gewissen Konstellationen schädlich für einen Cache sein. Bevor hierauf in Kapitel 5.2.4 näher eingegangen wird, werden erst die heterogenen Zyklen besprochen.

5.2.3 Heterogene Zyklen

Ein Zyklus ist heterogen, wenn er sich bei mindestens einer Tabelle auf mehr als eine Spalte erstreckt. Ein Zyklus-RCC kommt also in einer Spalte einer Tabelle an, der folgende Zyklus-RCC verlässt diese Tabelle aber in einer anderen Spalte (Abbildung 14b).

Das Problem hierbei kommt durch den Sprung in dieser Tabelle zustande: Bei homogenen Zyklen bleibt es während des ganzen Nachladedurchlaufs bei dem einen neu hinzugekommenen Wert, bei heterogenen Zyklen können durch diesen Sprung immer wieder neue Werte eingebracht werden, die den Zyklus weiter anstoßen! Wird in eine Tabelle (z.B. T in Abbildung 14b) ein neuer Wert geladen (z.B. „7“), wird dieser den Zyklus-RCCs folgend in den weiteren Zyklus-Tabellen (S) nachgeladen, bis in einer Tabelle ein Sprung vorkommt (S). Die in dieser Tabelle nachgeladenen Datensätze (z.B. (7;8) und (7;9)) bringen völlig neue Werte („8“ und „9“) in die Spalte mit dem ausgehenden RCC (S.2). In die nachfolgenden Tabellen des Zyklus (R und T) werden nun diese neuen Werte nachgeladen und wertvollständig gemacht, bis zum nächsten Sprung, also spätestens wieder dem in der gerade betrachteten Tabelle (S). Hier ergeben sich in der Ausgangsspalte (S.2) wiederum neue Werte. Je nach Inhalt der Datensätze in den Zyklustabellen kann die Nachladeprozedur somit sehr lange andauern, im Extremfall so lange, bis sich sämtliche Datensätze der Backend-Tabelle im Cache befinden. Während der Nachladezeit ist der Cache vollständig ausgelastet und steht (allein schon der Inkonsistenz der Constraints wegen) auch nicht für parallele Anfragebearbeitung zur Verfügung. Darüber hinaus ist es nicht Sinn eines Caches, dass er die Backend-Tabellen spiegelt, hierfür sind Repliktechniken gedacht. Somit stellen heterogene Zyklen in Cache Groups eine Gefahr für die Stabilität des Caches dar.

Um die Sicherheit des Caches zu garantieren, müsste der Optimierungs-Algorithmus heterogene Zyklen behandeln. Allerdings gestaltet sich dies problematisch, denn wenn einfach einer der Zyklus-RCCs gelöscht wird, so ändert sich auch die vom Administrator vorgegebene Semantik der Cache Group. Aus diesem Grund beschränkt sich der Algorithmus auf die Zyklus-Erkennung; Wird ein heterogenes Exemplar in einer Cache Group gefunden, so wird die Bearbeitung mit einem entsprechenden Benutzerhinweis abgebrochen, damit dieser die Grund-Konfiguration der Cache Group überdenken und verbessern kann.

5.2.4 Stark verschachtelte Zyklen

Obwohl ein homogener Zyklus wie bereits erläutert unschädlich ist, kann er in Kombination mit weiteren homogenen Zyklen einen zusammengesetzten heterogenen Zyklus bilden, welcher dennoch zu endlosen Nachladefolgen führt. Dies ist dann der Fall, wenn eine Tabelle an mehreren Zyklen beteiligt ist, wobei diese jeweils eine andere ihrer Spalten berühren (Abbil-

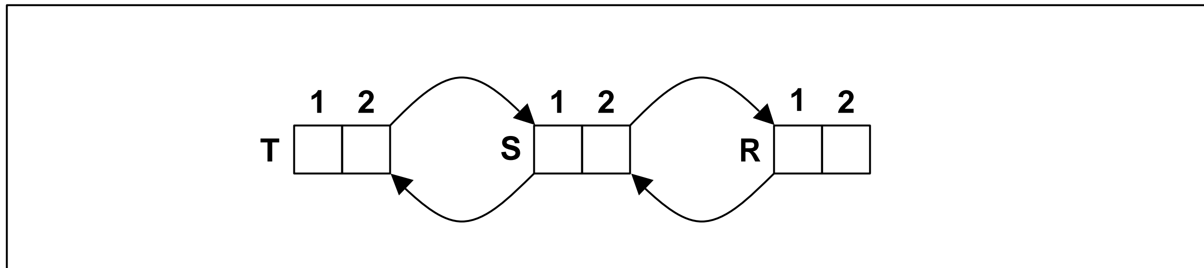

Abbildung 15: verschachtelter Zyklus

dung 15). In diesem Fall ergeben die zwei homogenen Zyklen (T, S) und (S, R) zusammen den großen heterogenen Zyklus (T, S, R, S).

