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

Konzeption und Realisierung eines Constraint-basierten Datenbank-Cache

Diplomarbeit

von
Christian Merker

Betreuer:
Dipl.-Inf. Andreas Bühmann

Oktober 2005

Obwohl wir weiterhin davon überzeugt sind, in einer Industriegesellschaft zu leben, sind wir in Wirklichkeit auf dem Weg zu einer Gesellschaft, die auf Erstellung von Informationen und deren Verbreitung basiert.

John Naisbitt (*1930), amerikanischer Prognostiker

Inhaltsverzeichnis

1 Motivation 1

2 Einführung in das Constraint-basierte Datenbank-Caching 5

2.1 Die Idee 5
2.1.1 Cache Groups 6
2.1.2 Füllspalten 8
2.1.3 Referenzieller Cache-Constraint (RCC) 10
2.1.4 Anfragebearbeitung im Cache 11
2.1.5 Cache-Group-Föderation 13

2.2 Flexible Einstiegspunkte 14
2.2.1 Das neue Sondierungsverfahren 14
2.2.2 Negatives Datenbank-Caching 15
2.2.3 Vor- und Nachteile des neuen Sondierungsverfahrens 17

2.3 Probleme des Constraint-basierten Datenbank-Caching 18
2.3.1 Gute Zyklen, schlechte Zyklen 18
2.3.2 Verwendung mehrerer Non-Unique-Füllspalten 19
2.3.3 Optimierungs-RCCs 20

2.4 Regeln für den Entwurf von Cache Groups 21

3 Konzepte zum Caching von dynamischen Inhalten 23

3.1 Query-Result-Caching 23
3.1.1 Query-Result-Caching am Beispiel WDBAccel 24
3.1.2 Query-Containment-Algorithmen 25

3.2 MTCache: Der Prototyp von Microsoft 27
3.2.1 Bearbeitung von lesenden Anfragen 28
3.2.2 Bearbeitung von Änderungsoperationen 29

3.3 DBCache: Der Prototyp von IBM 30
3.3.1 Bearbeitung von lesenden Anfragen 31
3.3.2 Bearbeitung von Änderungsoperationen 33

3.4 Vor- und Nachteile von DBCache und MTCache 34

4 Unser Datenbank-Cache-Prototyp 37

4.1 Starten des CBCS 38

4.2 Middle Tier: Die Schnittstelle zu den Datenbanken 39

4.3 Die Initialisierung des CBCS 40
4.3.1 Normale Initialisierung 40
4.3.2 Erweiterte Initialisierung 43

4.4 Die Query-Worker-Komponente 45
4.4.1 Bearbeitung von Leseoperationen 45
4.4.2 Bearbeitung von Änderungsoperationen 49

4.5 Der Fill Daemon 52
4.5.1 Strategien zum Füllen des Cache 53

4.6 Der Hit Counter 61

4.7 Der Garbage Collector 64
4.7.1 Verdrängungsstrategien 65
4.7.2 Generierung der Delete-Anweisungen 66

4.8 Die JDBC-Schnittstelle 72

5 Implementierung des Prototyps 73

5.1 Die Umgebung 73

5.2 Implementierung der CBCS-Komponenten 74

5.2.1 Die Initialisierung 75
5.2.2 Die Warteschlangen 81
5.2.3 Der Query Worker 82
5.2.4 Der Fill Daemon 85
5.2.5 Der Hit Counter 86
5.2.6 Der Garbage Collector 87
5.2.7 Die Middle-Tier-Komponente 88
5.2.8 Die JDBC-Schnittstelle 89

6 Zusammenfassung und Ausblick 91

Anhang A – Weitere Datenbank-Cache Implementierungen 95

Anhang B – XML-Datei zum Filmdatenbank-Beispiel 99

Anhang C – EBNF-Notation der verwendeten SQL-Syntax 105

Anhang D – Einrichten des CBCS 109

Literaturverzeichnis 113

Abbildungsverzeichnis

1.1 Mehrschichtarchitektur ... 2
1.2 Mehrschichtarchitektur mit Datenbank-Caching ... 3

2.1 Beispiel für Cache Groups ... 6
2.2 Funktionsweise von bereichsvollständigen Füllspalten ... 9
2.3 Beispiel für den Einsatz von RCCs ... 11
2.4 Cache-Group-Föderation ... 13
2.5 Existenztest in den Quellspalten eingehender RCCs ... 15
2.6 Beispiel für negatives Caching ... 16
2.7 Kontrolltabellen mit RCCs auf die Füllspalten ... 17
2.8 Homogener RCC-Zyklus ... 18
2.9 Heterogener RCC-Zyklus ... 19
2.10 Induzierte Bereichsvollständigkeit ... 21

3.1 WDBAccel Architektur und Prozessfluss ... 25
3.2 Beispiel für semantische Regionen ... 26
3.3 Materialisierte Sicht und Kontrolltabelle ... 28
3.4 SQL Erweiterung zur Sicherung der Konsistenz ... 29
3.5 Definition von Cache-Tabellen in DBCache ... 30
3.6 Aufbau des Janus-Plans ... 31
3.7 Insert-Anweisung des Cache-Daemon ... 33

4.1 Komponenten des CBCS ... 38
4.2 Objekt-Typen der internen Datenstruktur ... 42
4.3 Beispiel für erlaubte Werte in einer Füllwert-Tabelle ... 44
4.4 Syntax der unterstützten SQL-Grammatik ... 46
4.5 Verankerung der Cache-Tabellen ... 48
4.6 Aufbau der Existenz-Anfragen ... 48
4.7 Beispiel für die Generierung der Insert-Anweisungen ... 53
4.8 Generierungsreihenfolge der Insert-Anweisungen ... 55
4.9 Aufbau der generierten Insert-Anweisungen ... 56
4.10 Einteilung der Cache-Tabellen in atomare Zonen ... 58
4.11 Topologische Sortierung der atomaren Zonen ... 58
4.12 Einteilung einer Cache Group mit Zyklus in atomare Zonen ... 59
4.13 Beispiel für die Generierung der Update-Anweisungen ... 62
4.14 Aufbau der generierten Update-Anweisungen ... 63
4.15 Beispiel für die Generierung der Delete-Anweisungen ... 67
4.16 Aufbau der generierten Delete-Anweisungen ... 68
4.17 Generierungsreihenfolge der Delete-Anweisungen ... 68
4.18 Zyklenbehandlung während des Löschvorgangs ... 69
4.19 Cache-Group mit mehreren Zyklen ... 69
4.20 Problem der konkurrierenden Threads ... 70

5.1 Die wichtigsten Java-Pakete des CBCS ... 74
5.2 Struktur der Cache Group in XML-Notation ... 76
5.3 Interface der Statement-Factory-Klasse ... 77
5.4 Einrichten des föderierten Datenbanksystems ... 78
5.5 Generierung der Insert-Anweisungen in Pseudo-Code ... 80
5.6 Implementierung der Warteschlangen in Pseudo-Code ... 82
5.7 Erstellte Baumstruktur des SQL-Parsers ... 83
5.8 Implementierung des Fill Daemon in Pseudo-Code ... 85
5.9 Implementierung des Hit Counter in Pseudo-Code ... 86
5.10 Implementierung des Garbage Collector in Pseudo-Code ... 87
5.11 Schnittstellen der Middle-Tier-Komponente ... 88
5.12 Klassenstruktur der JDBC-Schnittstelle ... 89

A.1 Code-Transformation lesender und schreibender Zugriffe ... 96

D.1 Ausgabe der verfügbaren Eingabeparameter ... 110
D.2 Inhalt der CBCS-Konfigurationsdatei ... 110
D.3 Aufruf des Help-Befehls in der Kommandozeile des CBCS ... 112

1 Motivation

Das Internet erfreut sich immer größerer Beliebtheit. Immer mehr Menschen nutzen dieses Medium zur Informationsrecherche, oder um zum Beispiel ihre Bestellungen und Bankgeschäfte bequem von zu Hause zu erledigen. Auch Unternehmen haben das Internet für sich entdeckt und sehen darin eine Chance, ihre Kunden auf eine völlig neue Art und Weise anzusprechen und ihnen Produkte anzubieten. Mit Hilfe von so genannten Web-Anwendungen können Web-Seiten für jeden Kunden individuell gestaltet und für ihn interessante Produkte angeboten werden. Registrierte Benutzer erhalten oftmals sogar die Möglichkeit, die Informationen einer Internetseite ihren persönlichen Bedürfnissen anzupassen. Web-Anwendungen erfreuen sich daher immer größerer Beliebtheit bei den Internetbenutzern und werden dementsprechend häufig besucht. Um den Erwartungen der Benutzer gerecht zu werden, sollte eine moderne Web-Anwendung folgende Anforderungen erfüllen:

- Kurze Reaktionszeit
- Interaktive Bedienbarkeit
- Informationsaktualität
- Anpassung an persönliche Bedürfnisse
- Hohe Verfügbarkeit des Dienstes

Die Erfüllung dieser Anforderungen stellt die Anbieter von Web-Anwendungen vor immer neue Herausforderungen. Um die große Menge an Daten zu verwalten, werden heutzutage zentrale Datenbanksysteme eingesetzt. Die Internetseiten werden durch eine Anwendungslogik dynamisch generiert, wobei diese Anwendungslogik üblicherweise auf einem separaten Server, dem Anwendungsserver, zwischen Benutzer und Datenbanksystem angesiedelt ist (vgl. Abbildung 1.1). Ruft nun der Benutzer eine Internetseite auf, so wird eine Anfrage an den Anwendungsserver gestellt. Dieser fordert beim Datenbanksystem die vom Benutzer gewünschten Informationen an und erstellt daraus eine Internetseite, die wieder an den Benutzer zurück geschickt wird. Das zentrale Datenbanksystem wird üblicherweise von mehreren Anwendungsservern angesprochen, welche wiederum von vielen Benutzern angesprochen werden. Abbildung 1.1 zeigt eine typische Mehrschichtarchitektur mit Benutzern, die über eine Menge von Anwendungsservern auf ein Datenbanksystem zugreifen.

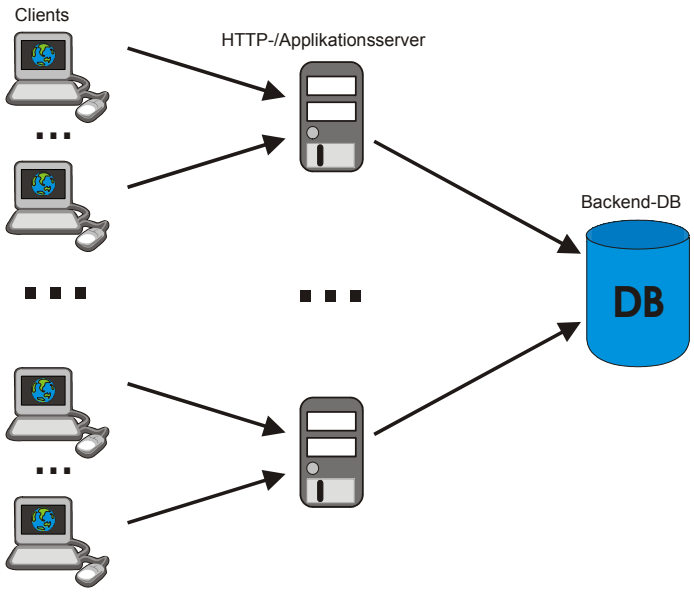


Abbildung 1.1– Mehrschichtarchitektur

Aus Abbildung 1.1 ist leicht ersichtlich, dass das zentrale Datenbanksystem (Backend-DB) unter hoher Auslastung sehr schnell zum Flaschenhals in dieser Architektur wird. In einem Versuch von [LZWM04] mit dem TPC-W Benchmark [TPCW05] und 2000 emulierten Benutzern zeigte sich, dass die Antwortzeiten von einigen Millisekunden, bei geringer Last, auf mehrere Sekunden unter Volllast ansteigen. Solche Antwortzeiten führen dazu, dass aus Sicht des Benutzers sich das „World Wide Web“ in ein „World Wide *Wait*“ [Moha04] verwandelt.

Es gibt jedoch verschiedene Möglichkeiten und Ansätze, um den oben genannten Flaschenhals zu verhindern. Eine der einfachsten Lösungen ist die Replikation des Backend-Datenbanksystems. Dadurch können die Anfragen von den Anwendungsservern auf mehrere Datenbanksysteme mit identischen Daten verteilt werden. In [PlAl04] wird ein System namens Ganymed vorgestellt, welches aus einem Scheduler und mehreren Datenbanksystemen besteht. Der Scheduler verteilt die Last der ankommenden Anfragen gleichmäßig auf die einzelnen Datenbankensysteme. Ein Datenbanksystem übernimmt dabei die Rolle des Masters, während die restlichen Systeme als Slaves arbeiten. Der Master bekommt vom Scheduler nur die Änderungsoperationen (Insert, Update, Delete) zugeteilt, während die Slaves nur Leseoperationen erhalten. Die Daten auf den Slaves bleiben durch Schnappschüsse des Masters immer auf dem aktuellen Stand.

Durch die Replikation des Datenbanksystems erhöht sich zusätzlich die Verfügbarkeit der Daten, da alle Daten redundant vorhanden sind. Der größte Nachteil dieses Verfahrens sind jedoch die Kosten. Im Allgemeinen kostet ein Datenbanksystem ein Vielfaches von dem, was die Anwendungsserver kosten. Beispielsweise kostet ein IBM eServer xSeries 440 System rund 700.000 US-Dollar [TPCW05], und ist damit rund sechzigmal so Teuer wie ein Anwendungsserver [LZWM04].

Einen anderen Ansatz zur Entlastung des Backend-Datenbanksystems bietet das Caching von Daten. Im Bereich von Web-Seiten mit ausschließlich statischem Inhalt hat man durch das Zwischenspeichern (Caching) von Daten sehr gute Ergebnisse erzielt. Beim Web-Caching werden die vom Benutzer angeforderten Daten auf dem Weg zwischen Client und Web-/Anwendungsserver auf Web-Cache-Servern (auch Proxys genannt) zwischengespeichert. Fordert nun ein Benutzer dieselben Daten wiederholt an oder werden die Daten von einem anderen Benutzer ebenfalls angefordert, so muss die Anfrage nicht bis zum Web-/Anwendungsserver geschickt werden, sondern kann unterwegs vom Web-Cache beantwortet werden. Dadurch erhöht sich die Verfügbarkeit der Daten und die Latenzzeit verringert sich. Ein gut konfigurierter Web-Cache kann rund 70% der Anfragen beantworten, was eine wesentliche Entlastung des Web-/Anwendungsservers bedeutet. Das Caching von statischen Inhalten ist heute ein sehr gut erforschtes Gebiet und soll deshalb an dieser Stelle nicht weiter vertieft werden. Allerdings stößt dieses Verfahren im Bereich von dynamisch generierten Web-Seiten schnell an seine Grenzen. Angenommen, es existieren zwei Web-Seiten, die von der Anwendungslogik generiert wurden. Beide Seiten enthalten eine Liste mit denselben Daten, allerdings in unterschiedlicher Sortierreihenfolge. Für einen Web-Cache sind diese Seiten verschieden, obwohl sie dieselben Daten enthalten.

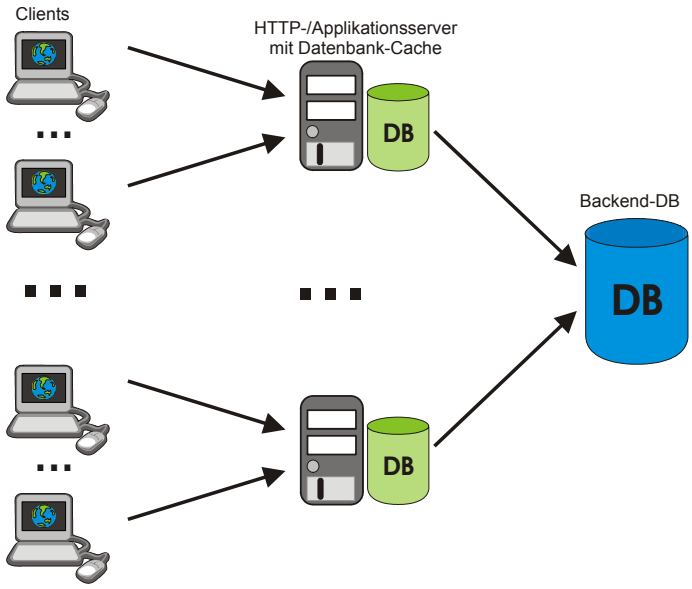


Abbildung 1.2 – Mehrschichtarchitektur mit Datenbank-Caching

Ein Ansatz für das Caching von dynamischen generierten Web-Seiten ist das Datenbank-Caching. Während man sich beim Caching von statischen Web-Seiten auf den Weg zwischen Benutzer und Anwendungsserver konzentriert, liegt beim Datenbank-Caching das Augenmerk auf dem Weg zwischen Anwendungslogik (Anwendungsserver) und Backend-Datenbank. Der Datenbank-Cache (kurz: Cache) befindet sich, wie in Abbildung 1.2 zu sehen, in der Nähe des Anwendungsservers.

Der Cache speichert aus der Backend-Datenbank eine Teilmenge des Datenbestands, der in jüngster Vergangenheit häufig referenziert wurde. Der Anwendungsserver sendet die Anfragen an den Cache. Dieser versucht nun, die Anfrage zuerst mit Hilfe der Daten im Cache zu beantworten. Befinden sich die benötigten Daten nicht im Cache, wird die Anfrage weiter an die Backend-Datenbank geschickt.

In einem Web-Cache werden die Daten üblicherweise auf der Ebene von referenzierten Web-Seiten gespeichert. Beim Datenbank-Caching erfolgt die Speicherung der Daten auf einer tieferen Ebene. Hier werden die Daten (Anfrageergebnisse aus der Backend-Datenbank) zum Beispiel in einer Datenbank gespeichert, die anders als ein Web-Cache eine Anfragebearbeitung mit SQL erlaubt [HäBü04b].

In den folgenden Kapiteln werden wir genauer auf das Datenbank-Caching und die Verwaltung der Daten im Cache eingehen. In Kapitel 2 stellen wir das Constraint-basierte Datenbank-Caching vor. Dazu werden wir die Konzepte und Begriffe dieses Verfahrens ausführlich vorstellen und diskutieren.

Anschließend werden wir in Kapitel 3 weitere Konzepte zum Caching von dynamischen Inhalten, wie beispielsweise dem Query-Result-Caching kennen lernen. Zusätzlich werden wir zwei Datenbank-Cache Prototypen der Firmen Microsoft und IBM erläutern und eine Stärken-Schwächen-Analyse beider Systeme durchführen.

Den von uns entwickelten Prototyp eines Constraint-basierten Datenbank-Cache, sowie dessen Funktionsweise stellen wir in Kapitel 4 vor. Dabei werden wir in diesem Kapitel die einzelnen Komponenten des Prototyps und deren Funktionen auf abstrakter Ebene betrachten. Im Anschluss daran werden wir in Kapitel 5 auf die Implementierung der einzelnen Komponenten des Prototyps mit Hilfe von Programmcode-Fragmenten eingehen.

Nach einem kurzen Ausblick und dem Aufzeigen von weiteren Optimierungsmöglichkeiten fassen wir die Ergebnisse der Arbeit in Kapitel 6 noch einmal kurz zusammen.

2 Einführung in das Constraint-basierte Datenbank-Caching

Für das Caching von dynamischen Web-Seiten wurden neue Konzepte entwickelt, wie beispielsweise das Query-Result-Caching oder das Constraint-basierte Datenbank-Caching. In diesem Kapitel konzentrieren wir uns auf das Constraint-basierte Datenbank-Caching und werden die wichtigsten Begriffe dieses Konzepts erläutern. Das Konzept des Query-Result-Caching erläutern wir ausführlich in Kapitel 3.

Die hier vorgestellten Begriffe dienen als Einstieg in die Welt der Constraint-basierten Caching-Verfahren und bilden die Grundlage für die folgenden Kapitel. Alle hier vorgestellten Bezeichnungen und Definitionen finden sich auch in [HäBü04a].

2.1 Die Idee

Wie bereits erwähnt, sind die Anforderungen von Benutzern an Web-Anwendungen sehr hoch. Um diese Ansprüche zu erfüllen, ist es notwendig, einen Teil der Daten aus dem zentralen Datenbanksystem (Backend-Datenbank) in der Nähe der Anwendungslogik zu replizieren. Die Daten werden in einem Datenbanksystem, das im Folgenden als Cache bezeichnet wird, gespeichert. Der Cache kann sich auf demselben Rechner wie die Anwendungslogik oder auf einem Rechner in unmittelbarer Nähe der Anwendungslogik befinden. Die Speicherung der Daten sollte aus Sicht des Benutzers völlig transparent geschehen, d. h. der Benutzer sollte die Existenz des Cache nicht bemerken. Weiterhin sollten keine Änderungen an den Web-Anwendungen vorgenommen werden müssen. Eine weitere Anforderung an den Cache ist die Anpassung an das Anfrageverhalten der Benutzer, d. h., wenn eine Anfrage nicht durch den Cache-Inhalt beantwortet werden kann, so muss der Cache selbstständig die fehlenden Daten von der Backend-Datenbank abfragen und für zukünftige Anfragen einlagern.

Das Konzept des Constraint-basierten Datenbank-Caching erfüllt die beiden eben genannten Anforderungen. Wie der Name bereits andeutet, handelt es sich hierbei um ein Konzept, welches Bedingungen oder Einschränkungen benutzt, um einen transparenten, adaptiven Datenbank-Cache zu realisieren.

Der Inhalt des Cache wird durch Cache-Constraints spezifiziert. Diese Cache-

Constraints können den Cache-Inhalt dynamisch an die vorhandenen Gegebenheiten anpassen. Zusätzlich bilden sie einen Mechanismus, der das Füllen des Cache mit Datensätzen steuert. Damit Anfragen korrekt beantwortet werden können, muss der Cache dafür sorgen, dass alle Cache-Constraints eingehalten werden. Wir werden in den Abschnitten 2.1.2 und 2.1.3 genauer auf die einzelnen Cache-Constraints eingehen.

Beim Constraint-basierten Datenbank-Caching wird ein Ausschnitt der Backend-Datenbank im Cache abgebildet. Die Struktur der abgebildeten Tabellen bzw. des abgebildeten Ausschnitts wird in der Form von so genannten *Cache Groups* beschrieben.

2.1.1 Cache Groups

Eine Cache Group wird durch eine Menge von Cache-Tabellen und Cache-Constraints spezifiziert. Ein Beispiel für Cache Groups zeigt die in Abbildung 2.1 dargestellte Filmdatenbank.

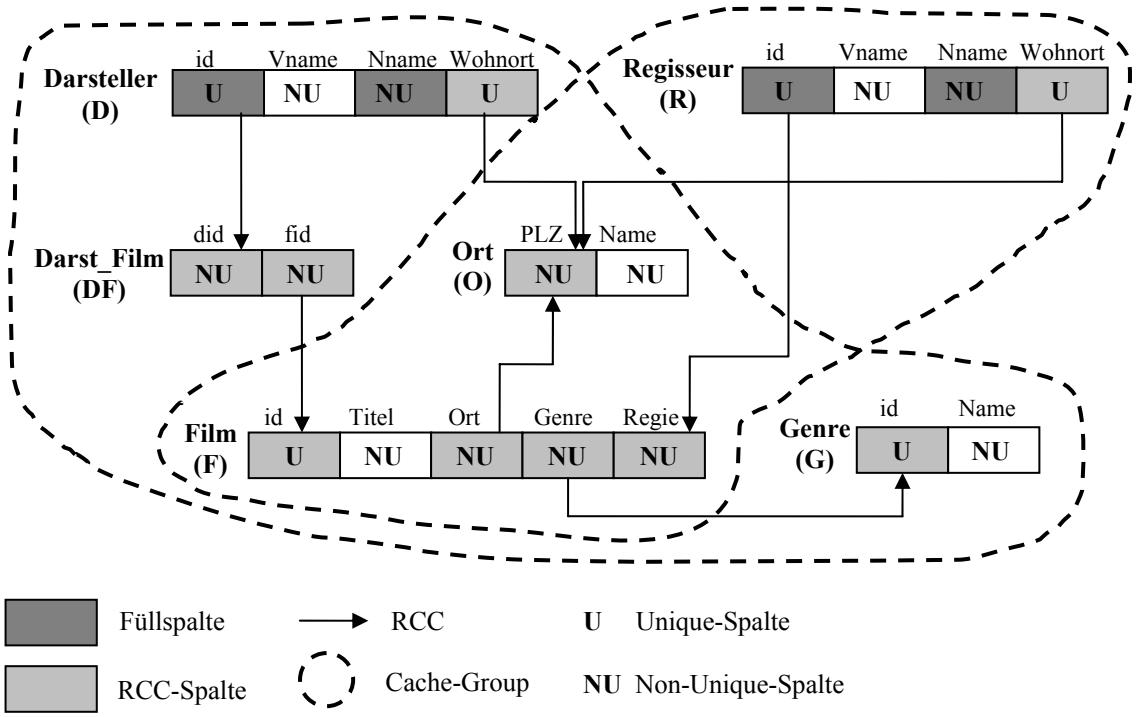


Abbildung 2.1 – Beispiel für Cache Groups

Die einzelnen Cache Groups mit ihren Cache-Tabellen sind durch gestrichelte Linien abgegrenzt. Die dunkelgrau unterlegten Felder kennzeichnen Füllspalten, einen Cache-Constraint-Typ, den wir in Abschnitt 2.1.2 erklären. Die hellgrauen und mit Pfeilen verbundenen Felder stellen referenzielle Cache-Constraints (RCC) dar. RCCs werden dazu verwendet, um die Abhängigkeiten zwischen den Inhalten einzelner Cache-Tabellen zu beschreiben. Sie sind ein weiterer Cache-Constraint-Typ, der in Abschnitt 2.1.3 vorgestellt wird.

Die Cache-Tabellen stimmen mit den Tabellen in der Backend-Datenbank überein. Jedoch existierten nicht alle Tabellen der Backend-Datenbank auch im Cache. Im Cache werden nur die Tabellen abgebildet, die häufig referenziert werden. Bei der Einlagerung von Datensätzen aus der Backend-Datenbank in den Cache werden diese vollständig gespeichert, d. h., die Werte aus allen Spalten werden in den Cache übernommen. Der Cache enthält so stets eine Teilmenge der in der Backend-Datenbank vorhandenen Datensätze.

Mit Hilfe von Cache-Constraints werden nicht nur die Inhalte einer Cache Group beschrieben, sondern auch die Abhängigkeiten zwischen den Inhalten einzelner Cache-Tabellen. Die Cache-Constraints definieren somit die gültigen Zustände des Cache. Dadurch kann für eine gegebene Anfrage leicht entschieden werden, ob sie mit den Datensätzen aus dem Cache beantwortet werden kann, oder ob sie zur Auswertung an die Backend-Datenbank geschickt werden muss.

Die Cache-Verwaltung ist dafür zuständig, dass alle definierten Cache-Constraints durchgesetzt werden, denn nur so kann *Prädikatvollständigkeit* gewährleistet werden.

> **Definition 1 (Prädikatvollständigkeit):** Eine Menge von Tabellen heißt prädikatvollständig in Bezug auf ein Prädikat p, wenn sie alle Datensätze beinhaltet, die zur Auswertung von p benötigt werden.

Die Menge aller Datensätze, die zur Auswertung eines Prädikats p benötigt werden, nennen wir *Extension*.

Im Folgenden bezeichnen wir alle Tabellen im Cache mit Großbuchstaben ($S, T, \ldots$) und die zugehörigen Spalten mit Kleinbuchstaben ($a, b, \ldots$). Wenn eine Zuordnung von Spalten zu Tabellen notwendig ist, so wird dem Namen der Spalte a der Name der Tabelle S vorangestellt, in der a enthalten ist ($S.a$). Die Tabellen und Spalten der Backend-Datenbank erhalten zusätzlich einen Index B ($S_B, a_B, T_B.b, \ldots$).

Der einfachste Typ von Cache Groups basiert auf Gleichheitsprädikaten. Wir betrachten dazu einfache Anfragen, die nur aus Projektionen, Selektionen und Joins (PSJ-Anfrage) zusammengesetzt sind. Eine Spalte $S.a$ einer PSJ-Anfrage kann durch ein Gleichheitsprädikat auf einen bestimmten Wert w fixiert werden ($S.a = w$). Weiterhin können durch Gleichheitsprädikate Equijoins zwischen zwei Spalten $S.a$ und $T.b$ spezifiziert werden ($S.a = T.b$). Cache Groups auf der Basis von Gleichheitsprädikaten sollen Anfragen unterstützen, die die Form einer Kette $T_1.s_1 = w \wedge T_1.r_1 = T_2.s_2 \wedge \ldots$ besitzen. Damit solche Prädikate vom Cache ausgewertet werden können, müssen ihre Extensionen vollständig im Cache gehalten werden. Durch den Einsatz von *Füllspalten* und *referenziellen Cache-Constraints* wird die Vollständigkeit der Extension eines Prädikats gewährleistet. In den folgenden beiden Abschnitten werden wir die Aufgaben von Füllspalten und Referenziellen Cache-Constraints ausführlich erläutern.

2.1.2 Füllspalten

Eine Füllspalte ist eine spezielle Spalte in einer Cache Group, die für das Füllen der Cache Group mit Datensätzen aus der Backend-Datenbank verantwortlich ist. Zu jeder Füllspalte existiert zusätzlich eine Liste R, die alle Werte w spezifiziert, die nicht im Cache gespeichert werden sollen. Es ist also möglich, Werte, die eine geringe Selektivität in der Backend-Datenbank besitzen, von der Einlagerung im Cache auszuschließen. Dadurch kann verhindert werden, dass diese Werte, die eine große Anzahl von Datensätzen betreffen, nicht den wertvollen Speicherplatz des Cache belegen. Alle Anfragen, die sich auf einen Wert in der Liste R beziehen, können folglich nicht durch den Cache beantwortet werden und müssen immer an die Backend-Datenbank weitergeleitet werden. Im Folgenden gehen wir davon aus, dass die Liste R leer ist, d. h., alle Werte w können in Cache eingelagert werden.

Wollen wir nun eine Anfrage mit Prädikat $S.a = w$ im Cache beantworten, wobei $S.a$ eine Füllspalte ist und w ein Wert, so müssen wir zuerst überprüfen, ob die benötigten Datensätze im Cache vorhanden sind. Dazu führen wir einen einfachen Existenztest auf der Spalte $S.a$ durch. Diese Vorgehensweise bezeichnen wir als *Sondierung*.

Liefert die Sondierung ein positives Ergebnis, so wissen wir, dass mindestens ein Datensatz im Cache gefunden wurde, der das Prädikat $S.a = w$ erfüllt. Dennoch können wir die Anfrage nicht korrekt im Cache auswerten, da wir keine Aussage darüber machen können, ob alle Datensätze der Backend-Datenbank, die das Prädikat $S.a = w$ erfüllen, auch im Cache eingelagert wurden. Eine Anfrage kann nur dann korrekt ausgewertet werden, wenn der Wert w *wertvollständig* in der Spalte $S.a$ ist.

> **Definition 2 (Wertvollständigkeit):** Ein Wert w heißt wertvollständig (kurz: vollständig) in einer Spalte $S.a$ genau dann, wenn alle Sätze aus $\sigma_{a=w}S_B$ in S enthalten sind.

Die Werte in Unique-Spalten sind immer wertvollständig, da jeder Wert w nur einmal vorkommen darf. Sind alle Werte w wertvollständig in einer Spalte $S.a$, so bezeichnet man die Spalte $S.a$ als *bereichsvollständig*.

> **Definition 3 (Bereichsvollständigkeit):** Eine Cache-Spalte $S.a$ heißt bereichsvollständig genau dann, wenn alle Werte w in $S.a$ wertvollständig sind.

Das vorgestellte Sondierungsverfahren ist nur für bereichsvollständige Spalten sinnvoll. Sobald wir einen Datensatz in der Spalte $S.a$ finden, der w enthält, garantiert uns die Bereichsvollständigkeit, dass alle Datensätze, die w enthalten, im Cache vorhanden sind und die Anfrage korrekt auf dem Cache-Inhalt ausgewertet werden kann.

Mit Hilfe der Sondierung ist die Cache-Verwaltung in der Lage zu entscheiden, ob eine Anfrage im Cache beantwortet werden kann, oder ob sie an die Backend-Datenbank weitergeleitet werden muss.

Die Sondierung wird üblicherweise auf Füllspalten durchgeführt, die dann als Einstiegspunkt in die Cache Group dienen. Die Füllspalten müssen in diesem Fall zusätzlich bereichsvollständig sein. Die Cache-Verwaltung hat, durch entsprechendes Nachladen von Datensätzen, für die Bereichvollständigkeit der Spalten zu sorgen.

Der unvorsichtige Einsatz von bereichsvollständigen Füllspalten führt jedoch zu Problemen, die wir an folgendem Beispiel diskutieren wollen.

$Darsteller_B$	id	Vname	Nname	…
	…	…	…	…
	10	Walter	Maier	
	20	Heinz	Müller	
	30	Günter	Maier	
	40	Walter	Schmidt	
	50	Günter	Müller	
	60	Heinz	Schmidt	
	…	…	…	…

a) Inhalt der Backend-Datenbank

Darsteller	id	Vname	Nname	…
	20	Heinz	Müller	
	50	Günter	Müller	

b) Inhalt des Cache nach dem Füllen

Abbildung 2.2 – Funktionsweise von bereichsvollständigen Füllspalten

In Abbildung 2.2a sehen wir eine Tabelle der Backend-Datenbank mit einigen Datensätzen. Die Spalte *id* sei eine Unique-Spalte, *Vname* und *Nname* seien Non-Unique-Spalten. Zusätzlich definieren wir die Spalten *id* und *Nname* als bereichsvollständige Füllspalten. Der Cache sei am Anfang leer. Eine Anfrage mit Prädikat $id = 20$ kann vom leeren Cache nicht beantwortet werden und wird zur Backend-Datenbank geschickt und dort beantwortet. Da die Spalte *id* eine bereichsvollständige Füllspalte ist, werden anschließend alle Datensätze mit $id = 20$ in den Cache eingelagert (Tabelle Darsteller in Abbildung 2.2b). Die Cache-Verwaltung muss nach jeder Einlagerung von Daten die Gültigkeit der Cache-Constraints überprüfen und gegebenenfalls wieder einen gültigen Cache-Zustand herstellen. Da *id* eine Unique-Spalte ist, ist sie implizit bereichsvollständig, d. h., es müssen keine weiteren Datensätze nachgeladen werden. Für die Non-Unique-Spalte *Nname* sieht es anders aus. Diese Spalte muss wieder bereichsvollständig gemacht werden. Würden wir eine weitere Anfrage an den Cache, mit Prädikat $Nname = 'Müller'$, stellen, so würde die Sondierung ein positives Ergebnis liefern. Die Anfrage würde also vom Cache beantwortet werden. Das Ergebnis der Anfrage ist jedoch nicht korrekt, da nicht alle Datensätze im Cache vorhanden sind, die das Prädikat $Nname = 'Müller'$ erfüllen.

Die Cache-Verwaltung muss daher eine Anfrage an die Backend-Datenbank stellen und alle Datensätze anfordern, die das Prädikat $Nname = 'Müller'$ erfüllen, damit die Spalte *Nname* wieder bereichsvollständig ist.

In unserem Beispiel ist nun ein zusätzlicher Datensatz im Cache gespeichert worden. Für diesen Datensatz muss nun die Spalte *id* wieder bereichsvollständig gemacht werden. Aber wie bereits erwähnt, ist die Spalte *id* durch ihre Unique-Eigenschaft immer bereichsvollständig. Damit ist das Nachladen von Datensätzen beendet, und der Cache hat wieder einen gültigen Zustand. Der Cache besitzt nun die in Abbildung 2.2b dargestellten Datensätze.

Dieses Beispiel lässt schon erahnen, was passiert, wenn wir statt der Spalte *id* die Spalte *Vname* als bereichsvollständige Füllspalte definieren würden. Das Nachladen von Datensätzen für die Spalte *Nname* verursacht wieder neue Werte in der Spalte *Vname*. Die Cache-Verwaltung muss anschließend die Spalte *Vname* wieder bereichsvollständig machen, was wieder zu neuen Werten in der Spalte *Nname* führt und so fort. Dadurch entsteht ein rekursives Ladeverhalten, das im schlimmsten Fall dazu führt, dass der komplette Inhalt der Backend-Tabelle in den Cache eingelagert wird. Der Grund dafür, dass in dem eben vorgestellten Fall ein rekursiver Ladevorgang entsteht, liegt im Sondierungsverfahren. Liefert die Sondierung ein positives Ergebnis, so können wir die Korrektheit der Anfrage nur dann garantieren, wenn die Spalte bereichsvollständig ist.

In Abschnitt 2.2.1 werden wir ein weiteres Sondierungsverfahren vorstellen, das keine bereichsvollständigen Füllspalten benötigt.

Achtet man allerdings beim Entwurf einer Cache Group darauf, dass nur eine Non-Unique-Spalte als bereichsvollständige Füllspalte definiert wird (Unique-Spalten sind von dieser Regel nicht betroffen), so kann auch für das bereits bekannte Sondierungsverfahren dieser spezielle Fall von rekursivem Ladeverhalten unterbunden werden. In Abschnitt 2.3 werden wir auf die Problematik der rekursiven Ladevorgänge noch etwas genauer eingehen.

2.1.3 Referenzieller Cache-Constraint (RCC)

Durch das Konzept der Füllspalten können wir einfache Anfragen mit Gleichheitsprädikaten korrekt auf dem Cache beantworten. Die Anfragen dürfen sich allerdings nur auf eine Tabelle beziehen und mindestens ein Prädikat muss sich auf eine Füllspalte dieser Tabelle beziehen.

Bei Join-Operationen mit Spalten in anderen Cache-Tabellen können wir die Korrektheit der Ergebnisse nicht garantieren. Betrachten wir dazu den in Abbildung 2.3 dargestellten Ausschnitt aus Abbildung 2.1.

Angenommen wir möchten die Anfrage mit den Prädikaten $D.id = 10$ und $D.id = DF.did$ beantworten. Gehen wir davon aus, dass die Sondierung für das Prädikat $D.id = 10$ ein positives Ergebnis geliefert hat. Wir wissen, dass $D.id$ eine bereichs-

vollständige Füllspalte ist, somit muss der Wert $D.id = 10$ vollständig sein. Da die Spalte $DF.did$ eine Non-Unique-Spalte ist und nicht als Füllspalte definiert wurde, können wir keine Aussage über die Vollständigkeit der Werte in dieser Spalte machen. Eine Auswertung der Anfrage auf dem Cache kann so zu falschen Ergebnissen führen.

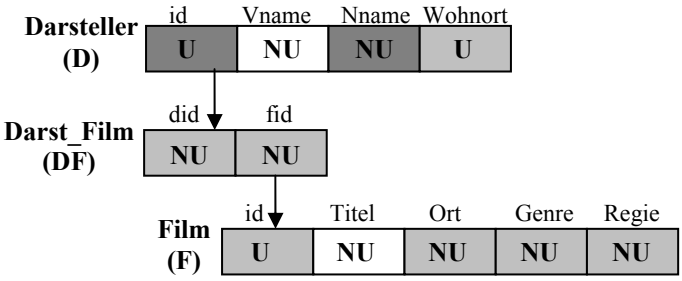


Abbildung 2.3 – Beispiel für den Einsatz von RCCs

Damit wir Join-Operationen ebenfalls im Cache beantworten können, benötigen wir die *referenziellen Cache-Constraints (RCC)*. Ein RCC definiert zwischen zwei Spalten der Cache-Tabellen eine Wertebeziehung und gewährleistet, dass für jeden Wert in der Quellspalte des RCC die entsprechenden Join-Partner in der Zielspalte des RCC ebenfalls im Cache vorhanden sind.

> **Definition 4 (Referenzieller Cache-Constraint, RCC):** Ein referenzieller Cache-Constraint $S.a \rightarrow T.b$ von einer Quellspalte $S.a$ zu einer Zielspalte $T.b$ ist genau dann erfüllt, wenn alle Werte w aus $S.a$ vollständig in $T.b$ sind.

Die Cache-Verwaltung muss, analog zu den Füllspalten, auch für die Einhaltung der RCCs sorgen und gegebenenfalls Datensätze aus der Backend-Datenbank anfordern und im Cache einlagern.

2.1.4 Anfragebearbeitung im Cache

Mit den nun bekannten Cache-Constraint-Typen können wir Anfragen der Form $T_1.s_1 = w \wedge T_1.r_1 = T_2.s_2 \wedge \ldots \wedge T_{n-1}.r_{n-1} = T_n.s_n$ bearbeiten und überprüfen, ob wir sie mit dem Inhalt des Cache beantworten können.

Durch die Cache-Constraints entsteht zusätzlicher Aufwand für die Cache-Verwaltung. Der Einsatz von Cache-Constraints ist daher nur sinnvoll, wenn möglichst viele Anfragen mit Hilfe des Cache-Inhalts beantwortet werden können und somit der Mehraufwand kompensiert wird.

Beim Constraint-basierten Caching werden zwei Schritte durchgeführt, um eine Anfrage zu beantworten. Zuerst wird mit Hilfe der Sondierung und der RCCs festgestellt, ob der Cache die benötigten Datensätze enthält. Zusätzlich werden in diesem Schritt alle

Teile der Anfrage bestimmt, die im Cache beantwortet werden können. Die Erstellung des Ausführungsplans geschieht im zweiten Schritt. Da die Cache-Tabellen identisch zu den Backend-Tabellen sind, kann der Ausführungsplan auf herkömmliche Art und Weise erstellt werden. Der aus Schritt zwei resultierende Plan kann sowohl Cache-Tabellen als auch Backend-Tabellen beinhalten.

Der erste Schritt stellt sicher, dass einzelne Backend-Tabellen korrekt durch die entsprechenden Cache-Tabellen ersetzt werden und damit das Anfrageergebnis korrekt ist.

Die Überprüfung im ersten Schritt wird für jede Anfrage durchgeführt und sollte daher effizient implementiert sein. Dies kann beispielsweise durch möglichst einfache Operationen auf dem Cache-Inhalt, d. h. sowohl auf den Daten als auch auf den Metadaten, geschehen.

In unserem Fall können wir die Gleichheitsprädikate überprüfen, indem wir einfache Testanfragen an den Cache stellen.

Für die Anfrage $T_1.s_1 = w \wedge T_1.r_1 = T_2.s_2 \wedge \ldots \wedge T_{n-1}.r_{n-1} = T_n.s_n$ treffen wir an dieser Stelle folgende Annahmen:

(A1) Die Spalte $T_1.s_1$ ist die einzige Spalte in T_1, die als bereichsvollständige Füllspalte definiert wurde (dadurch verhindern wir ein mögliches rekursives Ladeverhalten).

(A2) Es existieren RCCs $T_i.r_i \rightarrow T_{i+1}.s_{i+1}$ mit $i \in 1,\ldots,n-1$

Für unser Beispiel müssen wir nun überprüfen ob das Prädikat $T_1.s_1 = w$ erfüllt ist. Dazu genügt es folgendes SQL-Prädikat auszuwerten:

```
EXISTS(SELECT *
       FROM T1
       WHERE T1.s1 = w)
```

Wenn dieses Prädikat für mindestens einen Datensatz in T_1 wahr ist, dann ist der Wert w vollständig, da $T_1.s_1$ bereichsvollständige Füllspalte ist. Die Spalte $T_1.s_1$ wird dann als *Einstiegspunkt* in die Cache Group bezeichnet.

Wenn wir für eine Anfrage einen Einstiegspunkt gefunden haben, können wir alle Equijoins entlang der ausgehenden RCCs dieser Tabelle korrekt im Cache auswerten. Auf unser Beispiel bezogen bedeutet dies: Existiert ein RCC $T_1.r_1 \rightarrow T_2.s_2$ und ist $T_1.s_1$ ein Einstiegspunkt für $T_1.s_1 = w$, dann können wir auch das erweiterte Prädikat $T_1.s_1 = w \wedge T_1.r_1 = T_2.s_2$ korrekt im Cache auswerten.

Ohne diese Beschränkung des RCCs in der Quelltabelle auf eine bestimmte Prädikatextension ist es nicht möglich, einen Equijoin entlang eines RCC durchzuführen. Die Ursache hierfür liegt darin, dass bei einem unbeschränkten Join alle Datensätze, die sich nicht im Cache befinden, jedoch im Backend vorhanden sind, fehlen und somit die Anfrage ein falsches Ergebnis liefert. Ist die Beschränkung auf eine Prädikatextension erfüllt und können wir einen RCC anwenden, so bezeichnen wir

dies als *Verankerung* des RCC im Cache; entsprechend heißt die Quelltabelle des RCC in diesem Fall verankert.

Ist die Spalte $T_1.s_1$ ein Einstiegspunkt, so wird dadurch die Tabelle T_1 verankert und alle von ihr ausgehenden RCCs. Die Anwendung des RCC $T_1.r_1 \rightarrow T_2.s_2$ für einen Equijoin $T_1.r_1 = T_2.s_2$ verankert wiederum die Tabelle T_2, von der nun weitere RCCs angewendet werden können.

Durch die Annahmen (A1) und (A2) können wir durch eine einzige Existenzanfrage entscheiden, ob die gesamte Anfrage durch den Cache beantwortet werden kann oder nicht.

2.1.5 Cache-Group-Föderation

In Abschnitt 2.1.1 sind wir bereits auf den Begriff Cache Group eingegangen. Wir wissen mittlerweile, dass eine Cache Group durch eine Menge von Cache-Tabellen und Cache-Constraints wie Füllspalten und RCCs spezifiziert wird. Abbildung 2.4 veranschaulicht dies nochmals graphisch.

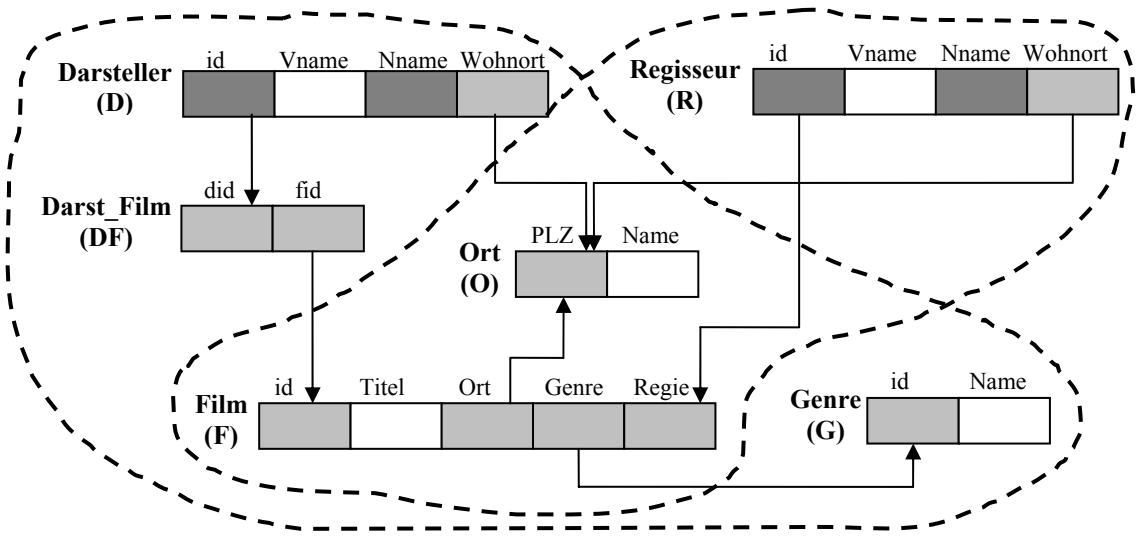


Abbildung 2.4 – Cache-Group-Föderation

Typischerweise wird für jeden Anwendungsfall eine eigene Cache Group erzeugt. In Abbildung 2.4 sind zwei Cache Groups dargestellt. Jede Cache Group ist durch eine gestrichelte Linie markiert. Es kann natürlich vorkommen, dass es Tabellen gibt, die in mehreren Cache Groups enthalten sind. Solche Überlappungen treten in der Praxis häufig auf. Würde man die gemeinsam genutzten Tabellen für jede Cache Group anlegen, so besteht die Gefahr, dass die Speicherung der Daten für den Benutzer und die Anwendung nicht mehr Transparent sind. Daher werden die gemeinsam genutzten Tabellen nicht mehrfach im Cache gespeichert. Die überlappenden Tabellen werden zu einer *Cache-Group-Föderation* zusammengefasst.

Dies führt zu neuen Herausforderungen, die den Entwurf von Cache Groups betreffen. In Cache-Group-Föderationen können sich die einzelnen Cache Groups beim Einlagern von Datensätzen gegenseitig beeinflussen, was wieder zu rekursivem Ladeverhalten führen kann. Auf diese Problematik werden in Abschnitt 2.3 genauer eingehen.

2.2 Flexible Einstiegspunkte

Wir haben bis jetzt erfahren, wie wir mit Hilfe von Cache-Constraints Anfragen aus dem Cache beantworten können. Eine zentrale Rolle spielen dabei die Einstiegspunkte in die Cache Group, durch die es erst möglich wird, Equijoins zwischen Tabellen auszuwerten. Dabei werden als Einstiegspunkte nur Spalten in Erwägung gezogen, in denen zu jeder Zeit alle Werte vollständig sind. Eine neue und wesentlich flexiblere Methode einen Einstiegspunkt zu finden wird erstmals in [Bühm05a] beschrieben. Dort wird ein vorher nicht beachteter Aspekt von RCCs aufgedeckt, denn die Zielspalten von RCCs können ebenfalls als Einstiegspunkte in Cache Groups verwendet werden.

Gemäß der Definition eines RCC werden zu einem Wert w in einer Quellspalte $S.a$ alle Join-Partner aus der Backend-Tabelle T_B in die Zielspalte $T.b$ geladen, sodass die Bedingung $S.a = T.b$ im Cache ausgewertet werden kann. Die Spalte $T.b$ wird also für den Wert w wertvollständig gemacht.

Betrachtet man nun die ganze Situation aus Sicht der Zielspalte, so bedeutet dies, dass ein Wert w in einer Spalte $T.b$ wertvollständig ist, wenn es mindestens einen RCC gibt, der auf diese Spalte verweist und in dessen Quellspalte der Wert w auftritt. Es reicht also aus, einen Existenztest auf den Quellspalten durchzuführen um die Spalte $T.b$ als Einstiegspunkt in die Cache Group nutzen zu können. In der Spalte $T.b$ können darüber hinaus auch noch Werte existieren, die nicht wertvollständig sind. In diesem Fall ist $T.b$ nicht bereichsvollständig.

2.2.1 Das neue Sondierungsverfahren

Beim neuen Sondierungsverfahren machen wir einen Schritt zurück, um die Spalten für die Sondierung zu finden: Wir laufen vom potenziellen Einstiegspunkt zurück entlang der eingehenden RCCs und führen die Sondierung auf den Quellspalten dieser RCCs durch. Dieses Vorgehen wollen wir an einem Beispiel verdeutlichen.

Das in Abbildung 2.5 dargestellte einfache Beispiel zeigt die Spalte $O.PLZ$, die von zwei RCCs mit den Quellspalten $D.Wohnort$ und $R.Wohnort$ referenziert wird. Wenn wir nun die Spalte $O.PLZ$ als Einstiegspunkt für ein Gleichheitsprädikat $O.PLZ = w$ in betracht ziehen wollen, so müssen wir lediglich eine Sondierung auf den

beiden Quellspalten *D.Wohnort* und *R.Wohnort* vornehmen. Tritt der Wert *w* in einer der beiden Quellspalten auf, so können wir durch *Definition 4* sicher sein, dass der Wert *w* in der Spalte *O.PLZ* wertvollständig ist. Finden wir jedoch den Wert *w* weder in der Spalte *D.Wohnort* noch in *R.Wohnort*, so können wir keine Aussage über die Vollständigkeit von *w* in der Spalte *O.PLZ* machen (es ist dennoch möglich, dass *w* vollständig in *O.PLZ* ist). Das neue Sondierungsverfahren bietet im Gegensatz zum vorher vorgestellten Verfahren (zukünftig als altes Sondierungsverfahren bezeichnet), bei dem das Sondieren direkt in den (bereichsvollständigen) Spalten durchgeführt wurde, viel mehr potenzielle Einstiegspunkte. Jede Zielspalte eines RCCs wird so zum potenziellen Einstiegspunkt in die Cache Group.

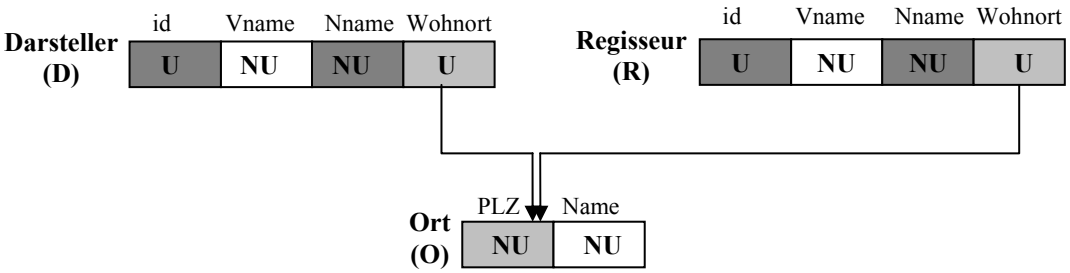


Abbildung 2.5 – Existenztest in den Quellspalten eingehender RCCs

2.2.2 Negatives Datenbank-Caching

Normalerweise werden beim Caching Objekte der Backend-Datenbank im Cache gespeichert. Dabei stehen existierende Objekte im Vordergrund. Werden nun Informationen zwischengespeichert, die das Nicht-Vorhandensein von Objekten in der Backend-Datenbank beschreiben, so nennen wir dies *negatives Caching*.

Der Begriff „negatives Caching“ taucht bereits bei [Andr98] im Zusammenhang mit dem Caching von DNS-Anfragen auf:

> „It is the storage of knowledge that something does not exist, cannot or does not give an answer that we call negative caching.“

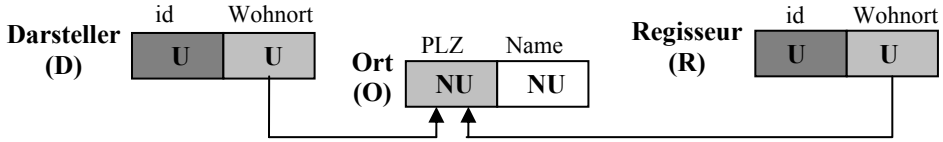
Auch beim Constraint-basierten Caching speichern wir unbewusst Informationen über Datensätze, die in der Backend-Datenbank nicht existieren. Diese Informationen können wir dazu nutzen, noch mehr Anfragen im Cache zu beantworten.

Angenommen, ein Wert *w* einer Spalte *S.a* existiert nicht in der Backend-Tabelle. Folglich können wir diesen Wert auch nicht im Cache zwischenspeichern. Erhalten wir nun eine Anfrage mit dem Prädikat $S.a = w$, so würden wir zuerst eine Sondierung dieses Prädikat im Cache vornehmen. Die Sondierung schlägt fehl und wir schicken die Anfrage zur Backend-Datenbank. Die Backend-Datenbank teilt uns als Ergebnis mit,

dass keine passenden Datensätze gefunden wurden.

Das neue Sondierungsverfahren erlaubt uns auch die Beantwortung solcher Anfragen im Cache, die ein leeres Ergebnis liefern. Durch den Schritt zurück entlang der RCCs machen wir einen Existenztest auf den Quellspalten der RCC. Liefert uns der Existenztest ein positives Ergebnis, so können wir sicher sein, dass in der Zielspalte alle Datensätze mit diesem Wert vollständig sind (auch wenn keine Datensätze existieren). Die Anfrage kann also im Cache korrekt beantwortet werden.

Zum besseren Verständnis werden wir das negative Caching an einem ausführlichen Beispiel erläutern.



Spalte	Vorhandene Werte
D.Wohnort	55743,
R.Wohnort	67292,
O.PLZ	55743, 55595
$O.PLZ_B$	55743, 55595

Abbildung 2.6 – Beispiel für negatives Caching

Betrachten wir die in Abbildung 2.6 dargestellte Cache Group. Wir verwenden eine konkrete Belegung der Spalten, um verschiedene Anfragen mit Gleichheitsprädikaten auf der Spalte $O.PLZ$ durchzuführen:

- $O.PLZ = 55743$: Wir überprüfen alle eingehenden RCCs der Spalte $O.PLZ$ und stellen fest, dass der Wert 55743 in der Quellspalte $D.Wohnort$ des RCC $D.Wohnort \rightarrow O.PLZ$ vorhanden ist. Dadurch ist garantiert, dass der Wert 55743 vollständig in $O.PLZ$ ist: Wir können also $O.PLZ$ als Einstiegspunkt für den Wert 55743 verwenden.
- $O.PLZ = 67292$: Der Wert 67292 existiert nicht in der Spalte $O.PLZ_B$ in der Backend-Datenbank. Dennoch existiert der Wert in der Quellspalte des RCC $R.Wohnort \rightarrow O.PLZ$. Der RCC garantiert wieder, dass $O.PLZ$ bezüglich des Werts 67292 vollständig ist, obwohl er nicht existiert. Analog zum Fall $O.PLZ = 55743$ können wir die Spalte $O.PLZ$ als Einstiegspunkt verwenden. Die Auswertung des Prädikats im Cache liefert ein korrektes Ergebnis, nämlich ein Leeres.
- $O.PLZ = 55595$: Der Wert 55595 existiert zwar in der Spalte $O.PLZ$, jedoch in keiner der eingehenden RCCs dieser Spalte. Wir können keine Aussage darüber machen, ob der Wert 55595 vollständig in $O.PLZ$ ist. Daher kann $O.PLZ$ in diesem Fall nicht als Einstiegspunkt verwendet werden.

2.2.3 Vor- und Nachteile des neuen Sondierungsverfahrens

Der größte Vorteil des neuen Sondierungsverfahrens ist sicherlich die Möglichkeit, mehr Einstiegspunkte in eine Cache Group zu finden. Nicht nur die bereits vom alten Sondierungsverfahren bekannten Einstiegspunkte können genutzt werden, sondern zusätzlich kann jede Spalte, die eine Zielspalte eines RCCs ist, als potenzieller Einstiegspunkt dienen.

Dies führt allerdings auch zu einem der Nachteile dieses Verfahrens. Durch die Sondierung in den Quellspalten der RCCs muss auf jedem eingehenden RCC ein Existenztest durchgeführt werden. Zwar kann beim ersten positiven Test abgebrochen werden, aber bei vielen eingehenden RCCs erhöht sich der Aufwand für jede Anfrage. Die Effizienz des Caching wird also vermindert.

Ein weiteres Problem entsteht durch die Füllspalten. Diese Spalten besitzen keinen RCC, der sie referenziert, und müssten somit im neuen Sondierungsverfahren speziell behandelt werden. Abhilfe schafft hier allerdings die Einführung von Kontrolltabellen [Bühm05b]. Die Kontrolltabellen besitzen nur eine Unique-Spalte und verweisen mit einem RCC auf die Füllspalte.

Abbildung 2.7 zeigt zwei Kontrolltabellen $K1$ und $K2$ mit jeweils einer Unique-Spalte k_1 bzw. k_2. Die Füllspalten *id* und *Nname* werden jeweils durch einen RCC $k_1 \rightarrow id$ bzw. $k_2 \rightarrow Nname$ referenziert.

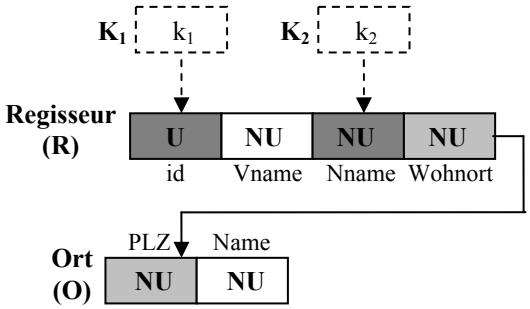


Abbildung 2.7 – Kontrolltabellen mit RCCs auf die Füllspalten

Durch die Einführung von Kontrolltabellen kann nun auch auf Füllspalten das neue Sondierungsverfahren verwenden werden. Die Einführung von Kontrolltabellen beseitigt auch ein weiteres Problem: Bisher wurde keine Trennung zwischen Kandidaten-Werten, die direkt referenziert wurden, und Werten, die auf anderem Weg in die Füllspalte gelangt sind, gemacht: In beiden Fällen wurden die Werte vollständig gemacht (vgl. Abbildung 2.2 auf Seite 9). Dieser Unterschied wird durch die Kontrolltabellen berücksichtigt. Nur die Werte in der Kontrolltabelle werden in der Füllspalte vollständig gemacht, alle anderen Werte der Füllspalte werden nicht vollständig gemacht. Dadurch wird unnötiger Zusatzaufwand bei der Wartung des Cache vermieden. Denn jeder unnötig geladene Datensatz kann durch RCCs zum Laden zusätzlicher Datensätze führen.

2.3 Probleme des Constraint-basierten Datenbank-Caching

Wir haben bereits mehrfach ein Problem von Constraint-basierten Caching-Verfahren erwähnt: rekursives Ladeverhalten, das durch einen schlechten Entwurf von Cache Groups verursacht werden kann. In diesem Abschnitt wollen wir näher auf dieses Verhalten eingehen.

2.3.1 Gute Zyklen, schlechte Zyklen

Damit ein rekursives Ladeverhalten zustande kommt, muss ein Zyklus unter den definierten RCCs einer Cache Group vorhanden sein. Ein Zyklus bedeutet in unserem Fall: Folgen wir, beginnend bei der Tabelle T, den ausgehenden RCCs, dann kommen wir irgendwann wieder an der Tabelle T an.

Man unterscheidet zwei Arten von Zyklen: *homogene* und *heterogene* Zyklen. Nicht alle Zyklen führen automatisch zum rekursiven Laden von Datensätzen. Homogene Zyklen sind *sicher*, das heißt, sie verursachen kein rekursives Ladeverhalten, im Gegensatz zu den meisten heterogenen Zyklen. Wir werden an dieser Stelle nur auf die wichtigsten Aspekte von Zyklen eingehen, eine ausführliche Behandlung dieses Themas findet sich in [HäBü04a].

Homogene Zyklen

Abbildung 2.8 zeigt einen modifizierten Ausschnitt aus Abbildung 2.1. Die Abbildung zeigt einen homogenen Zyklus zwischen den beiden Cache-Tabellen *Darsteller* und *Regisseur*. Die RCCs im Zyklus beginnen und enden alle in Non-Unique-Spalten.

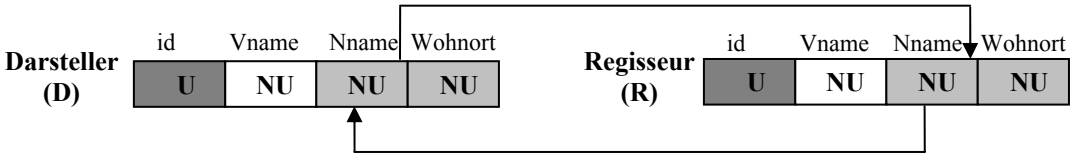


Abbildung 2.8 – Homogener RCC-Zyklus

Gehen wir einmal davon aus, dass eine Anfrage Datensätze benötigt, die momentan nicht in der Spalte $D.id$ existieren und von der Backend-Datenbank angefordert werden müssen. Die Einlagerung dieser Datensätze bildet den Ausgangspunkt für unseren Zyklus. Durch die Einlagerung der Datensätze wurde der RCC $D.Nname \rightarrow R.Nname$ verletzt und die Cache-Verwaltung muss die entsprechenden Werte in der Spalte $R.Nname$ vollständig machen.

Dies führt zur Verletzung des RCC $R.Nname \rightarrow D.Nname$ und der Einlagerung von weiteren Datensätzen. Anschließend beginnt der Zyklus von vorne.

Da sich alle eingehenden und ausgehenden RCCs einer Cache-Tabelle auf dieselbe

Spalte beziehen, endet der Zyklus nach einem Durchlauf. Der Grund dafür ist, dass durch diese Struktur keine neuen Werte in der Initialtabelle (in unserem Fall *D.Nname*) hinzukommen. Nicht alle am Zyklus beteiligten Spalten sind zwingend bereichsvollständig, wie ein Gegenbeispiel in [Bühm05a, HäBü04a] zeigt.

Da ein homogener Zyklus jedoch nicht zu rekursiven Ladeverhalten führt, bezeichnen wir einen homogenen Zyklus als sicher.

Heterogene Zyklen

Der Unterschied zwischen einem homogenen und einem heterogenen Zyklus lässt sich sehr leicht bei einem Vergleich der Abbildung 2.8 mit Abbildung 2.9 erkennen. Bei einem heterogenen Zyklus sind die, am Zyklus beteiligten, eingehenden und ausgehenden RCCs einer Cache-Tabelle auf unterschiedlichen Spalten der Tabelle definiert.

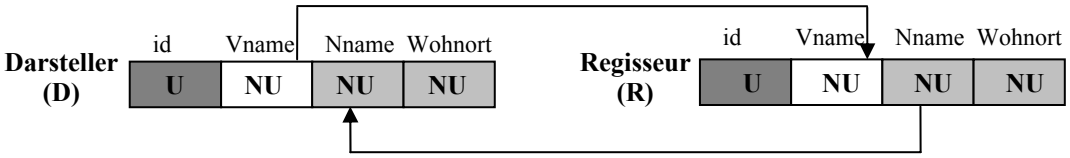


Abbildung 2.9 – Heterogener RCC-Zyklus

Das in Abbildung 2.9 dargestellte Beispiel zeigt, dass von *D.Vname* ein RCC ($D.Vname \rightarrow R.Vname$) ausgeht. Der eingehende RCC auf der Tabelle *Darsteller* ist jedoch auf der Spalte *D.Nname* definiert.

Gehen wir davon aus, dass die Spalte *D.id* wieder der Ausgangspunkt für unseren Zyklus ist. Der RCC $D.Vname \rightarrow R.Vname$ wurde durch die Einlagerung von Datensätzen verletzt. Die Cache-Verwaltung muss wieder einen gültigen Cache-Zustand herstellen und lagert die entsprechenden Datensätze in der Tabelle *Regisseur* ein. Dies führt dazu, dass der RCC $R.Nname \rightarrow D.Nname$ verletzt wird. Durch die Einlagerung der fehlenden Datensätze in *Darsteller*, gelangen neue Werte in die Spalte *D.Vname* und der Zyklus beginnt von vorne.

Ein heterogener Zyklus ist im Allgemeinen nicht sicher und kann zu rekursiven Ladeverhalten führen, was zum Laden aller Datensätze der im Zyklus beteiligten Tabellen führen kann.

2.3.2 Verwendung mehrerer Non-Unique-Füllspalten

Auch Füllspalten können zu unerwünschtem Ladeverhalten führen. Erinnern wir uns an Abbildung 2.2 auf Seite 9: Dort führte die Definition von zwei Non-Unique-Füllspalten zu einem rekursiven Ladeverhalten. Um dieses Problem zu vermeiden, darf, beim Einsatz des alten Sondierungsverfahrens, in einer Cache-Tabelle maximal eine Non-

Unique-Spalte als Füllspalte definiert werden. Die Anzahl der Unique-Füllspalten ist beliebig.

Damit haben wir sichergestellt, dass durch Füllspalten kein rekursives Laden verursacht wird. Im Falle einer Cache-Group-Föderation kann dieses Szenario jedoch durchaus wieder auftreten. Überlappen sich in einer Cache-Group-Föderation Tabellen, auf denen Füllspalten definiert sind, so dürfen auch diese überlappenden Tabellen maximal eine Non-Unique-Füllspalte besitzen.

Beim neuen Sondierungsverfahren dürfen mehrere Non-Unique-Spalten als Füllspalten spezifiziert werden, da die Spalten, auf denen eine Sondierung durchgeführt wird, nicht bereichsvollständig sein müssen.

2.3.3 Optimierungs-RCCs

Beim Entwurf einer Cache Group und der Festlegung der RCCs kann man sich folgende Fragen stellen:

- Ausgehend von den definierten RCCs, welche RCCs gelten darüber hinaus?
- Welche RCCs in einer Cache Group tragen nicht zum Füllen der Cache Group bei, welche sind also redundant?

Wir haben beim neuen Sondierungsverfahren (Abschnitt 2.2.1) gesehen, dass die Zielspalten von RCCs als Einstiegspunkte in die Cache Group dienen können. Eine Antwort der ersten Frage würde also zu weiteren potenziellen Einstiegspunkten führen. Gleichzeitig würden dadurch auch weitere Join-Möglichkeiten zwischen Cache-Tabellen entstehen.

Eine Antwort auf die zweite Frage bietet Vorteile in der Wartung von Cache Groups. Beim Einlagern von Werten müssen diese RCCs nicht überprüft werden, um einen gültigen Cache-Zustand zu garantieren. Ebenfalls kann durch das Weglassen von redundanten RCCs das neue Sondierungsverfahren effizienter betrieben werden (unnötige RCCs fallen weg und müssen bei Anfragen nicht überprüft werden). Allerdings existiert momentan noch kein Algorithmus zum Auffinden von *Optimierungs-RCCs*.

Durch das Auffinden von redundanten RCCs und dem anschließenden Markieren dieser RCCs ist es möglich, induziert bereichsvollständige Spalten zu finden. In Abbildung 2.10 ist die Spalte $DF.did$ *induziert bereichsvollständig*. Da der RCC $D.id \rightarrow DF.did$ der einzige RCC ist, der dazu führt, dass die Tabelle DF mit Datensätzen gefüllt wird und da $D.id$ bereichsvollständig ist (Unique-Spalte) wissen wir, dass auch die Spalte $DF.did$ bereichsvollständig ist (induziert bereichsvollständig).

> **Definition 5 (Induzierte Bereichsvollständigkeit):** Eine Spalte $S.a$ heißt induziert bereichsvollständig, wenn sie nicht durch direkte Spezifikation bereichsvollständig ist.

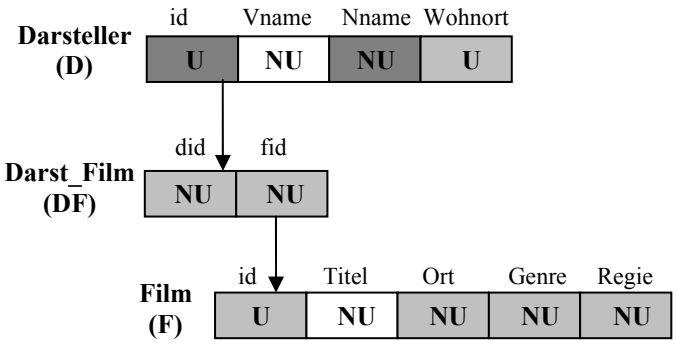


Abbildung 2.10 – Induzierte Bereichsvollständigkeit

Würden weitere RCCs von anderen Tabellen ebenfalls in der Spalte *DF.did* enden, so würde die induzierte Bereichsvollständigkeit weiterhin gelten. Eingehende RCCs auf anderen Spalten von *DF* würden jedoch diese induzierte Bereichsvollständigkeit zerstören. Dies lässt sich ganz einfach veranschaulichen: Angenommen, es existiert ein weiterer RCC, der auf der Spalte *DF.fid* endet. Durch diesen RCC wird beispielsweise ein Datensatz mit einem neuen, vorher noch nicht vorhandenen Wert in *DF.did* eingelagert. Alle anderen Datensätze mit diesem neuen Wert in *DF.did* befinden sich allerdings in der Backend-Datenbank, und somit ist *DF.did* nicht mehr bereichsvollständig.

2.4 Regeln für den Entwurf von Cache Groups

Im Laufe dieses Kapitels haben wir viele Begriffe rund um das Constraint-basierte Caching erklärt und diverse Probleme, die beim Entwurf von Cache Groups beachtet werden müssen, erläutert. An dieser Stelle wollen wir ein paar Regeln wiederholen, die beim Entwurf von Cache Groups beachtet werden sollten:

> **Regel 1**: Verwenden wir das alte Sondierungsverfahren, so darf in einer Cache-Tabelle einer Cache Group maximal *eine* Non-Unique-Spalte als Füllspalte definiert werden. Die Anzahl der Unique-Spalten, die als Füllspalten definiert werden, ist beliebig.

Diese Regel sollte auch bei Cache-Group-Föderationen beachtet werden, da überlappende Cache-Tabellen wieder mehrere Non-Unique-Füllspalten in einer Cache-Tabelle verursachen können.

Beim Einsatz des neuen Sondierungsverfahrens muss *Regel 1* nicht beachtet werden. Durch die Kontrolltabellen müssen die Füllspalten nicht mehr bereichsvollständig sein und der rekursive Ladevorgang bei mehreren Non-Unique-Spalten kommt nicht zustande.

Regel 2: Heterogene RCC-Zyklen sind sowohl in Cache Groups als auch in Cache-Group-Föderationen nicht erlaubt.

Heterogene Zyklen verursachen rekursive Ladevorgänge, die im schlimmsten Fall alle Datensätze aus der Backend-Datenbank in die entsprechenden Cache-Tabellen speichern.

Wie wir in Abschnitt 2.3.1 gesehen haben, endet ein homogener Zyklus nach einem Durchlauf. Die Verwendung von homogenen Zyklen ist daher erlaubt. Die einzelnen am Zyklus beteiligten Spalten sind jedoch *nicht zwingend bereichsvollständig*.

3 Konzepte zum Caching von dynamischen Inhalten

Wie bereits motiviert, erfreuen sich Web-Anwendungen immer größerer Beliebtheit, und die Auslastungen der Backend-Datenbanken werden entsprechend größer. Der Einsatz von statischen Caching-Verfahren ist in diesem Fall nicht sonderlich effektiv. Daher mussten neue Konzepte entwickelt werden, von denen wir bereits das Constraint-basierte Caching in Kapitel 2 kennen gelernt haben. In diesem Kapitel wollen ein weiteres Verfahren zum Caching von dynamischen Web-Seiten vorstellen: das Query-Result-Caching. Des Weiteren werden wir jeweils den aktuellen Prototyp auf dem Gebiet des Datenbank-Caching von Microsoft und IBM vorstellen und ausführlich erläutern.

3.1 Query-Result-Caching

Eines der ältesten Konzepte zum Caching von dynamischen Web-Inhalten ist das Query-Result-Caching. In der Literatur findet sich eine Vielzahl von unterschiedlichen Ansätzen zu diesem Konzept [vgl. LuNu01, LZWM04, CLKS04, APTP03a]. Die Grundidee ist jedoch bei allen Ansätzen gleich: Die Ergebnisse von Anfragen sowie die Anfrage selbst werden in einem Cache nahe der Anwendungslogik zwischengespeichert. Wird dieselbe Anfrage erneut gestellt, so kann sie mit den Daten aus dem Cache beantwortet werden.

Die verschiedenen Ansätze für Query-Result-Caching kann man grob in zwei Kategorien einteilen: deklarativ und dynamisch. Beim deklarativen Query-Result-Caching werden alle Anfragen, die vom Cache beantwortet werden sollen, im Vorfeld festgelegt, d. h., der Cache kann nur ganz bestimmte Anfragen beantworten. Alle vorher nicht definierten Anfragen werden an die Backend-Datenbank geschickt und dort ausgewertet. Ein Beispiel für diesen Ansatz findet sich in [ZhTZ02]. Dort wird ein Krankheitsbeobachtungssystem vorgestellt, das deklaratives Query-Result-Caching verwendet. Das System wird von mehr als 45 Krankenhäusern in den USA zur Analyse von medizinischen Daten verwendet. Die Analyse beinhaltet viele Joins auf sehr großen Datenmengen, was bereits bei wenigen Zugriffen zu einer hohen Auslastung der Datenbank führt. Durch das Query-Result-Caching können einige der Auswertungen bereits

im Vorfeld durchgeführt und im Cache gespeichert werden. Damit die Daten des Cache aktuell bleiben, werden in regelmäßigen Intervallen die Auswertungen wiederholt und der Cache aktualisiert.

Im Gegensatz zum deklarativen Ansatz wird beim dynamischen Query-Result-Caching der Inhalt des Cache durch die Anfragen der Benutzer gefüllt. Der Cache ist somit in der Lage, sich dem Anfrageverhalten der Benutzer anzupassen. Die Arbeitsweise eines dynamischen Query Result Cache wollen wir an dem in [CLKS04] vorgestellten Prototyp WDBAccel veranschaulichen. Eine kurze Beschreibung weiterer Caching-Implementierungen findet sich in Anhang A.

3.1.1 Query-Result-Caching am Beispiel WDBAccel

Die Aufgabe des WDBAccel-Systems ist, die Ergebnisse häufig referenzierter Anfragen im Cache zu speichern und diese Ergebnisse bei neuen Anfragen wiederzuverwenden. Die Architektur von WDBAccel ist in Abbildung 3.1 dargestellt.

Den Kern des WDBAccel-Systems bildet der Query Redirector, der alle Anfragen der Anwendungslogik entgegennimmt (1) und die Ergebnisse zurückliefert (6). Enthält die Anfrage Änderungsoperationen (Insert-, Update-, Delete-Anweisungen), so wird sie direkt an den Consistency Manager weitergeleitet. Dieser leitet die Anfrage weiter an die Backend-Datenbank, wo die Änderungen durchgeführt werden. Zusätzlich ist der Consistency Manager dafür verantwortlich, dass die Daten im Cache konsistent zu den Daten in der Backend-Datenbank gehalten werden. In [CLKS04] wird jedoch nicht näher erläutert, wie der Consistency Manager die Konsistenz der Daten im Cache gewährleistet.

Lesende Anfragen werden vom Query Redirector an den Fragment Processor weitergeleitet (2). In Zusammenarbeit mit dem Cache Dictonary ermittelt der Fragment Processor, ob die Anfrage mit Hilfe des Cache-Inhalts (Cache Pool in Abbildung 3.1) beantwortet werden kann (3). Wir werden in Abschnitt 3.1.2 genauer auf die Vorgehensweise der Anfrageauswertung eingehen.

Kann die Anfrage durch den Cache-Inhalt beantwortet werden, so wird das Ergebnis aus dem Cache Pool ausgelesen (4) und über den Query Redirector an die Anwendungslogik zurückgegeben (5). Andernfalls wird die Anfrage an die Backend-Datenbank weitergeleitet und das Ergebnis ebenfalls an die Anwendungslogik zurückgeliefert. Zusätzlich werden die Anfrage und ihr Ergebnis an den Cache Controller geschickt. Der Cache Controller fügt die Anfrage in das Cache Dictonary und das Ergebnis in den Cache Pool ein.