Beispielsweise könnte ein Zyklus einen neuen Wert (z.B. „3“ über den RCC von T) in eine Spalte (S.1) der gemeinsamen Tabelle (S) laden. In diesem Zuge würde dieser Wert wertvollständig in S.1 gemacht (z.B. durch Nachladen der Datensätze (3;4) und (3;5) in S). Nun müssten auch die neuen Werte der Spalte S.2 („4“ und „5“) über den rechten Zyklus auch in R.1 und in S.2 selbst wertvollständig gemacht werden. Hierdurch könnten wiederum neue Werte in Spalte S.1 gelangen (z.B. „6“, „7“, „8“, und „9“ über die Datensätze (6;4), (7;4), (8;5) und (9;5)). Auch diese werden wieder über den Umweg des linken Zyklus wertvollständig in T.2 und S.1 gemacht... Wie bei heterogenen Zyklen kann die Nachladekaskade abhängig vom Datensatzinhalt bis zur vollständigen Abbildung der Backend-Tabelle dauern.

Andererseits gibt es auch hier wieder Ausnahmen. Wenn eine der von einem der RCCs berührten Spalten in der gemeinsamen Tabelle (S) unique ist (z.B. S.1), dann stoppt der Nachladezyklus nach einem Durchlauf. Wenn ein neuer Wert in S.2 gelangt, wird dieser über den rechten Zyklus wertvollständig gemacht. Die daraus resultierenden neuen Werte in S.1 werden im linken Zyklus wertvollständig gemacht; Wieder zurück bei S.1 ist die Kaskade jedoch beendet, da S.1 aufgrund der unique-Eigenschaft bereits bereichsvollständig ist, und kein weiteres Nachladen notwendig ist.

Verschachtelte Zyklen werden vom Durchlaufalgorithmus, wie er in Kapitel 4.4 vorgestellt wurde, erkannt (wobei allerdings auch die genannte Ausnahme fälschlicherweise als gefährliche Variante klassifiziert würde). Wichtig hierfür ist, dass beim Auffinden eines homogenen Zyklus nicht sofort der weitere Durchlauf blockiert wird, sondern dass etwaige noch nicht benutzte RCCs weiterverfolgt werden. In dem Beispiel in Abbildung 15 sähe dies so aus:

Der Durchlauf startet z.B. bei Tabelle T, geht dann weiter über S nach R und weiter nach S. Hier findet der Algorithmus den homogenen Zyklus (S, R). Trotzdem sucht er von hier aus weiter. Um keinen Endlosdurchlauf zu verursachen wird der RCC nach R, der schon im Durchlaufpfad vorhanden ist, nicht nochmals verfolgt. Somit bleibt noch der RCC von S nach T, und in T wird ein weiterer Zyklus (T, S, R, S) gefunden, da sich auch diese Tabelle im Durchlaufpfad befindet. Dieser Zyklus hat allerdings zwei Spaltensprünge in Tabelle S, und wird daher als heterogen erkannt. Alternativ könnte es aber auch passieren, dass der Durchlaufalgorithmus beim ersten Besuch von Tabelle S zuerst den RCC nach T verfolgt. In diesem Fall würde zuerst der homogene Zyklus (T, S) gefunden. Da von T aber keine weiteren RCCs ausgehen, würde der Algorithmus zurück zu S gehen, den RCC von S nach T aus dem Durch-

laufpfad streichen und von S aus den RCC nach R verfolgen. Der restliche Durchlauf gestaltet sich wie im ersten Fall, und der verschachtelte heterogene Zyklus würde trotzdem gefunden.

Durch das Durchlaufprinzip ist der Algorithmus sehr flexibel und findet unterschiedlichste Kombinationen von verschachtelten Zyklen. Allerdings kann nicht ausgeschlossen werden, dass ganz spezielle Mehr-Zyklus-Konstellationen trotzdem nicht erkannt werden.