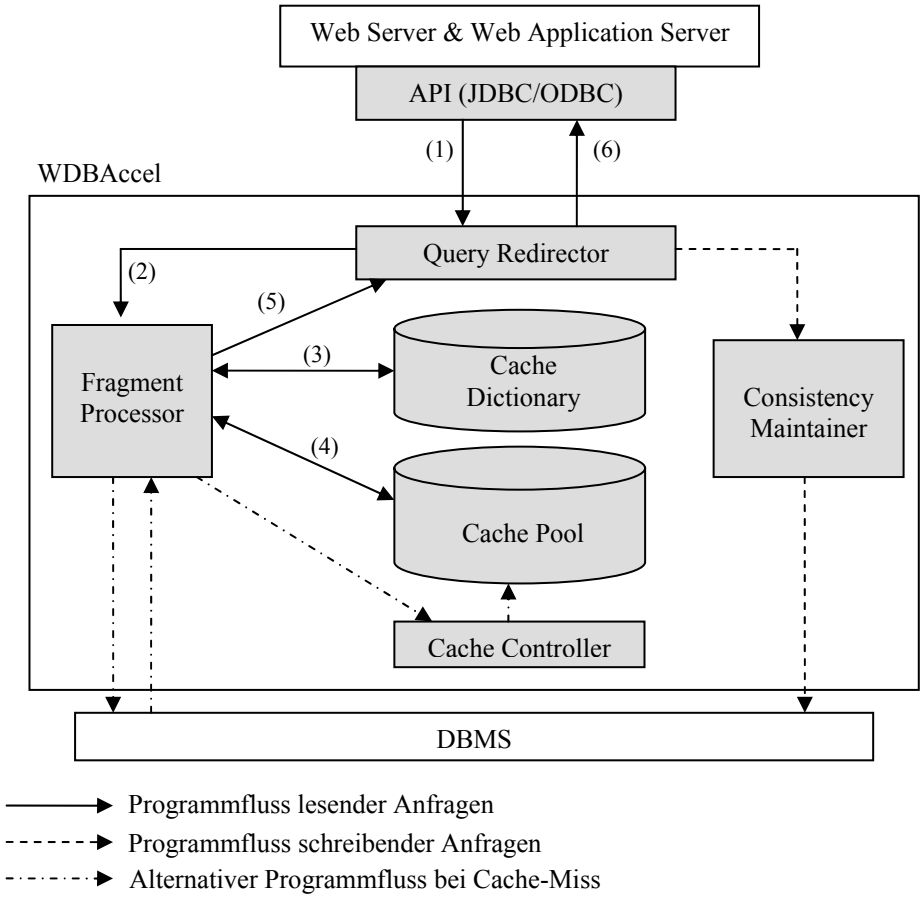


Abbildung 3.1 – WDBAccel Architektur und Prozessfluss (nach [CLKS04])

3.1.2 Query-Containment-Algorithmen

Damit eine Anfrage korrekt auf dem Cache beantwortet werden kann, muss überprüft werden, ob die Anfrage logisch in einer oder mehreren der gespeicherten Anfragen enthalten ist. Dies ist der Fall, wenn alle benötigten Datensätze, mit den benötigen Spalten, zuvor von anderen Anfragen im Cache gespeichert wurden. Zusätzlich sollten die Prädikate der neuen Anfrage dazu führen, dass das Ergebnis eine Untermenge der Daten im Cache ist.

Wir wollen an dieser Stelle einige Verfahren vorstellen, die für eine Anfrage entscheiden, ob sie durch den Cache beantwortet werden kann oder nicht. Angenommen, im Cache befindet sich das Ergebnis einer Anfrage Q_1 und die Anfrage Q_2 soll im Cache ausgewertet werden. Das Ergebnis der Anfrage Q_2 ist genau dann in der Ergebnismenge von Q_1 enthalten, wenn folgende drei Bedingungen gelten [APTP03a]:

- *Attribut-Abdeckung*: Die Projektion von Q_2 muss eine Untermenge der Projektion von Q_1 sein, d. h., die Select-Liste von Q_1 muss mindestens alle Spalten der Select-Liste von Q_2 enthalten.
- *Tupel-Abdeckung*: Für jeden Datensatz d der Backend-Datenbank gilt: Ist d

in der Ergebnismenge von Q_2 enthalten, so muss d auch in der Ergebnismenge von Q_1 enthalten sein. Oder anders ausgedrückt: Besitzt Q_1 ein Prädikat P_1 und Q_2 ein Prädikat P_2 in der Where-Klausel, dann kann die Tupel-Abdeckung wie folgt ausgedrückt werden: $\forall t : P_2(t) \Rightarrow P_1(t)$. Diese Bedingung ist genau dann wahr, wenn der Ausdruck $P_2 \wedge \neg P_1$ unerfüllbar ist.

- *Selektierungsfähigkeit*: Alle Spalten, die in der Where-Klausel oder in zusätzlichen Klauseln von Anfrage Q_2 auftreten, müssen in der Select-Liste von Q_1 enthalten sein.

Das einfachste Verfahren ist sicherlich die Überprüfung, ob eine bereits gespeicherte Anfrage Q_1 mit einer neuen Anfrage Q_2 *übereinstimmt*. Dieses Verfahren ist sehr leicht zu implementieren, liefert jedoch nicht so viele Treffer wie die nachfolgenden Verfahren.

Das zweite Verfahren arbeitet mit Hilfe von so genannten *semantischen Regionen* [DFJS96]. Jede semantische Region wird durch Prädikate beschrieben. Betrachten wir dazu die in Abbildung 2.1 auf Seite 6 dargestellte Tabelle *Regisseur*. Wir beschränken uns auf die Spalten *Wohnort* und *Alter* dieser Tabelle. Weiterhin gehen wir davon aus, dass im Cache die Ergebnisse zweier Anfragen mit folgenden Prädikaten gespeichert wurden:

- $Q_1 : (Nname = 'Spielberg' \wedge Alter \leq 60)$
- $Q_2 : (30 \leq Alter \leq 40)$

Daraus ergibt sich die in Abbildung 3.2 gestrichelt dargestellte semantische Region, die sich wie folgt beschreiben lässt:

$$R_1 = ((Nname = 'Spielberg' \wedge Alter \leq 60) \vee (30 \leq Alter \leq 40))$$

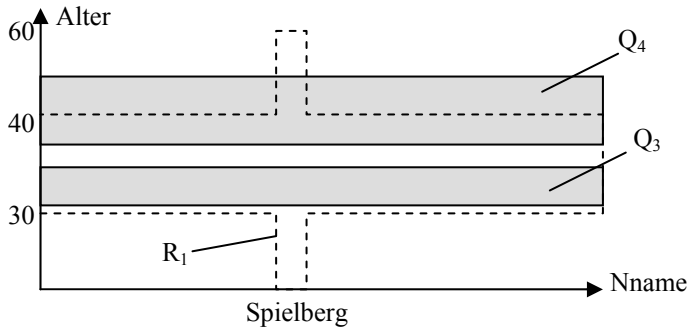


Abbildung 3.2 – Beispiel für semantische Regionen

Um eine neue Anfrage vollständig im Cache beantworten zu können, muss das Ergebnis der Anfrage vollständig in der semantischen Region R_1 enthalten sein. Beispielsweise kann die Anfrage $Q_3 : (32 \leq Alter \leq 35)$ mit Hilfe des Cache-Inhalts beantwortet werden. Die Anfrage $Q_4 : (38 \leq Alter \leq 50)$ kann nicht vollständig im Cache beantwortet werden, wie Abbildung 3.2 zeigt. In diesem Fall kann die Anfrage entweder komplett an die Backend-Datenbank geschickt werden, oder so modifiziert werden, dass

soviel wie möglich durch den Cache beantwortet wird und der Rest an die Backend-Datenbank geschickt wird.

Als Letztes wollen wir noch kurz ein Verfahren vorstellen, das im Prototyp DBProxy beschrieben wird [APTP03b]. In diesem Verfahren wird jede Anfrage einer Vorlage (Template) zugewiesen. Das Verfahren wird daher auch *Template-based query containment* genannt.

Eine neue Anfrage muss nicht mehr mit allen im Cache gespeicherten Anfragen verglichen werden, sondern lediglich mit den Anfragen, die derselben Vorlage angehören. Die Einteilung der Anfragen in Vorlagen ist bei genauerer Betrachtung von Web-Anwendung durchaus sinnvoll. In vielen Web-Anwendungen werden vordefinierte SQL-Anweisungen (*Prepared Statements*) verwendet, in denen einzelne Parameter zur Laufzeit gesetzt werden. Der Großteil der Anfrage wird jedoch nicht verändert. Angenommen, es existieren zwei Anfragen Q_1, Q_2 mit Prädikaten P_1 und P_2:

$P_1 = (Alter \geq 50 \wedge Nname = 'Spielberg') \vee (Alter < 40)$

$P_2 = (Alter \geq 60 \wedge Nname = 'Lucas') \vee (Alter < 40)$

Vergleicht man beide Prädikate, so ist sehr leicht ersichtlich, dass es zwischen beiden Gemeinsamkeiten gibt.

Daraus lässt sich eine Vorlage $P_t = (Alter \geq ? \wedge Nname = ?) \vee (Alter < 40)$ erstellen. Die Idee hinter diesem Verfahren ist, dass durch die Zuordnung einer Anfrage in eine Vorlage nur noch die variablen Teile weiter untersucht werden müssen. Im Normalfall ist das Verhältnis von variablen zu festen Teilen relativ klein, wodurch die Bearbeitung einer Anfrage wesentlich beschleunigt wird.

Zur Einordnung der Anfragen in die Vorlagen wurde in DBProxy eine zusätzliche Komponente implementiert [APTP03b], auf die wir an dieser Stelle nicht eingehen werden.

3.2 MTCache: Der Prototyp von Microsoft

MTCache ist ein Datenbank-Cache-Prototyp von Microsoft für den Microsoft SQL Server. Die Datensätze werden, für den Benutzer transparent, aus der Backend-Datenbank in den MTCache eingelagert. Der MTCache fungiert dabei als Mittelschicht zwischen Anwendungsserver und Backend-Datenbank.

Die Einlagerung von Daten in den MTCache geschieht mit Hilfe von materialisierten Sichten, die vom Datenbank-Administrator festgelegt werden. Für jede materialisierte Sicht existiert eine eigene Kontrolltabelle, die alle Werte enthält, die in der Sicht auftreten dürfen. Die Werte in den Kontrolltabellen werden vom Datenbankadministrator spezifiziert. Die materialisierte Sicht besitzt eine Klausel, die dafür sorgt, dass nur Werte aus der Kontrolltabelle eingelagert werden (vgl. Abbildung 3.3).

Die Subselect-Anweisung in der Sicht-Definition sorgt dafür, dass in der Sicht nur die in der Kontrolltabelle spezifizierten Regisseure im MTCache eingelagert werden. Dieses Verfahren hat den Vorteil, dass beispielsweise die Datensätze vom Caching ausgeschlossen werden, die relativ selten referenziert werden, oder die eine geringe Selektivität aufweisen. Jedoch finden sich weder in [LGGZ04] noch in [LaGZ04] Hinweise, ob die Werte in den Kontrolltabellen zur Laufzeit angepasst werden können und ob dies manuell oder automatisch geschehen kann.

```
CREATE TABLE Regisseur_CtrlTbl (id int);

CREATE VIEW Regisseur_View AS
      SELECT id, VName, Nname, …
      FROM Regisseur
      WHERE Nname = 'Spielzwerg'
      AND id IN (SELECT id FROM Regisseur_CtrlTbl)
```

Abbildung 3.3 – Materialisierte Sicht und Kontrolltabelle (nach [LGGZ04])

Zusätzlich zu den Sichten wird im MTCache eine so genannte *Schatten-Datenbank* angelegt. Diese Schatten-Datenbank beinhaltet alle Tabellen, Constraints, Indizes, Sichten und Zugriffsrechte der Backend-Datenbank. Die Schatten-Datenbank beinhaltet jedoch keine Datensätze, sondern lediglich statistische Werte aus dem Backend. Die Statistiken werden vom Optimierer des MTCache zur Erstellung des Anfrageplans benötigt.

Alle Anfragen, die den MTCache erreichen, werden nach lesenden und ändernden Datenbankoperationen sortiert. Die Verarbeitung der Datenbankoperationen wollen wir in den nächsten beiden Abschnitten diskutieren.

3.2.1 Bearbeitung von lesenden Anfragen

Alle lesenden Anfragen werden an den kostenbasierten Optimierer weitergeleitet, der anhand der verfügbaren Statistiken über Tabellen, Indizes, usw. einen Ausführungsplan erzeugt. Damit der Optimierer bei der Erstellung des Ausführungsplans auch die Übertragungskosten von Datensätzen aus der Backend-Datenbank berücksichtigt, wurden ein neuer Operator *DataTransfer* und ein neues Attribut *DataLocation* eingeführt. DataLocation gibt dem Optimierer an, wo sich die Daten befinden (lokal oder entfernt). Der Operator DataTransfer ändert den Wert des Attributs DataLocation.

Bei der Erstellung des Ausführungsplans muss eine Anfrage nicht zwangsläufig im Cache ausgeführt werden, selbst wenn alle Daten im Cache vorhanden sind. Durch die zur Verfügung stehenden Statistiken kann der Optimierer eine Anfrage auch komplett im Backend ausführen, wenn beispielsweise die Backend-Datenbank durch bestehende Indizes geringere Ausführungskosten verursacht. Der umgekehrte Fall ist auch möglich:

Selbst wenn der Cache nicht die benötigten Daten besitzt, kann der Optimierer die entsprechenden Datensätze aus der Backend-Datenbank anfordern und die Anfrageauswertung lokal durchführen (vgl. [LGGZ04]). Der vom Optimierer erstellte finale Ausführungsplan kann sich daher komplett auf den Cache, komplett auf die Backend-Datenbank oder auf beide beziehen.

3.2.2 Bearbeitung von Änderungsoperationen

Alle Anfragen, die eine Änderung der Datensätze verursachen (Insert-, Update-, Delete-Anweisungen), werden vom MTCache direkt an die Backend-Datenbank propagiert. Dort werden die Änderungen durchgeführt und anschließend asynchron die betroffenen MTCache-Server informiert. Diese erzeugen in regelmäßigen Abständen Schnappschüsse der benötigten Daten der Backend-Datenbank und bleiben somit aktuell. Weder in [LGGZ04] noch in [LaGZ04] werden diese Schnappschüsse näher erläutert. Es ist daher unklar, ob sie durch eine erneute Auswertung der Sichtdefinition entstehen, d. h. alle Daten der Sicht aktualisiert werden oder ob lediglich die seit dem letzten Schnappschuss veränderten Datensätze aktualisiert werden.

Es ist wichtig, dass zusätzlich zu den aktuellen Daten auch der Zeitpunkt der Einlagerung gespeichert wird. Die im MTCache replizierten Daten können ohne diese Information keine Auskunft über das Alter der Daten geben. Das Alter der Daten spielt jedoch für Web-Anwendungen, die immer die aktuellsten Daten benötigen (z. B. Börsenticker o. ä.) eine entscheidende Rolle. Dazu wurde die Syntax von SQL erweitert. Durch diese Erweiterung ist der Benutzer möglich seine Aktualitäts- und Konsistenzanforderungen auszudrücken. Der MTCache kann durch diese Erweiterung in Form einer zusätzlichen Klausel (Abbildung 3.4) entscheiden, ob die eingelagerten Daten für die Anfrage aktuell genug sind.

```
SELECT …
FROM Regisseur R, Film F
WHERE F.regie = R.id and R.alter < 35
CURRENCY BOUND 10 min ON R, 30 min ON F
```

Abbildung 3.4 – SQL Erweiterung zur Sicherung der Konsistenz (nach [LGGZ04])

Das Schlüsselwort *Currency bound* in Abbildung 3.4 gibt an, wie alt die jeweiligen Daten sein dürfen. In dem Beispiel dürfen die Daten aus der Sicht *Regisseur* maximal 10 Minuten alt und die Daten aus der Sicht *Film* maximal 30 Minuten alt sein. Sind die Daten älter als gefordert, so wird dies vom Optimierer im Anfrageplan berücksichtigt, und die entsprechenden Teile werden in der Backend-Datenbank ausgewertet.

Wir wollen die Arbeitsweise des MTCache an dieser Stelle nicht weiter vertiefen und verweisen auf die entsprechenden Stellen in [LaGZ04, LGGZ04].

3.3 DBCache: Der Prototyp von IBM

Der DBCache ist der Prototyp eines Constraint-basierten Datenbank-Cache, der von IBM in den Forschungslabors von Almaden entwickelt wurde. Der Prototyp setzt auf die bestehende DB2 Universal Database von IBM auf. Dazu wurde ein neuer Typ von Datenbankobjekten entwickelt: die *Cache-Tabelle* [BAKM03]. Eine Cache-Tabelle hat denselben Namen wie die ihr zugeordnete Tabelle in der Backend-Datenbank. Sie beinhaltet eine Teilmenge der Datensätze aus dem Backend und kann als deklarativ oder dynamisch definiert werden.

In einer deklarativen Cache-Tabelle werden alle benötigten Datensätze im Voraus gespeichert. In diesem Fall übernimmt die Cache-Tabelle die Aufgabe einer materialisierten Sicht. Zusätzlich besitzt sie einige Erweiterungen, die für die Erstellung des Anfrageplans benötigt werden (vgl. [ABKM03]).

Cache-Tabellen, die als dynamisch definiert wurden, werden durch Anfragen aus den Anwendungsservern mit Datensätzen gefüllt. Diese Cache-Tabellen müssen über einen *Nickname* mit der entsprechenden Tabelle der Backend-Datenbank verbunden werden. Im Gegensatz zu deklarativ definierten Cache-Tabellen wird bei den dynamisch definierten Cache-Tabellen kein Datenbank-Administrator benötigt, der den genauen Inhalt des Cache spezifiziert. Die Cache-Tabellen werden durch die in Abbildung 3.5 dargestellten Anweisungen definiert.

```
CREATE CACHE TABLE <Cache Table Name> AS
      SELECT * FROM <Nickname Name>
      WHERE <Predicate Definition>
```

a) Definition einer deklarativen Cache-Tabelle

```
CREATE CACHE TABLE <Cache Table Name>
      FOR <Nickname>
```

b) Definition einer dynamischen Cache-Tabelle

Abbildung 3.5 – Definition von Cache-Tabellen in DBCache

Die Definition der Cache-Tabellen kann entweder durch den Datenbank-Administrator geschehen oder durch das Initialisierungsprogramm *DBCacheInit* [LKMP02]. Der Administrator muss lediglich die Tabellen aus der Backend-Datenbank auswählen, die als Cache-Tabellen verwendet werden sollen.

DBCacheInit sammelt daraufhin alle benötigten Informationen aus dem Backend und erzeugt anschließend die Cache-Tabellen und für jede Tabelle in der Backend-Datenbank einen Nickname.

Der Inhalt des DBCache wird durch Cache Groups und Cache-Constraints beschrieben (vgl. Abschnitt 2.1). Das Füllen des Cache wird durch Cache-Keys angestoßen, die dafür sorgen, dass die entsprechenden Spalten bereichsvollständig sind.

Cache-Keys sind den aus Kapitel 2 bekannten Füllspalten ähnlich. Sie fordern jedoch im Gegensatz zu Füllspalten, dass die spezifizierten Cache-Key-Spalten zu jeder Zeit bereichsvollständig sind. Weiterhin bieten sie nicht die Möglichkeit, bestimmte Werte von der Einlagerung auszuschließen. Referenzielle Cache-Constraints (RCCs) garantieren die Korrektheit von Equijoins zwischen Cache-Tabellen [ABKM03]. Die aus Abschnitt 2.3 bekannten Probleme und Entwurfsregeln müssen auch beim DBCache beachtet werden. Der unvorsichtige Einsatz von RCCs kann andernfalls zum unerwünschten rekursiven Ladeverhalten führen.

3.3.1 Bearbeitung von lesenden Anfragen

Für alle lesenden Anfragen erstellt der DBCache zwei Anfragepläne: einen, der sich auf alle verwendbaren Cache-Tabellen bezieht, und einen, der sich ausschließlich auf die Tabellen in der Backend-Datenbank bezieht. Zusätzlich wird eine Anfrage erstellt, die die Existenz der benötigten Datensätze im Cache überprüft. Je nach Ergebnis dieser Existenz-Anfrage wird der erste oder zweite Anfrageplan ausgeführt. Diese Vorgehensweise wird von IBM als *Janus-Plan* bezeichnet (Abbildung 3.6).

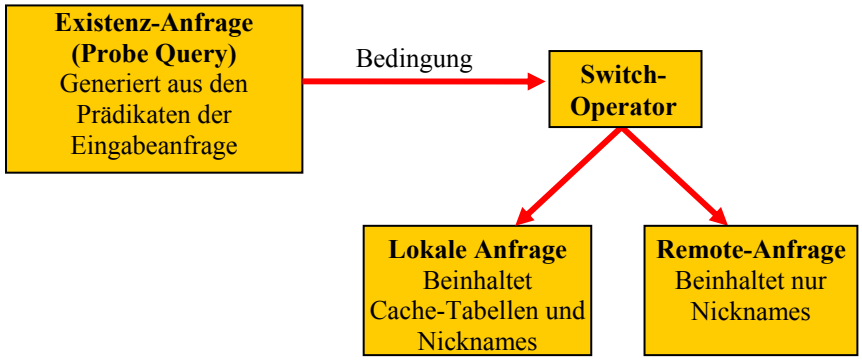


Abbildung 3.6 – Aufbau des Janus-Plans (nach [ABKM03])

Der Janus-Plan wird in vier Schritten erzeugt:

Schritt 1: In der eintreffenden Anfrage werden alle Tabellennamen durch Nicknames ersetzt und daraus ein Anfrageplan (Remote-Anfrageplan) erstellt.

Schritt 2: In diesem Schritt wird eine Existenz-Anfrage generiert, die für alle Gleichheitsprädikate der Anfrage überprüft, ob diese mit Hilfe der Datensätze im Cache ausgewertet werden können. Die in den Prädikaten angegebenen Spalten dienen dabei als Einstiegspunkte in die Cache Group. In der Existenz-Anfrage sind alle zu überprüfenden Gleichheitsprädikate konjunktiv verknüpft. Dies ist jedoch nicht sehr effektiv, da die Existenz-Anfrage bereits scheitert, wenn ein Prädikat nicht erfüllt ist. Die

Cache-Verwaltung führt in diesem Fall den in Schritt 1 erstellten Remote-Anfrageplan aus, selbst wenn alle anderen Prädikate durch den Cache beantwortet werden können. Die Backend-Datenbank wird dadurch unnötig belastet.

Nachdem die Existenz-Anfrage generiert wurde, werden die Equijoin-Prädikate der Anfrage ausgewertet. Für jedes Equijoin-Prädikat wird überprüft, ob ein entsprechender RCC existiert und ob die Quelltabelle des RCC bereits verankert ist. Ist die Quelltabelle noch nicht verankert, so wird das Prädikat zurückgestellt und im nächsten Durchlauf nochmals überprüft. Ist die Quelltabelle verankert, so wird auch die Zieltabelle verankert. Diese Vorgehensweise wird solange wiederholt, bis keine Verankerungen mehr möglich sind.

Schritt 3: Im diesem Schritt wird vom Anfrageplan aus Schritt 1 zuerst eine Kopie erstellt. Für jedes Prädikat, das im Cache beantwortet werden kann, werden in der Kopie (lokaler Anfrageplan) die Nicknames durch die entsprechenden Cache-Tabellen ersetzt. Alle Nicknames, die nicht ersetzt werden können, bleiben im lokalen Anfrageplan erhalten. Dieser kann also sowohl Cache-Tabellen als auch Nicknames enthalten. Um diesen Plan auszuführen, muss die Anfrage auf mehrere Datenbanken (Cache und Backend) verteilt werden. Die Verteilung der Anfrage übernimmt im Falle von DBCache die DB2, die Funktionen bereitstellt, um auf föderierte Datenbanken zuzugreifen.

Schritt 4: Im letzten Schritt werden beide Anfragepläne an einen Switch-Operator angehängt. Der Switch-Operator entscheidet zur Laufzeit, abhängig vom Ergebnis der in Schritt 2 generierten Existenz-Anfrage, welcher Anfrageplan ausgeführt wird.

Alle Schritte können zur Compile-Zeit durchgeführt werden, lediglich die Entscheidung des Switch-Operators findet zur Laufzeit statt.

Wurde ein benötigter Wert w in einer Cache-Key-Spalte nicht gefunden, so muss dieser Wert in den Cache eingelagert werden. Das Nachladen der Datensätze geschieht asynchron zur Anfragebearbeitung, da es durch die Cache-Constraints vorkommen kann, dass eine große Anzahl von Datensätzen in den Cache eingelagert werden muss. Diese Einlagerung würde die Antwortzeit der Anfrage enorm vergrößern.

Die Einlagerung von Datensätzen übernimmt im DBCache ein Cache-Daemon. Wird während der Anfragebearbeitung ein Wert in einer Cache-Key-Spalte nicht gefunden, so erhält der Cache-Daemon eine Nachricht, die in einer Queue gespeichert wird. Der Cache-Daemon arbeitet diese Queue ab und lädt für jeden Eintrag die benötigten Datensätze aus der Backend-Datenbank nach. Dazu werden Insert-

Anweisungen, wie sie das Beispiel in Abbildung 3.7 zeigt, generiert und anschließend ausgeführt. Der Cache-Daemon ist auch dafür verantwortlich, dass nach dem Füllen alle Cache-Constraints wieder erfüllt sind und der Cache einen gültigen Zustand besitzt.

```
INSERT INTO Regisseur
SELECT * FROM Nickname_Regisseur
WHERE Nname = 'Spielberg'
INSERT INTO Film
SELECT * FROM Nickname_Film
WHERE Regie IN (SELECT id
      FROM Nickname_Regisseur
      WHERE Nname = 'Spielberg)
```

Abbildung 3.7 – Insert-Anweisung des Cache-Daemon

3.3.2 Bearbeitung von Änderungsoperationen

In der aktuellen Version des DBCache (Stand 2004) werden alle Änderungsoperationen direkt an die Backend-Datenbank weitergeleitet und dort verarbeitet. Der Cache-Daemon erhält vom Backend eine Nachricht mit Informationen über die durchgeführten Änderungen. Aus der Nachricht erkennt der Cache-Daemon, ob die Änderungen in der Backend-Datenbank durch eine Insert-, eine Update- oder eine Delete-Anweisung verursacht wurden. Folgende Aktionen werden, wie in [BAMP04] beschrieben, vom Cache-Daemon durchgeführt:

- *Insert-Anweisung*: Wurden in der Backend-Datenbank neue Datensätze hinzugefügt, so werden diese nur dann in den Cache übernommen, wenn auf der Cache-Tabelle eingehende RCCs existieren und die Werte der neuen Datensätze in einer der Quellspalten dieser RCCs vorhanden sind, oder wenn in der Cache-Tabelle eine Non-Unique-Spalte als Cache-Key definiert wurde und die Werte der neuen Datensätze bereits in dieser Spalte vorhanden sind. Die Generierung der Insert-Anweisungen erfolgt analog zu den Insert-Anweisungen in Abbildung 3.7.
- *Update-Anweisung*: Bei einer Update-Anweisung müssen zwei Fälle unterschieden werden:
  - Betraf das Update nur Spalten, die weder als Cache-Key noch als Quell- bzw. Zielspalte eines RCC definiert wurden, so kann das Update direkt auf dem Cache ausgeführt werden.
  - Ist dies nicht der Fall, so wird die Update-Anweisung in eine Delete-Anweisung mit anschließender Insert-Anweisung transformiert. Das heißt, alle vom Update betroffenen Datensätze werden zuerst gelöscht und anschließend wieder eingefügt. Der Cache-Daemon stellt beim Einfügen wieder einen gültigen Cache-Zustand her.

- *Delete-Anweisung*: Wurden Datensätze in der Backend-Tabelle gelöscht, so müssen diese Datensätze auch im Cache gelöscht werden. Nach dem Löschen dieser Datensätze müssen zusätzlich noch alle Datensätze entlang ausgehender RCCs gelöscht werden, die weder durch andere Cache-Keys noch eingehende RCCs referenziert werden.

In periodischen Abständen löscht ein Garbage Collector Werte einer Cache-Key-Spalte aus dem Cache. Dabei wird ein Wert auf den Cache-Key-Spalten als Opfer bestimmt und, beginnend in der Cache-Key-Spalte, alle Datensätze entlang der ausgehenden RCCs gelöscht. Datensätze die durch andere RCCs oder andere Cache-Key-Spalten referenziert werden, werden vom Löschvorgang ausgeschlossen. Die Überprüfung, welche Datensätze in einer Tabelle gelöscht werden können, ist sehr aufwendig und muss sorgfältig durchgeführt werden, da das Löschen noch benötigter Datensätze zu einem ungültigen Cache-Zustand führt.

3.4 Vor- und Nachteile von DBCache und MTCache

Sowohl der Prototyp von Microsoft (MTCache) als auch der von IBM (DBCache) haben in ihrer Konzeption bestimmte Vorteile, aber auch Nachteile.

Ein Vorteil des MTCache gegenüber dem DBCache ist der Einsatz von Kontrolltabellen, die beschreiben, welche Werte in den Cache eingelagert werden sollen und welche nicht. Die Definition von Cache-Keys in DBCache führt dazu, dass alle Werte einer Cache-Key-Spalte wertvollständig gemacht werden. Dadurch können auch Werte in den Cache gelangen, die im Backend eine sehr geringe Selektivität aufweisen.

Die Kontrolltabellen des MTCache besitzen allerdings auch einen Nachteil. Da der MTCache auf Basis von materialisierten Sichten entwickelt wurde, werden die Datensätze aller Werte in den Kontrolltabellen auf einmal in die jeweiligen Sichten geladen. Der DBCache löst dieses Problem eleganter, indem die Datensätze bei Bedarf aus der Backend-Datenbank nachgeladen werden.

Das Sondierungsverfahren des DBCache zeigt einen weiteren großen Nachteil dieses Prototyps. Das System überprüft nicht für jedes Prädikat, ob es als ein möglicher Einstiegspunkt dienen kann, sondern es werden alle Prädikate konjunktiv Verbunden und in *einer* Existenz-Anfrage überprüft. Ist nur eines der Prädikate nicht erfüllt, so scheitert die gesamte Sondierung und die Anfrage wird komplett an die Backend-Datenbank geschickt. Vermutlich wurde diese Vorgehensweise gewählt, um schnell entscheiden zu können, welcher der beiden Ausführungspläne bearbeitet werden soll. Nach Angaben von IBM soll diese Vorgehensweise jedoch in zukünftigen Versionen des Prototyps verbessert werden [ABKM03].

Sowohl beim MTCache als auch beim DBCache geschieht das Caching transparent

für den Benutzer und die Anwendungslogik. Es müssen daher keine Änderungen, auch bei einem Wechsel der Caching-Strategie, in den Web-Anwendungen durchgeführt werden.

Beide Systeme sind sehr inflexibel, was den Typ der eingesetzten Cache-Datenbank betrifft. MTCache wurde speziell für den Microsoft SQL-Server entwickelt. Wegen der Erweiterung des SQL-Sprachumfangs, um die Aktualität der Ergebnisse festlegen zu können, kann der MTCache nicht ohne weiteres auf einem anderen Datenbank-System eingesetzt werden. Auch der DBCache ist an die DB2 von IBM gebunden. Wie bereits erwähnt, setzt der DBCache direkt auf der DB2 auf, d. h., es wurden Änderungen im Datenbank-System durchgeführt (Einführung neuer Datenbank-Objekttypen), sodass als Cache-Datenbank nur die DB2 von IBM eingesetzt werden kann.

Diese Gründe gaben den Ausschlag für den Entwurf eines eigenen Prototyps, der die Vorteile beider Systeme vereint und zugleich unabhängig von der eingesetzten Cache-Datenbank ist. Das Konzept für diesen Prototyp stellen wir im folgenden Kapitel genauer vor.

4 Unser Datenbank-Cache-Prototyp

In diesem Kapitel wollen wir den von uns entwickelten Prototyp genauer vorstellen. Dieser Prototyp orientiert sich an den bereits bestehenden Systemen von IBM und Microsoft (vgl. Abschnitten 3.2 und 3.3). Motiviert, die Vorteile beider Systeme zu vereinen und gleichzeitig die beschriebenen Nachteile zu vermeiden, war es unser Ziel, einen Constraint-basierten Datenbank-Cache zu entwickeln, der den aktuellen Stand der Forschung repräsentiert. Der Prototyp trägt den Namen CBCS (Constraint-based Caching System) und ist ein adaptiver Datenbank-Cache. Die Transparenz des Cache gegenüber dem Benutzer wird durch eine implementierte JDBC-Schnittstelle gewährleistet.

Die SQL-Anfragen der Benutzer werden von einer Komponente des CBCS, dem Query Worker, automatisch analysiert und derart verändert, dass das Ergebnis gleich bleibt, jedoch möglichst viele Teile der Anfrage auf dem Cache beantwortet werden können. Dazu verwendet die Query-Worker-Komponente die in Kapitel 2 beschriebenen Konzepte sowie das in Abschnitt 2.2.1 vorgestellte Sondierungsverfahren. Die Query-Worker-Komponente bildet den Kern des CBCS, da alle Benutzeranfragen von dieser Komponente analysiert und bearbeitet werden.

Das Füllen des Cache wird von einem Fill Daemon übernommen. Wird während der Analyse der Anfrage festgestellt, dass ein Wert in einer Füllspalte nicht vorhanden ist, so sorgt der Fill Daemon dafür, dass alle Datensätze mit diesem Wert in den Cache eingelagert werden. Dies schließt auch die Einlagerung der benötigten Datensätze in den Tabellen mit ein, die durch ausgehende RCCs referenziert werden.

Sollte der Speicher in der Cache-Datenbank einmal knapp werden, so sorgt eine Löschkomponente (Garbage Collector) dafür, dass veraltete Datensätze aus dem Cache entfernt werden. Sowohl der Fill Daemon also auch der Garbage Collector müssen dafür sorgen, dass sich der Cache nach ihrer Arbeit wieder in einem gültigen Zustand befindet.

Das CBCS verwaltet auch Statistiken über die Zugriffshäufigkeit von Datensätzen im Cache. Diese Aufgabe wird von der Hit-Counter-Komponente übernommen. Für jede von der Query-Worker-Komponente bearbeitete Benutzeranfrage werden durch den Hit Counter die entsprechenden Statistiken aktualisiert. Wir werden auf die Bedeutung dieser Statistiken in den Abschnitten 5.2.5 und 5.2.6 ausführlich eingehen.

Abbildung 4.1 zeigt eine Übersicht aller Komponenten des CBCS. Die Aufgaben dieser Komponenten wollen wir in den folgenden Abschnitten ausführlich erläutern. Dabei werden wir in diesem Kapitel die Komponenten auf abstrakter Ebene betrachten, bevor wir in Kapitel 5 auf ihre konkrete Implementierung eingehen.

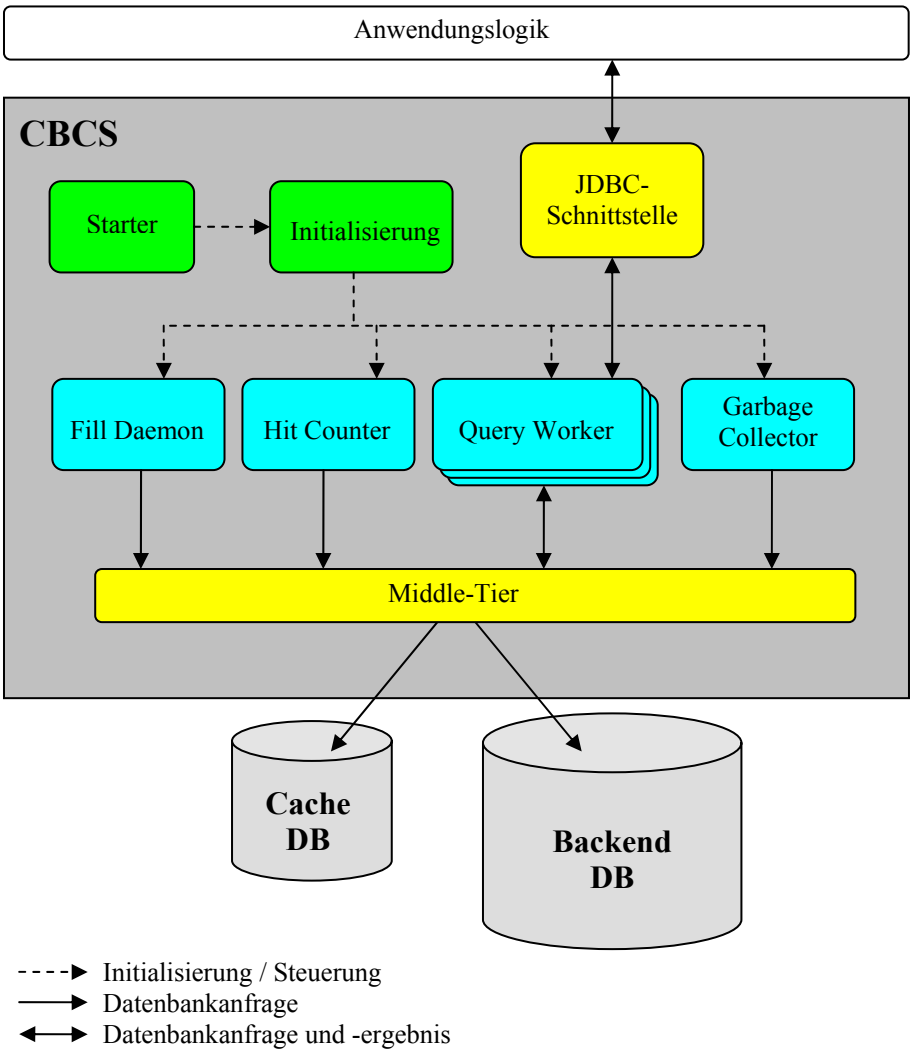


Abbildung 4.1 – Komponenten des CBCS

4.1 Starten des CBCS

Das Starten des CBCS erfolgt über eine eigenständige Komponente, die im Wesentlichen für drei Aufgaben zuständig ist:

1. *Starten der Initialisierungskomponente und Überwachung des Initialisierungsvorgangs*. Bevor die Initialisierungskomponente gestartet werden kann, müssen zunächst alle Eingabeparameter verarbeitet und eventuelle Konfigurationsdateien geladen werden. Diese Aufgabe wird von der Starterkomponente übernommen. Anschließend beginnt die Initialisierungsphase, die von der

Starterkomponente überwacht wird. Zusätzlich behandelt die Starterkomponente alle während der Initialisierung auftretenden Fehler.

2. *Schnittstelle zur Interaktion des Administrators mit dem System*. Die Starterkomponente bietet dem Administrator eine Schnittstelle zur Steuerung des CBCS in Form einer Kommandozeile. Der Administrator hat über diese Schnittstelle beispielsweise die Möglichkeit, einzelne Komponenten zu schließen oder manuell Werte einer Füllspalte in den Cache zu laden. Da es sich beim CBCS um den ersten Prototyp handelt, ist der Funktionsumfang der Schnittstelle noch relativ begrenzt, und der Großteil der Befehle dient dem Testen der einzelnen Komponenten.
3. *Ordnungsgemäßes Schließen aller Komponenten beim Beenden des Systems*. Erhält das CBCS vom Administrator den Befehl zum Beenden des Systems, so sorgt die Starterkomponente dafür, dass alle Komponenten und alle Verbindungen zur Datenbank ordnungsgemäß beendet werden. Auch dieser Vorgang wird von der Starterkomponente überwacht und eventuelle Fehler werden behandelt.

4.2 Middle Tier: Die Schnittstelle zu den Datenbanken

Diese Komponente verwaltet die Verbindungen zur Cache- und Backend-Datenbank und stellt den anderen Komponenten eine Schnittstelle zum Zugriff auf diese Datenbanken zur Verfügung. Dadurch erhalten die übrigen Komponenten eine homogene Sicht auf die Tabellen der Cache- und Backend-Datenbank. Anfragen der anderen Komponenten, wie beispielsweise des Query Worker, werden über die Middle-Tier-Komponente an die entsprechenden Datenbanken verteilt und die zurückgegebenen Ergebnisse anschließend zusammengeführt und der aufrufenden Komponente zurückgegeben.

Die Schnittstellen der Middle-Tier-Komponente umfassen alle vom CBCS benötigten Funktionen zum Zugriff auf die Datenbanken. Dazu gehört unter anderem die Möglichkeit, Transaktionen zu beginnen und zu beenden (Commit und Rollback) sowie lesende und ändernde Anfragen auszuführen.

Der Einsatz der Middle-Tier-Komponente hat den Vorteil, dass das CBCS unabhängig von den verwendeten Datenbanksystemen ist. Wird ein anderes Datenbanksystem eingesetzt, so muss die Middle-Tier-Komponente an die neuen Gegebenheiten angepasst werden. Die restlichen Komponenten des CBCS können ohne Änderungen weiterverwendet werden.

4.3 Die Initialisierung des CBCS

Bevor das CBCS seinen Betrieb aufnehmen kann, müssen einige Vorbereitungen durchgeführt werden, die von der Initialisierungskomponente übernommen werden. Sie ist dafür verantwortlich, dass Komponenten wie der Fill Daemon, der Hit Counter oder auch der Garbage Collector richtig eingerichtet werden. Auch die wichtigste Komponente des Systems, der Query Worker, wird von der Initialisierungskomponente für den laufenden Betrieb vorbereitet.