6 Präsentation des Analysetools

Teil dieser Arbeit ist ein Java-Tool, welches den in dieser Arbeit vorgestellten Optimierungsalgorithmus mit den vier Regeln implementiert. An dieser Stelle wird der Aufbau des Tools beschrieben (Kapitel 6.1), ferner wird auf den Ablauf des Optimierungsvorgangs (Kapitel 6.2) sowie das Ein- und Ausgabeformat für Cache Groups (Kapitel 6.3) eingegangen.

6.1 Übersicht über die Klassen

Das Java Tool besteht aus den folgenden Klassen:

Start
ist die Klasse mit der Main-Methode, die den groben Ablauf des Programms steuert. Aus ihr heraus werden die essentiellen Methoden für den Cache-Group-Import, die Optimierung und die Cache-Group-Ausgabe aufgerufen.

metadata.CacheGroup, metadata.Table, metadata.Column und **metadata.RCC**
bilden den Rahmen für die interne objektorientierte Darstellung einer beliebigen Cache Group. Dabei bauen die Bestandteile aufeinander auf: Zu Anfang wird eine einzige CacheGroup instanziert. Diese bietet Methoden, mit der ihr einzelne Tabellen hinzugefügt werden, indem eine Instanz von Table generiert wird und in der CacheGroup verknüpft wird. Auf dieselbe Weise lassen sich auch Column-Objekte zu bestimmten Tables hinzufügen. In jeder Column sind mit Variablen Informationen über den Zustand der Spalte hinterlegt, also ob sie unique, bereichsvollständig oder spaltenabhängig ist. Das Paket metadata bildet somit das Grundgerüst, mit dem Cache Groups abgebildet werden können. Zusätzlich bietet die Klasse CacheGroup eine Methode, mit der RCCs in das Tabellengerüst eingefügt werden können. Hierzu wird die Klasse RCC instanziert, die auch Zustandsinformationen wie NRCC oder RRCC speichert. RCCs werden in einer Liste in CacheGroup verwaltet, um einfachen Zugriff auf die Gesamtheit der RCCs zu ermöglichen. Darüber hinaus wird jedes RCC-Objekt auch beidseitig mit den betroffenen Column-Objekten verknüpft.

Hervorzuheben ist, dass über Zugriffsmethoden sämtliche Cache-Group-Objekte untereinander erreichbar sind. Von einer Tabelle kann jede ihrer Spalten erreicht werden und umgekehrt von einer Spalte auf ihre übergeordnete Tabelle zugegriffen werden. Weiterhin können von jeder Spalte (und Tabelle) alle in ihr aus- und eingehenden RCCs erreicht werden, und aus einem RCC beide miteinander verknüpften Spalten. Somit besteht die Möglichkeit, sozusagen durch die Cache Group zu wandern und sich von einem Objekt zum nächsten „durchzuhangeln“. Von dieser Möglichkeit macht der Optimierungsalgorithmus ausführlich Gebrauch, wie im nächsten Kapitel dargestellt wird.

xml.XML und **xml.XMLErrorHandler**
ermöglichen den Import und Export von Cache Groups im XML-Format. Mit der Klasse Xml wird bei Programmstart eine Eingabedatei ausgelesen und als DOM-Tree abgebildet. Aus dieser „flachen“ Darstellung wird nun die interne „plastische“ Objekt-Darstellung generiert,

wie gerade beschrieben wurde. Der ErrorHandler ist hierbei für die aussagekräftige Ausgabe von XML-Fehlern zuständig. Nach der Optimierung werden die Änderungen des Objektmodells in das DOM-Modell übernommen, woraufhin dieses wieder als XML-Datei ausgegeben wird.

rule.Rule, rule.AbstractRule, rule.CheckUnique, rule.CheckDC, rule.CheckBackRCC und **rule.CheckSiblingRRCCs**
implementieren die Regeln, die in Kapitel 3 vorgestellt wurden. Dabei ist rule.Rule das Interface, über das einzelne Regeln auf eine Cache-Tabelle angewendet werden. Demgegenüber ist rule.AbstractRule die abstrakte Oberklasse für die vier konkreten Regelklassen, welche je eine der vier Optimierungsregeln abbilden.