Dabei ist es wichtig, dass bei der Initialisierung der Komponenten keine Fehler auftreten, da dies zu Folgefehlern im laufenden Betrieb führen kann. Tritt dennoch ein Fehler auf, so wird dieser an die Starterkomponente weitergeleitet, die daraufhin entscheidet, ob die Initialisierung fortgeführt werden kann oder der Fehler zu schwerwiegend war und das ganze System beendet werden muss.

Zusätzlich zur Initialisierung der oben genannten Komponenten müssen noch weitere Vorbereitungen getroffen werden, damit ein reibungsloser Betrieb des CBCS möglich ist. Dafür bietet das CBCS zwei Arten der Initialisierung an: eine normale und eine erweiterte Initialisierung. Bei der normalen Initialisierung muss als Voraussetzung die Cache-Datenbank bereits eingerichtet sein. Die Einrichtung des Cache wird in der erweiterten Initialisierung durchgeführt. Durch einen Parameter beim Start des CBCS kann der Administrator entscheiden, welche Art der Initialisierung er durchführen möchte. In den beiden folgenden Abschnitten wollen wir diese beiden Initialisierungsarten vorstellen und erläutern.

4.3.1 Normale Initialisierung

Die normale Initialisierung wird standardmäßig vom CBCS durchgeführt. Diese Initialisierung setzt voraus, dass die Cache-Datenbank bereits ordnungsgemäß eingerichtet wurde, d. h., während der normalen Initialisierung werden keine Änderungen an der Cache-Datenbank vorgenommen, sondern lediglich die Komponenten des CBCS und benötigte Datenstrukturen korrekt eingerichtet und gestartet.

Die normale Initialisierung gliedert sich in vier Schritte:

1. Auslesen der Konfigurationsdatei mit Informationen über Cache-Tabellen, spezifizierten Füllspalten und RCCs
2. Erzeugung einer Datenstruktur aus den ausgelesenen Informationen
3. Erzeugen von Prepared Statements für den Fill Daemon, den Hit Counter und den Garbage Collector
4. Initialisieren des Fill Daemon, des Hit Counter, des Garbage Collector und der Query-Worker-Instanzen

Auslesen der Konfigurationsdatei

Alle Informationen über spezifizierte Cache-Tabellen, Füllspalten und RCCs befinden sich in einer Konfigurationsdatei, die während der Initialisierungsphase ausgelesen wird.

Die Konfigurationsdatei enthält für jede spezifizierte Cache-Tabelle den Schema- und Tabellennamen der Tabelle in der Backend-Datenbank und in der Cache-Datenbank. Diese Namen dienen der Abbildung von Backend-Tabellen auf Cache-Tabellen. Weiterhin müssen für jede Cache-Tabelle Informationen über die enthaltenen Spalten sowie die Primärschlüssel der zugehörigen Backend-Tabelle in der Konfigurationsdatei vorhanden sein.

Eine Beschreibung einer Spalte in der Konfigurationsdatei umfasst den Namen der Spalte, der identisch zum Spaltennamen in der Backend-Tabelle ist und ihren Datenbanktyp. Zusätzlich wird für jede Spalte spezifiziert, ob sie eine Unique-Spalte ist und ob sie Null-Werte enthalten darf.

Die Konfigurationsdatei enthält darüber hinaus auch alle Angaben über die spezifizierten RCCs und Füllspalten. Jeder RCCs referenziert eine Quell- und eine Zielspalte. Die Einträge für die Füllspalten enthalten eine Referenz auf die entsprechende Spalte der Cache-Tabelle und Informationen darüber, welche Werte in den Cache eingelagert werden dürfen und welche nicht. Dies wird in der aktuellen Version des CBCS mit einem Prozentwert beschrieben, den wir in Abschnitt 4.3.2 genauer betrachten.

Nachdem die Informationen aus der Konfigurationsdatei ausgelesen wurden, werden sie in der im folgenden Abschnitt beschriebenen Datenstruktur gespeichert und den anderen Komponenten des CBCS zugänglich gemacht.

Aufbau der internen Datenstruktur

Die interne Datenstruktur des CBCS besteht aus drei Objekt-Typen, die in Abbildung 4.2 als UML-Klassendiagramm dargestellt sind. Über diese Datenstruktur, die allen Komponenten des CBCS zugänglich ist, ist ein effizienter Zugriff auf alle benötigten Informationen bezüglich der Cache-Tabellen, deren Spalten sowie der spezifizierten RCCs möglich.

Für jeden Cache-Tabelleneintrag in der Konfigurationsdatei wird ein Tabellen-Objekt erzeugt, in dem die spezifizierten Informationen enthalten sind. Die zugehörigen Spalteneinträge der Cache-Tabelle werden als Spalten-Objekte dem Tabellen-Objekt hinzugefügt. Ist eine Spalte als Füllspalte spezifiziert worden, so werden zwei zusätzliche Tabellen-Objekte erzeugt und vom Spalten-Objekt referenziert. Die erste der beiden Tabellen ist die Kontrolltabelle, in der Informationen über die eingelagerten Werte in der Füllspalte (Wert, Einlagerungszeitpunkt, letzter Zugriff, …) gespeichert werden. Die zweite Tabelle ist die Füllwert-Tabelle. Diese enthält alle Werte der Backend-Datenbank, die in der Füllspalte eingelagert werden dürfen.

Alle in der Konfigurationsdatei spezifizierten RCCs werden in RCC-Objekten gespeichert. Jedes RCC-Objekt enthält Referenzen auf die Quellspalte des RCC und die zugehörige Zielspalte. Eine Referenz auf das RCC-Objekt wird sowohl in der Quellspalte als auch in der Zielspalte gespeichert. Jedes Spalten-Objekt besitzt also zwei Mengen: Eine Menge enthält alle eingehenden RCCs dieser Spalte, die andere Menge enthält alle ausgehenden RCCs.

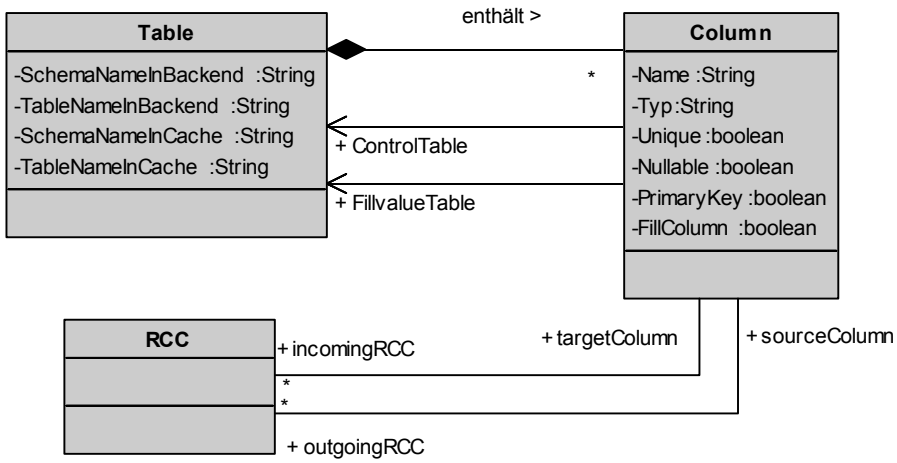


Abbildung 4.2 – Objekt-Typen der internen Datenstruktur

Mit Hilfe dieser Datenstruktur ist eine einfache Navigation über die RCCs der Cache Group möglich. Die Speicherung der RCCs in den Spalten-Objekten und die Trennung in eingehende und ausgehende RCCs haben den Vorteil, dass wir für die Sondierung sehr effizient alle eingehenden RCCs zu einer Spalte bestimmen können. Die einfache Navigation über die RCCs zu den Tabellen-Objekten erleichtert auch die im folgenden Abschnitt beschriebene Generierung von Prepared Statements.

Generierung der Anfragen

Während der Laufzeit des CBCS werden viele Anfragen an den Cache und die Backend-Datenbank gestellt. Ein Teil dieser Anfragen wird von den Benutzern gestellt, ein anderer, nicht unerheblicher Teil entsteht durch das CBCS. Dazu gehören unter anderem Anfragen, die die Existenz eines Wertes in einer Füllspalte überprüfen oder die neue Datensätze in den Cache einlagern.

Da diese Anfragen teilweise sehr komplex sind und häufig ausgeführt werden, haben wir uns dafür entschieden, alle Anfragen, die vom CBCS im laufenden Betrieb benötigt werden, bereits während der Initialisierungsphase zu generieren. Dazu verwenden wir Prepared Statements. Diese haben den Vorteil, dass die Anfragen dem Anfrageoptimierer der Datenbank bereits vor der eigentlichen Ausführung zur Verfügung stehen und dieser mehr Aufwand für die Optimierung verwenden kann. Zum Ausführungszeitpunkt sind die Anfragepläne bekannt und können ohne weiteren Optimierungsaufwand ausgeführt werden.

Die Komponenten des CBCS verwenden insgesamt vier unterschiedliche Typen von Anfragen. Auf die Generierung der einzelnen Anfragetypen werden wir in den entsprechenden Abschnitten genauer eingehen.

- Existenz-Anfragen: Dieser Typ wird hauptsächlich von der Query-Worker-Komponente für die Sondierung verwendet (vgl. Abschnitt 4.4).
- Insert-Anfragen: Dieser Anfragentyp wird vom Fill Daemon benötigt, um neue Datensätze in den Cache einzulagern (vgl. Abschnitt 4.5).
- Update-Anfragen: Der Hit Counter verwendet diese Anfragen, um die Informationen in den Kontrolltabellen zu aktualisieren (vgl. Abschnitt 4.6).
- Delete-Anfragen: Anfragen von diesem Typ werden vom Garbage Collector verwendet, um Datensätze aus dem Cache zu löschen (vgl. Abschnitt 4.7).

Initialisierung der übrigen Komponenten

Am Ende der Initialisierungsphase müssen noch die restlichen Komponenten des CBCS eingerichtet werden. Dazu gehören der Fill Daemon, der Hit Counter und der Garbage Collector, die alle in eigenständigen Threads gestartet werden.

Von der Query-Worker-Komponente erzeugen wir mehrere Instanzen, damit gleichzeitig mehrere Benutzeranfragen bearbeitet werden können. Dadurch wird der Durchsatz des Systems erhöht. Die Anzahl der Query Worker wird vom Administrator in der Konfigurationsdatei des CBCS festgelegt.

4.3.2 Erweiterte Initialisierung

Während der erweiterten Initialisierung wird zusätzlich zu den Aufgaben der normalen Initialisierung die Einrichtung der Cache Group in der Cache-Datenbank durchgeführt. Dazu sind drei zusätzliche Schritte notwendig, die nach der Erzeugung der Datenstruktur und vor der Erzeugung der Prepared Statements durchgeführt werden:

1. Erzeugen der spezifizierten Cache-Tabellen sowie entsprechender Kontrolltabellen
2. Anlegen von Tabellen, die die erlaubten Werte der zugehörigen Füllspalte enthalten (Füllwert-Tabelle)
3. Anlegen von Indizes

Bevor wir Anfragen auf dem Cache ausführen können, benötigen wir die entsprechenden Cache-Tabellen. Die Cache-Tabellen werden aus den Informationen in den Tabellen-Objekten der Datenstruktur generiert. Dazu wird für jedes Tabellen-Objekt eine Tabelle im Cache erzeugt. Die Cache-Tabellen können dabei in beliebiger Reihenfolge erzeugt werden, da wir im Cache keine Fremdschlüsselbeziehungen abbilden. Zusätzlich zu den spezifizierten Cache-Tabellen werden die Kontrolltabellen und die Füllwert-Tabelle in diesem Initialisierungsschritt erzeugt.

Nachdem alle Cache-Tabellen erzeugt wurden, müssen wir im nächsten Schritt alle zulässigen Werte der Füllspalten in die jeweiligen Füllwert-Tabellen einfügen. In der aktuellen Version des CBCS können die erlaubten Werte einer Füllspalte nur über einen Prozentwert spezifiziert werden. Der Prozentwert gibt eine Obergrenze für die Selektivität eines Werts in der Füllspalte an. Wir wollen dies an einem kleinen Beispiel verdeutlichen. Betrachten wir dazu die in Abbildung 4.3 dargestellte Tabelle der Backend-Datenbank. Die Spalte *Nname* sei im Cache eine Füllspalte und der angegebene Prozentwert wäre 25%. Weiterhin gehen wir davon aus, dass die Tabelle 10.000 Datensätze enthält. Die in Abbildung 4.3 dargestellten Spalten *Absolut* und *Relativ* sind keine Spalten der Tabelle *Darsteller* , sondern geben die absolute und relative Häufigkeit an, mit der ein bestimmter Wert in dieser Tabelle vorkommt. Zur besseren Veranschaulichung wurden die Datensätze absteigend nach ihrer Häufigkeit geordnet.

$Darsteller_B$	Nname	Absolut	Relativ
	…	…	…
	Maier	3120	31,2%
	Müller	2560	25,6%
	Bäcker	2340	23,4%
	Schmidt	810	8,1%
	…	…	…

Abbildung 4.3 – Beispiel für erlaubte Werte in einer Füllwert-Tabelle

Der Prozentwert von 25% bedeutet nun, dass nur die Werte in die Füllwert-Tabelle eingetragen werden, die eine relative Häufigkeit von 25% oder weniger besitzen. Für das Beispiel bedeutet dies, dass der Wert *Bäcker* und alle nachfolgenden Werte der Spalte *Nname* in die Füllwert-Tabelle eingetragen werden. Der Wert *Müller* und alle davor liegenden Werte erfüllen das Selektivitätskriterium nicht und werden daher nicht in die Füllwert-Tabelle übernommen.

Nur die in der Füllwert-Tabelle enthaltenen Werte dürfen später vom Fill Daemon in den Cache eingelagert werden. Die Füllwert-Tabelle sollte in regelmäßigen Abständen (z. B. alle 24 Stunden) aktualisiert werden, damit Veränderungen der Selektivität in der Backend-Datenbank, die durch Änderungsoperationen entstanden sind, berücksichtigt werden.

Im letzten Schritt der erweiterten Initialisierung werden auf allen notwendigen Spalten Indizes erstellt, um den Zugriff auf diese Spalten zu beschleunigen. Durch die Sondierung werden sehr viele Existenz-Anfragen auf den Quellspalten der RCCs durchgeführt. Dabei wollen wir bei der Sondierung lediglich wissen, ob ein bestimmter Wert bereits im Cache vorhanden ist oder nicht. Ein Index auf der Quellspalte der RCCs beschleunigt diese Suche, da die Datenbank nicht die gesamte Tabelle durchsuchen muss, sondern lediglich eine Suche auf dem B*-Baum des Index durchführt. Der Durchlauf auf dem B*-Baum ist wesentlich effizienter, da im Normalfall nur ein bis zwei

Festplattenzugriffe zum Auffinden eines Datensatzes benötigt werden. Wir erzeugen in diesem Schritt auf jeder Quellspalte eines RCCs und auf jeder Unique-Spalte[1] einen Index.

Im Anschluss an diese drei Schritte werden analog zur normalen Initialisierung die Prepared Statements erzeugt und die restlichen Komponenten eingerichtet.

4.4 Die Query-Worker-Komponente

Den Kern des CBCS bildet der Query Worker. Diese Komponente ist für die Verarbeitung von Benutzeranfragen zuständig. Damit mehrere Benutzer gleichzeitig Anfragen ausführen können, existieren zur Laufzeit mehrere Instanzen der Query-Worker-Komponente, die in einem Pool verwaltet werden. Bei Bedarf werden diese Instanzen den Benutzern zugeteilt und nach getaner Arbeit wieder in den Pool zurückgelegt.

Über die JDBC-Schnittstelle, die wir in Abschnitt 4.8 genauer erläutern werden, erhält der Query Worker die Anfrage des Benutzers. Der Query Worker sortiert die eintreffenden Anfragen nach Lese- und Änderungsoperationen. Die Verarbeitung der Leseoperationen betrachten wir im folgenden Abschnitt etwas genauer.

In der aktuellen Version des CBCS ist die Verarbeitung von Änderungsoperationen noch nicht möglich, dennoch wollen wir in Abschnitt 4.4.2 einige Vorgehensweisen zur Bearbeitung von Änderungsoperationen diskutieren.

Nach der Verarbeitung der Datenbankoperation im Query Worker wird die Anfrage über die Middle-Tier-Komponente ausgeführt und das Ergebnis über die JDBC-Schnittstelle an den Benutzer zurückgegeben.

4.4.1 Bearbeitung von Leseoperationen

Alle lesenden Anfragen werden im Query Worker zuerst gegen eine Grammatik validiert. Entspricht die Anfrage nicht der Grammatik, so bedeutet dies, dass die Anfrage vom Query Worker nicht bearbeitet werden kann. In diesem Fall wird die Anfrage unverändert an die Backend-Datenbank geschickt und dort ausgewertet. Der Grund für diese Vorgehensweise ist, dass wir in der aktuellen Version des CBCS noch nicht die komplette SQL-Select-Syntax unterstützen. Alle Anfragen, die der Syntax in Abbildung 4.4 entsprechen, können vom Query Worker weiterverarbeitet werden.

Wie aus Abbildung 4.4 leicht ersichtlich ist, werden aktuell nur einfache Select-Anfragen unterstützt. Wir haben uns gegen die Verwendung von geschachtelten Select-Anfragen entschieden, um die Komplexität der Anfragen niedrig zu halten.

[1] Viele der existierenden Datenbankverwaltungssysteme erzeugen automatisch Indizes für Unique-Spalten.

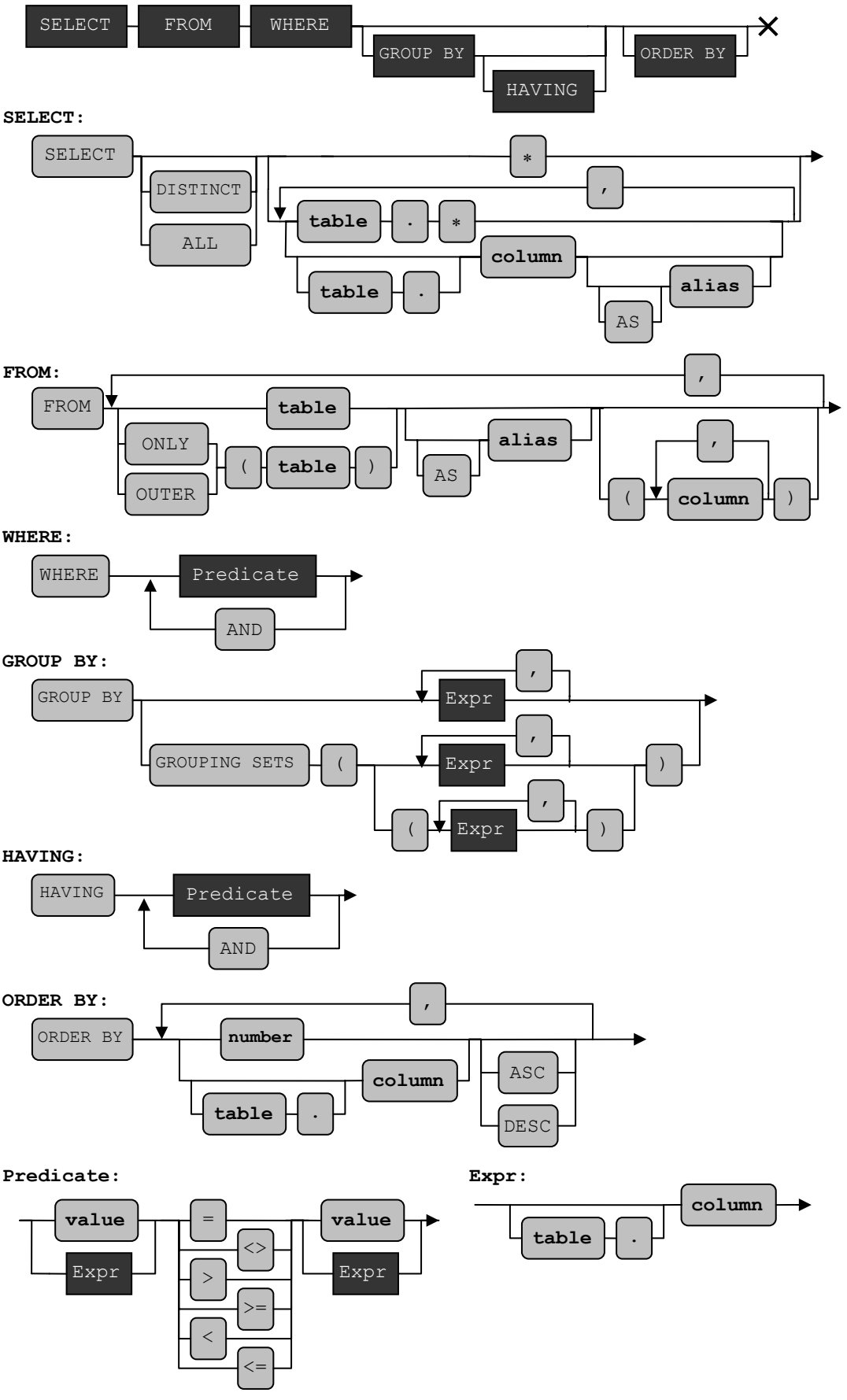


Abbildung 4.4 – Syntax der unterstützten SQL-Grammatik

Entspricht eine Anfrage der in Abbildung 4.4 dargestellten Syntax, so wird die Anfrage vom Query Worker in mehreren Schritten weiterverarbeitet. Zum besseren Verständnis wollen wir die einzelnen Bearbeitungsschritte an einem ausführlichen Beispiel durchführen. Dazu betrachten wir die folgende Anfrage:

```
SELECT r.name, titel, Genre.name
FROM Regisseur r, Film, Genre
WHERE r.id = regie
AND genre = Genre.id
AND r.id = '4711'
ORDER BY Genre.name ASC
```

Nachdem überprüft wurde, dass die Anfrage der SQL-Syntax entspricht, wird die Anfrage in ihre einzelnen Klauseln zerlegt. Der Query Worker extrahiert zuerst alle Tabellennamen inklusive Aliasnamen der From-Klausel und speichert diese in einer Liste. Als Nächstes wird die Where-Klausel untersucht, und es werden alle Prädikate ausgelesen und einzeln analysiert.

Für alle Prädikate der Form $Spalte = Spalte$ muss der Query Worker überprüfen, ob ein entsprechender RCC zwischen diesen Spalten existiert. Die Reihenfolge, in der die Spalten im Prädikat auftreten, spielt dabei keine Rolle, da sowohl die eingehenden als auch ausgehenden RCCs der Spalten überprüft werden.

An dieser Stelle erleichtert die eingeführte Datenstruktur uns die Arbeit: Da wir sowohl die eingehenden als auch die ausgehenden RCCs im Spalten-Objekt gespeichert haben, müssen wir lediglich eine der beiden Spalten zur Analyse betrachten.

Existiert ein RCC zwischen diesen beiden Spalten, so erzeugt der Query Worker einen gerichteten Graphen, in dem die zu den Spalten gehörenden Tabellen als Knoten eingefügt werden. Jeder Knoten besitzt eine Referenz auf eine Tabelle, daher bezeichnen wir diese Knoten im Folgenden als Tabellenknoten. Die beiden Tabellenknoten werden entsprechend der Richtung des RCC mit einer Kante verbunden. Existiert bereits einer oder beide Tabellenknoten im Graph, so werden diese beiden Tabellenknoten entsprechend dem RCC durch eine Kante verbunden.

Für das konkrete Beispiel bedeutet dies, dass nach Verarbeitung des Prädikats $r.id = regie$ der Query Worker den in Abbildung 4.5a dargestellten Graph erzeugt hat. Für das Prädikat $genre = Genre.id$ existiert ebenfalls ein RCC und der Query Worker speichert die zugehörigen Tabellen wieder als Tabellenknoten im Graph. Der Tabellenknoten $Film$ ist bereits im Graph enthalten und somit wird der Tabellenknoten $Genre$ durch eine Kante mit dem bestehenden Knoten $Film$ verbunden. Das Ergebnis ist in Abbildung 4.5b dargestellt.

Das letzte Prädikat der Where-Klausel besitzt die Form $Spalte = Wert$ und wird vom Query Worker als potenzieller Einstiegspunkt behandelt. Für jeden potenziellen Einstiegspunkt wird eine Sondierung durchgeführt. Ist die Spalte eine Unique-Spalte, so

führen wir die Sondierung direkt auf dieser Spalte durch. Wenn die Spalte keine Unique-Spalte ist, wird eine Sondierung auf jeder Quellspalte der eingehenden RCCs durchgeführt. Sobald die Sondierung für eine der Quellspalten positiv ausfällt, brechen wir den Vorgang ab.

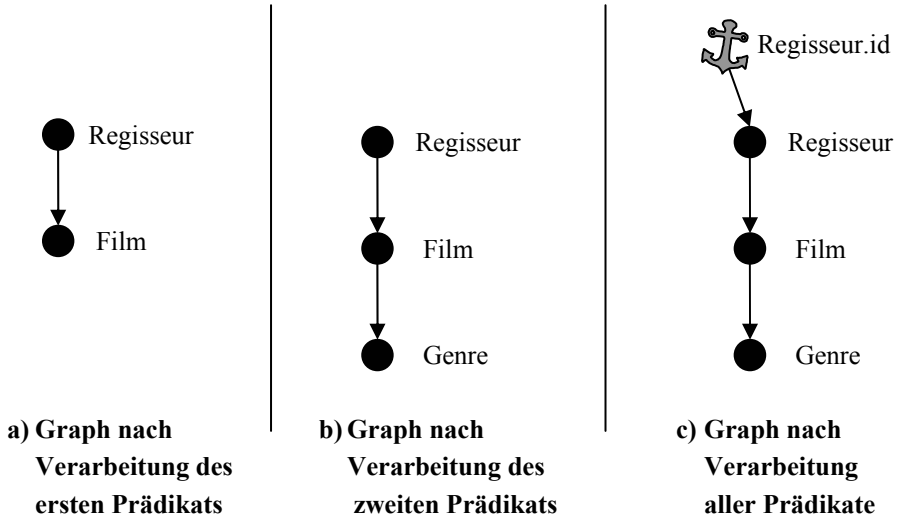


Abbildung 4.5 – Verankerung der Cache-Tabellen

Die Anfragen, mit der wir die Sondierung in beiden Fällen durchführen, besitzen die in Abbildung 4.6 dargestellte Form. Da in SQL boolesche Ausdrücke nicht direkt ausgewertet werden können, mussten wir diese Form für die Anfrage wählen, damit sie einerseits den gewünschten Effekt erzielt und andererseits die Bearbeitung der Anfrage wenig Resourcen benötigt. Die Existenz-Anfragen werden während der Initialisierungsphase für jeden potenziellen Einstiegspunkt in die Cache Group generiert. Als potenzielle Einstiegspunkte gelten dabei alle Unique-Spalten und Quellspalten von RCCs.

```
SELECT 1
FROM TABLE(VALUES 1) AS
tmp
WHERE EXISTS
(SELECT *
 FROM <CacheTable>
 WHERE <Column> = ?)
```

Abbildung 4.6 – Aufbau der Existenz-Anfragen

Liefert die Sondierung ein positives Ergebnis, so können wir die zur Spalte gehörende Tabelle als Einstiegspunkt in die Cache Group verwenden und verankern (Anker in Abbildung 4.5c). Der Query Worker markiert diese Tabelle als Einstiegspunkt und überprüft, analog zu den Prädikaten $Spalte = Spalte$, ob ein entsprechender Tabellenknoten bereits im Graphen enthalten ist und stellt gegebenenfalls eine entsprechende Kante her. Alle Tabellenknoten, die von einem Einstiegspunkt erreichbar sind, sind automatisch verankert. Je nach Anfrage können auch mehrere Einstiegspunkte mit

angehängten Tabellenknoten existieren. Ebenso kann es auch vorkommen, dass ein Tabellenknoten von keinem Einstiegspunkt erreicht wird. Alle verankerten Tabellen werden nach der Bearbeitung der Prädikate zu einer Menge zusammengefasst.

Angenommen, für unser Beispiel liefert die Sondierung ein positives Ergebnis, dann besitzt der Graph nach der Verarbeitung aller Prädikate die in Abbildung 4.5c dargestellte Form.

Schlägt die Sondierung fehl, d. h., ist der gesuchte Wert nicht im Cache vorhanden, so überprüft der Query Worker, ob die Spalte eine Füllspalte ist. Bei einem fehlenden Wert in einer Füllspalte bittet der Query Worker den Fill Daemon, den fehlenden Wert einzulagern. Nachdem der Wert in den Cache eingefügt wurde, steht er für die folgenden Anfragen zur Verfügung (vgl. Abschnitt 4.5).

Nachdem alle Prädikate der Where-Klausel analysiert wurden und die verankerten Tabellen bestimmt wurden, wird die ursprüngliche Anfrage des Benutzers umgeschrieben. Dazu durchläuft der Query Worker die gesamte Anfrage und überprüft für jeden gefundenen Tabellennamen, ob dieser in der Menge der verankerten Tabellen enthalten ist. Ist der gefundene Tabellenname verankert, so wird dieser durch den Namen der entsprechenden Cache-Tabelle ersetzt.

Nach dem Durchlauf der kompletten Anfrage und der Ersetzung aller verankerten Tabellen führt der Query Worker die modifizierte Anfrage über die Middle-Tier-Komponente aus und liefert das Ergebnis über die JDBC-Schnittstelle an den Benutzer zurück.

4.4.2 Bearbeitung von Änderungsoperationen

Wie bereits erwähnt unterstützt die aktuelle Version des CBCS keine Änderungsoperationen. Dennoch möchten wir an dieser Stelle einige Vorgehensweisen diskutieren und auch, welche Erweiterungen notwendig sind, um diese Funktionalität zu unterstützen.

Im Allgemeinen sind Änderungsoperationen sehr aufwendig, da sie sowohl auf der Backend-Datenbank als auch auf den betroffenen Cache-Datenbanken durchgeführt werden müssen. Um die eigentliche Aufgabe des Cache, das Beschleunigen von lesenden Operationen, nicht allzu sehr zu beeinträchtigen, gehen wir davon aus, dass nur wenige Änderungsoperationen auf den Cache-Tabellen durchgeführt werden.

In Szenarien, in denen viele Änderungsoperationen auf bestimmten Tabellen durchgeführt werden, ist es sinnvoll, diese Tabellen nicht als Cache-Tabellen zu spezifizieren. Dadurch werden alle Änderungsoperationen automatisch an die Backend-Datenbank weitergeleitet und eine aufwendige Aktualisierung der Datensätze in den Cache-Datenbanken entfällt.

Ein weiterer wichtiger Punkt ist die Konsistenzanforderung an die Datensätze im

Cache. Sollen diese immer den aktuellen Zustand der Backend-Datenbank repräsentieren, so ist ein sehr hoher Synchronisationsaufwand notwendig.

Im Folgenden stellen wir zwei Verfahren vor, die beim Einsatz von Änderungsoperationen angewendet werden können. Bei beiden Verfahren müssen je nach Änderungsoperation bestimmte Maßnahmen auf dem Cache durchgeführt werden. Diese Maßnahmen sind für beide Verfahren identisch, jedoch unterscheiden sich die Zeitpunkte, zu denen sie durchgeführt werden. Die folgenden Maßnahmen müssen nach einer Änderungsoperation auf dem Cache durchgeführt werden:

(M1) Insert-Operation: Ein eingefügter Datensatz in der Backend-Datenbank muss nur dann in den Cache eingelagert werden, wenn mindestens ein eingehender RCC auf einer der Spalten der Cache-Tabelle definiert ist und der Wert des eingefügten Datensatzes in der Quelltabelle des RCCs vorhanden ist. Zusätzlich müssen natürlich alle ausgehenden RCCs überprüft werden und eventuelle Datensätze eingelagert werden, sodass der Cache wieder in einen gültigen Zustand erhält.

(M2) Delete-Operation: Wenn der Benutzer einen Datensatz in der Backend-Datenbank löscht, so muss dieser Datensatz auch im Cache gelöscht werden. Das Löschen dieses Datensatzes im Cache kann analog zum Löschen in der Backend-Datenbank geschehen. Eventuell müssen weitere Datensätze entlang der ausgehenden RCCs der Tabelle gelöscht werden, damit keine „Datenleichen“ im Cache zurückbleiben, die später vom Garbage Collector nicht mehr erreicht werden können.

(M3) Update-Operation: Bei der Änderung von Datensätzen müssen wir zwei Fälle unterscheiden:

1. Das Update bezieht sich nur auf Spalten, die weder Quell- noch Zielspalte eines RCCs sind. In diesem Fall können wir das Update ohne weitere Maßnahmen auf dem Cache ausführen.
2. Bezieht sich das Update auf mindestens eine Spalte, die Quell- oder Zielspalte eines RCCs ist, so bestimmen wir die Datensätze, die von der Änderung betroffen sind, und führen das Update auf diesen Datensätzen durch. Anschließend betrachten wir alle Spalten dieser Datensätze, von denen mindestens ein RCC ausgeht, und sorgen dafür, dass die entsprechenden Join-Partner in die Zieltabellen eingelagert werden. Diesen Vorgang wiederholen wir auf allen Cache-Tabellen, die über RCCs von der Tabelle, auf der die Änderungsoperation durchgeführt wurde, erreichbar sind. Dabei müssen wir, je nachdem wie der Garbage Collector verwaiste Datensätze behandelt, auch dafür sorgen, dass verwaiste Datensätze während der Update-

Operation gelöscht werden. Im Gegensatz zur Einlagerung von Werten in die Füllspalten (vgl. Abschnitt 4.5.1) müssen hier alle Einfügeoperationen (und ggf. die Löschoperationen) in derselben Transaktion wie die Update-Operation geschehen, da der Cache andernfalls zwischenzeitlich ungültige Zustände besitzt.

Asynchrone Aktualisierung

Bei diesem Verfahren werden die einzelnen Cache-Datenbanken zu einem späteren Zeitpunkt als die Backend-Datenbank aktualisiert. Dazu ist es notwendig, dass die Backend-Datenbank alle vorhandenen Cache-Datenbanken kennt, um diese bei einer Änderungsoperation zu benachrichtigen. Diese Funktionalität kann über eine zusätzliche Komponente, nennen wir sie einmal Synchronizer, realisiert werden, die sich in der Nähe der Backend-Datenbank befindet und bei der sich alle CBCS beim Start registrieren.

Eine Änderungsoperation in einem CBCS wird an den Synchronizer weitergeleitet, der diese auf der Backend-Datenbank ausführt. Im Anschluss an die Änderungen im Backend, benachrichtigt der Synchronizer alle registrierten CBCS, die daraufhin die entsprechenden Maßnahmen (M1, M2, M3) durchführen.

Da der Synchronizer immer alle CBCS informiert, kann es vorkommen, dass Nachrichten an eine Cache-Datenbank geschickt werden, die die entsprechenden Tabellen gar nicht beinhaltet. Mit einer kleinen Optimierung kann jedoch auch dieses Problem behoben werden. Dazu muss der Synchronizer zu jeder Cache-Datenbank auch die vorhandenen Cache-Tabellen kennen. Bei einer Änderungsoperation kann der Synchronizer anhand dieser Informationen alle CBCS bestimmen, die die geänderte Backend-Tabelle als Cache-Tabelle besitzen und diese gezielt benachrichtigen.

Die Aktualisierung auf dem Cache geschieht eine Zeitspanne δ nach der Ausführung der Änderungsoperationen auf der Backend-Datenbank, wobei δ nach Möglichkeit klein sein sollte. Dadurch kann es passieren, dass ein Benutzer seine eigenen Änderungen nicht sieht. Dieses Problem kann beispielsweise durch das nachfolgende Verfahren verhindert werden.

2-Phasen-Commit

Bei diesem Verfahren wird die Änderungsoperation sowohl auf der Backend-Datenbank als auch auf der Cache-Datenbank, über deren CBCS die Änderungsoperation aufgetreten ist, gleichzeitig ausgeführt. Auch bei diesem Verfahren benötigen wir wieder einen Synchronizer in der Nähe der Backend-Datenbank sowie die Registrierung der einzelnen CBCS beim Start. Die im vorhergehenden Abschnitt beschriebene Optimierung des Synchronizer kann auch bei diesem Verfahren eingesetzt werden.

Erhält ein CBCS eine Änderungsoperation von einem Benutzer, so leitet es diese an

den Synchronizer weiter. Dieser ändert innerhalb der Transaktion des Benutzers sowohl die Daten im Cache als auch in der Backend-Datenbank. Dabei muss das CBCS auf Cache-Seite, je nach Änderungsoperation (Insert, Update, Delete), folgende Aktionen durchführen:

- Insert-Operation: Bei Insert-Operationen kann das CBCS die üblichen Maßnahmen (M1) durchführen, d. h., falls die neuen Datensätze relevant sind, so werden sie eingefügt. Anschließend muss der Cache wieder in einen gültigen Zustand gebracht werden.
- Delete-Operationen: Analog zu Maßnahme M2 kann das CBCS bei Delete-Operationen verfahren.
- Update-Operationen: Änderungsoperationen, die durch ein Update hervorgerufen werden, benötigen eine spezielle Behandlung. Das CBCS muss zuerst alle betroffenen Datensätze im Cache bestimmen und gegebenenfalls aus dem Backend die aktuellste Version nachladen. Anschließend werden auf den aktualisierten Datensätzen die Maßnahmen für Update-Operationen (M3) auf dem Cache und das Update auf der Backend-Datenbank durchgeführt.

Beendet der Benutzer die Transaktion durch ein Commit, so wird zwischen der Backend-Datenbank und dem Cache ein 2-Phasen-Commit-Protokoll durchgeführt. Anschließend werden wie beim asynchronen Verfahren alle übrigen registrierten CBCS vom Synchronizer benachrichtigt.

Dieses Verfahren ermöglicht Benutzern, ihre eigenen Änderungen direkt zu sehen, dafür ist der Aufwand für die Änderungsoperation höher als beim asynchronen Verfahren.

Welchen Ansatz man schließlich im CBCS realisiert, hängt unter anderem auch von der gewünschten Konsistenz des Cache ab. Die beiden hier vorgestellten Verfahren sollen daher eher als Beispiele dienen.

4.5 Der Fill Daemon

Der Fill Daemon ist ein eigenständiger Prozess, der während der Initialisierungsphase eingerichtet und gestartet wird. Seine Aufgabe ist das Füllen des Cache mit Datensätzen. Der Fill Daemon wird von den Query-Worker-Instanzen informiert, sobald eine Anfrage einen Wert in einer Füllspalte verlangt, der nicht im Cache vorhanden ist. Dazu schreibt der Query Worker eine Nachricht, in der der benötigte Wert vermerkt ist, und stellt diese Nachricht in die Warteschlange des Fill Daemon. Die Warteschlange wird vom Fill Daemon abgearbeitet und für jede Nachricht in der Warteschlange werden die entsprechenden Datensätze aus der Backend-Datenbank in den Cache eingelagert.

Das Einlagern der Datensätze muss mit großer Sorgfalt geschehen, da der Cache nach jedem Einlagerungsvorgang wieder einen gültigen Zustand besitzen muss. Nur so ist gewährleistet, dass die Query Worker korrekte Ergebnisse zurückliefern. Daher müssen nicht nur Datensätze in die Tabelle, die die Füllspalte enthält eingelagert werden, sondern zusätzlich auch in allen von dieser Tabelle über ausgehende RCCs erreichbare Cache-Tabellen.

4.5.1 Strategien zum Füllen des Cache

Die Insert-Anweisungen für den Fill Daemon werden während der Initialisierungsphase generiert und in einem Container gespeichert. An dieser Stelle wollen wir den Aufbau der Insert-Anweisungen genauer betrachten.

Alle Einlagerungen werden durch fehlende Werte in Füllspalten verursacht; daher benötigen wir Insert-Anweisungen für die Füllspalten und für alle Tabellen, die über ausgehende RCCs der Tabelle, in der die Füllspalte liegt, erreichbar sind. Wir nutzen die Mächtigkeit von SQL, um die Einlagerung der Datensätze für eine Tabelle in einer Anweisung durchzuführen. Die Komplexität verlagert sich dadurch von der Fill-Daemon-Komponente in die generierten Anweisungen.

Zum besseren Verständnis werden wir die Generierung der Insert-Anweisungen an dem in Abbildung 4.7 dargestellten Beispiel erläutern.

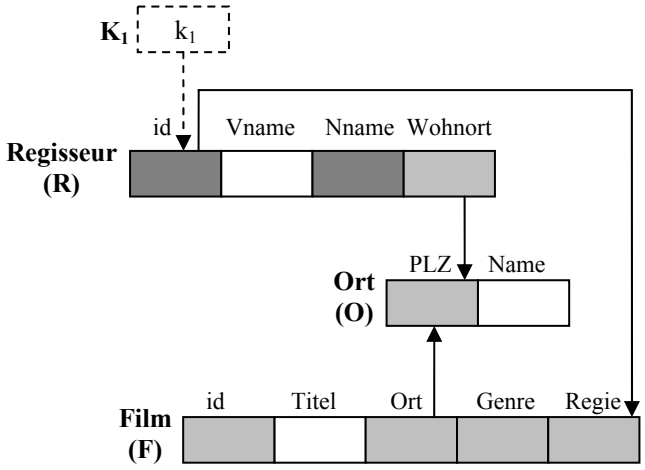


Abbildung 4.7 – Beispiel für die Generierung der Insert-Anweisungen

Gehen wir davon aus, dass die Spalte *Regisseur.id* die Füllspalte ist, für die wir die Insert-Anweisungen erstellen wollen. Da jede Füllspalte eine Kontrolltabelle besitzt, erzeugen wir zuerst die folgende Insert-Anweisung für die Kontrolltabelle.

```
INSERT INTO K1
      (ID, TIME, LASTACCESS, HITCOUNTER) VALUES (?, ?, ?, 0)
```

Als Nächstes wird die Insert-Anweisung für die Tabelle *Regisseur* generiert.

```
INSERT INTO REGISSEUR
      SELECT *
      FROM REGISSEUR_B
      WHERE ID = ?
      AND (ID) NOT IN (SELECT ID FROM REGISSEUR)
```

Dabei lagern wir nur die Datensätze aus der Backend-Datenbank ein, die den Wert des Parameters in der entsprechenden Spalte besitzen. Die letzte Zeile der Anweisung sorgt dafür, dass von allen Datensätzen aus dem Backend, die sich für die Einlagerung qualifiziert haben, nur diejenigen eingelagert werden, die noch nicht im Cache vorhanden sind. Somit können keine Fehlermeldungen durch Einfügen von doppelten Datensätzen entstehen.

Als Nächstes verfolgen wir rekursiv alle ausgehenden RCCs der Tabelle *Regisseur*. Dabei übergeben wir den kursiv gedruckten Anteil (Where-Klausel) der Insert-Anweisung dem nächsten Rekursionsschritt. Dieser Anweisungsteil beschreibt den Weg, den wir über die RCCs zurückgelegt haben. Dadurch ist es möglich, dass wir für jede Insert-Anweisung denselben Parameterwert verwenden können. Für jede erreichte Tabelle entlang eines RCC-Pfades erzeugen wir eine Insert-Anweisung und verfolgen anschließend deren ausgehende RCCs. Die Erzeugungsvorschrift für die Where-Klausel in Rekursionsschritt n ist dabei wie folgt:

```
WHERE <RCC Zielspalte> IN
            (SELECT <RCC Quellspalte>
             FROM <Backend-Tabelle von RCC Quelltabelle>
             <übergebene Where-Klausel aus Schritt n-1>)
   AND (<Primary Key>) NOT IN
            (SELECT <Primary Key> FROM <RCC Zieltabelle>)
```

Für unser Beispiel bedeutet dies: Wir erreichen über den RCC $R.id \rightarrow F.Regie$ die Tabelle *Film*. Die generierte Anweisung hat die folgende Form:

```
INSERT INTO FILM
      SELECT *
      FROM FILM_B
      WHERE REGIE IN
            (SELECT ID FROM REGISSEUR_B WHERE ID = ?)
      AND (ID) NOT IN (SELECT ID FROM FILM)
```

Diese Anweisung ist im Aufbau mit der Insert-Anweisung für die Tabelle *Regisseur* identisch. Lediglich der kursiv dargestellte Anteil der Anfrage wurde erweitert. Dieser erweiterte Anteil wird in den nächsten rekursiven Schritt weitergegeben. Dadurch wächst die Anfrage linear mit der Länge des verfolgten RCC-Pfads. Hat eine Tabelle keine ausgehenden RCCs so endet der RCC-Pfad. Wird beim Verfolgen der ausgehenden RCCs einer Tabelle festgestellt, dass ein Zyklus vorhanden ist, d. h. ein RCC wurde schonmal verfolgt, so wird dieser RCC ausgeschlossen und nur noch die

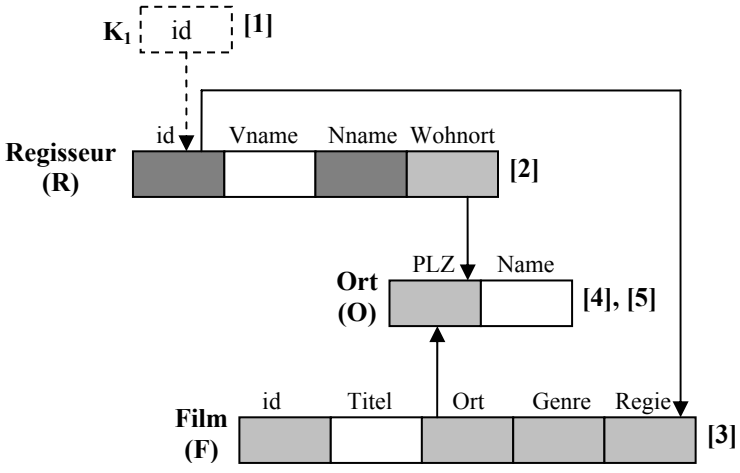
restlichen ausgehenden RCCs weiter verfolgt. Handelt es dabei um einen heterogenen Zyklus (vgl. Abschnitt 2.3.1), so wird eine Fehlermeldung erzeugt und an die Starter-Komponente weitergeben. Der Abbruch nach *einem* Durchlauf des Zyklus ist auch im Falle von homogenen Zyklen korrekt. Wir haben in Abschnitt 2.3.1 gesehen haben, dass bei einem homogenen Zyklus nach einem Durchlauf keine neuen Datensätze in die am Zyklus beteiligten Cache-Tabellen gelangen.

In unserem Beispiel verfolgen wir, nach der Generierung der Insert-Anweisung für die Tabelle *Film*, den RCC $F.Ort \rightarrow O.PLZ$. Die Insert-Anweisung für die Tabelle *Ort* sieht wie folgt aus:

```
INSERT INTO ORT
      SELECT *
      FROM ORT_B
      WHERE PLZ IN
            (SELECT ORT FROM FILM_B WHERE REGIE IN
                  (SELECT ID FROM REGISSEUR_B WHERE ID = ?) )
   AND (PLZ, NAME) NOT IN (SELECT PLZ, NAME FROM ORT)
```

Da die Tabelle *Ort* keine ausgehenden RCCs besitzt, endet der RCC-Pfad und wir fallen zurück auf die Tabelle *Film*. Auch diese besitzt keine weiteren ausgehenden RCCs und wir fallen zurück auf die Tabelle *Regisseur*. Dort existiert noch ein ausgehender RCC, den wir anschließend verfolgen.

Abbildung 4.8 fasst noch mal graphisch die Reihenfolge zusammen, in der wir durch die Cache Group gewandert sind und die Insert-Anweisungen generiert haben.



[x]: Reihenfolge der Generierung

Abbildung 4.8 – Generierungsreihenfolge der Insert-Anweisungen

Alle generierten Insert-Anweisungen für das in Abbildung 4.7 dargestellte Beispiel sind in Abbildung 4.9 nochmals kurz zusammengefasst. Die Werte in den eckigen Klammern am Anfang jeder Anweisung beziehen sich auf die in Abbildung 4.8 dargestellte Generierungsreihenfolge.

Nach der Generierung der Anweisungen für eine Füllspalte werden die Insert-Anweisungen in eine geordnete Liste gespeichert und in einem Container hinterlegt. Der Fill Daemon entnimmt dem Container während der Laufzeit die benötigte Liste mit Insert-Anweisungen und führt die Anweisungen aus. Dabei spielt die Ausführungsreihenfolge für die Effizienz des Cache eine große Rolle, wie wir in den folgenden beiden Abschnitten zeigen werden.

```
[1] INSERT INTO K1
      (id, time, lastaccess, hitcounter) VALUES (?, ?, ?, 0)

[2] INSERT INTO REGISSEUR
      SELECT *
      FROM REGISSEUR_B
      WHERE ID = ?
      AND (ID) NOT IN (SELECT ID FROM REGISSEUR)

[3] INSERT INTO FILM
      SELECT *
      FROM FILM_B
      WHERE REGIE IN
            (SELECT ID FROM REGISSEUR_B WHERE ID = ?)
      AND (ID) NOT IN (SELECT ID FROM FILM)

[4] INSERT INTO ORT
      SELECT *
      FROM ORT_B
      WHERE PLZ IN
            (SELECT ORT FROM FILM_B WHERE REGIE IN
                  (SELECT ID FROM REGISSEUR_B WHERE ID = ?) )
      AND (PLZ, NAME) NOT IN (SELECT PLZ, NAME FROM ORT)

[5] INSERT INTO ORT
      SELECT *
      FROM ORT_B
      WHERE PLZ IN
            (SELECT WOHNORT FROM REGISSEUR_B WHERE ID = ?)
      AND (PLZ, NAME) NOT IN (SELECT PLZ, NAME FROM ORT)
```

Abbildung 4.9 – Aufbau der generierten Insert-Anweisungen

Top-Down-Ansatz

Beim Top-Down-Ansatz führen wir beim Einlagern zuerst die Insert-Anweisung auf der Kontrolltabelle aus und steigen rekursiv hinab, entlang der ausgehenden RCCs, d. h., wir führen die Insert-Anweisungen in derselben Reihenfolge aus, wie sie generiert wurden.

Der ganze Einlagerungsvorgang muss dabei innerhalb einer Transaktion geschehen, da nach dem Einfügen von Datensätzen in eine Cache-Tabelle im Normalfall die ausgehenden RCCs dieser Tabelle verletzt sind, d. h., der Cache besitzt einen ungültigen Zustand. Jeder Einfügevorgang auf einer Tabelle verursacht jedoch eine Schreibsperre auf dieser Tabelle, was im schlimmsten Fall zu Schreibsperren auf allen Cache-Tabellen führen kann.

Gehen wir einmal von der höchsten Konsistenzstufe im Datenbanksystem aus, so können die Query Worker während der Einlagerung keine Anfragen auf einem Teil der Cache-Tabellen ausführen, da alle lesenden Anfragen vom Datenbanksystem blockiert werden. Damit die Effizienz des CBCS nicht zu sehr unter den Einlagerungsvorgängen leidet, sollten die Schreibsperren auf einer Cache-Tabelle so kurz wie möglich sein. Diese Idee verfolgt der nachfolgend beschriebene Bottom-Up-Ansatz.

Bottom-Up-Ansatz

Wir haben gesehen, dass der Top-Down-Ansatz sehr lange Schreibsperren auf den Cache-Tabellen verursachen kann. Eine Möglichkeit, dies zu umgehen, ist der im CBCS verwendete Bottom-Up-Ansatz. Wir gehen wieder davon aus, dass die höchste Konsistenzstufe im Cache eingestellt ist.

Bevor wir beim Bottom-Up-Ansatz die Generierung der Insert-Anweisungen beginnen, müssen wir zunächst einige vorbereitende Maßnahmen durchführen, um später die Korrektheit des Einlagerungsvorgangs garantieren zu können. Zuerst teilen wir die Cache Group in atomare Zonen ein. Eine atomare Zone enthält eine oder mehrere Cache Tabellen, die während des Einlagerungsvorgangs innerhalb einer Transaktion gefüllt werden müssen. Die Einteilung der Cache Group in atomare Zonen ist notwendig, da wir einerseits keine langen Schreibsperren auf den Tabellen verursachen wollen, und andererseits alle an einem homogenen Zyklus beteiligten Tabellen gleichzeitig gefüllt werden müssen, um ungültige Cache-Zustände zu vermeiden. Die ungültigen Cache-Zustände entstehen durch die zyklische Referenzierung der Datensätze in den Cache-Tabellen.

Die Einteilung der Cache Group in atomare Zonen ist relativ einfach. Zu Beginn befindet sich jede Cache-Tabelle in einer eigenen atomaren Zone. Anschließend durchlaufen wir alle RCC-Pfade der Cache Group. Entdecken wir dabei zwischen zwei oder mehreren Tabellen einen homogenen Zyklus, so werden die atomaren Zonen, in denen sich die Tabellen befinden zu einer Zone zusammengefasst. Wird beim Durchlaufen der RCC-Pfade ein heterogener Zyklus gefunden, so bricht die Suche mit einer Fehlermeldung ab.

Nach der Einteilung der Cache Group in atomare Zonen muss eine Ausführungsreihenfolge unter den Zonen festgelegt werden. Dazu sortieren wir die Zonen topologisch, indem wir die Abhängigkeiten der Zonen untereinander betrachten.

Anschließend werden die Insert-Anweisungen gemäß dem zu Anfang dieses Abschnitts beschriebenen Schema erzeugt und in einem Container gespeichert. Zur Laufzeit führt der Fill Daemon die für den Einlagerungsvorgang benötigten Anweisungen aus.

Wird das zu Anfang dieses Abschnitts beschriebene Generierungsschema verwendet, so ist die Ausführungsreihenfolge der Anweisungen innerhalb einer atomaren

Zone egal, da wir in jeder Anweisung den zurückgelegten RCC-Pfad von der Kontrolltabelle gespeichert haben. Bei Einsatz eines anderen Generierungsschemas kann die Ausführungsreihenfolge jedoch eine Rolle spielen, da beispielsweise für das Einlagern von Datensätzen Informationen über die Datensätze in der Quelltabelle des RCCs benötigt werden.

Die Einlagerung der Datensätze beim Bottom-Up-Ansatz erfolgt in umgekehrter Reihenfolge zum Top-Down-Ansatz, d. h., wir führen zuerst die Insert-Anweisungen der letzen atomaren Zone der Liste aus und beenden die Einlagerung mit der atomaren Zone, die die Insert-Anweisung für die Kontrolltabelle enthält.

Die eben beschriebenen Schritte des Bottom-Up-Ansatzes wollen wir an dem Beispiel aus Abbildung 4.7 nochmals verdeutlichen. Abbildung 4.10 zeigt die Ausgangssituation, nachdem jede Cache-Tabelle einer atomaren Zone zugeordnet wurde.

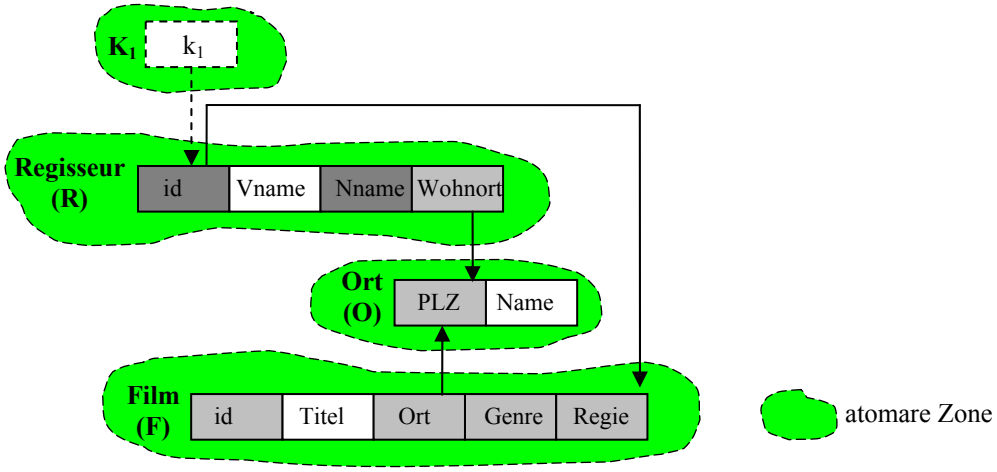


Abbildung 4.10 – Einteilung der Cache-Tabellen in atomare Zonen

Da das Beispiel keinen homogenen Zyklus enthält, können beim Durchlaufen der RCC-Pfade keine Zonen zusammengefasst werden. Die topologische Sortierung der Zonen ist in Abbildung 4.11 dargestellt.

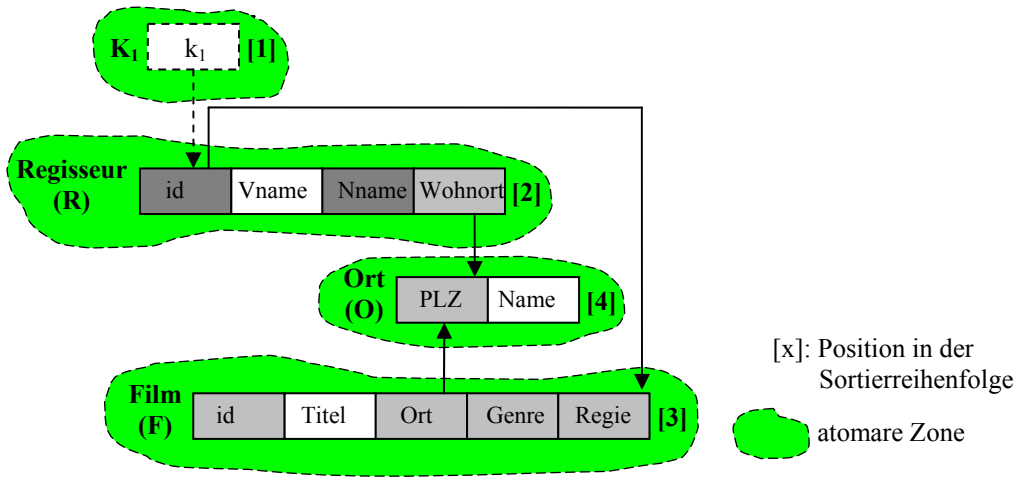
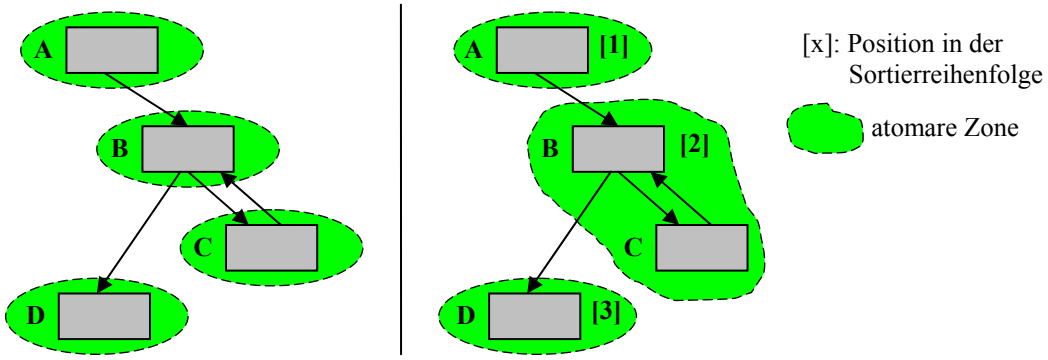


Abbildung 4.11 – Topologische Sortierung der atomaren Zonen

Die Zahlen in eckigen Klammern geben die Sortierreihenfolge der atomaren Zonen an. Im Anschluss werden die Insert-Anweisungen, wie in Abbildung 4.9 dargestellt, generiert und den einzelnen atomaren Zonen zugeordnet. Für das Beispiel bedeutet dies, dass jede Zone, mit Ausnahme der Zone 4, eine Insert-Anweisung enthält. Die Tabelle *Ort* in Zone 4 wird über zwei RCC-Pfade erreicht und enthält daher zwei generierte Anweisungen.

Die Cache Group im eben erläuterten Beispiel hatte keinen Zyklus, daher wurden während des Durchlaufs der RCC-Pfade keine atomaren Zonen zusammengefasst. Abbildung 4.12 zeigt eine Cache Group mit einem homogenen Zyklus. Wie beim vorherigen Beispiel befindet sich jede Cache-Tabelle zu Beginn in einer eigenen atomaren Zone (Abbildung 4.12a). Beim Durchlaufen der RCC-Pfade wird der Zyklus zwischen den Tabellen *B* und *C* erkannt und die beiden atomaren Zonen dieser Tabellen zusammengefasst. Anschließend liefert die topologische Sortierung die in Abbildung 4.12b dargestellte Reihenfolge. Die weitere Generierung der Insert-Anweisungen erfolgt analog zum vorherigen Beispiel.



a) Initiale Einteilung der atomaren Zonen

b) Atomare Zonen nach Zusammenfassung und topologischer Sortierung

Abbildung 4.12 – Einteilung einer Cache Group mit Zyklus in atomare Zonen

Ist während der Laufzeit die Einlagerung eines Wertes in die Kontrolltabelle K_1 notwendig, so entnimmt der Fill Daemon aus einem Container die entsprechende Liste mit Insert-Anweisungen und führt die Insert-Anweisungen der einzelnen Zonen, beginnend bei der letzten Zone aus, d. h. für unser Beispiel, dass der Fill Daemon zuerst die Anweisungen von Zone 4 ausführt, dann Zone 3 und so fort.

Dieser Ansatz hat den entscheidenden Vorteil, dass der Cache nach der Ausführung aller Insert-Anweisungen einer atomaren Zone einen gültigen Zustand besitzt, d. h., es werden nur Schreibsperren für die Cache-Tabellen der aktuellen atomaren Zone benötigt. Dadurch werden lange Schreibsperren vermieden und die Query Worker werden nicht unnötig bei ihrer Arbeit blockiert. Durch die Ausführung aller Insert-Anweisungen einer Zone innerhalb einer Transaktion und die topologische Sortierung der Zonen, befindet sich der Cache stets in einem gültigen Zustand. Durch die Einteilung der Cache

Group in Zonen, deren Cache-Tabellen innerhalb einer Transaktion gefüllt werden, reduziert sich im Normalfall die Anzahl der gleichzeitigen Schreibsperren auf den Cache-Tabellen. Die topologische Sortierung der Zonen und die Reihenfolge, in der wir die Zonen abarbeiten, sorgen dafür, dass wir sicher sein können, dass sobald wir die Insert-Anweisungen einer atomaren Zone ausführen, bereits alle Insert-Anweisungen der nachfolgenden atomaren Zonen in der Liste ausgeführt worden sind.

In Verbindung mit dem verwendeten Sondierungsverfahren ergibt sich ein weiterer Vorteil des Bottom-Up-Ansatzes, nämlich dass die eingelagerten Datensätze einer Cache-Tabelle bereits als Einstiegspunkte verwendet werden können, bevor der komplette Einlagerungsvorgang abgeschlossen ist. Dieser Vorteil entsteht durch die topologische Sortierung der atomaren Zonen und der Tatsache, dass wir die Insert-Anweisungen einer atomaren Zone in einer Transaktion ausführen. Sobald die neuen Datensätze in die Cache-Tabellen einer atomaren Zone eingelagert wurden, ist durch die Sortierung der Zonen sichergestellt, dass die Einlagerung der Datensätze in den durch ausgehende RCCs referenzierten Tabellen bereits abgeschlossen ist. Diese Datensätze in den Zieltabellen der RCCs können mit dem verwendeten Sondierungsverfahren zu diesem Zeitpunkt bereits wieder als Einstiegspunkte dienen. Auch Equijoins entlang der vom Einstiegstiegspunkt ausgehenden RCCs können durch die topologische Sortierung korrekt ausgewertet werden.

Einen Nachteil hat jedoch auch der Bottom-Up-Ansatz: Ein Großteil der Where-Klausel jeder Anweisung wird auf der Backend-Datenbank ausgewertet und daher können die definierten Indizes des Cache nicht verwendet werden. Schauen wir uns dazu die Insert-Anweisung für die Tabelle *Film* nochmals etwas genauer an:

```
INSERT INTO FILM
      SELECT *
      FROM FILM_B
      WHERE REGIE IN
            (SELECT ID FROM REGISSEUR_B WHERE ID = ?)
      AND (ID) NOT IN (SELECT ID FROM FILM)
```

Die Tabellen, die wir im kursiv markierten Teil der Where-Klausel verwenden, beziehen sich alle auf Tabellen der Backend-Datenbank. Da wir den Cache von unten nach oben füllen, wissen wir nicht, welche Datensätze bzw. welche Werte in den Quellspalten der RCCs existieren. Beim Top-Down-Ansatz sind diese Datensätze bzw. diese Werte zum Ausführungszeitpunkt der Anweisung bereits im Cache vorhanden. Wir könnten daher an den betreffenden Stellen die Backend-Tabellen durch Cache-Tabellen ersetzen. Die Auswertung des kursiven Teils der Where-Klausel würde beim Bottom-Up-Ansatz immer eine leere Menge liefern. Daher müssen wir diesen Teil der Anweisung auf der Backend-Datenbank ausführen. Die Backend-Datenbank besitzt jedoch im Normalfall keine Indizes auf den verwendeten Spalten. Die Suche der Datensätze in den Backend-Tabellen geschieht daher durch Tabellenscans und die Auswertung der

Join-Operationen durch Nested-Loop-Joins. Dies ist zwar nicht sehr effizient, jedoch kann durch entsprechende Maßnahmen auf der Backend-Datenbank, beispielsweise Erzeugung von Indizes auf denselben Spalten wie im Cache, die Effizienz noch gesteigert werden.

Bevor wir diesen Abschnitt abschließen, wollen wir an dieser Stelle noch einen alternativen Einlagerungsansatz vorstellen. Wir haben bisher die ganze Komplexität der Einlagerung in die generierten Anweisungen verlegt. Alternativ kann jedoch die Komplexität auch in den Fill Daemon gelegt werden. Der Einlagerungsvorgang läuft in diesem Fall wie folgt ab:

Der Fill Daemon fügt den benötigten Wert in der Kontrolltabelle ein. Anschließend lagert er alle Datensätze mit dem benötigten Wert aus der Backend-Datenbank für die Tabelle, die die Füllspalte enthält, ein. Die Werte in den Quellspalten der ausgehenden RCCs dieser Tabelle merkt sich der Fill Daemon und fordert danach die Datensätze mit diesen Werten für die Zieltabellen der RCCs an. Dort werden wieder die Werte in den Quellspalten der ausgehenden RCCs gesammelt und für die weitere Einlagerung verwendet. Die Einlagerung erfolgt bei dieser Vorgehensweise nach dem oben beschriebenen Top-Down-Ansatz, d. h., die komplette Einlagerung muss in einer Transaktion geschehen, was zu langen Schreibsperren auf den Cache-Tabellen führt. Zwar sind die Insert-Anweisungen für diesen Ansatz relativ einfach (Equijoins entlang des zurückgelegten RCC-Pfads entfallen), dafür müssen wir jedoch lange Schreibsperren in Kauf nehmen.

4.6 Der Hit Counter

Der Hit Counter ist wie der Fill Daemon ein eigenständiger Prozess, der die Aufgabe besitzt, Statistikdaten zu sammeln, die der Garbage Collector (vgl. Abschnitt 4.7) für seine Verdrängungsstrategie verwendet.

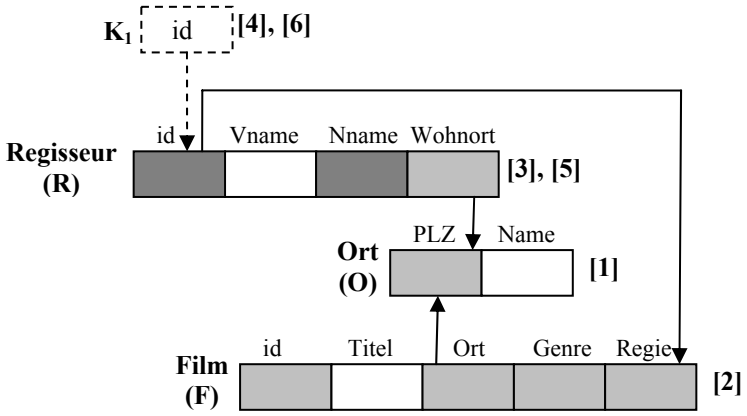
Auch der Hit Counter besitzt eine Warteschlange, die er kontinuierlich abarbeitet. Die Warteschlange wird durch die Query Worker gefüllt, die für jeden gefundenen Einstiegspunkt in der Anfrage eine Nachricht in der Warteschlange hinterlassen. Der Hit Counter entnimmt die Nachricht und führt auf allen Kontrolltabellen ein Update auf denjenigen Werten durch, die für die Einlagerung des Wertes, der als Einstiegspunkt dient, verantwortlich sind. Dabei wird in der Kontrolltabelle die Spalte *Hitcounter* um eins inkrementiert und in die Spalte *Lastaccess* die aktuelle Zeit eingetragen.

Um diese Updates auf den Kontrolltabellen durchzuführen, benötigt der Hit Counter entsprechende Anweisungen, die ähnlich den Insert-Anweisungen während der Initialisierungsphase generiert werden. Dies ist nur möglich, weil wir bereits während der Initialisierung wissen, welche Spalten als potenzielle Einstiegspunkte dienen können,

nämlich alle Unique-Spalten und Zielspalten von RCCs. Für jeden potenziellen Einstiegspunkt generieren wir die entsprechenden Update-Anweisungen.

Dazu beginnen wir bei der Zielspalte eines RCCs und steigen rekursiv entlang der eingehenden RCCs auf, bis wir eine Kontrolltabelle finden. Erreichen wir dabei dieselbe Kontrolltabelle mehrmals, so werden die einzelnen Anweisungen disjunktiv verknüpft. Erkennen wir beim Aufstieg einen Zyklus, so verfolgen wir diesen RCC-Pfad nicht weiter und erstellen auch keine Update-Anweisung.

Diese Vorgehensweise wollen wir wieder an einem Beispiel verdeutlichen.



[x]: Reihenfolge, in der die Cache-Tabellen besucht werden

Abbildung 4.13 – Beispiel für die Generierung der Update-Anweisungen

Angenommen wir beginnen bei der Spalte *Ort.PLZ* wie in Abbildung 4.13 dargestellt. Diese Spalte besitzt zwei eingehende RCCs. Zur Generierung der Update-Anweisung müssen wir daher beide RCCs verfolgen. Bevor wir jedoch die RCCs verfolgen, erzeugen wir eine Where-Klausel, die wie bereits bei den Insert-Anweisungen den zurückgelegten RCC-Pfad beschreibt. Diese Where-Klausel hat in unserem Beispiel die folgende Form:

```
WHERE PLZ = ?
```

Anschließend beginnen wir die Rekursion, indem wir rückwärts entlang des eingehenden RCC $F.Ort \rightarrow O.PLZ$ zur Tabelle *Film* laufen. Die zuvor generierte Where-Klausel wird dabei, wie bei den Insert-Anweisungen, mit übergeben. Da die Tabelle *Film* keine Kontrolltabelle ist, erweitern wir die Where-Klausel wie folgt:

```
WHERE ORT IN (SELECT PLZ FROM ORT WHERE PLZ = ?)
```

Die erweiterte Klausel übergeben wir wieder dem nächsten Rekursionsschritt. Durch Zurückverfolgen des eingehenden RCC $R.id \rightarrow F.Regie$ erreichen wir die Tabelle *Regisseur*. Dort erweitern wir unsere Klausel auf die folgende Form und verfolgen den einzigen eingehenden RCC dieser Tabelle.

```
WHERE ID IN (SELECT REGIE FROM FILM WHERE ORT IN
                  (SELECT PLZ FROM ORT WHERE PLZ = ?))
```

Der eingehende RCC der Tabelle *Regisseur* führt uns zur Kontrolltabelle K_1. Damit endet dieser RCC-Pfad und wir erstellen die folgende Update-Anweisung:

```
UPDATE K1
SET lastAccess = ?, hitcounter = hitcounter + 1
WHERE (ID IN
      (SELECT ID FROM REGISSEUR WHERE ID IN
            (SELECT REGIE FROM FILM WHERE ORT IN
                  (SELECT PLZ FROM ORT WHERE PLZ = ?))))
```

Nach der Generierung der Update-Anweisung fallen wir zurück zur Tabelle *Regisseur* und verfolgen die übrigen eingehenden RCCs. Da die Tabelle *Regisseur* keine weiteren eingehenden RCCs besitzt, fallen wir zurück zur Tabelle *Film* und von dort zurück zur Tabelle *Ort* . Die Tabelle *Ort* besitzt einen weiteren eingehenden RCC, den wir analog zum vorherigen RCC zurückverfolgen. Dabei erreichen wir im Beispiel am Ende nochmals die Kontrolltabelle K_1. Da wir bereits eine Update-Anweisung für diese Kontrolltabelle erstellt haben, verknüpfen wir die, auf dem zweiten RCC-Pfad erstellte, Where-Klausel disjunktiv mit der Ersten. Damit endet die Generierung der Update-Anweisungen für das Beispiel in Abbildung 4.13. Die komplette Update-Anweisung für die Spalte *Ort.PLZ* ist in Abbildung 4.14 dargestellt.

```
UPDATE K1
SET lastAccess = ?, hitcounter = hitcounter + 1
WHERE (ID IN
      (SELECT ID FROM REGISSEUR WHERE ID IN
            (SELECT REGIE FROM FILM WHERE ORT IN
                  (SELECT PLZ FROM ORT WHERE PLZ = ?))))
OR (ID IN
      (SELECT ID FROM REGISSEUR WHERE WOHNORT IN
            (SELECT PLZ FROM ORT WHERE PLZ = ?)))
```

Abbildung 4.14 – Aufbau der generierten Update-Anweisungen

Bei genauerer Betrachtung von Abbildung 4.14 fällt auf, dass bei jeder Update-Anweisung die Spalte *hitcounter* in der Kontrolltabelle um eins inkrementiert wird. Enthält nun die Warteschlange des Hit Counter sehr viele Nachrichten mit gleichem Inhalt, so müssen sehr viele einzelne Update-Anweisungen ausgeführt werden. In diesem Fall wäre die folgende Vorgehensweise sinnvoll: Der Hit Counter entnimmt eine Nachricht aus der Warteschlange. Anschließend überprüft er, ob die Warteschlange weitere Nachrichten mit demselben Inhalt enthält. Er merkt sich die Anzahl der gleichen Nachrichten und führt eine Update-Anweisung auf der Kontrolltabelle aus, wobei die Spalte *hitcounter* um die Anzahl der gleichen Nachrichten erhöht wird.

Nach der Generierung aller Update-Anweisungen für eine Spalte werden diese analog zu den Insert-Anweisungen als Liste in einem Container gespeichert. Der Hit

Counter entnimmt diesem Container während der Laufzeit die benötigte Liste mit Update-Anweisungen und führt diese anschließend aus. Dabei ist im Gegensatz zum Fill Daemon die Ausführungsreihenfolge nicht relevant.

Wie wir gesehen haben, verläuft auch die Generierung der Update-Anweisungen nach einem Schema, das wir an dieser Stelle zusammen mit einer Aufwandsbetrachtung kurz erläutern wollen. Die Where-Klausel in Rekursionsschritt n wird nach folgendem Schema erstellt:

```
WHERE <RCC Quelltabelle> IN
      (SELECT <RCC Zieltabelle>
       FROM <RCC Zieltabelle>
       <übergebene Where-Klausel aus Schritt n-1>)
```

Bei genauer Betrachtung dieses Schemas fällt auf, dass, im Gegensatz zu den Insert-Anweisungen, in der Where-Klausel keine Tabellen der Backend-Datenbank referenziert werden, d. h., die Update-Anweisungen werden vollständig auf dem Cache ausgewertet. Weiterhin fällt auf, dass die Join-Operationen nur zwischen Quell- und Zielspalten von RCCs stattfinden. Dadurch können zur Anfrageauswertung die definierten Indizes auf dem Cache verwendet werden. Im Anfrageplan muss im Normalfall nur für die innerste Select-Anweisung ein Tabellenscan durchgeführt werden, alle Join-Partner der übergeordneten Select-Anweisungen können durch Suche auf den Indizes bestimmt werden. Dadurch kann die Anfrage sehr effizient vom Datenbanksystem verarbeitet werden.

4.7 Der Garbage Collector

Die letzte Komponente, die als eigenständiger Thread während der Initialisierungsphase eingerichtet wird, ist der Garbage Collector. Der Garbage Collector sorgt dafür, dass der Cache nicht zu voll wird, indem er periodisch überprüft, ob die Cache-Datenbank eine spezifizierte Füllmarke überschritten hat. Wird diese Füllmarke überschritten, so führt der Garbage Collector einen oder mehrere Löschvorgänge durch.

Ein Löschvorgang umfasst das Entfernen eines Werts aus einer Kontrolltabelle sowie das rekursive Absteigen entlang der ausgehenden RCCs. Dabei hat der Garbage Collector darauf zu achten, dass nicht irrtümlich Datensätze gelöscht werden, die noch von RCCs referenziert werden. Es gilt wie beim Fill Daemon die Regel, dass nach Durchführung eines Löschvorgangs der Cache sich wieder in einem gültigen Zustand befinden muss.

Die Delete-Anweisungen, die der Garbage Collector für seine Arbeit verwendet, können analog zu den Anweisungen des Fill Daemon und des Hit Counter während der Initialisierungsphase generiert werden.

Im folgenden Abschnitt wollen wir uns etwas ausführlicher mit der Verdrängungsstrategie des Garbage Collector beschäftigen. Im Anschluss daran werden wir in Abschnitt 4.7.2 die Generierung der Delete-Anweisungen erläutern.

4.7.1 Verdrängungsstrategien

Damit die Datensätze nicht willkürlich aus dem Cache gelöscht werden, benötigen wir eine Verdrängungsstrategie. Als Vorlage dienten uns hierfür die bekannten Seitenverdrängungsstrategien für den Hauptspeicher, wie FIFO (First in first out), LFU (least frequently used) und LRU (least recently used). Es existieren zwar noch weitere Verdrängungsstrategien, dennoch haben wir uns auf diese kleine Auswahl beschränkt, da diese Verfahren relativ leicht zu implementieren sind. Die Umstellung auf komplexere Verfahren kann zu einem späteren Zeitpunkt ohne größeren Aufwand durchgeführt werden.

Damit die oben genannten Verfahren verwendet werden können, benötigen wir Informationen darüber, wie lange ein Datensatz sich im Cache befindet, wie oft er referenziert wurde und wann er das letzte Mal referenziert wurde. Da der Aufwand zur Speicherung dieser Informationen für jeden einzelnen Datensatz zu aufwendig wäre, haben wir uns dafür entschieden, diese Informationen nur für die Einträge in den Kontrolltabellen zu sammeln. Jede Kontrolltabelle besitzt zusätzlich zu der Unique-Spalte, die die eingelagerten Werte enthält, noch drei weitere Spalten, in denen der Einlagerungszeitpunkt, der Zeitpunkt des letzten Zugriffs sowie die Anzahl der Zugriffe gespeichert sind.

Die Existenz der Kontrolltabellen ist dem Benutzer im Allgemeinen nicht bekannt, sodass keine expliziten Anfragen von Benutzerseite auf diesen Tabellen zu erwarten sind, d. h., Schreibsperren auf den Kontrolltabellen beeinflussen nur bedingt den Benutzerbetrieb (z. B. bei der Sondierung auf einer Füllspalte).

Die Hit-Counter-Komponente spielt beim Sammeln der benötigten Verdrängungsinformationen eine wesentliche Rolle. Sie aktualisiert die Kontrolltabellen, nachdem die Sondierung einen Einstiegspunkt gefunden hat. Dadurch besitzen wir die benötigten Informationen über die Anzahl der Zugriffe und den Zeitpunkt der letzten Referenzierung (vgl. Abschnitt 4.6). Den Zeitpunkt der Einlagerung liefert der Fill Daemon, der bei jeder Einlagerung den Eintrag in der Kontrolltabelle mit einem Zeitstempel versieht.

Mit Hilfe der Informationen in den Kontrolltabellen können wir alle drei der genannten Verdrängungsverfahren im Garbage Collector realisieren. Bei genauer Betrachtung der Verfahren fällt jedoch auf, dass einige Verfahren sich besser eignen als andere.