Rules
kreiert von jeder der gerade aufgezählten rule-Klassen eine Instanz, und wendet diese Regel-Objekte auf die interne Darstellung der Cache Group an. Hierzu ist auch der Algorithmus des Durchlaufprinzips in dieser Klasse definiert. Der genaue Optimierungsablauf wird in Kapitel 6.2 ausführlicher dargestellt.

Path
beinhaltet letztlich alles, um Zyklen finden und auf Gefährlichkeit hin überprüfen zu können. Hierzu implementiert diese Klasse einen Stack, in dem der RCC-Pfad eines Durchlaufs von der Wurzeltabelle an verfolgt wird. Zudem beinhaltet Path eine Methode, die bei Auftreten eines Zyklus prüft, ob es sich um ein heterogenes Exemplar handelt oder nicht.

6.2 Implementierung der Regeln und des Durchlaufprinzips

Nachdem für die Eingabe-Cache-Group das Objektmodell aufgebaut wurde, wird dieses systematisch analysiert und optimiert. Als erster Schritt wird Regel 1 (unique-Spalten) angewendet. Diese Regel ist von den übrigen separierbar, da im Optimierungsverlauf keine weiteren unique-Spalten entstehen, was ein wiederholtes Überprüfen auf diese Eigenschaft überflüssig macht. Zur Ausführung dieser Regel werden alle Spalten aller Tabellen der Reihe nach durchlaufen. Falls eine Spalte mit der Eigenschaft unique gefunden wird, wird für diese das dc-Flag für Bereichsvollständigkeit gesetzt.

Alle übrigen Regeln müssen gemeinsam und wiederholt angewendet werden, um alle gültigen Constraints zu finden. Hierzu wird das in Kapitel 4 beschriebene Durchlaufverfahren angewendet. Von jeder Tabelle mit einer Füllspalte aus werden Durchläufe ausgeführt, bis keine Änderungen mehr an der Cache Group vorgenommen werden. Dabei sieht ein Durchlauf so aus: Für jeweils eine Tabelle werden die Voraussetzung der Regeln 2 bis 4 geprüft. Wenn eine Regel angewendet werden kann, werden direkt die Optimierungen in das Modell eingefügt. Dies geschieht je nach Regel durch das Setzen von dc- oder cd-Flags in einer Spalte, durch das Einfügen eines neuen RCC mit „Redundant-Flag“ oder das Degradieren eines NRCC zum RRCC, indem das Redundant-Flag von false auf true gesetzt wird. Im Anschluss daran wird die nächste Tabelle ausgewählt. Hierzu wird der Reihe nach jeder von der aktuellen Tabelle ausgehende NRCC ausgewählt und die von diesem NRCC abhängige Tabelle als aktuelle

gewählt. Für diese Tabelle fängt die Analyse wieder bei der Regelanwendung an. Sind alle ausgehenden NRCCs einer Tabelle abgearbeitet, dann geht der Algorithmus zurück zu der Tabelle, von der aus die aktuelle ausgewählt wurde; Von dort werden weitere ausgehende NRCCs verfolgt. Durch dieses Verfahren entsteht die beschriebene Tiefensuche über die Cache Group. Die vernetze Struktur des Cache-Group-Objektmodells unterstützt das Durchlaufen der einzelnen Tabellen und Spalten hierbei optimal. Hervorzuheben ist an dieser Stelle, dass RRCCs zur Ermittlung der Durchlaufreihenfolge nicht einbezogen werden, da RRCCs wie der Name schon sagt redundant sind. Durch die Beschränkung auf NRCCs wird der Durchlauf so effizient wie möglich gehalten. Durch Regelanwendung kann es geschehen, dass einzelne NRCCs zu RRCCs degradiert werden und so im nächsten Durchlauf nicht mehr berücksichtigt werden. Allerdings ist durch die Redundanz gewährleistet, dass nach wie vor alle Tabellen über die verbleibenden NRCCs erreicht werden können.

Wichtig ist die Zyklus-Erkennungs-Funktion, allein schon deshalb, damit der Optimierungsalgorithmus bei der Tiefensuche nicht endlos im Kreis läuft. Um dies zu vermeiden, wird, wie bereits in der Klassenübersicht beschrieben, ein Stack benutzt, der die im aktuellen Durchlauf bereits verfolgten NRCCs von der Wurzel (Füllspalte von der aus aktuell gesucht wird) an speichert. Beim Zurückgehen im Baum wird jeweils der letzte NRCC gelöscht. Bei jeder Tabelle wird nun geprüft, ob ein von ihr ausgehender NRCC bereits in dieser Liste steht. In diesem Fall liegt nämlich ein Zyklus an dieser Stelle vor.