Das FIFO-Verfahren wählt den ältesten Eintrag in der Kontrolltabelle als „Opfer“

aus. Diese Vorgehensweise ist jedoch schlecht, wenn der Wert dieses Eintrags sehr häufig referenziert wurde. Ein Löschen dieses Werts aus der Kontrolltabelle führt dazu, dass der Fill Daemon den Wert kurze Zeit später wieder einlagern muss.

Das LFU-Verfahren ist für den Cache ebenfalls nicht geeignet, da zuerst die Einträge in den Kontrolltabellen als Opfer ausgewählt werden, die die wenigsten Zugriffe erhalten haben. Einträge mit vielen Zugriffen bleiben eventuell sehr lange nach ihrem letzten Zugriff im Cache, bevor sie wieder verdrängt werden und wertvollen Speicherplatz freigeben. Dieses Verfahren ist daher nicht flexibel genug um sich den Benutzeranfragen anzupassen.

Das LRU-Verfahren wählt den Eintrag in einer Kontrolltabelle als Opfer aus, der am längsten nicht mehr referenziert wurde. Zwar kann es auch bei diesem Verfahren vorkommen, dass wir einen Eintrag in der Kontrolltabelle löschen, der kurze Zeit später wieder referenziert wird, aber für die Anpassung an die Benutzeranfragen stellt das LRU-Verfahren die beste Lösung unter den drei Verfahren dar.

Nachdem der Garbage Collector ein „Opfer“ bestimmt hat, indem er in allen Kontrolltabellen nach dem am längsten nicht referenzierten Eintrag sucht, werden die entsprechenden Datensätze aus dem Cache gelöscht. Dabei verwendet der Garbage Collector die Delete-Anweisungen, die während der Initialisierungsphase generiert wurden. Im folgenden Abschnitt wollen wir kurz auf die Generierung der Anweisungen eingehen.

4.7.2 Generierung der Delete-Anweisungen

Die Generierung der Delete-Anweisungen ist sehr einfach, da wir für jede Tabelle lediglich diejenigen Datensätze löschen müssen, die durch keinen eingehenden RCC referenziert werden. Dies führt zu Problemen bei Zyklen, die wir weiter unten diskutieren werden. Die Generierung der Anweisungen beginnt auf der Kontrolltabelle und verläuft anschließend entlang der ausgehenden RCCs.

Auch die Generierung der Delete-Anweisung wollen wir an einem kleinen Beispiel durchführen. Als Beispiel dient die Cache Group in Abbildung 4.15. Wir beginnen mit der Generierung auf der Kontrolltabelle K_1. Die Delete-Anweisung für die Kontrolltabelle hat die folgende Form:

```
DELETE FROM K1
WHERE id = ?
```

Danach folgen wir dem ausgehenden RCC $K_1.id \rightarrow R.id$ zur Tabelle *Regisseur* und erzeugen eine Delete-Anweisung für diese Tabelle:

```
DELETE FROM REGISSEUR
WHERE (ID NOT IN (SELECT ID FROM K1))
```

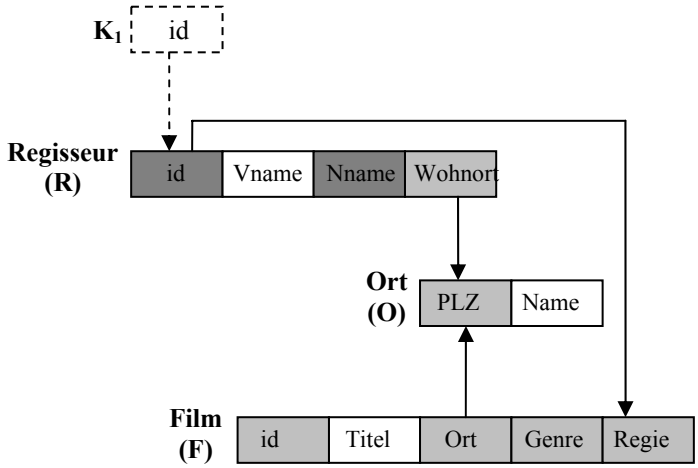


Abbildung 4.15 – Beispiel für die Generierung der Delete-Anweisungen

Die Where-Klausel in der Anweisung sorgt dafür, dass nicht versehentlich Datensätze gelöscht werden, die noch durch eingehende RCCs referenziert werden. Die Generierung der Delete-Anweisungen für eine Cache-Tabelle unterliegt dabei dem folgenden Schema, wobei für jeden eingehenden RCC dieser Tabelle ein entsprechender Ausdruck für die Where-Klausel erzeugt und mit den übrigen konjunktiv verknüpft werden muss. Die Größe der Anfrage hängt also von der Anzahl der eingehenden RCCs ab und wächst mit dieser linear an.

```
DELETE FROM <Cache-Tabelle>
WHERE (<RCC Zielspalte> NOT IN
      (SELECT <RCC Quellspalte> FROM <RCC Quelltabelle>))
```

Von der Tabelle *Regisseur* verfolgen wir den RCC $R.Wohnort \rightarrow O.PLZ$ und erreichen die Tabelle *Ort*. Da die Tabelle *Ort* zwei eingehende RCCs enthält müssen wir auch in der Delete-Anweisung zwei entsprechende Ausdrücke erzeugen. Als Ergebnis erhalten wir die folgende Anweisung:

```
DELETE FROM ORT
WHERE (PLZ NOT IN (SELECT WOHNORT FROM REGISSEUR))
AND (PLZ NOT IN (SELECT ORT FROM FILM))
```

Da die Tabelle *Ort* keine ausgehenden RCCs besitzt, endet der RCC-Pfad und wir kehren zur vorherigen Tabelle zurück, um dort die restlichen ausgehenden RCCs zu verfolgen. Wird während der Generierung ein homogener Zyklus entdeckt, so wird der RCC, der zum Zyklus geführt hat, ausgeschlossen. Alle anderen ausgehenden RCCs der Tabelle werden jedoch weiter verfolgt. Ein heterogener Zyklus führt zu einer Fehlermeldung, die an die Starter-Komponente weitergeleitet wird.

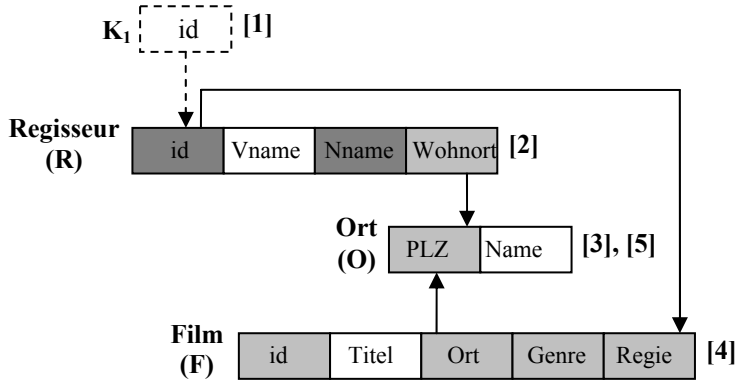
Die Generierung der Delete-Anweisungen des anderen RCC-Pfads geschieht analog und soll hier nicht weiter dargestellt werden. Der Vollständigkeit halber sind alle Delete-Anweisungen des Beispiels noch einmal in Abbildung 4.16 zusammengefasst.

```
[1] DELETE FROM K1
    WHERE lastaccess = ?

[2] DELETE FROM REGISSEUR
    WHERE (ID NOT IN (SELECT ID FROM K1))

[3] DELETE FROM ORT
    WHERE (PLZ NOT IN (SELECT WOHNORT FROM REGISSEUR))
    AND (PLZ NOT IN (SELECT ORT FROM FILM))

[4] DELETE FROM FILM
    WHERE (REGIE NOT IN (SELECT ID FROM REGIESSEUR))

[5] DELETE FROM ORT
    WHERE (PLZ NOT IN (SELECT WOHNORT FROM REGISSEUR))
    AND (PLZ NOT IN (SELECT ORT FROM FILM))
```

Abbildung 4.16 – Aufbau der generierten Delete-Anweisungen

Wie aus Abbildung 4.16 und Abbildung 4.17 schön zu erkennen ist, werden insgesamt zwei Delete-Anweisungen für die Tabelle *Ort* erzeugt. Dies ist notwendig, da es nach dem Löschen von Datensätzen in der Tabelle *Film* wieder nicht referenzierte Datensätze in der Tabelle *Ort* geben kann.

Nach der Generierung werden alle zu einer Kontrolltabelle gehörenden Delete-Anweisungen als Liste in einem Container gespeichert. Überschreitet der Cache die spezifizierte Füllmarke, so entnimmt der Garbage Collector aus dem Container die zum gewählten „Opfer“ gehörende Liste mit Delete-Anweisungen und führt die Anweisungen gemäß der Generierungsreihenfolge in Abbildung 4.17 aus.



[x]: Reihenfolge der Generierung

Abbildung 4.17 – Generierungsreihenfolge der Delete-Anweisungen

Der Vorteil bei dieser Top-Down-Vorgehensweise ist, dass der Cache nach jeder Delete-Anweisung in einem gültigen Zustand ist und wir somit jeden Löschvorgang auf einer Cache-Tabelle in einer eigenen Transaktion durchführen können. Dadurch entstehen keine langen Schreibsperren und die Query Worker werden in ihrer Arbeit nur wenig behindert. Ein weiterer Vorteil ergibt sich durch den Aufbau der Delete-

Anweisungen: Alle nicht referenzierten Datensätze einer Tabelle werden beim Löschvorgang entfernt, d. h. sollte beispielsweise der Fill Daemon eine Einlagerung durch einen Fehler nicht komplett abschließen (der Cache ist trotzdem in einem gültigen Zustand; vgl. Abschnitt 4.5.1), so werden diese Datensätze beim nächsten Löschvorgang wieder entfernt und belegen nicht den wertvollen Speicherplatz des Cache.

Löschen in Cache Groups mit Zyklen

Die vorgestellte Vorgehensweise beim Löschen von Datensätzen besitzt jedoch auch einige Nachteile, die wir an dieser Stelle aufzeigen wollen. Die Existenz eines Zyklus in der Cache-Group führt beim Löschvorgang dazu, dass die Datensätze in den am Zyklus beteiligten Tabellen nicht gelöscht werden. Ein kleines Beispiel soll dem besseren Verständnis dienen.

Abbildung 4.18 zeigt eine Cache Group mit einer konkreten Wertebelegung. Der Einfachheit halber besitzt jede Tabelle nur eine Spalte. Angenommen, Tabelle *A* sei eine Kontrolltabelle, auf der wir den Wert 5 als Opfer bestimmt haben. Wir löschen den Wert auf der Tabelle *A*. Anschließend gelangen wir über den ausgehenden RCC zur Tabelle *B*. Diese Tabelle ist jedoch Teil eines Zyklus. Der eingehende RCC von Tabelle *D* verhindert, dass wir auf der Tabelle *B* den Wert 5 löschen können, da durch die Delete-Anweisungen nur diejenigen Datensätze aus einer Tabelle gelöscht werden, die durch keinen eingehenden RCC referenziert werden. Auch auf der Tabelle *E* können wir den Wert 5 nicht löschen, da dies durch den RCC von Tabelle *B* verhindert wird.

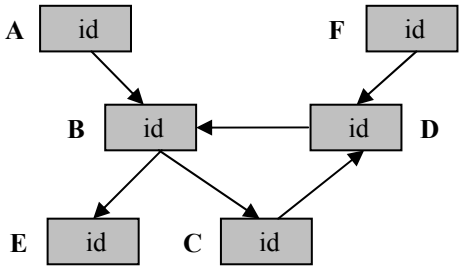


Tabelle	Vorhandene Werte		
A	5	7	…
B	5	7	…
C	5	7	…
D	5		…
E	5	7	…
F	5		…

Abbildung 4.18 – Zyklenbehandlung während des Löschvorgangs

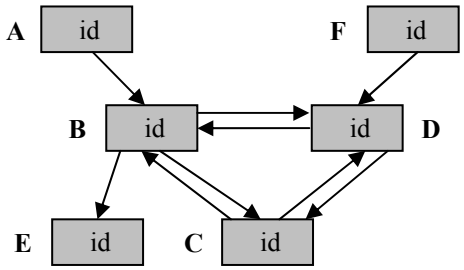


Abbildung 4.19 – Cache-Group mit mehreren Zyklen

Eine Lösung für dieses Problem ist jedoch nicht trivial. Es ist klar, dass der Zyklus zwischen den Tabellen aufgebrochen werden muss, damit die Datensätze gelöscht

werden können. Das Aufbrechen des Zyklus führt aber unweigerlich dazu, dass der Cache einen ungültigen Zustand erhält, da mindestens ein RCC verletzt wird. Des Weiteren können die am Zyklus beteiligten Tabellen noch an weiteren Zyklen beteiligt sein (vgl.Abbildung 4.19).

Zur Lösung dieses Problems sollten auch die in Abschnitt 4.5.1 vorgestellten atomaren Zonen in Betracht gezogen werden. Durch die zyklische Referenzierung der Datensätze in den Tabellen *B*, *C* und *D* ist es notwendig den Löschvorgang auf diesen drei Tabellen gleichzeitig, d. h., innerhalb einer Transaktion durchzuführen, andernfalls entstehen ungültige Cache-Zustände. Dieses Problem trat bereits bei der Generierung der Insert-Anweisungen auf (vgl. Abschnitt 4.5.1) und konnte mit Hilfe der atomaren Zonen erfolgreich gelöst werden. Löschen wir jedoch aus dem obigen Beispiel den Wert 5 in den am Zyklus beteiligten Tabellen, so erhalten wir einen ungültigen Cache-Zustand, da der Wert 5 auch in der Tabelle *F* existiert. Wir müssen also beim Löschen berücksichtigen, dass die Tabellen innerhalb einer atomaren Zone von Tabellen aus verschiedenen Zonen durch RCCs referenziert werden können. An dieser Stelle wollen wir diese Problematik jedoch nicht weiter vertiefen.

Probleme beim gleichzeitigen Einlagern und Löschen

Ein weiteres Problem entsteht durch die gleichzeitige Arbeit des Fill Daemon und des Garbage Collector auf dem Cache. Da beide in ihrer Arbeit darauf ausgerichtet sind, nur kurze Schreibsperren zu verursachen, kann durch eine ungünstige Konstellation ein ungültiger Cache-Zustand entstehen. Betrachten wir dazu die Abbildung 4.20. Angenommen der Fill Daemon führt gerade einen Einlagerungsvorgang für die Füllspalte *R.id* durch und hat bereits die benötigten Datensätze in die Tabelle *Ort* eingefügt. Da er in einem eigenständigen Thread läuft, ist seine Zeitscheibe irgendwann abgelaufen und er muss die CPU freigeben.

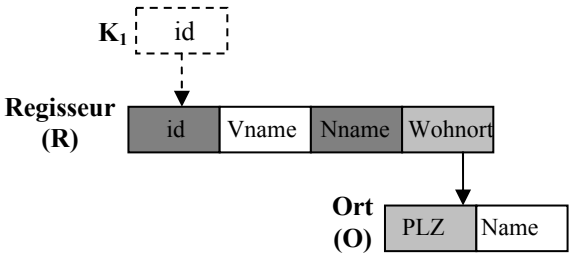


Abbildung 4.20 – Problem der konkurrierenden Threads

Gehen wir nun davon aus, dass als Nächstes der Garbage Collector die CPU verwenden darf und alle Voraussetzungen für einen Löschvorgang gegeben sind. Der Garbage Collector beginnt seine Arbeit auf der Kontrolltabelle K_1 und läuft entlang des RCC-Pfads bis zur Tabelle *Ort*. In der Tabelle *Ort* löscht er nun alle vom Fill Daemon neu eingelagerten Datensätze, da die entsprechenden Datensätze in der Tabelle *Regis-*

seur noch nicht eingelagert wurden und somit keine Referenz auf die neuen Datensätze besteht. Der Garbage Collector beendet daraufhin seine Arbeit und hinterlässt einen gültigen Cache-Zustand. Anschließend erhält der Fill Daemon wieder die CPU und fährt mit dem Einlagerungsvorgang fort. Sobald er jedoch die Einlagerung auf der Tabelle *Regisseur* abgeschlossen hat, befindet sich der Cache in einem ungültigen Zustand.

Dieses Szenario kann jedoch vermieden werden, wenn beide Komponenten vor dem Beginn ihrer Arbeit entsprechende Sperren anfordern. Da wir allerdings auch weiterhin nur kurze Schreibsperren auf der Datenbank verwenden wollen, war es notwendig ein Sperrverfahren auf höherer Ebene zu realisieren. Wir haben dazu im CBCS zwei Typen von Sperren entwickelt: eine G-Sperre für den Garbage Collector und eine F-Sperre für den Fill Daemon. Für die beiden Sperren gilt folgende Kompatibilitätsmatrix:

		aktuelle Sperre		
		keine	G	F
angeforderte Sperre	G	+	+	-
	F	+	-	+

+: Sperre wird gewährt
- : Sperre wird nicht gewährt

Wie man aus der Matrix leicht erkennt, können gleichzeitig entweder nur Fill-Daemon-Instanzen oder Garbage-Collector-Instanzen auf dem Cache arbeiten, da die Sperren sich gegenseitig ausschließen.

Bevor eine der beiden Komponenten ihre Arbeit beginnt, muss zuerst die entsprechende Sperre angefordert werden, erst dann darf diese Komponente ihre Arbeit durchführen. Durch die Einführung dieses Sperrverfahrens wird das oben beschriebene Szenario vermieden. Bei der Einführung von weiteren Komponenten, die Änderungen auf dem Cache durchführen, muss überprüft werden, ob Konfliktsituationen entstehen können und gegebenenfalls müssen weitere Sperren definiert werden.

Bei der Erzeugung des Anfrageplans für die Delete-Anweisungen kann der Datenbank-Optimierer die Indizes auf dem Cache verwenden. Betrachten wir dazu nochmals das Schema zur Generierung der Delete-Anweisungen.

```
DELETE FROM <Cache-Tabelle>
   WHERE (<RCC Zielspalte> NOT IN
   (SELECT <RCC Quellspalte> FROM <RCC Quelltabelle>))
```

Der Optimierer kann auf der Tabelle, auf der gelöscht werden soll, einen Tabellenscan durchführen und für jeden Datensatz über den Index auf der Quellspalte des RCC überprüfen, ob der Wert dieses Datensatzes in der Quelltabelle des RCC vorhanden ist.

4.8 Die JDBC-Schnittstelle

Eine der Anforderungen an einen Cache ist, dass der Benutzer von der Existenz des Cache nach Möglichkeit nichts bemerkt. Damit das CBCS für den Benutzer transparent arbeiten kann, war es notwendig, eine JDBC-Schnittstelle zu entwickeln, die alle Anfragen des Benutzers an das CBCS weiterleitet. Die JDBC-Schnittstelle des CBCS umfasst momentan rund 80% des üblichen Funktionsumfangs eines JDBC-Treiber. Wir werden auf die Einschränkungen der JDBC-Schnittstelle detailliert in Abschnitt 5.2.7 eingehen. Der Funktionsumfang wird jedoch in zukünftigen Versionen des CBCS noch erweitert.

Die Einbindung der JDBC-Schnittstelle in neue oder bereits bestehende (Web-) Anwendungen ist sehr einfach. Dazu muss in der Anwendung lediglich der bestehende JDBC-Treiber ersetzt werden und schon werden alle Anfragen an das CBCS weitergeleitet und dort bearbeitet.

Sobald der Benutzer eine Verbindung zur Datenbank herstellen möchte, fordert die JDBC-Schnittstelle beim CBCS einen freien Query Worker an, der die Anfragen des Benutzers bearbeitet. Führt der Benutzer eine Anfrage aus, so leitet die JDBC-Schnittstelle die Anfrage an den Query Worker weiter. Dort wird sie, wie in den Abschnitten 4.4.1 und 4.4.2 beschrieben, verarbeitet und ausgeführt. Das Ergebnis wird anschließend über die JDBC-Schnittstelle dem Benutzer zugänglich gemacht.

Durch die einfache Einbindung der JDBC-Schnittstelle in bestehende Anwendungen ist die Verwendung des CBCS aus Sicht der Anwendungsentwickler weder zeitnoch kostenintensiv.

Nachdem wir nun in diesem Kapitel die einzelnen Komponenten des CBCS und ihre Funktionen kennen gelernt haben, werden wir im nächsten Kapitel einen Einblick in die Implementierung der JDBC-Schnittstelle und der übrigen Komponenten geben.

5 Implementierung des Prototyps

In diesem Kapitel wollen wir auf die Implementierung der einzelnen Komponenten des CBCS eingehen. Die Funktionsweise der verschiedenen CBCS-Komponenten werden wir mit Hilfe von Code-Fragmenten genauer erläutern und veranschaulichen. Eine kurze Anleitung zum Installieren und Einrichten des CBCS findet sich in Anhang D. Dort findet sich auch eine Beschreibung zum Aufbau der Konfigurationsdateien und Erläuterungen zu den einzelnen Startparametern und den Befehlen der Kommandozeile.

5.1 Die Umgebung

Bevor wir jedoch auf die eigentliche Implementierung eingehen, möchten wir zuerst die verwendete Entwicklungsumgebung vorstellen. Das CBCS wurde komplett in Java entwickelt, dazu verwendeten wir das JDK in der Version 1.5 (bzw. 5.0) von Sun [J2SE05]. Der Programmcode des CBCS verwendet Erweiterungen, die erst in der Java-Version 1.5 hinzugekommen sind, daher ist eine Kompilierung des Programmcodes unter früheren Java-Versionen nicht möglich.

Als Entwicklungsumgebung verwenden wir die frei verfügbare Eclipse-Plattform in der Version 3.1 [Eclipse]. Diese Plattform bietet ab Version 3.1 eine vollständige Unterstützung aller Erweiterungen von Java 1.5. Die Gründe für den Einsatz dieser Entwicklungsumgebung waren unter anderem die relativ intuitive Benutzeroberfläche und die große Anzahl an frei verfügbaren Erweiterungskomponenten (Plugins).

Für die Cache-Datenbank sowie für die Backend-Datenbank verwenden wir eine IBM DB2 Version 8.1 [IBM05a]. Die IBM DB2 Funktionalität zum Zugriff auf föderierte Datenbanken war einer der Hauptgründe für den Einsatz dieser Datenbank. Durch diese Funktionalität ist es möglich, mehrere Datenbanksysteme miteinander zu verbinden und über eine Schnittstelle anzusprechen. Die DB2 fungiert dabei als eine Middleware, in der die Anfragen auf die verbundenen Datenbanksysteme verteilt und die einzelnen Anfrageergebnisse zusammengeführt werden. In unserem Fall konnten wir dadurch die Cache-Datenbank mit der Backend-Datenbank verbinden; dadurch wurde die Middle-Tier-Komponente weniger komplex.

5.2 Implementierung der CBCS-Komponenten

In den folgenden Abschnitten wollen wir einige Programmcode-Fragmente des CBCS vorstellen und erläutern. Leider können wir nicht auf den gesamten Programmcode eingehen, da dies den Rahmen dieser Arbeit sprengen würde. Wir werden dennoch die wichtigsten Passagen des Programmcodes herausfiltern und an dieser Stelle etwas genauer betrachten.

Bevor wir jedoch die einzelnen Code-Fragmente vorstellen, wollen wir einen Überblick über die Paketstruktur des CBCS geben (Abbildung 5.1).

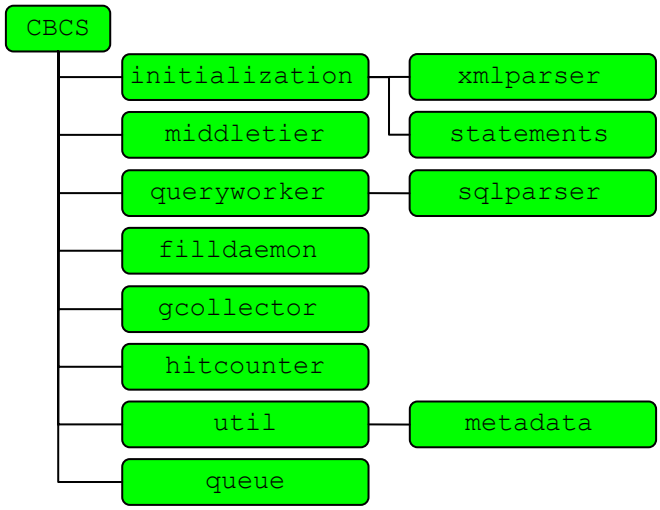


Abbildung 5.1 – Die wichtigsten Java-Pakete des CBCS

Wie aus Abbildung 5.1 zu sehen ist, besteht das CBCS auf oberster Ebene aus insgesamt acht Java-Paketen. Jedes Paket repräsentiert eine Komponente des CBCS und beinhaltet gegebenenfalls weitere Pakete. Ziel bei der Entwicklung des Prototyps war es, eine sehr geringe Kopplung der Komponenten untereinander zu erreichen. Dadurch ist es möglich einzelne Komponenten zu ersetzen, ohne dass größere Änderungen am Gesamtsystem vorgenommen werden müssen. Das CBCS ermöglicht somit einen einfachen Austausch der Komponenten, um beispielsweise neue Strategien zum Füllen oder Löschen des Cache zu realisieren.

Die Kommunikation der Komponenten untereinander geschieht über Nachrichten, die in entsprechenden Warteschlangen gespeichert werden. Da wir im CBCS keine synchronen Benachrichtigungen zwischen den Komponenten benötigen, konnten wir die Kommunikation über Warteschlangen realisieren, die in einem separaten Paket untergebracht sind. Wir werden auf die Warteschlangen in Abschnitt 5.2.2 nochmals genauer eingehen.

Das Java-Paket *util* beinhaltet die Klassen, die die interne Datenstruktur repräsentieren. Auf die Datenstruktur werden wir nicht weiter eingehen, da diese ausführlich in Abschnitt 4.3 (Seite 41) erläutert wurde.

In den nachfolgenden Abschnitten werden wir nacheinander auf alle in Abbildung 5.1 dargestellten Java-Pakete kurz eingehen und aus jedem Paket die wichtigsten Fragmente des Programmcodes erläutern.

5.2.1 Die Initialisierung

Wir haben den Initialisierungsvorgang des CBCS bereits ausführlich in Abschnitt 4.3 erläutert. In den folgenden Abschnitten wollen wir daher einen genaueren Blick auf den Aufbau der Konfigurationsdatei, die die Informationen über die Cache Group enthält, sowie die Erzeugung der SQL-Anweisungen für die Prepared Statements werfen.

Aufbau der Konfigurationsdatei

Das Auslesen der Konfigurationsdatei haben wir bereits in Abschnitt 4.3.1 ausführlich beschrieben. Wir wollen an dieser Stelle den Aufbau dieser Datei näher betrachten. Die Konfigurationsdatei liegt dem CBCS als XML-Datei vor, dadurch sind alle benötigten Informationen in einer übersichtlichen Struktur eingebettet, die mit Hilfe eines Parsers sehr leicht ausgelesen werden kann. Wir verwenden im CBCS einen DOM-Parser, der sich im Java-Paket *xmlparser* befindet. Der DOM-Parser ermöglicht eine elegante Navigation über die einzelnen Elemente der XML-Datei.

Zum besseren Verständnis wollen wir die Struktur der XML-Datei an dem in Abbildung 5.2 dargestellten Beispiel erläutern. Das Beispiel zeigt einen Ausschnitt der XML-Beschreibung unserer Filmdatenbank (Abbildung 2.1, Seite 6). Eine komplette XML-Beschreibung der Filmdatenbank sowie die zugrunde liegende Document-Type-Definition (DTD) finden sich in Anhang B.

Die XML-Datei ist in zwei Bereiche unterteilt. Der erste Bereich ist durch die Tags <*backend*> abgegrenzt und beinhaltet Informationen über die Schemata und Tabellen der Backend-Datenbank. Jeder Tabelleneintrag beinhaltet den Namen der Tabelle im Backend, sowie eine Menge von Spalten, die in derselben Reihenfolge wie im Backend spezifiziert werden. Zusätzlich werden die Spalten, die den Primärschlüssel der Tabelle bilden, als Referenz gespeichert (z. B. „c1“ für die Tabelle Regisseur).

Die Spalteneinträge enthalten den Namen der Spalte und ihren Datenbanktyp. Des Weiteren spezifizieren die beiden Variablen *unique* und *nullable*, ob die Spalte eine Unique-Spalte ist und ob in dieser Spalte Null-Werte erlaubt sind.

Der zweite Bereich in der XML-Datei wird durch die Tags <*cachegroup*> abgegrenzt und beinhaltet eine Beschreibung aller Cache-Tabellen, Füllspalten und RCCs. Die Beschreibung einer Cache-Tabelle umfasst den Namen der Cache-Tabelle und die Referenz auf die entsprechende Tabelle im Backend-Bereich. Diese Referenzierung mag auf den ersten Blick etwas verwirrend wirken, da beispielsweise eine Füllspalte in der Cache-Datenbank definiert ist und nicht in der Backend-Datenbank. Der Backend-

Bereich enthält bereits alle Informationen, die wir benötigten. Die Definition eines speziellen Cache-Bereichs würde die Speicherung von redundanten Informationen und einen höheren Aufwand bei Änderungen in der Konfigurationsdatei bedeuten. Aus diesem Grund haben wir die vorliegende Form der Referenzierung gewählt.

```
<dbcache xmlns="http://cbcs-project.de/dbcache">
  <backend>
    <schema name="SCHEMA1">
       <table id="t1" name="Regisseur" primarykey="c1">
          <column id="c1" unique="true" nullable="false">
             <name>id</name>
             <type>integer</type>
          </column>
          ...
          <column id="c5">
              <name>wohnort</name>
              <type>integer</type>
          </column>
          ...
       </table>
       <table id="t3" name="Ort" primarykey="c11 c12">
          <column id="c11" unique="false" nullable="false">
              <name>plz</name>
              <type>integer</type>
          </column>
          ...
       </table>
        ...
    </schema>
  </backend>
  <cachegroup>
    <schema name="SCHEMA1">
      <table id="ct1" ref="t1" name="CA_Regisseur"/>
      <table id="ct3" ref="t3" name="CA_Ort"/>
       ...
    </schema>
    <fillcolumn ref="c1" percentage="1"
            ctrl-table-name="Ctrl_Regisseur"/>
     ...
    <rcc from="c5" to="c11" user-defined="true"/>
     ...
  </cachegroup>
</dbcache>
```

Abbildung 5.2 – Struktur der Cache Group in XML-Notation

Der Name der Cache-Tabelle kann beliebig gewählt werden, da er nur zur Generierung der Cache-Tabellen und zur internen Zuordnung von Cache-Tabellen zu Backend-Tabellen verwendet wird. Der Cache-Tabellenname könnte z. B. auch mit dem Namen der zugehörigen Backend-Tabelle übereinstimmen. In der von uns verwendeten Umgebung war es dennoch notwendig, für die Cache-Tabellen eindeutige Namen zu wählen, da wir beschlossen haben den Nicknames denselben Namen wie den von ihnen referenzierten Backend-Tabellen zu geben. Die Verwendung gleicher Namen würde

daher zu Konflikten zwischen den Nicknames (vgl. Abschnitt 5.2.1) und den Cache-Tabellen führen und eine eindeutige Zuordnung verhindern. Wir haben daher beschlossen, alle Cache-Tabelle durch ein zusätzliches Präfix, wie es in Abbildung 5.2 dargestellt ist, zu erweitern, um eine eindeutige Zuordnung zu gewährleisten.

Alle Füllspalteneinträge benötigen eine Referenz auf die entsprechende Spalte im Backend-Bereich und zusätzlich einen Namen für die später erzeugte Kontrolltabelle. Die Variable *percentage* spezifiziert die in Abschnitt 4.3.2 (Seite 43) vorgestellte Obergrenze für die Selektivität der Daten in der Backend-Tabelle.

Die RCC-Einträge besitzen jeweils eine Referenz auf die verwendete Quell- und Zielspalte, sowie einen booleschen Wert, der angibt, ob der RCC vom Administrator spezifiziert wurde oder durch automatisierte Werkzeuge hinzugefügt wurde.

Nach dem Parsen der XML-Datei wird aus den Informationen die Datenstruktur aus Abschnitt 4.3.1 (Seite 41) erzeugt und den restlichen Komponenten zur Verfügung gestellt.

Erzeugung der SQL-Anweisungen

Während der erweiterten Initialisierung wird unter anderem die Cache-Datenbank eingerichtet, d. h. alle spezifizierten Cache-Tabellen, zusätzliche Indizes usw. müssen in der Cache-Datenbank angelegt werden. Die dazu notwendigen SQL-Anweisungen werden in einer speziellen Klasse, der *Statement Factory*, erzeugt. Diese Klasse befindet sich im Unterpaket *statements* des Pakets *initialization*.

Die Kapselung der Generierung von SQL-Anweisungen ist notwendig, da es trotz Standardisierung von SQL immer noch Datenbanksysteme existieren, die spezielle SQL-Dialekte verwenden. Die vorgenommene Kapselung ermöglicht einen einfachen Austausch der Statement-Factory-Klasse, so dass das CBCS auch auf Systemen mit SQL-Dialekten verwendet werden kann. Abbildung 5.3 fasst die wichtigsten Methoden dieser Klasse zusammen.

```
//Erzeugt ein neues Create-Table-Statement
public String createTableStmt(Table cacheTable);

// Erzeugt ein neues Create-Index-Statement
public String createIndexStmt(RCC rcc);

// Erzeugt eine SQL-Anfrage die alle zulässigen Werte für
// eine Spalte zurückliefert.
public String getAllowedFillColumnValues(String tableName,
      String columnName, double percentage);

// Erzeugt ein Statement für eine Existenzanfrage
public String createPrepExistsStmt(String tableName,
      String columnName);

// (Speziell für IBM DB2) Erzeugt ein Create-Nickname-Statement
public String createNickname(String localSchemaAndTableName,
      String remoteSchemaAndTableName, String server);
```

Abbildung 5.3 – Interface der Statement-Factory-Klasse

Alle Methoden liefern als Ergebnis einen SQL-String, der anschließend über die Middle-Tier-Komponente ausgeführt oder in ein Prepared Statement eingebettet werden kann.

Erzeugen der Nicknames

Wie wir bereits zu Anfang des Kapitels in Abschnitt 5.1 erläutert haben, verwenden wir sowohl als Cache-Datenbank als auch als Backend-Datenbank eine IBM DB2. Damit wir die Funktionalität der DB2 zur Föderation von Datenbanksystemen verwenden können, sind einige Einstellungen an der Cache-Datenbank notwendig, die vom Administrator vor der ersten Ausführung des CBCS durchgeführt werden müssen. Dazu sind folgenden Schritte notwendig:

1. Aktivieren der Funktionalität zur Föderation von Datenbanksystemen in der Cache-Datenbank
2. Katalogisieren des Backend-Datenbank-Knotens
3. Katalogisieren der Backend-Datenbank
4. Einrichten eines geeigneten Wrappers
5. Einrichten eines Servers, über den die Verbindung zur Backend-Datenbank hergestellt wird
6. Erzeugen einer Abbildung von Benutzerdaten zur automatischen Authentifizierung auf der Backend-Datenbank

Die Anweisungen zu den einzelnen Schritten sind in Abbildung 5.4 dargestellt. Alle Angaben in kursiver Schrift müssen durch die entsprechenden Werte ersetzt werden. Wird als Backend-Datenbank ebenfalls eine IBM DB2 verwendet, so kann in Schritt 4 der bereits vorhandene Wrapper DRDA verwendet werden. Weitere Informationen zum Einrichten der Funktionalität zur Föderation von Datenbanksystemen finden sich in [IBM05b].

1. update dbm cfg using federated yes
2. catalog tcpip node <*nodename*> remote <*ip-address backend*> server <*port*>
3. catalog database <*backend-db name*> as <*aliasname*> at node <*nodename*> authentication server
4. create wrapper *drda*
5. create server <*servername*> type db2/udb version 8.1. wrapper drda authorisation "<*username*>" password "<*password*>" options (dbname ,<*aliasname*>')
6. create user mapping for <*local-user-name*> server <*servername*> options (remote_authid '<*username*>', remote_password '<*password*>')

Abbildung 5.4 – Einrichten des föderierten Datenbanksystems

Nachdem die Einstellungen durch den Administrator durchgeführt wurden, kann das CBCS die Nicknames generieren. Die Nicknames werden in der Cache-Datenbank

angelegt und erlauben den Zugriff auf die Tabellen der Backend-Datenbank. Die dazu benötigten Anweisungen haben die folgende Form.

```
CREATE NICKNAME <nickname> FOR <servername>.<tablename>
```

Das CBCS erzeugt für jede Tabelle der Backend-Datenbank einen entsprechenden Nickname. Die Nicknames erhalten dabei denselben Namen wie die Backend-Tabelle, die sie referenzieren. Dadurch können Benutzeranfragen direkt auf der Cache-Datenbank ausgeführt werden, da die DB2 dafür sorgt, dass die Anfrage an die Backend-Datenbank weitergeleitet und dort ausgeführt wird. Eventuelle Anpassungen an SQL-Dialekte der Backend-Datenbank werden von dem in Abbildung 5.4 erzeugten Wrapper übernommen.

Erzeugen der Prepared Statements

Nachdem die Nicknames erzeugt wurden, führen wir in der Initialisierungsphase als nächstes die Generierung der SQL-Anweisungen durch. Dabei ist es wichtig, dass die Komponenten nach der Ausführung der generierten Anweisungen wieder einen gültigen Cache-Zustand hinterlassen. Beim Einlagern von Datensätzen in den Cache ist es beispielsweise wichtig, dass für alle von einer Füllspalte durch RCCs erreichbaren Cache-Tabellen entsprechende Insert-Anweisungen generiert werden. Ein weiteres Beispiel findet sich beim Löschen von Datensätzen aus dem Cache. Bei diesem Vorgang ist es wichtig, dass nur die nicht benötigten Datensätze gelöscht werden, da ansonsten der in Cache einen ungültigen Zustand besitzt.

Die Generierung der Existenzanfragen für die Sondierung ist sehr einfach und wird direkt von der Statement-Factory-Klasse unterstützt. Die anderen Anweisungstypen werden wegen ihrer Komplexität in einer separaten Klasse generiert.

Wir wollen an dieser Stelle exemplarisch die Erzeugung der Insert-Anweisungen erläutern. Die Erstellung der anderen Anweisungstypen geschieht, bis auf die Einteilung in atomare Zonen, analog. Abbildung 5.5 zeigt die Methode zur Erzeugung der Insert-Anweisungen in Pseudo-Code-Notation.

Zuerst werden die Cache-Tabellen in atomare Zonen eingeteilt (vgl. Abschnitt 4.5.1). Die topologische Sortierung geschieht in zwei Schritten. Im ersten Schritt werden Beziehungen zwischen den Zonen hergestellt. Wir überprüfen dazu, ob eine Tabelle in einer Zone einen RCC besitzt, der eine Tabelle in einer anderen Zone referenziert. Existiert ein solcher RCC, so verbinden wir die beiden Zonen durch eine gerichtete Kante. Im zweiten Schritt führen wir die topologische Sortierung durch. Dazu suchen wir alle Zonen, die keine eingehenden Kanten besitzen. Diese Zonen werden einer geordneten Liste hinzugefügt und anschließend aus der Menge der Zonen entfernt. Die ausgehenden Kanten dieser Zone werden ebenfalls entfernt. Diesen Schritt wiederholen wir, bis die Menge der Zonen leer ist.

```
createInsertStatement(){
  createAtomicZones();
  sortAtomicZonesTopological();

  for each fillcolumn do{
    create Insert-Statement for controltable;
    put Insert-Statement in the atomic zone that contains
            the controltable;
    clause = " WHERE fillcolum = ?";
    createRecursiv(fillcolumn, clause, duplicateList);
  }
  save all zones and their statements in a list;
}

createAtomicZones(){
  create an atomic zone for each control and cache table;
  for each RCC path do{
    if (RCC path contains a homogenous cycle)
      merge the atomic zones of the tables that take part in the cycle;
  }
}

sortAtomicZonesTopological(){
  sortedList; // list with topological ordered zones
  for each atomic zone A do{
    if(table in zone A contains an RCC that
            references table in another zone B)
      create directed link between zone A and zone B;
  }
  do{
      find all zones with no incoming links;
      add these zones to sortedList;
      remove these zones and their links from graph;
  } while(set contains zones)
}

createRecursiv(column, clause, duplicateList){
  get table that contains the column;
  create Insert-Statement for this table;
  put Insert-Statement in the atomic zone that contains this table;
  for each outgoing RCC of this table do {
    if(RCC is not in duplicateList){
      newClause = " WHERE RCCTargetColumn IN
            (SELECT RCCSourceColumn
             FROM thisTableInBackend " + clause + ")";
      add RCC to duplicateList;
      createRecursiv(RCCTargetColumn, newClause, duplicateList);
    }
  }
}
```

Abbildung 5.5 – Generierung der Insert-Anweisungen in Pseudo-Code

Nach der topologischen Sortierung der Zonen werden die Anweisungen, beginnend bei einer Füllspalte, rekursiv entlang der ausgehenden RCCs erzeugt. Für jede über die RCCs erreichte Tabelle erzeugen wir eine Insert-Anweisung. Die Anweisung wird der atomaren Zone, in der die Tabelle liegt, hinzugefügt und anschließend beginnt der rekursive Abstieg entlang der ausgehenden RCCs dieser Tabelle.

Zur Erkennung von Zyklen führen wir während des Generierungsprozesses eine Menge mit, in der alle RCCs des Pfades gespeichert werden. Befindet sich ein RCC bereits in der Menge, so haben wir einen Zyklus entdeckt und dieser RCC wird nicht weiter verfolgt.

5.2.2 Die Warteschlangen

Im CBCS verwenden wir zur Kommunikation zwischen den einzelnen Komponenten im wesentlichen asynchrone Nachrichten. Dies hat den Vorteil, dass der Sender der Nachricht (in unserem Fall ausschließlich die Query-Worker-Komponente) nicht unnötig blockiert wird.

Da wir mehrere Instanzen des Query Workers während der Initialisierungsphase erzeugen, kann es vorkommen, dass gleichzeitig mehrere Nachrichten an eine Komponente, beispielsweise den Fill Daemon, geschickt werden. Damit keine der Nachrichten verloren geht, verwenden wir Warteschlangen, in denen die eintreffenden Nachrichten gesammelt werden. Die Komponenten arbeiten die Warteschlangen von vorne nach hinten ab und führen die gewünschten Aktionen aus.

Es handelt sich jedoch nicht um persistente Warteschlangen, d. h., ein Systemabsturz hat zur Folge, dass alle Nachrichten in den Warteschlangen verloren gehen. Der Verlust der Nachrichten ist jedoch nicht weiter tragisch, da die Nachrichten lediglich Anweisungen zum Aktualisieren der Zugriffsstatistik oder zum Einlagern eines Werts beinhalten. Geht beispielsweise eine Nachricht zum Einlagern eines Werts verloren, so wird bei der nächsten Sondierung nach diesem Wert wieder eine neue Nachricht erzeugt und verschickt.

Die Warteschlangen befinden sich im Paket *queue* und besitzen insgesamt drei Methoden. Eine Methode zum Hinzufügen einer Nachricht, eine zum Entnehmen einer Nachricht und eine Methode, um die Warteschlage zu schließen. Nach dem Schließen der Warteschlange können keine weiteren Nachrichten mehr hinzugefügt bzw. entnommen werden. Alle Methoden der Warteschlange sind durch einen Semaphor geschützt, d. h., es erhält immer nur eine Komponente Zugriff auf die Methoden, alle anderen Komponenten müssen warten, bis diese Komponente die Methode wieder freigibt. Durch die Semaphore verhindern wir, dass mehrere Komponenten gleichzeitig auf die Warteschlange zugreifen können und dadurch eventuell Nachrichten verloren gehen.

Abbildung 5.6 zeigt die Implementierung der Warteschlangen in Pseudo-Code-Notation. Wie in Abbildung 5.6 zu sehen ist, sorgt die Methode *getEntry()* dafür, dass keine Fehler entstehen, wenn die Warteschlange leer ist, indem sie die aufrufende Komponente in den Wartezustand setzt. Die Komponente verlässt erst durch das Hinzufügen einer Nachricht oder das Schließen der Warteschlange wieder den Wartezustand.

Wurde die Komponente durch das Schließen der Warteschlange wieder aktiv, so entnimmt sie keine Nachricht und verlässt die Methode.

```
public synchronized void addEntry(message) {
  if (queue not closed) {
     add message to Queue;
  }
  notify all waiting components;
}
public synchronized Message getEntry() {
  if (queue not closed) {
    while (queue is empty and not closed) {
        wait;
    }
    if (queue not closed) {
      get first message from queue;
    }
  }
  notify all waiting components;
  return message;
}
public synchronized void close() {
  close queue;
  notify all waiting components;
}
```

Abbildung 5.6 – Implementierung der Warteschlangen in Pseudo-Code

5.2.3 Der Query Worker

Von der Query-Worker-Komponente existieren im CBCS mehrere Instanzen, die von einer Manager-Komponente verwaltet werden. Fordert der Benutzer über die JDBC-Schnittstelle (vgl. Abschnitt 5.2.8) eine Datenbankverbindung an, so leitet die JDBC-Schnittstelle diese Forderung an die Manager-Komponente weiter, die daraufhin diesem Benutzer einen freien Query Worker zuweist. Alle Anfragen des Benutzers werden anschließend durch den zugewiesenen Query Worker bearbeitet und ausgeführt. Wird der Query Worker vom Benutzer nicht weiter benötigt, beispielsweise weil dieser die Datenbankverbindung beendet, so wird der Query Worker wieder von der Manager-Komponente in den Pool eingelagert.

Bevor eine Anfrage durch den Query Worker ausgeführt wird, wird die Anfrage zuerst gegen die Syntax aus Abbildung 4.4 (Seite 46) validiert. Wir verwenden dafür im CBCS einen SQL-Parser, der mit Hilfe des Parser-Generators ANTLR (ANother Tool for Language Recognition [Parr05]) erzeugt wurde. Dazu war es notwendig, die Syntax aus Abbildung 4.4 in eine EBNF-Grammatik (Extended Backus-Naur Form) zu überführen und diese als Eingabe dem Parser-Generator zu übergeben. Die komplette EBNF-Notation der SQL-Syntax und eine kurze Erläuterung finden sich in Anhang C.

Jede Anfrage des Benutzers wird vom SQL-Parser daraufhin untersucht, ob sie der

spezifizierten Grammatik entspricht. Entspricht die Anfrage des Benutzers nicht der Grammatik, so erzeugt der SQL-Parser eine Fehlermeldung, die vom Query Worker abgefangen wird. Die Fehlermeldung ist für den Query Worker der Hinweis, dass er diese Anfrage nicht bearbeiten kann und er sie an die Backend-Datenbank weiterleiten soll. Für alle Anfragen, die der spezifizierten Grammatik entsprechen, erzeugt der SQL-Parser eine Baumstruktur und gibt diesen Baum an den Query Worker zurück. Wir wollen den Aufbau der Baumstruktur an folgender Anfrage kurz erläutern:

```
SELECT titel FROM Film WHERE id = 4711
```

Der SQL-Parser erstellt aus dieser Anfrage den in Abbildung 5.7 dargestellten Baum. Wie man leicht erkennen kann, besitzt jede Klausel der SQL-Anfrage einen eigenen Knoten, der direkt an den Wurzelknoten angehängt ist. Der Select-Knoten besitzt jeweils einen oder mehrere Unterknoten (SelectItem-Knoten). Diese Unterknoten dienen zur Kapselung der Informationen einer Spalte der Ausgabe. Sie enthalten den Spaltennamen, den Tabellennamen und den Alias-Namen, soweit angegeben.

Analog zu den Unterknoten des Select-Knotens existieren auch Unterknoten für den From-Knoten. Diese Unterknoten kapseln die Informationen einer Tabelle der From-Klausel. Die Unterknoten der Where-Klausel fassen jeweils ein Prädikat der Klausel zusammen, wobei der Vergleichsoperator des Prädikats als direkter Unterknoten der Where-Klausel verwendet wird.

Der generierte Baum des SQL-Parsers mag auf den ersten Blick etwas verwirrend wirken, er bietet jedoch für die spätere Verarbeitung in der Query-Worker-Komponente einige Vorteile. Die Unterknoten der Where-Klausel ermöglichen das schnelle Auffinden aller Gleichheitsprädikate der Anfrage. Alle Prädikate, die einen anderen Vergleichsoperator besitzen, werden einfach ignoriert. Durch die beiden Knoten Leftside bzw. Rightside können wir sehr einfach feststellen, ob es sich um ein Prädikat der Form $Spalte = Wert$, $Spalte = Spalte$ oder $Wert = Wert$ handelt.

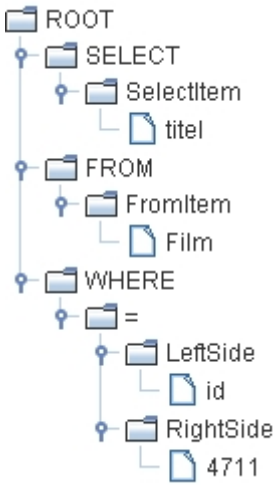


Abbildung 5.7 – Erstellte Baumstruktur des SQL-Parsers

Nachdem der Query Worker alle möglichen Einstiegspunkte gefunden hat und die entsprechenden Cache-Tabellen verankert wurden, können wir mit Hilfe der Unterknoten in der Select- und From-Klausel ganz einfach die Tabellen der Anfrage bestimmen, die durch Cache-Tabellen ersetzt werden müssen. Wir laufen dazu über alle SelectItem- und FromItem-Knoten und überprüfen die darin enthaltenen Tabellennamen.

Die Analyse und Verarbeitung der Anfrage in der Query-Worker-Komponente geschieht in vier Schritten, die wir im Anschluss näher erläutern werden:

1. Auslesen der Tabellen in der From-Klausel
2. Analyse der Prädikate der Where-Klausel, sowie Finden von Einstiegspunkten
3. Verankern der Tabellen und Modifizieren der ursprünglichen Anfrage
4. Ausführen der modifizierten Anfrage

Der erste Bearbeitungsschritt des Query Worker besteht darin, dass alle Tabellen- und Alias-Namen der From-Klausel ausgelesen und in einer Liste gespeichert werden. Dabei werden nur die Tabellen in der Liste gespeichert, die eine entsprechende Cache-Tabelle im Cache besitzen. Die Tabellen in der Liste werden später bei der Analyse der Prädikate in der Where-Klausel verwendet, um den Spaltennamen ihre zugehörige Tabelle zuzuordnen.

Im zweiten Schritt werden die Prädikate der Where-Klausel analysiert und ein Graph, wie in Abschnitt 4.4.1 auf Seite 45 beschrieben, erstellt. Wir wollen an dieser Stelle nicht weiter auf die Analyse der Prädikate eingehen, da diese bereits ausführlich in Abschnitt 4.4.1 erläutert wurde.

In Schritt 3 der Anfrageanalyse werden alle verankerten Tabellen bestimmt bzw. aus dem in Schritt 2 erstellten Graph ausgelesen und anschließend die Anfrage modifiziert. Dazu werden alle in Schritt 2 gefundenen Einstiegspunkte in einer Liste gespeichert, die im dritten Schritt als Ausgangspunkte für die Verankerung dienen. Für jeden Einstiegspunkt in der Liste werden rekursiv alle ausgehenden Kanten durchlaufen und aus allen erreichbaren Knoten die Tabelleninformationen ausgelesen. Diese Informationen werden in einer Menge gespeichert und ermöglichen eine Zuordnung zwischen Cache-Tabellen und den verwendeten Tabellen- bzw. Alias-Namen in der Anfrage.

Nach dem Auslesen aller verankerten Tabellen muss die ursprüngliche Anfrage des Benutzers modifiziert werden. Dabei werden alle Backend-Tabellennamen der Anfrage, die einer verankerten Tabelle zugeordnet sind, durch Cache-Tabellennamen ersetzt. Dazu wird der vom SQL-Parser generierte Baum vollständig durchlaufen und für jeden gefundenen Tabellennamen wird überprüft, ob sich dieser in der angelegten Menge befindet. Befindet er sich in der Menge, so wird dieser Tabellenname ersetzt.

Der letzte Bearbeitungsschritt des Query Worker ist das Ausführen der modifizier-

ten Anfrage. Dazu wird die modifizierte Anfrage über die Middle-Tier-Komponente ausgeführt und das Ergebnis der Anfrage über die JDBC-Schnittstelle an den Benutzer zurückgegeben.

5.2.4 Der Fill Daemon

Der Fill Daemon wird während der Initialisierung in einem eigenständigen Thread gestartet. Die Implementierung des Fill Daemon ist sehr einfach gehalten, da wir einen Großteil der Komplexität in die Insert-Anweisungen verlagert haben. Der Fill Daemon besteht daher nur aus zwei einfachen Methoden. Die erste Methode startet den Thread, in dem der Fill Daemon in einer Endlosschleife solange seinen Dienst verrichtet, bis er geschlossen wird. Zum Schließen des Fill Daemon dient die zweite Methode, in der ein boolescher Wert gesetzt wird, der zum Abbruch der Endlosschleife führt.

Während sich der Fill Daemon in der Endlosschleife befindet, entnimmt er seiner Warteschlange eine Nachricht. Er überprüft, ob die Nachricht eine Füllspalte enthält und ob der in der Nachricht vermerkte Wert überhaupt in den Cache eingelagert werden darf. Weiterhin überprüft der Fill Daemon, ob der Wert eventuell schon in den Cache eingelagert wurde. Da mehrere Query Worker parallel Anfragen von Benutzern bearbeiten, kann es vorkommen, dass zwei oder mehr Query Worker Nachrichten mit gleichem Inhalt in die Warteschlange stellen. Der Fill Daemon entnimmt die erste Nachricht und lagert die entsprechenden Daten ein. Bei der zweiten Nachricht würde die Einlagerung zu Fehlermeldungen führen. Durch die Überprüfung werden nicht nur diese Fehlermeldungen, sondern auch unnötige Einlagerungsvorgänge vermieden.

```
public void run() {
  while (filldaemon not closed) {

    get next message from queue;
    if (message returned) {
        if (column in message is not a fillcolumn)
           return;

        if(value in message is not allowed to cache)
           return;

        if(value in message already exists in cache)
           return;

        get F-Lock from semaphore;
        get list of insert statements from listOfStatements;
        set parameter in statements = value in message;
        execute statements;
        release F-Lock;
    }
  }
}
```

Abbildung 5.8 – Implementierung des Fill Daemon in Pseudo-Code

Stellt der Fill Daemon fest, dass der geforderte Wert in der Füllspalte noch nicht vorhanden ist, so fordert er eine F-Sperre an (vgl. Abschnitt 4.7.2). Anschließend entnimmt er aus einem Container die benötigten während der Initialisierung generierten Insert-Anweisungen und führt diese aus. Nach der erfolgreichen Einlagerung wird die F-Sperre wieder freigegeben und der Fill Daemon entnimmt die nächste Nachricht aus der Warteschlange und so fort. In Abbildung 5.8 ist die Vorgehensweise des Fill Daemon nochmals in Pseudo-Code-Notation zusammengefasst.

5.2.5 Der Hit Counter

Die Implementierung der Hit-Counter-Komponente ist der des Fill Daemon sehr ähnlich. Wir wollen sie trotzdem aus Gründen der Vollständigkeit in diesem Abschnitt kurz erläutern. Auch der Hit Counter enthält sehr wenig Programmcode, da wir, wie beim Fill Daemon, die Komplexität zum Aktualisieren der Statistiken in die SQL-Anweisungen verlagert haben.

Wie der Fill Daemon läuft auch der Hit Counter in einem eigenen Thread, der nach der Initialisierung gestartet wird. Auch der Hit Counter besitzt nur zwei Methoden: eine zum Starten und eine zum Schließen der Komponente. Der Programmcode der Hit-Counter-Komponente ist in Abbildung 5.9 in Pseudo-Code-Notation dargestellt.

```
public void run() {
  while (hitcounter not closed) {

    get next message from queue;
    if (message returned) {
      get list of update statements from listOfStatements;
      set parameter in statements = value in message;
      execute statements;
    }
  }
}
```

Abbildung 5.9 – Implementierung des Hit Counter in Pseudo-Code

Nach dem Start des Hit Counter läuft dieser in einer Endlosschleife, bis die Methode zum Schließen der Komponente aufgerufen wird. In jedem Durchlauf entnimmt er seiner Warteschlange eine Nachricht. Jede Nachricht enthält Angaben darüber auf welcher Spalte und für welchen Wert die Sondierung ein positives Ergebnis geliefert hat. Der Hit Counter besorgt sich daraufhin aus dem Container mit SQL-Anweisungen die Liste mit den entsprechenden Update-Anweisungen und führt diese aus. Anschließend beginnt der Vorgang von vorne.

5.2.6 Der Garbage Collector

Das Löschen von Datensätzen in der Cache-Datenbank wird durch den Garbage Collector übernommen. Der Garbage Collector läuft ebenfalls in einem eigenständigen Thread und überprüft periodisch, ob der Cache einen vom Administrator spezifizierten Füllstand überschritten hat. Wurde der Füllstand überschritten, so führt der Garbage Collector solange Löschvorgänge durch, bis der Füllstand wieder unterschritten wird.

Die Garbage-Collector-Komponente befindet sich im Paket *gcollector*. Auch bei der Implementierung dieser Komponente haben wir die Komplexität aus der Komponente in die Delete-Anweisungen verlagert. Dadurch ist der eigentliche Programmcode des Garbage Collectors sehr einfach.

Wie der Fill Daemon und der Hit Counter besitzt auch der Garbage Collector nur sehr wenige Schnittstellen nach außen. Diese beschränken sich auf Methoden zum Starten und Schließen der Komponente sowie einer Methode zum manuellen Durchführen eines Löschvorgangs. Auch der Garbage Collector läuft nach seinem Start in einer Endlosschleife, bis er durch die Methode *close()* beendet wird.

Bei jedem Durchgang überprüft der Garbage Collector, ob der vom Administrator spezifizierte Füllstand überschritten wurde. Wurde er noch nicht überschritten, so wartet der Thread eine vorgegebene Zeitspanne und überprüft den Füllstand erneut. Wird der spezifizierte Füllstand überschritten, so bestimmt der Garbage Collector in den Kontrolltabellen ein Opfer zum Löschen (vgl. Abschnitt 4.7.1). Anschließend wird eine G-Sperre (vgl. Abschnitt 4.7.2) angefordert; die entsprechenden Delete-Anweisungen aus einem Container entnommen und ausgeführt. Danach wird die G-Sperre wieder freigegeben und der Vorgang beginnt mit dem Prüfen des Füllstands von neuem. Abbildung 5.10 fasst die Vorgehensweise des Garbage Collectors nochmals in Pseudo-Code-Notation zusammen.

```
public void run() {
  while (gcollector not closed) {
    if (fill level not reached) {
       wait a certain time;
    } else {
      find victim in control tables;
      get G-Lock from semaphore;
      get list of delete statements from listOfStatements;
      set parameter in statements = value of victim;
      execute statements;
      release G-Lock;
    }
  }
}
```

Abbildung 5.10 – Implementierung des Garbage Collector in Pseudo-Code

5.2.7 Die Middle-Tier-Komponente

Die Middle-Tier-Komponente des CBCS ist eine der wenigen Komponenten, die bei Änderungen der Umgebung angepasst werden müssen. Wir wollen daher an dieser Stelle nur der Vollständigkeit halber auf diese Komponente kurz eingehen.

Die Middle-Tier-Komponente stellt, wie bereits mehrfach erwähnt, für die anderen Komponenten des CBCS die Schnittstelle zur Cache- und Backend-Datenbank dar. In Abbildung 5.11 sind alle Methoden der Middle-Tier-Komponente dargestellt, die vom CBCS benötigt werden.

Die Implementierung der einzelnen Methoden ist in unserem Fall sehr einfach, da die IBM DB2 Funktionalität zur Föderation von Datenbanksystemen anbietet, die die komplette Verteilung der Anfragen an Backend- und Cache-Datenbank übernehmen. Dies lässt sich sehr gut am Beispiel der Methode *executeStatement()* veranschaulichen. In der von uns verwendeten Umgebung (vgl. Abschnitt 5.1) führt der Aufruf dieser Methode lediglich dazu, dass wir die Anfrage auf der Cache-Datenbank ausführen, da wir mit Hilfe der Nicknames auch auf die Tabellen der Backend-Datenbank zugreifen können. In einer anderen Umgebung müssten wir die Anfrageoptimierung und die Verteilung der Anfrage auf die Cache- und Backend-Datenbank selbst übernehmen bzw. implementieren, ebenso wie das Zusammenführen der einzelnen Anfrageergebnisse der Datenbanken.

```
// Transaktion beginnen
public void begin()

// Transaktion abschließen
public void commit()

// Transaktion zurücksetzen
public void rollback()

// Schließen der Datenbankverbindung(en)
public void close()

// Ausführen einer lesenden Anweisung (Cache + Backend)
public ResultSet executeStatement(String stmt)

// Ausführen einer ändernden Anweisung (Cache + Backend)
public int executeUpdate(String stmt)

// Erzeugen eines Prepared Statements
public PreparedStatement getPreparedStatement(String stmt)

// Ausführen eines lesenden Prepared Statements
public ResultSet executePreparedStatement
      (PreparedStatement prepStmt)
```

Abbildung 5.11 – Schnittstellen der Middle-Tier-Komponente

5.2.8 Die JDBC-Schnittstelle

Die JDBC-Schnittstelle leitet alle Anfragen der Benutzer an das CBCS weiter. Für den Benutzer geschieht dieser Vorgang völlig transparent. Aus seiner Sicht arbeitet die von uns implementierte JDBC-Schnittstelle wie ein normaler JDBC-Treiber.

Alle zur JDBC-Schnittstelle gehörenden Klassen befinden sich in einer separaten Jar-Datei, die auf Client-Seite in die (Web-)Anwendungen eingebunden wird. Die Java-Klassen dieser Jar-Datei sind in Abbildung 5.12 dargestellt. Alle dargestellten Klassen implementieren die entsprechenden Interfaces aus dem Java-Paket *java.sql*, das bei jeder Java-Installation mitgeliefert wird.

Eine Verbindung zur Datenbank wird mit Hilfe der Klasse *DriverManager* hergestellt. Bei Aufruf der Methode *getConnection(url, port)* wird über das RMI-Protokoll (Remote Method Invocation) auf die durch die URL spezifizierte RMI-Registry zugegriffen, in der sich der Query Worker Manager registriert hat. Anschließend liefert der Query Worker Manager einen freien Query Worker zurück. Der Query Worker wird in ein neues Objekt der Klasse *Connection* verpackt und an den Benutzer bzw. die (Web-)Anwendung zurückgegeben.

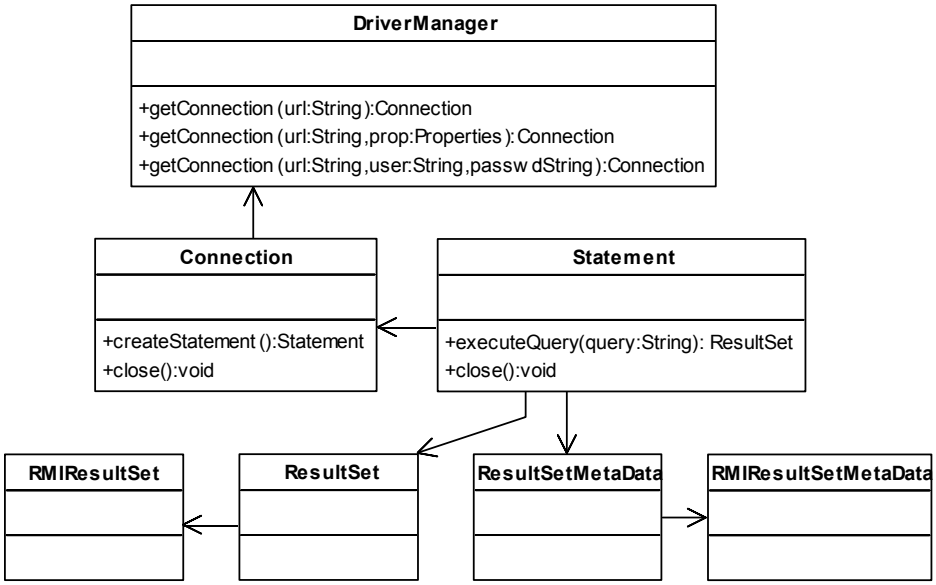


Abbildung 5.12 – Klassenstruktur der JDBC-Schnittstelle

Über das Connection-Objekt kann ein neues Objekt der Klasse *Statement* erzeugt werden, über das die Anfragen an die Datenbank gestellt werden können. Soll eine Anfrage ausgeführt werden, so muss die entsprechende Methode der Klasse Statement aufgerufen werden. Dabei wird intern die Anfrage an den Query Worker weitergeleitet, der diese bearbeitet und das Anfrageergebnis zurückliefert.

Das Anfordern eines freien Query Worker und die Bearbeitung der Anfragen geschehen über das RMI-Protokoll. Diese Vorgehensweise brachte allerdings einige Pro-

bleme mit sich, die wir an dieser Stelle kurz erläutern wollen. Das RMI-Protokoll erlaubt einem Client, Methodenaufrufe auf einem Server durchzuführen. Der Aufruf diese Remote-Methode geschieht analog zum Aufruf einer lokalen Methode. Im Gegensatz zum Aufruf einer lokalen Methode müssen beim Aufruf der Remote-Methode alle Übergabeparameter und Rückgabewerte das Interface *Remote* implementieren oder serialisierbar sein, d. h. die Übergabeparameter und Rückgabewerte müssen entweder über das Netz ansprechbar oder ohne weiteres in den Speicher des anderen Rechners übertragbar sein. In Java erfüllen alle einfachen Datentypen, wie Integer, String, Boolean usw. diese Bedingung und können daher ohne Probleme über das RMI-Protokoll an einen anderen Rechner geschickt werden.

Leider erfüllt die Klasse *ResultSet*, die die Ergebnisse einer Datenbankanfrage verwaltet, diese Bedingungen nicht und kann daher auch nicht über das RMI-Protokoll verschickt werden. Ein Grund dafür ist, dass das Interface *java.sql.ResultSet* einige Methoden bereitstellt, die mit Eingabeströmen arbeiten. Diese Eingabeströme sind nicht serialisierbar und verhindern dadurch die Serialisierbarkeit der ResultSet-Klasse.

Das Problem kann jedoch umgangen werden, indem das ResultSet-Objekt der Anfrage auf Server-Seite, d. h. beim CBCS, bleibt und der Zugriff auf die Methoden des ResultSet über eine serialisierbare Klasse geschehen. Wir haben dafür die beiden Klassen *RMIResultSet* und *RMIResultSetMetaData* implementiert, die den Zugriff auf die Interfaces *ResultSet* und *ResultSetMetaData* über das RMI-Protokoll ermöglichen. Alle Methoden der Klasse *ResultSet* und *ResultSetMetaData*, die die Serialisierung dieser Klassen verhindern, wurden dabei nicht implementiert und erzeugen beim Aufruf eine Fehlermeldung.

Auf Benutzerseite existieren die beiden Klassen *ResultSet* und *ResultSetMetaData*, die die Interfaces *ResultSet* und *ResultSetMetaData* implementieren und den Zugriff auf die Klassen *RMIResultSet* und *RMIResultSetMetaData* erlauben.

Durch das Weglassen der nicht-serialisierbaren Methoden haben wir auch einen Teil des Funktionsumfangs, den die Klasse ResultSet üblicherweise zur Verfügung stellt, verloren. Dennoch sind rund 80% der Methoden dieser Klasse verfügbar, und die Implementierung der übrigen Methoden ist für zukünftige Versionen des CBCS geplant.

6 Zusammenfassung und Ausblick

An dieser Stelle wollen wir alle wichtigen Aspekte, die wir in den vorangegangenen Kapiteln vorgestellt haben, kurz zusammenfassen. Darüber hinaus sollen einige Anregungen für die weitere Entwicklung des CBCS-Prototyps gegeben werden.

Wir haben gesehen, dass Web-Anwendungen sich immer größerer Beliebtheit erfreuen und einen großen Anteil am Erfolg von E-Commerce-Anwendungen haben. Diese Web-Anwendungen erzeugen üblicherweise dynamische Web-Seiten, die aus Informationen aus einer Datenbank generiert werden. Wir haben in Kapitel 1 gesehen, dass die Datenbank dadurch leicht zum Flaschenhals werden kann. Um diesem Problem entgegen zu wirken, mussten Verfahren entwickelt werden, die diese Last von den Datenbanken nehmen und auf andere Rechner verteilen.

Eines dieser Verfahren, das Constraint-basierte Datenbank Caching, haben wir in Kapitel 2 ausführlich erläutert. Wir haben die Grundbegriffe dieses Verfahrens wie beispielsweise Cache Group, Füllspalte, RCC und Sondierungsverfahren erklärt und das Konzept des Constraint-basierten Datenbank Caching durchleuchtet. Dazu haben wir unter anderem die einzelnen Cache-Constraints an ausführlichen Beispielen erläutert. Auch eine kritische Betrachtung des Constraint-basierten Datenbank-Caching haben wir vorgenommen, indem wir mögliche Probleme bei der Verwendung von RCC-Zyklen herausgearbeitet haben.

Kapitel 3 beschäftigte sich ausschließlich mit bereits bestehenden Prototypen anderer Forschungsgruppen. Wir haben in diesem Kapitel zum einen ein weiteres Verfahren zum Caching von dynamischen Web-Seiten, das Query-Result-Caching vorgestellt und ausführlich erläutert. Zum anderen haben wir die Datenbank-Caching Prototypen von Microsoft (MTCache) und IBM (DBCache) vorgestellt. Dabei ging es uns vor allem darum, die Funktionsweise beider Prototypen und die eingesetzten Verfahren zu erläutern und die Vor- und Nachteile beider Systeme zu analysieren. Wie wir am Ende von Kapitel 3 gesehen haben, besitzen beide Systeme in bestimmten Bereichen Vorteile gegenüber dem anderen Prototyp.

Diese Stärken-Schwächen-Analyse war auch der Grund für die Konzeption und Realisierung des CBCS-Prototyps. Diesen Prototyp haben wir in Kapitel 4 sehr ausführlich vorgestellt. Der Prototyp basiert auf dem Konzept des Constraint-basierten Daten-

bank-Caching. Er gliedert sich in mehrere Komponenten, die alle eine sehr geringe Kopplung untereinander besitzen und dadurch leicht ausgetauscht werden können. Den Kern des Prototyps bilden die Query Worker, die die Anfragen der Benutzer analysieren und versuchen möglichst große Teile der Anfrage durch den Cache zu beantworten. Drei zusätzliche Komponenten sorgen für das Füllen und Löschen des Cache sowie für das Führen von Zugriffsstatistiken.

Im letzten Kapitel sind wir dann etwas genauer auf die Implementierung des CBCS eingegangen und haben anhand von Code-Fragmenten die wichtigsten Passagen des Codes veranschaulicht. Dabei standen auch in diesem Kapitel die Query-Worker-Komponente sowie die Komponenten zum Füllen und Löschen des Cache im Vordergrund. Ebenfalls haben wir in diesem Kapitel die JDBC-Schnittstelle, die auf der Benutzerseite eingesetzt wird, vorgestellt.

Wie geht es weiter?

Die Entwicklung des CBCS wird auch in Zukunft weitergehen, da noch viele Erweiterungsmöglichkeiten für diesen Prototyp denkbar sind. Das Ziel des ersten CBCS-Prototyps war die Entwicklung eines funktionsfähigen Datenbank-Cache, der den aktuellen Stand der Forschung auf dem Gebiet des Constraint-basierten Datenbank Caching repräsentiert und zugleich die Nachteile der Prototypen aus Kapitel 3 vermeidet. Aus diesem Grund stand die Implementierung der wichtigsten Funktionen im Vordergrund, während die Geschwindigkeit des Systems zweitrangig war.

Das CBCS vereint die vom MTCache bekannten Kontrolltabellen mit dem Caching-Konzept des DBCache. Des Weiteren verwendet es ein weiterentwickeltes Sondierungsverfahren, das mehr potenzielle Einstiegspunkte in eine Cache Group liefert als das Sondierungsverfahren des DBCache. Die Sondierung selbst arbeitet wesentlich effektiver, da jedes Prädikat der Where-Klausel separat als Einstiegspunkt überprüft wird.

Auch ein Wechsel der verwendeten Datenbanken ist beim CBCS einfacher als bei den anderen beiden Prototypen. Zwar ist die Anpassung der Middle-Tier-Komponente an die neue Umgebung nicht trivial, jedoch wesentlich einfacher als die Anpassung eines kompletten Datenbanksystems.

Auch das CBCS besitzt einige Nachteile, die wir an dieser Stelle nicht verschweigen wollen. Einer der größten Nachteile ist, dass die Query Worker des CBCS noch nicht die komplette SQL-Syntax verstehen, bzw. verarbeiten können. Die entsprechenden Anfragen können zwar ausgeführt werden, werden dann allerdings an die Backend-Datenbank weitergeleitet bzw. erzeugen eine Fehlermeldung (z. B. Ausführung von Änderungsoperationen). Wir werden im Anschluss noch auf mögliche Erweiterungen des CBCS eingehen und diesen Punkt nochmals aufgreifen.

Funktionale Erweiterungen

Für die zukünftige Entwicklung des CBCS gibt es mehrere Richtungen, in die Verbesserungen vorgenommen werden können. Einer der wichtigsten Bereiche ist der Ausbau der Funktionalität des CBCS. Dazu gehört unter anderem die Erweiterung der von der Query-Worker-Komponente verstandenen SQL-Syntax. Ein Beispiel dafür wäre die Unterstützung von geschachtelten Select-Anweisungen. Bei der Bearbeitung muss natürlich darauf geachtet werden, dass ein gefundener Einstiegspunkt in einer inneren Select-Anweisung nicht zur Ersetzung von Tabellen in den äußeren Select-Anweisungen führt. Durch diese Erweiterung kann ein noch größerer Umfang an Anfragen durch die Query Worker analysiert und bearbeitet werden, was zu einer weiteren Entlastung der Backend-Datenbank führt.

Ebenfalls sollte die Möglichkeit zur Bearbeitung von Änderungsoperationen als Erweiterung in Betracht gezogen werden. Wir haben bereits in Abschnitt 4.4.2 dieses Thema kurz angesprochen. Dazu muss auf der einen Seite eine Komponente (Synchronizer) für die Backend-Datenbank entwickelt werden, die die Änderungsoperationen der einzelnen CBCS entgegennimmt und die Änderungen auf der Backend-Datenbank durchführt. Zusätzlich muss diese Komponente alle von den Änderungen betroffenen CBCS informieren, damit diese ihren Datenbestand aktualisieren.

Auf der anderen Seite müssen im CBCS Erweiterungen durchgeführt werden, damit Änderungsoperationen unterstützt werden. Dazu gehört unter anderem die Erweiterung der Query-Worker-Komponente derart, dass diese die ankommenden Änderungsoperationen an den Synchronizer weiterleitet. Zusätzlich muss entweder eine neue Komponente implementiert oder der Fill Daemon um die entsprechende Logik erweitert werden, sodass die Nachrichten des Synchronizers verarbeitet werden können. Die Maßnahmen, die bei der jeweiligen Nachricht durchgeführt werden müssen, haben wir in Abschnitt 4.4.2 ausführlich erläutert. In diesem Zusammenhang muss auch die für die Situation benötigte Konsistenzstufe berücksichtigt werden (vgl. Abschnitt 4.4.2). Zwar ist in diesem Bereich noch einige Forschungsarbeit notwendig; die Bearbeitung von Änderungsoperationen sollte in naher Zukunft jedoch vom CBCS unterstützt werden.

In den Bereich der Funktionserweiterung gehört auch die Weiterentwicklung der JDBC-Schnittstelle, sodass diese in naher Zukunft den Benutzern den kompletten Funktionsumfang eines JDBC-Treibers zur Verfügung stellt. Dazu müssen Lösungen gefunden werden, die den Aufruf der bisher noch nicht unterstützten Methoden der Klasse *ResultSet* ermöglichen.

Verbesserung der Performance

Ein weiterer wichtiger Bereich zur Weiterentwicklung des CBCS ist der Bereich Performance. Bislang haben wir noch keine größeren Stresstests mit dem CBCS durchgeführt, sodass wir zu diesem Zeitpunkt noch keine Ergebnisse vorlegen können, die die

Leistungsfähigkeit des CBCS veranschaulichen. Jedoch sind für die nahe Zukunft ausführliche Testreihen mit dem CBCS geplant.

Anhand dieser Tests wird es dann möglich sein, den Programmcode des CBCS weiter zu optimieren und gegebenenfalls bestehende Komponenten durch effizientere Implementierungen zu ersetzen. Dies betrifft vor allem die Query-Worker-Komponente sowie den Fill Daemon und den Garbage Collector. Im Falle des Garbage Collector müssen diese Tests auch zeigen, wie alltagstauglich die verwendete Verdrängungsstrategie (LRU) ist und ob eventuell andere Verdrängungsstrategien wie beispielsweise der Clock-Algorithmus bessere Ergebnisse liefern. Optimierungsmöglichkeiten bieten auch die generierten Insert-, Update- und Delete-Anweisungen. Untersuchungen müssen zeigen, ob diese Anweisungen weiter verbessert werden können, d. h. kann der Aufbau der Anweisungen vereinfacht werden und können gegebenenfalls mehrere Insert-Anweisungen auf eine Tabelle zusammengefasst werden, sodass während des Einlagerungsvorgangs nur eine Insert-Anweisung pro Cache-Tabelle durchgeführt werden muss. Diese Optimierungen müssen jedoch sehr genau überprüft werden, damit stets ein gültiger Cache-Zustand nach der Durchführung besteht.

Erweiterung der Benutzerfreundlichkeit

Ein weiterer Bereich zur Weiterentwicklung des CBCS ist die Benutzerfreundlichkeit, speziell die Bedienbarkeit des CBCS durch den Administrator. Wir haben bereits erwähnt, dass das CBCS über eine Kommandozeile verfügt, über die der Administrator einige der Komponenten beenden und wieder starten kann. Zusätzlich existieren einige Befehle zum Testen der CBCS-Komponenten. Vorstellbar wäre für diesen Bereich eine graphische Benutzeroberfläche, auf der der Administrator die komplette Konfiguration des CBCS übernehmen kann. Weiterhin könnte diese Benutzeroberfläche auch die Möglichkeit bieten, dass der Administrator auf Fehlermeldungen während der Initialisierung des Systems reagieren kann und gegebenenfalls die Möglichkeit erhält, vorhandene Probleme wie beispielsweise heterogene Zyklen in der Cache-Group-Definition aufzulösen.

Wie sich in den genannten Erweiterungsmöglichkeiten schon erkennen lässt, ist die Entwicklung des CBCS noch lange nicht beendet. Solange die Forschung auf dem Gebiet des Constraint-basierten Datenbank-Caching noch offene Fragen besitzt, die nach einer Antwort verlangen, kann das CBCS auch verbessert und weiterentwickelt werden.

Anhang A – Weitere Datenbank-Cache-Implementierungen

An dieser Stelle wollen wir noch einige Konzepte für Datenbank-Caching vorstellen, auf die wir im Hauptteil nicht genauer eingehen konnten. Dennoch sind diese Konzepte interessant genug, um hier vorgestellt zu werden.

Caching von dynamischen Inhalten mit Hilfe von Code-Transformation

Eine weitere Möglichkeit zum Caching von dynamisch generierten Web-Seiten wird in [BoMZ03] aufgezeigt. Der dort beschriebene Prototyp untersucht mit Hilfe eines Compilers den Byte-Code der Java-Web-Anwendungen auf dem Anwendungsserver und erweitert ihn. Durch die Erweiterung des Byte-Codes werden zusätzliche Informationen für den Cache generiert, die anschließend für ein besseres Caching sorgen. Diese Vorgehensweise wollen wir im Folgenden kurz beschreiben.