Wurde ein Zyklus auf diese Art gefunden, so muss untersucht werden, ob es sich um einen ungefährlichen homogenen oder einen gefährlichen heterogenen handelt. Hierzu wird eine Methode aufgerufen, die den NRCC-Pfad rückwärts analysiert, denn die letzten Einträge dieser Liste charakterisieren den gefundenen Zyklus. Die Unterscheidung ist einfach; Jeder Zyklus wird daraufhin untersucht, ob die Zyklus-NRCCs überall in derselben Spalte ein- und ausgehen. Ist dies für alle beteiligten NRCCs der Fall, so handelt es sich um einen homogenen Zyklus. In diesem Fall wird ein Hinweis ausgegeben, aber ansonsten die Analyse fortgesetzt – natürlich wird der bereits verfolgte NRCC ignoriert, damit der Algorithmus sich nicht in dem gerade gefundenen Zyklus verrennt. Tritt jedoch ein NRCC-Sprung bei einer der Zyklus-Tabellen auf, so ist der Zyklus heterogen. In diesem Fall wird die weitere Analyse aus den bereits erläuterten Gründen mit einer erläuternden und den Zyklus angebenden Meldung abgebrochen.

Wie erläutert werden Gesamtdurchläufe über alle Füllspalte aus so lange durchgeführt, bis sich keine Änderungen mehr ergeben. Dann ist die Optimierung beendet.

6.3 XML-Input- und -Output-Funktionalität

Der Einfachheit halber verwendet das Optimierungs-Tool dasselbe XML-Dateiformat für die Ein- und Ausgabe einer Cache-Group-Konfiguration. Dadurch ist es auch möglich, eine bereits optimierte Cache Group erneut einzugeben – es werden sich lediglich keine weiteren Änderungen ergeben.