Der Compiler fügt in jede Klasse, die eine Web-Seite aus SQL-Anweisungen dynamisch zusammenbaut, einen Pre-Processing-Schritt ein. In diesem Schritt wird überprüft, ob die benötige Seite bereits im Cache enthalten ist. Wurde die Seite im Cache gefunden, so wird sie zurückgegeben, und der restliche Programm-Code wird nicht bearbeitet. Zusätzlich zum Pre-Processing-Schritt wird im Byte-Code nach jeder lesenden SQL-Anweisung eine zusätzliche Zeile eingefügt, die für den Cache Informationen über die gerade ausgeführte SQL-Anweisung sammelt. Nach jeder Änderungsoperation (Insert, Update, Delete) fügt der Compiler eine Zeile ein, die Informationen zum Invalidieren des Cache beinhaltet.

Durch diese zusätzlichen Informationen ist es für den Cache möglich zu erkennen, ob die lesenden und schreibenden Zugriffe der Seiten im Konflikt zueinander stehen oder nicht. Werden zum Beispiel alle schreibenden Zugriffe auf Spalten ausgeführt, die niemals von lesenden Zugriffen referenziert werden, so müssen die Daten im Cache nach einer Änderungsoperation nicht aktualisiert werden. Im umgekehrten Fall kann der Cache durch die gesammelten Informationen effizient invalidiert werden.

Am Ende folgt ein Post-Processing-Schritt, der die gerade erzeugte Web-Seite und die gesammelten Informationen in den Cache schreibt. Ebenfalls wird im Post-Processing-Schritt der Cache mit Hilfe der gesammelten Informationen invalidiert. Eine Zusammenfassung dieser Schritte findet sich in Abbildung A.1.

Auf die Verwaltung des Cache-Inhalts sowie die Invalidierung der Daten im Cache wollen wir an dieser Stelle nicht weiter eingehen. Eine detaillierte Beschreibung der Cache-Verwaltung und Invalidierung findet sich [BoMZ03].

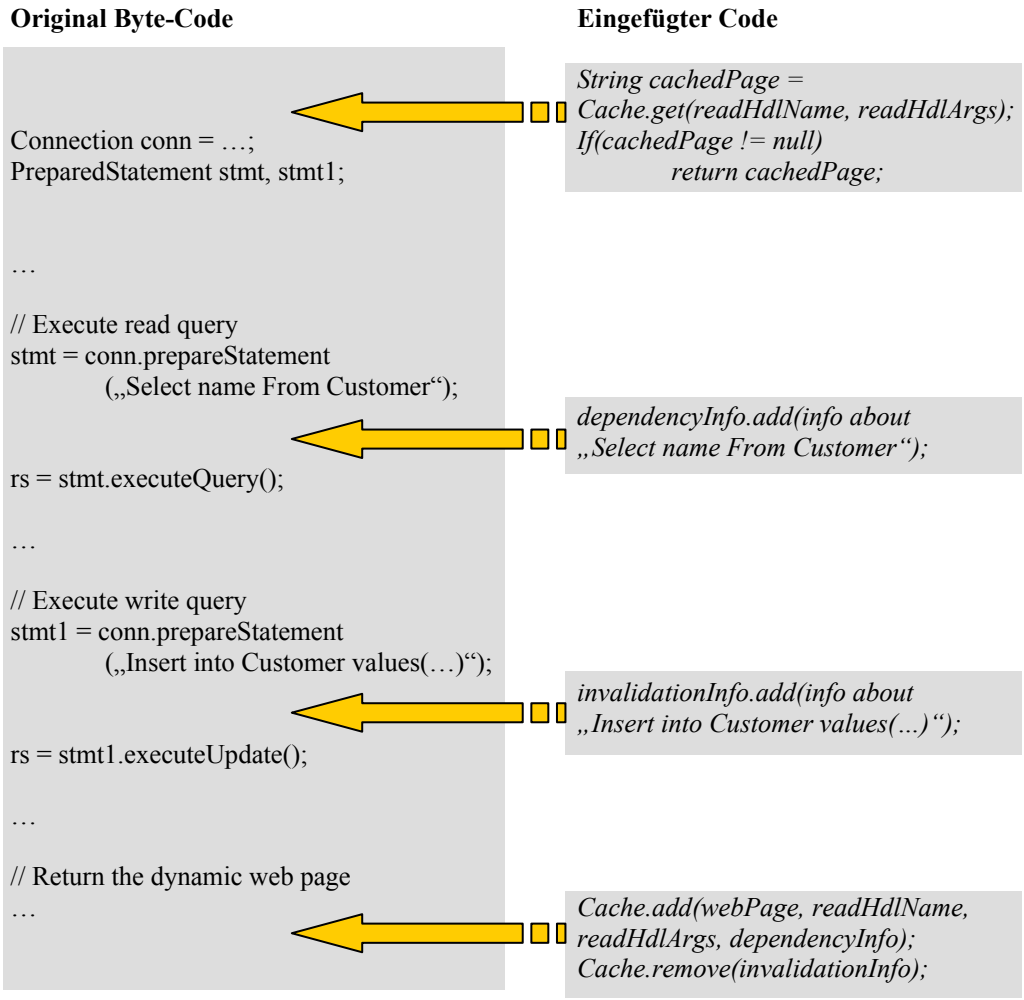


Abbildung A.1 – Code-Transformation lesender und schreibender Zugriffe

TimesTen: Der erste Constraint-basierte Datenbank-Cache

Der Prototyp einer Mittelschicht-Caching-Architektur des TimesTen-Teams ist die erste Realisierung eines Constraint-basierten Datenbank-Cache. Auf der Basis der Entwicklung eines eigenen Datenbank-Management-Systems, welches komplett im Hauptspeicher arbeitet [Time99], wurde der Datenbank-Cache-Prototyp entwickelt [Time02].

Hier finden sich auch die Wurzeln für Begriffe wie Cache Groups und Cache-Keys (damals noch als Cache-Instance-Key bezeichnet), die im Constraint-basierten Caching eine zentrale Rolle einnehmen. Der Inhalt des Cache wird vom Datenbank-Administrator durch Cache Groups spezifiziert, indem er eine Auswahl der in der Backend-Datenbank vorhandenen Tabellen trifft.

Im Gegensatz zu heutigen Constraint-basierten Datenbank-Caches ist der Prototyp von TimesTen für die Anwendung nicht transparent, denn die Einlagerung von Daten muss explizit von der Anwendung angefordert werden. Dafür stehen mehrere Möglichkeiten zur Verfügung:

- *Laden der gesamten Cache Group auf einmal*. Wenn der Inhalt der Cache

Group die Größe des Cache nicht übersteigt, können alle Daten auf einmal in den Cache eingelagert werden. Auch das vollständige Entfernen einer Cache Group aus dem Cache ist möglich.

- *Laden von Cache-Instanzen durch IDs*: Hierbei wird eine bestimmte ID einer Tabelle in den Cache geladen. Alle Join-Partner, die über Fremdschlüssel-Beziehungen von dieser Tabelle ausgehen, werden ebenfalls in den Cache geladen. Dieses Konzept wurde später zu den heute bekannten referenziellen Cache-Constraints weiterentwickelt.
- *Laden von Cache-Instanzen durch Where-Klauseln*: Analog zum Laden von Cache-Instanzen wird hier als Ausgangspunkt eine Where-Klausel verwendet, um den Cache zu füllen.

Die Bearbeitung von Änderungsoperationen stellt für den Prototyp von TimesTen kein Problem dar. Die Änderungsoperationen werden explizit von der Anwendung zum Backend propagiert (2-Phasen-Commit zwischen dem Cache und der Backend-Datenbank). Der Prototyp bietet dafür zwei Möglichkeiten an:

- *Propagate*: Wird diese Option eingeschaltet, so werden alle Änderungsoperationen automatisch an die Backend-Datenbank weitergeleitet.
- *Flush*: Wenn die Option *Propagate* nicht eingeschaltet ist, kann durch den Aufruf von Flush die Anwendung explizit eine Änderung an die Backend-Datenbank propagieren.

Für die Aktualisierung des Cache ist ebenfalls die Anwendung zuständig. Zur Aktualisierung kann entweder ein expliziter Aufruf stattfinden (*Refresh*), ein Zeitintervall festgelegt werden, nach dem der Cache automatisch aktualisiert wurde (*Full Autorefresh*), oder die Aktualisierung nur auf die veränderten Datensätze beschränkt werden (*Incremental Autorefresh*).

Obwohl der Prototyp von TimesTen noch einige Mängel aufweist (z. B. keine Transparenz für die Anwendung), so legte er doch den Grundstein für ein neues Konzept im Bereich des Datenbank-Caching.

Anhang B – XML-Datei zum Filmdatenbank-Beispiel

In Abschnitt 5.2 (Seite 75) haben wir kurz den Aufbau der XML-Datei beschrieben, die alle Angaben über die spezifizierten Cache Groups beinhaltet und dem CBCS als Eingabe dient.

Der Vollständigkeit halber stellen wir an dieser Stelle die komplette XML-Beschreibung für das Filmdatenbank-Beispiel dar. Die zugehörige Document-Type-Definition (DTD) ist im Anschluss an den XML-Ausdruck abgebildet.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dbcache SYSTEM "CBCS.dtd">

<dbcache xmlns="http://cbcs-project.de/dbcache">
  <backend>
    <schema name="SCHEMA1">
       <table id="t1" name="Regisseur" primarykey="c1">
          <column id="c1" unique="true" nullable="false">
             <name>id</name>
             <type>integer</type>
          </column>
          <column id="c2" unique="false" nullable="true">
             <name>vname</name>
             <type size="30">varchar</type>
          </column>
          <column id="c3">
             <name>nname</name>
             <type size="30">varchar</type>
          </column>
          <column id="c4">
             <name>alter</name>
             <type>integer</type>
           </column>
           <column id="c5">
              <name>wohnort</name>
              <type>integer</type>
           </column>
        </table>
        <table id="t2" name="Darsteller" primarykey="c6">
           <column id="c6" unique="true" nullable="false">
              <name>id</name>
              <type>integer</type>
           </column>
           <column id="c7">
              <name>vname</name>
              <type size="30">varchar</type>
           </column>
           <column id="c8">
              <name>nname</name>
              <type size="30">varchar</type>
           </column>
           <column id="c9">
              <name>alter</name>
```

```
            <type>integer</type>
         </column>
         <column id="c10" unique="false" nullable="true">
            <name>wohnort</name>
            <type>integer</type>
         </column>
      </table>
      <table id="t3" name="Ort" primarykey="c11 c12">
         <column id="c11" unique="false" nullable="false">
            <name>plz</name>
            <type>integer</type>
         </column>
         <column id="c12" unique="false" nullable="false">
            <name>name</name>
            <type>integer</type>
         </column>
      </table>
      <table id="t4" name="Darst_Film" primarykey="c13 c14">
         <column id="c13" unique="false" nullable="false">
            <name>did</name>
            <type>integer</type>
         </column>
         <column id="c14" unique="false" nullable="false">
            <name>fid</name>
            <type>integer</type>
         </column>
      </table>
      <table id="t5" name="Film" primarykey="c15">
         <column id="c15" unique="true" nullable="false">
            <name>id</name>
            <type>integer</type>
         </column>
         <column id="c16" unique="false" nullable="true">
            <name>titel</name>
            <type size="100">varchar</type>
         </column>
         <column id="c17" unique="false" nullable="true">
            <name>regie</name>
            <type>integer</type>
         </column>
         <column id="c18" unique="false" nullable="true">
            <name>genre</name>
            <type>integer</type>
         </column>
         <column id="c19" unique="false" nullable="true">
            <name>ort</name>
            <type>integer</type>
         </column>
      </table>
      <table id="t6" name="Genre" primarykey="c20">
         <column id="c20" unique="true" nullable="false">
            <name>id</name>
            <type>integer</type>
         </column>
         <column id="c21" unique="false" nullable="true">
            <name>name</name>
            <type size="100">varchar</type>
         </column>
      </table>
   </schema>
</backend>
```

```
    <cachegroup>
      <schema name="SCHEMA1">
        <table id="ct1" ref="t1" name="CA_Regisseur"/>
        <table id="ct2" ref="t2" name="CA_Darsteller"/>
        <table id="ct3" ref="t3" name="CA_Ort"/>
        <table id="ct4" ref="t4" name="CA_Darst_Film"/>
        <table id="ct5" ref="t5" name="CA_Film"/>
        <table id="ct6" ref="t6" name="CA_Genre"/>
      </schema>
      <fillcolumn ref="c1" percentage="1"
              ctrl-table-name="Ctrl_Regisseur"/>
      <fillcolumn ref="c3" percentage="0.05"
              ctrl-table-name="Ctrl_Regisseur2"/>
      <fillcolumn ref="c6" percentage="1"
              ctrl-table-name="Ctrl_Darsteller"/>
      <fillcolumn ref="c8" percentage="0.06"
              ctrl-table-name="Ctrl_Darsteller2"/>
      <rcc from="c1" to="c17" user-defined="true"/>
      <rcc from="c3" to="c6" user-defined="true"/>
      <rcc from="c5" to="c11" user-defined="true"/>
      <rcc from="c6" to="c13" user-defined="true"/>
      <rcc from="c8" to="c3" user-defined="true"/>
      <rcc from="c10" to="c11" user-defined="true"/>
      <rcc from="c14" to="c15" user-defined="true"/>
      <rcc from="c18" to="c20" user-defined="true"/>
      <rcc from="c19" to="c11" user-defined="true"/>
    </cachegroup>
</dbcache>
```

Der Aufbau der DTD ist wie folgt:

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT dbcache    (backend|cachegroup)*>
<!ELEMENT backend    (schema*)>
<!ELEMENT schema     (table*)>
<!ELEMENT table      (column*)>
<!ELEMENT column     (name, type)>
<!ELEMENT cachegroup (schema|fillcolumn|rcc)*>
<!ELEMENT fillcolumn EMPTY>
<!ELEMENT rcc        EMPTY>
<!ELEMENT name       (#PCDATA)>
<!ELEMENT type       (#PCDATA)>
<!ELEMENT size       (#PCDATA)>

<!ATTLIST dbcache
    xmlns:cbcs   CDATA #FIXED "http://cbcs-project.de/dbcache"
>
<!ATTLIST schema
    name         CDATA #REQUIRED
>
<!ATTLIST table
    id           ID    #REQUIRED
    name         CDATA #REQUIRED
    ref          IDREF #IMPLIED
    primarykey   IDREFS #IMPLIED
>
```

```
<!ATTLIST column
    id          ID    #REQUIRED
    unique      (true|false) "false"
    nullable    (true|false) "true"
>
<!ATTLIST fillcolumn
    ref             IDREF #REQUIRED
    percentage      CDATA #IMPLIED
    ctrl-table-name CDATA #IMPLIED
>
<!ATTLIST rcc
    from         IDREF #REQUIRED
    to           IDREF #REQUIRED
    user-defined (true|false) "true"
    redundant    (true|false) "false"
>

<!ATTLIST type
    size         CDATA #IMPLIED
>
```

Anhang C – EBNF-Notation der verwendeten SQL-Syntax

Die Query Worker des CBCS validieren alle eintreffende Leseanfragen des Benutzers gegen die spezifizierte SQL-Syntax (vgl. Abbildung 4.4; Seite 46). Nur wenn die Anfrage dieser Syntax entspricht, kann sie vom Query Worker weiter bearbeitet werden und gegebenenfalls Teile der Anfrage auf der Cache-Datenbank ausgewertet werden.

Im Folgenden findet sich eine Zusammenfassung dieser Syntax in EBNF-Notation. Diese EBNF-Notation dient als Eingabe für den Parser-Generator ANTLR [Parr05] und wurde speziell dafür angepasst. Die Grammatik ist unterteilt in einen Parser- und einen Lexer-Teil. Der Parser-Teil enthält die einzelnen Grammatikregeln, die sowohl Terminalsymbole als auch Nicht-Terminalsymbole enthalten können. Der Lexer-Teil enthält Regeln, die einem Terminalsymbol ein oder mehrere Zeichen zuordnet. Die Terminalsymbole übernehmen im Lexer-Teil die Rolle von Nicht-Terminalsymbolen. Der Parser verarbeitet die Eingabe und erzeugt als Ergebnis wird ein AST (Abstract Syntax Tree), wenn die Eingabe der spezifizierten Syntax entspricht. Entspricht die Eingabe nicht der Syntax, so wird eine Fehlermeldung zurückgegeben.

An einigen Stellen finden sich in der Grammatik zusätzlich zu den Regeln auch Ausdrücke in Java-Notation (z. B. für die Regel *selectItem*). Diese zusätzlichen Ausdrücke werden vom Parser-Generator bei der Generierung des Parsers in den Code übernommen. Somit ist es möglich, Funktionalitäten direkt in der Grammatik festzulegen, die automatisch in den generierten Parser übernommen werden, was eine spätere Bearbeitung des Parsers von Hand überflüssig macht. In unserer Grammatik verwenden wir die Java-Notationen ausschließlich zum Umbenennen von Knotennamen oder zum Erstellen von zusätzlichen Knoten im AST.

Wir wollen an dieser Stelle das Thema nicht weiter vertiefen. Für interessierte Leser verweisen wir auf die Dokumentation des ANTLR, die sich unter anderem bei [Parr05] findet. Die Dokumentation enthält alle Informationen über Aufbau und notwendige Struktur für eine Grammatik-Datei, die als Eingabe für den Parser-Generator dienen soll.

```
// PARSER
header {
    package cacheManager.queryWorker.sqlParser;
}

class SqlParser extends Parser;
options {
    k = 2;
    buildAST = true;
    defaultErrorHandler = false;
}
```

```
tokens{
   TABLENAME;
   COLUMNNAME;
   ALIASNAME;
   VALUE;
   SELECTITEM;
   FROMITEM;
   PREDICATE;
   LEFTSIDE;
   RIGHTSIDE;
   GROUPBYITEM;
}

// Start rule
statement
    : selectStatement (NEWLINE!)?
    ;
selectStatement
    :
    selectClause fromClause whereClause
      (groupByClause (havingClause)?)? (orderByClause)?
    ;

// -------------- SELECT-Rules -------------------------
selectClause
    : SELECT^ (ALL | DISTINCT)? selectList
    ;

selectList
    : STAR
    | selectItem (COMMA! selectItem)*
    ;

selectItem
// Erzeugen eines neuen Knotens mit Namen "SelectItem"
{AST tmp_AST = null;
 tmp_AST = astFactory.create(LT(1));
 tmp_AST.setText("SelectItem");
 tmp_AST.setType(SELECTITEM);
 astFactory.makeASTRoot(currentAST, tmp_AST);}

    : columnName (aliasName | (AS aliasName))?
    | tableName DOT columnName (aliasName | (AS aliasName))?
    | tableName DOT_STAR
    ;

// -------------- FROM-Rules ---------------------------
fromClause
    : FROM^ fromList
    ;

fromList
    : fromItem (COMMA! fromItem)*
    ;
fromItem
// Erzeugen eines neuen Knotens mit Namen "FromItem"
{AST tmp_AST = null;
 tmp_AST = astFactory.create(LT(1));
 tmp_AST.setText("FromItem");
 tmp_AST.setType(FROMITEM);
 astFactory.makeASTRoot(currentAST, tmp_AST);}

    : tableName (correlationClause)?
    | (ONLY | OUTER) LBRACKET tableName RBRACKET
      (correlationClause)?
    ;
```

```
correlationClause
    : AS aliasName
      (LBRACKET columnName (COMMA columnName)* RBRACKET)?
    | aliasName(LBRACKET columnName (COMMA columnName)* RBRACKET)?
    ;


// -------------- WHERE-Rules ----------------------------
whereClause
    : WHERE^ searchCondition
    ;

searchCondition
    : predicate (AND! predicate)*
    ;

predicate
    : leftSide
      (ASSIGNEQUAL^ | NOTEQUAL1^ | NOTEQUAL2^ |
      LESSTHANOREQUALTO1^ | LESSTHANOREQUALTO2^
      | LESSTHAN^ | GREATERTHANOREQUALTO1^ |
      GREATERTHANOREQUALTO2^ | GREATERTHAN^)
      rightSide
    ;

leftSide
// Erzeugen eines neuen Knotens mit Namen "LeftSide"
{AST tmp_AST = null;
 tmp_AST = astFactory.create(LT(1));
 tmp_AST.setText("LeftSide");
 tmp_AST.setType(LEFTSIDE);
 astFactory.makeASTRoot(currentAST, tmp_AST);}

    : (tableName DOT)? columnName | value
    ;

rightSide
// Erzeugen eines neuen Knotens mit Namen "RightSide"
{AST tmp_AST = null;
 tmp_AST = astFactory.create(LT(1));
 tmp_AST.setText("RightSide");
 tmp_AST.setType(RIGHTSIDE);
 astFactory.makeASTRoot(currentAST, tmp_AST);}

    : (tableName DOT)? columnName | value
    ;


// -------------- GROUP BY-Rules -------------------------
groupByClause
      : GROUP^ BY (groupingExpr (COMMA groupingExpr)* |
      groupingSets)
      ;
groupingExpr
      : tableName DOT columnName
      | columnName
      ;

groupingSets
      : GROUPING SETS
         LBRACKET
            (groupingExpr (COMMA groupingExpr)*
            | LBRACKET groupingExpr (COMMA groupingExpr)*
           RBRACKET)
         RBRACKET
      ;
```

```
// -------------- HAVING-Rules ------------------------
havingClause
    : HAVING^ searchCondition
    ;
// -------------- ORDER BY-Rules ------------------------
orderByClause
    : ORDER^ BY sortKey (ASC | DESC) (COMMA sortKey (ASC | DESC))*
    ;
sortKey
    : tableName DOT columnName
    | columnName
    | NUMBER
    ;
// -------------- GENERAL-Rules ------------------------
tableName
    : ALPHA_NUM_STRING
      // Umbenennen des Knotens in "TableName"
      {((AST)currentAST.root).setType(TABLENAME);}
    ;
columnName
    : ALPHA_NUM_STRING
      // Umbenennen des Knotens in "ColumnName"
      {((AST)currentAST.root).setType(COLUMNNAME);}
    ;
aliasName
    : ALPHA_NUM_STRING
      // Umbenennen des Knotens in "AliasName"
      {((AST)currentAST.root).setType(ALIASNAME);}
    ;
value
    : NUMBER {((AST)currentAST.root).setType(VALUE);}
      | APOSTROPHE (ALPHA_NUM_STRING)+ APOSTROPHE
      // Umbenennen des Knotens in "Value"
      {((AST)currentAST.root).setType(VALUE);}
    ;
// LEXER
class SqlLexer extends Lexer;
options {
    defaultErrorHandler = false;
    testLiterals = true;
    k = 2;
    caseSensitive = false;
    caseSensitiveLiterals = false;
    charVocabulary='\u0000'..'\uFFFE';
}
tokens {
    AND = "and";
    AS = "as";
    ASC = "asc";
    BY = "by";
    DESC = "desc";
    DISTINCT = "distinct";
    FROM = "from";
    GROUP = "group";
    GROUPING = "grouping";
    HAVING = "having";
    ONLY = "only";
    ORDER = "order";
    OUTER = "outer";
    SELECT = "select";
    SETS = "sets";
    WHERE = "where";
}
```

```
// Operators
STAR : '*';
DOT : '.';
DOT_STAR : ".*";
COMMA : ',';
LBRACKET : '(';
RBRACKET : ')';
APOSTROPHE : '\'';
ASSIGNEQUAL : '=' ;
NOTEQUAL1 : "<>" ;
NOTEQUAL2 : "!=" ;
LESSTHANOREQUALTO1 : "<=" ;
LESSTHANOREQUALTO2 : "!>" ;
LESSTHAN : "<" ;
GREATERTHANOREQUALTO1 : ">=" ;
GREATERTHANOREQUALTO2 : "!<" ;
GREATERTHAN : ">" ;

WHITESPACE
    : (' ' | '\t' | '\n' | '\r') { _ttype = Token.SKIP; }
    ;

ALPHA_NUM_STRING
    : ('a'..'z' | '_' | '#' | '\u0080'..'\ufffe')
      (LETTER | DIGIT)*
    ;

NUMBER : (DIGIT)+ ;

protected
LETTER : 'a'..'z' | '_' | '#' | '@' | '\u0080'..'\ufffe';

protected
DIGIT : '0'..'9';

NEWLINE : '\r' '\n';
```

Anhang D – Einrichten des CBCS

Nachdem wir in den Kapiteln 4 und 5 die Funktionsweise des CBCS ausführlich beschrieben haben, wollen wir an dieser Stelle noch eine kleine Anleitung zur Installation des CBCS geben. Die folgende Beschreibung der durchzuführenden Maßnahmen bezieht sich dabei auf die von uns verwendete Laufzeitumgebung (vgl. Abschnitt 5.1; Seite 73).

Bevor das CBCS gestartet werden kann, sollte sichergestellt sein, dass beide Datenbanken korrekt installiert und eingerichtet wurden. Weiterhin muss in der Cache-Datenbank die Funktionalität zur Föderation von Datenbanksystemen aktiviert werden, sowie eine entsprechende Verbindung mit der Backend-Datenbank hergestellt werden. Die dazu nötigen Befehle sind hier nochmals kurz zusammengefasst:

1. update dbm cfg using federated yes
2. catalog tcpip node <*nodename*> remote <*ip-address backend*> server <*port*>
3. catalog database <*backend-db name*> as <*aliasname*> at node <*nodename*> authentication server
4. create wrapper *drda*
5. create server <*servername*> type db2/udb version 8.1. wrapper drda authorisation
   "<*username*>" password "<*password*>" options (dbname ,<*aliasname*>')
6. create user mapping for <*local-user-name*> server <*servername*>
   options (remote_authid '<*username*>', remote_password '<*password*>)

Weitere Informationen zur Einrichtung und zum Zugriff auf föderierte Datenbanksysteme über die IBM DB2 finden sich Abschnitt 5.2.1 oder in [IBM05b]. Zusätzlich zu den Datenbanken benötigen wir ein installiertes JRE (Java Runtime Enviroment) in der Version 5.0.

Das Einrichten des CBCS ist relativ einfach. Alle Jar-Dateien des CBCS sowie die Jar-Dateien mit den IBM Datenbanktreibern müssen in den Klassenpfad (Classpath) eingetragen werden. Der Aufruf *java –jar cbcs.jar* startet das CBCS. Zusätzlich können einige Startparameter an das CBCS übergeben werden, die wir im Folgenden kurz vorstellen wollen. Wir verwenden an dieser Stelle die Kurzformen der Parameter; alternativ kann auch die jeweilige Langform eingesetzt werden.

- -h: Gibt eine kurze Hilfe auf dem Bildschirm aus, in der alle verfügbaren Eingabeparameter angezeigt werden. Abbildung D.1 zeigt die Ausgabe dieses Eingabeparameters.
- -i: Ist dieser Eingabeparameter gesetzt, so führt das CBCS die erweiterte Initialisierung durch (vgl. Abschnitt 4.3.2)
- -c: Dieser Parameter gibt den Pfad zur XML-Datei an, die die Angaben zur

Konfiguration des CBCS enthält. Der Aufbau dieser Datei ist in Abbildung D.2 dargestellt. Erläuterungen zur Konfigurationsdatei werden wir im Anschluss an die Beschreibung der Eingabeparameter vornehmen.

- -x: Dieser Parameter gibt den Pfad zur XML-Datei an, die die Informationen über die Cache Group enthält. Den Aufbau dieser Datei findet sich in Anhang B bzw. in Abschnitt 5.2.1 (Seite 75).

```
usage: CBCS [-x file] [-i] [-h] [-c file]
-i,--init                 Erweiterte Initialisierung durchführen
                          (Cache-Datenbank einrichten)
-x,--cache-config <file>  Datei zur Cache-Konfiguration verwenden
-c,--config <file>        Datei zur CBCS-Konfiguration verwenden
-h,--help                 Diesen Hilfebildschirm anzeigen
```

Abbildung D.1 – Ausgabe der verfügbaren Eingabeparameter

In Abbildung D.2 ist der Aufbau der CBCS-Konfigurationsdatei dargestellt. Insgesamt besitzt das CBCS zwei Konfigurationsdateien: eine mit den Standardwerten und eine mit Benutzerwerten. Alle Angaben in Abbildung D.2, die in kursiver Schrift dargestellt sind, befinden sich in der speziellen Konfigurationsdatei des Benutzers. Diese Datei enthält üblicherweise Angaben über die URL der Backend- und Cache-Datenbank sowie die dafür benötigten Benutzernamen und Passwörter.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
   <entry key="Backend:Driver">com.ibm.db2.jcc.DB2Driver</entry>
   <entry key="Backend:URL">jdbc:db2:backend</entry>
   <entry key="Cache:Driver">com.ibm.db2.jcc.DB2Driver</entry>
   <entry key="Cache:URL">jdbc:db2:cache</entry>
   <entry key="NumOfQueryWorkers">10</entry>
   <entry key="RMI:Host">localhost</entry>
   <entry key="RMI:Port">1099</entry>
   <entry key="RMI:Start">true</entry>
   <entry key="GCollectorSleepTime">60000</entry>
   <entry key="CriticalSpaceLevel">0.9</entry>
   <entry key="TablespaceName">USERSPACE1</entry>
   <entry key="CacheDescription">CacheGroupConfig.xml</entry>
   <entry key="Backend:URL">URL-der-Backend-Datenbank</entry>
   <entry key="Backend:Username">username</entry>
   <entry key="Backend:Password">password</entry>
   <entry key="Cache:URL">URL-der-Cache-Datenbank</entry>
   <entry key="Cache:Username">username</entry>
   <entry key="Cache:Password">password</entry>
   <entry key="Backend:URL">jdbc:db2:backend</entry>
   <entry key="Cache:URL">jdbc:db2:cache</entry>
   <entry key="FedDBServer:Name">beserver</entry>
   <entry key="FedDBServer:SchemaNameBackend">BACKEND</entry>
   <entry key="FedDBServer:SchemaNameCache">CACHE</entry>
</properties>
```

Abbildung D.2 – Inhalt der CBCS-Konfigurationsdatei

In unserem Fall sind weiterhin spezielle Informationen in dieser Datei eingetragen, die für die Benutzung der DB2 Funktionalität zur Föderation von Datenbanksystemen benötigt werden. Es ist jedoch auch möglich, die Standardwerte aus der anderen Konfigurationsdatei zu überschreiben, dazu muss lediglich der entsprechende Eintrag in die benutzerspezifische Konfigurationsdatei übernommen und mit neuen Werten belegt werden.

Der obere Teil der in Abbildung D.2 dargestellten Konfigurationsdatei enthält die Angaben, die sich in der Datei mit den Standardwerten befinden. Dort befinden sich unter anderem Informationen über den Ort des Datenbanktreibers, die Anzahl der Query Worker die während der Initialisierung angelegt werden sollen. Die Datei enthält ebenfalls Angaben über die URL der RMI-Registry und ob eine eigene RMI-Registry gestartet werden soll. Dieser Bereich enthält ebenfalls Informationen darüber, bei welchem Füllgrad des Cache der Garbage Collector aktiv werden soll und wie lange nach einem Löschvorgang gewartet werden soll, bevor die nächste Überprüfung durchgeführt wird. Die Information zum Schlüsselwort *CacheDescription* beziehen sich auf den Pfad und Dateinamen der XML-Datei, die die Angaben über die Cache Group enthält.

Zum Auffinden von Fehlern und zur Darstellung der durchgeführten Aktionen des CBCS verwenden wir das frei verfügbare Apache Log4J [Log4J]. Alle Ausgaben, die während der Laufzeit des CBCS erzeugt werden, werden über die Schnittstellen des Log4J ausgegeben. Auch für die Konfiguration des Log4J verwenden wir eine XML-Datei. Die Datei trägt den Namen *log4j.xml* und befindet sich im Stammverzeichnis des CBCS. Wir werden an dieser Stelle nicht auf den Aufbau dieser XML-Datei eingehen und verweisen auf die Dokumentation in [Log4J].

Nach dem erfolgreichen Starten des CBCS erscheint die Kommandozeile des Systems. Es gibt verschiedene Befehle, die über die Kommandozeile ausgeführt werden können. Die Eingabe des Befehls *help* zeigt alle verfügbaren Befehle mit einer kurzen Erklärung auf dem Bildschirm. Die Ausgabe dieses Befehls ist in Abbildung D.3 dargestellt. Wir werden an dieser Stelle auf eine ausführliche Erläuterung der einzelnen Befehle verzichten, da diese selbsterklärend sind.

Für die Eingabe von SQL-Select-Anweisungen wird kein zusätzlicher Befehl benötigt. Es erfolgt also lediglich die Eingabe der Select-Anweisung und die Bestätigung durch drücken der Enter-Taste. Das Ergebnis der Anfrage wird anschließend vom CBCS aufbereitet und in tabellarischer Form in der Kommandozeile dargestellt.

Die Befehle *exit* bzw. *quit* führen zum Beenden des CBCS. Dieser Vorgang kann einige Sekunden in Anspruch nehmen, da zuerst alle Komponenten aufgefordert werden, ihre Exit-Routine durchzuführen. Ein Fill Daemon bzw. Garbage Collector, der gerade einen Füll- bzw. Löschvorgang durchführt, wird diesen Vorgang zuerst ord-

nungsgemäß abschließen, bevor er seine Exit-Routine durchführt. Dies dient dazu, dass der Cache nicht einen ungültigen Zustand erhält. Weiterhin müssen die Einträge des CBCS aus der RMI-Registrierung gelöscht und die generierten Prepared Statements geschlossen werden, was in der Regel auch wieder einige Sekunden in Anspruch nimmt. Eine kurze Ausgabe auf dem Bildschirm bestätigt das ordnungsgemäße Beenden des CBCS.

```
CBCS v1.0 Kommandozeilenbefehle:
--------------------------------
Ausführen einer Select-Anweisung: "select ..."
Neustarten des FillDaemons: "filldaemon restart"
Schließen des FillDaemons: "filldaemon close"
Einlagern eines Werts in den Cache:
      "filldaemon insert <cachetable> <fillcolumn> <value>"
Neustarten des HitCounter: "hitcounter restart"
Schließen des HitCounter: "hitcounter close"
Neustarten des GCollector: "gcollector restart"
Schließen des GCollector: "gcollector close"
Löschen einer Cache-Instanz: "gcollector delete"
Beenden des CBCS: "exit" oder "quit"
```

Abbildung D.3 – Aufruf des Help-Befehls in der Kommandozeile des CBCS

Literaturverzeichnis

ABKM03 Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Berthold Reinwald:
Cache Tables: Paving the Way for an Adaptive Database Cache
VLDB 2003: 718 – 729

Andr98 M. Andrews:
Negative Caching of DNS Queries (DNS NCACHE)
Request for Comments (RFC) 1998
ftp://ftp.rfc-editor.org/in-notes/rfc2308.txt
Gefunden April 2005

APTP03a Khalil Amiri, Sanghyun Park, Renu Tewari, Sriram Padmanabhan:
Scalable template-based query containment checking for web semantic caches
ICDE 2003: 493 – 504

APTP03b Khalil Amiri, Sanghyun Park, Renu Tewari, Sriram Padmanabhan:
DBProxy: A dynamic data cache for Web applications
ICDE 2003: 821 – 831

BAKM03 Christof Bornhövd, Mehmet Altinel, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Berthold Reinwald:
DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures
SIGMOD Conference 2003: 662

BAMP04 Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, Berthold Reinwald:
Adaptive Database Caching with DBCache
Bulletin of the TC on Data Engineering, IEEE Computer Society, 2004

BoMZ03 Sara Bouchenak, Sumit Mittal, Willy Zwaenepoel:
Using Code Transformation for Consistent and Transparent Caching of Dynamic Web Content
EPFL Technical Report ID: 200383, 2003

BüHä04 Andreas Bühmann, Theo Härder:
Database Caching: Analysis of Constraint-based Approaches Exemplified by Cache Groups
16. GI-Workshop Grundlagen von Datenbanken, 2004: 38 – 42
(Inhaltlich ähnlich zu [HäBü04b])

Bühm05a Andreas Bühmann:
Einen Schritt zurück zum negativen Datenbank-Caching
BTW 2005: 107 – 124

Bühm05b Andreas Bühmann:
Ein Schritt zurück ist kein Rückschritt – Mit flexibler Sondierung zum negativen Datenbank-Caching
Informatik – Forschung und Entwicklung, 2005

CLKS04 Seunglak Choi, Jinwon Lee, Su Myeon Kim, Junehwa Song, Yoon-Joon Lee:
Accelerating Database Processing at e-Commerce Sites
EC-Web 2004: 41 – 50

DFJS96 Shaul Dar, Michael J. Franklin, Björn T. Jónsson, Divesh Srivastava, Michael Tan:
Semantic Data Caching and Replacement
VLDB 1996: 330 – 341

Eclipse Eclipse 3.1
http://www.eclipse.org
Gefunden Mai 2005

HäBü04a Theo Härder, Andreas Bühmann:
Value Complete, Domain Complete, Predicate Complete – Magic Words Driving the Design of Cache Groups
http://wwwdvs.informatik.uni-kl.de/pubs/papers/HB04.Magic.html
Gefunden April 2005

HäBü04b Theo Härder, Andreas Bühmann:
Datenbank-Caching – Eine systematische Analyse möglicher Verfahren
Informatik – Forschung und Entwicklung 19(1), 2004: 2 – 16

IBM05a IBM DB2 Universal Database
http://www-306.ibm.com/software/data/db2
Gefunden Mai 2005

IBM05b IBM DB2 Universal Database
SQL Reference Volume 1
http://users.sdsc.edu/~jrowley/db2/SQL%20Reference%20V1.pdf
Gefunden August 2005

J2SE05 Sun Microsystems J2SE Development Kit 5.0
http://www.sun.com
Gefunden Mai 2005

LaGZ04 Per-Åke Larson, Johnathan Goldstein, Jingren Zhou:
MTCache: Transparent Mid-Tier Database Caching in SQL Server
ICDE 2004: 177 – 189

LGGZ04 Per-Åke Larson, Jonathan Goldstein, Hongfei Guo, Jingren Zhou:
MTCache: Mid-Tier Database Caching for SQL Server
Bulletin of the IEEE Computer Society Technical Commitee on Data Engineering, 2004

LKMP02 Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, Jeffrey F. Naughton:
Middle-Tier Database Caching for E-Business
SIGMOD Conference 2002: 600 – 611

Log4J Apache Log4J
http://www.logging.apache.org/log4j/docs/
Gefunden Juni 2005

LuNu01 Quion Luo, Jeffrey F. Naughton:
Form-Based Proxy Caching for Database-Backed Web Sites
VLDB 2001: 191 – 200

LZWM04 Fujian Liu, Yanping Zhao, Wenguang Wang, Dwight Makaroff:
Database Server Workload Characterization in an E-Commerce Enviroment
MASCOTS 2004: 475 – 483

Moha01 C. Mohan:
Caching Technologies for Web Applications
VLDB 2001: 726
http://wwwconf.ecs.soton.ac.uk/archive/00000478/
Gefunden April 2005
(Inhaltlich Ähnlich zu [Moha04])

Moha04 C. Mohan:
DBCache: A Project on Database Caching Support for Web Applications
Stanford Database Seminar, 2004
http://www-db.stanford.edu/dbseminar/Archive/WinterY04/
Gefunden April 2005

Parr05 Terence Parr
ANTLR – ANother Tool for Language Recognition
http://www.antlr.org
http://www.antlr.org/doc/getting-started.html
Gefunden Juni 2005

PlAl04 Christian Plattner, Gustavo Alonso:
Ganymed: Scalable and Flexible Replication,
Middleware 2004: 155 – 174

Time02 The TimesTen Team:
Mid-Tier Caching: The TimesTen Approach
SIGMOD Conference 2002: 588 – 593

Time99 The TimesTen Team:
In-Memory Data Management for Consumer Transactions – The TimesTen Approach
SIGMOD Conference 1999: 528 – 529

TPCW05 TPC-W Benchmark
http://www.tpc.org/tpcw/default.asp
Gefunden April 2005

ZhTZ02 Zhen Liu, Fu-Chiang Tsui, Xiaoming Zeng:
Cache Table Design for Disease Surveillance System
AMIA 2002: 1086
http://rods.health.pitt.edu/Technical%20Reports/ANIMA02-Liu.pdf
Gefunden April 2005