Das Format entspricht dem von Merker eingeführten [Merk05], damit ist auch die Schnittstelle zu Merkers Cache-Manager hergestellt. Im Folgenden wird dieses Format kurz charakterisiert:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dbcache SYSTEM "dtdfile.dtd">
<dbcache>
  <backend>
      <schema name="schemeE">
        <table id="T1" name="backendtable" primarykey="C1">
            <column id="C1" unique="false" nullable="false">
              <name>id</name>
              <type>integer</type>
            </column>
            (more columns belonging to that table)
        </table>
        (more tables)
      </schema>
  </backend>
  <cachegroup>
      <schema name="dbcache">
        <table id="CT1" ref="T1" name="cachetable"/>
        (more cache tables)
      </schema>
      <fillcolumn ref="C1" percentage="1" ctrl-table-name="controltable"/>
      (more fill columns)
      <rcc from="c1" to="c17" user-defined="true" redundant="false"/>
      (more RCCs)
  </cachegroup>
</dbcache>
```

7 Ausblick

Der in dieser Arbeit vorgestellte Algorithmus ermöglicht die Optimierung von vorgegebenen Cache-Group-Konfigurationen, indem weitere, daraus ableitbare Constraints vom Typ RCC und Bereichsvollständigkeit ergänzt werden. Darüber hinaus werden potentiell gefährliche Cache Groups identifiziert und der Benutzer vor deren Einsatz gewarnt.

Dies stellt einen wichtigen Schritt hin zum effizienteren Datenbank-Caching dar.

In wieweit zusätzlich gefundene Optimierungs-Constraints die Performance eines Caches erhöhen, das kann theoretisch nur schwer beurteilt werden. Insbesondere, weil der Effizienzgewinn stark von den Auswertungs-Algorithmen des Cache-Managers abhängt, müssen praktische Tests mit verschiedenen Datenbank-Konfigurationen und Anfrage-Szenarien durchgespielt werden, um eine fundierte Antwort auf diese Frage geben zu können.

Allerdings ist zu bedenken, dass es sich bei diesem Optimierungsansatz um rein statische Verbesserungen handelt – der Algorithmus verlässt sich vollkommen auf die inhaltliche Qualität der vom Administrator vorgegebenen Cache Group. Datenbankinhalte und reale Datenbank-Anfragen werden nicht in den Optimierungsprozess mit einbezogen.

Aus diesem Grund ist zu erwarten, dass in Zukunft die wichtigsten Impulse zur weiteren Caching-Optimierung die Laufzeitebene betreffen, also den Cache-Manager, der einen realen Cache optimal verwalten soll.

Der Cache-Manager hat Zugriff auf wesentlich weiter reichende Nutzungs-Informationen, die für die Optimierung extrem nützlich sein können. Beispielsweise wäre es denkbar, dass der Cache-Manager die an ihn gestellten Datenbank-Anfragen systematisch auswertet. Ein sinnvolles Resultat hieraus wäre zum Beispiel die Erkenntnis, dass ein bestimmter RCC, der aktuell noch nicht in der Cache Group vorhanden ist, den Cache auf viele Anfragen besser vorbereiten könnte, so dass weniger Rückfragen an das Backend nötig wären. Durch die Echtzeitumgebung besteht auch die Möglichkeit, dynamisch auf inhaltliche Veränderungen zu reagieren. Wenn sich mit der Zeit viele Anfragen auf Tabellen ergeben, die vorher irrelevant waren, so könnte der Cache-Manager als Antwort diese Tabelle mit in die Cache Group und somit in den Cache aufnehmen, und andere, weniger frequentierte Tabellen daraus entfernen. Die Möglichkeiten an dieser Stelle sind beachtlich.

Abschließend kann gesagt werden: Es bleibt zu hoffen, dass der Ansatz des Datenbank-Cachings mit Cache Groups weiterhin gute Fortschritte erzielt, und dass sich die Erwartungen der Wissenschaft in das Konzept des constraintbasierten Datenbank-Cachings bewahrheiten!

Danksagung

Für seine Geduld und die tatkräftige Unterstützung möchte ich mich sehr bedanken bei Andreas Bühmann, der bei der Programmierung des Optimierungs-Tools mitgewirkt hat und diese Ausarbeitung während der Entstehung regelmäßig korrekturgelesen hat.

Literaturverzeichnis

Baye04 *Bayerlein, Christian* (2004): Datenbank-Caching: Modellierung des Füllverhaltens von Cache Groups.
URL: http://wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Bay04.DA.pdf (28.11.05).

Bühm04 *Bühmann, Andreas* (2004): Einen Schritt zurück zum negativen Datenbank-Caching. In: BTW, Karlsruhe, März 2005, S. 107-124.

Bühm05 *Bühmann, Andreas* (2005): Ein Schritt zurück ist kein Rückschritt: Mit flexibler Sondierung zum negativen Datenbank-Caching. In: Informatik – Forschung und Entwicklung, Online First, Mai 2005.

BüHM06 *Bühmann, Andreas; Härder, Theo; Merker, Christian* (2006): A Middleware-Based Approach to Database Caching.
URL: http://wwwdvs.informatik.uni-kl.de/pubs/papers/BHM06.ADBIS.pdf (22.03.06).

HäBü04a *Härder, Theo; Bühmann, Andreas* (2004): Datenbank-Caching: Eine systematische Analyse möglicher Verfahren. In: Informatik – Forschung und Entwicklung, Band 19 (2004), Nr. 1, S. 2-16.

HäBü04b *Härder, Theo; Bühmann, Andreas* (2004): Query Processing in Constraint-Based Database Caches. In: Data Engineering Bulletin, Vol. 7 (2004), No. 2, pp. 3-10.

HäBü04c *Härder, Theo; Bühmann, Andreas* (2004): Database Caching: Towards a Cost Model for Populating Cache Groups. In: Advances in Databases and Information Systems, 8th East European Conference, Budapest, Sep. 2004.

HäBü05 *Härder, Theo; Bühmann, Andreas* (2004): Value Complete, Column Complete, Predicate Complete: Magic Words Driving the Design of Cache Groups.
URL: http://wwwdvs.informatik.uni-kl.de/pubs/papers/HB05b.Magic.pdf (17.10.05).

Merk05 *Merker, Christian* (2005): Konzeption und Realisierung eines Constraint-basierten Datenbank-Cache.
URL: http://wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Mer05.DA.pdf (12.12.05).

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Studienarbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Kaiserslautern, 28.04.06

(Wolfgang Scholl)